A Training Report

On

AMERICAN CHECKERS - AI

Submitted in partial fulfilment of requirements for the award of the

Degree of

Bachelor of Technology

In

Computer Science & Engineering

Submitted By

Priyanshi Gupta

(Enrolment No: 03311502718)

Under the guidance of

Mr. Ankush Singla & Mr. Parikh Jain

(Instructors, Coding Ninjas, New Delhi)

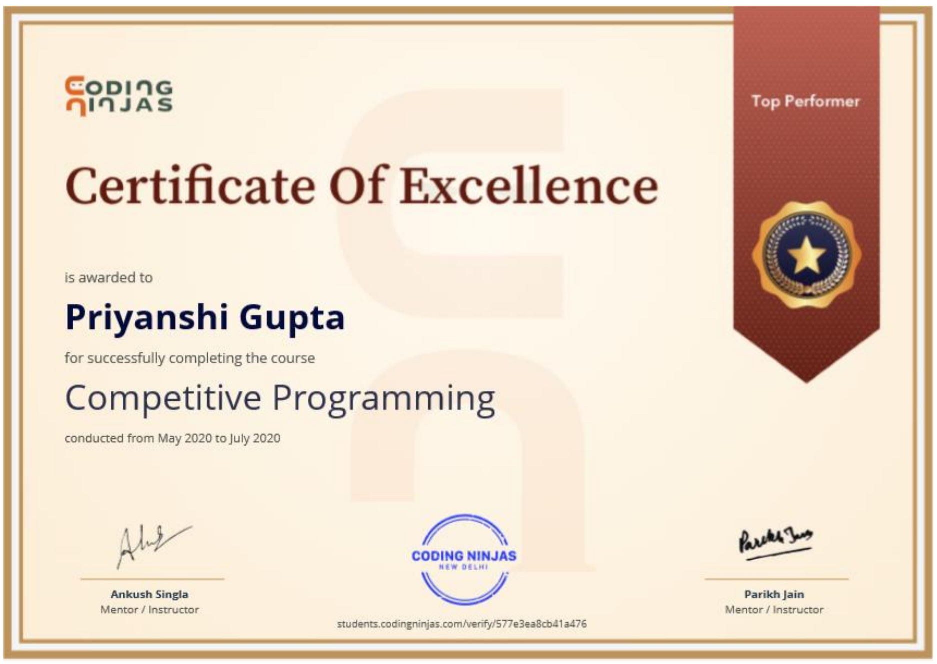
Department of Computer Science & Engineering

Bharati Vidyapeeth's College of Engineering

A-4, Paschim Vihar, New Delhi-110063

June, 2020

CERTIFICATE BY THE COMPANY


CODING NINJAS

Top Performer

Certificate Of Excellence

is awarded to

Priyanshi Gupta

for successfully completing the course

Competitive Programming

conducted from May 2020 to July 2020

[illegible]

CODING NINJAS
NEW DELHI

[illegible]

Ankush Singla
Mentor / Instructor

Parikh Jain
Mentor / Instructor

students.codingninjas.com/verify/577e3ea8cb41a476

CANDIDATE'S DECLARATION

I hereby declare that the work presented in this report entitled "AMERICAN CHECKERS GAME USING AI", in partial fulfilment of the requirement for the award of the degree **Bachelor of Technology** and submitted in **Department of Computer Science & Engineering, Bharati Vidyapeeth's College of Engineering**, , **New Delhi** (**Affiliated to Guru Gobind Singh Indraprastha University**) is an authentic record of my own work carried out during the period from June – July 2019 under the guidance of **Mr. Varun Srivastava, Assistant Professor** at Bharati Vidyapeeth's College of Engineering, New Delhi - 110063.

The work reported in this has not been submitted by me for award of any other degree of this or any other institute.

Priyanshi Gupta
03311502718

ACKNOWLEDGEMENT

I express my deep gratitude to **Mr. Parikh Jain and Mr. Ankush Singla**, Instructors and Founder of Coding Ninjas, for their valuable guidance and suggestion throughout my training. We are thankful to Ms. Preeti Nagrath (CSE, Morning) for her valuable guidance.

Note: Students may thank to the persons whom they would like to acknowledge.

Priyanshi Gupta
03311502718

COMPANY PROFILE

Coding Ninjas India

Coding Ninjas is an EdTech start-up founded in the year 2016. Mr. Ankush Singla, CEO and Co-founder and Mr. Parikh Jain, Co-founder developed one of the largest online tech education company in India.

Coding Ninjas provides online programming courses that range from Basic foundation-level courses as well as Advanced-level courses, like C++, Java, Python, Android, Machine Learning, Data science, Web Development, etc. Coding Ninjas has a pool of faculty from some of the best educational institutes like Stanford University, IITs, IIITs and DTU. They have industrial experience of working with the biggest tech companies like Facebook, Amazon, Adobe etc.

Coding Ninjas currently has a monopoly position across the college market in India. With the vision to teach millions in a scalable way, Coding Ninjas has pioneered a proprietary online teaching platform, which completely mirrors the offline classroom experience into online, and thus delivers a world-class learning experience to students. With an in-house placement cell, Coding Ninjas is actively involved in sourcing relevant tech openings and showcasing Coding Ninjas student profiles to get them a rewarding career in tech.

Contents

PREFACE

This report is prepared for partial fulfilment the requirement for the Bachelor in Technology program of Bharati Vidyapeeth’s College of Engineering, New Delhi on "American Checkers using AI”. I have chosen Game theory as my training project since it focuses on implementation and visualisation of competitive programming algorithms.

The prime focus of this study is to implement a 1-player checker game using minimax algorithm and alpha-beta pruning and   For development I've used JavaScript, HTML and CSS since it is the leading technology is creating GUI's for board games.

This project report is divided into several sections. The first section gives an introduction of checkers games and its probable computations along with rules. The second section is the literature survey which tells about the existing computer programs to solve checkers and its history associated with it. The third section is the methodology section which gives details about the methods and fields used in the project. The fourth section is the analysis and comparison section which gives detailed analysis of the algorithms used in the project and its comparison with other algorithms. The fifth section is the implementation section in which the applicative view of the algorithm is briefly explained along with implementation of GUI in the game and its snapshots. The sixth section is the utility and application section which tells the basic uses, features and the system environment of the game developed. The last section is the conclusion section in which difficulties, achievements, results and findings along with future scope of the project is mentioned.

