Chess AI

William Brophy Tyree

December 14, 2016

1 Introduction

For our fourth lab in Artificial Intelligence, we were assigned to write a program for playing chess. We were provided the basic structure of the Chess game, as well as a library (Chesspresso) written by Bernhard Seybold. For me, the most challenging part of this lab was parsing the provided code and libraries, attempting to figure out what the various parts' functions were. This challenge was rooted in the sparse nature of the documentation for Chesspresso. Once I figured out how the system was working, I got down to the real work. // The Chess system is designed so that the user can play by inputing commands into a text field. The pieces then move according to these commands. Our task was to design artificial intelligence bots to play against the human and against eachother. In order to do that, we implemented the MiniMax algorithm as well as Alpha Beta Pruning. These two algorithms search through the possible moves up until a given depth and return the best move given the current position. These algorithms work under the assumption that their opponent is also trying to maximize their score and win the game.

2 MiniMax and Cutoff Test

The first step in our implementation was to design a depth-limited minimax search. In order to understand minimax, you must first understand the game tree associated with the search algorithm. The tree is composed of alternating levels. The first level is the maximizing player: the player that the miniMax is searching for. The edges leaving the top node are all of the possible first moves. The next level of nodes is a new player, the minimizer. The minimizer is the opponent of the maximizer in the game. Every node (position) has a value representing the evaluation of the position from the eyes of the maximizer. The maximizer is looking for nodes that will return the greatest evaluation number, while the minimizer tries to get to the nodes that minimze that value. The two alternate taking turns. The value of the position is calculated using the following functions...

```java
public int calculateScore(Position position) {
    int player = position.getToPlay();

    if (position.isStaleMate()) {
      return 0;
    } else if (position.isMate()) {
      if (player == maxPlayerNumber) {
        //System.out.println("CHECKMATE!");
        return Integer.MIN_VALUE;
      } else {
        //System.out.println("CHECKMATE!");
        return Integer.MAX_VALUE;
      }
    } else {
      return materialValue(position);
    }
  }
```

```

private int materialValue(Position position) {
    int player = position.getToPlay();
    int value = 0;
    if (player == maxPlayerNumber) {
      value = position.getMaterial() + (int)position.getDomination();
    } else {
      value = -(position.getMaterial()) - (int)position.getDomination();
    }
    return value;
  }
```

The first function is the one called from the main miniMax functions. It checks to see if a checkmate is present in the position. If it is, it returns the max integer value if the maximizer has won and the minimum integer if the minimizer has won. Otherwise, it calls materialValue(), which evaluates the position. This function takes into account two different things: the number of pieces that the player to move has as well as how dominant of a position that player has. The getMaterial function calculates the score by assigning values to each piece (1 to pawn, 5 to rook etc.). By counting the number of pieces and their values, the function gets a good approximation of who is winning.

Now that we have our background, let us look at minimax. Below is the implementation of the minimax search...

```
   public short miniMax(Position position) throws IllegalMoveException {
    short[] moves = position.getAllMoves();
    short bestMove = moves[0];
    int highestUtility = Integer.MIN_VALUE;

    for (int i = 0; i < moves.length; i++) {
      try {
        position.doMove(moves[i]);
        numberStatesVisited++;

        if (cutoffTest(1, position)) {
          if (calculateScore(position) == Integer.MAX_VALUE) {
            bestMove = moves[i];
            bestUtility = Integer.MAX_VALUE;
            position.undoMove();
            System.out.println("Minimax checkmate!");
            break;
          }
        }

        int compare = minValue(1, position);
        if (highestUtility < compare) {
          highestUtility = compare;
          bestMove = moves[i];
          bestUtility = highestUtility;
        }
        position.undoMove();
      } catch (IllegalMoveException e) {
        System.out.println("illegal move");
      }
    }
    System.out.println("The maximum depth reached before making the move was " +
    maxDepthReached);
    System.out.println("The number of states visited was " + numberStatesVisited);
    numberStatesVisited = 0;
    maxDepthReached = 0;
    return bestMove;
  }
```

