1. Abstract

This report is all about the project that we successfully developed using C++ Programming language with the Object Oriented approach. Computer games have become as inseparable part of our modern daily life. Chess, one of the best globally played indoor mind game, helps in sharpening our mind. Hence our project was to design the chess game which can be played with the computer itself using artificial intelligence (AI). The scope of this project has been briefly described in the end. We wished to implement a chess game which would run on a Windows machine, which provides us with a great deal of practice with data structures and algorithms implemented in C++ and object oriented approach. In this project, we have used object oriented approach and almost all features of C++ like objects, virtual functions, abstract classes, inheritance, manipulators, constructors, function overloading, operator overloading, string stream, user defined manipulators and templates as per requirement of this project. Furthermore, for interactive GUI (Graphics User Interface) we linked SFML library to code blocks and DEV C++ coded for our project. The main problem with creating a chess game was the problems which were associated with the amount and complexity of rules in the normal chess game. The other problem that we faced in implementing a chess game was developing an AI which was capable of making somewhat intelligent decisions. The seperation of work load made it a bit easier in this project as Sangam and Nischal developed the AI version of chess game and Shiva and Sunil worked on parts of game interface (GUI).

2. Table of Content

Contents

3. List of Figures

4. List of Abbreviations

AI = Artificial Intelligence

CPU = Central Processing Unit

GCC = GNU Compiler Collection

GPU = Graphics Processing Unit

GUI = Graphical User Interface

IDE = Integrated Development Environment

OOP = Object Oriented Programming

OPENGL = Open Graphics Library

SFML = Simple and Fast Multimedia Library

5. Objectives

The major objectives of this project can be enlisted as follows:-

1) The main objective of this project is to develop our skill and knowledge about the Object oriented Programming.
2) To be able to use various features of C++ Programming which were absent in C Programming.
3) To be able to cope up in group and be familiar with process while making a project and proposal writing as well as report writing.
4) To make optimal use of Object Oriented approach and other features of C++ programming language like classes, manipulators, virtual functions, inheritance, file handling, templates and exception handling.
5) To provide a user friendly interactive environment to the users that aid them to play chess game with a lot ease.
6) To provide help to the users in playing chess in case if they are confused with the rules of chess game and different moves of different pieces etc are being explained to the users, if they require.
7) The care has been taken that the application has less CPU usage, so that other applications can be performed, if required.

6. Introduction

As a part of curriculum of Bachelors in Electronics and Communication Engineering we are supposed to do a project of Object Oriented Programming in C++ programming language which is done with the objective of testing our knowledge and skills in Object Oriented programming as well as to enhance our creativity and team coordination. It has application of various features of object oriented programming such as objects, class, polymorphism, inheritance, data hiding, data encapsulation, function overloading, operator overloading, and other features which distinctly separates our program from procedure oriented programming. We have designed our project in the Codeblocks and DEV C++ IDE which use the GCC Compiler. Since we have intentions to do our project with graphics works we carried out our project in the Codeblocks IDE and DEV C++ IDE after linking SFML library in them.

Computer games have become an indissoluble part of our modern daily life. They are source of entertainment and can even generate some problem solving skills among players and game developers. Chess, being one of globally played indoor mind game helps in sharpening and broadening of our brain. Chess is a two-player strategy board game played on a chessboard, a checkered game board with 64 squares arranged in an 8×8 grid. The game is played by millions of people worldwide. Each player begins with 16 pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. Each of the six piece types moves differently, with the most powerful being the queen and the least powerful the pawn. The objective is to checkmate the opponent's king by placing it under an inescapable threat of capture. To this end, a player's pieces are used to attack and capture the opponent's pieces, while supporting each other. In addition to checkmate, the game can be won by voluntary resignation of the opponent, which typically occurs when too much material is lost or checkmate appears inevitable. Hence our project is to design the chess game which can be played with the computer itself, i.e., Artificially intelligent (AI) chess game.

7. Application And Scopes

In this modern world of technology, artificial intelligence has taken vital role in the lives of present generation. Also the entertainment is not the exception. Hence briefly our project focuses on the entertainment and gaming which is based on the artificial intelligence. Initial experimentation with the game of chess(de Groot, 1965) lead to heavy expectation in the field of artificial intelligence and our project being based on artificial intelligence has a larger scope and application in present and in upcoming future. The problems associated with achieving a general artificial intelligence have generally fallen into two categories: computational power and training models.

Today, modern computing power and the advent of the internet , which can provide access to massive amounts of information to any machine learning algorithm, are eroding the first restriction. Research into the use this new power is being actively explored. Even at present various AI technologies such as SIRI and CORTANA have been developed. And that day is not so far when AI will be governing the world hence the idea of this project will obviously have a better scope in the future. In this project we have aimed to use various algorithm such as MinMax algorithm, alpha beta pruning and others. MinMax is the heart of almost every computer board game. It applies to all the games where players take turns, underlying assumption and have perfect information. But in the upcoming future rule based AI is not sufficient to model human intelligence.

The another part of our project is Graphics. Computer Graphics is one of the major field of the Computer Programming. It is very essential in the field of gaming and other entertaining sources. In this project we have decided to aim to design 2D graphics. But its scope is vast in the upcoming future. Holography and 3D modeling are various sectors where use of graphics are extremely essential. Hence concept of our project has various application and scope in present and future.

8. Literature Survey

8.1. Brief introduction to C++ and its features

The C++ programming language was created by Bjarne Stroustrup and his team at Bell Laboratories (AT&T, USA) to help implement projects in an object-oriented and efficient way. C++ is not a purely object-oriented language but a hybrid that contains the functionality of the C programming language. This means that you have all the features that are available in C and other features including object oriented approach.

Here is a short description of the features of C++ as object oriented programming implemented in this project:

8.1.1. Namespace

The namespace feature in C++ allows us to specify a scope with a name, that is, namespace is a named scope. The namespace feature helps us to reduce the problem of polluting the namespace. Namespace is defined as follows:

```
namespace npn

{

int item;

void showitem(int it)

{

cout<<it;

}

}
```

8.1.2. Classes and Objects

The **classes** are the most important feature of **C++** that leads to **Object Oriented programming. Class** is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating instance of that **class**, i.e., it helps in abstraction and encapsulation of data. It is a specification for any number of objects based on that class. Class itself is a template (data type) for objects. When we actually define the object, of a class we are requesting to allocate memory for that object to hold its data.