Priyanshi Gupta

03311502718

List of Figures

Weekly Progress

WEEK	CONTENT COVERED
Week 1	• Prerequisites • Introduction to Competitive Programming • Pointers
Week 2	• Dynamic Allocation • Basics of Recursion • Time Complexity Analysis
Week 3	• STL • Assignment • Searching & Sorting
Week 4	• Advanced Recursion • Backtracking • Assignment
Week 5	Project Development
Week 6	Project Development

1 Introduction

Checkers is a set of strategy board games for two players that involve diagonal movements of uniform game pieces and compulsory captures by jumping over opponent pieces [1]. The name comes from the verb for drawing or moving [2]. American checkers is the most famous form of checkers. It is played on an 8×8 board of light and dark squares in the famous “checkerboard” pattern, with 12 pieces per side. The pieces move and capture forward diagonally until they enter the opposite end of the board where they are crowned and can move and capture backwards and forward afterwards. Like in all types of checkers, it is played by two opponents, taking turns on opposite sides of the board. The typical pieces are black, red, or white. Enemy pieces are caught by jumping over them. The rules are explained more elaborately below [3]:-

- Each player starts at the dark squares of the three rows closest to that player's side with 12 men. The closest row to each player is called the row of kings, or crown head. The player with the darker-colored pieces moves first and then each of the player take alternate turns.
- In each turn, a player can make a simple move, a single jump, or a multiple jump move.
  - *Simple move*: Single pieces can move one adjacent square diagonally forward away from the player. A piece can only move to a vacant dark square.
  - *Single jump move*: A player captures an opponent's piece by jumping over it, diagonally, to an adjacent vacant dark square. The opponent's captured piece is removed from the board. The player can never jump over, even without capturing, one of the player's own pieces. A player cannot jump the same piece twice.
  - *Multiple jump move*: Within one turn, a player can make a multiple jump move with the same piece by jumping from vacant dark square to vacant dark square. The player must capture one of the opponent's pieces with each jump. The player can capture several pieces with a move of several jumps.

- If a jump move is possible, the player must make that jump move. A multiple jump move must be completed. The player cannot stop part way through a multiple jump. If the player has a choice of jumps, the player can choose among them, regardless of whether some of them are multiple, or not.
- When a single piece reaches the row of the board furthest from the player, i.e the king-row, by reason of a simple move, or as the completion of a jump, it becomes a king. This ends the player's turn. The opponent crowns the piece by placing a second piece on top of it.

Fig. 1. An 8x8 checkers board with 12 pieces of red and black color on each side

- A king follows the same move rules as a single piece except that a king can move and jump diagonally forward away from the player or diagonally backward toward the player. Within one multiple jump move, the jumps can be any combination of forward or backward jumps. At any point, if multiple jumps are available to a king, the player can choose among them.
- A player who loses all of their pieces to captures loses the game.

The motivation to implement a 1 player checkers game was due to the following reasons :

- The number of possible positions in American Checkers is 500,995,484,682,338,672,639 [4] and it has a game-tree complexity of approximately 1040 [5]. By comparison, chess is estimated to have between 1043 and 1050 legal positions. Therefore checkers has a smaller search space than chess which made it considerably solvable.
- The rules as mentioned above are considerably simple due to limited and similar moves which reduced the intrinsic complexity of the applied algorithm [6].
- The interactions between the pieces is very less in checkers as compared to chess due to different types of pieces.

2 Literature Survey

The computer program for the first American checkers was written by Christopher Strachey, M.A. In February 1951, Strachey finished the programme, written in his spare time, at the National Physical Laboratory in London [7]. On 30 July 1951, it ran on NPL's Pilot ACE for the first time. He was easy to change the software to run on Manchester Mark 1.
The second computer program was written by IBM researcher Arthur Samuel in 1956. Apart from being one of the most complex game-playing programs written at the time, it is also considered to be one of the first adaptive programmes. It learned by playing games against modified versions of itself, with surviving victorious versions. The program's main function was a search tree of board positions which can be reached from the current state. Since he had only a very limited amount of computer memory available, Samuel implemented alpha-beta pruning [8]. Instead of searching each path until it came to the conclusion of the game, Samuel developed a scoring function at any given time, based on the board's position. This function attempted to evaluate the chance of winning at the given position for each side. It took things like the number of pieces on either hand, the number of kings and the proximity of pieces to "kinged" into consideration. The program chose its move based on minimax approach , which means it made the move that optimized the value of this function, assuming that from its point of view, the opponent was attempting to optimize the value of the same feature [9].
The strongest program was made in the 1990's called Chinook, written in 1989 by a team led by Jonathan Schaeffer at the University of Alberta. It is the first computer program to win a world champion title in the checkers competition against a human. The program algorithm for Chinook contains an opening book, a library for opening moves from games played by grandmasters; a deep search algorithm; a strong movement evaluation function; and an end-game database for all positions with eight or fewer pieces. The linear handcrafted evaluation method takes into account some of the game board's features, including piece count, kings count, trapped kings, switch, runaway checkers (unimpeded route to be kinged) and other minor. Most knowledge of Chinook has been designed by its developers, rather than developed by artificial intelligence. The July 2007 announcement by Chinook's team stating that the game had been solved must be understood in the sense that, with perfect play on both sides, the game will always finish with a draw. However, not all positions that could result from imperfect play have been analysed [10].
In this project, 1 player checker game using minimax algorithm and alpha beta pruning is implemented.

3 Methodology

3.1 Game Theory

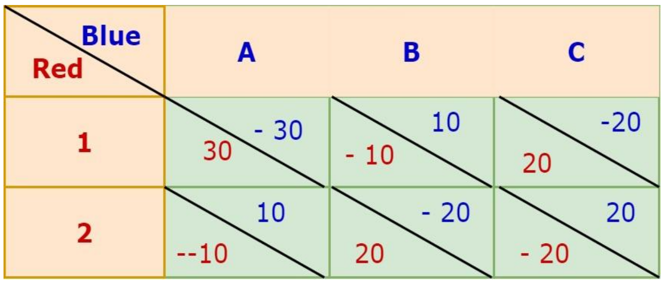
Decision making process is a process which includes decision makers, actors, environmental factors, objectives, strategies and criteria. In competitive environments, effectiveness of decision process depends on determining all environmental factors and evaluating them according to objectives. Decision makers aim to find optimal strategies for conflicting objectives. Game theory is an approach based on mathematics in which strategies of players are evaluated reciprocally by considering environmental effects [11]. Game theory provides tools for analysing situations in which parties, called players, make decisions that are interdependent. This interdependence causes each player to consider the other player's possible decisions, or strategies, in formulating his own strategy. A solution to a game describes the optimal decisions of the players, who may have similar, opposed, or mixed interests, and the outcomes that may result from these decisions.
Game theory has been applied to a wide variety of situations in which the choices of players interact to affect the outcome. In stressing the strategic aspects of decision making, or aspects controlled by the players rather than by pure chance, the theory both supplements and goes beyond the classical theory of probability.

