Team 2

Chess Please

Software Specifications V 0.0.2

Team Kingsmen

Kim | Bonecutter | Chang | Wu | Matosian | Bagade

The University of California, Irvine, The Henry Samueli School of Engineering

Table of Contents:

Glossary:

Chess.c/h:

- Piece Struct: A generalized structure that can represent any piece in the game according to the values stored in its member variables: int type, int val, char color, bool hasMoved, and bool LongMove.
- Player Struct: A structure that keeps track of the players through member variables: char color and bool comp.
- Board Struct: A structure that represents the game_board and keep track of the game and the players through the member variables: Player player1, Player player2, and Piece * board [8][8].

Opponent.c/h:

- Minimax Algorithm: Implemented with a recursive function. It will search all possible move branches within n turns and returns the parent node of the optimal branch for the active player.

Move.c/h:

- MoveList: A structure that stores moves from user input. Has member variables: MoveEntry * First, MoveEntry * Last, and int ListLength.

- MoveEntry: A structure for linked list structure. Has member variables: MoveEntry * Prev, MoveEntry * Next, and MoveVector * Vector.

- MoveVector: A structure for storing the attributes of each moves. Has member variables: int PieceType, bool Player, Vector * StartPoint, Vector * EndPoint, and bool Capture.

- Vector: A structure for storing a [x][y] position from a Board Structure.

Main.c:

- Displays the ASCII board or the GUI depending on user inputs

- Allows you to load and save game

- Has three levels of difficulty for AI when in one player mode. Also allows two player mode between two users.

1. Software Architecture Overview

1.1 Main Data Types and Structures

Chess.c/h:

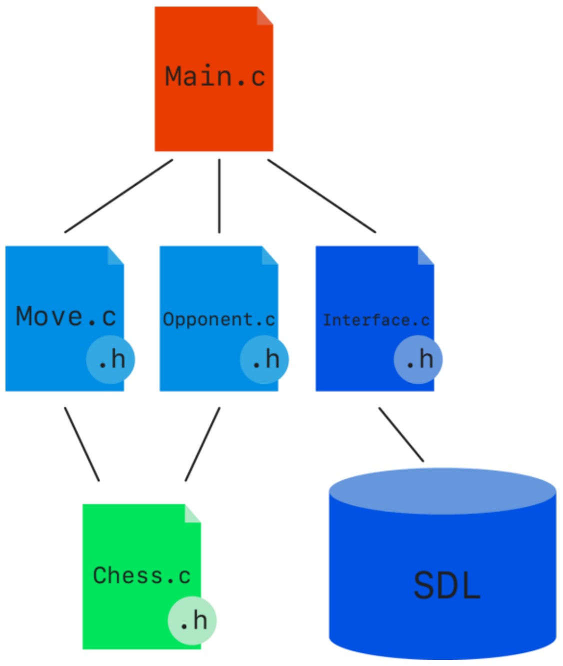
- PIECE Struct: This structure is generalized to allow us to input variables to distinguish which kind of chess piece is on the board, or if it is an empty space on the board.
- PLAYER Struct: This structure is meant to hold the values of a player. This way we can tell what color a player is and if they are a computer.
- BOARD Struct: This structure is to create a game board that the players can play on. It keeps track of the first player, the second player, and a 2D 8 by 8 PIECE pointer array. The array represents an actual chess board that we can access a location on by the x and y coordinates. It also includes EdSwitch flag for the EnPassant condition.

Move.c/h:

