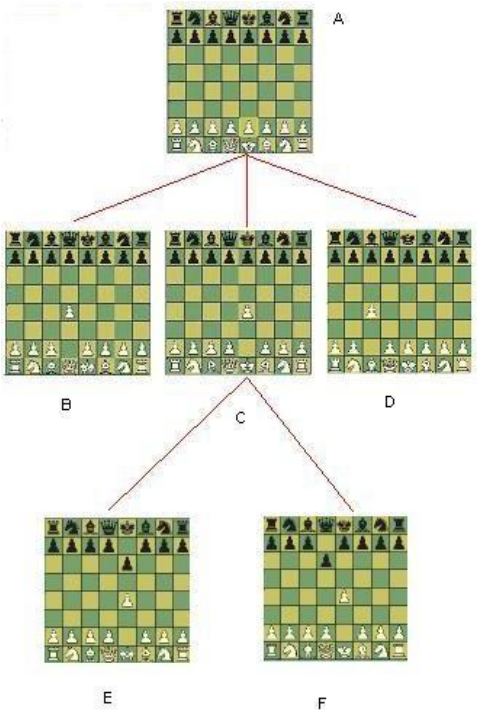
Project Report for Chess B-Tree & Min-max Search Algorithms



Group 10

Written by:

Quinn Smith, David Motta, and Felipe Pareja

CSC-212: Data Structures and Abstractions

December 5, 2022

Introduction

Chess, the thousand-year-old game, is a pass-time rich with history and intrigue. Since the dawn of time man has been inventing new strategies to master this beautiful game. In the 1990s chess underwent a large jump in popularity due to the work of computer scientists at IBM who created the supercomputer Deep Blue that was able to defeat the then champion chess grandmaster Garry Kasparov. Utilizing machine learning and tree search algorithms, these scientists and engineers were able to pull off something not thought possible. Ever since, computer science and search algorithms have been intertwined with the game of chess. Our project is an attempt to emulate the many great chess engines that came before it through the use of B-Tree and Min-max based search algorithms. To be specific, the goal of this project is to create a chess engine that handles user input and then responds on the computer's turn by evaluating the score of the position. The computer does this by considering each position as a 'node' in the tree, then the computer proceeds down the tree and evaluates every possible position from the previously explored node. Thus, in a fully-fledged chess engine upon each and every position the tree branches off evaluating hundreds to thousands of different positions. This branching off is determined by 'depth'. Depth is a value of chess engine analysis which indicates the number of moves made by one side the engine looks ahead. Our program currently has a depth of three meaning the computer calculates 3^n moves into the future and compares the utility of those possible moves against each other to give itself a competitive advantage.

Methods

A B-Tree is a self-balancing tree data structure that allows searches, accesses, insertions, and deletions in logarithmic time complexity O(log n). A B-Tree is a form of binary search tree that unlike its predecessor can have more than two nodes. Different from other self-balancing binary search trees, the B-Tree is also well suited for storage systems that read and write large blocks of data, such as databases and file systems. Among the possible search algorithms available to be selected, B-tree most closely resembles the search algorithm most primarily used throughout the program Min-max. Min-max can be considered as a more advanced tree search algorithm as it is recursive and uses the time complexity O(b^d) where b is the branching factor or in chess terms the amount of legal moves available and d is the count of the depth of the tree. This means that a Min-max algorithm should be able to calculate more potential moves more efficiently than a traditional B-tree algorithm could. However, this also results in a Min-max having to use a lot more storage space than other algorithms as its space complexity is defined by b*d whereas with something like a B-tree it is O(log n).

The Min-Max algorithm is utilized in mainly two player games, such as tic-tac-toe, go, and chess. Although very different, all of the games listed have one thing in common, they are all logic-based games. This means that they have a clearly defined set of rules that involve no randomness or variation. According to these rules, it is also possible to know from any given point in the game what are the next moves available to the given player. This can be described as them being full information games.

In Figure 1 below you can see a representation of a search tree. The squares, also known as nodes, represent points of the decision making in the search. All of its nodes are connected by branches. The search starts at the root node and at each decision point, new nodes for the available search paths are generated, until no more decisions are possible.