An object is a variable of a user defined data type class, also referred to as an instance of the class. When an object is created, memory is allocated to the data members and initialized with suitable values. Object data is usually private: so that no functions other than those within the same object can access it. On the other hand, the functions within the object are usually public. They can be accessed by the other functions in the program including the functions in the other objects. So the function in one object can access the data in the other object through the function of that object.

8.1.3. Overloaded function

Overloaded function appears to perform different operation depending on the kind of data sent to it. It performs one operation on one kind of data but another operation on a different kind. The reason behind using the overloaded function is because of its convenience to use the same function name for different operation, i.e., function overloading.

For example:

```
void convert();              //takes no argument

void convert(int n);         //takes one argument of type int

void convert(float,int);     //takes two argument of type float and int
```

The number of arguments, and their data types, are used to distinguish one function from another.

8.1.4. Operator Overloading

Using the same operator such as +, -, *,/ , etc on different scenario in the same program can be termed as operator overloading. We have already studied that we can make user defined data types behave in much the same way as the built-in types. C++ also permits us to use operators with user defined types in the same way as they are applied to the basic types. We need to represent different data items by objects and operations on those objects by operators. Operator can be overloaded for those different operations. Most programmers implicitly use overloaded operators regularly. For example, the addition operator (+) operates quite differently on integers, float and doubles and other built-in types because operator (+) has been overloaded in the C++ language itself. Operators are overloaded by writing a function definition as you normally would, except that the function name now becomes the keyword operator followed by the symbol for the operator being overloaded. Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators for your class.

Syntax

```
<return type> operator <operator_symbol> ( <parameters> )
{
<statements>;
}
```

8.1.5. Inheritance

Inheritance is a process of organizing information in a hierarchial form. Through inheritance we can create new classes, called derived classes, from existing class called base class. Inheritance is the concept by which the properties of one entity are made available to another. It allows new classes to be built from older and less specialized classes instead of being rewritten from scratch. The class that inherits properties and functions is called the subclass or the derived class and the class from which they are inherited is called the super class or the base class. The derived class inherits all the properties of the base class and can add properties and refinements of its own. The base class remains unchanged. It provides reusability of code.

An AI Chess game requires a lot of things to be cared about for best algorithm generation and move generation. Some of the things used in this project regarding chess game for best move generation and algorithm development are described as follows:

8.2. Negamax Search

This is the fundamental structure around which the rest of chess tree searching algorithms are based. To put it simply, Negamax tree searching implements the idea that "The worse your opponent's best reply is, the better your move." Implementing this is surprisingly easy. It uses the fact that chess is a symmetric game, and that therefore the analysis function must give symmetric scoring. That is to say that at any point, the score for white is exactly minus the score for black, or equivalently the sum of the two scores always equals zero. This is quite straightforward to understand. If white is winning by one pawn, then clearly black is losing by the same amount. This principal can be extended to positional advantages, i.e. if white has doubled rooks on one file, then white has a bonus score, whereas black's position is weaker by the same amount because of this.

Negamax simply implements the following rough search steps;

```
Loop through all moves
Play move
move_score = - Opponent's_best_move_score
if ( move_score > best_move_score ) then ( best_move = move )
Undo Move
End of Loop
```

As you can see, this is a recursive algorithm. To calculate the score for the first player's move, you must calculate the best of his opponent's moves. For each of his opponent's moves, you must calculate the best score for the replies to those moves, and so on. Clearly we need to define some sort of cutoff depth, or else this will continue forever. To do this, we implement the following slightly-refined version of the above code;

```
(Set depth initially to required value)
SEARCHING_FUNCTION {
Decrease depth by 1
```

```
Loop through all moves
Play move
if ( depth = 0 ) move_score = static_position_score
else move_score = - Opponent's_best_move_score
if ( move_score > best_move_score ) then ( best_move = move )
Undo Move
End of Loop
Return best_move_score
END
```

Here, the static position score routine simply returns a score for the current position based on various considerations. We can imagine a simple one which simply adds up the points values for all the current side's pieces, and subtracts the points values for the opponent's pieces. Clearly this will be very fast, but will play extremely bad chess because two positions can be very much better or worse for one player, but with the same material scores. As a simple example, imagine the endgame position where white has a king on h1 and a pawn on h2, black has a king on a3 and a pawn on a2. Clearly this is won for black, regardless of who is to play next. However, they both have exactly the same material.

8.3. Analysis Function

Modern chess engines use a complicated set of analysis functions depending on what material is on the board. I have several separate endgame analysis functions for certain types of endgame such as kings and pawns only. The main analysis routine, however, implements the following simple ideas;

Firstly, loop through the board, and make up an array of attack and defence values for each square. This is a list of how many times each square is attacked by an opponent piece or defended by a friendly one. Also keep a list of how many pawns are on each file for future reference. Next, loop through the board again, but this time do a much deeper analysis. Award points for how much empty squares are attacked and defended. Analyse each piece on the board separately by considering factors which are important for that particular type of piece. For example, reward knights for occupying squares near the centre of the board, and reward kings for staying out of trouble in the corners. Also add on a material score for each piece you own on the scale of a pawn=100 points. In ColChess I used variable piece values depending on the stage of the game, i.e., bishops become more valuable in open endgames. Finally, add on a load of other important general factors, for example reward pawn chains, bishop pairs, connected rooks and control of the board. Penalise doubled or tripled pawns and also blocked or 'hung' pieces (that is, pieces which are attacked and not defended). I do checkmate testing elsewhere, so that is not included in the static analysis function. Specialised endgame functions have different weightings, for example passed pawns become much more dangerous when there are no opposing pieces around to stop them from promoting.

8.4. Iterative Deepening

Often in games the computer player will be forced to search for a certain time, rather than to a fixed depth. Iterative deepening allows it to do this, but has many other important bonuses connected with the transposition table. The theory is simply this;

Start at a shallow depth, say 2 ply. That means that you search each of your moves, and for each of them you find your opponent's best reply and immediately return that static score. Once you have searched the tree to depth 2 ply, then increase the depth to 3 and search again. Continue doing this until you've searched to the minimum required depth and either (a) you've run out of time, or (b) you have also reached the maximum allowed depth. The first obvious advantage of this method is that you will always have some result to show from your search, and you know that it won't be a total mistake, at least to a few ply. Imagine a program which guesses the search depth to work to based on the time allocation and the board complexity, and then searches the first thirty of 31 available moves at a particular game stage. Then the program runs out of time and is forced to return the best move found so far. These 30 moves might all be complete blunders, immediately losing a piece or worse. However, the 31st move might be the only one to save his pieces, and win the game. Using iterative deepening, you know that you have a good foundation to build on. A 2 ply search takes no time at all, usually a tiny fraction of a second on fast computers. Therefore you know that when you search to depth 3 ply, you already have a good idea of which move might end up being the best. Furthermore, you can use all of the information derived in previous shallower searches to speed up vastly the subsequent deeper searches. Counter-intuitively, searching all the shallower depths first normally reduces the time for the overall search compared to just starting at a deep level initially.