3.2 Zero-Sum Game

In game theory and economic theory, a zero-sum game is a statistical description of a situation throughout which the gain or loss of value of each participant is precisely compensated by the other participants' losses or gains in usefulness. If the participants' s total gains are added, and the total losses are subtracted, they will add up to zero. For instance in cake cutting, where taking a bigger slice decreases the amount of cake available to others as much as it increases the amount available to that taker, is a zero-sum game if all participants value each unit of cake equally. Non-zero-sum, by contrast, describes a situation in which the net gains and losses of the interacting parties may be less than or greater than zero. Also a zero-sum game is called a purely competitive game whereas non-zero-sum games can be competitive or non-competitive. Zero-sum games are most commonly solved with the minimax theorem, which is closely related to the duality of linear programming [11], or Nash equilibrium. Many people have a cognitive predisposition to see situations as zero-sum, known as zero-sum bias.
The different game-theoretical solutions principles of Nash equilibrium, minimax, and maximin all give the same solution for two-player finite zero-sum games. If the players are allowed to play a mixed strategy, there's always a balance in the game.

Example

The payoff matrix for a game is a convenient representation. Consider the zero-sum two-player game depicted in the figure 2. The order of play proceeds as follows: The first player (red) selects one of two actions 1 or 2 in secret; the second player (blue), unaware of the choice of the first player, selects one of three actions A, B or C in secret. Then the choices are revealed, and the total points of each player are affected for those choices according to the payoff.
Red opts for action 2 and Blue opts for action B. When allocating reward, Red receives 20 points and Blue loses 20 points. Instead of agreeing on a definite action to be taken, the two players allocate probabilities to their respective actions, then use a random system that selects an action for them according to these probabilities. Each player measures the probabilities so as to reduce the predicted maximum point-loss regardless of the opponent 's strategy. This leads to a linear programming problem with each player having the optimal strategies. This minimax method can probably calculate optimal strategies for all zero-sum games with two players. The final probability that Red should choose action 1 comes out to 4/7 and for action 2, it comes out to be 3/7. For Blue, the probabilities for actions A, B, and C are 0, 4/7, and 3/7 respectively. Consequently Red is the winner with 20/7 points on average per game.



Blue Red	A	B	C
1	- 30 / 30	10 / - 10	-20 / 20
2	10 / --10	- 20 / 20	20 / - 20

Fig. 2. An example of payoff matrix for a zero-sum two player game

3.3 Heuristic Function

Heuristic in computer science , artificial intelligence, and mathematical optimization is a methodology designed to solve a problem faster when classic methods are too slow, or to search an approximate solution when classic methods fail to find an exact solution. This is achieved through trade for optimality, completeness , accuracy, or speed accuracy. This could be called a shortcut, in a way.
A heuristic function is a function that ranks alternatives in search algorithms at each branching stage based on the information available to determine which branch to follow. It could approximate the exact solution [12].
In this project, heuristic function is used to make the searching in the algorithm faster. Initially the heuristic, like the full-space search algorithm, tries every possibility at each stage. However if the current possibility is actually worse than the best option actually found, it may interrupt the search at any time. A heuristic is used in such search problems to try good choices first, so that bad paths can be eliminated early like alpha-beta pruning. In the case of best-first search algorithms like A * search, the heuristic increases the convergence of the algorithm while preserving its consistency as long as the heuristic is admissible.

The heuristic value is calculated through evaluation functions.

3.3.1 EVALUATION FUNCTIONS

As mentioned before the algorithms use different heuristics to form various evaluation functions. The function used in our project is:

Heuristic Function(PieceSumDifference)
Pawn’s Value = 1
King’s Value =3

3.4 Search Tree

A search tree is a tree data structure which is used to find unique keys within a collection. For a tree to act as a search tree, the key for each node must be larger than any subtree keys on the left, and less than any subtree keys on the right [13].
The benefit of search trees is that their efficient search time provided the tree is relatively balanced, meaning the leaves are of equal depths at either end. There are numerous search-tree data structures, some of which also allow for the efficient insertion and deletion of elements, which operations must then preserve tree balance.
In this project, Binary Search Tree is implemented. Binary Search Tree is a node-based data structure of binary tree which has the following properties:

- A node's left subtree includes only nodes with smaller keys than the key of the node.
- A node's right subtree includes only nodes that have keys greater than the key of the node.
- Every left and right subtree must be a binary search tree, too.

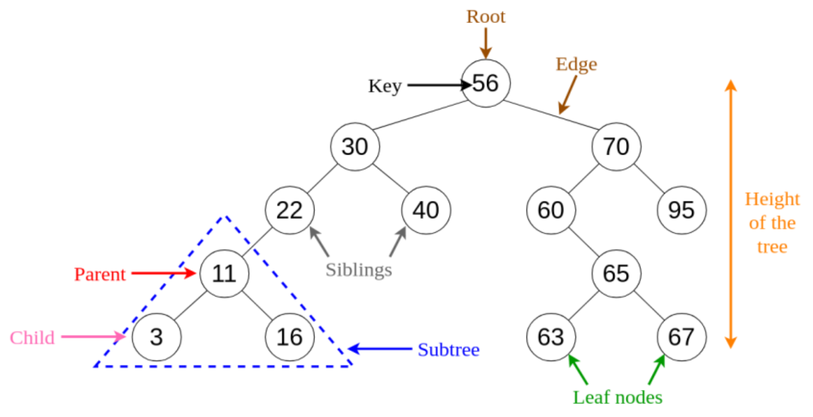


Fig. 3. Binary search tree example

3.5 Minimax Approach

Minimax is a decision rule that is used in artificial intelligence and game theory so as to reduce the possible loss while maximising the potential gain. It can also be thought of as maximising the minimum gain called as maximin. The rule was originally developed for two-player zero-sum game theory for two cases-players make simultaneous moves and players make alternate moves. Each player minimizes the maximum gain possible for the other player. Being zero-sum game, a player also maximizes his own minimum gain. However, minimax and maximin are not equivalent. Maximin may be used in non-zero-sum game scenarios. In this project, minimax algorithm [14] is used as a recursive algorithm in the two player game to select the next move. Every location or state of the game has a value. This value is calculated using a function of position evaluation, and it shows how good it would be for a player to reach that position. Then, the player makes the move that maximizes the minimum position value resulting from the probability of the opponent following moves. If it is A 's turn to move, A gives a value to each of their legal moves.
One can think of the algorithm as exploring the nodes of a game tree. The tree's efficient branching factor is the average number of children per node (i.e. , the average number of legal moves in a position). The number of nodes to be explored usually increases exponentially with the number of plies (in the case of evaluating forced moves or repeated positions it is less than exponential). Consequently, the number of nodes to be explored for the analysis of a game is roughly the branching factor raised to the power of the number of plies. The efficiency of the minimax algorithm can be significantly improved by the use of alpha-beta pruning, without affecting the outcome.