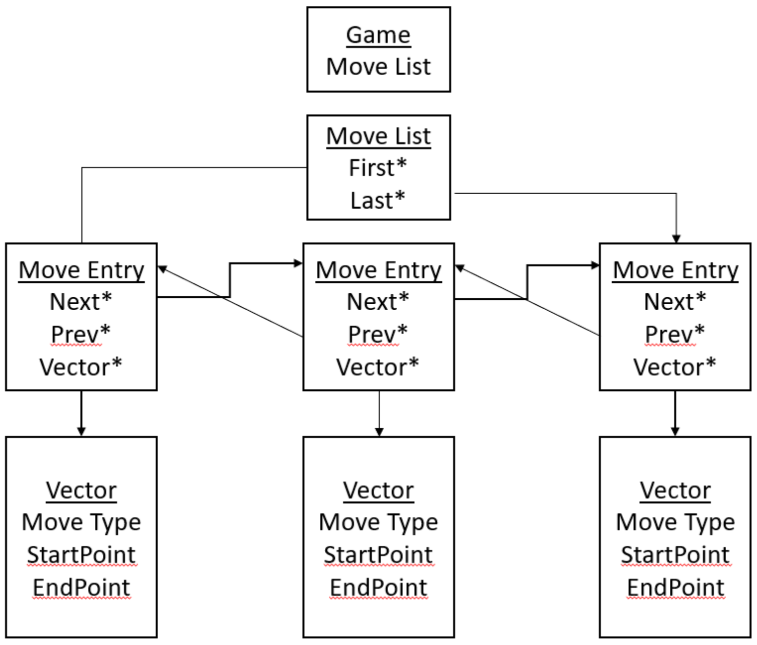
- MoveList struct
    - Member variables: int ListLength, MoveEntry* First, MoveEntry *Last
- MoveEntry struct
    - Member variables: MoveEntry* Next, MoveEntry* Prev MoveVector* vector
- MoveVector struct
    - Member variables: PieceType* piecetype, Vector* StartingPoint, Vector* EndPoint, Bool Capture, Bool Promotion, Bool EnPassant, Bool Castling
- Vector struct:
    - Member variables: int x and int y

1.2 Major Software Components

Model Hierarchy:



MoveList:



1.3 Module Interfaces

Chess.c/h:

- `Piece * CreatePiece(void)`
    - Allocates memory for a piece variable
- `void DeletePiece(Piece * deleteMe)`
    - Frees the memory allocated to prevent memory leaks
- `void SetPiece(int type, int val, char color, bool hasMoved, bool longMove, int repetition, int steps [6], Piece * piece)`
    - Assigns a set of given arguments to the piece variable
- `Player * CreatePlayer(void)`
    - Allocates memory for a player variable
- `void DeletePlayer(Player * deleteMe)`
    - Frees memory for a player variable to avoid memory leaks
- `void SetPlayer(char color, bool comp, bool active, Player * player)`
    - Takes a given set of arguments and assigns them to a PLAYER variable
- `Board * CreateBoard(void)`
    - Allocates memory for a BOARD variable
- `void DeleteBoard(Board * deleteMe)`
    - Frees the memory for a BOARD variable to avoid memory leaks
- `void InitializeBoard(Board * board)`
    - Sets all the pieces on the board to start a new game and can be used to reset the game
- `Piece * GetPiece(unsigned char x, unsigned char y, Board * board)`
    - Takes any coordinates from the board and will return a piece pointer pointing to the piece

Opponent.c/h:

The opponent module is dedicated to the computer AI portion of Chess Please. It has only three functions and no typedefs or structs. The major function is defined as follows:

- `Move *makeBestMove(moveList *possibleMoves, Board *currBoard, bool activePlayer, unsigned char depth)`

    - Takes in the list of possible moves, the current chess board, which players' turn it is, and how far it should search and returns the current optimal move
- `short int FindTotalScore(Board * board)`
    - Calculates the total score (minimax) of the board for the Opponent.c file

Move.c/h:

- `MoveList* CreateMoveList(void )`
    - This function creates MoveList and returns the pointer.
- `void DeleteMoveList(MoveList* list)`
    - This function deletes the MoveList.
- `MoveEntry* DeleteLastEntry(MoveList * list)`
    - This function deletes the last entry of the MoveList and returns the new last MoveEntry.
- `MoveEntry* CreateMoveEntry(MoveVector* vector)`
    - This function creates a new MoveEntry in the Movelist.
- `void AppendMove(MoveList* list, MoveVector *vector)`
    - This function appends new MoveEntry with the MoveVector in the MoveList
- `bool OutofBoard(MoveVector * vector)`
    - This boolean function returns true if the piece moved out of the board.
- `bool IsValidMove(MoveVector *vector, Board *CurrBoard)`
    - This function includes below sub-functions. The purpose of this function is to check the validity of the move:
    - Returns true if the king made a move shown in white. X represents the horizontal coordinate of the piece. Y represents the vertical coordinate of the piece.
    - `bool King(MoveVector *vector, Board *CurrBoard)`

