Software Project, spring 2015

Due by 07.10.2015, 23:55

This is your final software project; it builds on the infrastructure developed in ex. 3, with the necessary alterations to the interface and body. In your final project you will create an interactive chess program with a graphical user interface (GUI) and computer AI.

Introduction

The Chess Program project is an implementation of a fully-functioning two-player Chess game.

The program has two modes:

graphical mode (Gui mode) - The graphical mode presents the user with visual menus and controls enabling the user to play Chess, choose the game's players (user or AI), and set the game's difficulty.

Console mode - The console mode will operate in a similar way to your implementation for the Draughts game in hw3.

For AI, the Minimax algorithm will be used. The Minimax algorithm is mostly the same as the algorithm in ex. 3, using **pruning** to improve its efficiency.

High-level description

The project consists of 6 parts:

- Console user interface for the Chess game
- Generic graphics framework to be used by the program.
- User interface for the Chess program.
- Generic Minimax algorithm for AI.
- (Bonus)
    - 5 pts. Chess player with improved scoring function and minimax depth selection which outperforms the trivial one (see below).
    - 2 pts. Casteling move implementation (see below).
    - Note that the maximum grade for a project is 105.

The executable for the program will be named "$chessprog$". The graphical mode is defined by a command line argument:

./chessprog console – will start the program in console mode.

./ chessprog gui – will start the program in Gui mode.

./ chessprog – will execute the program with the default execution mode - console

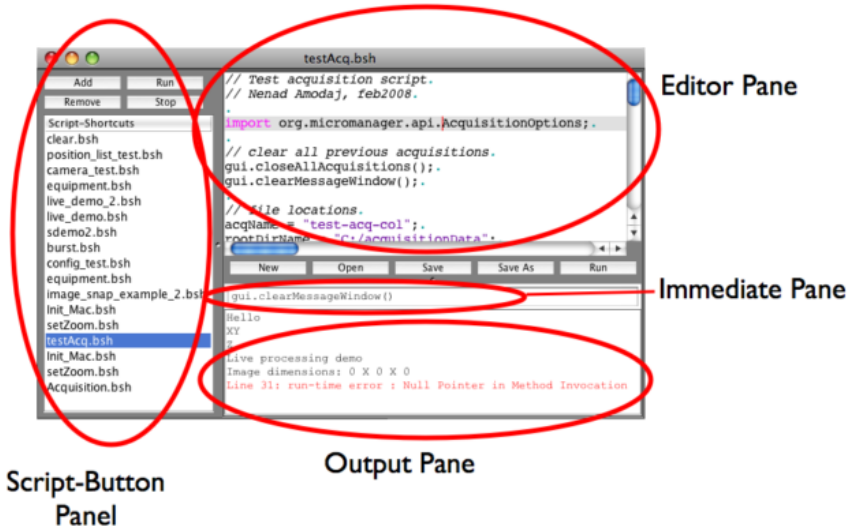
Graphical Framework

The Chess project contains a GUI for each of its components – the program's windows, dialogs and menus, and also the game itself.

We'd like to prevent code duplication and creating complex bulks of code for drawing elements to the screen, and for that we'd like to create a graphical framework that will provide us with generic and convenient functions and elements to allow us to create UI windows efficiently.

First, we'll define the controls we need. A **control** (also called a GUI widget) is any graphical component that is drawn (or assists in drawing) to our screen.
The project will use ***at least*** the following controls: window, panel, label/image, and button.

A **window** represents the root of the screen, containing a list of controls within it. A **panel** is similar to a window, however it's a control contained inside a window, creating a hierarchy. Since we don't use text in SDL, each **label** control is actually an **image** control, and so is the graphical representation of a **button**.



Panels inside a window

Note that all of these controls can be created with a single $struct$ with various data members and function pointers, with each control having a specific "factory" function returning an instance of it (e.g., $control * createButton(< params >)$).
The specific implementation is up to you.

UI Tree

After having all of the controls, a common way to create windows is with a **UI tree**.
In a UI tree, the window control is the root of the tree, and each control contained within it is its child node. A panel has child nodes of its own – the controls contained within the panel. This allows us to easily create a window by constructing its relevant UI tree, and easily draw it by scanning the UI tree with **DFS** and drawing each control in that order.

When drawing the UI, certain controls can overlap. In such a case, we need to determine which control is drawn on top of which. It's clear that controls that are drawn *later* are the ones that are drawn on top of controls that were drawn *previously*.

When using DFS, we have a tree structure (the UI tree) that represents our window. If scanning from left-to-right, this means that overlapping controls are drawn according to their position in the tree – right children are drawn over left children.
An easy way to change this is to change the order of the children of a node (window or panel), and in such a way change the order in which they are drawn.

