# FINAL PROJECT

SUBMISSION DUE DATE: 08/04/2018 23:00

## INTRODUCTION

In the final project you will create an interactive **Chess** program with both a Graphical User Interface (GUI) and Artificial Intelligence (AI) components.

The Chess program has two modes:

**Console mode** – operates in a similar way to the Connect-4 implementation from HW3, with user commands. However, you are required to develop the console mode such that it is easier to later integrate with your GUI implementation, and both use the same backend logic.

**Graphical mode** – presents the user with visual menus and controls, enabling it to choose any option from within the graphical interface.

For AI, the Minimax algorithm will be used. It is mostly the same as the algorithm in HW3, except we use **pruning** to improve its efficiency. You are no longer required to construct a tree and can execute the algorithm recursively. A description of pruning is in Moodle.

A compiled Chess executable with only Console mode and limited capabilities is attached to this project. Use this executable to create input/output files and save-file examples, and to check correctness of your execution. Comparing your performance with this executable is also recommended.

## HIGH-LEVEL DESCRIPTION

The project consists of 4 parts and an optional bonus:

- Console user interface
- Minimax AI algorithm
- Graphical user interface
- Chess game logic (independent of interface)
- *(Bonus)* Concurrent implementation:
    - Minimax algorithm with improved runtime
    - Enabling user interaction (both in GUI and Console) while the minimax algorithm is executing.

The executable for the program will be named "*chessprog*". It receives a single optional command-line argument, as follows:

*./chessprog -c* – will start the program in console mode.
*./chessprog -g* – will start the program in GUI mode.
*./chessprog* – will use the default execution mode – console.

## CHESS

A brief introduction on Chess is given in this section. You can find further details in Wikipedia: https://en.wikipedia.org/wiki/Chess.

Note that for simplicity, we ignore some rules regarding the standard Chess game. Thus, you should read the following carefully even if you are already familiar with the rules.

### RULES

Each Player begins the game with 16 pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. The game board is an 8X8 grid, in which each of the pieces are in a predetermined order (look online for more information). Each of the piece types moves differently. Pieces are used to attack and capture the opponent's pieces. Capturing is performed by moving a piece to a position occupied by an opponent's piece. Besides the pawn, all pieces capture in the same direction they move.

A victory is achieved when a player captures (or kills) the opponent's king. Victory is declared before the actual move is executed. Each opponent has his own color (either black or white). As it is accepted worldwide, the white player plays first. **Further notice that the player may not move any of its pieces to a state where the king is threatened in the next round.**

The piece types

- Pawn:
  - Movement: a single move forward in the same column. The user may move the pawn two steps forward only if the pawn is located at its starting position, and the path is not blocked.
  - Capturing: one diagonal step forward
  - Console representation: m/M (lowercase is white, capital is black).
  - Do not implement pawn promotion or *en passant* moves.

- Bishop
  - Movement: can move any number of squares diagonally (backward and forward), but may not leap over other pieces.
  - Capturing: diagonally in any direction
  - Console representation: b/B

- Rook:
  - Movement: can move any number of squares along its line or column on the board, but may not leap over other pieces. (horizontal moves)
  - Console representation: r/R

- Knight:
  - Movement: "L"-shape: two squares vertically and one square horizontally, or two squares horizontally and one square vertically. The knight is the **only** piece that can leap over other pieces.
  - Console representation: n/N

- Queen:
  - Movement: combines the power of the rook and bishop and can move any number of squares along rank, file, or diagonal, but it may not leap over other pieces.
  - Console representation: q/Q

- King:
  - Movement: moves one square in any direction
  - Console representation: k/K
  - Do not implement the *castling* move.

## CHECK, MATE, AND DRAW

Check

- When the opponent threatens a player's king, it is called "check". A king is threatened if there is an opponent's piece that can capture (kill) the king in the current settings of the gameboard. A response to a check is a legal move **only** if it results in a position where the king is no longer in threat.
- The player must move one of its pieces so that the king is not threatened anymore. Other moves are illegal. If there is no legal move, see Checkmate.
- Notice that the current player may never perform any movement that will result in its king being in check state.

Checkmate

- When the king is threatened by the opponent ("check") and it king cannot be saved (there are no legal moves).
- Checkmate terminates the game (in Chess, victory is achieved before the capturing of the king is executed).