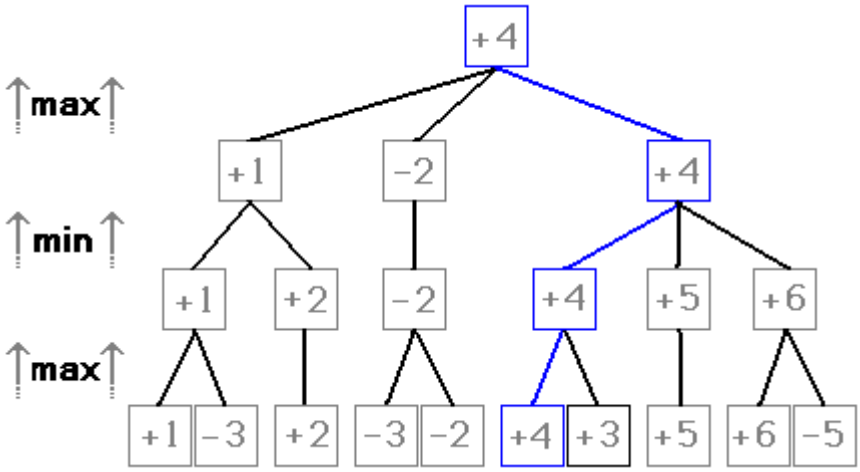


Figure 1

In this algorithm there are two players involved, they are defined as the Max and the Min. A search tree is generated from the current game position up to the end game position (only checkmate or stalemate in a non-timed chess game). Then, the final game position is evaluated from the Max's point of view. As seen in Figure 1 the inner node values of the tree are filled from the bottom up with the evaluated values. The nodes that belong to the Max player receive the maximum value of their children and the nodes for the Min player will receive the minimum value of their children.

The values represent how good a game move is. How good a move is can be demonstrated by the move’s ability to capture an opponent's pieces or increase the likelihood of doing so. In chess each unique group of pieces has a different point value in terms of their importance in gaining a tactical advantage. In order, the relative strength of pieces is as follows: pawn, knight, bishop (knight & bishop are interchangeable), rook, queen, and king. So the Max player will try to select the move with the highest value in the end and the algorithm then analyzes what the Min player or opponent would likely select for a move to increase their odds and minimize Max’s outcome.

Because chess is so complex and has millions of different board states, our code had to be optimized towards game events with probabilities and shortcuts, rather than having full information. Instead of calculating the full path of moves that leads to victory, decisions were calculated by analyzing paths that may lead to victory. This optimization is defined by the depth of the search tree. For simplicity's sake and the limitations of our

computer memory a depth of three was chosen for our project. This means that each node in the tree has three children. The number of nodes per depth can be represented by 3^n:

Depth	Node #
1	3
2	9
3	27
n	3^n

The second optimization used to determine the utility of any given move or piece at a current position are estimation functions. Estimation functions utilize heuristics. Heuristics represent the knowledge that we have about the chess game. For example, in chess, pieces at corners and the rim of the board are able to move to less available squares as they are confined either on the left, right, bottom, top, or any combination of the four. As such the middle of the board squares tend to produce the most amount of options for potential future moves so a chess engine will naturally gravitate towards moving pieces there (there are exceptions in scenarios where pawns on the rim are likely to promote).

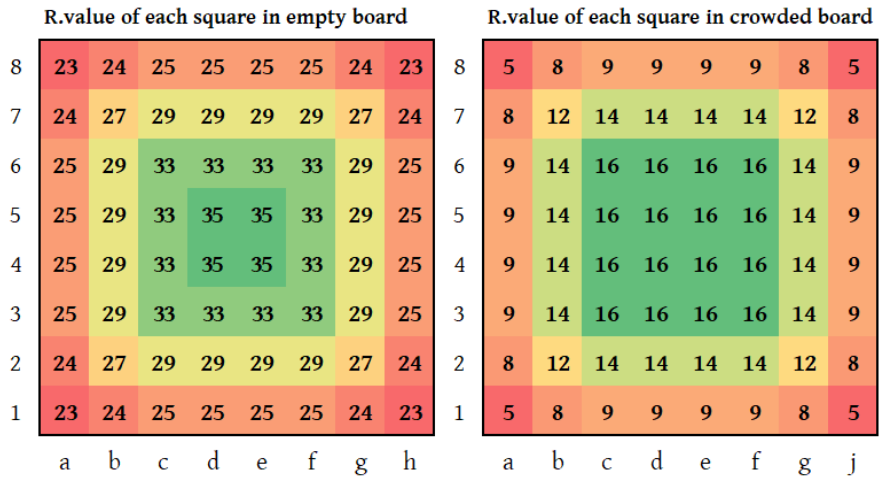


Figure 2

Implementation

Insertion method which will be used to add a node and then nodes to that node based on depth number.

```
void Tree::insert(Node* node){
    if(this->root == nullptr){
        this->root = node;
        return;
    }
    this->root->insert(node);

    return;
    //this->root->nexts.push_back(node);
}
```

Function to check if position on the board is available with no pieces. Also makes sure that this position would be on the inside of the chess board.

```
bool is_free(int row, int col){
    if(row < 0 || row > 7 || col < 0 || col > 7){
        return false;
    }
    if(board[row][col] == 0){
        return true;
    }
    return false;
}
```

Function to check if position on the board is takeable as no pieces are there or opposing players' pieces are there for the taking. Function also checks that position would be inside the chess board.

```
bool is_takeable(int row, int col){
    if(row < 0 || row > 7 || col < 0 || col > 7){
        return false;
    }
    char space = board[row][col];
    if(space == 0 || space == 'p' || space == 'r' || space == 'n' || space == 'b' || space == 'q' || space == 'k'){
        return true;
    }
    return false;
}
```