This can be seen easily with the following example. Imagine you are planning to search a position to depth 10, say, and you start off iterative deepening at 2 ply, then 3 ply etc. until you reach 6 ply where you spot that all but one of the possible moves loses to checkmate! Clearly there is no point fully searching this one remaining move to 10 ply as you have no choice but to play it! You simply quit the search there and return the only safe move. Of course this move may also lose to checkmate in more than 6 ply, but that's no worse than any of the other move options and it makes it less likely that your opponent has spotted the win.

8.5. Alpha-Beta Pruning

Alpha-beta pruning is a technique for enormously reducing the size of your game tree. Currently using the negamax algorithm we are searching every reply to every move in the tree. In the average chess position there are about 30 legal moves, and let's say for sake of argument that our program analyses 50,000 moves per second. Now, let's see how deep we can search.

You can see that the search tree quickly gets extremely large, and that's assuming a rather fast analysis function. Searching to 6 or 7 ply in the middlegame is vital for a good chess program playing blitz. The total time limit might be 5 minutes, perhaps giving 5-10 seconds per move. Using our current method we can't even search to ply 4. What we need to do is reduce the tree size enormously. This is where alpha-beta pruning comes in.Alpha-beta pruning is just a complicated-sounding name for "Don't check moves which cannot possibly have an effect on the outcome of your search." The theory is essentially simple, but takes a bit of thought before you get used to it. Imagine the following scenario;

Your chess program is busy searching all its moves at the top level. So far it has searched the first six moves, and the best score has been 15 points. It starts searching the seventh move and considers its opponent's replies. Remember that your score is minus your opponent's best scores. The program steps through all of its opponent's replies to this one move, getting the following scores;

-20,-22,-15,-16,-11,-18,-20,-30,-70,-75

Now the best amongst these is -11, which gives your program a score for its move of -(-11) which is 11. This is worse than the previous best move at this level so this particular move is ignored. But we have wasted a lot of time here searching the 6th to 10th of the opponent's replies. As soon as one of the opponent's replies scored -11, we knew that this move of ours could not possibly beat the best score so far of 15. Whatever the rest of the replies scored, the best reply was guaranteed to score at least -11, meaning that our move would score at most 11 and therefore that we would disregard it. So what we should have done here was to quit the search as soon as the score of -11 was discovered, and stop wasting our time.

This is the principle of alpha-beta pruning. Its implication to negamax search goes something like this;

```
initially alpha = -INFINITY, beta=INFINITY
search(position,side,depth,alpha,beta) {
best_score = -INFINITY
for each move {
do_move(position, move)
if ( depth is 0 ) move_score = static_score(position, side)
else move_score = - search(position, opponent side, depth-1, -beta, -alpha)
undo_move(position,move)
if ( move_score > best_score ) best_score = move_score
if ( best_score > alpha ) alpha = best_score
if ( alpha >= beta ) return alpha
}
return best_score
}
```

Apologies for moving into pseudo-c but this algorithm requires that the correct variables are passed to the next level and it would be impossible to show it in any other way. I hope that all the symbols are obvious. Whenever we are searching we check to see if the move we have is greater than alpha. If it is then we replace alpha by the new score. This way alpha keeps track of the best score so far. If a move scores less than alpha then we're not interested in it because it's not good enough to worsen the score of the move to which it is a reply. If the score is between alpha and beta then it will reduce the score of the opponent's previous move, but not enough to make the same previous move bad. If a reply move scores equal to or greater than beta then this move is so good that the opponent's last move becomes bad, and the opponent would have never played it and let this situation arise, so we need search no more. We return this value of the best score immediately. This is called a fail-high cutoff. Alpha-beta search effectively reduces the branching factor at each node from 30-40 to about 7 or 8 provided that you have a reasonably good initial move ordering routine. It is clearly advantageous to have the best moves near the top of the list so that all the other moves are much more likely to cause cutoffs early. Even basic maths can tell you that such a reduction in the branching factor will almost double the depth to which you can search in a fixed time. Alpha-beta search is used by all good chess programs. It doesn't change the outcome on a fixed depth search one bit as all the branches it prunes out are irrelevant and wouldn't have altered the move score. Moreover, with twice the search depth you can get vastly better moves and play much stronger chess.

8.6. Principal Variation Search

This is a further refinement on negamax search with alpha-beta cutoffs. The idea behind it is rather simple, but it is a little tricky to implement because of a few annoying subtleties. Basically, one expects to have a good move ordering function which takes the list of all legal moves at each position, and naively puts the moves which are likely to prove to be the best near the top. One usually considers captures first, followed by pawn pushes and checks, moves towards the centre, and then the rest. This tends to put the best move at or near the top of the list in the majority of cases.So assuming that your preliminary move ordering is good, it is unnecessary to search the rest of the moves quite so thoroughly as the first. You don't need to know what they score exactly, you just need to make sure that they aren't better than the best move so far. To this end, we just search the first move properly, and then for each subsequent move we do a search with a narrow window of 1 point using the following code;

```
move_score = - search(position, opponent side, depth-1, -alpha-1, -alpha)
```

instead of ...

```
move_score = - search(position, opponent side, depth-1, -beta, -alpha)
```

If we were correct in our assumption, and the best move really was at the front of the list, then this search will return an approximate value much faster than doing a full search. However, if we were wrong then it will exit, hopefully quickly, with a fail high beta cutoff, indicating that this move will actually improve alpha, and that it therefore needs to be searched properly. (Remember that 'improving alpha' means that this move is going to beat the best score so far)

The expectation is that the gain in time caused by carrying out so many searches with narrow search bounds will vastly offset the penalty of occasionally having to search again after a cutoff. As with all alpha-beta methods, the better the preliminary move ordering, the more efficiently the pruning works, and the faster your program runs.

9. Existing System

Chess is one of the greatest challenging mind games. It is good fit for computers. It has clearly defined rules, and is game of complete information and complexity. The existing system depends upon its long history. In 1950, Claude Shannon did programming in computer for playing Chess, but the first computer program that could play a complete chess game was made in 1958. The chess game uses search trees and evaluates the best move or position and tries to bring the possible maximum output. Depth first search is performed and leaf nodes are evaluated and this branching carries on. The branching factor of typical chess position is 40. The complexity of searching d ply ahead is O(b*b*….*b). With branching factor (b) of 40 it is crucial to be able to prune the search tree. PC programs today can compute 14-17 ply ahead, for example Deep Blue computed 12 ply against Kasparov in 1997, Hydra (64 nodes with FPGAs) computed at least 18 ply.

