The Chess Game

Designed by:

- Mohab Mosaad Attia
- Ahmed Mohamed Mohamed Hamdy

" It is not enough just to write code that works. It is more important--to write code well ; code that is legible, maintainable, reusable, fast, and efficient."

Acknowledgments:

We would like to express our sincere gratitude to our mentors Dr. Nagia Ghanem, Eng. Ahmed Ghazal and Eng. Omar Salah for their patient guidance, encouragements and wise supervision of the project. We would also like to thank Computer Systems and Engineering Department as well as the Faculty of Engineering for supplying the required lab periods and teaching staff.

Abstract:

This report demonstrates the basis of our chess engine and illustrates the implementation techniques and their scientific dimensions and the advantages of the used techniques over the other alternatives, following the demonstrated efficiency standards we managed to make our software core optimum in memory and time consumption as well as maintaining the code in a maintainable, editable and easily comprehended form. Using simple calculations, we have managed to take use of every possible bit to use –and we really mean it- and also our software has proved to execute 8x faster than the average runtime of the competing softwares. The report also demonstrates steps of software decomposition and optimizing the core functions to obtain the mentioned speed. The report also documents the development process and the following procedures of testing and documentation.

Software Requirements(Analysis process):

In order get an optimum output the project requirements were interpreted into software requirements specification(SRS) and functions were assigned accordingly.

Our software was cracked out into 8 main operations :

- Acquiring initial board position for game start.
- Displaying game specs as required in the project announcement.
- User input scanning.
- Input validation which on its own was cracked into:
  - Format validation.
  - Target validation.
  - Piece validation.
  - Path validation(move validation).
- Game update.
- Detecting threat to both kings.
- Detecting any possible moves for threatened kings(checkmate checks).
- Detecting stalemate for imprisoned kings.

Code Hierarchy overview and dependency distribution:

The Unified Modeling Language (UML) diagram in Fig1 explains the implemented functions dependencies such that fulfilling a certain task is dependent on another.