Draw

- When the player doesn't have any legal moves, but the king is not threatened by the opponent (no "check").
- A draw terminates the game.
- Note that in Chess there are more option to reach a draw. Do not implement them.

# IMPLEMENTATION DETAILS

## GAME SETTINGS

In our Chess games we will have the following settings:

- **Game mode:** there are two operating modes, 1 player or 2 players, represented by 1 and 2, respectively. In 1-player mode, the user plays against an AI opponent. In 2-player mode, the game is played with two different opponents. The default game mode value is 1.
- **Difficulty level:** in 1-player mode there are 5 difficulty levels: 1-5. These numbers represent the following levels: amateur, easy, moderate, hard, expert. The default difficulty value is 2. Note that the expert difficulty (5) is considerably heavy and pretty much unplayable unless you implement the bonus. Do not worry about this, and you should still provide the option to the user.
- **User color:** in 1-player mode the user may specify her color. The color can be either black or white, represented by 0 and 1, respectively. The default value is 1.

In both Console and Graphical mode, the user may choose these settings, as described later.

## CONSOLE MODE

When the program starts, it first prints the title " Chess\n" and then a line of 7 dash '-' characters. This is printed only **once**, first thing when the program starts.

The program may be in two states during its executing: settings state, or game state. Initially, the game starts in the settings state, where the user may change the settings of the game prior to the game itself.

In both states, whenever the user enters an invalid command or a command with wrong parameters (for which there is no specific output described below) or a wrong syntax, the program outputs: "ERROR: invalid command\n". It's OK to include this output for blank lines.

Note that it is OK if a command has extra parameters! Ignore any parameters beyond those defined. If the command is valid without these extra parameters, then it is a valid command.

### SETTINGS STATE

When the program is in the settings state the following message is printed to the user:

"Specify game settings or type 'start' to begin a game with the current settings:\n"

Whenever the game enters the settings state, the above message is printed **once**, including upon startup of the program, and when the user explicitly enters the settings state with the *reset* command (described later).

Immediately after the message is printed, the program repeatedly prompts for user commands. Once the user enters the *start* command, the program enters the game stat with the current settings and the game is started.

The commands in the settings state are:

1. game_mode x
    - Sets the game mode:
        i. 1 – one player mode (a player vs. AI)
        ii. 2 – two players mode
    - After executing this command, the program prints the message: "Game mode is set to XYZ\n", where XYZ is either "2-player" or "1-player".
    - In case a wrong game mode is entered (wrong or invalid game mode, or no game mode parameter supplied), the program prints "Wrong game mode\n" and the command is not executed.
2. difficulty X
    - Sets the difficulty level of the game. The value of *X* can be 1-5.
    - Only legal if the game is set to 1-player mode. Otherwise, you must treat this command as any other invalid command.
        i. 1,2,3, 4 or 5 – This command sets the difficulty level to amateur, easy, moderate, hard, and expert, respectively. The program prints the message: "Difficulty level is set to XYZ\n", where XYZ is the name of the difficulty (amateur, easy, etc.).
        ii. If the user enters a value that is not in the range 1 to 5, then the message "Wrong difficulty level. The value should be between 1 to 5\n" is printed, and the difficulty value is not set.
3. user_color X
    - Valid only if the game is set to 1-player mode. The user's color will be set accordingly (either black or white). The value of X can be *0* or *1* for black and white, respectively.
        i. If the user enters a value that is not 0 or 1, then the message "Wrong user color. The value should be 0 or 1\n" is printed.
        ii. If the user enters a correct value, then the message "User color is set to XYZ\n" is printed, where XYZ is 'black' or 'white'.
    - Notice that like the difficulty command, this command is only allowed in 1-player mode. For 2-player mode, this command should be treated as an illegal command.
4. load X
    - Loads the game setting from a file with the name "X", where x includes a full or relative path of the file.
    - In case the file does not exist or cannot be opened, the programs prints "Error: File doesn't exist or cannot be opened\n" and the command is not executed.
    - You may assume that the file contains valid data and is correctly formatted.

5. default
    - Resets all game setting to the default values. The program prints "All settings reset to default\n".