This function takes as a parameter the current position of the board. The Position is a class of the Chesspresso library that keeps track of the state of the game and the location of the pieces. The function keeps track of the best possible move (which is initialized as the first possible move) and the numerical utility of this best move ($initializedasadummyvalueofInteger.MIN_VAlUE$). The function then loops over all possible moves. For each move, it calls position.doMove(move) which actually does the move and modifies the position class. It then checks to see if this new position fits the criteria to cut off the search by calling cutoffTest(). The implementation for cutoffTest() is below...

```
  public boolean cutoffTest(int depth, Position position) {
    if (depth >= maxDepth) {
      return true;
    } else if (position.isTerminal() || position.isMate()) {
      return true;
    } else {
      return false;
    }
  }
```

This function checks to see if the maxDepth has been hit or if the state is terminal. If the state is terminal, it returns true to the caller. The reason that it calls it where it does in my miniMax function is to check if the new position is a checkMate. If it is ("$calculateScore(position) == Integer.MAX_VALUE$"), then the function returns that move as the best move, not bothering to continue searching. If it is not checkmate, the function continues, calling minValue() on the new position. The essential helper functions for miniMax are minValue and maxValue.

minValue is a recursive function that searches through the game tree, looking for the move that will guarentee the lowest possible score for the maximizing player. The implementation for that is below...

```
private int minValue(int depth, Position position) throws IllegalMoveException {
    maxDepthReached = Math.max(maxDepthReached, depth);
    if (cutoffTest(depth, position)) {
      return calculateScore(position);
    }

    int lowestUtility = Integer.MAX_VALUE;
    short [] moves = position.getAllMoves();
    for (int i = 0; i < moves.length; i++) {
      try {
        position.doMove(moves[i]);
        numberStatesVisited++;
        lowestUtility = Math.min(lowestUtility, maxValue(depth + 1, position));
        position.undoMove();
      } catch (IllegalMoveException e) {
        System.out.println("illegal move");
      }
    }

    return lowestUtility;
  }
```

The minValue function first checks to see if the position is terminal, returning the evaluation of that position if it is. If not, it loops through all moves, keeping a variable to store he lowest utility that can be achieved. It does so by doing each move. Then, it finds the minimum between the lowest utility variable and the value returned by maxValue. Below is the implementation of maxValue()...

```
private int maxValue(int depth, Position position) throws IllegalMoveException {
    maxDepthReached = Math.max(maxDepthReached, depth);
    if (cutoffTest(depth, position)) {
      return calculateScore(position);
    }

    int highestUtility = Integer.MIN_VALUE;
    short [] moves = position.getAllMoves();
    for (int i = 0; i < moves.length; i++) {
      try {
        position.doMove(moves[i]);
        numberStatesVisited++;
        highestUtility = Math.max(highestUtility, minValue(depth + 1, position));
        position.undoMove();
      } catch (IllegalMoveException e) {
        System.out.println("illegal move");
      }
    }

    return highestUtility;
  }
```

MaxValue is almost identical to MinValue, except it is looking for the highest possible utility and is calling minValue. As you can see, these functions represent the minimizing player and the maximizing player. They call each other recursively until the basecase is hit (terminal position or max depth). Then, the minimum and maximum values are returned up the game tree until it hits the initial position that was supplied to the minimax function. When there, the minimax function can know the lowest possible score that he can get by moving each piece. The function then chooses the move with the highest guarenteed minimum.

Lastly, I added iterative deepening to the minimax functioning by just looping around it. At each depth, a variable holds the best move. Since I did not implement a timer, the iterative deepening is not actually necessary and so I did not use it in the final testing because it adds computational time. It does yield the same results though. You can easily experiment with it in Minimax by going to the getMove function and changing miniMax to iterativeMiniMax.

