

TWO PLAYER GAMES - CHECKERS

Project 2 – CS5346: Advanced Artificial Intelligence

Abstract

This report examines the design, implementation, and execution of an expert system, which simulates a series of checkers games. Report covers how the program employs course concepts and how it makes use of data structures and algorithms.

Author: Borislav S. Sabotinov
bss64@txstate.edu | TXST Student ID: A04626934

Team members: David Torrente and Randal Henderson

Table of Contents

1	Introduction	6
1.1	Purpose of Report	6
1.2	System Scope	6
1.3	Terms and Definitions	7
1.4	Report Overview	8
1.5	Contributions	9
1.5.1	Individual Contributions	9
1.5.2	Group Member Contributions	10
2	System Description	11
2.1	System Perspective	11
2.2	System Features	11
2.3	Design and Implementation Constraints	11
2.4	Assumptions and dependencies	12
3	System Design and Specification	12
3.1	How to Build and Run the Program	13
3.2	Why use C++11?	15
3.5	System Class Diagram	16
3.6	Analysis of Results	18
3.6.1	How Good Are the Results?	18
3.6.2	Memory and Speed	18
3.6.3	Optimizations	24
3.7	Evaluation Functions	26
3.7.1	Evolution of Evaluation Function Three (EF-3)	26
3.7.2	Final state of EF-3	36
3.7.3	Evaluation Function One (EF-1)	46
3.7.4	Evaluation Function Two (EF-2)	46
3.8	UI Design Considerations	47
4	Classes Deep-Dive	50
4.1	Pieces	50
4.1.1	Data Structures Used	52
4.2	Board	52
4.2.1	Data Structures Used	54
4.3	Player	57
4.3.1	Data Structures and Algorithms	59
4.4	Game	59
4.4.1	Data Structures and Algorithms	61

4.5 Algorithm	61
4.5.1 Data Structures and Algorithms.....	63
4.6 Simulation	66
4.6.1 Data Structures and Algorithms.....	66
4.7 Main	67
5 Sample Runs.....	68
5.1 Sample Run #1: ALPHA-BETA-SEARCH with EF-1 vs ALPHA-BETA-SEARCH with EF-3 at Depth 2	69
5.2 Sample Run #2: ALPHA-BETA-SEARCH with EF-1 vs ALPHA-BETA-SEARCH with EF-3 at Depth 4	70
5.3 Sample Run #3: Minimax-a-b with EF-3 vs Alpha-Beta-Search with EF-3 at Depth 6 (extra).....	72
5.4 Sample Run #4: ALPHA-BETA-SEARCH with EF-1 vs ALPHA-BETA-SEARCH with EF-3 at Depth 8 (extra).....	73
5.5 Sample Run #5: Human Player vs AI (Minmax-A-B with EF-3).....	74
5.6 Sample Run #6: Printing the Help Menu.....	75
6 Conclusion.....	76
References	77
Appendix A: Source Code.....	78
Pieces.hpp	78
Pieces.cpp	79
Board.hpp	84
Board.cpp	86
Player.hpp	115
Player.cpp.....	117
Game.hpp.....	121
Game.cpp	123
Algorithm.hpp	127
Algorithm.cpp	130
Simulation.hpp	167
Simulation.cpp	170
main.cpp	179
Appendix B: Sample Output for 18 Games	189
Game 1: (Minmax-A-B, EF-1) vs (A-B-Search, EF-1) – Depth 2.....	190
Game 2: (Minmax-A-B, EF-2) vs (A-B-Search, EF-2) – Depth 2.....	252
Game 3: (Minmax-A-B, EF-3) vs (A-B-Search, EF-3) – Depth 2.....	253
Game 4: (Minmax-A-B, EF-1) vs (Minimax-A-B, EF-2) – Depth 2	254
Game 5: (Minmax-A-B, EF-1) vs (Minimax-A-B, EF-3) – Depth 2	255
Game 6: (Minmax-A-B, EF-2) vs (Minimax-A-B, EF-3) – Depth 2	256
Game 7: (A-B-Search, EF-1) vs (A-B-Search, EF-2) – Depth 2	257
Game 8: (A-B-Search, EF-1) vs (A-B-Search, EF-3) – Depth 2	258
Game 9: (A-B-Search, EF-2) vs (A-B-Search, EF-3) – Depth 2	260

Game 10: (Minmax-A-B, EF-1) vs (A-B-Search, EF-1) – Depth 4.....	262
Game 11: (Minmax-A-B, EF-2) vs (A-B-Search, EF-2) – Depth 4.....	323
Game 12: (Minmax-A-B, EF-3) vs (A-B-Search, EF-3) – Depth 4.....	324
Game 13: (Minmax-A-B, EF-1) vs (Minimax-A-B, EF-2) – Depth 4	325
Game 14: (Minmax-A-B, EF-1) vs (Minimax-A-B, EF-3) – Depth 4	326
Game 15: (Minmax-A-B, EF-2) vs (Minimax-A-B, EF-3) – Depth 4	327
Game 16: (A-B-Search, EF-1) vs (A-B-Search, EF-2) – Depth 4	328
Game 17: (A-B-Search, EF-1) vs (A-B-Search, EF-3) – Depth 4	329
Game 18: (A-B-Search, EF-2) vs (A-B-Search, EF-3) – Depth 4	330

List of Tables

Table 1: Terms and Definitions	7
Table 2: Timing Comparison for minmax-a-b vs a-b-search; the latter slightly outperforms	22
Table 3: Results for 9 runs at depth 2	23
Table 4: results for 9 runs at depth 4.....	24
Table 5: 100mil copy test by David Torrente	25
Table 6: 30mil comparisons test by David Torrente	25

List of Figures

Figure 1: GitHub Contributions – 12 items split equally 3 ways for 4 items per team member	10
Figure 2: Main Menu.....	13
Figure 3: UML Class Diagram for the Checkers AI System	17
Figure 4: Memory profile of Minmax with EF-1 vs ABS with EF-1	19
Figure 5: Memory Profile of Minmax with EF-2 vs Minmax with EF-2	20
Figure 6: Memory Profile of ABS with EF-2 vs ABS with EF-3	21
Figure 7: CPU, RAM, and OS type used for testing	22
Figure 8: EF-3 wins against an AI player who selects the 1st move by letting the enemy block itself in a no-moves state	29
Figure 9: Welcome message	47
Figure 10: Highlighting choices to guide the user.....	47
Figure 11: A checkers board as represented on the CLI, with color enabled	48
Figure 12: Draw end state ASCII art	48
Figure 13: One player wins end state ASCII art.....	49
Figure 14: End screen.....	49

Figure 15: Pieces UML Class Node	51
Figure 16: Board UML Class Node.....	53
Figure 17: Moves and Jumps for Square 16.....	54
Figure 18: A Board with a Black move from 25 to 18 (a jump) and a capture at 22.....	55
Figure 19: Player UML Class Node	58
Figure 20: Game UML Class Node.....	60
Figure 21: Algorithm UML Class Node	62
Figure 22: Minimax-a-b Futility Cutoff.....	64
Figure 23: Alpha-Beta-Search Algorithm	65
Figure 24: Simulation UML Class Node	66
Figure 25: Main application driver.....	67
Figure 26: Sample Run 1	69
Figure 27: Sample Run 2, Turn 8	70
Figure 28: Sample Run 2 End state	71
Figure 29: Depth 6 - MAB EF-3 vs ABS EF-3, Draw.....	72
Figure 30: Sample Run 4 Depth 8!	73
Figure 31: Sample Run 5 - Manual Game Ending.....	74
Figure 32: Sample Run #6 - Print the Help Menu	75
Figure 33: Minimax-1 vs ABS-1 Depth 2.....	190
Figure 34: Minimax-2 vs ABS-2 Depth 2.....	253
Figure 35: Minimax-3 vs ABS-3 Depth 2.....	254
Figure 36: Minimax-1 vs Minimax-2 Depth 2.....	255
Figure 37: Minimax-1 vs Minimax-3 Depth 2.....	256
Figure 38: Minimax-2 vs Minimax-3 Depth 2.....	257
Figure 39: ABS-1 vs ABS-2 Depth 2	258
Figure 40: ABS-1 vs ABS-3 Depth 2	259
Figure 41: ABS-2 vs ABS-3 Depth 2	260
Figure 42: Minimax-1 vs ABS-1 Depth 4.....	262
Figure 43: Minimax-2 vs ABS-2 Depth 4.....	324
Figure 44: Minimax-3 vs ABS-3 Depth 4.....	325
Figure 45: Minimax-1 vs Minimax-2 Depth 4.....	326
Figure 46: Minimax-1 vs Minimax-3 Depth 4.....	327
Figure 47: Minimax-2 vs Minimax-3 Depth 4.....	328
Figure 48: ABS-1 vs ABS-2 Depth 4	329
Figure 49: ABS-1 vs ABS-3 Depth 4	330

Figure 50: ABS-2 vs ABS-3 Depth 4	331
Figure 51: Full Simulation of 72 total games!	332

1 Introduction

The application covered by this report serves as a two-player checkers game system using artificial intelligence to conduct the game. The application employs two main algorithms for this purpose – MINMAX-A-B as described by Rich and Knight and ALPHA-BETA-SEARCH as described by Russel and Norvig.

Checkers is a full deterministic two-player game; it currently has a mathematical “solution” that may be considered optimal. Jonathan Schaeffer and his team designed an AI algorithm, which, when pitted against an optimal opponent who does not make a mistake, ends the game in a draw. Thus, we may say that checkers is solved and if neither player makes a mistake, the game will end in a draw. Other two-player games, such as Chess and Go, also fall in this category and while they too likely have a solution, it is not yet discovered. The computational complexity of these games is larger than it is for checkers.

This system employs three different static evaluation functions to evaluate a given state of a checkers board and determine whether it is a good state for a player to move into or not. These functions are the “heart” of the AI engine in this program and are what enable the minmax-a-b and alpha-beta-search algorithms to find the “best” move a player can make. The game of checkers is evaluated using the two algorithms, three evaluation functions, and several measures of game complexity – game-tree size and complexity and decision and computational complexity to name a few.

For additional evaluation and flexibility, the program for Project 2 also offers several simulation and game options, which will be covered in detail throughout the report. In short, the application allows a user to conduct a full simulation of a series of games, a single game with custom configuration, test their skills against an AI player, or play a game manually. The application is thorough and goes beyond the scope of the requirements in some areas to provide additional results for analysis and consideration. Results obtained from the simulations show that minmax-a-b and alpha-beta-search expand the same number of nodes and reach the same conclusion, given the same initial configuration. For example, a Red player using minimax-a-b and the first evaluation function versus Black player using minimax-a-b and evaluation 2 simulation reaches the same conclusion as a simulation where both players use alpha-beta-search and the same evaluation function. This is in line with our expectations and confirm that the algorithms are correctly implemented.

1.1 Purpose of Report

The intent of this report is to demonstrate a thorough understanding of class concepts and provide a detailed explanation of the Project 2 checkers artificial intelligence application. This report will attempt to exhaustively cover every aspect of the design, implementation, execution, and analysis of the program.

1.2 System Scope

The application for Project 2 is an intelligent system for playing checkers. It uses either minmax-a-b or alpha-beta-search to determine the best move, which is evaluated and scored by three unique evaluation functions.

The system shall:

- Display a menu to the user with prompts for input.
- Read in user input for the simulation they desire and the configuration of a game, if applicable.
- Provide users the ability to print verbose output or suppress it as needed.
- Implement the required algorithms and evaluation functions as described on Canvas.
- Play a checkers game using AI through completion.
 - Either one of the players wins or the game ends in a draw after a predefined number of turns.

The system is designed to compile and run on the Texas State Linux hosts. It was additionally tested on Windows – provided certain prerequisites are met, it can execute on the Windows OS as well.

Complete code and documentation, as well as this report, may be found on GitHub here: <https://github.com/TXST-CS5346-AI/project-two>

1.3 Terms and Definitions

Table 1: Terms and Definitions

Term	Definition
The system	“The system” in this report shall refer to the application designed and implemented for Project 2, an intelligent two-player checkers game engine. The application can do a full simulation given the project requirements, a single game with a custom configuration, a player-vs-AI game, or a manual player-vs-player game.
EF	Evaluation Function, abbreviated to EF for conciseness
EF-1	Evaluation function #1, developed by Davit Torrente
EF-2	Evaluation function #2, developed by Randall Henderson
EF-3	Evaluation function #3, developed by Borislav Sabotinov
Black wins	Either the opposing player (Red) is out of moves or pieces
Red wins	Either the opposing player (Black) is out of moves or pieces
Draw	After 80 rounds, where each player has 80 moves for a total of 160 maximum allowed moves per game, end in a draw
Ply / Turn	Used interchangeably throughout the report. In a two-player game, a ply is one turn taken by one of the players.
CLI	Command line interface, a text-based user interface for interacting with this application on Texas State’s Linux servers.
ABS	Alpha-Beta-Search algorithm
MAB / Minimax	Minmax-A-B algorithm
R / r	An upper-case R shall represent a red KING piece, lower-case r will represent a MAN
B / b	An upper-case B shall represent a black KING piece, lower-case b will represent a MAN
MAN	A standard piece in checkers, also known as a Man. May only move forward diagonally.

KING	Once a player's Man reaches the opposite side of the board (i.e., opponent's back row) the piece becomes a King and can move backwards on the board.
MSB	Most significant bit
LSB	Least significant bit
OS	Operating System

1.4 Report Overview

The Table of Contents shows where each section is located and contains a list of tables and figures used throughout the report for ease of reference.

Section 1 covers general, introductory information about the system created for this project. It shows terms and definitions used in this report.

Section 2 describes the system at a high level. It defines the intended audience and the types of users served by this system, along with a list of features. Any constraints encountered, or assumptions made, during design and implementation are also outlined here.

Section 3 dives into the system design in greater detail. It shows how to build and run the program, providing a complete listing of all available commands. We examine the overall UML class diagram. We analyze the results of the program and the program's efficiency. The evaluation functions are examined; I focus on the function I designed and implemented – EF-3 – in greater detail, showing the evolution of the function over time and how I solved problems encountered. Finally, we briefly cover UI design considerations for providing a pleasant user experience.

Section 4 dives into the source code and outlines each class, as well as any noteworthy data structures or algorithms employed.

Section 5 comprehensively covers the system by exercising it against all unique available options. Due to the size of certain outputs, only screenshots of the final state of a game are provided in the project report. A complete text output is made available externally, in the **Project2_depth_#_results** directories (where # is the depth) – there are two such directories, one for depth two and another for depth four.

The appendices provide large images of all diagrams (where necessary) for readability, as well as the complete source code. Any diagram too large to be readable in its entirety is dissected into sections and each section is presented separately, expanded to fill the screen. Appendix A provides the source code while Appendix B provides sample output for all required 18 runs and lists which files contain the full output.

1.5 Contributions

1.5.1 Individual Contributions

In this project, our team ensured **all** code was equally distributed. There were 12 items (i.e., GitHub “issues” from a project management perspective) – each team member implemented 4 items equally. We designed the application together as a team and engaged in troubleshooting activities equally.

My contributions to this project include:

- Four main items out of the twelve total: Main.cpp, Simulation.hpp and Simulation.cpp, the alpha-beta-search algorithm and all of its helper functions, and evaluation function three.
- Assisted the team to collectively design the application, including how a Player takes a turn and how the main game loop runs.
- Implemented other functions in Algorithm.cpp, which is over 1K lines of code long.
- Came up with the idea to number the squares on the printed board to improve readability and with ANSI color scheme.
- Participated in group troubleshooting activities, helping to fix code issues across all headers and class implementation files.
- Created the GitHub repo and initial workspace.
- Lead the UML class diagram design – in this project, we designed up-front more than the last and this investment paid off.
- Facilitated pull requests/merges to ensure code synchronization; served as a project manager to facilitate meetings.

Please note that in this project, all team members really did contribute equally to design, implementation, and troubleshooting of the project code base.

12 Open ✓ 0 Closed			
Author	Label	Projects	Milestones
Assignee	Sort		
Game enhancement	Randy		
#15 opened 12 days ago by bss8			
Player enhancement	Randy		
#14 opened 12 days ago by bss8			
Minimax AB enhancement	Randy		
#13 opened 12 days ago by bss8			
Main enhancement	Boris		
#12 opened 12 days ago by bss8			
Simulation enhancement	Boris		
#11 opened 12 days ago by bss8			
AB Prune enhancement	Boris		
#10 opened 12 days ago by bss8			
Pieces enhancement	David		
#9 opened 12 days ago by bss8			
Board enhancement	David		
#8 opened 12 days ago by bss8			
MoveGen enhancement	David		
#7 opened 12 days ago by bss8			
Evaluation Function Three enhancement	Boris		
#6 opened 12 days ago by bss8			
Evaluation Function Two enhancement	Randy		
#5 opened 12 days ago by bss8			
Evaluation Function One enhancement	David		
#4 opened 13 days ago by bss8			

Figure 1: GitHub Contributions – 12 items split equally 3 ways for 4 items per team member

1.5.2 Group Member Contributions

I felt all members were active, equal participants in this project and came together to make it a success. All members participated in planning, design, implementation, team meetings, and debugging activities.

Randy worked on four out of twelve items (equally split three ways): Game.hpp and Game.cpp, Player.hpp and Player.cpp, the minimax-a-b algorithm and all its helper functions, and evaluation function two. Randy also assisted in troubleshooting code throughout the entirety of the application, came up with the idea of adding checks before couts for debug output (allowing us to disable it), and implemented helper functions in Algorithm.

David worked on four out of twelve items (equally split three ways): Pieces.hpp and Pieces.cpp, Board.hpp and Board.cpp, the MoveGen function to generate a list of available moves for a given Player, and evaluation function one. He compared a traditional 2D matrix against a bitmap representation of the checkers board and advocated for the latter to improve performance. David also assisted in troubleshooting code throughout the entirety of the application.

There were **142** commits and **40** closed pull requests into the code repository, which is **3,803** lines long.

2 System Description

2.1 System Perspective

The system is designed to closely simulate a real-life checkers game. We visually display the board and the pieces on it and re-print it each time a player makes a move to show the new state. The sequence of boards therefore constitutes the full path of the game. While the system perspective is focused on playing the game of checkers, it also provides additional verbose output by default to show what is transpiring under the surface. The minmax-a-b and alpha-beta-search algorithms print out (within reason) what they are doing internally. Some of the evaluation functions also print to the screen what decisions they are making at certain points of interest (e.g., sacrificing a MAN to capture an enemy KING receives a higher score).

2.2 System Features

The following lists shall comprise a complete listing of the system's available features:

1. Display a help menu on request.
2. Display a welcome message to the user on start-up.
3. Display a main menu with option codes for the user to select.
4. On option 1, run a full, exhaustive simulation of 72 games.
5. On option 2, prompt the user for the algorithms, evaluation functions, and depth for a single game between two AI players.
6. On option 3, allow the player to play a manual game against themselves or another human player.
7. On option 4, prompt the user for an algorithm, EF, and depth of an AI player and play a human vs AI game.
8. The Ctrl + C command will terminate the program at any time.

2.3 Design and Implementation Constraints

The system was built with Texas State Linux servers in mind. While it may operate properly in other environments (e.g., Windows-10, Ubuntu, SLES, etc.) it is guaranteed to run on either the Eros or Zeus TXST servers. Section 3.1 outlines detailed build and execution instructions, which are also available in a shortened format in the Project2-README-A04626934.txt file submitted alongside the source code.

Our team created a pleasant user interface, opting to fully display the entire checkers board, as close to its real-life counterpart as possible, making use of supported ANSI colors on the Texas State hosts. The program was designed to present the user with some common simulation scenarios – a full, exhaustive simulation, a manual game between two human players, an option to play against an AI player, and most importantly – the ability to manually run a single game with a custom

configuration. This option is very useful, as it allows a user to focus on what they want. While the first option is useful to exhaustively simulate all possible combinations (including reversing the starting positions for the same algorithm and EF combination), the output is quite large.

The output size was a constraint and full output for each simulated game will be provided in a separate file along with the report. It should also be noted that our team opted to focus on efficiency of the data structures, to minimize space utilization and reduce the size of the game tree (not the number of nodes expanded but the size of each one).

Another approach explored with storing each piece as a full class. Again, to stay lightweight and save space, we opted for a bitmap and representing a piece with the value of one. This means, however, that we lose the ability to add additional contextual information along with a “Piece” as they are no longer represented with a full-fledged class but as a single bit instead. The object-oriented approach of representing a piece as a class would allow us to add information to each piece. For example, we notice after a certain number of turns pieces start going “back-and-forth” between a few spots – jittering if you will. If we could add a Boolean variable “visitedHomeRow” for each piece that is a King, the piece could “remember” a certain contextual action. Namely that it became a King and travelled from the enemy back row all the way back to our own. We can then use this information to change how the piece behaves by treating it differently – say by using the enemy’s weighted move table instead of our own to force it to traverse across the board in the opposite direction. This specific problem of jittering will be examined in detail below.

2.4 Assumptions and dependencies

We assume users of this program will be familiar with using a command line interface (CLI) on the Texas State Linux servers.

The program is dependent on:

- A C++ compiler, such as the GNU C++ compiler for Linux (g++).
- The C++11 language standard.

3 System Design and Specification

The program is initially user-input driven. At the beginning a main menu will be displayed (Figure below), prompting the user to select an option. Option one is an exhaustive simulation of each algorithm and each depth, including swapping the player starting positions. This option is provided because results may vary between the same combination of algorithm and evaluation function for different starting positions. For example, Minimax with EF-1 as Red vs Minimax with EF-3 Black may be a draw whereas Minimax EF-3 Red vs Minimax EF-1 Black is a win for black. This is because in our program, Black always moves first. In checkers, just as in chess, moving first is an advantage. It may not be a sizeable advantage, but it does permit the first player to make a move to, say, take the center of the board early.

```
[bss64@eros project-two]$ ./CheckersAI
Welcome to the Checkers AI Program.
Authors: David Torrente (dat54@txstate.edu), Randall Henderson (rrh93@txstate.edu), Borislav Sabotinov (bss64@txstate.edu).
Re-run this program with -h or -help CLI argument to see a help menu or refer to README for instructions.

NOTE: If 1 is selected below, you will NOT be prompted further for any eval function or algorithm. All will be simulated in order.
For RED player, r = MAN and R = KING; for BLACK player, b = MAN and B = KING.

Choose a game mode below:
1. Full Simulation (recommended)
2. Single Custom Simulation
3. Player vs Player (manual game)
4. Player vs AI (will be asked to select AI playstyle)
Ctrl + C to terminate program at any time.

Your choice (1, 2, 3, or 4):
```

Figure 2: Main Menu

Regarding the game of checkers itself, a "piece" is a checker piece used in the game. It may be either red or black. Color is an immutable property.