Algorithm

```
MAX(x,y) = -MIN(-x, -y)
function minimax(node, depth)
Step 1: if (leaf node) or (depth <= 0)
    return(node value);
Step 2: Initiate alpha = -∞;
Step 3: for child in node{
                alpha = MAX(alpha, -minimax(child, depth-1))
                }
Step 4: return (alpha);
```

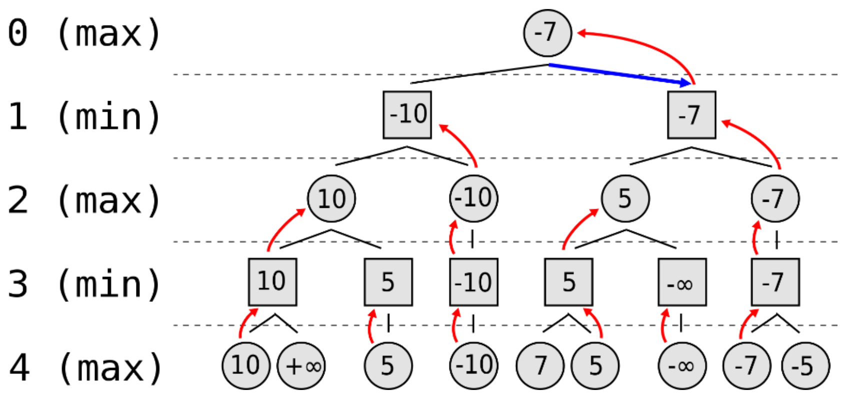


Fig. 4. A minimax tree example

3.6 Alpha-Beta Pruning

Minimax approach amounts to exhaustive search of solution space. In a realistic game such as checkers, the search space is large. It is therefore more efficient to heuristically reduce the search space in case of minimax solutions. Alpha-beta pruning algorithm provides mechanism to decrease the number of count of the nodes thus reducing the search space in minimax tree [15]. Apart of checkers, it is also used in other two player games such as tic-tac-toe and chess. Further the search is stopped once it encounters a branch where at least a single value has been found that confirms that the branch would be worst in comparison to earlier payoffs. Therefore, these moves can be avoided in the game to reduce unnecessary computation. Alpha-beta pruning does the same. It discards the branches of the search tree without impacting any of the decisions to return the same result in spite of pruning [16]. Apart from reducing the search time, it is possible to have a deeper search allowing a greater depth of sub-tree to be scanned. The algorithm comprises of two values – alpha and beta. Alpha value holds the maximum(lowest) score that the maximising player is assured of. Beta value holds the maximum score that the minimizing player is guaranteed to earn. The initial condition is equal to – and =+. As the search progresses, this difference becomes smaller. When beta value becomes lower than alpha, it signifies that the search beyond the current node can be restricted.

Algorithm

```
function alpha_beta(player, position, alpha, beta)
Step 1: if (game decided in current board position)
            Return (Winning Player);
Step 2: children = all permissible moves for a player from node
Step 3: if (Maximizing Player to Play){
            for (each child){
            Step 3.1: value = alpha_beta (other player, child, alpha, beta)
            Step 3.2: if (value > alpha) alpha = value;
            Step 3.3: if (alpha >= beta) return (alpha);
            }
            Step 3.4: return (alpha);
            }
Step 4. else (Minimizing Player to Play){
            for(each child){
        Step 4.1: value = alpha_beta(other player, child, alpha, beta)
        Step 4.2: if (value < beta) beta = value;
        Step 4.3: if (alpha >= beta) return beta;
            }
```

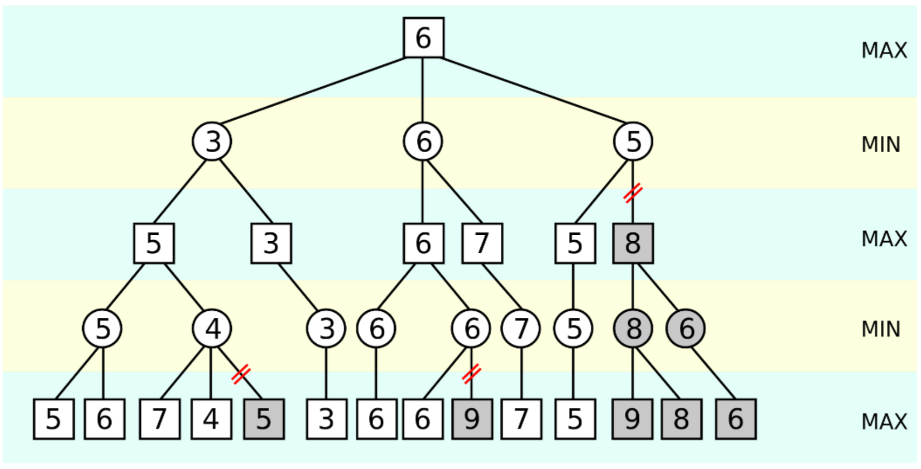


Fig. 5. An illustration of alpha beta pruning

In the figure 5, the grayed-out subtrees (when moves are evaluated from left to right) are discarded by alpha beta pruning and are not needed to be explored, since it is known that the group of subtrees as a whole yields the value of an equivalent subtree or worse, and as such cannot influence the end result. The max and min levels respectively reflect the player's turn and the opponent's turn.

4 Analysis and Comparison

4.1 Minimax vs Alpha-Beta Pruning

The benefit of alpha – beta pruning lies in the ability to delete search tree branches. By this way, the search time can be reduced to the 'more promising' subtree, and at the same time a broader search can be carried out. Unlike its predecessor, it is a part of the algorithm branch and bound class. Optimization reduces the effective depth to just over half that of simple minimax if the nodes are evaluated in an optimal or near optimal order (best option for side-by - side movement ordered at each node first). As shown in Figure 6, Alpha-Beta is more time efficient than Minimax Algorithm. Typically during alpha-beta, the subtrees are temporarily dominated by either a first player advantage (when many first player movements are fine, and the first move tested by the first player at each search depth is adequate, but all second player responses are needed to try to find a refutation), or vice versa. This advantage will repeatedly turn sides during the quest if the ordering of the move is wrong, each time leading to inefficiency. As the number of searched positions decreases exponentially every move closer to the current position, considerable effort is worth spending on sorting early moves.

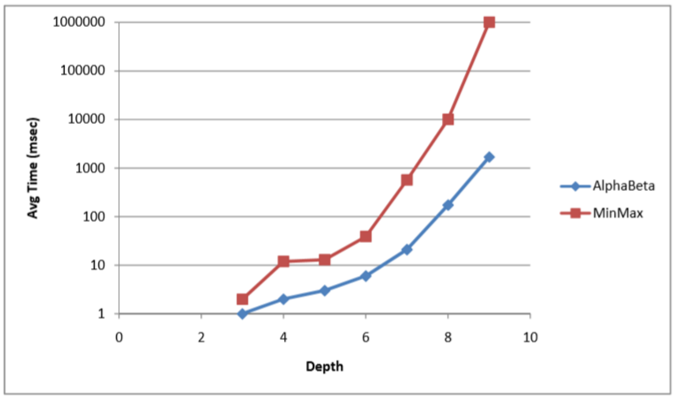


Fig. 6. Average Time per move vs Depth graph on logarithmic scale using Minmax and Alpha-Beta Algorithm