6. print_settings
    - Prints the current game settings to the console in the following format:
        - For 2-player mode:
          “SETTINGS:\n
          “GAME_MODE: 2-player\n”
        - For 1-player mode:
          “SETTINGS:\n
          “GAME_MODE: 1-player\n”
          “DIFFICULTY: X\n”
          “USER_COLOR: Y\n”
          Where X is the difficulty level (easy, amateur, hard, etc.) and Y is the user color (either the string 'black' or 'white').
7. quit
    - Terminates the program. All memory resources must be freed. Once the user invokes this command the program prints: “Exiting...\n”
8. start
    - Starts the game with the current settings. The program prints "Starting game…\n"

Special Remarks:

- If the user enters two commands that affect the same parameter, then the latest update is taken:
    - For example: if the user first sets the difficulty level to 1 and afterwards she changes it 3 then the game difficulty level is set to 3.
- If a game setting parameter is not specified, then the default value is taken.

## GAME STATE

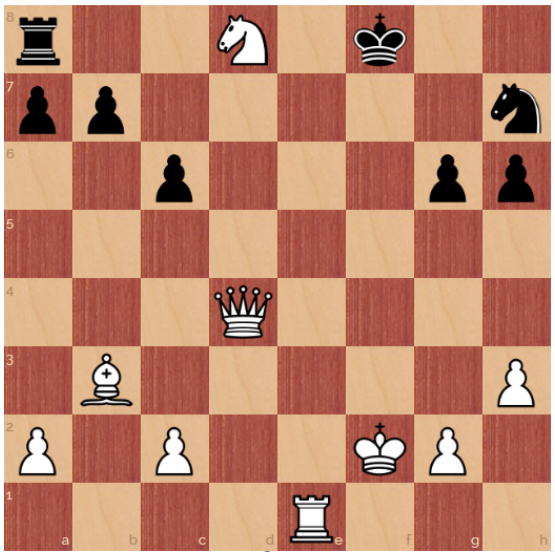
Once the user enters all settings and invokes the *start* command, the game begins by switching to the game state.

The board is printed to the screen at the beginning of each user’s turn in the following way: If the game is in 2-players mode then board is printed in each turn, otherwise the game board is printed only at the beginning of the user’s turn. Notice that this means that the board is printed at most once in each turn. The board is always printed at the start of the game, i.e., if the computer is the white player, the board is printed followed by the computer move and then an additional printing of the board for the user turn.

The game board is printed as described in the following format:

1- The board is an 8x8 grid. Rows are numbered 1-8 in upward order, and columns are lettered A-H in left-to-right order.
2- Black pieces are represented by uppercase letters, and white pieces are represented by lowercase letters.

3- Each piece is represented by its first letter, except for the knight piece, which is represented by the letter **n or N**.
4- An underscore '_' represents blank squares.
5- One space separates all pieces, and all blank squares.
6- Each row starts with its corresponding number followed by the pipe '|' character and a space, and ends with a space and a pipe '|' character.
7- After the final row (numbered 1!) a line of 2 spaces followed by 17 dashes is printed.
8- Another row (below the dashes) should be printed, which contains 3 spaces followed by A to H separated by one space.



```
8| R _ _ n _ K _ _ |
7| M M _ _ _ _ _ N |
6| _ _ M _ _ _ M M |
5| _ _ _ _ _ _ _ _ |
4| _ _ _ q _ _ _ _ |
3| _ b _ _ _ _ _ m |
2| m _ m _ _ k m _ |
1| _ _ _ _ r _ _ _ |
  -----------------
  A B C D E F G H
```

After the board is printed, the program asks the relevant user to enter her move: "Enter your move (XYZ player):\n", where XYZ stands for 'black' or 'white'.

The above message is printed to the user again, until it performs a successful "move" command, i.e., if an error occurred then an error is printed followed by the above message again, and if the user executes a command other than "move", "reset", or "quit", then that command is handled followed by the above message again.

Notice that in 1-player mode the message is printed only before the user's turn, and in 2-player mode the message is printed before each turn.

The user can execute the following commands each turn:

1. move <x,y> to <i,j>
    - This command executes the user turn by moving the piece at <x,y> to location <i,j>. *x* and *i* represent the row number, which can be between 1-8. The values of *y* and *j* represents the column letter, which can be between A-H (upper case). For example: *move 2,A to 3,A*
      You need to handle the following errors:

    (1) If either one of the locations is invalid, the program prints "Invalid position on the board\n".

(2) If position <x,y> does not contain a piece of the user's color, the program prints "The specified position does not contain your piece\n".
(3) If the move is illegal for the piece in the position <x,y>, the program prints "Illegal move\n".
(4) If the king is threatened ("Check") and will still be threatened after the move, the program prints "Illegal move: king is still threatened\n".
(4) If the move causes the player's king to be threatened, the program prints "Ilegal move: king will be threatened\n".