A player has 12 starting pieces of a given color. A standard checkers piece may move only forwards in a diagonal manner. If the piece reaches the opposite end of the board, it becomes a "king" piece. This means it may now move backwards diagonally. A piece from player A may take a piece from player B by "jumping" over it diagonally if the square on the opposite side is empty. Jumps are mandatory - a player must make a jump if one is available to them. If multiple jumps are available, the player simply selects among the choices. A jump may be continuous - the piece must continue capturing enemy pieces if more are available for capture. It may not stop in the middle of a jump. Our program follows the standard rules for checkers as outlined above.

3.1 How to Build and Run the Program

The submission for this project consists of 9 files:

1. Project2-A04626934.cpp: source code file.
2. Project2-Report- A04626934.docx: the project report in Word format
3. Project2-Report- A04626934.pdf: the project report in PDF format
4. Project2-README-A04626934.txt: instructions for building and running the program.
5. Project2-ResultLogs-A04626934.zip:

- a. This ZIP contains two directories: Project2_depth_2_results_A04626934 and Project2_depth_4_results_A04626934. Each directory contains 18 log files, two files per each of the 9 games played at the respective depth.
 - b. The two files per game in these directories are as follows:
 - i. abs-1-abs-2-2-full.log and abs-1-abs-2-2-short.log. The first is with full debug output, the second is only the Board and move displayed for brevity. The naming convention for the output files is: <alg>-<eval>-<alg>-<eval>-<depth>-<verbosity>.log, where alg can be either min (for minimax-a-b) or abs (for alpha-beta-search), eval can be either 1, 2 or 3 (representing the evaluation functions), depth is either 2 or 4, and verbosity is either “short” or “full.”
6. Project2-fullSimulation- A04626934.log: 72 games with debug output suppressed.
 7. Project2-fullSimulationVerbose- A04626934.zip: a very large file, compressed as a ZIP. It contains verbose output for all 72 games simulated.
 8. Project2-Results- A04626934.xlsx: results of the 18 games outlining number of nodes expanded, number of leaf nodes, total nodes, memory usage, and more details. It is also included directly in this report in tables two and three.
 9. Project2-UMLDiagram-A04626934.png: the UML Class diagram image for ease of access and readability; zooming in will display details more clearly.

This application is primarily designed to run on Texas State (TXST) Linux servers.

Eros: EROS.CS.TXSTATE.EDU (147.26.231.153)

Zeus: ZEUS.CS.TXSTATE.EDU (147.26.231.156)

You may use WinSCP, FileZilla, or equivalent FTP software to transfer the project files to a TXST Linux host.

1. Project2-A04626934.cpp must be in a directory you own on a TXST host.
2. Build with this command:
g++ -o Project2 Project2-A04626934.cpp -std=c++11
3. Run with this command (will use default options):
./Project2
4. To see help menu (optional):
./Project2 -h
5. To run with no color:
./Project2 -nc
6. To run with no debug output:
./Project2 -no
7. To run with no debug output and no color:
./Project2 -ncno
8. To put std output in a log file:
./Project2 -nc | tee myFile.log

It is recommended to use option 2 from the main menu and run a game with one configuration at a time.

3.2 Why use C++11?

A small aside but worth mentioning. Why use the C++11 compiler? C++11 now supports:

- lambda expressions,
- automatic type deduction of objects,
- uniform initialization syntax,
- delegating constructors,
- deleted and defaulted function declarations,
- nullptr,
- rvalue references

The language was overhauled with new container classes, algorithms, smart pointers, async capability, and multithreading support, in addition to other useful features. The C++11 compiler is readily available on Texas State Linux servers.

3.5 System Class Diagram

Figure 10 below shows a class diagram for the program. I intentionally do not use a simplified UML diagram but instead opt to present full method signatures, including the arguments the member functions will expect. Each class is reviewed in detail in Section 4.

This program is longer and more complex than the one presented in Project-1. It is 3,803 lines long and contains several unique data structures to represent the checkers pieces and the board on which the game is played.

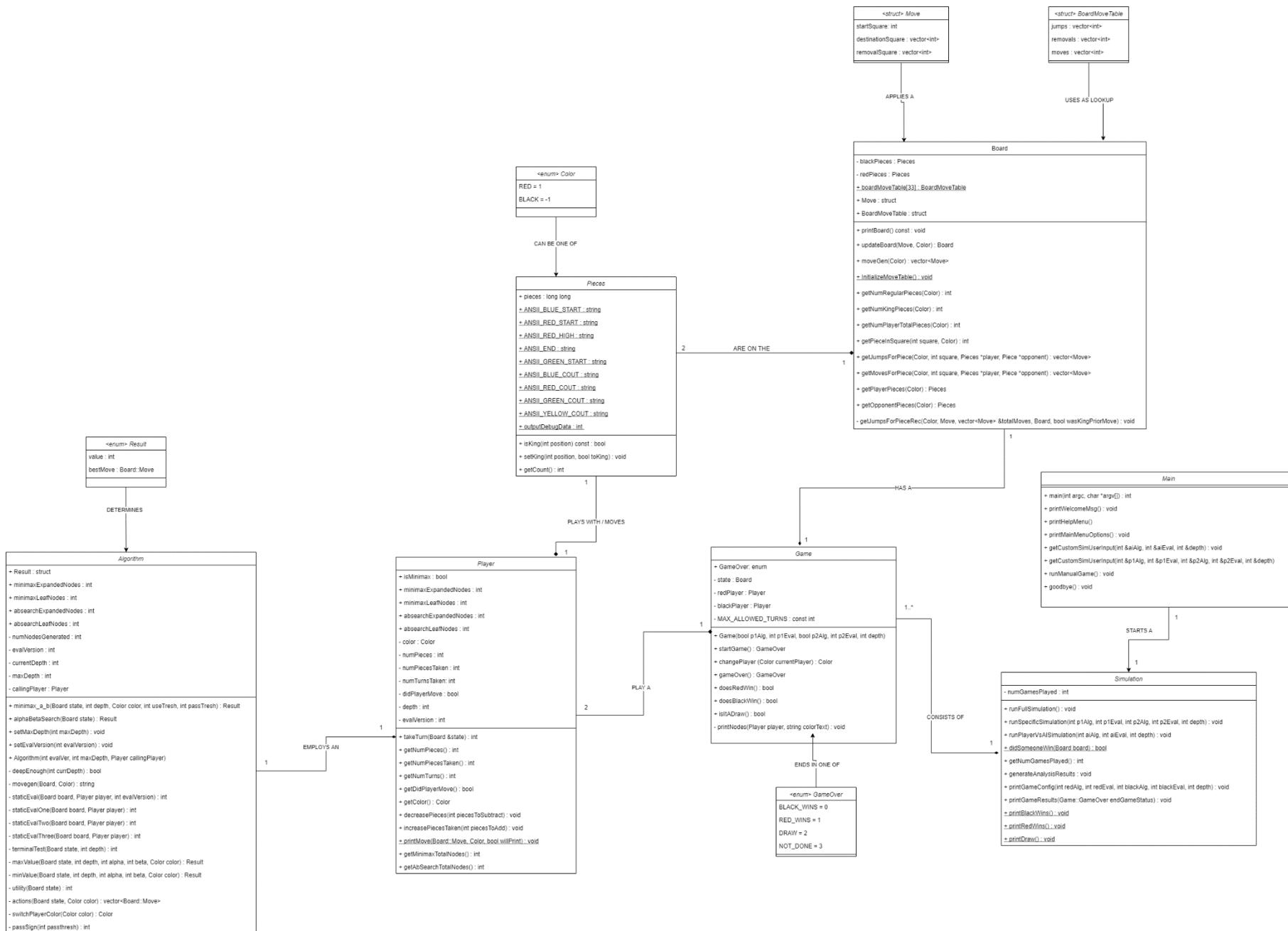


Figure 3: UML Class Diagram for the Checkers AI System

3.6 Analysis of Results

3.6.1 How Good Are the Results?

The program is successful in producing correct, repeatable results. It is also robust – it anticipates several edge cases and error scenarios and handles them appropriately. As we will see below, the two main algorithms produce the same game path, reach the same conclusion, and expand the same number of nodes given the same configuration (e.g., comparing Minimax-a-b with alpha-beta-search, both using EF-1). The program was thoroughly tested by the team and is generally free of bugs and flaws. The overall simulation results are in line with our expectation – minimax-a-b and alpha-beta-search are implemented in two different ways, but they produce identical results. Regarding the time it takes for each algorithm to perform the work, it appears alpha-beta-search is slightly faster at depths two and four, although the difference is negligible.

My personal preference is alpha-beta-search. It is very clear in the implementation who is being called (MIN or MAX). There is no need to flip the value, invert or swap the alpha and beta variables (known as passThresh and useThresh in minimax-a-b).

I ran the program 18 times – once for each required simulation (double if we include the runs I did with no output versus with verbose output). Additionally, I ran the program as follows:

- with the “Player vs AI” option to test myself against EF-2 and how it felt to play against the AI
- with a manual game to test that it works
- with depths 6 and again with 8 with the following configuration:
 - i. ALPHA-BETA-SEARCH with EF-1 vs ALPHA-BETA-SEARCH with EF-3
- twice for each of the three rows in table 2 below for a total of six runs
- once for the full simulation, option 1

Total 29 runs, double if we include the runs I did for capturing verbose and silent output log files included along with this report.

3.6.2 Memory and Speed

Disclaimer: just as in Project-1, only one student had Visual Studio, which contained a memory profiler. Our team collectively used this tool to profile the application’s memory usage and take some snapshots. The screenshots below are from a team activity that was collectively performed during a Zoom meeting.

The program performs well and there are no noticeable slow-downs or performance issues.

The three subsequent figures below show a captured recording of the program memory profile using Visual Studio 2019. We can see that the program is relatively lightweight and efficient at depth 4. We did test with higher depths and the amount of time the program took to complete a game increased significantly beyond depth 8, likely due to the size of the game tree being explored by the algorithms even with alpha and beta cutoffs.

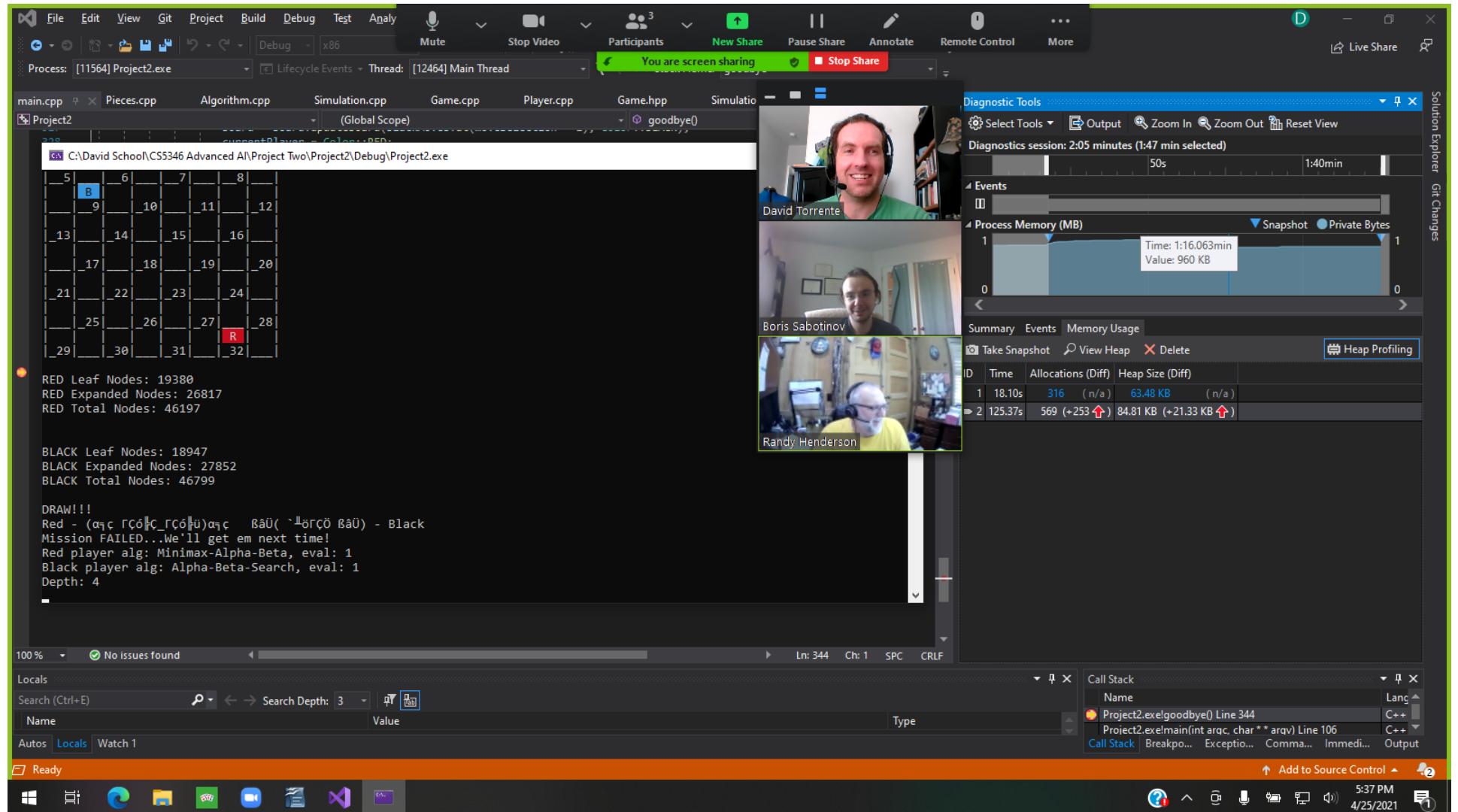


Figure 4: Memory profile of Minmax with EF-1 vs ABS with EF-1

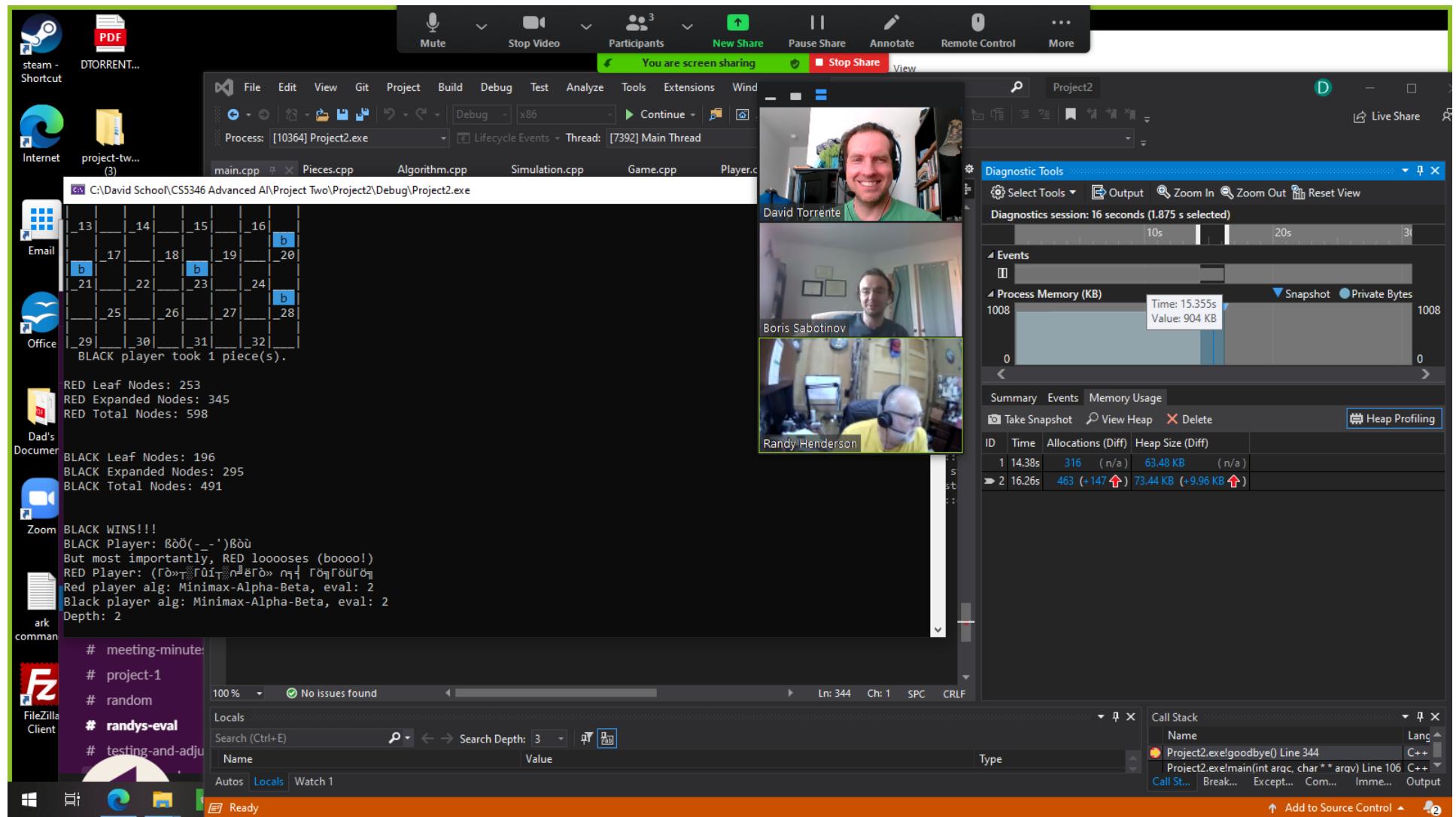


Figure 5: Memory Profile of Minmax with EF-2 vs Minmax with EF-2

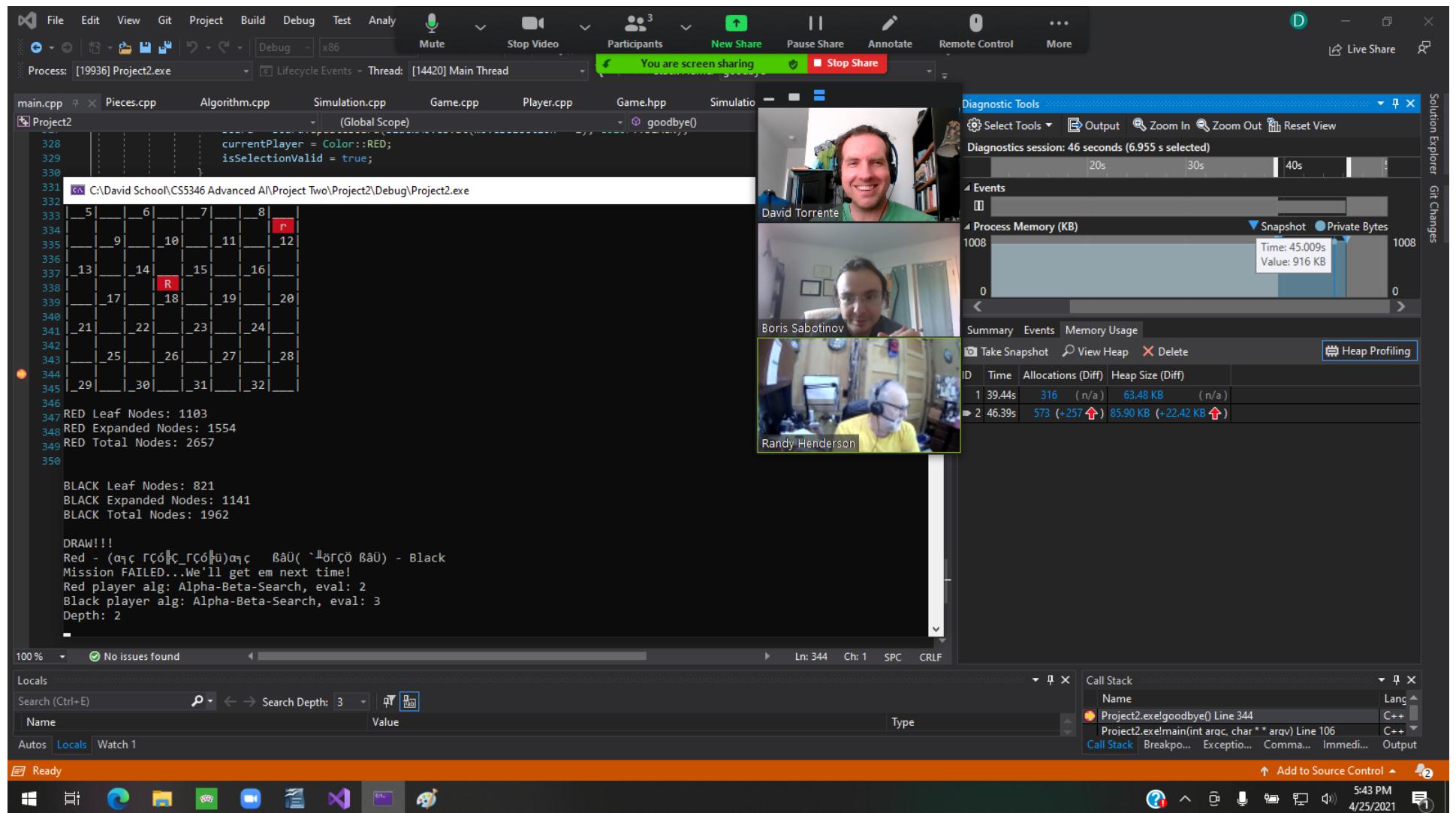


Figure 6: Memory Profile of ABS with EF-2 vs ABS with EF-3

Table 2: Timing Comparison for minmax-a-b vs a-b-search; the latter slightly outperforms

Algo 1	Algo 2	Eval 1	Eval 2	Min Time Depth 2	ABS Time Depth 2	Min Time Depth 4	ABS Time Depth 4	Min Time Depth 6	ABS Time Depth 6
Minmax	ABS	1	1	74.7621ms	83.1409ms	470.561ms	507.63ms	3167.95ms	3902.60ms
Minmax	ABS	2	2	2.8452ms	2.7658ms	217.269ms	191.182ms	6683.54ms	3662.31ms
Minmax	ABS	3	3	29.7645ms	31.11ms	115.978ms	187.228ms	4389.26ms	4281.88ms

From Table 2 above, we can see that the alpha-beta-search algorithm slightly outperforms minmax-a-b as it has a lower average runtime at higher depths, but at both depths two and four the algorithms perform similarly with minimax-a-b faring slightly better. Green values indicate the “winner” in terms of timing. At depth 6 alpha-beta-search did better overall in timing. However, we must keep in mind that the results heavily depend on the evaluation function being used, as its implementation may be as lightweight or as heavy as the author desires for evaluation of how good a given state may be. The results may also be influenced by the hardware used for testing (CPU and RAM used may be seen in the figure below). Also, what other programs were running at the time of the test may also influence the timing, depending on how available the CPU was at the time.