4.2 Best Case Analysis of Alpha Beta Pruning

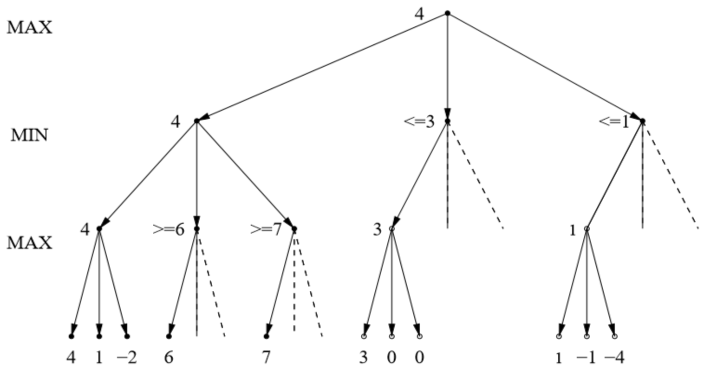


Fig. 7. An example of best-case behavior in alpha-beta pruning

Within this study we find the alpha-beta pruning's best-case behavior. It turns out that this correlates fairly well to normal actions [17]. In the figure 7, the branching factor is 3, and the depth is 3. Notice that when states are considered for MAX's move, they are considered from highest to lowest evaluation. For MIN's move, they are considered from lowest to highest. A good way to approximate this ordering is to simply apply the evaluation function and sort the states accordingly. Generally, the maximum/minimum state will be among the first few. Now assuming that best-case analysis is a reasonable thing to do in this case, let's first consider recursive equations which give the number of states to be considered. From the equations, we will determine a rough bound on the branching factor. Note that in the figure, we either know the value of a state exactly, or we know a bound on its value. To determine the exact value of a state, we need the exact value of one of its children, and bounds on the rest of its children. To determine a bound of a state's value, we need the exact value of one of its children. Based on these observations, let S(k) be the minimum number of states to be considered k ply from a given state when we need to know the exact value of the state. Similarly, let R(k) be the minimum number of states to be considered k ply from a given state when we need to know a bound on the state's value. As usual, let b be the branching factor. Thus, we have:

$$S(k) = S(k-1) + (b-1)R(k-1)$$

i.e., the exact value of one child and bounds on the rest, and

$$R(k) = S(k-1)$$

i.e., the exact value of one child. The base case is S(0) = R(0) = 1. Note, for the above figure, this gives S(3) = b2 + b−1 = 11 for b = 3. When we expand the recursive equation, we get:

$$\begin{aligned} S(k) &= S(k-1) + (b-1)R(k-1) \\ &= (S(k-2) + (b-1)R(k-2)) + (b-1)S(k-2) \\ &= bS(k-2) + (b-1)R(k-2) \\ &= bS(k-2) + (b-1)S(k-3) \end{aligned}$$

It is obvious that S(k−3) < S(k−2), so:

$$S(k) < (2b-1)S(k-2) < 2bS(k-2)$$

That is, the branching factor every two levels is less than 2b, which means the effective branching factor is less than √2b. So, for even k, we derive $S(k) \leq (\sqrt{2b})^k$, which is not too far off the asymptotic upper bound of $(\sqrt{b} + ½)^{k+1}$. In effect, alpha-beta pruning can nearly double the depth that a game tree can be searched in comparison to straightforward minimax.

5 Implementation

5.1 Game Algorithm

The game algorithm is implemented using JavaScript. Figure 8 describes the implementation in the form of a flowchart. The procedure is as the player completes its turn, a search algorithm is called which is what allows the program to look ahead at the possible future positions before deciding what move it wants to make in that current position.

It’s white turn to move now. In every move, there are only two possible moves to choose between. We can visualize these moves as two separate branches at the end of which are two new positions, of course, it is black’s turn to move now. We continue expanding these moves till either we reach the end of the game or we decide to stop because going deeper would take too much time. Either way, at the end of the tree we now have to perform the static evaluation on these final positions. The static evaluation means try to estimate how good the position is on one side without making any more moves.

Large values would favor white and small values would favor black. For this reason, white constantly tries to maximize the evaluation hence known as the Maximizing player. And black is always trying to minimize the evaluation, hence known as the Minimizing player.

We start by evaluating the positions on the bottom left. In the previous position, it was white’s turn to move, and since white will always choose the value that leads to the maximum evaluation, we assign the value to the node accordingly and complete the evaluation of that node using the right branch as well.

Now, black will try to minimize the evaluation function, so we assign the position the lower value by comparing. And we go up the tree hence returning the maximum-minimum gain and getting the higher probable position for white to move. This is where pruning comes into the picture. It would take a lot of time to go down all the branches to get the best value. Without exploring all the nodes through the branch, it tries to get the least or the max value from the already evaluated position. This would result in the player to know that he already has a better option available and that he won’t have to go down the other branch. These checks are made through alpha-beta parameters. This observation concludes that we don’t have to waste any computation in evaluating the final position. Hence, we’ve pruned that position from the tree.