Below is the iterativeMiniMax function...

```
public short iterativeMiniMax(Position position) throws IllegalMoveException {
    short bestMove = 0;

    for (int i = 0; i < maxDepthFinal; i++) {
      maxDepth =  i;
      bestMove = miniMax(position);
    }

    System.out.println("The maximum depth reached before making the move was " +
    maxDepthReached);
    System.out.println("The number of states visited was " + numberStatesVisited);
    numberStatesVisited = 0;
    maxDepthReached = 0;
    maxDepth = maxDepthFinal;
    return bestMove;

  }
```

3 Alpha-Beta Pruning

Next, we expanded the capabilities of our search algorithm by implementing Alph-Beta pruning. The goal of the AB Pruning is to eliminate nodes and whole branches of the search tree by maintaining the alpha and the beta values of the search, otherwise known as the low and the high values. Simply put, let's say we are at a maximizing node. We search its first child move and determine what the guarenteed minimum value for this move is. Lets say the minimum score for that move is 5. Then we move on to the next child move. Now we are at the Minimizing node. After searching the minimizer's first move, we see that he can guarentee a score of 2 with that first node. Now, we do not need to search the rest of the minimizers nodes. We know that this move by the maximizer will guarentee at least a 2, since the Minimizer's goal is to return the lowest possible score. If we know that we can guarentee a score no less than 5 with move 1, the maximizer will never choose move 2 which guarentees a minimum score of 2 or less. The same is true for the maximizer, as the minimizer will never choose a move that guarentees a higher score than one it has already found. This process eliminates a huge number of nodes and allows us to dramatically increase our maximum depth.

Below is the implementation of AB pruning...

```java
public short abPrune(Position position, int low, int high, int depth) throws
    IllegalMoveException {
    int highestUtility = Integer.MIN_VALUE;
    short [] moves = position.getAllMoves();
    short bestMove = moves[0];

    for (int i = 0; i < moves.length; i++) {
      position.doMove(moves[i]);
      numberStatesVisited++;
      if (cutoffTest(1, position)) {
        if (calculateScore(position) == Integer.MAX_VALUE) {
          bestMove = moves[i];
          position.undoMove();
          break;
        }
      }
      int compare = minValue(2, low, high, position);
      if (highestUtility < compare) {
        highestUtility = compare;
        bestMove = moves[i];
      }
      position.undoMove();
    }
    System.out.println("The maximum depth reached before making the move was " +
    maxDepthReached);
    System.out.println("The number of states visited was " + numberStatesVisited);
    numberStatesVisited = 0;
    maxDepthReached = 0;
    return bestMove;
  }

  private int maxValue(int depth, int low, int high, Position position) throws
    IllegalMoveException {
    maxDepthReached = Math.max(maxDepthReached, depth);
    if (cutoffTest(depth, position)) {
      return calculateScore(position);
    }

    int utility = Integer.MIN_VALUE;
    short [] moves = position.getAllMoves();
    for (int i = 0; i < moves.length; i++) {
      try {
        position.doMove(moves[i]);
        numberStatesVisited++;
        utility = Math.max(utility, minValue(depth + 1, low, high, position));
        position.undoMove();

        if (utility >= high) {
          return utility;
        }
        low = Math.max(utility, low);
      } catch (IllegalMoveException e) {
        System.out.println("illegal move");
      }
    }
    return utility;
  }


  private int minValue(int depth, int low, int high, Position position) throws
    IllegalMoveException {
    maxDepthReached = Math.max(maxDepthReached, depth);
    if (cutoffTest(depth, position)) {
      return calculateScore(position);
    }

    int utility = Integer.MAX_VALUE;
    short[] moves = position.getAllMoves();
    for (int i = 0; i < moves.length; i++) {
      try {
        position.doMove(moves[i]);
        numberStatesVisited++;
        utility = Math.min(utility, maxValue(depth + 1, low, high, position));
        position.undoMove();

        if (utility <= low) {
          return utility;
        }

        high = Math.min(utility, high);
      } catch (IllegalMoveException e) {
        System.out.println("illegal move");
      }
    }
    return utility;
  }
```

As you can see, the actual abPrune function is very similar to minimax, except for the fact that it keeps track of the lowest and highest guaranteed score for every move. The pruning takes place in minValue and maxValue, where the function checks to see if (utility ¡= low) or (utility ¿= high).

4 Testing

The difficult part of the lab is finding test cases that prove that our code is working. The test cases are below:

Test 1: In this test, we set up an initial position to see if the two search algorithms will win the game (checkmate) if the winning move is within the maximum depth. Using a max depth of 3 for both, I ran the two search algorithms on a board where one move will cause checkmate. Both AB Prune and MiniMax did the same move. AB Prune searched 21 states, while Minimax serached 278 states.