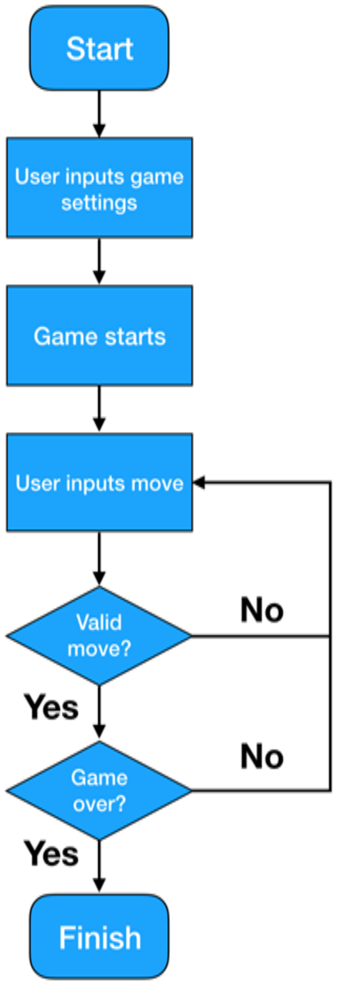
        - This function returns true if the endpoint fit the logic that X+/-1 || Y+/-1.
    - `bool Queen(MoveVector *vector, Board *CurrBoard)`
        - This function returns true if the endpoint fit the logic that X++/-- && Y++/--- at the same time or X++/-- or Y++/--.
    - `bool Rook(MoveVector *vector, Board *CurrBoard)`
        - This function returns true if the endpoint fit the logic that only X++ or Y++
    - `bool Bishop (MoveVector *vector, Board *CurrBoard)`
        - This function returns true if the endpoint fit the logic that X++/-- && Y++/--- at the same time.
    - `bool Knight(MoveVector *vector, Board *CurrBoard)`
        - This function returns true if the endpoint fit the logic that X+/-1 and Y+/-2 or Y+/-1 and X+/-2
    - `bool Pawn(MoveVector *vector, Board *CurrBoard)`
        - This function returns true if the endpoint fit the logic that Y+/-1 or 2.
- `bool Capture(MoveVector *vector, Board *CurrBoard)`
    - This function returns true if the move captured enemy chess.
- `bool Block(MoveVector *vector, Board *CurrBoard)`
    - This function returns true if the move is not valid because of block
- `bool IsGameOver(Board * board)`
    - Checks for checkmate and draw conditions and will return true if any of these conditions are met
- `void UpdateBoard(Board *CurrBoard, MoveEntry * MoveEntry)`
    - This function from a piece current location to its next location and sets an empty space in its previous location. It deals with special cases like castling and promotion.

Main.c:

- `Void DisplayBoard (Board *GameBoard)`

    - Prints the game board with current chess piece positions in ASCII format.
- `int whoWon(Board *GameBoard)`
    - Pass in the game board as the argument, then return 0 (false) if player 1 has won, or return 1 (true) if player 2 has won.
- `void SaveFile(moveList *MoveList, char *fname)`
    - Pass in 3 arguments : the file name, log of moves, and game board. This function will save the current game board and the log of moves, so that the user can save the current game and play it later.
- `Board LoadFile(Movelist * movelist, char *fname)`
    - Pass in the file name as the argument to load the corresponding game the user wants to play.
- `void FillBoardSpace(int x, int y, SDL_Rect board[][8], Uint32 light, Uint32 dark, SDL_Surface * screen)`
    - Checks the space of a rectangle on the board and fills in the appropriate color. Also calls SDL_UpdateRect.