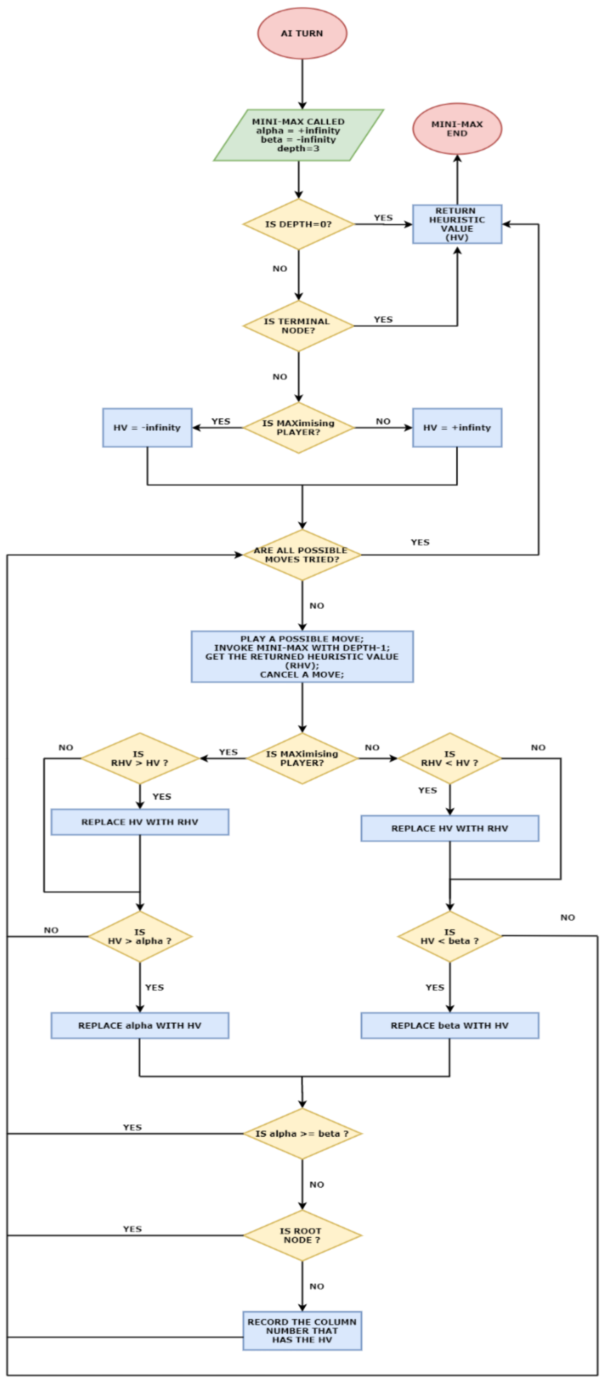


Fig. 8. Flowchart depicting the implementation algorithm

5.2 Graphical User Interface

The Graphical User Interface is built using Html and CSS. The primary reason for building the algorithm in JavaScript was due to fact that it is one of the primary building blocks of web development. Therefore the whole game is implemented as a web page. The board game is given a real checker board like feature to simulate real-time game playing making it visually more attractive. Besides the board, the front end contains rules and tutorials so that the user with no previous knowledge can also play the game without any difficulty. A little brief about the history of checkers board is also mentioned so that the user can know about the origin of the checkers game. Lastly information about chinook is also given. The rule based features such as the upgradation of a piece into king is also depicted in a visually appealing form by depicting a crown on the piece (See Figure 12 and 13).