10. Methodology

As mentioned before this project was developed by four students of Electronics and Communication Engineering as a part of a course of B.E. related to the subject Object Oriented Programming (C++) in the third Semester. This Project was assigned to us about two months ago so total time duration to develop this project was approximately 2 months. First of all we four members of the project discussed what kind of concept would be appropriate for the project. We did a lot of brainstorming. Many ideas came across during brainstorming such as making Rubix's cube solver, etc. At last we finalized that we would be doing Chess game with Artificial Intelligence algorithm to check for best possible move for our project. We planned to use various AI algorithms and graphics library to develop this project. Since there were lot of tasks and we didn't have a lot of time so the work load was divided among four of us. First of all we started researching and collecting knowledge about various concept of the Object Oriented Programming. As our course is related to C++ Programming we gathered the knowledge of various syntax and the features of C++. Then according to our work load we divided this project into different parts as algorithm, graphics, animation, designing and assigned each of the member a task as mentioned. According to the task each member researched their section and started the coding process. One of us started to learn about various Artificial Intelligence Algorithms like MinMax Theorem, Alpha Beta Puring, depth learning etc., whereas next member started to research the perfect graphics library. According to the amount of time SFML Library was decided to be used as a platform for Graphics and Animation part. After various problems encountered and using our knowledge and skills and concepts of Object Oriented Programming in C++ we finally completed our project. The problem encountered and how we tackled them in different parts of the project are described in problems faced and solutions topic later on.

11. Implementation

We were trying to implement a graphical chess game where computer will play with the user intelligently. Our motive is to make computer think against human thought to make challenge among the best chess player generating the possible moves and deciding the best move in the constraints of the game. This game uses artificial intelligence (AI) which is the field of interest for all four of us and for others too. While introducing the project titles, our friends introduced the chess project as a challenging and difficult which made us choose this project.

11.1. Block diagram

Every complex project can be described in simplest way with the help of block diagram. Block diagram gives the gist of the whole project. The block diagram of our project can be diagrammatically represented as follows:-

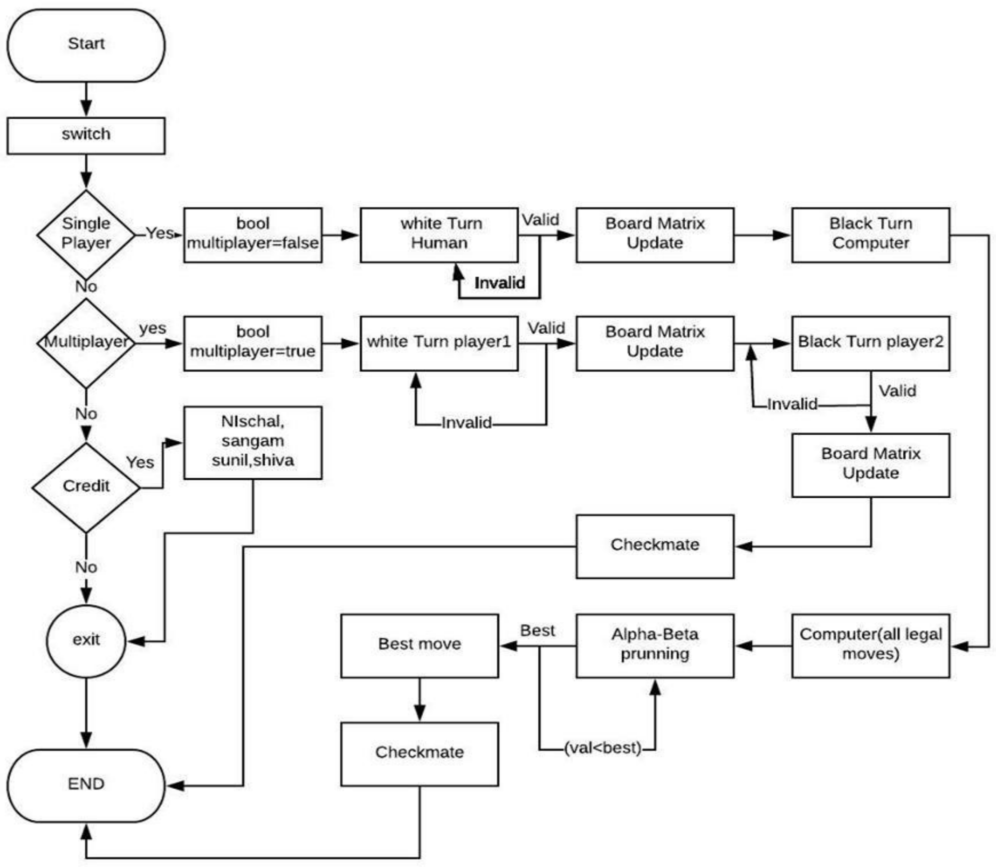


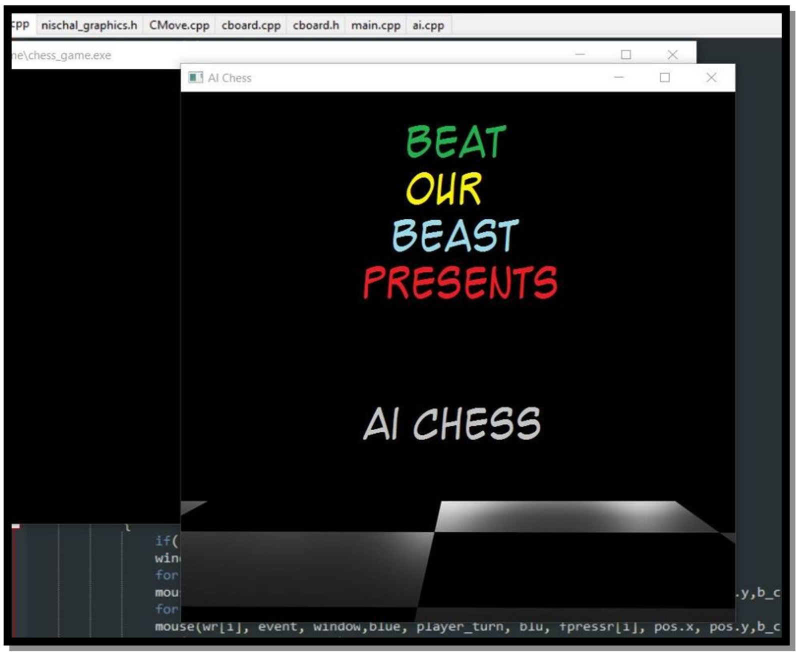
Fig:Flow chart

Figure 1: Block diagram of AI chess game

12. Results

Here, first of all AI chess game would take input from the user/s (player/s) whether to play as single player mode or multiplayer mode. In case, if there is single player then black pieces becomes computer and the player (user) with the white piece will start the game. When user's turn comes, the computer asks to decide the move and checks whether the move is valid or not. If not, computer asks the user again to decide his/her move. In this way, the game proceeds.