Panel

A panel is a special control, as it hosts other controls within it. It's similar to a window, but provides more functionality. In addition, there must only be exactly a **single** window control in a UI tree, and it's always the root.

Besides grouping together controls under a single node, a Panel provides boundaries.
This means two things:

1. Controls inside a panel are drawn *relative* to the panel.
   For instance, if a panel is at position $(5,5)$, and contains a button with position $(10,10)$, the actual position (**of the window**) in which the button is drawn to is $(15,15)$!
2. Controls inside a panel must reside within its borders.
   A panel defines, beyond its position, a width and height. If a control exceeds it – it should only be drawn partially.
   In the previous example, if the panel's size is $(10,10)$, the button exceeds the boundaries, and thus only a portion of size $(5,5)$ of the button should be drawn.
   This can be achieved by limiting the drawn controls' rectangles according to any boundaries they are in.
   Note: a window has no boundaries, and no position – but can be assumed to be at position $(0,0)$ with a size equal to the current resolution, for convenience.

It's important to note that panels can contain additional panels within them, creating a hierarchy. These additional panels themselves obey these rules (such that panel A contained within panel B is relative to B's position, and a button inside A is relative to A's position – which itself is relative to B, etc.)

Your implementation should mind these situations and handle them properly.

User Interface

Resolution:

The resolution of the game should not exceed 1024x768. The preferable (but not mandatory) resolution is 800x600.

Main window:

When starting up, the program will display the **main window** to the user, which will show some logo and present the user with the following options:

- **New Game**
  Start a new Chess game.
- **Load Game**
  Asks the user for a location (file slot) and then opens the relevant file and resumes the game that was previously saved in it (save/load and file slots are described later).
- **Quit**
  Frees all resources used by the program and exits.

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

Players Selection Window:

After selecting a new/load game, a new dialog will be shown presenting us with player selection options – how will each player be controlled (user or computer).
You should present the options to set:

1. Game mode : **Player vs. Player** and **Player vs. Computer**.
2. Set the board: This should be an optional step. You should provide an interface for updating the pieces on the board (and obviously avoid the illegal initialization). The default setting should be a full board (all pieces) for a new game, or the loaded board for a game loaded from a saved slot.
3. Next player: default value should be "white"

In addition, add a $Cancel$ button that should take us back to the main menu.
After the selection, the user is shown the “Game Window” or the "Settings window", depends on the selected game type.

AI Settings Window:

This window will be shown only in case the user chooses "player vs. AI". This window enables the setting of the color for the user (or the computer), as well as the difficulty.
There are two possible types of values for the diffuculty:

1. A constant depth for the minimax algorithm (a value between 1 to 4 that is specified by the user)
2. Best depth – this option allows the computer to execute minimax algorithm in different depths, depending on the state of the game. The number of evaluated boards **should not exceed 10^6**. This means that the depth of the minimax algorithm depends on the possible moves for each player, and how efficient your pruning implementation is. Notice that you must determine the depth of the algorithm before you start executing minimax. You can determine it according to the number of pieces on the board (for x pieces there is an upper bound on the possible moves, and the number pieces cannot grow during the game).

Board Setting:

In the case of a new game (i.e. a game that wasn't loaded from a saved slot), you need to provide the user with the option to update the board before the game starts, but the user should not be forced to do so (As opposed to the "AI settings window" which is a part of the flow in player vs. AI mode). In case the user chooses to update the board, you should load a full board and allow the user to move/remove/add pieces. You should prevent the user from illegal initializations (too many pieces of a type, etc). Make sure the user is prevented from starting a game with one of the kings missing.

Game Window

We do not provide UI instructions for the game window – it's up to you.
However, the game window must provide the user with the following options:

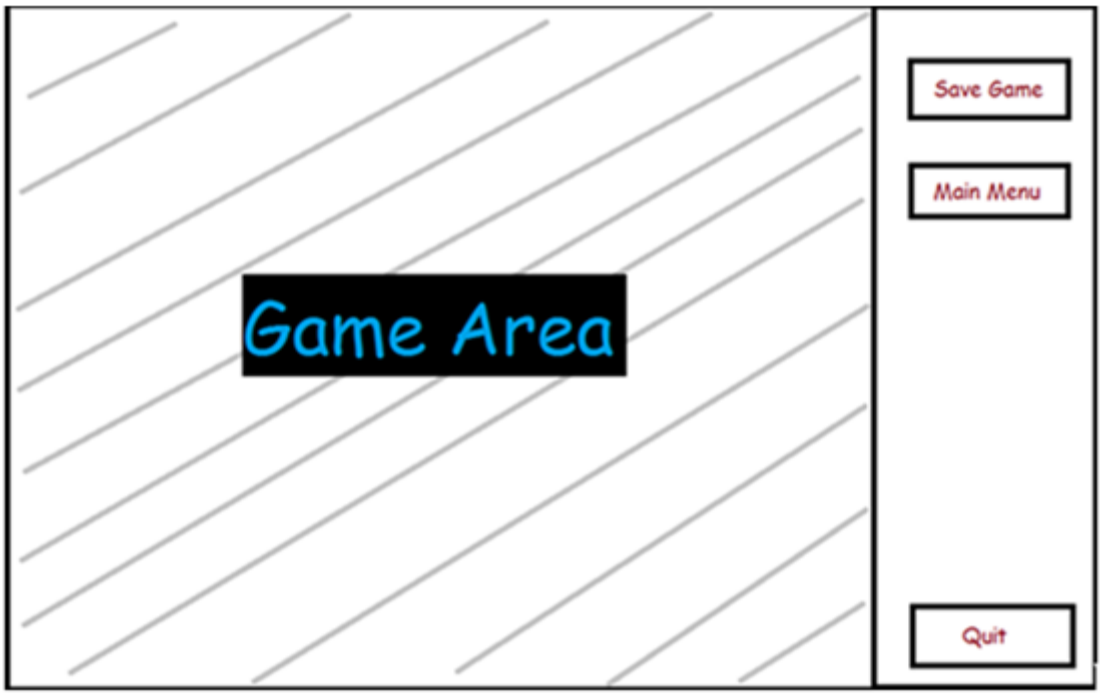
- **Save Game**
  Asks the user for a slot (described later) and saves the game's current state to the relevant file. The game then continues regularly from where it was left off.
- **Main Menu**
  Frees all resources used by the game and returns to the main window, without saving.
- **Quit**
  Frees all resources used by the program and the game, and exits.

There are two possibilities to draw the parts relating to the specific game we're playing:

1. The "Game Area" (shown below) is a branch of the UI Tree. Its root is a Panel, under which all of the game components reside (including any controls, such as additional panels which have additional child nodes of their own).
   The game overlay then only needs to hold a special "game area" node which is replaced with the game chosen, to be able to draw the entire game window generically.
2. Creating a special control – a "Game Control", which holds a pointer to a game struct (described later). The "Game Area" in the UI tree of the overlay is actually a control of that type. Whenever the control is drawn, it calls the game's specific draw function along with the coordinates of the game area (so it doesn't exceed its boundaries).

The choice of which method to use is up to you.

Here is an example of a game overlay:



When the user chooses an option that requires a selection (changing difficulty, etc.), a **dialog** will be shown to the user.
A dialog can be drawn on top of any existing window, or can be a new window entirely, and will show a **generic** series of buttons to the user, constructed according to the available (such as a list of difficulties for current game, etc).

The user interface for the program will be written in SDL 1.2, constructing a framework to provide both a **GUI** and an input system for handling mouse and/or keyboard events.

Whenever the user needs to choose a GUI input (press a button, choose an entry from a list, etc.), it can be implemented by a mouse click, a keyboard travel, or both.
Keyboard travel means that the user can navigate using the arrow keys, each time highlighting his current choice. The user makes his selection by pressing "$enter$".

When the game is over, all overlay options should still be available (including "Save Game"!). However, all options in the game area should be disabled – such that they are not selectable by the user, or that when the user selects them the selection is ignored.

Your use interface must provide the player with all the options to play a chess game. The player should be able to make legal move (illegal moves should not be executed). The user should be notified about Check, Mate and Tie. Also, you must provide an option for the user to see the best move for him. For "computer vs. AI" mode, the depth of the minimax will be the depth of the AI. For "2 players" the player should choose the depth of the minimax (don't forget the "best" option).

Chess

Rules:

Each Player begins the game with 16 pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. Each of the six piece types moves differently. Pieces are used to attack and capture the opponent's pieces. Unlike in Draughts, capturing is not obligatory in this game. 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 thy move).
A victory is achieved when the player can capture the opponent's king. Victory is declared before the actual move is executed.

You may find more information here about the game:
http://en.wikipedia.org/wiki/Chess
The pieces' types:

- Pawn:
  - Movement: a single move forward in the same column.
  - Capturing: diagonally, one square forward and to the left or right
    Promotion - A pawn that advances all the way to the opposite side of the board (the opposing player's first rank) is promoted to another piece of that player's choice, other than a king.
  - Console representation: m/M (lower case for white, capital letter for black).
  - There is no need to implement *en passant* moves.

- Bishop
  - Movement: can move any number of squares diagonally, but may not leap over other pieces.
  - 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.
  - Console representation: r/R

- Knight:
  - Movement: moves to any of the closest squares that are not on the same rank, file, or diagonal, thus the move forms an "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
  - You are not required to implement *casteling* move, however, if you do implement it you will get a bonus (see bonus section)

Check, Mate and Tie:

Check -

- When player's king is threatened by the opponent, it's called "check". A response to a check is a legal move if it results in a position where the king is no longer under direct attack.
- A player cannot perform a move that will result in his own king being threatened by the opponent. Such a move is considered to be illegal and should be treated as any illegal move by the player.