Processor: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz
 Installed memory (RAM): 16.0 GB (15.8 GB usable)
 System type: 64-bit Operating System, x64-based processor

Figure 7: CPU, RAM, and OS type used for testing

In the two tables below (3 and 4), we see an interesting result. Minimax-a-b and alpha-beta-search perform exactly the same in terms of the number of nodes each algorithm expands. For example, we see that MAB, EF-1 vs MAB, EF-2 expands 3616 red nodes and 4209 black nodes in total. Comparing to ABS, EF-1 vs ABS, EF-2, the results are the same – 3616 red and 4209 black nodes were expanded. This makes intuitive sense. Both algorithms use recursion, MAB directly via a recursive call to itself and ABS indirectly by having two helper functions minValue() and maxValue() call each other. Both functions use alpha and beta values to determine where to perform a cutoff and prune a branch that does not need to be explored further (as a “best” value has already been found elsewhere).

Seeing the results and the two different algorithms perform the same, giving us the same game path, the same conclusion, and the same number of expanded nodes shows us that the algorithms are correctly implemented and behaving as expected.

At depth two, we do not go above 12K generated nodes in total. The game is rather limited in options and the moves are simple and straightforward. All games end in a Draw for us, except Minmax with EF-2 vs ABS with EF-2 (same EF against itself); we get a win for Black, likely because of the slight “first move” advantage Black has.

Table 3: Results for 9 runs at depth 2

Runs at Depth 2 (r = red, b = black)										
Algo 1	Algo 2	Eval 1 r	Eval 2 b	Path Len	Leaf Nodes	Expanded Nodes	Total Nodes (leaf)	Time (sec)	Peak mem	Win / Loss
Minmax	ABS	1	1	80	2596r 3004b	3295r 3675b	5891r 6679b	17.9	968 KB	Draw
Minmax	ABS	2	2	25	253r 196b	345r 295b	598r 491b	1.6	904 KB	Black
Minmax	ABS	3	3	80	1158r 1202b	1674r 1533b	2832r 2735b	9.07	924 KB	Draw
Minmax	Minmax	1	2	80	1648r 1620b	1968r 2589b	3616r 4209b	10.18	940 KB	Draw
Minmax	Minmax	1	3	80	1604r 1640b	2511r 1950b	4115r 3590b	13.37	932 KB	Draw
Minmax	Minmax	2	3	80	1103r 821b	1554r 1141b	2657r 1962b	6.18	916 KB	Draw
ABS	ABS	1	2	80	1648r 1620b	1968r 2589b	3616r 4209b	10.15	924 KB	Draw
ABS	ABS	1	3	80	1604r 1640b	2511r 1950b	4115r 3590b	13.25	936 KB	Draw
ABS	ABS	2	3	80	1103r 821b	1554r 1141b	2657r 1962b	6.27	928 KB	Draw

At depth four, we are already seeing a tremendous increase in the number of nodes we expand. The game tree explodes in complexity from under 7K nodes at depth 2 (which only is observed on row 1, Min w/ EF-1 vs ABS w/ EF-1) to over 63K nodes at depth 4, a near 10x increase. I tested at depth 6 and saw the same thing, about a 10x increase in nodes generated, where the average was around 250,000 leaf nodes for one player (compared to 25,000 at depth 4). We can infer that increasing the depth by two leads to a tenfold increase in the number of nodes we can expect to generate. One notable exception is my evaluation function, EF-3, when tested against itself at depth 4 led to fewer expanded nodes. Perhaps because each player made the expected or “rational” move from each other’s perspective.

Table 4: results for 9 runs at depth 4

Runs at Depth 4 (r = red, b = black)											
Algo 1	Algo 2	Eval 1 r	Eval 2 b	Path Len	Leaf Nodes	Expanded Nodes	Total Nodes (leaf)	Time (sec)	Peak mem	Win / Loss	
Minmax	ABS	1	1	80	19380r 18947b	26817r 27852b	46197r 46799b	95.44	960 KB	Draw	
Minmax	ABS	2	2	80	28641r 29419b	44847r 39043b	73488r 68462b	39.27	964 KB	Draw	
Minmax	ABS	3	3	22	3052r 4894b	4814r 7203b	7866r 12097b	30.08	952 KB	Red	
Minmax	Minmax	1	2	80	19139r 14910b	25786r 26728b	44925r 41638b	69.42	980 KB	Draw	
Minmax	Minmax	1	3	80	24683r 24683r	37489r 51299b	62172r 89789b	185.51	972 KB	Draw	
Minmax	Minmax	2	3	80	23521r 14272b	39900r 18544b	63421r 32816b	58.69	984 KB	Draw	
ABS	ABS	1	2	80	19139r 14910b	25786r 26728b	44925r 41638b	68.07	960 KB	Draw	
ABS	ABS	1	3	80	24683r 24683r	37489r 51299b	62172r 89789b	187.04	948 KB	Draw	
ABS	ABS	2	3	80	23521r 14272b	39900r 18544b	63421r 32816b	59.69	968 KB	Draw	

3.6.3 Optimizations

Our team used object-oriented design principles while keeping some data structures (for the board) low-level to save time and space. The minimax-a-b and alpha-beta-search algorithms are implemented as-is. They are an exact implementation of the algorithms provided on Canvas and the respective book referenced for each algorithm.

A notable improvement is our decision to forego a traditional 2D array representation of a board and use a bitmap.

We initially examined a “traditional” approach for representing the checkers board and its pieces – a class Piece to represent each piece and the board represented as a 2D array with row and column. After simulating this approach against a bitmap representation of the board, we quickly decided to go with the latter as it provided a sizeable difference in performance and runtime.

David Torrente simulated copying a board 100 million times, comparing an int[8][4] vs char[8][4] vs a long long bitfield.

The results from this test are shown below (only the first 4 runs, though 10 were performed).

Table 5: 100mil copy test by David Torrente

Test # (time in sec):	1	2	3	4	Avg (for 10 runs)
int[8][4]	6.08	6.05	5.92	5.83	5.863
char[8][4]	5.73	5.80	5.83	5.80	5.8
long long	0.28	0.29	0.28	0.28	.288

Before we started implementing this project, we wanted to decide on the data structure to represent the Board and pieces. In the table above, we observed that a bitmap is 20 times more efficient on average than either an int or a char 2D array. Bitmaps were the clear winner. In the table below, however, we see a drawback – checking the position of a piece in an array is $O(1)$ because we simply access it directly via the index (e.g., `board[1][3]`). Whereas with bitmaps we need to perform a shift and compare. When tested with 30 million comparisons, bitmaps were about 4 times slower than the competition.

Table 6: 30mil comparisons test by David Torrente

Test # (time in sec):	1	2	3	4	Avg (for 10 runs)
int[8][4]	2.63	2.5	2.6	2.5	2.605
char[8][4]	2.56	2.51	2.31	2.48	2.528
long long	11.8	11.51	11.48	11.47	11.566

Our team opted to sacrifice a bit of speed to gain a significant improvement in memory utilization and save on space.

3.7 Evaluation Functions

There are three EFs implemented for this program, EF-1 as written by David Torrente, EF-2 as written by Randall Henderson, and EF-2 as written by Borislav Sabotinov. Each evaluation function takes in a Board and a Color and returns an integer score, which signifies how good we think is the provided state.

I will begin by discussing EF-3, as it is the function I designed and implemented for this project.

3.7.1 Evolution of Evaluation Function Three (EF-3)

The third evaluation function was the first to be implemented and pushed to the repository for testing in a way that returned an integer value with some meaning behind it. Initially we simply always returned 1, which would create a good predictable game – AI players would always take the first move available to them. In the initial state of EF-3 referenced below, we can see I began by assigning each position on the board a value or weight. The intent was to guide the behavior of the pieces using a strategy I employed when I played checkers against an AI online to get a feel for the game.

My strategy was to keep two checker pieces on the back row, in squares that had two available moves. That way, I could “defend” my back row against enemy checkers that tried to get a King piece while remaining safe (back row cannot be captured). While I placed some value on the sides due to their relative safety from captures, I placed a higher value on squares in the center of the board. I also calculated a “mobility” score – given the provided state of the game, where could my pieces move to – would I be able to control squares with a higher value like in the center? The final value returned by this initial EF was a summation of the number of pieces I had including kings, the mobility score, and the score of my current position. As we did not have other evaluation functions to test against at the time, I tested it against itself. Results were either a draw or a win for black, who moves first. When I played against a player who used a “always take the first available move” simple strategy, the algorithm would lose at both depths.

Initial state of EF-3

```
1. int Algorithm::evalFuncThree(Board state, Color color)
2. {
3.     int squareValuesForRed[] = { 100, 1, 100, 1,
4.                                 2, 1, 1, 2,
5.                                 1, 1, 1, 2,
6.                                 2, 3, 3, 1,
7.                                 2, 3, 3, 4,
8.                                 2, 3, 3, 1,
9.                                 3, 3, 3, 4,
10.                                10, 10, 10, 10 };
11.
12.    int squareValuesForBlack[] = { 10, 10, 10, 10,
13.                                 4, 3, 3, 3,
14.                                 1, 3, 3, 2,
15.                                 4, 3, 3, 2,
16.                                 1, 3, 3, 2,
17.                                 2, 1, 1, 1,
18.                                 2, 1, 1, 2,
19.                                1, 100, 1, 100 };
```

```

20.
21.     int numPieces = state.getNumPlayerTotalPieces(color);
22.     int numKingsScore = state.getNumKingPieces(color) * 10;
23.
24.     std::vector<Board::Move> moves = state.moveGen(color);
25.
26.     int advancementScore = 0;
27.     for (int move = 0; move < moves.size(); move++)
28.     {
29.         if (color == Color::RED)
30.         {
31.             advancementScore += squareValuesForRed[moves.at(move).destinationSquare.back()];
32.         }
33.         else if (color == Color::BLACK)
34.         {
35.             advancementScore += squareValuesForBlack[moves.at(move).destinationSquare.back()];
36.         }
37.     }
38.
39.     int positionScore = 0;
40.     Pieces playerPieces = state.getPlayerPieces(color);
41.
42.     for (int piece = 0; piece < 32; piece++)
43.     {
44.         if (color == Color::RED)
45.         {
46.             int pieceBit = (playerPieces.pieces >> piece) & 1;
47.             if (pieceBit == 1)
48.             {
49.                 positionScore += squareValuesForRed[piece];
50.             }
51.
52.         }
53.         else if (color == Color::BLACK)
54.         {
55.             int pieceBit = (playerPieces.pieces >> piece) & 1;
56.             if (pieceBit == 1)
57.             {
58.                 positionScore += squareValuesForBlack[piece];
59.             }
60.         }
61.
62.     }
63.
64.     int compositeScore = numPieces + numKingsScore + advancementScore + positionScore;
65.
66.     return compositeScore;
67. }

```

After some trial and error, I added additional logic to the function and arrived at the second snapshot of EF-3 as seen below.

After testing, I discovered that this second version of EF-3 had a flaw. Starting on line 115 and again on 152 in the second snapshot below, I was attempting to determine if the player could retaliate after losing a piece. I would look for the surrounding squares, but my calculations were off as I did not notice the mathematical difference between squares differs depending on whether we are on an odd or even row. Suppose an enemy takes our piece and lands on the second row in square 6. We would add 3 to get the bottom left square of 9 and add 4 to get right square of 10. But suppose now the enemy landed on 10 instead, which is on row 3 – we would have to add 4 to get the bottom left square and 5 to get the bottom right. I had not noticed this when designing the algorithm, leading to unpredictable scoring. The algorithm would assume we could retaliate when in fact we could not, for instance.

My strategy now was to use two additional tables to drive the behavior of King pieces. The values for each player are mirror opposites and the intent was to play very aggressively – simply move all the pieces forward, towards the opponent’s back row in the hopes of getting a king, then move the king to the center of the board and score some captures on the way. I had a captures section to create a score of how many pieces, if any, the player could hope to capture on this state. Similarly, I had a section for casualties – how many pieces the player would lose. I decided in an aggressive checkers strategy losing a piece is not always a bad thing – if we do not lose a king, if the enemy does not gain a king by landing on our back row, and if we can retaliate immediately on the next turn, we would want to trade the piece for a position advantage.

At this stage we had preliminary implementations for both EF-1 and EF-2 and despite its flaws this version of EF-1 forced a Draw against both. It also forced a Draw against my arch-nemesis, the simple “take the first move” strategy of our test player mentioned earlier. However, it won against this test player in a unique way at depth four! When I first saw the board, I thought there was a bug but the win for Black in Figure 8 below is legitimate. Red has a 10 to 1 advantage in pieces and 4 kings to my one remaining king piece. But because my piece was able to sneak behind enemy lines so to speak, the simple AI enemy boxed itself in and ran out of moves at turn 43, giving Black the win.

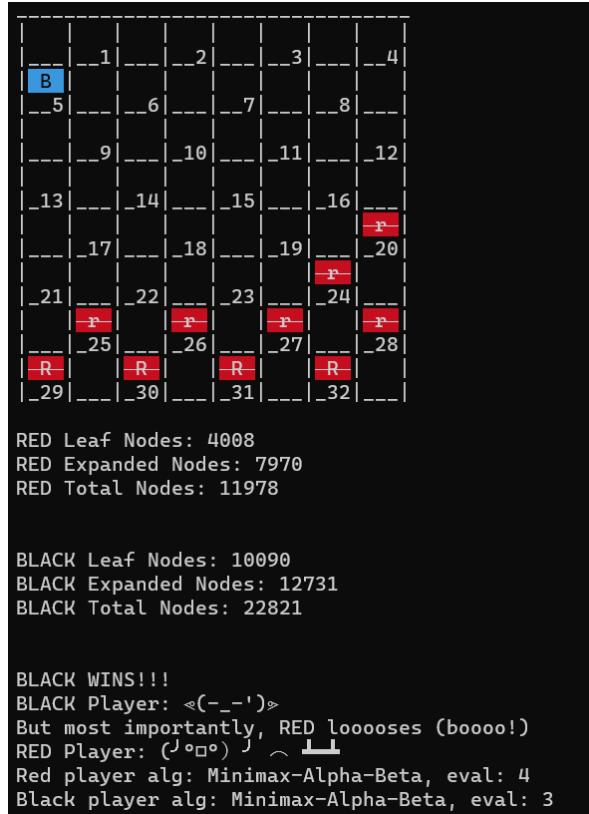


Figure 8: EF-3 wins against an AI player who selects the 1st move by letting the enemy block itself in a no-moves state

Second snapshot of EF-3 evolution in design and implementation

```

1. int Algorithm::evalFunctThree(Board state, Color color)
2. {
3.   int squareValuesForRed[] = { 100, 0, 100, 0,
4.                             1, 1, 1, 1,
5.                             1, 1, 1, 1,
6.                             1, 9, 9, 9,
7.                             10, 10, 10, 1,
8.                             1, 11, 11, 11,
9.                             12, 12, 12, 10,
10.                            110, 110, 110, 110};
11.   int squareValuesForBlack[] = { 110, 110, 110, 110,

```

```

12.                     10, 12, 12, 12,
13.                     11, 11, 11, 1,
14.                     1, 10, 10, 10,
15.                     9, 9, 9, 1,
16.                     1, 1, 1, 1,
17.                     1, 1, 1, 1,
18.                     0, 100, 0, 100};

19.     int squareValuesForRedKing[] = { 3, 1, 1, 5,
20.                                         5, 2, 2, 3,
21.                                         3, 10, 10, 5,
22.                                         5, 10, 10, 3,
23.                                         3, 10, 10, 5,
24.                                         5, 10, 10, 10,
25.                                         3, 2, 2, 5,
26.                                         5, 1, 1, 3};

27.     int squareValuesForBlackKing[] = { 3, 1, 1, 5,
28.                                         5, 2, 2, 3,
29.                                         3, 10, 10, 5,
30.                                         5, 10, 10, 3,
31.                                         3, 10, 10, 5,
32.                                         5, 10, 10, 10,
33.                                         3, 2, 2, 5,
34.                                         5, 1, 1, 3};

35.     const int KING = 2, MAN = 1;
36.     int numPieces = state.getNumPlayerTotalPieces(color);
37.     int numKingsScore = state.getNumKingPieces(color);
38.     int casualtyScore = 0, captureScore = 0, positionScore = 0, playerPiece = 0, enemyPiece = 0;
39.     std::vector<Board::Move> playerMoves = state.moveGen(color);
40.     std::vector<Board::Move> enemyMoves = state.moveGen(switchPlayerColor(color));
41.     Pieces playerPieces = state.getPlayerPieces(color);
42.     Pieces opponentPieces = state.getOpponentPieces(color);
43.     Pieces *p_playerPieces = &playerPieces;
44.     Pieces *p_opponentPieces = &opponentPieces;
45.     std::vector<Board::Move> playerJumpsForPiece;
46.     std::vector<Board::Move> opponentJumpsForPiece;
47.     const long long redBackRowGrp = (1LL << 1) | (1LL << 2) | (1LL << 3) | (1LL << 4);

```

```

48. const long long blackBackRowGrp = (1LL << 29) | (1LL << 30) | (1LL << 31) | (1LL << 32);
49. const long long sideColumnGrp = (1LL << 5) | (1LL << 13) | (1LL << 21) | (1LL << 12) | (1LL << 20) | (1LL << 28);
50. // CHECK TERMINAL STATE - Enemy is out of moves, this is a game ending move
51. if (enemyMoves.size() == 0)
52.     return std::numeric_limits<int>::max() - 1000;
53. else if (playerMoves.size() == 0)
54.     return 0; // bad move for us!
55. // BEGIN ADVANCEMENT SCORE SECTION
56. int advancementScore = 0;
57. for (int move = 0; move < playerMoves.size(); move++)
58. {
59.     if (color == Color::RED)
60.         advancementScore += squareValuesForRed[playerMoves.at(move).destinationSquare.back()];
61.     else if (color == Color::BLACK)
62.         advancementScore += squareValuesForBlack[playerMoves.at(move).destinationSquare.back()];
63. }
64. // END ADVANCEMENT SCORE SECTION
65. /* BEGIN CAPTURES SECTION
66. If for any given piece we can make more than one capture, we are done.
67. That is a very good move and we prioritize it.
68. Otherwise if we can make at least one capture, we give this state a high score
69. but we still want to examine others to see if they may be better
70. */
71. for (int i = 0; i < playerMoves.size(); i++)
72. {
73.     if (playerMoves.at(i).removalSquare.size() > 1)
74.     {
75.         if (Pieces::outputDebugData)
76.             std::cout << "We can capture multiple pieces!" << std::endl;
77.         return std::numeric_limits<int>::max() - 10; // great move!
78.     }
79.     else if (playerMoves.at(i).removalSquare.size() == 1)
80.     {
81.         return std::numeric_limits<int>::max() / 2; // great move but can we do better?
82.     }
83. }

```

```

84.     for (int j = 0; j < enemyMoves.size(); j++)
85.     {
86.         if (enemyMoves.at(j).removalSquare.size() > 1)
87.         {
88.             if (Pieces::outputDebugData)
89.                 std::cout << "Enemy can capture multiple pieces!" << std::endl;
90.             return 0; // bad move!
91.         }
92.         else if (enemyMoves.at(j).removalSquare.size() == 1)
93.         {
94.             int capturedPieceType = state.getPieceInSquare(enemyMoves.at(j).removalSquare.at(0), color);
95.             if (capturedPieceType == KING)
96.                 return 0; // bad move!
97.             else if (capturedPieceType == MAN)
98.             {
99.                 int opponentDestinationSqr = enemyMoves.at(j).destinationSquare.at(0);
100.                if (color == Color::RED)
101.                {
102.                    // if we are RED, opponent is Black; if BLACK enemy lands on our back row, avoid at all cost
103.                    // we are therefore trying to minimize the chance of an enemy getting a KING
104.                    if ((1 << opponentDestinationSqr) & redBackRowGrp)
105.                        return 0;
106.                    // opponent lands on their own back row; not so bad but we can't retaliate so avoid
107.                    else if ((1 << opponentDestinationSqr) & blackBackRowGrp)
108.                        return 0;
109.                    // opponent lands on one of the side squares, where we cannot retaliate. Avoid as well
110.                    else if ((1 << opponentDestinationSqr) & sideColumnGrp)
111.                        return 0;
112.                    // finally, we exhaust negative scenarios and are where we want to be
113.                    else
114.                    {
115.                        // get the square numbers around the enemy
116.                        int playerTopLeftSqr = opponentDestinationSqr - 4;
117.                        int playerTopRightSqr = opponentDestinationSqr - 3;
118.                        int playerBottomLeftSqr = opponentDestinationSqr + 4;
119.                        int playerBottomRightSqr = opponentDestinationSqr + 5;

```

```

120.         // get our pieces in those squares, if we have any
121.         int playerTopLeftPiece = state.getPieceInSquare(playerTopLeftSqr, Color::RED);
122.         int playerTopRightPiece = state.getPieceInSquare(playerTopRightSqr, Color::RED);
123.         int playerBottomLeftPiece = state.getPieceInSquare(playerBottomLeftSqr, Color::RED);
124.         int playerBottomRightPiece = state.getPieceInSquare(playerBottomRightSqr, Color::RED);
125.         // Since we are RED, our MEN can only advance forward top to bottom. Top pieces adjacent to where the opponent
126.         // landed can be MAN or KING. Bottom pieces however must be KING, otherwise we can't move backwards
127.         if (playerTopLeftPiece == MAN || playerTopRightPiece == MAN ||
128.             playerTopLeftPiece == KING || playerTopRightPiece == KING ||
129.             playerBottomLeftPiece == KING || playerBottomRightPiece == KING)
130.         {
131.             casualtyScore += 500;
132.         }
133.         else
134.             return 0; // we lost a piece and there is nothing we can do about it, avoid this state
135.     }
136. }
137. else
138. {
139.     // if we are BLACK, opponent is Red; if RED enemy lands on our back row, avoid at all cost
140.     // we are therefore trying to minimize the chance of an enemy getting a KING
141.     if ((1 << opponentDestinationSqr) & blackBackRowGrp)
142.         return 0;
143.         // opponent lands on their own back row; not so bad but we can't retaliate so avoid
144.     else if ((1 << opponentDestinationSqr) & redBackRowGrp)
145.         return 0;
146.         // opponent lands on one of the side squares, where we cannot retaliate. Avoid as well
147.     else if ((1 << opponentDestinationSqr) & sideColumnGrp)
148.         return 0;
149.         // finally, we exhaust negative scenarios and are where we want to be
150.     else
151.     {
152.         // get the square numbers around the enemy
153.         int playerTopLeftSqr = opponentDestinationSqr - 4;
154.         int playerTopRightSqr = opponentDestinationSqr - 3;
155.         int playerBottomLeftSqr = opponentDestinationSqr + 4;

```

```

156.         int playerBottomRightSqr = opponentDestinationSqr + 5;
157.         // get our pieces in those squares, if we have any
158.         int playerTopLeftPiece = state.getPieceInSquare(playerTopLeftSqr, Color::RED);
159.         int playerTopRightPiece = state.getPieceInSquare(playerTopRightSqr, Color::RED);
160.         int playerBottomLeftPiece = state.getPieceInSquare(playerBottomLeftSqr, Color::RED);
161.         int playerBottomRightPiece = state.getPieceInSquare(playerBottomRightSqr, Color::RED);
162.         // Since we are BLACK, our MEN can only advance forward bottom to top. Top pieces adjacent to where the opponent
163.         // landed can KING only or we cannot move backwards. Bottom pieces can be either MAN or KING
164.         if (playerTopLeftPiece == KING || playerTopRightPiece == KING ||
165.             playerBottomLeftPiece == MAN || playerBottomRightPiece == MAN ||
166.             playerBottomLeftPiece == KING || playerBottomRightPiece == KING)
167.         {
168.             casualtyScore += 500;
169.         }
170.         else
171.             return 0; // we lost a piece and there is nothing we can do about it, avoid this state
172.     }
173. }
174. }
175. }
176. }
177. // MAIN LOOP FOR SCORING CALCULATIONS
178. for (int piece = 0; piece < 32; piece++)
179. {
180.     // BEGIN POSITION SCORE SECTION
181.     if (color == Color::RED)
182.     {
183.         playerPiece = state.getPieceInSquare(piece, color);
184.         if (playerPiece == MAN)
185.             positionScore += squareValuesForRed[piece];
186.         else if (playerPiece == KING)
187.             positionScore += squareValuesForRedKing[piece];
188.         enemyPiece = state.getPieceInSquare(piece, Color::BLACK);
189.         if (enemyPiece == MAN)
190.             positionScore -= squareValuesForBlack[piece];
191.         else if (enemyPiece == KING)

```

```
192.         positionScore -= squareValuesForBlackKing[piece];
193.     }
194.     else if (color == Color::BLACK)
195.     {
196.         playerPiece = state.getPieceInSquare(piece, color);
197.         if (playerPiece == MAN)
198.             positionScore += squareValuesForBlack[piece];
199.         else if (playerPiece == KING)
200.             positionScore += squareValuesForBlackKing[piece];
201.     }
202.     // END POSITION SCORE SECTION
203. }
204. int compositeScore = numPieces + numKingsScore + advancementScore + positionScore + captureScore + casualtyScore;
205. return compositeScore;
206. }
```

3.7.2 Final state of EF-3

Below is the final version of EF-3. I redesigned it section by section, applying lessons learned from previous iterations. I removed the many returns at different locations and opted to simply provide an integer value for those situations. I removed one of the king arrays, so both colors now use the same position value table. I also marked the first row of the player with a higher score than the one in front of it. From Red's perspective it is the 3rd row from the top of the displayed board (squares 9, 10, 11, and 12). From Black's perspective it is the 3rd row from the bottom (squares 21, 22, 23, and 24). Because only this row has moves available, the first row will advance. But once this happens, the second row now has a piece which can move into the gap left by the first piece. This ensures we advance somewhat in waves – one piece from the front row moves but then the gap is filled by a piece behind it and so forth. I still opted to keep two pieces on defense in the back row, giving me 10 pieces to use offensively.