The output of the source code can be described better with the following diagrams and short descriptions as follows:



```
if(
win
for
mou                                                            .y,b_c
for
mouse(wr[i], event, window,blue, player_turn, blu, fpressr[i], pos.x, pos.y,b_c
```

Figure 2: First window that appears

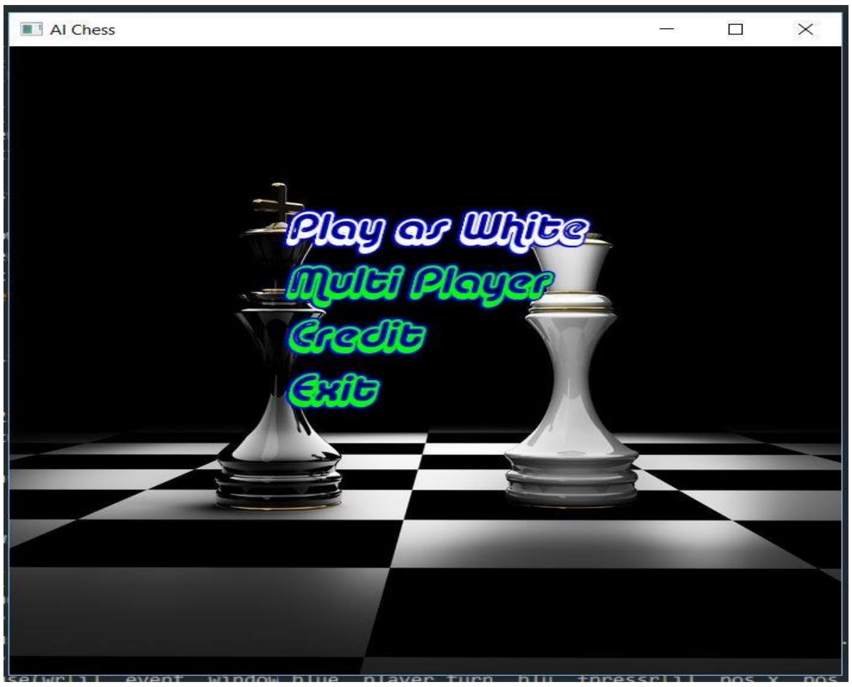


Figure 3: Main menu of game

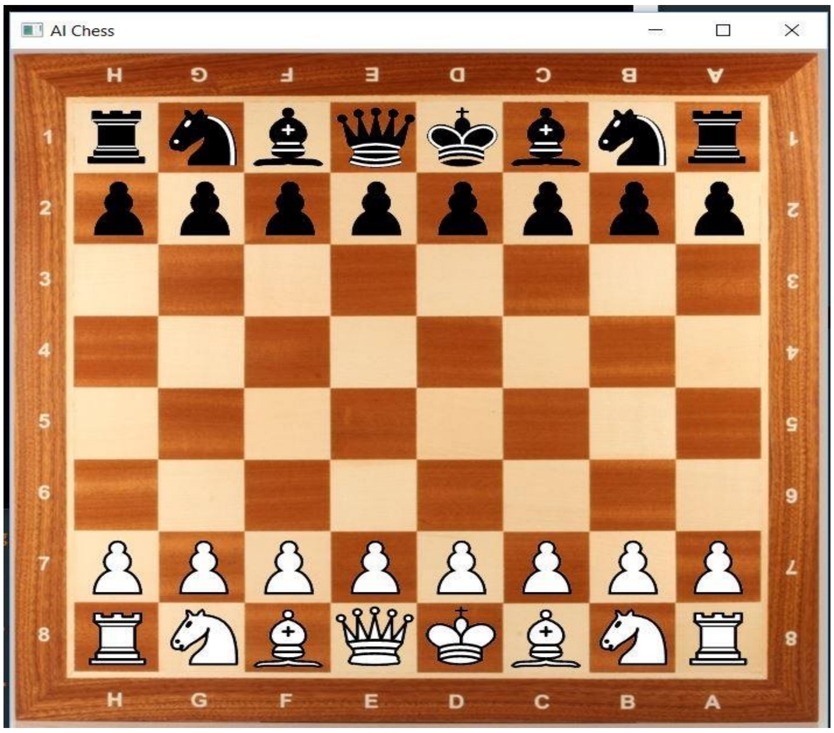
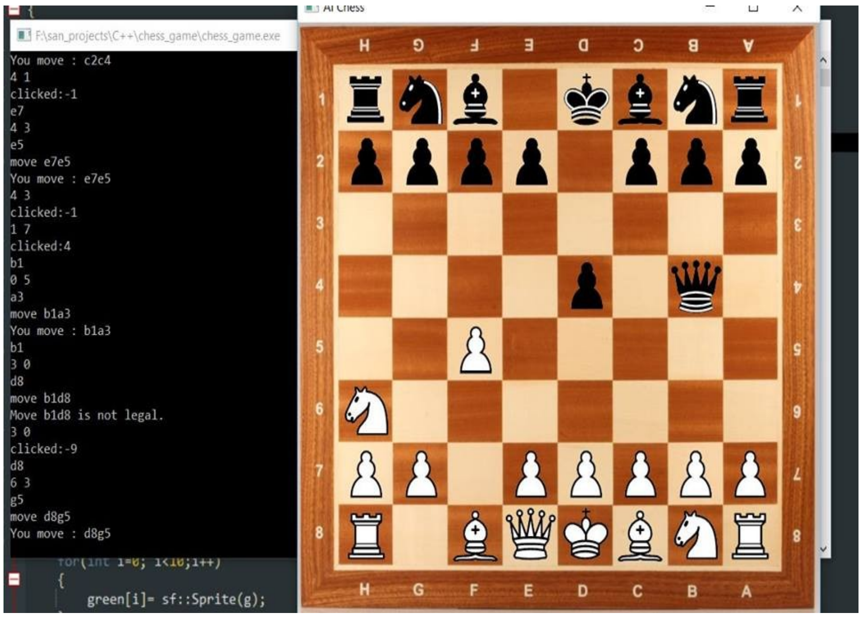