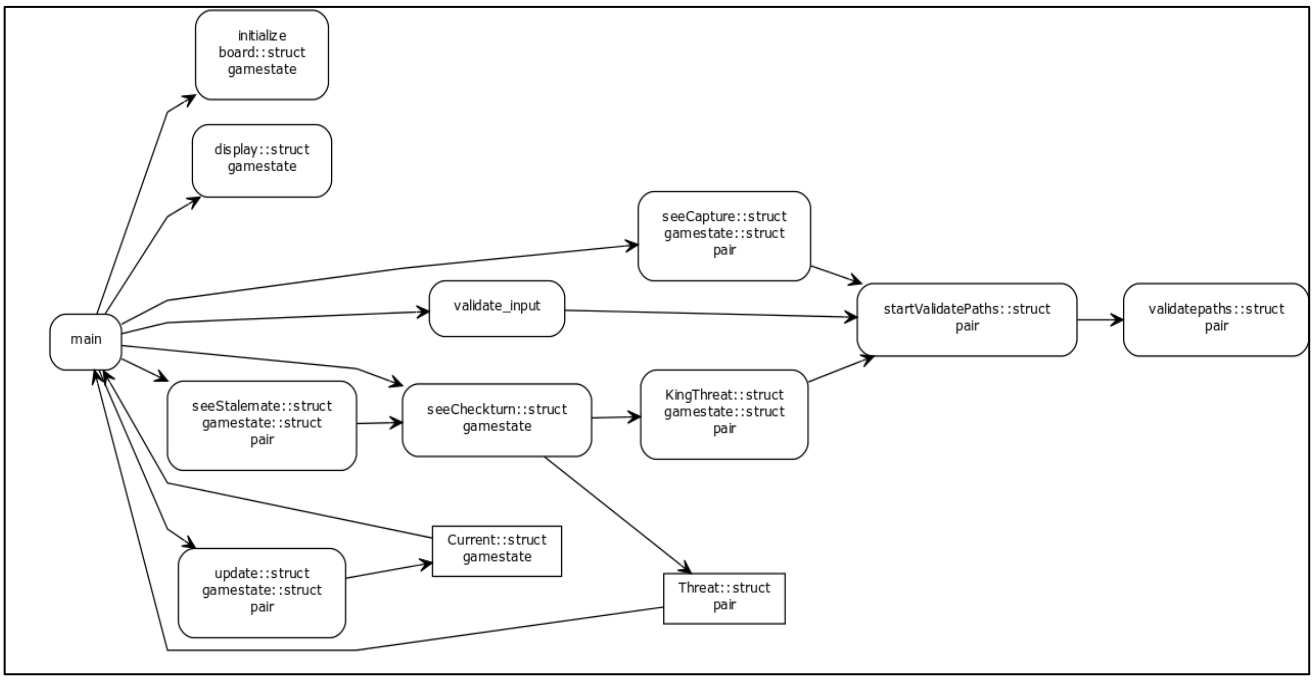

Figure1: UML Diagram

The round brackets in the diagram represent the functions pointing to the functions/structures they depend on. The square brackets resemble the structures and variables of them.

Game structures(The data holders):

In our software's layer decomposition the game structures represents the data access layer(DAL). The structure has been created to group together a set of data that assumedly represents the current game state where the game state is defined as the least information about the game needed to start over the game again as well as some data that continuously grouped throughout our code.

1. **Pair structure:** A structure presents coordinates on game board(row,column)
2. **Player Structure:** A structure that groups all positions of a player's pieces.
3. **gameState structure:**

The structure contains fields that hold:

- Board Representation as a matrix of characters.
- Board representation as an unsigned long integer (matrix of 1s and zeros). This representation is used in path validation and is the main knot of optimizing our run time(binary representation).
- An integer that represents whose turn it is.
- 2 Player structures to hold all pieces positions.
- A matrix that holds indices for pieces in player structures(used to speed up pieces access).
- Am integer representing the game indicator(game play, check, checkmate, stalemate)

```
r  h  b  q  k  b  h  r          1 1 1 1 1 1 1 1
_  p  _  p  p  p  p  p          0 1 0 1 1 1 1 1
B  _  .  _  .  _  .  _          1 0 0 0 0 0 0 0
P  .  p  .  _  .  _  Q          1 0 1 0 0 0 0 1
.  _  .  _  P  _  .  _          0 0 0 0 1 0 0 0
_  .  _  .  _  .  _  .          0 0 0 0 0 0 0 0
.  P  P  P  .  P  P  P          0 1 1 1 0 1 1 1
R  H  B  .  K  .  H  R          1 1 1 0 1 0 1 1
```

Figure2:Board representation data

The Main Algorithm (The Maestro):

In this section we are going to provide a vivid detailed explanation of our main algorithm, A flow chart is also provided for code tracing. Code pieces are used with confusing points. Any non detailed procedures are functions that are explained later.

The main algorithm performs the following tasks repetitively:

1. Initializes the game by creating the initial board state.
2. Displays current game state (Board, Taken Out pieces Game turn, game turn).
3. Scans the new user input.
4. Checks if the input is undo, redo, new, load, save, exit and executes the suitable function.
5. If the input is not an implemented feature the main passes it to the validation function that validates if it is a valid game move or not.
6. If the input was a valid game move the main passes the move and the current game state to the updating function that modifies the game board and positions according to the move.
7. After updating the game the main uses the updated game to check threat for both kings and undoes the updated move if it leaves the king of the current player threatened.
8. If the opponent king is threatened with no possible moves the main runs the capturing & blocking algorithm that checks the ability of blocking the threat or capturing the threat. If not it announces checkmate.
9. Finally if the new player's king has no possible moves and is not threatened the main runs the stalemate algorithm that indicates if it is a stalemate or not.