The next test was to see if, given the same position, the two algorithms would make the same first move yielding the same score. I set it up such that there is an obvious capture move that would yield a large score increase. The AB prune and minimax in fact did the same move. The AB prune searhced 678 states, while the MiniMax searched 21042 states.

The last test was to measure the effectiveness of the AB pruning against the minimax. When I played Alpha Beta against MiniMax, I experimented with different depth values to see who would win. I started with a max depth of 3 for both AB and MM I found that there was no algorithm that was particularly better than the other in this case. AB won sometimes and MM won sometimes. When I increased the max depth for minimax past four, it became unreasonably slow. But I needed not have the same depth, as Alpha Beta removed so many nodes from the search that it was exploring a small fraction of those explored by minimax. So I increased the maxDepth for Alpha Beta to 4 and then 5 (6 yielded too much computational time). While it was exploring a very similar number of nodes to minimax, the domination was evident. Alpha Beta

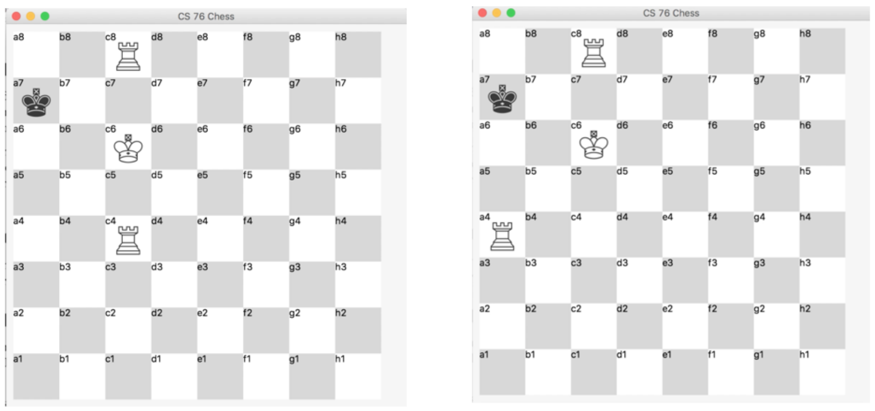

Figure 1: Robot Arm Start

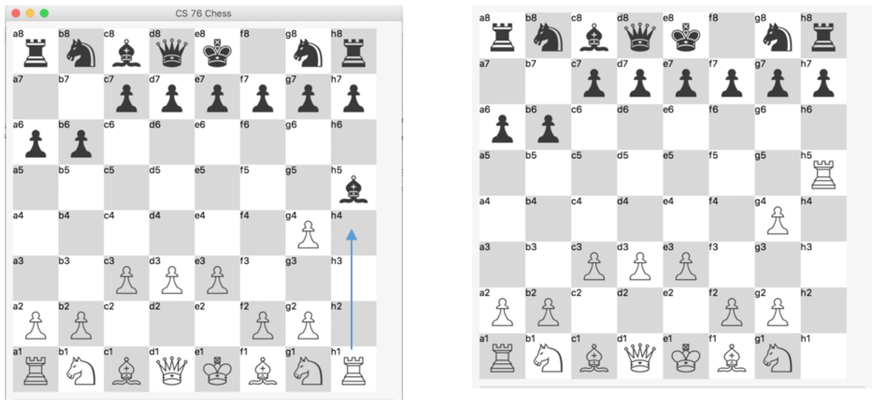

Figure 2: Robot Arm Middle

was computing just as fast as minimax, but dominated it throughout the game and won every time. Starting with a fresh board, I let the game run and saw that AB was making much smarter moves and eventually won quite handly.

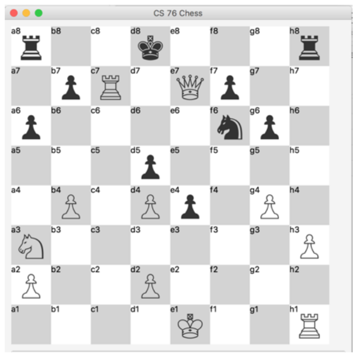

Figure 3: Robot Arm End