- You must print only one error message for each command. In case more than one error occurs, you need to print the error with the lowest ID as specified above.
- If no errors occur, the board is updated, and the following messages may be printed, depending on the case:
    a. If there is a checkmate, the program prints: "Checkmate! XYZ player wins the game\n", where XYZ is either black or white.
    b. If a king is threatened but there is no checkmate, print "Check: XYZ king is threatened\n", where XYZ is either black or white.
    c. If there is a draw, print "The game ends in a draw\n".
    d. For both checkmate and draw, the program releases all resources and then terminates.

2. get_moves <x,y>
    - This command prints all possible moves of the piece located at <x,y>.
        i. If the position <x,y> in the command is invalid, print "Invalid position on the board\n".
        ii. If position <x,y> does not contain a piece, print "The specified position does not contain a player piece\n".
    - If the position is correct and contains a player piece, each possible move of that piece will be printed in a new line. If there are no possible moves for that piece, nothing is printed.
        i. Each move is printed as "<x,y>\n".
        ii. If the move is threatened by an opponent, then a star will be printed, for example, "<7,A>*".
        iii. If the move captures a piece: "<x,y>^".
        iv. If the move satisfies both: "<x,y>*^".
    - Note that a user can get moves of a piece of a color different than her own.
3. save X
    - Saves the current game state to the specified file, where X represents the file's relative or full path.
    - If the file cannot be created or modified, the program prints the following message: "File cannot be created or modified\n" and the command is not executed.
    - If no error occurs, then the current game setting and the board status is saved to the specified file, and the program prints "Game saved to: XYZ\n", where XYZ is the filename provided by the user.

4. undo
    - Undo previous moves done by the user. The user may undo up to 3 moves. Thus, you need to store the previous 3 moves for the user and the opponent (the AI or another user). A move that is "forgotten" cannot be undone.
    - If the user is trying to invoke the *undo* command and the history is empty, the program prints "Empty history, no move to undo\n"
    - In case the *undo* command can be executed, then the program will undo the previous two moves (one for each player) and prints the following messages for each move: "Undo move for XYZ player: <x,y> -> <w,z>\n", where XYZ, <x,y> and <w,z> represents the player color ('black' or 'white'), the position of the piece after the last move executed and its original position.
    - If there is only one move to undo, then the program will undo the previous move only.
    - If the undo move was executed successfully (for one or two moves), the program prints the board again.
5. reset
    - The program state is switched to the setting state. The following message is printed after the execution of this command "Restarting...\n"
6. quit
    - Terminates the program. Prior to terminating, all resources and memory should be freed, and the following message is printed: "Exiting...\n".

## COMPUTER TURN

For each AI move, the program prints the computer move in the following format, where each move is represented by the original position <x,y> and the destination <i,j>:

"Computer: move [pawn|bishop|knight|rook|queen] at <x,y> to <i,j>\n".

Once the board is updated the following happens:

1. Check for checkmate or draw:
    a. If the user is unable to make any move in the next turn then:
        - In the case of "mate": the message "Checkmate! XYZ player wins the game\n" is printed, where XYZ stands for 'black' or 'white'.
        - In the case of a draw, print the message "The game ends in a draw\n".

Note that in both cases (checkmate and draw) the program terminates.

    b. If the AI is threatening the opponent's king, the program prints "Check: XYZ king is threatened\n", where XYZ is either 'black' or 'white'.
2. In case the game is not over, the game moves to the next turn (the user's turn).