Figure 4: 8*8 Chessboard with black and white pieces


F:\san_projects\C++\chess_game\chess_game.exe

```
You move : c2c4
4 1
clicked:-1
e7
4 3
e5
move e7e5
You move : e7e5
4 3
clicked:-1
1 7
clicked:4
b1
0 5
a3
move b1a3
You move : b1a3
b1
3 0
d8
move b1d8
Move b1d8 is not legal.
3 0
clicked:-9
d8
6 3
g5
move d8g5
You move : d8g5
```

```
    for(int i=0; i<10;i++)
    {
        green[i]= sf::Sprite(g);
```

Figure 5: Possible moves generation and best move selection by AI

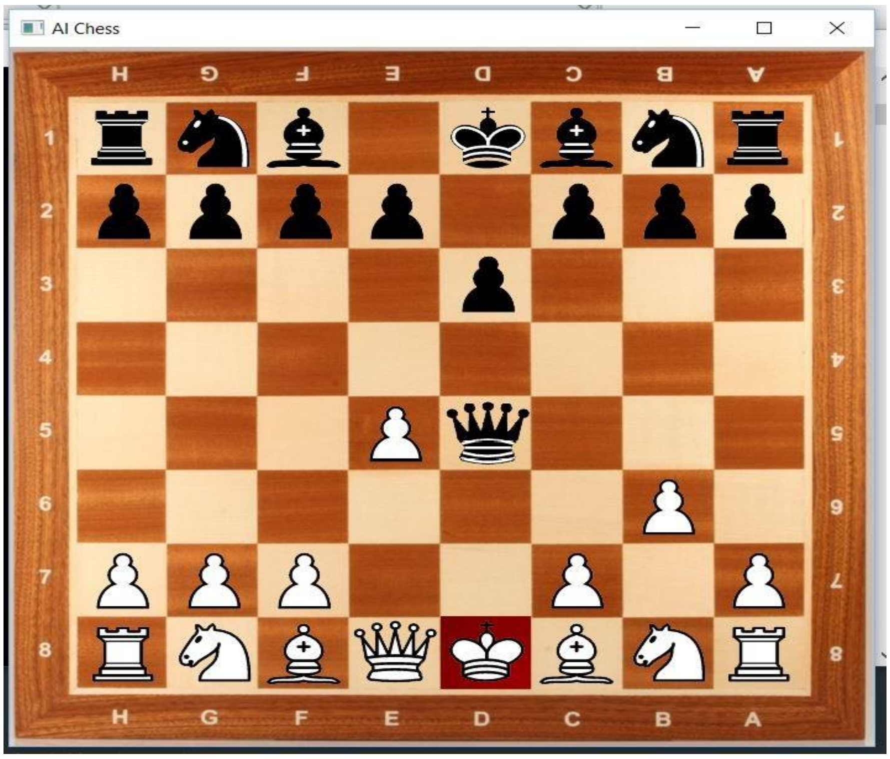


Figure 6: Check given to white king

In this way the sequence of outputs are generated as above for this AI chess game. This chess game uses the graphical interface library, i.e., SFML library which gives extra features in chess game with realistic looks while playing game.

13. Problems Faced and Solutions

During development of this project lots of problems were encountered. The biggest problem was to combine the fragments of projects which were divided among the members of the project. Due to difference in the point of view while combining all the modules such as algorithm, animations and graphics there were a lot of error encountered. Anyhow by teamwork we were able to debug the errors and finally the project was completed.

Another problem encountered was in the **graphics section**. First of all linking the SFML library (Simple and Fast Multimedia Library) to the project was the first problem we encountered. It was a really tedious and time consuming thing. By the help of internet tutorials regarding the same we overcame this problem.

Since we were using functions declared in the SFML library we were sometimes unaware about how the functions works hence due to some misconception some of invalid and function performing some different objectives were used to carry out some other task which created a lot of problem. Also another problem encountered was that the SFML was unresponsive when the algorithm tried to take a bit long time or when memory used was a bit more. The library contained various functions, we were unaware about which could have solved the problem easily instead of tedious processes and concepts which consumed our time and effort which could have been used in improving the project. These were problems that were encountered in the graphics portion.

14. Limitations and Future Enhancement

As we are just the beginners in artificial intelligence (AI), and AI has wide application in chess game so we came across many limitations. Despite the project was a success in regard of good algorithm of chess game as alpha beta pruning, negamax search, analysis functions, iterative deepening and principal variation search were used but it has some of the things missing due to limitation of time and resources in that regard. It takes a bit longer time for searching, its best move since searching is done by the CPU but if we used GPU and more advance technique to cut the unwanted branch, we can increase depth of game in short time. Some of the topics regarding chess game that could not be covered due to limitation of time chave been discussed here:

14.1. Quiescence Search

The problem with abruptly stopping a search at a fixed depth is something called the 'horizon effect'. It might be that you have just captured an opponent's pawn at depth 0, then you return that score being justifiably proud. However, if you had searched another ply deeper you would have seen that the opponent could recapture your queen! To get round this problem, ColChess, and Beowulf (like all good chess programs) implement a method called 'quiescence searching'. As the name suggests, this involves searching past the terminal search nodes (depth of 0) and testing all the non-quiescent or 'violent' moves until the situation becomes calm. This enables programs to detect long capture sequences and calculate whether or not they are worth initiating.

There are several problems with this method which need to be overcome. Firstly, it could indeed cause an explosion in the size of the game tree if used unwisely. Beowulf uses a rather simple implementation of this algorithm. However, ColChess has three levels of quiescence searching, all with different limits to overcome this problem;

1. Full width quiescence search
2. Reduced width quiescence search
3. Capture search

Full width search is not much different to the original search at depth>0, generating all the possible available moves and testing to see which one is the best. Clearly this is slow so it is used only in exceptional circumstances. By default this is not performed at all, but with the use of quiescence width extensions it occasionally comes into play. Generally it is only used in extremely dangerous situations where the side to move is in check and may well lose out because of this. Reduced width search is carried out at only a shallow quiescence depth and the exact length of this depends on how dangerous the position is. Generally it is only 0 or 1 ply, but occasionally 2 or 3 ply beneath the terminal leaf nodes (depth = 0). In this search regime I consider only a subset of all the available moves, namely those moves which have potential to drastically change the current static score. Those are captures, checks and pawn advances. If ColChess comes to do a reduced-width quiescence search at a particular node and it finds that it is in check then it does a full-width search instead. A capture search is performed after the maximum specified reduced-width search. Just as the name suggests, this involves testing only the capture moves at every node. Because this is much faster, and very important to do, I let this search continue until each branch is quiescent and there are no more captures possible. That is to say this is an infinite depth search, though of course there are only a finite number of pieces to be captured!