Mate-

- When player's king is threatened by the opponent ("check") and the king cannot be saved (there are no legal moves)
- Mate terminates the game (In Chess, victory is achieved before the actual capturing of the king is executed)

Tie-

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

Command line mode:

At the beginning of the program in command line mode, the game board is being initialized and printed to the console.
You may assume the length of the user's command will not exceed 50 characters.

Settings state:

The program prints the initialized board and then prints the message `"Enter game settings:\n"` and repeatedly prompts for the user's commands. This message will be printed every time the user is expected to enter a command.
The commands in the chess game are based on the Draughts game you implemented in hw3, with a few changes.

1.
   - command: game_mode x
   - description: this command sets the game mode. X can accept two possible values – *1* or *2*:
     i. 1 - two players mode.
     ii. 2 – player vs. AI (computer) mode.
   - After executing this command, the program prints the message: `"Running game in XXX mode\n"`, XXX can either be `"2 players"` or `"player vs. AI"`.
   - In case a wrong game mode is entered, the program prints `"Wrong game mode\n"` and the command is not executed.
   - default value: 1 (two players mode)
2.
   - command: difficulty [depth x| best]
   - description: this command is only legal when the game is set to "player vs. computer" mode. Otherwise, you must treat is as any other invalid

command. This command sets the number of step for the minimax algorithm in one of the following ways:

i. difficulty depth x – This commands sets a constant depth for the minimax algorithm to x. x should be between 1 to 4, otherwise the message `"Wrong value for minimax depth. The value should be between 1 to 4\n"` will be printed, and the value will not be set.

ii. difficulty best – best argument allows the computer to execute minimax algorithm in different depths, depending on the state of the game. Using this option, the number of evaluated boards should not exceed 10^6.

- defaut value: the default value is a constant depth of 1.

3.

- command: user_color x
- description: sets the color of the human player in player vs. AI mode. The value of x is either *black* or *white*
- Notice that like the difficulty command, this command is only allowed in "player vs. AI" mode. For 2 players mode, this command should be treated as an illegal command.

4.

- command: load x
- description: this command loads the game settings from the file x, x being the full or relative path to the file.
- After loading the file the loaded game board is being printed.
- In case the file does not exist, the programs prints `"Wrong file name\n"` and the command is not executed.
- You can assume that contains valid data and is correctly formatted. More on the input files format is the **Files** section.
-

5.

- command: clear
- description: this command clears the check board (removes all pieces).

6.

- command: next_player a
- description: this command sets the player who plays first. The value of **a** will either be *black* or *white* By default, this would be the white player,

7.

- command: rm <x,y>
- description: this command removes the piece located in <x,y> position, x represents the column and y represents the row.
- If the position <x,y> is empty, nothing happens (no message is printed)
- if **x** values is not within the range of 'a' to 'h' or **y** is not within the range of 1 to 8, the program prints the message `"Invalid position on the board\n"` and the command is not performed.
-

8.

- command: set <x,y> a b

- description: this command places a piece of color a and type b in <x,y> position.
- The value of **a** should either be *white* or *black*. The value of **b** should either be one of the piece types (*king*, *queen*, *rook*, *knight*, *bishop*, *pawn*)
- if **x** values is not within the range of 'a' to 'h' or **y** is not within the range of 1 to 8, the program prints the message `"Invalid position on the board\n"` and the command is not performed.
- If the added piece exceeds the amount of allowed pieces on the board (i.e. adding a second king, third rook of the same color), the program prints the message `"Setting this piece creates an invalid board\n"` and the command is not performed.

9.
- command: print
- description: this command prints the current game board to the console.

10.
- command: quit
- description: this command terminated the program.

11.
- command: start
- description: after this command is entered the program move the **game state** and the user cannot enter settings command anymore.
- If one of the kings is missing, the following message will be printed `"Wrong board initialization\n"` , and the command will not be executed.
- You should handle the case when the board is initialized such that the first player will not move (tie or lose). You can handle it either within the start command or before the first player make his move. It's up to you. Make sure you print the right message (see the end of game messages in "game state").

Game state:

User's turn.
The program prints message `"XXX player - enter your move:\n".` XXX stands for the current player's color. Since we have 2 players mode, we must specify the current player's color. This message will be printed each time a user has to make a move.

The following commands are allowed.

1.
- command: **move** <x,y> **to** <i,j> x
- description: This command executes the user turn operating the move represented by <i,j>.
- X is an optional argument for a promotion command (see pawn promotion syntax).
- You may assume that X, if specified, is a valid piece type name.
- (1) if either one of the positions in the command is invalid, the program prints the following message `"Invalid position on the board\n"`.