## COMPUTER AI

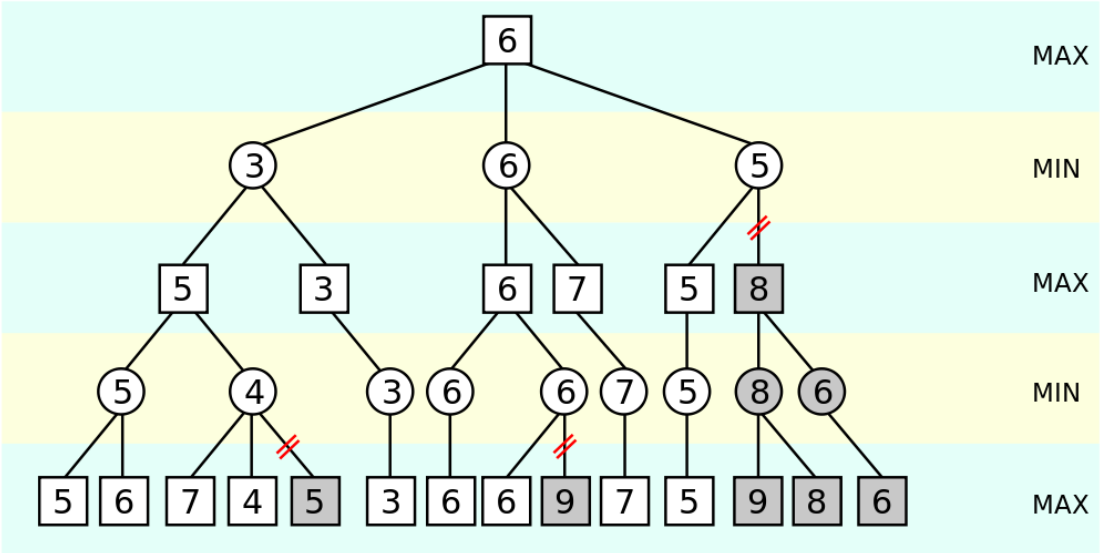
The AI used in the project is the Minimax algorithm from HW3. The depth of the Minimax (number of steps to look ahead) is defined by the difficulty level.

The Minimax algorithm's basics stay the same; however, in the final project you are not required to construct the Minimax tree in a different step, and are in fact encouraged to perform your calculations while building the tree (notice that this time you should avoid creating an actual tree structure). In addition, we add **pruning** to the algorithm as described next.

In the original Minimax algorithm from HW3, we scan all possible moves for each game – sometimes more than necessary.

For example, suppose a MAX node has a child node X for which we finished calculating values, getting a final value of 5. Proceeding to the calculations of its next child node Y, supposed Y has a child node K with a value of 4 (a grandchild of our original MAX node), then since Y is a MIN node, it will always choose the lowest possible value which, due to K, can't be greater than 4. This means we can stop evaluating that branch of the tree (the subtree rooted in Y) – it's value is at most 4, and we already calculated a higher value 5 for the node X, thus we can proceed to the next node for calculation.

The following image shows an example of pruning, where the Minimax tree is scanned from left to right. You should understand why the grayed-out nodes need not be evaluated at all:




The pruning algorithm is left for you to understand and implement accordingly. Example slides are provided in Moodle, and A description of pruning along with pseudo-code can be found here: http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning.

*In order to ensure our Minimax is deterministic, in case there are two (or more) moves with the same score, we will prefer the move with the lowest: source column (x), source row (y), destination column, destination row, e.g., we prefer (5,A)=>(7,A) over (6,A)=>(7,A), we prefer (5,A)=>(7,A) over (5,A)=>(7,B), we prefer (5,A)=>(7,A) over (5,A)=>(8,A), etc.*

## SCORING FUNCTION

We use a naïve scoring function which outputs a score according to the pieces on the board, disregarding piece positions and possible moves. The scores for each piece are:

Pawn = 1, Knight = 3, Bishop = 3, Rook = 5, Queen = 9, King = 100.

If the board state is a draw, our scoring function outputs **0**. For Checkmate, our scoring function outputs +1000/-1000 (according to the winning player). For Check, no special handling is required and the naïve scoring (according to the pieces on the board) is used.

## SDL

Simple DirectMedia Layer (SDL) is a cross-platform, free and open-source multimedia library written in C that presents a simple interface to various platforms' graphics, sounds, and input devices. It is extensively used in the industry in both large and small projects. Over 700 games, 180 applications, and 120 demos have been posted on its website.


SDL is a wrapper around operating-system-specific functions. The main purpose of SDL is to provide a common framework for accessing these functions. For further functionality beyond this goal, many libraries have been created to work on top of SDL. However, we will use “pure” SDL – without any additional libraries.

The SDL version we’re using is **SDL 2.0.5**, a documentation can be found in the following link: https://wiki.libsdl.org/FrontPage.

Following are a few examples of common SDL usage. However, this isn’t a complete overview of SDL, but only a short introduction – it is up to you to learn how to use SDL from the documentation, and apply it to your project.