Of course there is another very important problem with quiescence searching. It is often more advisable for a player not to initiate a capture line because that player will lose out in the long run by doing so. In this case, at every node I give the computer the option of 'not moving' and just accepting the static move evaluation at that point. If this turns out to be better than any of the capturing move options then it will return that value. Often the best move is a quiescent one and forcing the computer to make a violent move might severely worsen its position.

The only exception to this is of course when I do a full-width quiescence search. Because this involves considering all of the possible moves, I do not allow the computer the possibility of accepting the current static score. This means that I can catch slightly deeper checkmates if they are forced because the quiescence search considers extremely narrow, but dangerous lines.

14.2. Quiescence Width Extensions

I have already mentioned this method in a small amount of detail, but it is worth clarifying. This is simply a way of improving the accuracy of the quiescence search when you think that it is necessary. Clearly there is little point in performing a full width search in all conditions, but if one of the kings is under heavy attack, it might well be worth it. I keep track of a variable at each ply which tallies up the total number of extensions so far. This is set to zero initially. Any form of check at any point in the move analysis sets it to one. If at any point one of the sides has only one legal move in any position then the extension value is increased (to a maximum of 2 ply) and similarly if a side is in check and avoids it by moving its king. I always test positions that are in the principal variation with a depth of 3 ply. This algorithm, amongst others, helped ColChess to make two large jumps in test score results between versions 5.4, 5.5 and 5.6.

14.3. Bitboards

Most top chess programs these days use bitboards. Bitboards are a wonderful method of speeding up various operations on chessboards by representing the board by a set of 64 bit numbers. For those of you who understand binary notation and bitwise operations then skip the next few paragraphs. Beowulf has an advanced rotated bitboard move generator. Binary notation is a simple way of representing numbers using only the digits 0 and 1. In conventional decimal notation, you write numbers like 45,386 where each place represents a certain number of 10's or 100's or units etc... Binary representation instead uses a 'base' of 2 instead of 10. That means that instead of representing the number of 1's, 10's, 100's etc. we keep count of the number of 1's, 2's, 4's, 8's and so on, doubling each time. A little bit of thought shows that it is possible to represent each number uniquely using this method, and that every whole number can be represented this way.

For example, the number 25 can be converted into binary by taking the largest power of 2 which is smaller than or equal to it, keeping a record that this number is included, and then continuing with the remainder after the largest power of two has been subtracted. With the number 25, the largest power of two not greater than it is 16 (2*2*2*2). We keep track of this, and then look at the remainder, 25-16 = 9. The largest power of two not larger than 9 is 8 (2*2*2). We count this too. However, we are now left with only 9-8=1. This is smaller than the next two smallest powers of 2 (4 and 2), so we place two zeros there. All we are left with is 1, leaving us with the final answer 25d = 11001b, using the subscripts 'd' and 'b' for decimal and binary respectively. Each of these '0's and '1's is called a 'bit'. Hence a 5 bit number can be anything from 00000b to 11111b, or 0 to 31 in decimal. Bitwise operators act on numbers in binary notation. The simplest one is NOT, which does exactly what it says. For each '1', it replaces it with a '0', and vice versa. Hence, NOT(1b) = 0b, NOT(10b) = 01b. We can also define operators that act on two numbers and produce a third, rather like plus and minus. One of these is AND, where 'x AND y' returns a number which has a '1' in every place where there is a '1' in both x and y, and '0's elsewhere. Hence, 11001b AND 10010b = 10000b. 'OR' is similar, where it simply places a '0' if there is a '0' at that location in both input numbers, or a '1' otherwise. Hence, 11001b OR 11010b = 11011b. Bitboard notation uses these above ideas in a rather clever way. Supposing, for example, we generate a 64 bit number which represents the pawn structure on the chess board. We do this by starting at a8, and moving across then down in the sense a8,b8,c8,...,h8,a7,b7,c7,...,g1,h1. For each square, we put a '1' if there is a pawn, or a '0' otherwise. Simple. Then we have just one number on which we can perform all sorts of useful operations.

An example of the kind of operation we might want to perform is this;

Imagine we have a bitboard of all white pieces, one of all black pieces, and one of all white pawns. How do we generate all possible white pawn moves? In conventional notation, we cycle through the board, pick out each pawn, and then evaluate the moves directly. In bitboard notation we can do much better. Firstly we take the pawn bitboard and shift it right 8 bits. Right-shift does just what it says, moving the bits 8 places to the right, and filling the 8 most significant bits on the left with zeroes. Effectively, this is the same as a divide by 2^8, or 256. What this gives us is a list of all the places on the board that are exactly one square in front of a white pawn. Next we need to make sure that we don't push a pawn into another piece. To do this we simply take an AND of this shifted board with another board consisting of all the empty squares on the board. Either this is stored separately, or it is simply equal to ~(WHITEPIECES | BLACKPIECES), where the tilde sign represents bitwise NOT, and the | sign represents bitwise OR. We can simply add in all initial 2 square pushes by taking all pawns on Rank 2 (a2-h2), and then shifting them right 16 (two full ranks), and then checking that both the target square and the intermediate 'step-over' square are empty. The command (WHITEPAWNS & (255<<48)) gives us all of white's unmoved pawns. '<<' means 'shift left', 255 is a full row of 8 1's, and thus 255 << 48 gives us a row of 1's between a2 and h2. We can now get the legal moves by taking the AND of these pawns with ~(ALLPIECES << 8) and also ~(ALLPIECES << 16). The final target squares are the result of this final board >> 16. Hopefully this approximately makes sense. If not then write out the boards as they would appear in this bit notation. For example, we can easily write out the opening position in bitwise format like so;

ALLPIECES;

```
11111111

11111111

00000000

00000000

00000000

00000000

11111111

11111111
```

WHITEPAWNS;

```
00000000

00000000

00000000
```

00000000

00000000

00000000

11111111

00000000

etc…

Here we have to format the 64 bit number in to 8 rows of 8 bits each, like a chessboard. Moves for knights and kings are also easy to calculate. Simply generate an array of bitboards which, for each square in the board, give the available knight and king moves. For example,