1.4 Overall Program Control Flow



2. Installation

2.1 System Requirements, Compatibility

- CentOS - Release 6.9 (Final)
- Hardware Requirements:
  - Linux Enabled Machine
  - > 1MB of Storage
  - > 1GB RAM

2.2 Setup and Configuration

- Enter the linux command line and enter `tar -xzf chess_please.tar.gz`

2.3 Building, Compilation, Installation

- If you are running linux in a shell, ensure that you have connected to the server with the `-X -Y` options or have connected with an X-11 enabled client
- Type `./ChessPlease_1.0` in your terminal

3. Documentation of Packages, Modules, Interfaces

3.1 Detailed Description of Data Structure

- Critical snippets of source code
- The Minimax Algorithm implemented as findBestMove:

```
bestMove = -10000;

for(int i = 0; i < length; i++)

  {

     while(!isValidMove(currMove) && try_count++ < length)

           currMove = currMove->Next;

     currScore = findTotalScore(UpdateBoard(testBoard,
     findBestMove(currBoard, false, depth - 1));

     testBoard = prevBoard();

     if(currScore > bestScore)

     {

      bestScore = currScore;

      bestMove = currMove;

           }

}

DestroyMoveList(possibleMoves);

return bestMove;

      //And a similar case is mirrored for the opposite player

      //In that case bestMove will be set to 10000 and bestScore

      //will

      //be replaced with current score if currScore is less than

      //bestScore
```

- Data Types for Chess.c/h:

```
typedef struct piece {

    int type;

    int val;
```

```
        char color;
        bool hasMoved;
        bool longMove;
    } Piece;
    // Contains the type of piece, the 'value' of the piece, its
    // color, whether it has moved, and if it can move more than 1
    // space.
    typedef struct player {
        bool color;
        char comp;
    } Player;
    // Contains the color of the pieces the player controls and
    // whether the player is a computer (AI) or not.
    typedef struct gameboard {
        Player * Player1;
        Player * Player2;
        Piece * game_board [8][8];
    } Board;
    // Contains a 2D array of pointers of Piece structs to allow for
    // a quick navigation while also using pointers.
```

- Data Types for Move.c/h:

```
    typedef struct MoveList   MoveList;
    typedef struct MoveEntry  MoveEntry;
    typedef struct MoveVector MoveVector;
    typedef struct ChessGame  Game;
    typedef struct Vector     Vector;
    struct MoveVector{
        int Piecetype;
        bool Player;
    Vector *Startpoint;
    Vector *Endpoint;
    bool Capture;
}
// Contains the stratpoint, the endpoint, the type of piece, and
// the player of any move.
struct MoveEntry{
    MoveEntry  *Next;
    MoveEntry  *Prev;
    MoveVector *Vector;
}
// Points to a MoveVector, next MoveEntry, and previous
// MoveEntry in a doubly linked list.
struct MoveList{
    int ListLength;
    MoveEntry *First;
    MoveEntry *Last;
}
// Contains an integar pertaining to the length of the list and
// the first and last entries of the list.
struct Vector{
    int x;   // x and y position
    int y;
}
// Contains a pairing of the x and y coordinates of a Board
// struct.
struct Game{
    MoveList *NewList;
}
```

```
// Contains a list of all possible moves.
```

3.2 Detailed Description of Functions and Parameters

Chess.c/h:

- `Piece * CreatePiece(void)`
  - This function uses `void* malloc(sizeof(Piece))` to allocate memory for the size of a Piece variable. Then it will assign this location in memory to a Piece pointer. After, it sets all the variable values to that of an empty space on the board. Finally, it returns the Piece pointer.
- `void DeletePiece(Piece * deleteMe)`
  - First this function uses `void assert(int expression)` to verify that the argument Piece pointer (deleteMe) is not pointing to NULL. If it is not NULL, then it frees the memory at the location stored in the pointer with `void free(void * ptr)`. Lastly, the function will set the pointer to NULL.
- `void SetPiece(int type, int val, char color, bool hasMoved, bool longMove, int repetition, int steps [6], Piece * piece)`
  - This function takes a set of values and a Piece pointer, and assigns the given values to the piece the pointer is pointing to.
- `Player * CreatePlayer(void)`
  - The CreatePlayer function uses `void * malloc(sizeof(Player))` to allocate memory for the size of a Player variable. Then it assigns this location in memory to a Player pointer. Next, it sets all the variable values to the default color white and non-computer input. Lastly, it returns the Player pointer.
- `void DeletePlayer(Player * deleteMe)`
  - This function uses `void assert(int expression)` to verify that the argument Player pointer (deleteMe) is not pointing to NULL. If it is not NULL, then it frees the memory at the location stored in the pointer with `void free (void * ptr)`. After, the function will set the pointer to NULL.
- `void SetPlayer(char color, bool comp, PLayer * player)`
  - This function is used to set the Player variables. Because when we create the player, we set everything to default, this function

will take arguments for color and comp and assigns it to the specified player.