- (2) if position <x,y> does not contain a piece of the user's color, the message `"The specified position does not contain your piece\n"` is printed.
- (3) If the move is not legal for the piece in the position <x,y>, the message `"Illegal move\n"` is printed.
- You must print only one error message for each command. The order of the tests is marked with (1), (2), (3) in the previous bullets.
- If there were no errors and the move was executed:
    a. The updated board is being printed to the console.
    b. Check for checkmate or a tie :
        - In the case of "mate" : the message `"Mate! XXX player wins the game\n"` is printed, XXX stands for the winner's color.
        - In the case of a tie: the message `"The game ends in a tie\n"`

    In both cases (mate and tie) the program terminates.
    c. If the user is threatening the opponent's king, the message `"Check!\n"` will be printed. This test should only be performed if there was not Mate.
    d. In case the game is still on, it's the computer's turn now.

2.

- command: **get_moves** <x,y>
- description: this command print all possible move for a player located in position <x,y>.
- (1) if the positions <x,y>in the command is invalid, the program prints the following message `"Invalid position on the board\n"`.
- (2) if the position <x,y> does not contain a piece of the user's color, the message `"The specified position does not contain your piece\n"` is printed.
- If the position is correct and contains one of the player's pieces, each move will be printed in a new line (you can print them in any order you wish). A move is represented by the position of the piece and the move itself. If there are no possible moves for this piece, nothing is printed.

2.

- command: **get_best_moves** d
- description: this command prints all the moves with the highest score for the current board. d is an argument for the minimax algorithm and can either be a number (1-4) or the value best. You can assume that the value of d is legal so you don't need to check it.
- the best move will be printed in the same format the moves are printed in get_moves command.
- In case there are several moves with the highest score, they all should be printed (the order doesn't metter).

3.

- command: **get_score** d m
- description: this command prints the score for the move m in a minimax tree of depth d (you can assume that d is either a number between 1-4 or the value *best*).
- The first level in the minimax tree is the move itself, meaning that for d=1, the score is the score for the board after executing the move m.

- The format of m is: **move** <x,y> **to** <i,j> x (the same format as performing the actual move).

3.
   - command: save x
   - description: this command saves the current game state to the specified file, x being the filename.
   - If the file cannot be created or overridden, the program prints the message `"Wrong file name\n"` and the command is not executed.
   - The data is saved in the file as described later in the files section.

4.
   - command: quit
   - description: this command terminates the game

Syntax for pawn promotion:
in case we need to promote a pawn, we'll represent a promotion move in the following way:
<a,7> to <a,8> queen
<a,7> to <a,8> bishop
<a,7> to <a,8> rook
<a,7> to <a,8> knight
(an example for a pawn located in <a,7>).
In get_moves command, you should specify these 4 options. In move command, you should also accept the option *move <a,7> to <a,8>*, which represents the default promotion and is equivalent to *move <a,7> to <a,8> queen*.
To execute a move, we'll add the word "move", like in the regular move command.
Notice that we must promote the pawn, otherwise it will be stuck in the last raw of the board.

Computer's turn:
The program prints the computes move in the following format:
`"Computer: move <x,y> to <i,j> x".`

- Each move is represented by 2 positions: the current position of the piece <x,y>, and the destination <i,j>.
- Two special moves: pawn promotion and castling.

Following this:

1. The updated board is being printed to the console.
2. Check for "checkmate" or a tie :
   a. If the opponent will not be able to make any move in its turn:
      - In the case of "mate" : the message `"Mate! XXX player wins the game\n"` is printed, XXX stands for the winner's color.
      - In the case of a tie: the message `"The game ends in a tie\n"`

      In both cases (mate and tie) the program terminates.
   b. If the AI is threatening the opponent's king, the message `"Check!\n"` will be printed.
3. In case the game is still on, it's the user's turn now.

Scoring function

The scoring function goes over the board and outputs a score according to the pieces on the board, same as for the Draughts game.
The scores for each piece:
pawn = 1 , knight = 3 , bishop = 3 , rook = 5, queen = 9, king=400
You can set a score for a winning/loosing board and also for tie. Bear in mind that tie should only be preferred over loosing. In any other case (victory or indecision), tie should is the worse option.
you should use this scoring function for difficulties 1-4. For "best" difficulty, you may implement another scoring function.

"Check" in graphical mode:

Make sure to indicate "check" situations after the relevant move is made (as long as it's clear, you can choose either way you please).