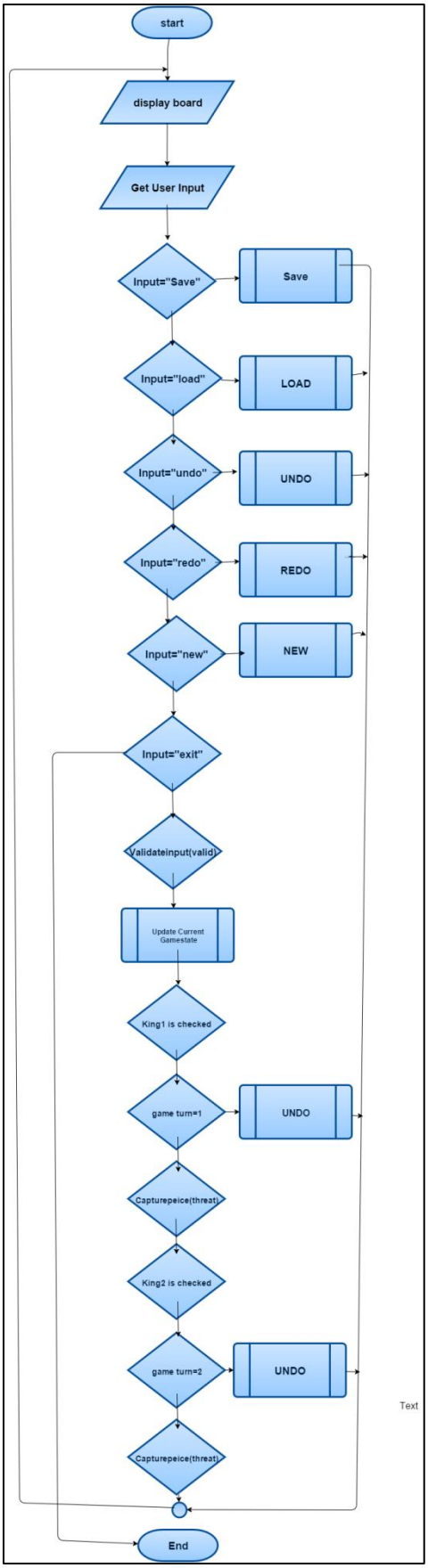

Figure3: Main Algorithm Flowchart

PS: King's Possible moves as well as being checked or not are all returned by the king check function.

Move Validation:

Unless an implemented feature, the user input passes by several steps of validation to determine a valid game move and reject any sort of invalid input such as malicious input and invalid game moves.

Steps of validation:

- Length Validation: A valid game move is either a 4 digit string or a 5 digit string -in case of promotion-any input of other length fails the first condition of validation and is not a valid input. An input of 5 characters activates the promotion validation.
- Input Transformation: After length validation the user input is transformed from the string form into pair form where the row of a piece is (8-inputrow) and the column is the difference between the indicating character and the 'A' character. Example: F6 is transformed into a pair of row=2 and column=5. The pair form is used in all upcoming validations.
- Border Validation: The initial coordinates and the target ones are validated to be inside the gaming board.
- Move Presence Validation: The initial and target coordinates are tested to be different.
- Self Validation: The target coordinates on board are tested not to contain pieces of the same kind of playing player.
- Promotion validation: if activated moving piece is tested to be pawn and the target row to be a valid promoting row.
- Piece Validation: initial position on board is tested to contain a piece of the playing player's pieces.
- Path Validation: Each valid piece activates a certain path test according to its own movement (diagonal, horizontal, vertical) and fires the validate paths function which is discussed in detail later on.

Path Validation:

Last but of course not the least the path validation function is the last checkpoint for user input validation, however it has been a core function for many other tasks such as check detection and stalemate detection(as shown in the Fig1)

The path validation decomposed into testing the suitability of the path to the moving piece and then testing the unpresence of any blocking pieces in the path.

The path in which the piece moves is represented as an unsigned long long integer. By applying some mathematical calculations on that data holder the unsigned long long int is a number that in the binary form corresponds to the path cells of the piece in the binary representation of the board.

By "Anding" the 2 64 bit integers (the board and the path) we get either

- A zero valued result indicating that the 2 integers –on the binary level- do not coincide , this means that no path included cell is a piece including cell.
- Any non zero result which indicates one or more coincidences between path included cells and piece including cells, hence path is loaded and move is invalid.