- `Board * CreateBoard(void)`
  - This function uses `void * malloc(sizeof(Board))` to allocate memory for the size of a Board variable. Then it assigns this location in memory to a Board pointer. After, it creates new players using the CreatePlayer function and stores it at the location of the player variable in Board. It also creates a new piece at every location and stores it in the 2D 8 by 8 array. Finally, it returns the Board.
- `void DeleteBoard(Board * deleteMe)`
  - This function uses `void assert(int expression)` to the passed in Board pointer (deleteMe) to check if it is pointing to NULL. Then it frees the memory at the location stored in the pointer with `void free (void * ptr)`. And after, it will set the pointer to NULL.
- `void InitializeBoard(Board * board)`
  - The InitializeBoard function calls the SetPiece function to create the pieces for a typical game setup and assigns the pieces to a location in an array. The 2D array we previously set up represent game boar. To refer to the game board we use a grid that begins at the bottom left corner where the x-axis starts 1 and goes up to 8 and the y-axis starts at A and moves to the right towards H.
- `Piece * GetPiece(unsigned char x, unsigned char y, Board * board)`
  - This function returns a pointer pointing to a piece on the board at the given x and y coordinate.

Opponent.c/h:

- `bool isOpener()`
  - Use a static variable to non intrusively keep count of how many moves have been made. A basic if control structure will return `true OR false` depending on how many turns have passed. However, this function will become more advanced as the program becomes closer to completion
- `Move *GiveOpeningMove()`

    - Holds a sequence of predefined opening moves in order to allow for the best initial setup. It will progress through this sequence and return the respective move
- `Move *FindBestMove(...)`
    - This functions uses a recursive control structure to implement the minimax algorithm to navigate through the tree of possible moves within n turns in the current game. It will return the move that yields the best position after n turns for the current player.
- `short int FindTotalScore(Board * board)`
    - This function takes in a board, and by iterating through two nested for loops, it adds up the values of the pieces in order to calculate the total score for the board. This is used for the minimax algorithm.

Move.c/h:

- `MoveList *CreateMoveList()`
    - Uses malloc() to allocate the memory for a new MoveList struct. It returns a pointer to the allocated space
- `Void DestroyMoveList(MoveList *list)`
    - Uses free() to de-allocate the memory space for a MoveList and DestroyMove(Move *move) to delete the memory space for all the move objects within the list
- `MoveEntry* DeleteLastEntry(MoveList * list)`
    - Use free() to de-allocate the memory space for the last MoveEntry of its parent MoveList and set the pointer to that MoveEntry to NULL and return the point MoveList->Last*.
- `MoveEntry* CreateMoveEntry(MoveVector* vector)`
    - Use malloc() to allocate the memory space for a new MoveEntry struct for each legal move. It returns a pointer to the allocated space.
- `void AppendMove(MoveList* list, MoveVector *vector)`
    - Appends a new MoveEntry to the Movelist as its last Entry. MoveList -> Last will be updated with the new MoveEntry.
- `void UpdateBoard(Board * currBoard, Move * move)`
    - Moves the piece from the p0 position within move to the p1 position and erases its previous position. Contains a special case for castling where both the king and rook are moved to their respective positions. Another special case exists for the

promotion of a pawn where the pawn is deleted and the piece of choice is spawned on the board.