Bonus

1. Up to 5-point bonus will be provided for improvements to the scoring function for the Chess player with **best difficulty** that would beat the trivial player in a set of pre-designed games.
    - The trivial player uses the trivial scoring function we defined in the exercise. Its difficutly is set as we defined for the best player (namely, the depth is defined according to the board), and the maximum depth will not exceed 4. You can improve the scoring function, improve pruning (ordering the moves), as well as come up with an optimal way to decide the depth of the Minimax for each board.
    - Notice that if you choose to change the scoring function, this change should be applied only to the best difficulty. For every other difficulty, the scoring function should be the one that is defined in this document.
    - You may also alter the minimax algorithm, but make sure it only effects "best" mode, and in any other mode, the minimax algorithm works as described.

2. A 2-point bonus will be provided for the implementation of the *casteling* move. The console command for casteling is:
    - command: castle <x,y>
    - description: <x,y> representing the rooks position. If <x,y> is illegal position on the game board, or <x,y> is not occupied buy a piece that belongs to the player, you must print the same error messages as in the *move* command.
    - If <x,y> does not contain a rook, the computer should print the following message `"Wrong position for a rook\n"`
    - Otherwise, if <x,y> is occupied buy a rook but casteling is not possible, the message `"Illegal castling move\n"` should be printed.
    - You can find an explanation about the casteling move in the appendix at the end of this document.

Note that the grade still can't exceed 105.

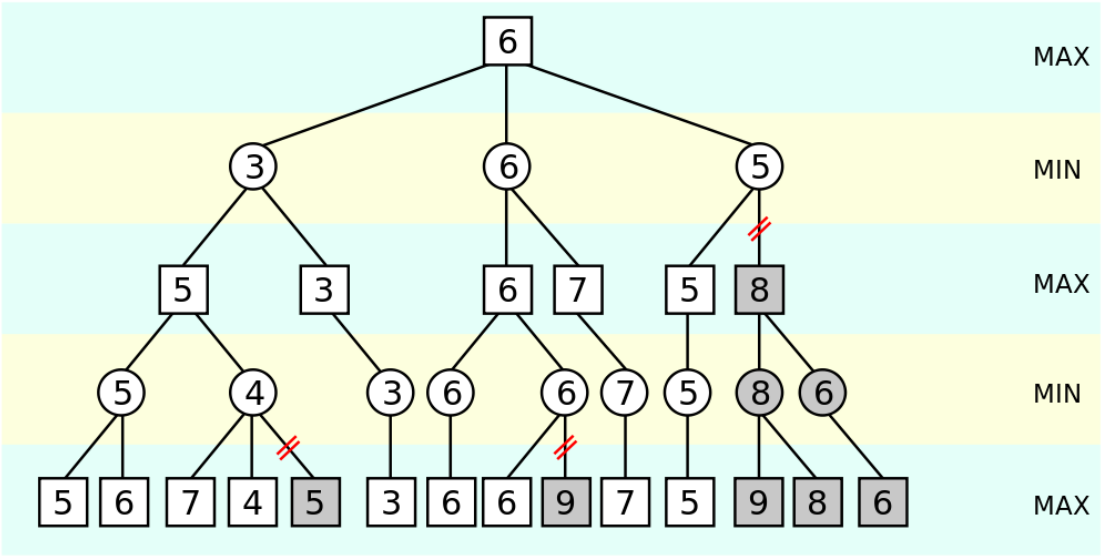
Computer AI

The AI used for the Chess game in the project is the minimax algorithm, learned as part of ex3.

The minimax algorithm's basics stay the same – however, for the project you are not required to construct the minimax tree in a different step, and can perform your calculations while creating the tree (or even recursively, without creating an actual tree structure).

In addition, we add **pruning** to the algorithm.
The minimax algorithm scans all possible moves for each game – sometimes more than necessary.
For example, suppose a MAX node has a child node for which we finished calculating values, getting a value of 5. When we proceed to calculate the values for its next node, suppose we found a node with a value of 4 under it (a grandchild of our MAX node). The parent node of it (a MIN node, a child of our MAX node) will always choose the lowest possible value, and thus won't choose a value which is greater than that 4.
This means we can stop evaluating that branch of the tree – it will contain a value of at most 4, and thus smaller than the 5 we already found, and we can proceed to the next nodes 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. A description of pruning along with pseudo-code can be found here:
http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

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, sound, and input devices. It is widely used due to its simplicity. Over 700 games, 180 applications, and 120 demos have been posted on its website.

SDL is actually 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 1.2**. This is important as most Google results direct us to the documentation of SDL 2.0.
The documentation for SDL 1.2 is a good starting point for any question regarding using SDL, and can be found here: http://www.libsdl.org/release/SDL-1.2.15/docs/index.html

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.

To compile a program using SDL we need to add the following flags to the GCC calls inside our makefile (including the apostrophes):

- For compilation we need to add `` `sdl − config— cflags` ``
- For linkage we need to add `` `sdl − config − −libs` ``

In our code files we need to include $SDL.h$ and $SDL\_video.h$, which will provide us with all of 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 $SDL\_Init(SDL\_INIT\_VIDEO)$.