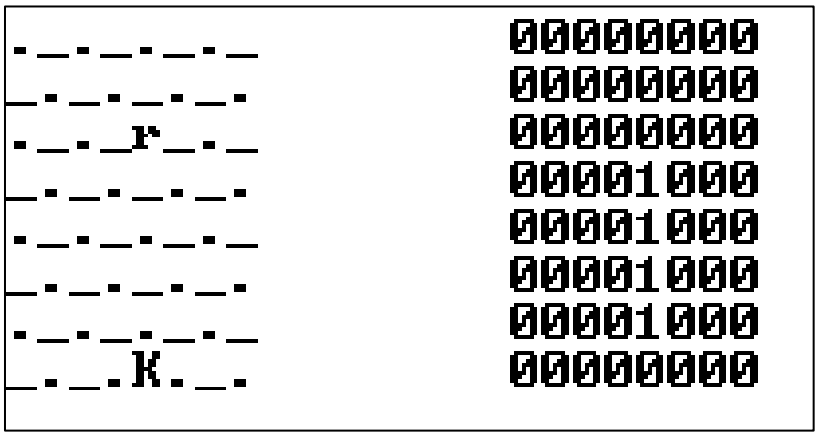

Figure4: binary path loading diagram

Let's go back to the loading process, an integer is represented by a 1in a certain bit and by adding 0 to 1<<(index of bit) or 1<<(8*row+column). By adding the same equation to several bit indexes we can load the path but of course adding them manually (bit by bit) gives us no advantage of time at all. The greatest trick is adding all these numbers in one single instruction.

By observing the required bits for loading in figure4 each bit in the pass is shifted more than the preceding bit by 8 bits. This equation can be written as

U(desired integer)=∑(1<<(b+8*n)) where b is the base row n is form 0 to row difference between target and initial position.

By using the substitution $1<<C=2^{C-1}$ * we can write our desired integer as $U=\sum 2^{b+8*n-1}$

preserving b to the base row and n from 0 to row difference. The past equation can be translated into a sum of finite number of elements in a geometric sequence whose base is 2^{b-1} and rise is 2^8 hence the result is calculated directly as $2^b(2^{8*N-1}-1)/(2^8-1)$** Where N is the row difference and this equation is a simple mathematical calculation that is executed –on reconverting the power of 2 to shift of 1-in modern computers in 1 instruction cycle.

The same Equation is to be applied on diagonal and horizontal loading with changing the rise value to 2^7 or 2^9 or 2.

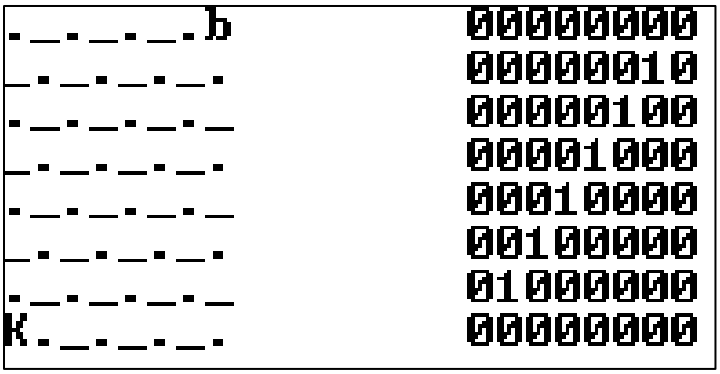

Figure5: decreasing slope binary diagonal path loading

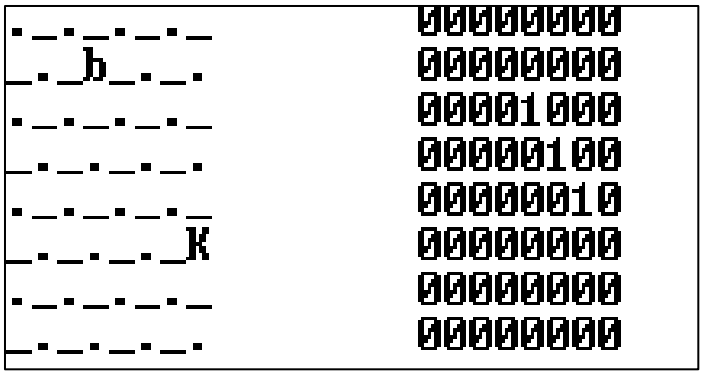

Figure6: increasing slope binary diagonal path loading

**Geometric sequence sum, *power 2 to shift 1 conversion

"Programming is just implementation to mathematical and logical solutions"