Main.c/h:

- `Void DisplayBoard (Board *GameBoard)`
    - This function will take in the *struct board* data type as its parameter, which contains a 2-D array of *struct piece* as its member variable. This function will iterate through the 2-D array and print out the location of the pieces.
- `int whoWon(Board *GameBoard)`
    - This function will be called once either King is in checkmate. It will iterate through the game board and check which King, black or white, is in checkmate. Then it will return 0 if player2(black)'s King is in checkmate, or return 1 if player1(white)'s King is in checkmate.
- `void SaveFile(moveList *MoveList, char *fname)`
    - This function will use the `fopen()` to open the text file that will store the information to be saved. Then `fprintf()` will be used to print the information on the text file.
- `Board LoadFile(Movelist * movelist, char *fname)`
    - This function will use `fscanf()` to read from the text file that contains the information of the previously saved games. Then a new `struct game` will be created and the saved list of moves in the text file will be copied to it. Then the function will return the `game.`
- `void FillBoardSpace(int x, int y, SDL_Rect board[][8], Uint32 light, Uint32 dark, SDL_Surface * screen)`
    - Uses if statements and module to find location on board through (x,y) coordinates and assigns the correct color according to the location on the board. Afterwards it updates the rectangle onto the screen.

3.3 Detailed Description of Input and Output Formats

Syntax/format of a move input by the user

- The user will input the current coordinate of the piece that he/she wants to move, then input the coordinate he/she wants to move it to.

  - Ex) `Enter your move :` e2e4
  - This will move the piece that is at coordinate e2 to coordinate e4, if the move is possible.

Syntax/format of a move recorded in the log file

- Everytime a player makes a move, the move is going to be saved in a text file in the order that it has been inputted.
  - Ex) 1. e2e4 2. g7g5 3. d8a5 4. a2a4

4. Development Plan and Timeline

4.1 Partitioning of Tasks

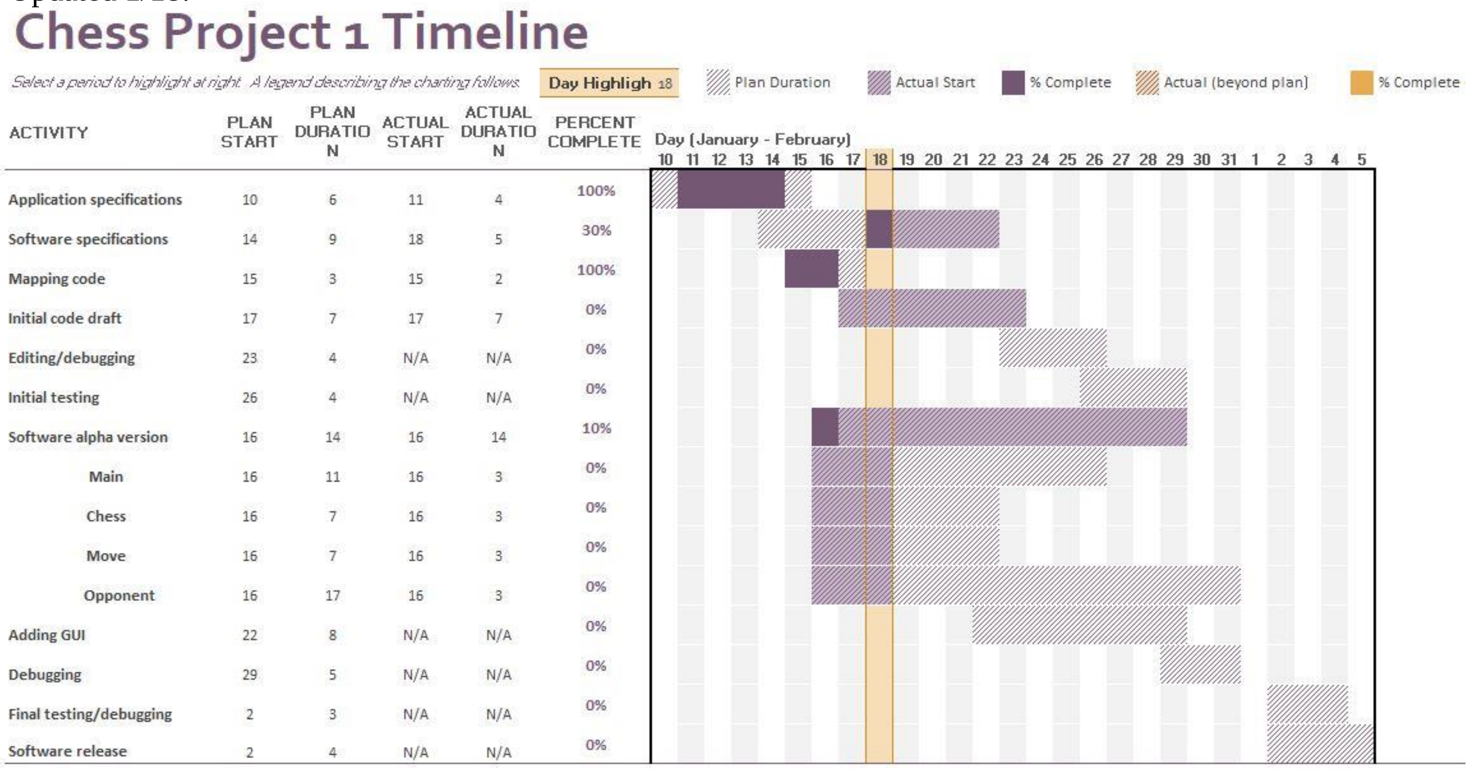
- Chess Please will be composed of 5 modules: Main.c/h, Chess.c/h, Move.c/h, Opponent.c/h, and Interface.c/h . Each module will be overseen by two team members as specified below.