After initializing, we can start issuing calls to SDL to perform various operations for us.
For example, to create a window we'll call:

$$SDL\_WM\_SetCaption("title1","title2");$$
$$SDL\_Surface * w = SDL\_SetVideoMode(640,480,0,0);$$

This will create a window and store it in variable w, with the first line setting the title, and the second line setting the resolution and creating the window.

After drawing to an SDL window – no results are shown. This is because everything we draw is stored to a buffer. To show it, we need to flip the buffer using $SDL\_Flip(SDL\_Surface *)$.

SDL also handles events – keyboard, mouse or window events can be raised by SDL. To handle these events, we need to poll SDL and check any waiting events.
We do this in the following way:

$$SDL\_Event\ e;$$
$$while\ (SDL\_PollEvent(\&e))! = 0)\ \{/*\ handle\ event\ e\ */\}$$

The type SDL_Event is a union of all possible events. To determine the type of event, we need to check its member $e.type$.
Sometimes we'd like to wait for a few milliseconds – especially when polling for events, in order to better utilize our processor. In order to do this, SDL provides us with a *sleep* function - $SDL\_Delay(ms)$.

Provided is an example SDL source file *sdl_test.c* showing an example of SDL usage with various SDL function calls (along with initialize and quit). You can use this file as the basis for your SDL program.

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

Files

Each game in the program can be saved at any point, and then later loaded to resume playing from that point.
To save a game, the user selects the relevant option while playing, and a dialog is presented asking the user where to save the game to.
Games can be loaded from the main menu. To load a game, the user selects the relevant option from the main window, and a dialog is presented asking the user where to load the game from.

Since SDL doesn't directly provide us with the ability to output text conveniently, and to prevent further complications, the user won't be able to directly select a file to save or load. The program will use **7** "slots" for saving and loading (use a constant in your code, don't assume it's always 7!).
Each slot will be associated with a pre-determined file, and the user will only choose the slot he wishes. The file name is not visible to the user from the program itself, but the file itself should be easy to find (somewhere in the project's folder).

This means that the save/load dialogs can be easily constructed from the same actual window, with a simple Boolean flag indicating save/load. However, it is not mandatory to implement it this way.

When the user selects save/load, he'll be presented with a dialog as described above. The user will select one of the slots, and the corresponding file will be loaded from / saved to.

Files Format

Each file contains the following information:

- Next turn – which player should play next (*black* or *white*)
- Game mode – 1 (two players) or 2 (player vs. AI).
- Users color and difficulty (relevant for player vs. AI mode)
- The board's state

The input/output files are in XML format. XML is stands for Extensible Markup Language (XML) - a markup language that defines a set of rules for encoding documents in a format

that is both human-readable and machine-readable. Xml files can be easily viewed in most browsers, by typing the path in the address line. They can be easily manipulated using notepad++.
The first line in the file is the XML decleration and should be the same in each file you create.
All the stored data is represented with markups and their content.
In the following example: <game_mode>2</ game_mode >, 2 is the content, and the markup contains 2 tags: opening: < game_mode > and closing </ game_mode >. In our XML files we will use the markups to represent the feature name, and the content will be the value.
The XML file for the program has the following structure (the order of the tags should be the same as in the example)

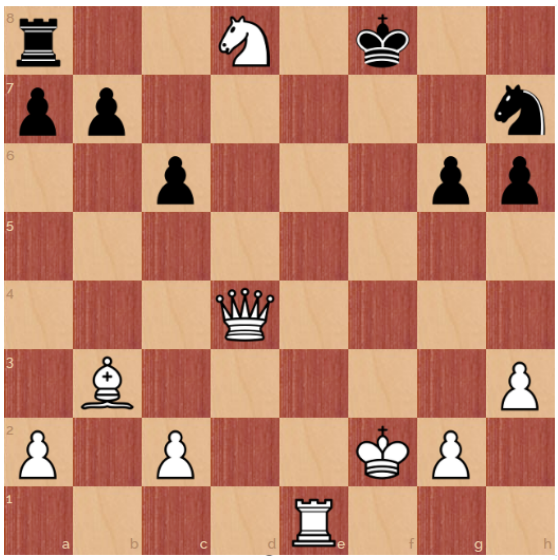
```
<?xml version="1.0" encoding="UTF-8"?>
<game>
	<next_turn>next</next_turn>
	<game_mode>mode</game_ mode >
	<difficulty>difficulty</difficulty>
	<user_color>color</user_color>
	<board>
		<row_8>8 characters that represent the row's content</row_8>
		<row_7>8 characters that represent the row's content</row_7>
		<row_6>8 characters that represent the row's content</row_6>
		<row_5>8 characters that represent the row's content</row_5>
		<row_4>8 characters that represent the row's content</row_4>
		<row_3>8 characters that represent the row's content</row_3>
		<row_2>8 characters that represent the row's content</row_2>
		<row_1>8 characters that represent the row's content</row_1>
	</board>
</game>
```

You must replace all the values in boldface with their actual values. Notice that user_color and difficulty are optional.
The possible values are:

- mode: 1 or 2
- difficulty: the numbers 1 – 4 and the option “best”, i.e. <difficulty>1</difficulty> or <difficulty>best</difficulty>
- user_color/next_turn: Black or White.
- for "2 players" mode, the tags <difficulty> and <user_color> should not contains any value.