Game Update:

After passing the validation tests the successfully validate move should be executed and the game state is changed. An update function receives the pairs indicating the move and a reference to the current game state. The function tests the effect of the move execution on all the structure data (the game state indicators).This can be listed as:

- Removing the piece from its initial position to target position.
- Updating piece position in the player structure.
- Viewing the black or white cell which was hidden by the initial piece.
- Checking whether the piece took out an opponent piece or not and adding the taken out pieces to the taken array and adjusting the index.
- Updating king's position if affected by the move.
- Removing the significant bit of the initial position and adding it to the new position in the binary representation matrix,This is done by anding the old matrix to ~(1<<initial),then Oring it to (1<<final).

Checking Check:

After maintaining the correct flow the game, it is now required to maintain the application of all of its rules correctly. The first and the backbone of all chess game is the check. Check is the situation at which the king is threatened.

Our Check algorithm is a greedy algorithm; however it is optimized to reduce running time. The processing function of the algorithm is responsible actually for determining all possible positions for the king piece to be in the next move. We can also name it the neighboring cells method as it detects the ability of the king to be in any of them including his own cell. The inability of the king to stay in his own cell indicates the check situation.

Neighboring cells Algorithm:

As mentioned before, the algorithm is a greedy one. It is implemented by directly accessing each of the opponent's pieces locations stored in the game state and validating the path -as well as the move- from it to any of the 9 cells containing the king

and marking it as invalid if it was possible for the opponent to move to it. The algorithm also marks the cell invalid if it was occupied by an ally or if it was outside the borders.

Marking all the cells invalid and valid and storing those in 1 data holder needed another transition to the bit level accessing. The first 9 bits of an integer were used to hold such data where each bit resembles a certain cell of the 9 ells surrounding the king and if such bit is high(1 in value) the cell is not valid for the king to move to else it is valid. This implementation facilitates all the "Check" process and also some of the succeeding processes as it easily indicates a check situation if the king containing cell is invalid to be in, it also indicates the possibility of checkmate y determining the ability of the king to move away of threat. More bits of such integer are marked high in case of double check, This is used as a flag to directly declare checkmate if the king is trapped.

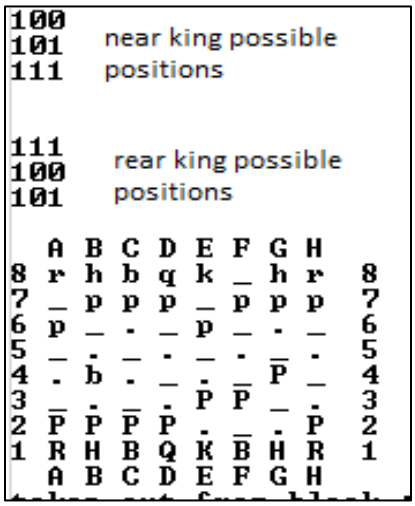

Figure7: neighboring cells representation for both kings.

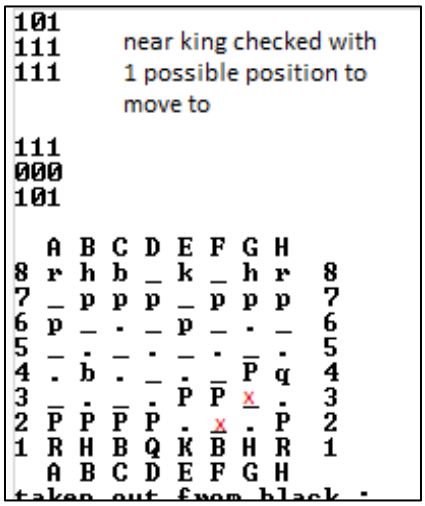

Figure8: neighboring cells indicating check.

Dependency of Neighboring cells algorithm:

As shown in the UML diagram the algorithm is dependent on the path validation algorithm as it uses it to test the ability of the opponent's pieces to reach the king and its neighbors.

Check To Checkmate:

A checked king who has the move is in a potential checkmate situation. According to chess rules the king in check can either move away to safe position, move another piece to capture the threat or move it to cover himself (block the path).