4.2 Team Member Responsibilities

- Arineh, Kaveri:
    - Chess.c/h
    - Document Formatting
    - Timeline
    - GUI
- Jae:
    - Main.c
- Handing, Chi:
    - Move.c/h
- Jordan:
    - Opponent.c/h
    - Interface.c/h
    - Diagrams

4.3 Timeline

Updated 1/18:



Chess Project 1 Timeline

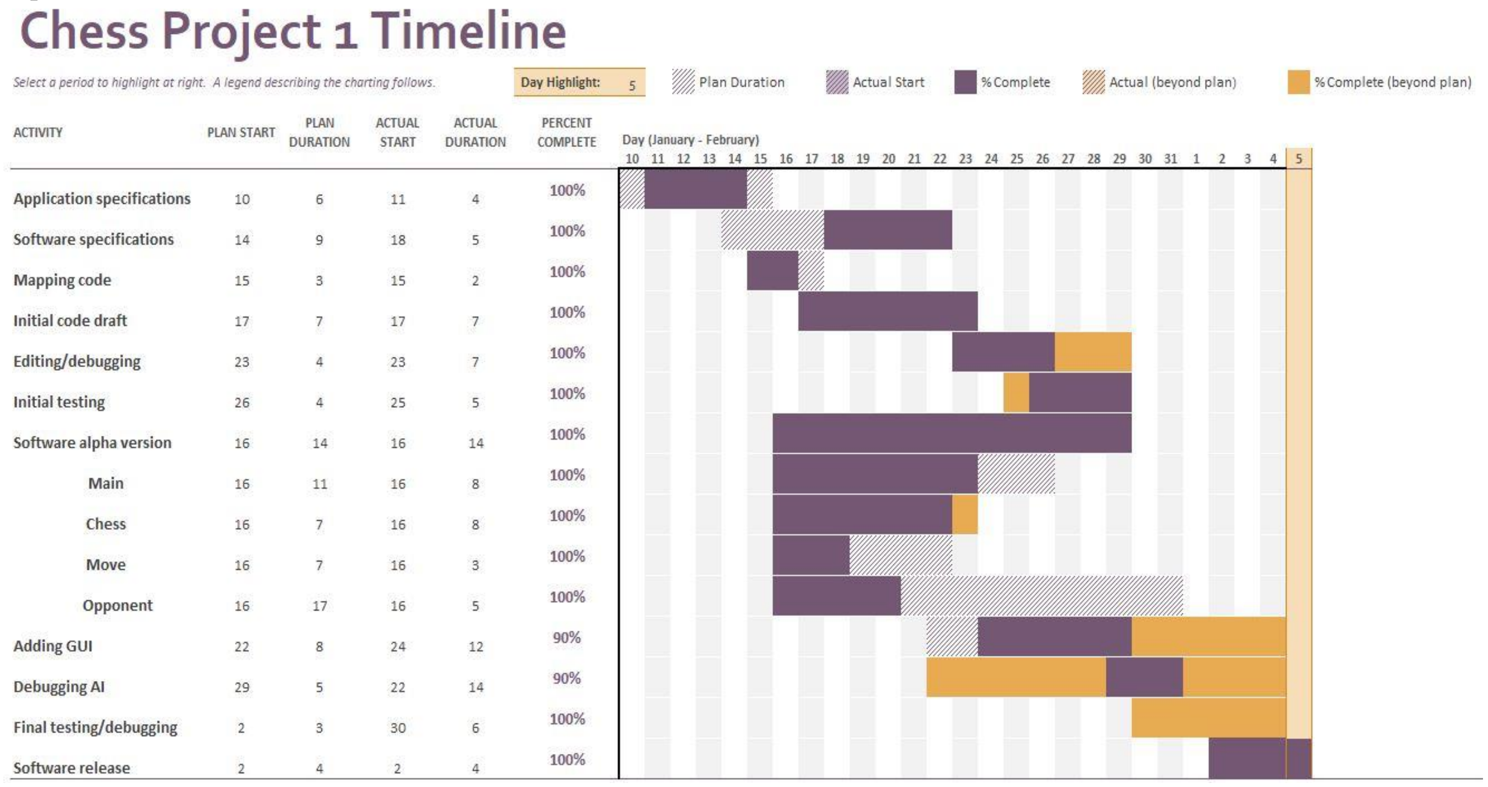
Select a period to highlight at right. A legend describing the charting follows.