The values get progressively higher as we get towards the opponent back row – I try to attack the “double corner” of the enemy, the square on their home row at the edge, which connects to two other squares. I try to stay away from getting boxed in the other corner which is only accessible via one adjacent square. This is an effective strategy at preserving pieces and advancing systematically towards a goal of obtaining a king. The drawback is that the king may get boxed in if we get lucky and our Men behind score some captures and follow closely behind, limiting our King's mobility.

This final version of EF-3 has a more intelligent method for scoring the number of pieces on the board. Each piece has an associated weight equal to the number of moves it may perform. Thus, a Man is worth 2 and a King is worth 4 points. We look at our pieces – their number and type – in relation to the enemy and have the option to assign a bonus to boards where we have a numerical advantage. This section starting on line 57 and ending on 65 is currently commented out as it led to worse performance against other AI players using EF-2 and EF-3, the reason for which needs to be investigated.

A clever use of bit shifting is used to quickly check if a piece is in a particular area of the board. Suppose we are Red, and we want to check if an enemy Black piece may land anywhere on our back row.

```
redBackRowGrp = (1LL << 1) | (1LL << 2) | (1LL << 3) | (1LL << 4);
```

On line 83 we declare the redBackRowGrp as shown above. It is of type long long, thus 64 bits. We shift left once, bitwise-OR with a shift left of two. Repeat for two and three. If we start with 0001 and shift left 1, we get 0010. We bitwise-OR this with 0001 shifted twice (0100), giving us 0110. For this example, we shall stop there as this provides the general idea. We can use this to check if an opponent has landed on our back row:

```
if ((1 << opponentDestinationSqr) & redBackRowGrp)
```

If the opponent landed on 2, we shift and get 0100. We then bitwise-AND with 0110 (our group value) and we get 0100. This is a non-zero value, which will evaluate to true in our if condition, so we know the opponent has landed somewhere on our back row.

I now also multiply the value of a square by the weight of the piece if we have a piece on that square. So, a state leading to a piece in square 1 would lead to 1000 for the position * 4 (value of a king) = 4000, a desirable state. Redesigned also are the captures and casualties sections – less returns, replaced with scores. Also, the method for determining if it is safe to conduct a trade and sacrifice a Man, where we can retaliate on the next turn, corrects the flaw described in the second

snapshot. We still return an integer value to signal how good this state is but the components we sum up are greater this time and the method by which we arrive at our score more complex.

What I learned while implementing the evaluation function is that even though we are returning a single number, much information can be conveyed in that one value if we arrive at it intelligently. Because the algorithms will look for a “good” state and prune unpromising branches, the more accurate the score provided by the EF is, the better we can perform.

One limitation, however, is that the AI player only knows their own evaluation function – we cannot read the enemy’s mind just as in real life. If we are playing against an AI using the same EF, results and the enemy behavior are more predictable. If, on the other hand, we are playing with EF-1 and the enemy with EF-2, our results simulate the enemy moves as if they were using EF-1. We are only as good as our evaluation function, so we must take great care to anticipate scenarios and evaluate the state of the board and its many possibilities.

One notable limitation all three evaluation functions encountered is a back-and-forth jiggering towards the late game, leading to a draw. This is because we can drive the pieces towards a destination relatively easily but once they get there, getting them to behave in a different manner is more complex. One possible solution was mentioned earlier – introduce a Piece class for each piece and have each checker piece remember if it has performed some action or visited some area. If it has, treat it differently from other pieces to drive it to traverse the board. Yet another solution is to remember the last few states of the game because a real human player would detect they are going back and forth and try something different. Especially when an AI player has a large piece advantage, there is really no need for the game to end in a Draw as the player can start moving towards the enemy. But here we are limited by the “horizon effect” of the defined depth. At depth 2, we can only see two moves ahead. If the enemy pieces are beyond two moves away, we do not see a capture possible and thus we have no reason to make different moves. At higher depths (8 and beyond) it is possible we may detect the enemy and a possible capture. Finally, a simple strategy of taking a random move if a back-and-forth jitter is detected may work as well, forcing the player into more unique states and possibly closer to the opposing player.

Final version of EF-3

```
1. int Algorithm::evalFuncThree(Board state, Color color)
2. {
3.     /* Declaration of "boards" with a weight for each square
4.     Two for regular pieces of each color and two for kings of each color
5.     Encourage player to keep two pieces in the back for defense
6.     leaving 10 pieces for offense. Advance to the center
7.     but somewhat in waves as a cluster, to avoid suicidal pieces that expose themselves
8.     Try to attack the opponent's "double corner" from where a kinged piece can escape faster
9. */
10.    int squareValuesForRed[] = {7, 1, 7, 1,
11.                                1, 2, 2, 2,
12.                                1, 5, 5, 5,
13.                                1, 3, 3, 3,
14.                                1, 4, 4, 4,
```

```

15.                     1, 5, 250, 250,
16.                     1, 250, 500, 500,
17.                     50, 100, 100, 1000};
18.
19.     int squareValuesForBlack[] = {1000, 100, 100, 50,
20.                                     500, 500, 250, 1,
21.                                     250, 250, 5, 1,
22.                                     4, 4, 4, 1,
23.                                     3, 3, 3, 1,
24.                                     5, 5, 5, 1,
25.                                     2, 2, 2, 1,
26.                                     1, 7, 1, 7};
27.
28. // Kings preference for center, with some traversal lines
29. // to attempt and avoid a "back-and-forth" pattern
30. int squareValuesForKing[] = {1, 1, 1, 1,
31.                             1, 5, 5, 55,
32.                             5, 15, 45, 1,
33.                             1, 5, 35, 5,
34.                             5, 25, 25, 1,
35.                             1, 15, 5, 15,
36.                             5, 5, 5, 10,
37.                             1, 1, 1, 1};
38.
39. std::string colorTxt = (color == Color::RED) ? " (RED is Friendly) " : " (BLACK is Friendly) ";
40. // KING has 4 moves max, so value is 4; MAN has 2 moves max so values is 2
41. const int KING = 2, MAN = 1, KING_VALUE = 4, MAN_VALUE = 2;
42.
43. int numPlayerTotalPieces = state.getNumPlayerTotalPieces(color);
44. int numEnemyTotalPieces = state.getNumPlayerTotalPieces(switchPlayerColor(color));
45. int numPlayerTotalKings = state.getNumKingPieces(color);
46. int numEnemyTotalKings = state.getNumKingPieces(switchPlayerColor(color));
47. int numPlayerTotalMen = numPlayerTotalPieces - numPlayerTotalKings;
48. int numEnemyTotalMen = numEnemyTotalPieces - numEnemyTotalKings;
49.
50. int numKingsScore = numPlayerTotalKings * KING_VALUE;

```

```

51. int numMenScore = numPlayerTotalMen * MAN_VALUE;
52.
53. int diffInNumPieces = numPlayerTotalPieces - numEnemyTotalPieces;
54. int diffInNumKings = numPlayerTotalKings - numEnemyTotalKings;
55. int diffInNumMen = numPlayerTotalMen - numEnemyTotalMen;
56.
57. // PIECE BONUS/PENALTY - UNCOMMENT TO ACTIVATE
58. // if diff in kings is positive, score is amplified with a bonus
59. // if diff in kings is negative, however, score is penalized accordingly (by adding a negative)
60. // numKingsScore += (40 * diffInNumKings);
61.
62. // if diff in men is positive, score is amplified with a bonus
63. // if diff in men is negative, however, score is penalized accordingly (by adding a negative)
64. // numMenScore += (20 * diffInNumMen);
65. // END PIECE BONUS/PENALTY
66.
67. int casualtyScore = 0, captureScore = 0, positionScore = 0, playerPiece = 0, enemyPiece = 0, advancementScore = 0;
68.
69. std::vector<Board::Move> playerMoves = state.moveGen(color);
70. std::vector<Board::Move> enemyMoves = state.moveGen(switchPlayerColor(color));
71.
72. Pieces playerPieces = state.getPlayerPieces(color);
73. Pieces opponentPieces = state.getOpponentPieces(color);
74. Pieces *p_playerPieces = &playerPieces;
75. Pieces *p_opponentPieces = &opponentPieces;
76.
77. std::vector<Board::Move> playerJumpsForPiece;
78. std::vector<Board::Move> opponentJumpsForPiece;
79.
80. // helper values to quickly check if a piece is in a certain notable location
81. // back rows for each color player to determine KING-ing
82. // sides indicate limited moves
83. const long long redBackRowGrp = (1LL << 1) | (1LL << 2) | (1LL << 3) | (1LL << 4);
84. const long long blackBackRowGrp = (1LL << 29) | (1LL << 30) | (1LL << 31) | (1LL << 32);
85. const long long sideColumnGrp = (1LL << 5) | (1LL << 13) | (1LL << 21) | (1LL << 12) | (1LL << 20) | (1LL << 28);
86.

```

```

87. // CHECK TERMINAL STATE
88. if (enemyMoves.size() == 0)
89.     return 7999999; // good for us if enemy has no moves left!
90. else if (playerMoves.size() == 0)
91.     return -7999999; // bad for us if we're out of moves!
92.
93. // MAIN LOOP FOR SCORING POSITION
94. for (int piece = 0; piece < 32; piece++)
95. {
96.     // BEGIN POSITION SCORE SECTION
97.     if (color == Color::RED)
98.     {
99.         playerPiece = state.getPieceInSquare(piece, color);
100.        if (playerPiece == MAN)
101.            positionScore += (squareValuesForRed[piece] * MAN_VALUE);
102.        else if (playerPiece == KING)
103.            positionScore += (squareValuesForKing[piece] * KING_VALUE);
104.    }
105.    else if (color == Color::BLACK)
106.    {
107.        playerPiece = state.getPieceInSquare(piece, color);
108.        if (playerPiece == MAN)
109.            positionScore += (squareValuesForBlack[piece] * MAN_VALUE);
110.        else if (playerPiece == KING)
111.            positionScore += (squareValuesForKing[piece] * KING_VALUE);
112.    }
113.    // END POSITION SCORE SECTION
114. }
115.
116. // Check our moves; 1000 points for a safe capture, 2000 points for a multi-jump
117. for (int i = 0; i < playerMoves.size(); i++)
118. {
119.     if (playerMoves.at(i).removalSquare.size() > 1)
120.     {
121.         if (Pieces::outputDebugData)

```

```

122.                     std::cout << " INSIDE EVAL-
123.             3: We " << colorTxt << " can capture multiple pieces on this state! "
124.             << "Start: " << playerMoves.at(i).startSquare << "End: " << playerMoves.at(i).destinationSquare.back()
125.             << " " << std::endl;
126.             captureScore += 2000;
127.         }
128.         else if (playerMoves.at(i).removalSquare.size() == 1)
129.         {
130.
131.             int enemyCaptureSqr = playerMoves.at(i).removalSquare.back();
132.             int enemyCaptureType = state.getPieceInSquare(enemyCaptureSqr, switchPlayerColor(color)); // it's an
133.             enemy piece, what is it's type?
134.
135.             if (enemyCaptureType == KING)
136.             {
137.                 if (Pieces::outputDebugData)
138.                     std::cout << " INSIDE EVAL-3: We can capture enemy KING! " << colorTxt << std::endl;
139.
140.             captureScore += 400;
141.         }
142.         else if (enemyCaptureType == MAN)
143.             captureScore += 200;
144.
145.         int destSqr = playerMoves.at(i).destinationSquare.back();
146.         std::vector<int> adjMoves = state.boardMoveTable[destSqr].moves;
147.
148.         if (color == Color::RED)
149.         {
150.             for (int j = 0; j < adjMoves.size(); j++)
151.             {
152.                 if (adjMoves.at(j) > destSqr) // check enemy MEN and KING below
153.                 {
154.                     int enemyPiece = state.getPieceInSquare(adjMoves.at(j), switchPlayerColor(color));
155.                     if (enemyPiece == MAN || enemyPiece == KING)

```

```

155.                     captureScore -= 100; // not safe
156.                 }
157.             else if (adjMoves.at(j) < destSqr) // we're red, anything above us can only capture if enemy
158.                 KING
159.             {
160.                 int enemyPiece = state.getPieceInSquare(adjMoves.at(j), switchPlayerColor(color));
161.                 if (enemyPiece == KING)
162.                     captureScore -= 100; // not safe
163.                 else
164.                     captureScore += 1000; // we're safe to capture
165.             }
166.         }
167.     else
168.     {
169.         for (int j = 0; j < adjMoves.size(); j++)
170.         {
171.             if (adjMoves.at(j) < destSqr) // check enemy MEN and KING above
172.             {
173.                 int enemyPiece = state.getPieceInSquare(adjMoves.at(j), switchPlayerColor(color));
174.                 if (enemyPiece == MAN || enemyPiece == KING)
175.                     captureScore -= 100; // not safe
176.             }
177.             else if (adjMoves.at(j) > destSqr) // we're black, anything below us can only capture if enemy
178.                 KING
179.             {
180.                 int enemyPiece = state.getPieceInSquare(adjMoves.at(j), switchPlayerColor(color));
181.                 if (enemyPiece == KING)
182.                     captureScore -= 100; // not safe
183.                 captureScore += 1000; // we're safe to capture
184.             }
185.         }
186.     }
187. }
188.

```

```

189.     // BEGIN CASUALTY SECTION
190.     for (int j = 0; j < enemyMoves.size(); j++)
191.     {
192.         //std::cout << "we are here" << std::endl;
193.         if (enemyMoves.at(j).removalSquare.size() > 1)
194.         {
195.             if (Pieces::outputDebugData)
196.                 std::cout << " INSIDE EVAL-
3: Enemy can capture multiple pieces, avoid!" << colorTxt << std::endl;
197.
198.             casualtyScore -= 40000; // we lose too much, really bad
199.         }
200.         else if (enemyMoves.at(j).removalSquare.size() == 1)
201.         {
202.             // friendly piece is captured, what is it's type?
203.             int capturedPieceType = state.getPieceInSquare(enemyMoves.at(j).removalSquare.at(0), color);
204.             if (capturedPieceType == KING)
205.             {
206.                 if (Pieces::outputDebugData)
207.                     std::cout << " INSIDE EVAL-3: Enemy can capture a KING, avoid!" << colorTxt << std::endl;
208.
209.                 casualtyScore -= 4000; // we lose a KING, a valuable piece
210.             }
211.             else if (capturedPieceType == MAN) // we lose one MAN
212.             {
213.                 int opponentDestinationSqr = enemyMoves.at(j).destinationSquare.at(0);
214.                 std::vector<int> adjMoves = state.boardMoveTable[opponentDestinationSqr].moves;
215.
216.                 if (color == Color::RED)
217.                 {
218.                     // if we are RED, opponent is Black; if BLACK enemy lands on our back row, avoid at all cost
219.                     // we are therefore trying to minimize the chance of an enemy getting a KING
220.                     if ((1 << opponentDestinationSqr) & redBackRowGrp)
221.                         casualtyScore -= 5000;
222.                     // opponent lands on their own back row; not so bad but we can't retaliate so avoid
223.                     else if ((1 << opponentDestinationSqr) & blackBackRowGrp)

```

```

224.         casualtyScore -= 2000;
225.         // opponent lands on one of the side squares, where we cannot retaliate. Avoid as well
226.         else if ((1 << opponentDestinationSqr) & sideColumnGrp)
227.             casualtyScore -= 2000;
228.         else
229.         {
230.             // if (diffInNumMen >= 1)
231.             // {
232.             for (int j = 0; j < adjMoves.size(); j++)
233.             {
234.                 int ourPiece = state.getPieceInSquare(adjMoves.at(j), switchPlayerColor(color));
235.                 // check if we have a king below; we're red, only our king can go upwards
236.                 if (adjMoves.at(j) > opponentDestinationSqr)
237.                 {
238.                     if (ourPiece == KING)
239.                         captureScore += 1000; // can retaliate
240.                 }
241.                 // we're red, we can retaliate with MAN or KING if enemy is above
242.                 else if (adjMoves.at(j) < opponentDestinationSqr)
243.                 {
244.                     if (ourPiece == MAN || ourPiece == KING)
245.                         captureScore += 1000; // can retaliate
246.                 }
247.                 else
248.                     captureScore += 0; // we cannot capture
249.             }
250.             // }
251.         }
252.     }
253.     else
254.     {
255.         // if we are BLACK, opponent is Red; if RED enemy lands on our back row, avoid at all cost
256.         // we are therefore trying to minimize the chance of an enemy getting a KING
257.         if ((1 << opponentDestinationSqr) & blackBackRowGrp)
258.             casualtyScore -= 5000;
259.         // opponent lands on their own back row; not so bad but we can't retaliate so avoid

```

```

260.             else if ((1 << opponentDestinationSqr) & redBackRowGrp)
261.                 casualtyScore -= 2000;
262.             // opponent lands on one of the side squares, where we cannot retaliate. Avoid as well
263.             else if ((1 << opponentDestinationSqr) & sideColumnGrp)
264.                 casualtyScore -= 2000;
265.             // if we've gotten this far, we lose one MAN and opponent lands somewhere we can retaliate
266.             // We ask - Can we? If yes, do it if we have piece parity or an advantage of more pieces
267.             else
268.             {
269.                 // if (numPlayerTotalPieces >= numEnemyTotalPieces)
270.                 // {
271.                     for (int j = 0; j < adjMoves.size(); j++)
272.                     {
273.                         int ourPiece = state.getPieceInSquare(adjMoves.at(j), switchPlayerColor(color));
274.                         if (adjMoves.at(j) < opponentDestinationSqr) // check our KING above
275.                         {
276.                             if (ourPiece == KING)
277.                                 captureScore += 1000; // can retaliate
278.                         }
279.                         // we're black, if enemy is below we can retaliate with MAN or KING
280.                         else if (adjMoves.at(j) > opponentDestinationSqr)
281.                         {
282.                             if (ourPiece == MAN || ourPiece == KING)
283.                                 captureScore += 1000; // can retaliate
284.                             }
285.                             captureScore += 0; // we cannot retaliate
286.                         }
287.                     // }
288.                 }
289.             }
290.         }
291.     }
292. }
//END CASUALTY SECTION
294.

```

```
295.     int compositeScore = numMenScore + numKingsScore + advancementScore + positionScore + captureScore + casualt
  yScore;
296.
297.     return compositeScore;
298. }
```

3.7.3 Evaluation Function One (EF-1)

Please refer to Appendix A, Algorithm.cpp for a complete implementation of this evaluation function, which was implemented by David Torrente. While this evaluation function will be covered in more detail in the respective authors report, I want to take a moment to provide a brief description of its goals using the final state.

This EF makes a more careful examination of the opponent's score than others. It tries to assign different weights to pieces and scores depending on the state of the game and the disparity between pieces, to adjust the playstyle. This is an interesting approach and can lead to unique behavior – for example it values kings more highly in the early stages and tries to be more aggressive. Depending on whether the player has a lead in number of pieces, the game is either more aggressive or more defensive (starting around line 195 and onward).

3.7.4 Evaluation Function Two (EF-2)

Please refer to Appendix A, Algorithm.cpp for a complete implementation of this evaluation function, which was implemented by Randall Henderson. While this evaluation function will be covered in more detail in the respective authors report, I want to take a moment to provide a brief description of its goals using the final state.

EF-2 appears to value Kings quite highly (3000 for King vs 560 for a piece on our side, 1000 vs 100 respectively for the opponent). It also makes use of an array containing values for each square on the board. It encourages kings to move to the center while regular pieces head for the opponent's back row.

3.8 UI Design Considerations

Games can be thought provoking, challenging, used as a tool for learning, or a form of competition. But fundamentally – games are fun!

We wanted to provide a pleasant user experience with this program, where we approximate a real-life checkers game as much as possible. The application provides several additional options to the user where they may either play against another human or test themselves against one of the evaluation functions.

While we are limited in our options on a CLI terminal, we do have some avenues to make the user interface and game experience fun. The Texas State Linux hosts support ANSI colors; special colors and color combinations are used throughout the program to both guide the user and brighten the experience that is normally just black and white text.