5.3 Snapshots of the Checkers Game

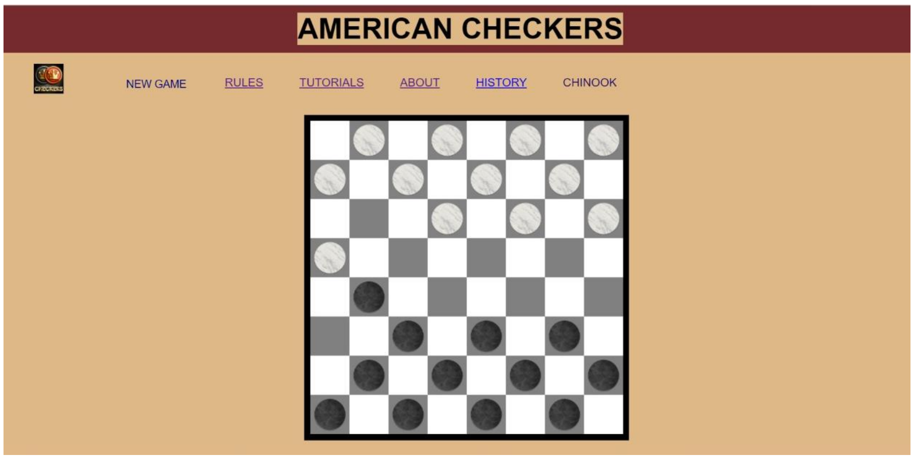


Fig. 9. Before single capture of a piece

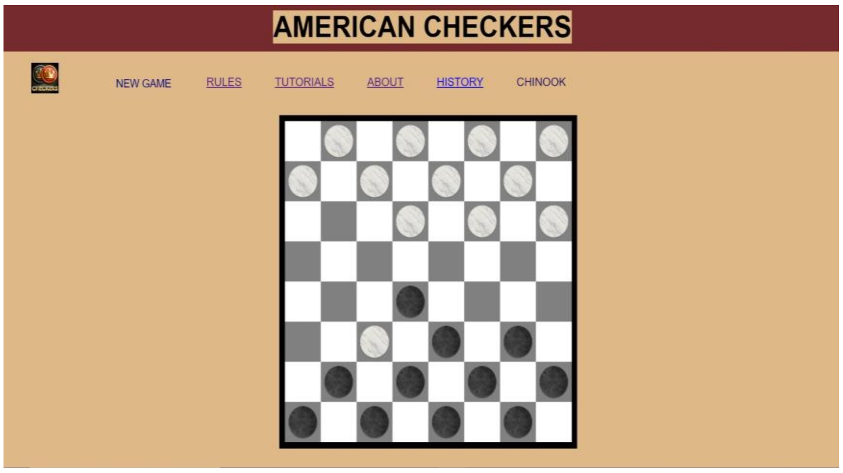


Fig. 10. After single capture of a piece

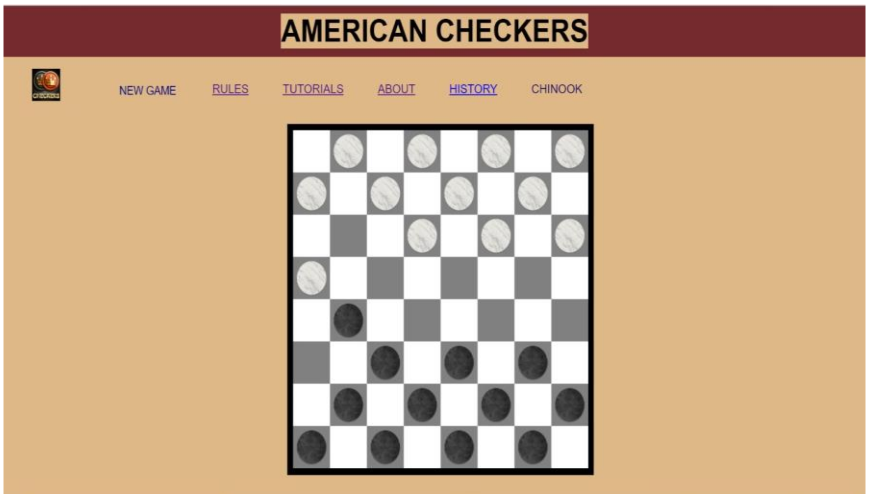


Fig. 11. Before multiple capture of pieces

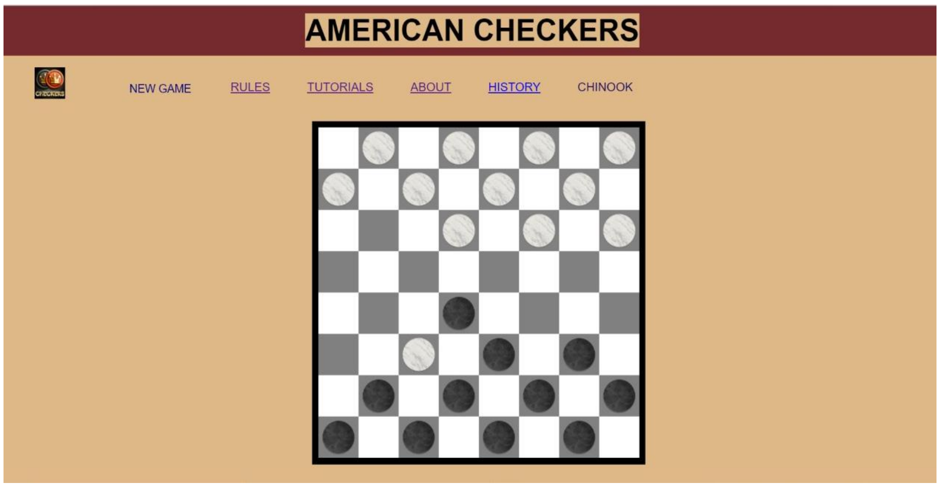


Fig. 12. After multiple capture of pieces

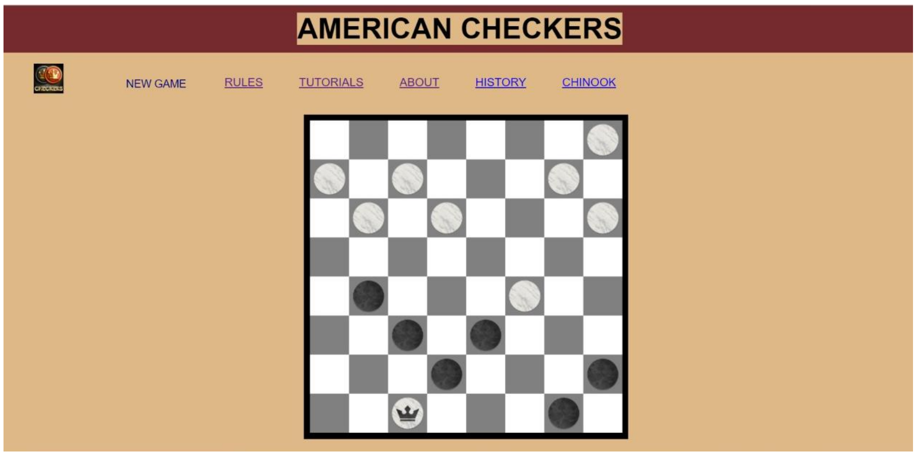
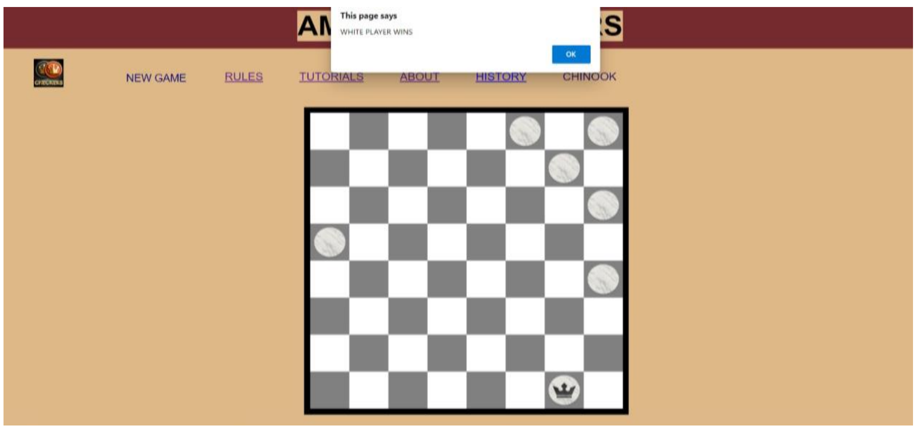


Fig. 13. Upgradation of a piece into king on reaching the end side of the opponent


This page says

WHITE PLAYER WINS

Fig. 14. Displaying the result of the game

6 Utility and Application

This project is an implementation of a strategic board game checkers. It can be used to play checkers in an interactive environment. The game's algorithm is developed using JavaScript which is one of the core technologies of web development. To provide a real-time playing simulation, the graphical user interface of the game is build using Html and CSS. The game is more accessible and user friendly as it is built as a website. The frontend of the web page contains the instructions and rules of the game so that the user can play the game even without having any previous knowledge.
The game is a 1 player checkers game where the user can play with the AI and test their skills in strategic and logical gaming. Some benefits of playing Checkers game are stated below :

- It develops concentration skills and promotes confident decision making.
- It reduces stress and serves as a fun way to overcome boredom.
- It boosts problem-solving and pre-mathematics skills