Implementation of insert method and find_all_moves function: Will store all moves in the moves vector of vector by calling find_all_moves function. Then move every single move based on depth number, add it as a node then add nodes to that node in the for loop.

```
//the logic behind black's moves.
void Board::black_moves(){
    //a vector or list that will hold all possible moves IN NODE FORM.

    //Position pos;
    std::vector<std::vector<int>> moves = find_all_moves();


    Tree moveTree;

    for(int i = 0; i < moves.size(); i++){
        Node * node = new Node(moves[i]);
        moveTree.insert(node);
    }

    int maxScore = -1000;
    int maxIndex = rand() % moves.size();

    moveTree.print();

    std::vector<int> bestMove = moveTree.get_best_move();
    board[bestMove[2]][bestMove[3]] = board[bestMove[0]][bestMove[1]];
    board[bestMove[0]][bestMove[1]] = 0;

}
```

Function which would create a node for every possible move and store them in moves vector of vectors. This function allows us to use min-max and find all possible moves to a given depth. Use is_free and or is_takeable function to check if a move could be possible depending on piece as shown in case switch statements and if so, would push back the int vector which would be created in add_postions which would calculate the score of the move and also include the move itself.

```
std::vector<std::vector<int>> Board::find_all_moves(){
    //positional score:
    //std::cout << "Score: " << score << "\n"; //Generate nodes for each possible move, store them in vector
    std::vector<std::vector<int>> moves;
    for(int i = 0; i < 8; i++){
        for(int j = 0; j < 8; j++){
            if(board[i][j] != 0){
                switch(board[i][j]){
                    case 'P':
                        if(is_free(i+1,j)){
                            //std::vector<int> temp = ;
                            moves.push_back(add_position(i,j,i+1,j));
                        }
                        if(!is_free(i+1,j+1) && is_takeable(i+1,j+1)){
                            moves.push_back(add_position(i,j,i+1,j+1));
                        }
                        if(!is_free(i+1,j-1) && is_takeable(i+1,j-1)){
                            moves.push_back(add_position(i,j,i+1,j-1));
                        }
                        break;
                    case 'R': //ROOK MOVES
                        for(int dis = 1; dis < 8; dis++){
                            if(is_takeable(i+dis,j)){
                                moves.push_back(add_position(i,j,i+dis,j));
                                if(!is_free(i+dis,j)){
                                    break;
                                }
                            }
                            else{
                                break;
                            }
                        }
                        for(int dis = 1; dis < 8; dis++){
                            if(is_takeable(i-dis,j)){
                                moves.push_back(add_position(i,j,i-dis,j));
                                if(!is_free(i-dis,j)){
                                    break;
                                }
                            }
                            else{
                                break;
                            }
                        }
                        for(int dis = 1; dis < 8; dis++){
                            if(is_takeable(i,j-dis)){
                                moves.push_back(add_position(i,j,i,j-dis));
                                if(!is_free(i,j-dis)){
                                    break;
                                }
                            }
                            else{
                                break;
                            }
                        }
                        for(int dis = 1; dis < 8; dis++){
                            if(is_takeable(i,j+dis)){
                                moves.push_back(add_position(i,j,i,j+dis));
                                if(!is_free(i,j+dis)){
                                    break;
                                }
                            }
                            else{
                                break;
                            }
                        }
                        ;break;
                    case 'B':
                        for(int dis = 1; dis < 8; dis++){
```

This function is called in find_all_moves function to create a vector which will be stored in the moves vector when called in the find_all_moves of vectors function. This function not only helps us store a vector with the specific move for the piece but also the score of the move.

```
std::vector<int> add_position(int x, int y, int xx, int yy){
    std::vector<int> temp;

    char car = board[xx][yy];
    board[xx][yy] = board[x][y];
    board[x][y] = 0;
    int score = 0;
    for(int i = 0; i < 8; i++){
        for(int j = 0; j < 8; j++){
            score += pos_eval(board[i][j]);
        }
    }
    temp.push_back(x);
    temp.push_back(y);
    temp.push_back(xx);
    temp.push_back(yy);

    temp.push_back(score);

    board[x][y] = board[xx][yy];
    board[xx][yy] = car;

    return temp;
}
```

Contributions

This project was a collective effort as all of the team members contributed to each aspect of the project in different ways. As pictured below the project was split up into separate sections with each member primarily focusing on their specific areas, but also coming together to collaborate with and critique each other's work when necessary.

Name	Contact Info	Contributions
Quinn Smith	qmith227@uri.edu	Primarily wrote Project Report, Contributed to Presentation, Source Code Troubleshooting, Command line argument tests
David Motta	david_motta@uri.edu	Researched chess engine components, wrote source code,
Felipe Pareja	felipepareja@uri.edu	Wrote implementation parts for Project report and slides. Contributed to Presentation, exception handling and Source Code Troubleshooting. Wrote the Readme markdown.