Moving the king away can simply be tested using the neighboring cell output as we already acquired all the possible moves for the king so it is only needed to check that output again. As for the threat capturing and path blocking they can be grouped into one requirement which is finding a piece that has a valid move to any path cell of the threat including the threat cell itself. This was the task of Capturing Function.

Capturing Function:

The capturing function is the final verdict of transforming a check into checkmate, the function algorithm is also a greedy one, the function at first calculates the increase in the target to reach king positions which is the path indication and then iterates on all of these cells and subsequently iterates on the current player's pieces and tests the path validation to check if the piece can move to the destinated cell.

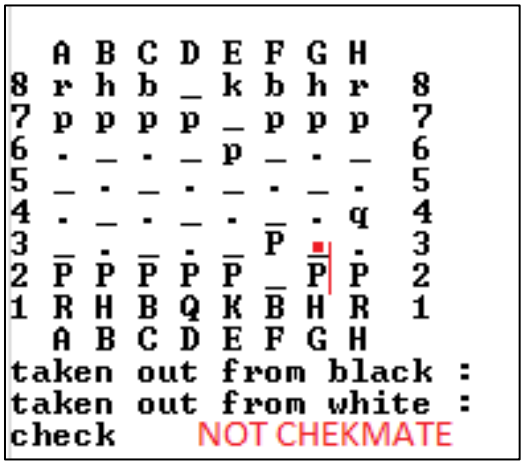

Figure9: blocking opportunity detected.

We can comment on this algorithm that it is a little bit consuming and we can see a clear optimization by determining the cell of the threat path that intersects the covering piece path this reduces the algorithm complexity to the factor of the path cells, but unfortunately the mentioned optimization failed some of the quality tests assigned by the developing team and the team chose the whole project completion over the algorithm optimization, but this would make a perfect start for the next version assessment.

Stalemate(the Draw situation):

In completion of the game rules integration the stalemate algorithm was added greedily and runs at the end of each valid move.

The stalemate situation is described in check by the state of having no legal move --for the playing player – that doesn't put the king in danger. The stalemate is also the last weapon of the weak sided player to have points of the game.

The stalemate algorithm:

The stalemate algorithm is a greedy algorithm too that iterates the turned player pieces and tries to move any / each of them the least possible moves for each piece. The piece of possible moves is removed from its place on the binary representation board then the binary representation board is sent to detect threat on king. The removed binary piece is returned on the board and if its removal was of no threat to the king stalemate situation is announced to be false. Otherwise the piece check is proceeded for the other pieces.

Dependency of the stalemate:

The stalemate function depends on the king check function so it may be considered a high layer of the code hierarchy. This was shown in the UML diagram.

Supported Features:

1. Save: The game program supports file accessing and saves the gamestate structure into it, The program saves the data as binary data to preserve time and memory.
2. Load: As well as writing the data into files the game program easily loads out the saved data from any present file. Safety features were taken into consideration where a loading file must be present and the loaded data is validated.

3. Undo and Redo: the game played can be undone to the first move and redone to the current move. This is done by pushing every updated gamestate into a stack of the structure. Popping and pushing the stack enables facile accessing to the previous and next moves. Safety features were also added to restrict undoing to the first move and redoing to the last updated move.
4. New: The game can be started over again at anytime.

User Manual:

The game starts by the initial board state at which each piece is located in its initial position. The User Interface displays the board and the taken out pieces of each player and announces whose player should play and whether it is a normal game move or if there is a check. The interface also prints"Invalid" for any user input that fails the mentioned validation tests. It also announces the end of the game at checkmate or stalemate situations and waits the demand of a new game or a loaded one.The game moves are supported by the algebraic notation method of chess where the move is the initial cell representation followed directly by the target cell and the row and column indications are printed in its correct locations. Example: A2A4 moves the demands to move the piece in cell A2 to cell A4.