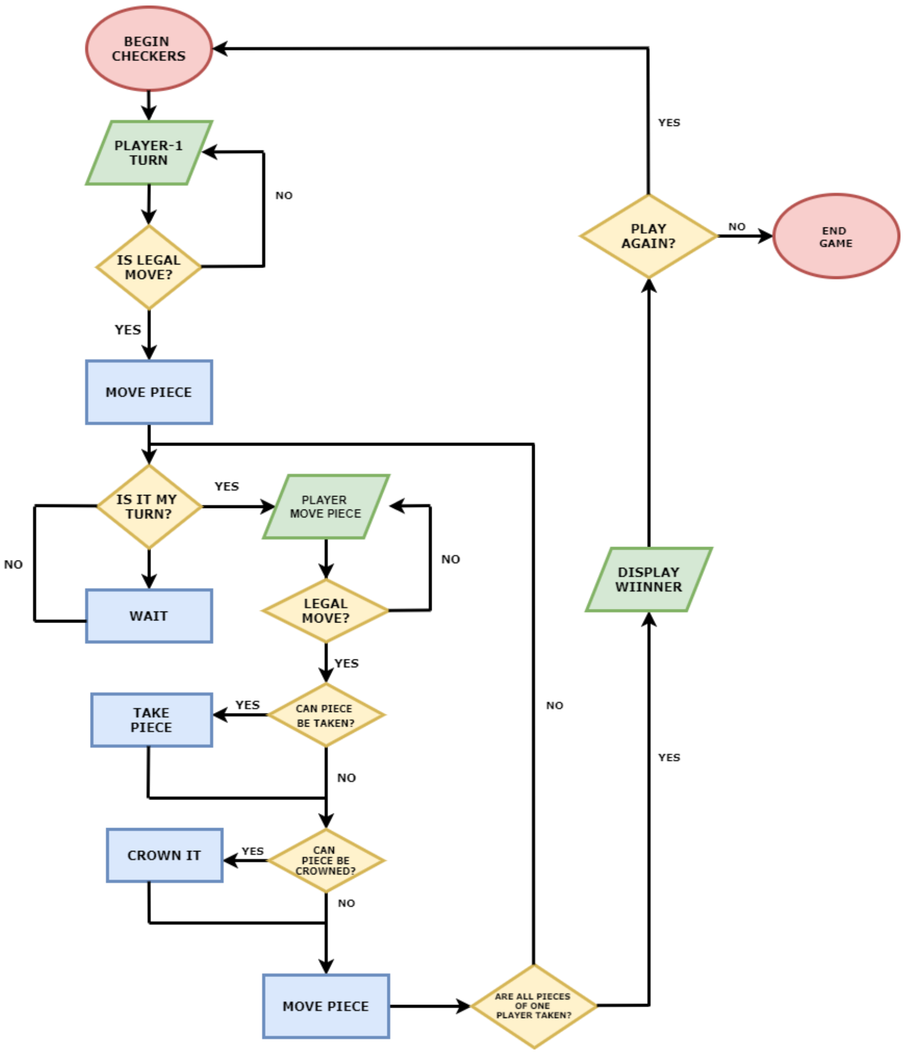


Fig. 15. Use Case Flowchart

6.1 System Environment

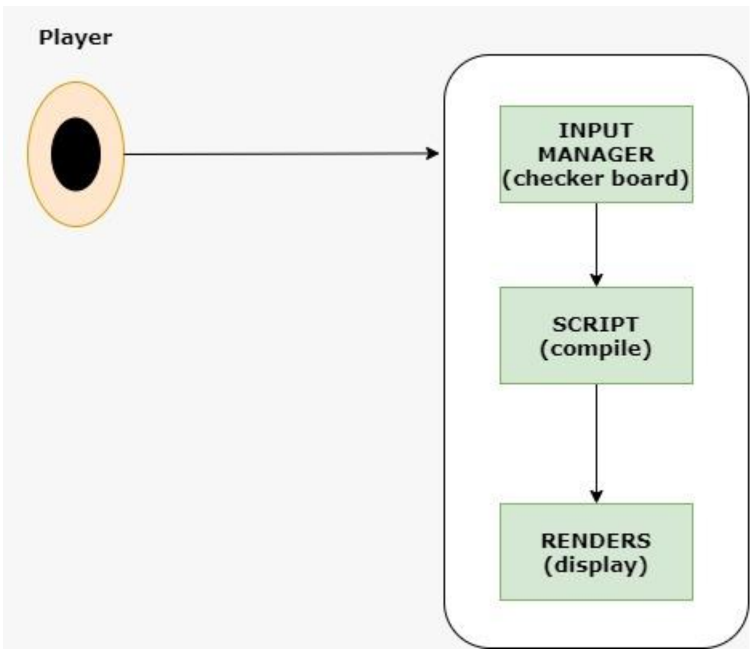


Fig. 16. System Environment Diagram

Gamer can interact with the system by giving input (selecting the pieces and dragging them to the available tile). System gives those input to the script , if any change occurs(if the value is changed), the object send to renders to display the things (a character can change its place).

7 Conclusion

7.1 Difficulties

- Working with JavaScript to write the game algorithm was a completely new experience for us as typically we implement algorithms in languages such as C++, java and python.
- Linking the JavaScript with the front-end built using Html and CSS required looking at details in each file.
- Defining heuristics was a challenging task.
- This project marked an introduction to game theory for us therefore an extensive research on each algorithm with their analysis and comparison was required.

7.2 Achievements

- Researching and building a game with algorithms used in game theory and artificial intelligent helped in getting familiar to the respective fields.
- Implementing binary search tree helped us in improving our knowledge on data structures and their implementation.
- The project helped us in getting expertise over various techniques of competitive programming such as recursion and searching.
- Achieved a new skill in the area of web development.
- As this was a group project, we successfully developed our communication skills and coordination between other group members.

7.3 Results and Findings

- The introduction of different features will significantly boost heuristics.
- The depth of the game tree affects significantly the com-player quality.
- The use of Minimax without optimization is not effective, although it can be a good solution with them.
- Alpha-beta pruning improves exponentially with respect to Minimax as the range increases.
- Simple algorithms as a random player have no chance at depth greater than 1 against Alpha-Beta.
- There are several other solutions to zero-sum play, but Minimax with optimization using heuristic function and alpha-beta pruning tends to be a successful one.

7.4 Future Scope

- Different levels based on the depth of the tree can be introduced in the game such as easy, medium and hard.
- Improve the GUI
- Built a data management system to store the details of the user such as name, number of wins and losses.
- Add new features such as 2 player game, different types of checkers.
- Using same algorithms make more 2 player games such as tic-tac-toe and chess.

References

1. Masters, James. "Draughts, Checkers - Online Guide". www.tradgames.org.uk.
2. Strutt, Joseph (1801). The sports and pastimes of the people of England. London. p. 255.
3. http://www.se.rit.edu/~swen-261/projects/WebCheckers/American%20Rules.html
4. "Chinook - Total Number of Positions". webdocs.cs.ualberta.ca. Retrieved 2017-11-18.
5. Schaeffer, Jonathan (2007). "Game over: Black to play and draw in checkers". ICGA Journal. 30 (4): 187–197. CiteSeerX 10.1.1.154.255. doi:10.3233/ICG-2007-30402.
6. J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, D. Szafron, Reviving the game of checkers, in: D.N.L. Levy, D.F. Beal (Eds.), Heuristic Programming in Artificial Intelligence 2: The Second Computer Olympiad, Ellis Horwood, Chichester, 1991, pp. 119–136.
7. The Proceedings of the Association for Computing Machinery Meeting, Toronto, 1952.
8. Richard Sutton (May 30, 1990). "Samuel's Checkers Player". Reinforcement Learning: An Introduction. MIT Press. Retrieved April 29, 2011.
9. Arthur, Samuel (1959-03-03). "Some Studies in Machine Learning Using the Game of Checkers". IBM Journal of Research and Development. 3 (3): 210–229. CiteSeerX 10.1.1.368.2254. doi:10.1147/rd.33.0210
10. Schaeffer, Jonathan (14 September 2007). "Checkers Is Solved". Science. 317 (5844): 1518–1522. doi:10.1126/science.1144079. PMID 17641166
11. Myerson, Roger B. (1991). Game Theory: Analysis of Conflict, Harvard University Press, p. 1. Chapter-preview links, pp. vii–xi
12. Pearl, Judea (1984). Heuristics: intelligent search strategies for computer problem solving. United States: Addison-Wesley Pub. Co., Inc., Reading, MA. p. 3. OSTI 5127296
13. Black, Paul and Pieterse, Vreda (2005). "search tree". Dictionary of Algorithms and Data Structures
    Figure 3: https://levelup.gitconnected.com/an-into-to-binary-search-trees-432f94d180da
14. Ken Binmore (2007). Playing for real: a text on game theory. Oxford University Press US. ISBN 978-0-19-530057-4., chapters 1 & 7
15. Russell, Stuart J.; Norvig, Peter (2003), Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, pp. 163–171, ISBN 0-13-790395-2
    Figure 4: By Nuno Nogueira (Nmnoguera) http://en.wikipedia.org/wiki/Image:Minimax.svg, created in Inkscape by author, CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=2276653d=2276653

16. Marsland, T.A. (May 1987). "Computer Chess Methods (PDF) from Encyclopedia of Artificial Intelligence. S. Shapiro (editor)" (PDF). J. Wiley & Sons. pp. 159–171. Archived from the original (PDF) on October 30, 2008. Retrieved 2006-12-21.
17. http://www.cs.utsa.edu/~bylander/cs5233/a-b-analysis.pdf
Figure 5: By Jez9999, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3708424