We display a colorful welcome message with the authors' names highlighted for visibility.

```
Welcome to the Checkers AI Program.  
Authors: David Torrente (dat54@txstate.edu), Randall Henderson (rrh93@txstate.edu), Borislav Sabotinov (bss64@txstate.edu).  
Re-run this program with -h or -help CLI argument to see a help menu or refer to README for instructions.
```

Figure 9: Welcome message

Whenever the user is prompted for a choice, the choices are highlighted in a pleasant blue color that is also readable. This guides the user by drawing their eye to their available options.

```
Your choice (1, 2, 3, or 4):
```

Figure 10: Highlighting choices to guide the user

The checkers board itself is visually represented, with all pieces in their color, in full. This provides a good user experience as it allows the player to immediately and intuitively understand the state of the game. They will know by looking at the printed board where their pieces are in relation to the opponent in the same way as they would looking at a real checkers board (or a fancy HTML web interface). The squares are also numbered, providing players with instant access to the location of their pieces on the board. Initially we discussed having two boards – one with pieces on it and another as a lookup table with square numbers. I advocated for the expanded board with layered, contextual information.

	r	r	r	r	r
1		2		3	4
r	r	r	r	r	
5	6	7	8		
r	r	r	r	r	
9	10	11		12	
13	14	15		16	
				b	
	17	18	19		20
b	b	b	b		
21	22	23		24	
b	b	b	b		b
25	26	27		28	
b	b	b	b		
29	30	31		32	

Figure 11: A checkers board as represented on the CLI, with color enabled

End game states are also considered – reporting the outcome of the game in a fun manner can improve the experience. Should the game end in a draw, as most games in checkers and in our program do, we display a fun graphic using ASCII characters. The first line in figure 10 is simply text, stating the outcome. The second line may require some imagination, but it depicts two boxers with their fists up squaring against one another. The third line is a reference to Call of Duty (a famous video game) and its well-known “Mission Failed” soundbite.

```
DRAW!!!
Red - ( „ •`_• ) „   „( `Д' „) - Black
Mission FAILED...We'll get em next time!
```

Figure 12: Draw end state ASCII art

Now suppose one of the players wins the game – that is a pretty exciting outcome, and we can have some fun here. The first line in figure 11 again states the outcome using text. The second line shows the winning player (in this case Red) sweating and posing with their muscles flexed. The third line taunts the losing player. And the fourth and last line shows the losing player's supposed reaction – flipping a table!

```
RED WINS!!!
RED Player: <(-_-')>
But most importantly, BLACK looooses (booooo!)
BLACK Player: (J°□°) J ~ ──
```

Figure 13: One player wins end state ASCII art

At the end of the game, a simple but pleasant graphic saying “Goodbye!” to the user is displayed.



Figure 14: End screen

4 Classes Deep-Dive

Refer to Appendix A for complete source code, which contains helpful comments. I will refer to file name and line number below.

We may say that class `Simulation` has one or more `Games`. A `Game` has two `Players` and one `Board`. A `Board` has two `Pieces`. A `Player` has an `Algorithm` and one `Pieces` instance (to represent the player's pieces for the game).

4.1 Pieces

This class represents a set of 12 initial pieces in their starting positions. It contains an enum class `Color`, which helps us avoid “magic numbers” – for example 1 is Red and -1 is Black but using an enum we can refer to them by name. We decided to use an enum class instead of an enum due to stronger type checking. Suppose we had two enums, one `Color` and another `Fruit`; if red were equal to 1 and apple also to 1, they would be equal. But with an enum class we avoid this potential issue as their type would be different.

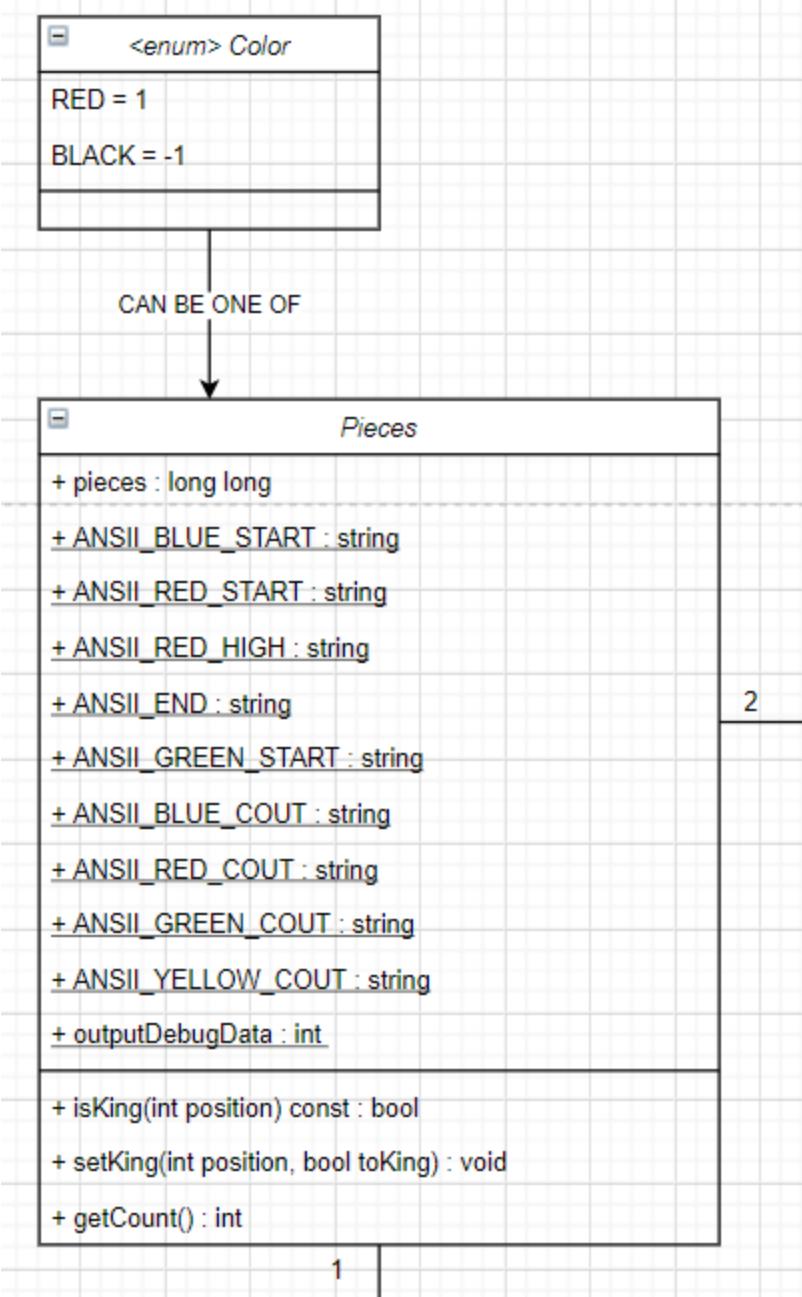


Figure 15: Pieces UML Class Node

4.1.1 Data Structures Used

The pieces are represented using a long long type, which is 64 bits. The lower 32 bits are used to determine the location of the piece. The upper 32 bits tell us if a piece is a king (1) or not (0). This is accessed by using an offset of 31 plus the number of the piece.

King bits

Piece position bits

Starting position for Red player, who is on top of the board, is 4095. All the lower 12 bits are set and they are not a King yet, so upper 32 bits are empty. We can intuitively see this is correct, two to the power of 12 is 4096 if the 12th bit (using 0 indexing) is set. Since all bits below (0 to 11) are set, we simply do $4096 - 1 = 4095$, which is what we expect. Black is a much higher starting value (4,293,918,720) as it occupies the upper 12 bits of the lower 32 bits available.

Black Player starting value: **MSB** 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 1111 0000 0000 0000 0000 0000 **LSB**

The `Pieces` class contains a few simple helper functions and some static constant strings to assist us with displaying color (if desirable) throughout the application. This is because the `Pieces` header is the most fundamental and at the top of the hierarchy. All other classes either directly import it or end up having access to it indirectly by importing a class which uses it.

Of import is the `isKing()` function which allows us to tell if a piece is of type King. We first must determine if the position is occupied via a separate call; if it is, we can call this helper function. We use the position of the piece, let us say 4, plus an offset of 31 for a total of 35. We shift right 35 bits and we bitwise-AND with 1 – if the result is 1, the piece is a King. Otherwise, it is a standard piece.

4.2 Board

Class board is used to represent the entire board and its current state. It is based on an 8 x 8 grid, with 32 possible spaces. Two piece data types are the primary memory consumers of this class. This class also includes a static member that acts as a move guide. This move guide determines the possible moves for a square on the board, not for a piece.

Board contains two helper structs – Move and BoardMoveTable. Move is used to track the moves we make on the board. It contains a starting square, a vector of squares to move across (if jumping), where the last square is the destination of the piece we are moving. It also contains a vector listing the squares of the enemy pieces we captured, if any. BoardMoveTable, on the other hand, contains static data types to significantly speed up how we search for moves given a position.

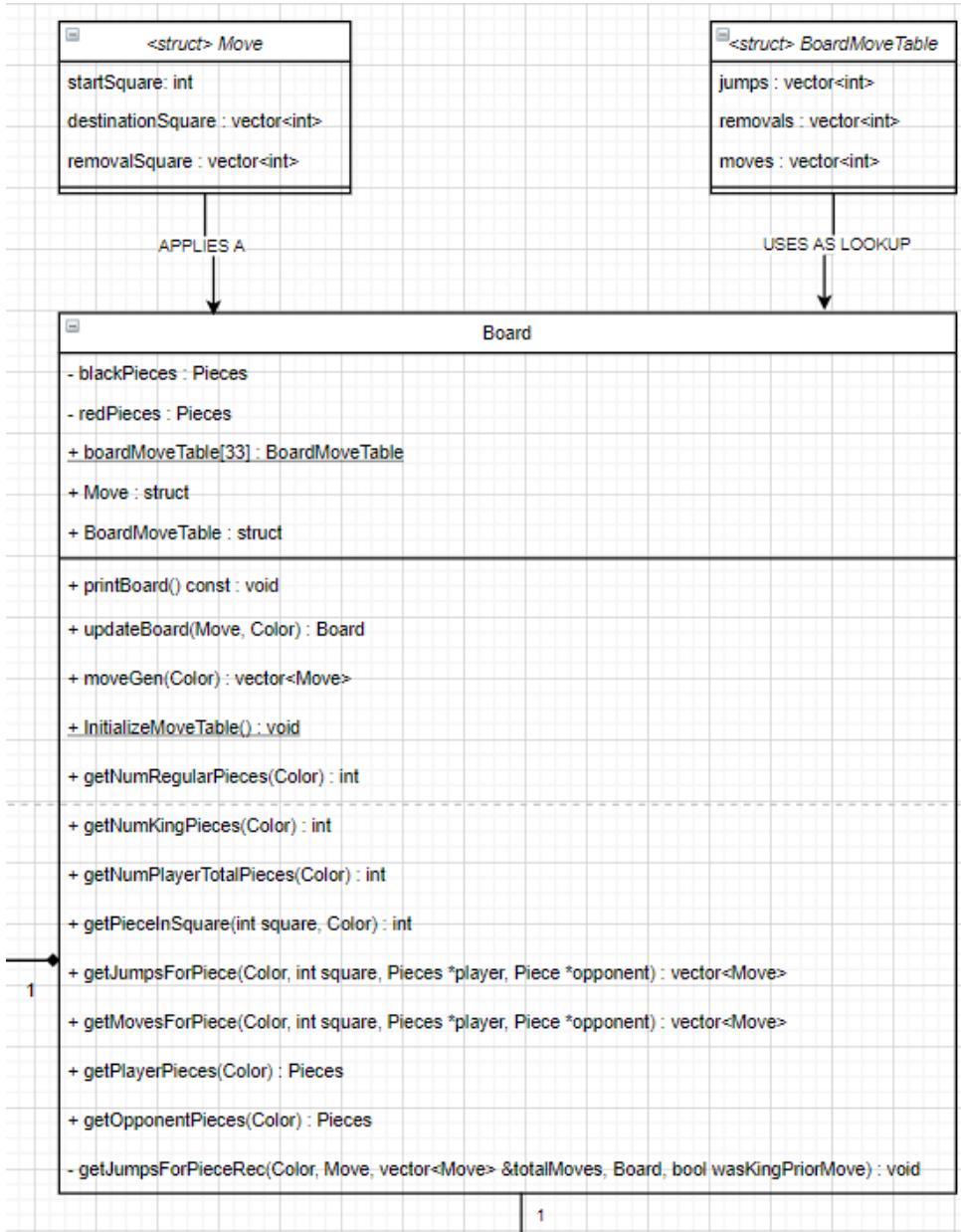


Figure 16: Board UML Class Node

4.2.1 Data Structures Used

The Board class makes clever use of a static move table – `boardMoveTable`. It provides us with $O(1)$ lookup time for the possible moves and jumps for each square.

Here is an example:

```
boardMoveTable[16].jumps.push_back(7);
boardMoveTable[16].removals.push_back(11);
boardMoveTable[16].jumps.push_back(23);
boardMoveTable[16].removals.push_back(19);
boardMoveTable[16].moves.push_back(11);
boardMoveTable[16].moves.push_back(12);
boardMoveTable[16].moves.push_back(19);
boardMoveTable[16].moves.push_back(20);
```

We see that the lookup table contains the following information accessible for square 16:

- It can move to squares 11, 12, 19, 20 (depending on the color and type of the piece)
- It can jump to 7, which would remove the piece in square 11
- It can jump to 23, removing piece in square 19

If we closely examine the figure below, we see this is true because 16 is the 2nd to last row from the right, so it cannot jump to the right. It may only jump across 11 to 12 or across 19 to 23. In this example, we would determine the type of piece that is on the board at 16 (RED, MAN), which would allow us to use the look up table properly. Because the piece is red and a standard piece, it may only move downward diagonally to either 19 or 20 below.

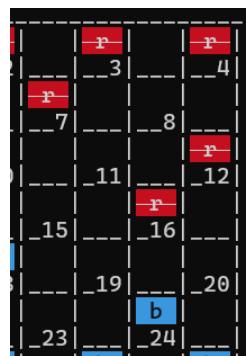


Figure 17: Moves and Jumps for Square 16

A Board is represented as an 8x8 grid though we only ever use 32 spots. When we print the board, we make the individual squares large enough to contain a colored piece and the square number, the latter of which does not change. A move is represented by struct Move, which has a starting square, a vector of squares to move to in sequence, and a vector of captured/removed enemy pieces. If we examine the figure below, we see Black moved from 25 to 18 and captured 1 piece (on 25). Thus move.startSquare = 25, move.destinationSquare.at(0) = 18, and move.removalSquare.at(0) = 22. These values would be determined by calling getMovesForPiece() or getJumpsForPiece(), which would in turn leverage the O(1) lookup table to determine where a piece may move to or jump. Jumps are mandatory so we must take them if offered, therefore moves for a piece are not generated if a jump is available.

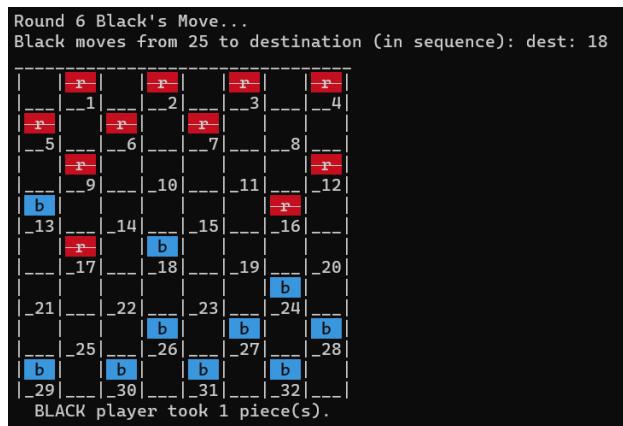


Figure 18: A Board with a Black move from 25 to 18 (a jump) and a capture at 22

Of particular note is the `getJumpsForPieceRec()`, a recursive function which gets a full jump chain (in case more than one is available in succession). The function keeps track if we are starting a jump or in the middle of it by looking at the size of the destination vector, which contains all the squares on which the player will land. We determine if a piece is of type King to be able to tell which moves are available. Then in the main for-loop we calculate if a jump is available. The loop appears to be $O(n)$ but is actually $O(1)$ complexity because we only loop up to `boardMoveTable[piece].jumps.size()`. Recall from previous discussions above, Board uses a lengthy look-up table to store all possible moves and jumps for each square individually. Thus, this size is known up-front and will never grow, giving us a constant rather than variable look-up time!

The `moveGen` function is important because it is how we get a list of available moves for a player. We pass in the color of the player and use it to check `Pieces`, where the player's pieces are stored as a long long bitmap. We go through each piece the player has on the board and use the $O(1)$ lookup table to determine where the pieces may move. It is important to note that if one or more jumps are available, only the jumps will be returned and not any other moves.

Here is the main for-loops, which drive this process. Both only loop up to and including 32. A bit offset based on the piece counter is used, so we can check each position in turn by shifting the bitmap and using bitwise-AND to compare if the square is occupied.

```

1. // Go through all 32 squares and see if it is one
2. // of the appropriate pieces belonging to player.
3. for (int pieceIter = 1; pieceIter <= 32; pieceIter++)
4. {
5.     // The offset is used to align to 0 - 31, but
6.     // the board in checkers is 1 - 32. Use an offset
7.     // here to align to the position bits properly.
8.     bitOffset = pieceIter - 1;
9.
10.    if (((playerPieces->pieces) >> bitOffset) & 1) == 1)
11.    {
12.        // Check for possible jump moves first.
13.        returnedMoves = getJumpsForPiece(color, pieceIter, playerPieces, opponentPieces);
14.        totalMoves.insert(totalMoves.end(), returnedMoves.begin(), returnedMoves.end());
15.    }
16. }
17. // Go through all 32 squares and see if it is one
18. // of the appropriate pieces belonging to player.
19. // Do this only if no jumps are possible. This is controlled
20. // by the if condition here. No need to get moves if there are
21. // jumps already found.
22. if (totalMoves.size() == 0)
23. {
24.     for (int pieceIter = 1; pieceIter <= 32; pieceIter++)
25.     {
26.         // The offset is used to align to 0 - 31, but
27.         // the board in checkers is 1 - 32. Use an offset
28.         // here to align to the position bits properly.
29.         bitOffset = pieceIter - 1;
30.
31.         // Check if player is in this space
32.         if (((playerPieces->pieces) >> bitOffset) & 1) == 1)
33.         {
34.             // Check for non jump moves here.
35.             returnedMoves = getMovesForPiece(color, pieceIter, playerPieces, opponentPieces);
36.             totalMoves.insert(totalMoves.end(), returnedMoves.begin(), returnedMoves.end());

```

```
37.         }
38.     }
39. }
40.
```

4.3 Player

The Player class defines a virtual player, in the case of this program an AI; no human player is used – that is to say, no manual input is sought from the user at any time when this class is used. The “player” will use an algorithm in combination with an evaluation function to determine the best move it wants to make. The player class has several straightforward helper functions to get the number of pieces a player has, or has taken, to get the player’s color, and so on. Of particular note is the overloaded constructor, which allows us to set what algorithm, evaluation function, and depth the player will leverage for the game. We also have four variables to keep track of the number of expanded and leaf nodes, two for each algorithm.

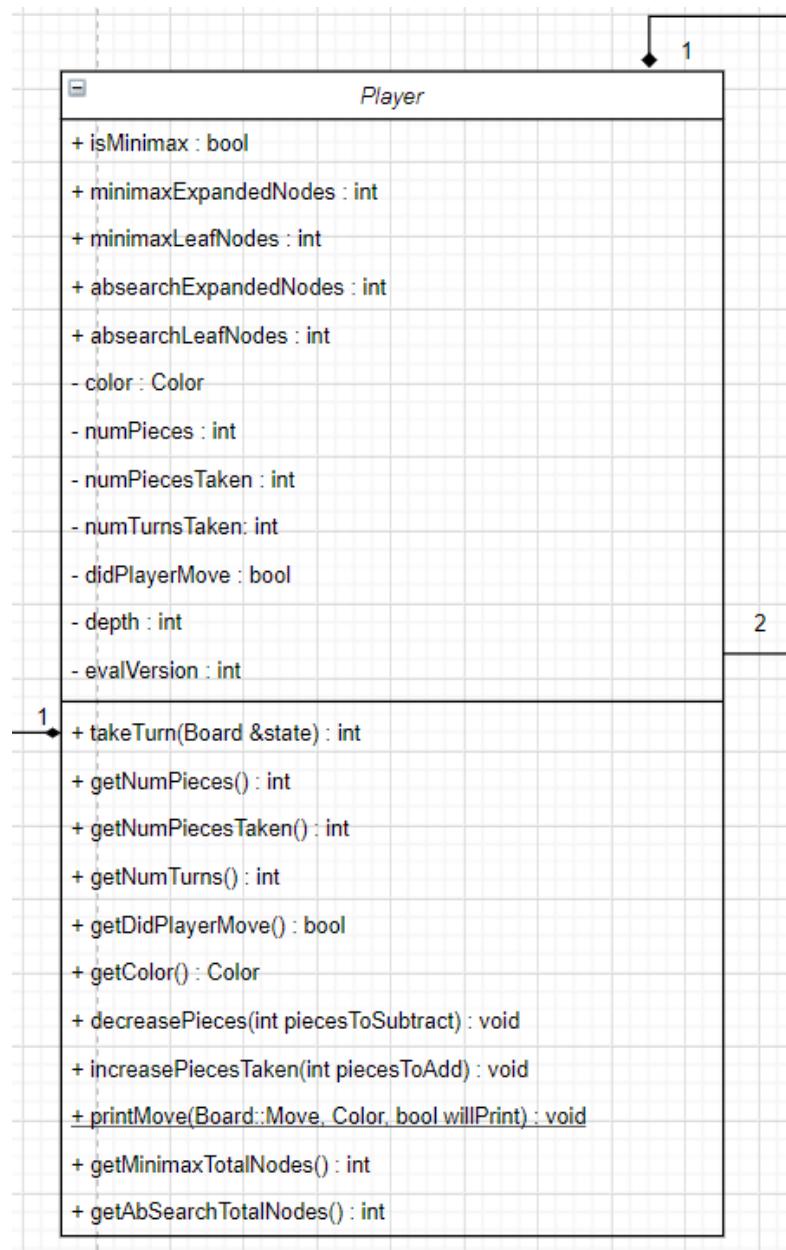


Figure 19: Player UML Class Node

4.3.1 Data Structures and Algorithms

The `takeTurn(Board &state)` method is where the interesting logic takes place. Here we instantiate an `Algorithm` with a depth, an evaluation function version, and a reference to the calling player. We check if `isMinimax` is set – if it is, the player will use minimax-a-b, otherwise it will use alpha-beta-search. We call the appropriate algorithm with its required initial values and get back a `Result` structure, which contains a move the player will make along with a value. If the player is not out of moves, we update the game state with the player's move returned to us by one of the algorithms. Note that class `Game` calls `player.takeTurn(Board &state)` and the state is updated directly as we passed it in by reference and not by value. When a player makes their move, we increment the turn counter and print the new board, which contains the players move.