The representation of each row is the console representation of each piece in the row, except for the empty squares, which will be represented with "_".
For example
The following board is represented by:

```
<board>
      <row_8>R__n_K__</row_8>
      <row_7>MM_____N</row_7>
      <row_6>__M___MM</row_6>
      <row_5>________</row_5>
      <row_4>___q____</row_4>
      <row_3>_b_____m</row_3>
      <row_2>m_m__km_</row_2>
      <row_1>____r___</row_1>
</board>
```



This picture is taken from:
http://www.ideachess.com/chess_tactics_puzzles/checkmate_n/

An example of an input file is attached to the project.

Additional information saved in XML:

- All additional information you choose to add to the XML should be saved under <game><general> (<general> is a new label under <game>). You can add as many labels as you need inside <general>.
- <general> tag should appear after the <board> tag

Project Material

Provided files:

- ***sdl_test.c*** – an example SDL program which creates a window, draws a rectangle and bitmap onto it, and handles keyboard and mouse events.
- ***makefile skeleton*** – building the test SDL program.
- ***a few input files examples***

The first thing you should do after reading this document is to compile using the makefile, then create your project's main function and structure, update makefile accordingly, and compile again for your project with the updated makefile.
It will be much easier to solve makefile problems at the beginning than after you have wrote some more code.

Error Handling

Your code should handle all possible errors that may occur (memory allocation problems, safe programming, SDL errors, etc.).
Files' names can be either relative or absolute and can be invalid (the file/path might not exist or could not be opened – 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 (as usual - do not exit unless you must).

Throughout your program do not forget to check:

- The return value of a function. You may excuse yourself from checking the return value of $printf$, $fprints$, $sprint$, 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's an error, print to the console – which is still available even if we're running a program with a GUI. You should output a message starting with "ERROR:" and followed by an appropriate informative message. The program will disregard the fault command and continue to run.

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.

Coding Requirements:

1. Your code should be divided into several c files, at least:
    - The minimax algorithm code
    - The Chess logic (getting moves, promoting the game state, etc)
    - The Game flow (interacting with the user, settings/game states, etc)
    - Gui
2. You should use structs to implements entities in the program. Use a struct at least for:
    - Location in the board.
    - Game move.
3. While generating possible moves for user, you must use a linked list. You can implement the move struct as a list node, or use another struct for list node and set the data type to be move.

In your implementation, also pay careful attention for use of constant values and use of memory. Do not forget to free any memory you allocated, and quit SDL.
Especially, you should aim to allocate only necessary memory and free objects (memory and files) as soon as it is possible.
Source files should be commented at critical points in the code and at function declarations.
In order to prevent lines from wrapping in your printouts – please avoid long code lines.

Submission Guidelines

Please check the course webpage for updates and Q&A.
Any questions should be posted to the course forum

The assignment zip file includes some submission file examples and input/output examples. Questions regarding the assignment can be posted in the Moodle forum. Also, please follow the detailed submission guidelines carefully.

Moodle Submission:

This assignment will be submitted in Moodle.
You should submit a zip file named ***username_project.zip***, username stands for you Moodle username.
The zipped file will contain.

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

Compilation

A skeleton makefile is provided, compiling and linking $sdl\_test.c$ with SDL libraries. Notice that you should **replace** the main function provided with your own.
You will add your files to the makefile. Use the flags provided for you to link to SDL. Your project should pass the compilation test with no errors or warnings, which will be performed by running **make all** command in a UNIX terminal windows using your updated and submitted makefile.

Appendix – casteling move

The requirements for casteling:

1. The king and the chosen rook are on the player's first rank (row).
2. Neither the king nor the chosen rook have previously moved.
3. There are no pieces between the king and the chosen rook.
4. The king is not currently in check.
5. The king does not pass through a square that is attacked by an enemy piece.
6. The king does not end up in Check. (True for any legal move.)

The move:

Casteling consists of moving the king two squares towards a rook on the player's first rank, then moving the rook to the square over which the king crossed, which is the square on the other side of the king.

In get_moves this move will be printed for both the rook and the king. The move will be printed the following way:

castle <x,y>

where <x,y> is the rook's position

Good Luck!