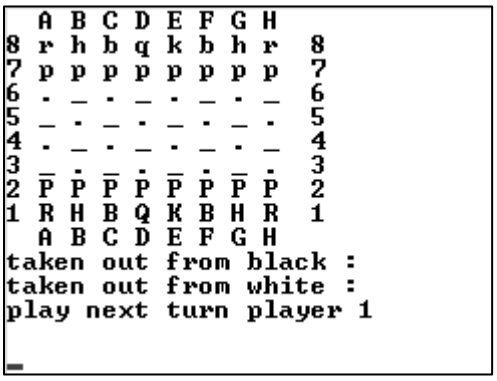

Figure10: User Interface Display

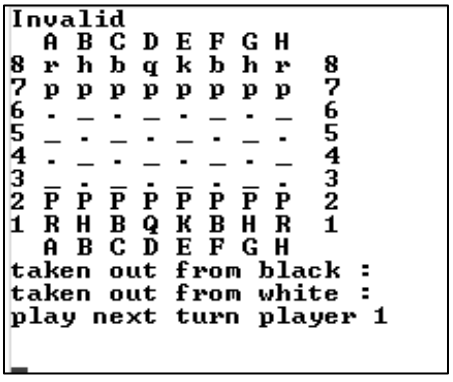

Figure11: User Interface Display

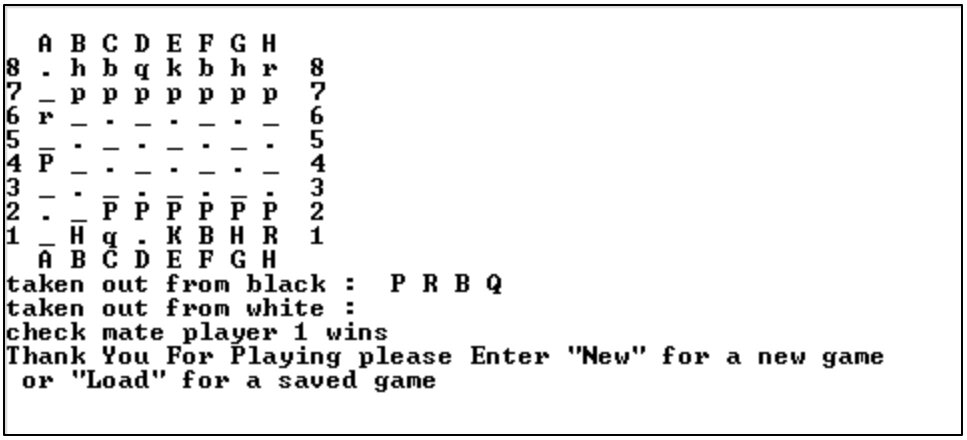

Figure12: User Interface Display

Program Complexity:

Our main advantage of the project is the optimum running time it was proven to execute more than 1000 moves (without display) in less than 1 second. The calculated complexity was also proved to be 8X faster than the average runtime of the competing programs (including online implementations). The hard coding and the high efficiency of the program did not affect its maintainability. The coding process was carried out on a scientific professional basis which is explained later on.

Future Plans:

Professionally designed software never ends up on the launching date, It is always in the developing stage. A total motab3a process is currently running on the project to detect possible optimizations some of them are already reported and we are currently working on them such as:

- Optimizing the greedy algorithms to process fewer time.
- Creating a new version of the User Interface.
- Implementing chess algorithms based on chess theory, artificial intelligence and probability theory that enables computer to play against various levels of amateur and professional chess players.

We promise to publish The Improved version of our project in a period of 2 months implementing the mentioned improvements and more features.

References:

Save: http://www.cplusplus.com/reference/cstdio/fwrite/

Load: http://www.cplusplus.com/reference/cstdio/fread/

Operators : http://www.cplusplus.com/doc/tutorial/operators/

Sequence: https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BitOp/bitshift.html

Stacks: http://www.c4learn.com/data-structure/basic-stack-concept/

Check handling : https://en.wikipedia.org/wiki/Check_(chess)#Getting_out_of_check

Stalemate definition: https://en.wikipedia.org/wiki/Stalemate

DAL: https://en.wikipedia.org/wiki/Data_access_layer

SRS: https://en.wikipedia.org/wiki/Software_requirements_specification

UML: http://www.uml.org/

Series: https://en.wikipedia.org/wiki/Geometric_progression#Geometric_series