Day Highligh 18 | ▨ Plan Duration | ▨ Actual Start | ■ % Complete | ▨ Actual (beyond plan) | ■ % Complete

Day (January - February) 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 1 2 3 4 5

ACTIVITY	PLAN START	PLAN DURATIO N	ACTUAL START	ACTUAL DURATIO N	PERCENT COMPLETE
Application specifications	10	6	11	4	100%
Software specifications	14	9	18	5	30%
Mapping code	15	3	15	2	100%
Initial code draft	17	7	17	7	0%
Editing/debugging	23	4	N/A	N/A	0%
Initial testing	26	4	N/A	N/A	0%
Software alpha version	16	14	16	14	10%
Main	16	11	16	3	0%
Chess	16	7	16	3	0%
Move	16	7	16	3	0%
Opponent	16	17	16	3	0%
Adding GUI	22	8	N/A	N/A	0%
Debugging	29	5	N/A	N/A	0%
Final testing/debugging	2	3	N/A	N/A	0%
Software release	2	4	N/A	N/A	0%

Updated 2/5:



Chess Project 1 Timeline

Select a period to highlight at right. A legend describing the charting follows.

Day Highlight: 5 | ▨ Plan Duration | ▨ Actual Start | ■ % Complete | ▨ Actual (beyond plan) | ■ % Complete (beyond plan)

Day (January - February) 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 1 2 3 4 5

ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Application specifications	10	6	11	4	100%
Software specifications	14	9	18	5	100%
Mapping code	15	3	15	2	100%
Initial code draft	17	7	17	7	100%
Editing/debugging	23	4	23	7	100%
Initial testing	26	4	25	5	100%
Software alpha version	16	14	16	14	100%
Main	16	11	16	8	100%
Chess	16	7	16	8	100%
Move	16	7	16	3	100%
Opponent	16	17	16	5	100%
Adding GUI	22	8	24	12	90%
Debugging AI	29	5	22	14	90%
Final testing/debugging	2	3	30	6	100%
Software release	2	4	2	4	100%

Copyright

A list of references used in the making of the program.

- Glossary Definitions/Rules:
  - https://en.wikipedia.org/wiki/Draw
  - https://eee.uci.edu/18w/18020/tas/letsplay.pdf
- Artificial Intelligence:
  - https://medium.freecodecamp.org/simple-chess-ai-step-by-step-1d55a9266977
  - https://en.wikipedia.org/wiki/Minimax#Pseudocode
- SDL References:
  - http://lazyfoo.net/SDL_tutorials/index.php
  - https://www.libsdl.org/
  - https://www.libsdl.org/release/SDL-1.2.15/docs/html/reference.html
  - https://www.youtube.com/user/thecplusplusguy/playlists

Disclaimer:

Chess Please was produced in 2018 at the University of California, Irvine by Team Kingsman. All rights are reserved.

Index