**Make sure your makefile compiles your Chess game on Nova, see the example makefile on moodle for reference.**

In our code files we need to include $SDL.h$ and $SDL\_video.h$, which will provide us with all the SDL functionality we’ll be using.

To use SDL we must initialize it, and we must make sure to **quit SDL** before exiting our program. Quitting SDL is done with the call $SDL\_Quit()$. To initialize SDL, we need to pass a parameter of flags for all subsystems of SDL we use. In our case the only subsystem is VIDEO, thus the call is $SDL\_Init(SDL\_INIT\_VIDEO)$.

A brief introduction to SDL was given in tutorial 3, and you may find examples on Moodle.

**Note:** SDL calls can fail! Check the documentation and validate each call accordingly.

## USER INTERFACE

In this section we describe the graphical user interface. You may choose to design the interface as you wish, so long that it follows the requirements in this document.

### RESOLUTION

The resolution of the game window should be at least 800x600 (width x height), and no more than 1024 x 768. All other windows must have a resolution of at least 400x400.

### MAIN WINDOW

When starting up, the program will display the **main window** to the user, which will present the user with (at least) the following options:

1. **New Game** – start a new Chess game
2. **Load Game** – allows the user to load a previously-saved game
3. **Quit** – frees all resources used by the program and terminates

The main window will also be displayed to the user whenever choosing to return to the main menu from within any game or window.

### NEW GAME

After pressing the New Game button, a new dialog will open to enable the user to set the following options:

1. **Game mode** - the user may specify 1-player or 2-player mode
2. **Difficulty level** – for 1-player mode, the user may specify the difficulty level of the game (amateur, easy, moderate, hard, and expert). This option should be disabled when 2-player mode is selected.
3. **User color** – for 1-player mode, the user may specify the color of its pieces. This option should be disabled when 2-player mode is selected.

In addition, the dialog should contain a *Back* button which returns to the main menu, and a *Start* button that starts a new game.

You may also implement this process as several dialogs (ask the game mode, then difficulty, then user color) – however, the user should always have the option to go back to the previous dialog or the main menu.

### GAME WINDOW

The UI of the game window is up to you, so long that it provides the user with the following options:

1. **Restart game** – restarts the game with the current game settings.
2. **Save game** – saves the game to a slot chosen by the user (described later).
3. **Load game** – loads a previously-saved game according to a slot chosen by the user (the load window is described later).

4. **Undo** – the user may undo previous moves if allowed, the history size is 3, as in Console mode. When Undo is unavailable, the options should be disabled or hidden.
5. **Main Menu** – frees all resources used by the game and returns to the main window.
6. **Quit** – frees all resources used by the program and terminates.

Note:

- If the user's current game is unsaved (the user didn't save it, or it was saved but moves were performed since then) – then before returning to the main menu or quitting, a dialog should pop up asking the user if she wants to save the current game or not (the options should be: Yes, No, and Cancel). You should use SDL_ShowMessageBox for this.
- It is recommended to give the user some indication whether the current game is saved or unsaved.
- If the user saves the game, performs a move, and then chooses undo, it's OK to treat it as an unsaved game.
- You need to indicate if there is/was a check/mate or a draw.

**Highlight move:**

- The user may get the available moves of the user (like *get_moves* in Console mode) by right-clicking a pawn.
- The relevant spaces on the board (of possible moves) should be highlighted by four different colors: a standard square, a threatened square (that is, a square that is threatened by an opponent piece), a capture square (occupied by an opponent piece), or a square that is both threatened and a capture square.
- You should **not** hide other pieces in the highlighted squares. Images with partial transparency will help!

### LOAD WINDOW

- Presents 5 save-game slots to the user.
- Slots that were not used yet should be disabled or hidden.
- The user may return to the previous window (game or main) by pressing the "Back" button.
- The same window should be presented to the user no matter where it came from.
- The user loads a game either by pressing the relevant slot, or by marking the slot (by pressing it) and then pressing the "Load" button – your choice.

## SAVED GAMES

Each game can be saved at any point, and then later loaded to resume playing from that point. To save a game, the user selects the Save button from within a game. To load a game, the user selects the Load option from the main menu or the game window.

Since SDL doesn't directly provide us with a convenient way to output text, we will not allow to user to specify a file to save or load, but rather limit it to 5 "slots" with fixed file-names. This number and the file names should be constants in your code, you should not assume it's always 5 (and your load-game GUI should update accordingly – preferably with a scrollbar).