4.4 Game

The `Game` class represents a checkers Game. A game consists of a board, two players (red and black), and 12 checkers pieces for each player. A game ends when either player loses all their pieces or is blocked and has no further moves available. The `startGame()` function will trigger the process and no input from the user is required. Red and Black players will be created. Each player will be given 12 checker pieces. The pieces will be appropriately placed on the board. Each player will execute their strategy using the `Algorithm` class. We use a `GameOver` enum class to avoid magic numbers and define what constitutes a “game over” condition. In our case, either one of the players may win (`BLACK_WINS` or `RED_WINS`), the game may end in a `DRAW`, or the game is `NOT_DONE` yet.

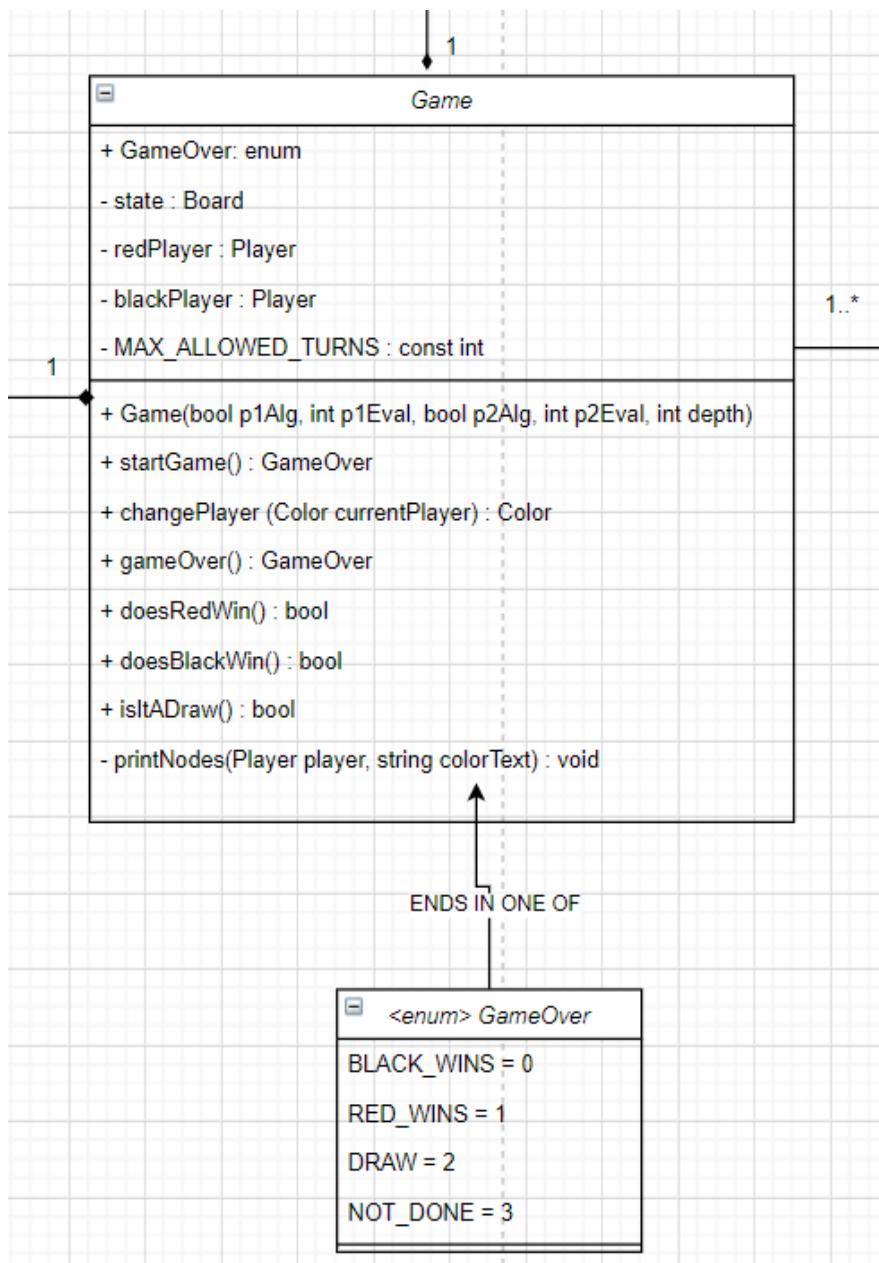


Figure 20: Game UML Class Node

4.4.1 Data Structures and Algorithms

Game's overloaded constructor allows us to pass in the player's respective algorithm, evaluation function, and the overall depth from main.cpp (where we obtain the user's input) to Simulation, which may start a series of games, to Game. From Game to Player. And from Player to Algorithm (which only needs the evaluation version and depth).

There are no special data structures in this class. It has a few straightforward helper functions to either change the color of the current player, check win conditions, and print some statistics about the game after it concludes.

Of interest is the startGame() function – it returns a GameOver struct. It has one main while(true) loop. We only break out of this loop when a game reaches a satisfactory conclusion – either one player wins or we exceed the number of permitted turns (80 per player for a total of 160 per game), in which case we get a draw. In this function, Black moves first. After the Black player takes a turn using the Player::takeTurn(Board &state) function discussed in section 4.3, we increase the player's captured pieces counter and decrease the opponent's piece counter if any jumps were present. We then unconditionally check all win conditions. This is because even though only Black moved, they may have boxed themselves in and ran out of moves leading to a game over event or it may be their last permitted turn. Next the Red player takes a turn and again we check win and loss conditions for all players.

4.5 Algorithm

This class encapsulates the algorithmic approach the AI uses to play Checkers. There are only two major algorithms supported.

1. Minimax-a-b: a depth first, depth limited search procedure from the Richard and Knight book. The minimax function has a heuristic value for leaf nodes (end nodes and nodes at the maximum permitted depth). Intermediate nodes get their value from a child/successor leaf node. It is a recursive function and instead of alpha and beta we use passThreshold and useThreshold values, which are swapped and inverted at each recursive call.
2. Alpha-Beta-Search: uses indirect recursion and does not require us to swap or negate values. There are two helper functions, one MAX and one MIN for each respective player. We stop evaluating a branch when at least one option is found to be worse than a previously examined move.

Note that both algorithms return the same result given the same evaluation function. This means that while the internal implementation of the two algorithms is quite different, they behave in the same way and will play a game the same turn by turn, leading to the same outcome and the same number of expanded nodes. This is a good thing and exactly what we expect to see; it indicates that the two algorithms are correctly implemented within the context of this program.

Class Algorithm uses a Result structure, which consists of an integer value (the score we give a particular state) and a Board::Move bestMove struct (the move the player needs to make to head in that direction in the game tree).

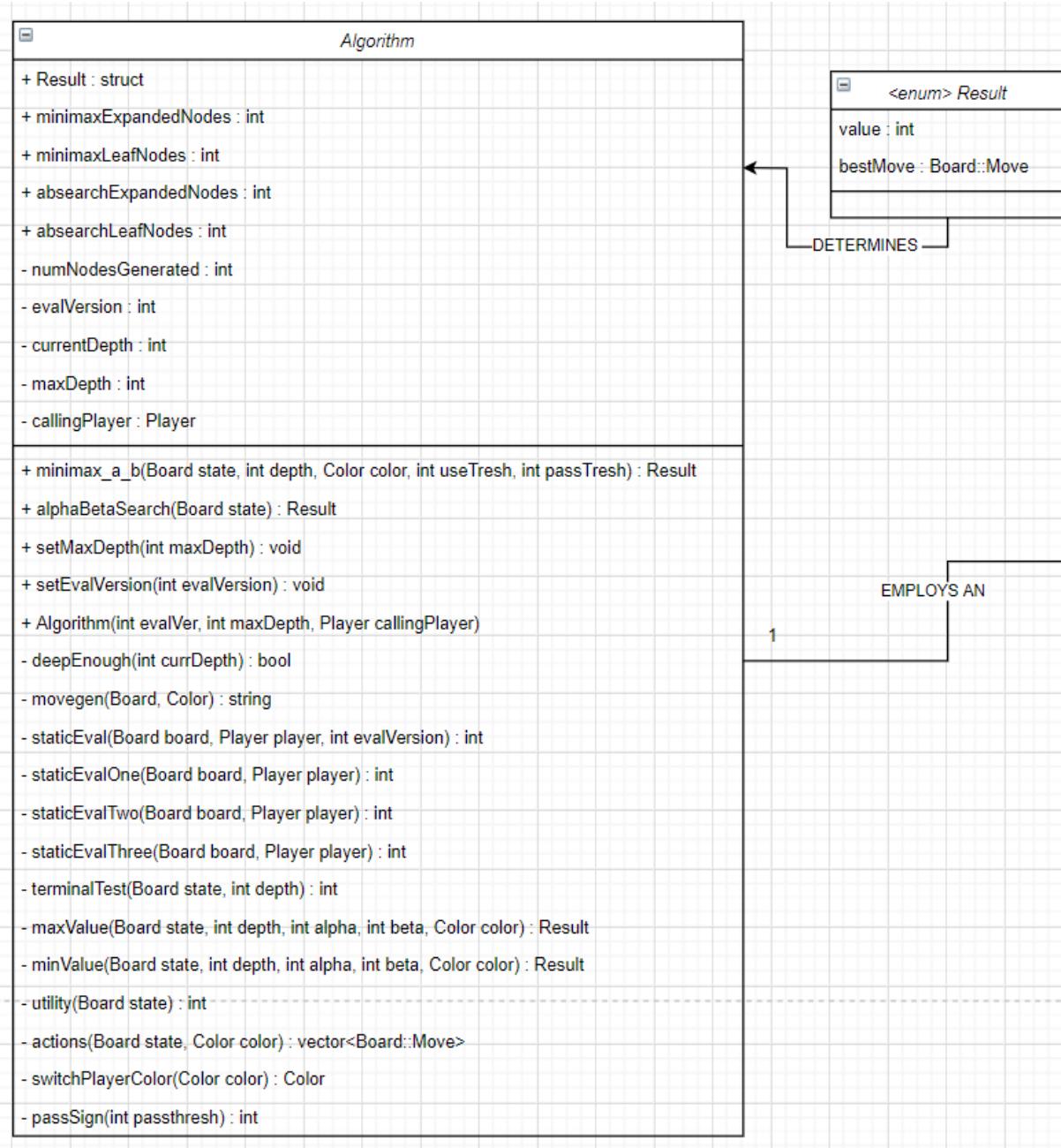


Figure 21: Algorithm UML Class Node

4.5.1 Data Structures and Algorithms

The evaluation functions are covered in more detail in section 3.7 above.

In minimax-a-b, we increment the number of leaf nodes if we are deep enough or at a terminal state. The number of intermediate expanded nodes is captured before we make the recursive call. In the for-loop, we must first create a new Board (i.e., state) by applying the move that is at `successors.at(successorsIndex)`. This is the move we pass into the recursive call of minimax-a-b. We decrement the depth, switch the player color, and both swap and invert `passThresh` and `useThresh`. Once this call returns, we invert the value that was obtained and store it in `newValue`. We first check if this is strictly greater (better) than `passThresh`, if it is, we found a better move. Next, we check if `passThresh` is greater or equal to `useThresh` and if it is, we already have the best move on the branch and we need not explore the branch deeper, so we terminate the for loop and return with the value and move. To deal with odd depths and keep values in the right order, we check if the original calling player's color is the same as the color passed into the function. If it is, there is no issue. Otherwise, we must negate the `result.value` we obtain. The algorithm remains correct for odd depths; but failure to guard against even depth inversion leads to undesirable behavior at even depths greater than four. In minimax-a-b, we use 8 million and negative 8 million for the best and worst cases instead of `INT_MAX` and `INT_MIN` maximum values. This is because inside the algorithm, we use negation. Because 2's complement is used, negating `INT_MIN` leads to undefined behavior. We use the lower reasonable values to get around this.

Alpha-Beta-Search does not have this risk and we can use `INT_MIN` and `INT_MAX`, as there is no negation. I use `std::numeric_limits<int>::min()` and `std::numeric_limits<int>::max()` respectively. The `minValue()` and `maxValue()` functions are effectively the same, just inverses of each other. We check if we are deep enough. Otherwise we generate a list of actions and check if we are at a terminal state. We set `result.value` to either `min` (if inside `maxValue`, as that is the worst case) or `max` (if inside `minValue`, as that is now the worst case). Inside the for-loop we create a new hypothetical board by applying the move from our list of available moves (or actions) by using `listOfActions.at(actionIndex)`. We do not negate or swap alpha and beta, they are passed in by value as-is. Alpha is associated with MAX nodes and cannot decrease; it starts out as the worst-case for MAX. Beta tracks MIN nodes and cannot increase; it starts out as the worst case for MIN.

This was the algorithm I implemented for this project; below are snapshots from both referenced textbooks. It is worth mentioning that the effectiveness of pruning depends on the order in which we examine available moves. If our luck is bad, the first move we examine is the worst available and each subsequent move is slightly better, where the best move is at the end. In this case, we are essentially performing a depth first search with no pruning as we keep updating the best move and going through the tree. And if the nodes are perfectly ordered, "the number of terminal nodes considered by a search to depth d using ... pruning is ... equal to twice the number of terminal nodes generated by a search to depth $d/2$ without alpha-beta." [Knuth and Moore, 1975]

The effectiveness of the alpha-beta procedure depends greatly on the order in which paths are examined. If the worst paths are examined first, then no cutoffs at all will occur. But, of course, if the best path were known in advance so that it could be guaranteed to be examined first, we would not need to bother with the search process. If, however, we knew how effective the pruning technique is in the perfect case, we would have an upper bound on its performance in other situations. It is possible to prove that if the nodes are perfectly ordered, then the number of terminal nodes considered by a search to depth d using alpha-beta pruning is approximately equal to twice the number of terminal nodes generated by a search to depth $d/2$ without alpha-beta [Knuth and Moore, 1975].

A doubling of the depth to which the search can be pursued is a significant gain. Even though all of this improvement cannot typically be realized, the alpha-beta technique is a significant improvement to the minimax search procedure. For a more detailed study of the average branching factor of the alpha-beta procedure, see Baudet [1978] and Pearl [1982].

The idea behind the alpha-beta procedure can be extended to cut off additional paths that appear to be at best only slight improvements over paths that have already been explored. In step 4(d), we cut off the search if the path we were exploring was not better than other paths already found. But consider the situation shown in Fig. 12.6. After examining node G, we see that the best we can hope for if we make move C is a score of 3.2. We know that if we make move B we are guaranteed a score of 3. Since 3.2 is only very slightly better than 3, we should perhaps terminate our exploration of C now. We could then devote more time to exploring other parts of the tree where there may be more to gain. Terminating the exploration of a subtree that offers little possibility for improvement over other known paths is called a *futility cutoff*.

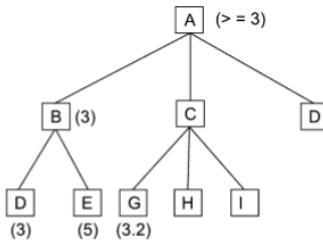


Fig. 12.6 A Futility Cutoff

Figure 22: Minimax-a-b Futility Cutoff

```

function ALPHA-BETA-SEARCH(state) returns an action
  v  $\leftarrow$  MAX-VALUE(state,  $-\infty$ ,  $+\infty$ )
  return the action in ACTIONS(state) with value v
function MAX-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v  $\leftarrow -\infty$ 
  for each a in ACTIONS(state) do
    v  $\leftarrow$  MAX(v, MIN-VALUE(RESULT(s,a),  $\alpha$ ,  $\beta$ ))
    if v  $\geq \beta$  then return v
     $\alpha \leftarrow \text{MAX}(\alpha, v)$ 
  return v
function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v  $\leftarrow +\infty$ 
  for each a in ACTIONS(state) do
    v  $\leftarrow$  MIN(v, MAX-VALUE(RESULT(s,a),  $\alpha$ ,  $\beta$ ))
    if v  $\leq \alpha$  then return v
     $\beta \leftarrow \text{MIN}(\beta, v)$ 
  return v

```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Figure 23: Alpha-Beta-Search Algorithm

Both minimax-a-b and alpha-beta-search correctly prune a branch that is not worth exploring as described in the textbook.

4.6 Simulation

The simulation class hands information to Game as needed and instantiates (and deletes when done) Games as needed. It gets back the GameOver struct from class Game once a game is complete, then prints the results along with some game statistics to the screen.

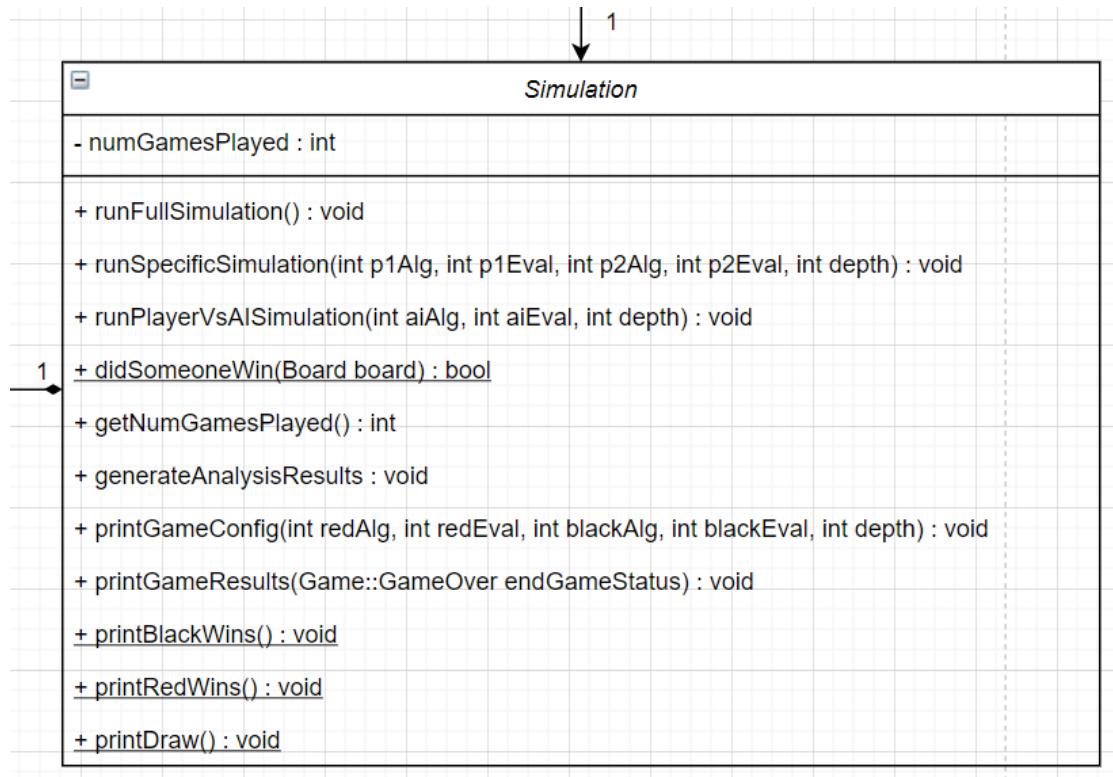


Figure 24: Simulation UML Class Node

4.6.1 Data Structures and Algorithms

To perform a full, exhaustive simulation (with limited redundancy), five nested for loops are used. The outer drives the depth. The first player's algorithm, the third the second. Fourth and fifth loops are for the evaluation functions of the first and second player, respectively. The `playerVsAI` function is interesting because we mix and match strategies – we use both a player with an algorithm for the AI and obtain user input from the console for the human player's moves.

4.7 Main

This is the main driver of the application and where we obtain the user's input for the simulated games. Helper functions are used to print prompts to the CLI and obtain user input. We mostly obtain user input and send to Simulation, which then takes care of the rest. In the case of a manual game, however, we cannot use the Game or Player classes. We interface directly with Board, get the list of available moves, prompt the user for a selection, and update the board manually. The game has no limit – it continues until one player wins or the program is terminated.

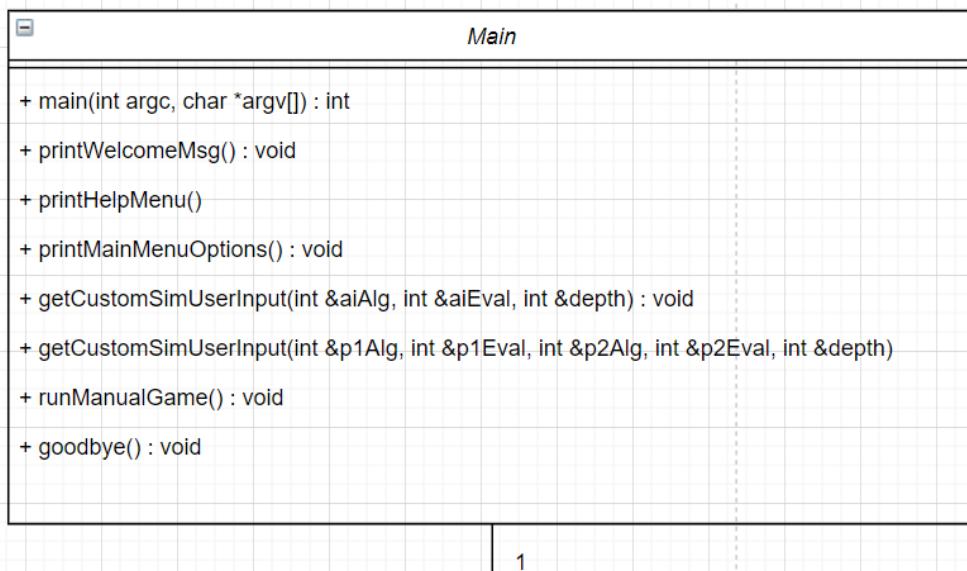


Figure 25: Main application driver

5 Sample Runs

The program prints the same content to the console at the beginning of each run. Both the welcome message and the CLI messages printed while parsing the knowledge base file do not differ between each subsequent run. A complete listing of this output may be found at the end of this report, under “[Appendix B: Complete Sample Output](#).” Games 1 and 10 contain a complete game path from start to finish; game 1 is with depth two and game 10 with depth 4. The output for all other games is similar enough that these examples are enough to provide an accurate representation of the game path generated by the application. The full output for all 18 simulated games would be too long to include in the report, so they are provided separately as log files. The program always displays a welcome message to the user, parses the KB text file and prints contents to the console as it does so, and pauses. After the user hits Enter, it asks if the user wants to display the KB in human readable output. As such, the beginning section is only available in Appendix B to avoid repeating hundreds of lines.