KnightMoves[c2] might look like this;

00000000

00000000

00000000

00000000

01010000

10001000

00000000

10001000

It is then a simple matter of taking each knight and king, and reading off the value for KnightMoves or KingMoves at that point, and then removing those which involve an illegal capture of a friendly piece. You know how to do this by now - it's just (for our example above) KnightMoves[c2] & ~WhitePieces, assuming that this knight on c2 is a white knight. Note that so far we've not considered whether or not the moves are actually legal or whether they leave us in check. This is time consuming, so we do a 'lazy evaluation' on this. That means we work it out only when we need to.Sliding Pieces are handled rather differently. They use a technique known as rotated bitboards. Here, by using a lot of precalculation and a bit of mind-bending transformations, we can actually generate sliding moves rather quickly. We do this by calculating occupancy numbers for ranks, files and diagonals. For the rank, this is simple, and it means the following; Take the rank, and then consider all the pieces (of any colour) on that rank. Now, ignoring the rest of the board, and just considering these 8 squares, convert this into a decimal number. In bitboard notation, what we've effectively done is; occupancy = (AllPieces >> (Rank*8)) & 255; where here 'Rank' is the rank in bitboard notation. We started at a8 = square 0 and worked across first, so the rank is 0 for the top row (a8-h8) and 1 for a7-h7 etc.. up to 7 for a1-h1. This is simply equal to the value of the square number divided by 8 and rounded down. Now here is where the precalculation comes in handy. We have already generated a large array indexed by square number (0-63) and occupancy number for the rank on which this square resides (0-255). For each entry, we precalculate the bitboard representing the squares to which a horizontally sliding piece can move with the given occupancy. We simply look up this bitboard and we instantly have a bitboard representing all the horizontal targets this sliding piece has. We then filter out illegal captures of friendly pieces with a simple AND, as before.

For vertical (file) moves, we do exactly the same, but with a board which we have stored separately which stores the position of all pieces rotated through 90 degrees. This way we can simply read off the occupancy number of the file under question by reading off the occupancy of a rank in this 'rotated bitboard'. Now diagonal sliders are a lot more tricky, but effectively use the same principle as above. We store a bitboard 'rotated' through 45 degress either clockwise or anti-clockwise. We treat the two diagonal directions separately (that is diagonals in the direction a1-h8 and those in the direction a8-h1). We also have to store the length of the diagonals under question, and the amount we have to shift the board to get that diagonal at the front. For example, the a1h8 bitboard might look like this;

a8,a7,b8,a6,b7,c8,a5,b6,

c7,d8,a4,b5,c6,d7,e8,a3,

b4,c5,d6,e7,f8,a2,b3,c4,

d5,e6,f7,g8,a1,b2,c3,d4,

e5,f6,g7,h8,b1,c2,d3,e4,

f5,g6,h7,c1,d2,e3,f4,g5,

h6,d1,e2,f3,g4,h5,e1,f2,

g3,h4,f1,g2,h3,g1,h2,h1

which may not look like much, but if you actually write it out slightly differently, then you should immediately see the point;

a8,

a7,b8,

a6,b7,c8,

a5,b6,c7,d8,

a4,b5,c6,d7,e8,

a3,b4,c5,d6,e7,f8,

a2,b3,c4,d5,e6,f7,g8,

a1,b2,c3,d4,e5,f6,g7,h8,

b1,c2,d3,e4,f5,g6,h7,

c1,d2,e3,f4,g5,h6,

d1,e2,f3,g4,h5,

e1,f2,g3,h4,

f1,g2,h3,

g1,h2,

h1

Now you can see that all we have is a bitboard with the diagonals arranged sequentially. Now, for example, if we want the occupancy number for the diagonal in the a1h8 sense starting on square b4, we simply do the following;

1. Calculate the necessary shift, in this case 15.

2. Shift the board 15 squares right so that the top row now starts a3,b4,c5,d6,e7,f8.

3. Look up the length of this diagonal, in this case 6 squares.

4. Take the bitwise AND with a string of 6 1's, or 63 (=111111b) to get the occupancy.

5. Now lookup the target bitboard using the square b4 and the occupancy obtained.

6. You now have a list of possible target squares.

7. AND your list of target squares with ~WhitePieces to get the final list.

Not exactly simple, but hopefully understandable. The a8h1 diagonals are exactly the same, but with a bitboard rotated the other way. Now do you begin to see the power of bitboards? These rotated bitboards can be calculated when they are needed, but this is rather slow. It is far simpler just to update them whenever a move is played, just as you update the standard board. Once we have a bitboard with all the possible destination squares set to '1', we now have to cycle through them one by one. To do this, we require two routines. Firstly we require a routine which locates the position of the first bit in the bitboard which is set, and secondly we need a simple macro which then sets this bit to zero. This was we just run some code like the following;

```
while (moves) {
target = GetFirst(moves);
...
Do required stuff with the target square
...
RemoveBit(moves,target);
}
```

These were the topics regarding chess algorithm that could not be covered up due to certain limitations of time. Besides, we used SFML library for graphics but for more flexibility of graphics and more interactive GUI OPENGL(might be some other too) could have been used. For future enhancement, we can have OPENGL linked to our IDE for better graphics interface. Besides that, Applications of **neural networks** in chess game could have been great which are learning of evaluation and search control but due to shortage of enough time and resources we could not implement it to our project.

15. Conclusion And Recommendation

Thus, Learning from object oriented approach in C++ programming language and its features to alpha-beta pruning and many other chess game related theories for algorithm development including some SFML stuffs regarding proper GUI, we successfully accomplished this project in time. Hence we designed the AI chess game within limited time and with adequate knowledge regarding C++ and object oriented approach.

Application of Neural network to AI chess game could be far better regarding the self learning algorithm so use of Neural network Algorithm are highly recommended in this regard.

References

Balagurusamy, E. (2016). *Object Orinted Programming With C++*. New Delhi: McGraw Hill Education(India) private Limited.

Baral, D. s. (March ,2010). *The Secrets of object Oriented Programming in C++*. Bagbazar, Kathmandu: Bhundipuran Prakashan.

Chess programming wiki. (2017, August 18). Retrieved from wiki: https://chessprogramming.wikispaces.com

Free code Camp. (2017, Mar 30). Retrieved from Free code Camp: https://medium.freecodecamp.org/simple-chess-ai-step-by-step-1d55a9266977

SFML. (2017). Retrieved from SFML: https://www.sfml-dev.org/