Each slot is associated with a predetermined file, and the user chooses a slot to save to or load from. The file name is not visible to the user from the program, but the file itself should be easy to find (somewhere in the project's folder).

### FILES FORMAT

The saved file should contain the following information:

1. Current turn – which player should play (*black* or *white*)
2. Game mode – 1-player or 2-player
3. User color (relevant for 1-player mode only)
4. Difficulty level (relevant for 1-player mode only)
5. The board's state

The file format should be text, and its contents should be the current player color ('black' or 'white') in the first line, followed by an output identical to the *print_settings* command, and finally an output of the board.

Your code should support both loading and saving for this exact format. Examples can be created from the supplied executable (compiled console Chess program).

## ERROR HANDLING

Your code should handle all possible errors that may occur (memory allocation problems, safe programming, SDL errors, etc.).

File names can be either relative or absolute and can be invalid (the file/path might not exist or could not be opened, even in graphical mode – do **not** assume that since they're not provided by the user, they're valid – treat them as if they were user input).

Make sure you check the return values of all relevant SDL functions – most SDL functions can fail and should be handled accordingly. In case of errors, **do not exit unless you must!**
For example, if you fail loading a file – show a message box to the user and return to the previous window.

Throughout your program do not forget to check:

- The return value of a function. You may excuse yourself from checking the return value of any of the following I/O functions: $printf$, $fprints$, $sprint$, and $perror$.
- Any additional checks, to avoid any kind of segmentation faults, memory leaks, and misleading messages. Catch and handle properly any fault, even if it happened inside a library you are using.

Whenever there is an error, print to the console – which is still available even if you are running a program with a GUI. You should output a message starting with "ERROR:" and followed by an appropriate informative message. Also show a message to the user (use a message box). The program will disregard the fault command and continue to run, if possible.

Terminate the program only if no other course of action exists. In such a case free all allocated memory, quit SDL, and issue an appropriate message before terminating.

## BONUS

For this project, you may implement the bonus for the possibility to earn **10** extra points. These points will be added to your project grade and, if exceeding 100, will also improve your HW grade (i.e., remaining points will improve the grade of HW3, then HW1 and HW2).

To implement the bonus, you should use the tools learned in the Concurrent Programming lecture (Tutorial 4), to create a concurrent implementation of the Minimax algorithm and user interaction while the Minimax algorithm is executing (both in GUI and Console).

You are **only** allowed to use the synchronization methods presented in the tutorial, and no other methods/functions/objects, unless you fully implement it yourself.

You must properly implement all the requirements below to earn the bonus. The bonus is a pass-fail bonus, i.e., either you receive all bonus points, or none, according to the quality of your submission. The requirements are as follows:

- Parallelize the Minimax algorithm – the performance of the Minimax algorithm should improve such that the expert difficulty (5) is playable.
- Allow user interaction – while the minimax algorithm is running, the user may enter the "quit" command in Console mode (any other command is considered invalid). In GUI mode, show a "Thinking" dialog with a Quit button.

## SUBMISSION GUIDELINES

The project will be submitted via Moodle.

You should submit a zip file named ***id1_id2_finalproject.zip***, where *id1* and *id2* are the IDs of both partners. The zipped file will contain:

- All project-related files (sources, headers, images).
- Your makefile.
- Partners.txt file according to the usual pattern.

## CODING

Your code should be gracefully partitioned into files (modules) and functions. The design of the program, including interfaces, functions' declarations, and partition into modules – is entirely up to you, and is graded. You should aim for modularity, reuse of code, clarity, and logical partition.

In your implementation, also pay careful attention for use of constant values and proper use of memory. Do not forget to free any memory you allocated, and quit SDL. You should especially aim to allocate only *necessary* memory and free objects (memory and files) as soon as possible.

Source files should be commented at critical points in the code and at function declarations. Please avoid long lines of code.

Recall to properly document each function. Properly separate public functions (declared in the header and available to other modules) from private functions (declared in the source file only and unavailable to other modules).

## COMPILATION

You should make your own makefile, which compiles all relevant parts of your code and creates an executable file named '`chessprog`'. Use the flags provided in the SDL tutorial (tutorial 3) to link the SDL library properly. Your project should pass the compilation test with no errors or warnings, which will be performed by running the **make all** command in a UNIX terminal on **nova**.

# GOOD LUCK