Complete output of this program was obtained on eros.cs.txstate.edu. Program was invoked normally but standard output was piped to tee. Tee is a command which reads our standard I/O and writes it to both standard output and a text file, in our case a log file.

This section aims to comprehensively cover all options of the implemented system. The complete output of a game or a simulation of a series of games is quite large (tens of thousands of lines long). As such, full output files are provided along with the report in two directories – one for runs at depth two and another for runs at depth four.

5.1 Sample Run #1: ALPHA-BETA-SEARCH with EF-1 vs ALPHA-BETA-SEARCH with EF-3 at Depth 2

The screenshots capture only the relevant portion of the output for this sample run. For complete program output, please refer to Appendix B.

```
Round 80 Black's Move...
Black moves from 5 to destination (in sequence): dest: 1

| B | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|
| 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|
| 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|
| 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|
| 21 | 22 | 23 | 24 | 25 |
|---|---|---|---|---|
| 25 | 26 | R | R | 27 |
|---|---|---|---|---|
| b | 30 | 31 | . | 32 |
|---|---|---|---|---|
| 29 | 30 | 31 | . | 32 |

RED Leaf Nodes: 1604
RED Expanded Nodes: 2511
RED Total Nodes: 4115

BLACK Leaf Nodes: 1640
BLACK Expanded Nodes: 1950
BLACK Total Nodes: 3590

DRAW!!!
Red - (я •••) - Black
Mission FAILED...We'll get em next time!
Red player alg: Alpha-Beta-Search, eval: 1
Black player alg: Alpha-Beta-Search, eval: 3
Depth: 2
Minimax-a-b took: 0ms
ABSearch took: 102.783ms
```

Figure 26: Sample Run 1

5.2 Sample Run #2: ALPHA-BETA-SEARCH with EF-1 vs ALPHA-BETA-SEARCH with EF-3 at Depth 4

In this sample run we run a game with ABS, EF-1 vs EF-3. End up in the back-and-forth state mentioned earlier, where black moves between 10 and 7 repeatedly and Red between 17 and 13. At around turn 8 we see Red can capture black by jumping over 19 and land on 23, which is what it does (as jumps are mandatory).

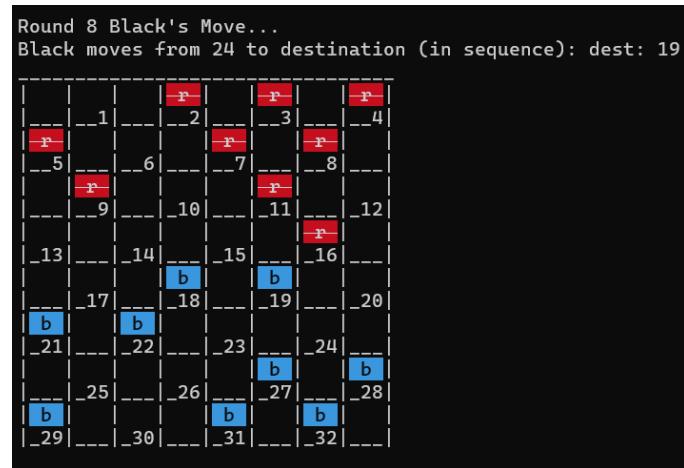


Figure 27: Sample Run 2, Turn 8

```
Round 80 Black's Move...
Black moves from 10 to destination (in sequence): dest: 7
```

	1	2	3	4
b	b	B		
5	6	7	8	
	9	10	11	12
R				
13	14	15	16	
	17	18	19	20
21	22	23	24	
	25	26	27	28
29	30	31	32	

```
RED Leaf Nodes: 24683
RED Expanded Nodes: 37489
RED Total Nodes: 62172
```

```
BLACK Leaf Nodes: 38490
BLACK Expanded Nodes: 51299
BLACK Total Nodes: 89789
```

```
DRAW!!!
Red - ( ० ० ) - Black
Mission FAILED...We'll get em next time!
Red player alg: Alpha-Beta-Search, eval: 1
Black player alg: Alpha-Beta-Search, eval: 3
Depth: 4
Minimax-a-b took: 1837.89ms
ABSearch took: 0ms
```

Figure 28: Sample Run 2 End state

5.3 Sample Run #3: Minimax-a-b with EF-3 vs Alpha-Beta-Search with EF-3 at Depth 6 (extra)

We now get into higher depths. At depth 6, we see a 10x explosion once again in the number of nodes generated though the game still ends in a draw after a back-and-forth stalemate for several turns. As we are using EF-3, which tends to attack Red's "double corner" on the top left, we see a cluster of black pieces which traveled to those squares, which are valued higher; this is in line with our expectations.

```
Round 80 Black's Move...
Black moves from 7 to destination (in sequence): dest: 2
+---+---+---+---+---+---+
|   | 1 | 2 | 3 | 4 |
+---+---+---+---+---+
| b | b |   |   |   |
+---+---+---+---+---+
|   | 6 |   | 7 | 8 |
+---+---+---+---+---+
| b |   | b |   |   |
+---+---+---+---+---+
| 9 | 10 | 11 | 12 |   |
+---+---+---+---+---+
| 13 | 14 | 15 | 16 |   |
+---+---+---+---+---+
| 17 | 18 | 19 | 20 |   |
+---+---+---+---+---+
| 21 | 22 | 23 | 24 |   |
+---+---+---+---+---+
| b |   | r |   |   |
+---+---+---+---+---+
| 25 | 26 | 27 | 28 |   |
+---+---+---+---+---+
| 29 | 30 | 31 | 32 |   |
+---+---+---+---+---+
RED Leaf Nodes: 170008
RED Expanded Nodes: 234729
RED Total Nodes: 404737

BLACK Leaf Nodes: 165402
BLACK Expanded Nodes: 280880
BLACK Total Nodes: 446282

DRAW!!!
Red - (я •_•)я ы( 'д' ы) - Black
Mission FAILED...We'll get em next time!
Red player alg: Minimax-Alpha-Beta, eval: 3
Black player alg: Alpha-Beta-Search, eval: 3
Depth: 6
Minimax-a-b took: 4290.13ms
ABSearch took: 4380.31ms
```

Figure 29: Depth 6 - MAB EF-3 vs ABS EF-3, Draw

5.4 Sample Run #4: ALPHA-BETA-SEARCH with EF-1 vs ALPHA-BETA-SEARCH with EF-3 at Depth 8 (extra)

In this sample run, we go to depth 8; in the beginning when the AI player is “thinking” each turn takes longer than at lower depths. The game significantly speeds up towards the end.

Round 80 Black's Move...
Black moves from 9 to destination (in sequence): dest: 6

	1	2	3	4
b	B			
5	6	7	8	
9	10	11	12	
13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
29	30	31	32	

RED Leaf Nodes: 685938
RED Expanded Nodes: 1005163
RED Total Nodes: 1691101

BLACK Leaf Nodes: 480809
BLACK Expanded Nodes: 814838
BLACK Total Nodes: 1295647

DRAW!!!
Red - () - Black
Mission FAILED...We'll get em next time!
Red player alg: Alpha-Beta-Search, eval: 1
Black player alg: Alpha-Beta-Search, eval: 3
Depth: 8
Minimax-a-b took: 0ms
ABSearch took: 40453ms

Figure 30: Sample Run 4 Depth 8!

5.5 Sample Run #5: Human Player vs AI (Minmax-A-B with EF-3)

I tested playing against the AI at depth 4 and opted to select a simple strategy of always selecting the first move presented. I ended up winning by boxing in the Red player after a few turns, which was surprisingly easy. I had at one-point optimized EF-3 to win or draw against this style of play, but it led to losses against EF 2 and opted to ignore the “take first move” strategy.”

BLACK WINS!!!
BLACK Player: <(-_-')>
But most importantly, RED looooses (booooo!)
RED Player: (J°□°) J ~ 

	B				
	1	2	3	4	
	5	6	7	8	
	9	10	11	12	
	13	14	15	16	
		b			
	17	18	19	20	
		b			
	21	22	23	24	
			r		
	25	26	27	28	
		b	b	b	
	29	30	31	32	

For more information, contact the Office of the Vice President for Research and Economic Development at 319-273-2500 or research@uiowa.edu.

5.6 Sample Run #6: Printing the Help Menu

The system also offers a help menu should the user invoke the program with the following command: ./Project2 -h

This is a relatively brief help message, giving the user guidance on how to interact with the program and where to refer to accompanying documentation for more guidance.

```
[bss64@eros:~/CS5436_AI/project-two]$ ./CheckersAI -h
To use this program, please read the instructions below and re-launch.
Additional details for building and execution are also available in the README file.

Run with -nc for No Color, with -no for No Debug Output, or with -ncno for both No Color AND No Debug Output.
When executing the program, you will be prompted to enter the algorithm and evaluation function for the simulation.
Please follow the instructions on the screen - if you do not care for any specific custom configuration, simply select Run All.
This will bypass everything and simply run the simulation for all algorithms and all evaluation functions sequentially.
This is the most common and preferred option.
```

Figure 32: Sample Run #6 - Print the Help Menu

6 Conclusion

This was a highly successful project. We correctly implemented all algorithms and learned a great deal from implementing the evaluation functions, which are the heart of the AI engine. We saw that for the same evaluation function and depth, both algorithms are identical – they reach the same end-game state via the same game path while generating the same number of nodes. Performance wise, alpha-beta-search appears to fare slightly better at higher depths, while at low depths the two are comparable. At depths two and four minimax-a-b is slightly faster, though the difference is not significant. Seeing how the paths are generated and different states of the board evaluated, it now makes sense why a game like checkers may be mathematically “solved” – we can choose an optimal path in relation to the opponent’s moves. Checkers is currently considered a “solved” game in that an optimal solution exists. If the opponent makes no mistakes, the game should always end in a Draw. Our evaluation functions, when pitted either against each other or themselves, lead mostly to a Draw as well, albeit for slightly different reasons. While the evaluation functions are detailed and intelligent, towards the end-game state we encounter the horizon-effect, especially if the opponent is farther away than our depth limit. We observe a back-and-forth jitter of a few pieces, which quickly exhausts the permitted number of turns, leading to a draw. Several possibilities for resolving this were explored – including but not limited to: introducing the concept of memory for each piece, so they may behave differently when needed; remembering the last several moves of the game to detect a back-and-forth repetition and break out of it; or simply giving a random score to force the piece to move and explore elsewhere.

We observed that the program is very fast at depths two and four, tolerable at depths six and eight, and quite time consuming at higher depths due to the size of the game tree and the vast number of nodes being explored. The team strived to not only meet the project requirements but exceed them in several ways. Designing a fun and pleasant user interface, analyzing the program at depths higher than four, allowing the user to play a manual game against a friend or challenge themselves against one of the AI evaluation functions are all extra items our team incorporated to stretch ourselves and make the program fun and usable. We provide options for enabling or disabling the color and verbose output at will. By default, the application runs with verbose output enabled as it is optimized to be used as a tool for learning and for evaluation from a computer science perspective.

References

1. Gaddis, Tony. "Starting out with C++ From Control Structures through Objects, Ninth Edition." Chapter 10 (c-strings & the string class), Chapter 17.3 the Vector Class.
2. Schaeffer, Jonathan & Björnsson, Yngvi & Kishimoto, Akihiro & Müller, Martin & Lake, Robert & Lu, Paul & Sutphen, Steve. (2007). Checkers Is Solved. *Science*. 317. 1518-1522. 10.1126/science.1144079.
3. C++03 Standard [2.1.1.2]. <https://gcc.gnu.org/legacy-ml/gcc/2001-07/msg01120.html>
4. Resources, PPT slides, and course materials on Canvas.
5. Knuth and Moore, 1975. An Analysis of Alpha-Beta Priming.

Appendix A: Source Code

Pieces.hpp

```
1. #ifndef PIECES_H
2. #define PIECES_H
3.
4. #include <string>
5.
6. /**
7. * The Color enumerator is used in place of values 1 and -1
8. * in order to make the code easier to read. Note the values
9. * of 1 and -1. This allows for alternating play by multiplying the current
10.* by 1.
11.
12.*/
13.enum class Color
14.{
15.    RED = 1,
16.    BLACK = -1
17.};
18.
19./**
20.* The Pieces class represents all pieces for a particular player. Each object
21.* of the class contains a single instance of pieces. In addition to this,
22.* ANSI code strings were included here in order to make the output
23.* of the code easier to read. These codes serve no functional purpose
24.* aside from changing text color.
25.*/
26.class Pieces
27.{
28.public:
29.    Pieces();
30.    Pieces(Color color);
```

```

31.
32. // ANSI codes for colored text, to improve UI and readability
33. static std::string ANSI_BLUE_START;
34. static std::string ANSI_RED_START;
35. static std::string ANSI_RED_HIGH;
36. static std::string ANSI_END;
37. static std::string ANSI_GREEN_START;
38. static std::string ANSI_BLUE_COUT;
39. static std::string ANSI_RED_COUT;
40. static std::string ANSI_GREEN_COUT;
41. static std::string ANSI_YELLOW_COUT;
42.
43. // Debug reporting level 3 == display all debug/status lines 2== important, 1 == basic, 0 == none
44. static int ouputDebugData;
45.
46. bool isKing(int position) const;
47. void setKing(int position, bool toKing);
48.
49. // This contains 64 bits to represent the entire piece set and king status for one side.
50. long long pieces;
51.};
52.
53.#endif // !PIECES_H
54.

```

Pieces.cpp

```

1. #include "Pieces.hpp"
2.
3. // ANSI codes for colored text, to improve UI and readability
4. std::string Pieces::ANSII_BLUE_START = "\033[0;30;46m";
5. std::string Pieces::ANSII_RED_START = "\033[0;31m";
6. std::string Pieces::ANSII_RED_HIGH = "\033[9;37;41m";
7. std::string Pieces::ANSII_END = "\033[0m";

```

```

8. std::string Pieces::ANSII_GREEN_START = "\033[0;32m";
9. std::string Pieces::ANSII_BLUE_COUT = "\033[0;30;46m";
10. std::string Pieces::ANSII_RED_COUT = "\033[41;1m";
11. std::string Pieces::ANSII_GREEN_COUT = "\033[0;30;42m";
12. std::string Pieces::ANSII_YELLOW_COUT = "\033[30;48;5;3m";
13.
14. int Pieces::outputDebugData = 3;
15.
16. /**
17. * Constructor | Pieces | Pieces
18. *
19. * Summary : Unused. Constructor with
20. *           a parameter is always required.
21. *
22. * @author : David Torrente
23. *
24. */
25. Pieces::Pieces()
26. {
27. }
28.
29. /**
30. * Constructor | Pieces | Pieces
31. *
32. * Summary : Sets up the player pieces depending on
33. *           the color passed in.
34. *
35. * @author : David Torrente
36. *
37. * @param Color color : The color to assign the pieces to.
38. *
39. */
40. Pieces::Pieces(Color color)
41. {
42.
43.     // The following are bit fields to be used for the board. Each value

```

```

44. // turns on or off a bit. For example, 4095 looks like:
45. // 1111 1111 1111, which will position a piece in squares 1 - 12
46. // top squares. The middle is blank, so 0000 0000, which covers
47. // squares 13 - 20. Lastly,
48. // black is placed in the lowest bits similar to white's pattern,
49. // but in spaces 21 - 32, hence, the large number. Neither start with
50. // kings, so bits 33 - 64 are all zeros on both sides.
51.
52. // Note that the commented out values are primarily for debugging
53. // special move cases or interesting situations. They are retained here as
54. // pairs. If used, each pair must be uncommented.
55. if (color == Color::RED) // red
56.     //pieces = 1;
57.     //pieces = 19455;
58.     //pieces = 1152921504875282432;
59.     pieces = 4095; // Initial board state
60. else // black
61.     //pieces = 4291952640;
62.     //pieces = 128;
63.     //pieces = 16974848;
64.     pieces = 4293918720; // Initial board state
65. }
66.
67. /**
68. * Member Function | Pieces | isKing
69. *
70. * Summary : Determines if the piece in a given position is
71. *           a king or not. Prior to calling this, the position
72. *           must be checked to see if the position is occupied
73. *           by a player.
74. *
75. * @author : David Torrente
76. *
77. * @param int position : The position on the board to
78. *           check to see if it is a king.
79. *

```

```

80. * @return bool          : Returns true if it is a king piece,
81. *                           false otherwise.
82. *
83. */
84.bool Pieces::isKing(int position) const
85.{
86.    // NOTE: it is 31, since the offset is 0 - 31
87.    // Could have written +32 -1, but better to just combine.
88.    // Checks the high bit (33 - 64) which is aligned with
89.    // the position bit (1 - 32).
90.    int kingBit = position + 31;
91.
92.    bool isKing = false;
93.
94.    if (((pieces >> kingBit) & 1) == 1)
95.    {
96.        isKing = true;
97.    }
98.
99.    return isKing;
100.   }
101.
102. /**
103. * Member Function | Pieces | setKing
104. *
105. * Summary :      Sets the king bit for a given position to either
106. *                 be on or off.
107. *
108. * @author : David Torrente
109. *
110. * @param int position : The position on the board to
111. *                       toggle to a king or not king.
112. *
113. * @param bool toKing   : True if the piece is to become
114. *                       a king, false if it is to be reduced
115. *                       to not a king. Reducing the piece type

```

```
116.     *                     is primarily done to keep the board clean.
117.     *
118.     */
119. void Pieces::setKing(int position, bool toKing)
120. {
121.     int kingBit = position + 31;
122.
123.     if (toKing == true)
124.     {
125.         pieces = pieces | (1ULL << kingBit);
126.     }
127.     else
128.     {
129.         pieces = pieces & ~(1ULL << kingBit);
130.     }
131. }
132.
```

Board.hpp

```
1. #ifndef BOARD_H
2. #define BOARD_H
3.
4. #include <vector>
5. #include <string>
6.
7. #include "Pieces.hpp"
8.
9. /**
10. * Class board is used to represent the entire board and its current state.
11. * It is based on an 8 x 8 grid, with 32 possible spaces. Two piece data
12. * types are the primary memory consumers of this class. This class also includes
13. * a static member that acts as a move guide. This move guide determines the possible
14. * moves for a square on the board, not for a piece.
15. */
16.class Board
17.{
18.
19.public:
20. /**
21. * Struct Move is used to track moves made on the board. It contains
22. * a single starting location, a vector of steps to a move,
23. * as well as all of the pieces to remove when this move is made.
24. */
25. struct Move
26. {
27.     int startSquare;
28.     std::vector<int> destinationSquare;
29.     std::vector<int> removalSquare;
30. };
31.
32. /**
33. * Struct BoardMoveTable is a static data type that is used to speed
```

```

34.     * up searching for moves for each position. It is shared among all
35.     * boards.
36.     */
37.     struct BoardMoveTable
38.     {
39.         std::vector<int> jumps;
40.         std::vector<int> removals;
41.         std::vector<int> moves;
42.     };
43.
44.     Board();
45.     ~Board();
46.     static void InitializeMoveTable();
47.     std::vector<Move> moveGen(Color color);
48.
49.     void printBoard() const;
50.     Board updateBoard(Move move, Color color);
51.
52.     int getNumRegularPieces(Color color);
53.     int getNumKingPieces(Color color);
54.     int getNumPlayerTotalPieces(Color color);
55.     int getPieceInSquare(int square, Color color);
56.
57.     std::vector<Move> getJumpsForPiece(Color color, int square, Pieces *playerPieces, Pieces *opponentPieces);
58.     std::vector<Move> getMovesForPiece(Color color, int square, Pieces *playerPieces, Pieces *opponentPieces);
59.
60.     Pieces getPlayerPieces(Color color);
61.     Pieces getOpponentPieces(Color color);
62.
63.     static BoardMoveTable boardMoveTable[33];
64.
65. private:
66.     Pieces blackPieces;
67.     Pieces redPieces;
68.

```

```
69. void getJumpsForPieceRec(Color color, Board::Move move, std::vector<Board::Move> &totalMoves, Board board, bool wa
  sKingPriorMove);
70.};
71.
72.#endif
```

Board.cpp

```
1. #include "Board.hpp"
2. #include "Pieces.hpp"
3. #include <iostream>
4. #include <iomanip>
5.
6. // Forward declare the static data member.
7. Board::BoardMoveTable Board::boardMoveTable[33];
8.
9. /**
10.  * Constructor | Board | Board
11. *
12. * Summary : Creates both the red player and black player
13. *           piece list. If not already done, it also
14. *           instantiates the static move table.
15. *
16. * @author : David Torrente
17. *
18. */
19. Board::Board()
20.{
21.   // Guarded internally to prevent from having moves added to it.
22.   InitializeMoveTable();
23.
24.   // Assign the proper pieces to the player, either red or black.
```

```
25.     redPieces = Pieces(Color::RED);
26.     blackPieces = Pieces(Color::BLACK);
27. }
28.
29. /**
30. * Destructor | Board | ~Board
31. *
32. * Summary : Class destructor. No special code required.
33. *
34. * @author : David Torrente
35. *
36. */
37. Board::~Board()
38. {
39. }
40.
41. /**
42. * Member Function | Board | getPlayerPieces
43. *
44. * Summary : Gets the pieces that belong to a particular player.
45. *
46. * @author : David Torrente
47. *
48. * @param Color color : The player color of pieces to get.
49. *
50. * @return Pieces : The pieces belonging to the player.
51. *
52. */
53. Pieces Board::getPlayerPieces(Color color)
54. {
55.     if (color == Color::RED)
56.     {
57.         return redPieces;
58.     }
59.     else
60.     {
```

```
61.         return blackPieces;
62.     }
63. }
64.
65. /**
66. * Member Function | Board | getOpponentPieces
67. *
68. * Summary : Gets the pieces that belong to the
69. *             opposing player.
70. *
71. * @author : David Torrente
72. *
73. * @param Color color : The player color of pieces to
74. *             get the opponent of.
75. *
76. * @return Pieces : The pieces belonging to the
77. *             opposing player.
78. *
79. */
80.Pieces Board::getOpponentPieces(Color color)
81.{  
82.    if (color == Color::RED)
83.    {
84.        return blackPieces;
85.    }
86.    else
87.    {
88.        return redPieces;
89.    }
90.}
91.
92. /**
93. * Member Function | Board | getNumRegularPieces
94. *
95. * Summary : Gets the count of man pieces that belong
96. *             to the player.
```

```
97. *
98. * @author : David Torrente
99.
100.    * @param Color color : The player color of pieces to
101.        * get the count for.
102.    *
103.    * @return int : A value 0 to 12 which is the count
104.        * of man pieces.
105.    *
106.    */
107. int Board::getNumRegularPieces(Color color)
108. {
109.     returngetNumPlayerTotalPieces(color) - getNumKingPieces(color);
110. }
111.
112. /**
113. * Member Function | Board | getNumKingPieces
114. *
115. * Summary : Gets the count of king pieces that belong
116. *             to the player.
117. *
118. * @author : David Torrente
119. *
120. * @param Color color : The player color of pieces to
121. *             get the count for.
122. *
123. * @return int : A value 0 to 12 which is the count
124. *             of king pieces.
125. *
126. */
127. int Board::getNumKingPieces(Color color)
128. {
129.     {
130.         int kingPieceCount = 0;
131.
132.         long long *playerPieces;
```

```

133.
134.        if (color == Color::RED)
135.        {
136.            playerPieces = &redPieces.pieces;
137.        }
138.        else
139.        {
140.            playerPieces = &blackPieces.pieces;
141.        }
142.
143.        for (int iter = 32; iter < 64; iter++)
144.        {
145.            if (((*playerPieces >> iter) & 1) == 1)
146.            {
147.                kingPieceCount++;
148.            }
149.        }
150.
151.        return kingPieceCount;
152.    }
153.}
154.
155. /**
156. * Member Function | Board | getNumPlayerTotalPieces
157. *
158. * Summary : Gets the count of total pieces that belong
159. *           to the player.
160. *
161. * @author : David Torrente
162. *
163. * @param Color color : The player color of pieces to
164. *           get the count for.
165. *
166. * @return int : A value 0 to 12 which is the count
167. *           of total pieces.
168. *

```

```

169.     */
170.     int Board::getNumPlayerTotalPieces(Color color)
171.     {
172.         int totalPieceCount = 0;
173.
174.         long long *playerPieces;
175.
176.         if (color == Color::RED)
177.         {
178.             playerPieces = &redPieces.pieces;
179.         }
180.         else
181.         {
182.             playerPieces = &blackPieces.pieces;
183.         }
184.
185.         for (int iter = 0; iter < 32; iter++)
186.         {
187.
188.             if (((*playerPieces >> iter) & 1) == 1)
189.             {
190.                 totalPieceCount++;
191.             }
192.         }
193.
194.         return totalPieceCount;
195.     }
196.
197. /**
198. * Member Function | Board | moveGen
199. *
200. * Summary : Gets all of the possible moves for the player
201. *           based on the color parameter. Note that this
202. *           is bound by the mandatory jump rule, meaning
203. *           that a player who can jump will not be given
204. *           the option to move to a adjacent space. These

```

```

205.     * adjacent space moves will not even be computed
206.     * if there is a jump possible.
207.     *
208.     * @author : David Torrente
209.     *
210.     * @param Color color : The player color to get the moves
211.     * for.
212.     *
213.     * @return vector<Board::Move> : A vector of possible moves
214.     * for the player.
215.     *
216.     */
217. std::vector<Board::Move> Board::moveGen(Color color)
218. {
219.     std::vector<Move> totalMoves;
220.     std::vector<Move> returnedMoves;
221.
222.     Pieces *playerPieces;
223.     Pieces *opponentPieces;
224.
225.     int bitOffset = 0;
226.
227.     if (color == Color::RED)
228.     {
229.         playerPieces = &redPieces;
230.         opponentPieces = &blackPieces;
231.     }
232.
233.     else
234.     {
235.         playerPieces = &blackPieces;
236.         opponentPieces = &redPieces;
237.     }
238.
239.     // Go through all 32 squares and see if it is one
240.     // of the appropriate pieces belonging to player.

```

```

241.     for (int pieceIter = 1; pieceIter <= 32; pieceIter++)
242.     {
243.         // The offset is used to align to 0 - 31, but
244.         // the board in checkers is 1 - 32. Use an offset
245.         // here to align to the position bits properly.
246.         bitOffset = pieceIter - 1;
247.
248.         if (((playerPieces->pieces) >> bitOffset) & 1) == 1
249.         {
250.             // Check for possible jump moves first.
251.             returnedMoves = getJumpsForPiece(color, pieceIter, playerPieces, opponentPieces);
252.             totalMoves.insert(totalMoves.end(), returnedMoves.begin(), returnedMoves.end());
253.         }
254.     }
255.
256.     // Go through all 32 squares and see if it is one
257.     // of the appropriate pieces belonging to player.
258.     // Do this only if no jumps are possible. This is controlled
259.     // by the if condition here. No need to get moves if there are
260.     // jumps already found.
261.     if (totalMoves.size() == 0)
262.     {
263.         for (int pieceIter = 1; pieceIter <= 32; pieceIter++)
264.         {
265.             // The offset is used to align to 0 - 31, but
266.             // the board in checkers is 1 - 32. Use an offset
267.             // here to align to the position bits properly.
268.             bitOffset = pieceIter - 1;
269.
270.             // Check if player is in this space
271.             if (((playerPieces->pieces) >> bitOffset) & 1) == 1
272.             {
273.                 // Check for non jump moves here.
274.                 returnedMoves = getMovesForPiece(color, pieceIter, playerPieces, opponentPieces);
275.                 totalMoves.insert(totalMoves.end(), returnedMoves.begin(), returnedMoves.end());
276.             }

```

```

277.         }
278.     }
279.     return totalMoves;
280. }
281.
282. /**
283. * Member Function | Board | getJumpsForPiece
284. *
285. * Summary : Gets all of the possible jumps for the specific
286. * piece based on the color parameter and location.
287. *
288. * This is only called after confirming that the player
289. * has a piece in the proper location.
290. *
291. * @author : David Torrente
292. *
293. * @param Color color : The player color of pieces to
294. * get the jumps for.
295. *
296. * @param int piece : The board location to get the jumps
297. * for.
298. *
299. * @param Pieces *playerPieces : A pointer to the player pieces.
300. * Needed since these can block a possible
301. * jump.
302. *
303. * @param Pieces *opponentPieces : A pointer to the opponent pieces.
304. * Needed since these can block a possible
305. * jump as well as to confirm that a jump
306. * can happen.
307. *
308. * @return vector<Board::Move> : A vector of possible jumps
309. * for the particular piece.
310. */
311. std::vector<Board::Move> Board::getJumpsForPiece(Color color, int piece, Pieces *playerPieces, Pieces *opponentP
ieces)

```

```

312.    {
313.        // The returned vector of all possible jumps carried out completely
314.        std::vector<Move> finalMoves;
315.
316.        Board board;
317.
318.        bool wasKingPriorMove = false;
319.
320.        if (color == Color::RED)
321.        {
322.            board.redPieces = *playerPieces;
323.            board.blackPieces = *opponentPieces;
324.        }
325.        else
326.        {
327.            board.redPieces = *opponentPieces;
328.            board.blackPieces = *playerPieces;
329.        }
330.        // In order to start the recursion, this is needed.
331.        Move move;
332.        move.startSquare = piece;
333.        wasKingPriorMove = board.getPlayerPieces(color).isKing(move.startSquare);
334.
335.        getJumpsForPieceRec(color, move, finalMoves, board, wasKingPriorMove);
336.
337.        return finalMoves;
338.    }
339.
340.    /**
341.     * Member Function | Board | getJumpsForPieceRec
342.     *
343.     * Summary : Gets all of the possible jumps in a single jump attempt.
344.     * Often called "double" jumping. This creates a full
345.     * jump chain. a recursive call.
346.     *
347.     * @author : David Torrente

```

```

348. *
349. * @param Color color : The player color of pieces to
350. * get the jump chain for.
351. *
352. * @param Move move : A starter move. Contains a starting
353. * space, with an empty destination vector.
354. * when passed in to a recursive call,
355. * this destination vector may contain
356. * moves, signifying that it is in the middle
357. * of a jump chain.
358. *
359. * @param vector<Board> & totalMovesAccumulator : Accumulates all possible
360. * jumps at the end of each jump chain. Passed
361. * in by reference and returned to calling
362. * function.
363. *
364. * @param Board board : The current state of the board, updated for each jump.
365. *
366. * @param bool wasKingPriorMove : A value used to determine if the king was a king prior
367. * to this move. Becoming a king stops a jump chain.
368. *
369. */
370. void Board::getJumpsForPieceRec(Color color, Board::Move move, std::vector<Board::Move> &totalMovesAccumulator,
  Board board, bool wasKingPriorMove)
371. {
372.
373.     int piece;
374.
375.     bool endOfJumpChain = true;
376.
377.     bool isKing = false;
378.
379.     int bitOffset = 0;
380.     int squareJumped = 0;
381.
382.     // This determines if we are starting the jump sequence or in the middle of it.

```

```

383.     if (move.destinationSquare.size() == 0)
384.     {
385.         piece = move.startSquare;
386.         // use the current board as it is.
387.     }
388.     else
389.     {
390.         piece = move.destinationSquare.back();
391.         board = updateBoard(move, color);
392.     }
393.
394.     // First, determine if it is a king. This is needed to see which moves
395.     // are valid for this piece/player. Move either up/down at first.
396.     isKing = board.getPlayerPieces(color).isKing(piece);
397.
398.     if (isKing == wasKingPriorMove)
399.     {
400.         // Check if a jump position is open for this piece. This goes through all of the jumps.
401.         for (int jumpIter = 0; jumpIter < Board::boardMoveTable[piece].jumps.size(); jumpIter++)
402.         {
403.             // Get the position of the jump, reduce it by one for an offset. Note
404.             // that while it is one less than the position, the direction check still works
405.             // since a jump will always be greater than the distance needed to determine
406.             // direction.
407.             bitOffset = (Board::boardMoveTable[piece].jumps.at(jumpIter) - 1);
408.
409.             // Check to see which direction it is going, and if it can go that direction.
410.             // King goes both ways      Red not a king goes "down"          Black not a king goes "up"
411.             if (isKing || ((piece - bitOffset) < 0 && color == Color::RED) || ((piece - bitOffset) > 0 && color
412. == Color::BLACK))
413.             {
414.                 // Combine both bit fields into one and check if the space is empty.
415.                 if (((board.getPlayerPieces(color).pieces | board.getOpponentPieces(color).pieces) >> bitOffset
416. ) & 1) == 0
417.                 {
418.                     // If it is open, get the space between. We will need to check it.

```

```

417.         // Again with the -1 to help with the offset
418.         squareJumped = Board::boardMoveTable[piece].removals.at(jumpIter) - 1;
419.
420.         if (((board.getOpponentPieces(color).pieces >> squareJumped) & 1) == 1)
421.         {
422.             // Yup, we are good!!!! this means that an opponent was in this spot and
423.             // we can jump them.
424.
425.             // Since we found a new jump, we need to keep going with the jump chain
426.             endOfJumpChain = false;
427.
428.             // Do not change start square. This is set to the overall start of the jump.
429.             // Commented out to show the thought process behind setting the move.
430.             // move.startSquare = piece;
431.
432.             move.destinationSquare.push_back(bitOffset + 1);
433.             move.removalSquare.push_back(squareJumped + 1);
434.
435.             //make recursive call here
436.             getJumpsForPieceRec(color, move, totalMovesAccumulator, board, wasKingPriorMove);
437.             // Remove the jump we just added to the chain. We've already made the recursive call
438.             move.destinationSquare.pop_back();
439.             move.removalSquare.pop_back();
440.         }
441.     }
442. }
443. }
444. }
445.
446. if (endOfJumpChain == true)
447. {
448.     // Once here, it means we are at the leaf of a jump or
449.     // that we became a king due to this jump.
450.     // We add it to the list at that point.
451.     // It will rise back up the chain when the stack unwinds.
452.

```

```

453.         if (move.destinationSquare.size() != 0)
454.         {
455.             totalMovesAccumulator.push_back(move);
456.         }
457.     }
458. }
459.
460. /**
461. * Member Function | Board | getMovesForPiece
462. *
463. * Summary : Gets all of the possible moves for a piece. This call knows that the
464. * piece is a proper piece for this color to play on prior to this call.
465. * Also note that prior to this call, jumps should be checked. If there
466. * are jumps, do not allow these moves.
467. *
468. * @author : David Torrente
469. *
470. * @param Color color : The player color of pieces to
471. * get the moves for.
472. *
473. * @param int piece : The board location to get the moves
474. * for.
475. *
476. * @param Pieces *playerPieces : A pointer to the player pieces.
477. * Needed since these can block a possible
478. * jump.
479. *
480. * @param Pieces *opponentPieces : A pointer to the opponent pieces.
481. * Needed since these can block a possible
482. * jump as well as to confirm that a jump
483. * can happen.
484. *
485. * @return vector<Board::Move> :
486. * A vector of possible moves
487. * for the particular piece.
488. */

```

```

489.     std::vector<Board::Move> Board::getMovesForPiece(Color color, int piece, Pieces *playerPieces, Pieces *opponentP
  ieces)
490.     {
491.         std::vector<Move> moves;
492.         Move move;
493.
494.         bool isKing = false;
495.
496.         int squareJumped = 0;
497.
498.         int bitOffset = 0;
499.
500.         // First, determine if it is a king. This is needed to see which moves
501.         // are valid for this piece/player. Move either up/down at first.
502.         if (playerPieces->isKing(piece))
503.         {
504.             isKing = true;
505.         }
506.         else
507.         {
508.             isKing = false;
509.         }
510.
511.         for (int moveIter = 0; moveIter < Board::boardMoveTable[piece].moves.size(); moveIter++)
512.         {
513.             bitOffset = Board::boardMoveTable[piece].moves.at(moveIter) - 1;
514.             if (isKing || ((piece - bitOffset) < 0 && color == Color::RED) || ((piece - bitOffset) > 0 && color == C
  olor::BLACK))
515.             {
516.                 if (((playerPieces->pieces | opponentPieces->pieces) >> bitOffset) & 1) == 0)
517.                 {
518.                     move.startSquare = piece;
519.                     move.destinationSquare.push_back(bitOffset + 1);
520.                     moves.push_back(move);
521.                     move.destinationSquare.clear();
522.                 }

```

```
523.         }
524.     }
525.
526.     return moves;
527. }
528.
529. /**
530. * Member Function | Board | printBoard
531. *
532. * Summary : Prints the current state of the board. It is a constant function.
533. *
534. * @author : David Torrente
535. *
536. */
537. void Board::printBoard() const
538.
539.     int squareOffset = 0;
540.
541.     // The board toggles back and forth in regards to the positioning
542.     // of pieces. This either adds the offset to the beginning of the
543.     // row, or to the end of it. Odd rows start with a space,
544.     // Even rows end with a space.
545.     bool oddRow = true;
546.
547.     // A basic bar to go across the top. Cosmetic.
548.     std::cout << " _____" << std::endl;
549.
550.     for (int rowIter = 0; rowIter <= 7; rowIter++)
551.     {
552.         std::cout << "|";
553.
554.         for (int colIter = 0; colIter <= 3; colIter++)
555.         {
556.             if (oddRow)
557.             {
558.                 std::cout << " |";
```

```
559.     }
560.
561.     squareOffset = (rowIter * 4 + colIter);
562.
563.     if (((redPieces.pieces >> squareOffset) & 1) == 1)
564.     {
565.         if (redPieces.isKing(squareOffset + 1))
566.         {
567.             std::cout << Pieces::ANSII_RED_HIGH << " R " << Pieces::ANSII_END << "|";
568.         }
569.         else
570.         {
571.             std::cout << Pieces::ANSII_RED_HIGH << " r " << Pieces::ANSII_END << "|";
572.         }
573.     }
574.     else if (((blackPieces.pieces >> squareOffset) & 1) == 1)
575.     {
576.         if (blackPieces.isKing(squareOffset + 1))
577.         {
578.             std::cout << Pieces::ANSII_BLUE_START << " B " << Pieces::ANSII_END << "|";
579.         }
580.         else
581.         {
582.             std::cout << Pieces::ANSII_BLUE_START << " b " << Pieces::ANSII_END << "|";
583.         }
584.     }
585.     else
586.     {
587.         std::cout << "   |";
588.     }
589.
590.     if (!oddRow)
591.     {
592.         std::cout << "   |";
593.     }
594. }
```

```

595.
596.        // Now print the numeric values of the squares.
597.        std::cout << std::endl;
598.
599.        std::cout << "|";
600.
601.        for (int colIterNum = 0; colIterNum <= 3; colIterNum++)
602.        {
603.            if (oddRow)
604.            {
605.                std::cout << "___|";
606.            }
607.
608.            squareOffset = (rowIter * 4 + colIterNum);
609.            std::cout << std::setfill('_') << std::setw(3) << squareOffset + 1 << "|";
610.
611.            if (!oddRow)
612.            {
613.                std::cout << "___|";
614.            }
615.        }
616.
617.        // Toggle between even and odd rows for the offset.
618.        oddRow = !oddRow;
619.        std::cout << std::endl;
620.    }
621.}
622.
623./**
624. * Member Function | Board | getPieceInSquare
625. *
626. * Summary : Gets the type of piece at a particular location for a particular color.
627. * Note that the position does not need to be checked for a piece prior to this
628. * call. If the position does not contain a piece of the matching color type,
629. * it will report as empty for that color.
630. *

```

```

631.     * @author : David Torrente
632.     *
633.     * @param int piece    :           The board location to get the piece type
634.     *                               for.
635.     *
636.     * @param Color color :           The player color to get the piece type for.
637.     *
638.     * @return int   :           a value, 0 to 2, specifying the piece type of the given
639.     *                               color. 0 = no piece for this color. 1 = man piece
640.     *                               for this color. 2 = king piece for this color.
641.     *
642.     */
643. int Board::getPieceInSquare(int position, Color color)
644. {
645.     int pieceType = 0;
646.     Pieces playerPieces;
647.
648.     if (Color::RED == color)
649.     {
650.         playerPieces = redPieces;
651.     }
652.     else
653.     {
654.         playerPieces = blackPieces;
655.     }
656.
657.     if (((playerPieces.pieces >> (position - 1)) & 1) == 1)
658.     {
659.         pieceType = 1;
660.
661.         if (playerPieces.isKing(position))
662.         {
663.             pieceType = 2;
664.         }
665.     }
666.

```

```

667.         return pieceType;
668.     }
669.
670.    /**
671.     * Member Function | Board | updateBoard
672.     *
673.     * Summary :      Updates the entire board based on a given move. Note that this updates the complete
674.     *               board for both sides. This will always operate on a single properly formatted move.
675.     *               It will also convert pieces into kings if they reach the back row of the opposing
676.     *               player.
677.     *
678.     * @author : David Torrente
679.     *
680.     * @param Move move      :          The move to apply to the board. Includes the destination
681.     *                                     and pieces to remove.
682.     *
683.     * @param Color color   :          The player color that is applying this move. Needed primarily
684.     *                                     to determine if a piece needs to be kinged.
685.     *
686.     * @return Board      :          Returns a new copy of the board. Note that this is only
687.     *                                     two bitfields (small).
688.     *
689.     */
690. Board Board::updateBoard(Move move, Color color)
691. {
692.
693.     Board updatedBoard;
694.     updatedBoard.blackPieces = blackPieces;
695.     updatedBoard.redPieces = redPieces;
696.
697.     Pieces *playerPieces;
698.     Pieces *opponentPieces;
699.
700.     if (color == Color::RED)
701.     {
702.         playerPieces = &updatedBoard.redPieces;

```

```

703.         opponentPieces = &updatedBoard.blackPieces;
704.     }
705.     else
706.     {
707.         playerPieces = &updatedBoard.blackPieces;
708.         opponentPieces = &updatedBoard.redPieces;
709.     }
710.
711.     // Position in final destination spot - probably check if it needs to be kinged here.
712.     // Do this first since you will need to clear the king flag.
713.     playerPieces->pieces = playerPieces->pieces | (1LL << (move.destinationSquare.back() - 1));
714.     if (playerPieces->isKing(move.startSquare))
715.     {
716.         playerPieces->setKing(move.startSquare, false);
717.         playerPieces->setKing(move.destinationSquare.back(), true);
718.     }
719.     else
720.     {
721.         // Check to see if we've landed in the kinging row, the back row opposite the starting side.
722.         if (color == Color::RED && (move.destinationSquare.back() >= 29) || (color == Color::BLACK && (move.destinationSquare.back() <= 4)))
723.         {
724.             playerPieces->setKing(move.destinationSquare.back(), true);
725.         }
726.     }
727.
728.     // Remove start pos
729.     if (move.startSquare != move.destinationSquare.back())
730.     {
731.         playerPieces->pieces = playerPieces->pieces & ~(1LL << (move.startSquare - 1));
732.     }
733.
734.     // Remove all jumped spots and set them back to not a king.
735.     for (int jumpedSpaceIter = 0; jumpedSpaceIter < move.removalSquare.size(); jumpedSpaceIter++)
736.     {

```

```

737.         opponentPieces->pieces = opponentPieces-
    >pieces & ~(1LL << (move.removalSquare.at(jumpedSpaceIter) - 1));
738.         opponentPieces->pieces = opponentPieces-
    >pieces & ~(1LL << (move.removalSquare.at(jumpedSpaceIter) + 31));
739.     }
740.
741.     return updatedBoard;
742. }
743.
744. /**
745. * Member Function | Board | InitializeMoveTable
746. *
747. * Summary : Sets up the static move table which is shared among all
748. * boards. Note that it is a move table, not a piece tracker.
749. * pieces are stored in the board. In order to prevent adding
750. * moves each time a constructor is called, the move table
751. * will not add any additional values after it is set up
752. * the first time.
753. *
754. * @author : David Torrente
755. *
756. */
757. void Board::InitializeMoveTable()
758. {
759.
    // In order to support faster look up,
    // The index of the position is used as
    // the initial key to an index. Each
    // position has only a few moves,
    // even less if there is a mandatory jump.
    // The table is big, but never changes.
766.
767. if (boardMoveTable[1].jumps.size() == 0)
768. {
769.     boardMoveTable[1].jumps.push_back(10);
770.     boardMoveTable[1].removals.push_back(6);

```

```
771.     boardMoveTable[1].moves.push_back(5);
772.     boardMoveTable[1].moves.push_back(6);
773.
774.     boardMoveTable[2].jumps.push_back(9);
775.     boardMoveTable[2].removals.push_back(6);
776.     boardMoveTable[2].jumps.push_back(11);
777.     boardMoveTable[2].removals.push_back(7);
778.     boardMoveTable[2].moves.push_back(6);
779.     boardMoveTable[2].moves.push_back(7);
780.
781.     boardMoveTable[3].jumps.push_back(10);
782.     boardMoveTable[3].removals.push_back(7);
783.     boardMoveTable[3].jumps.push_back(12);
784.     boardMoveTable[3].removals.push_back(8);
785.     boardMoveTable[3].moves.push_back(7);
786.     boardMoveTable[3].moves.push_back(8);
787.
788.     boardMoveTable[4].jumps.push_back(11);
789.     boardMoveTable[4].removals.push_back(8);
790.     boardMoveTable[4].moves.push_back(8);
791.
792.     boardMoveTable[5].jumps.push_back(14);
793.     boardMoveTable[5].removals.push_back(9);
794.     boardMoveTable[5].moves.push_back(1);
795.     boardMoveTable[5].moves.push_back(9);
796.
797.     boardMoveTable[6].jumps.push_back(13);
798.     boardMoveTable[6].removals.push_back(9);
799.     boardMoveTable[6].jumps.push_back(15);
800.     boardMoveTable[6].removals.push_back(10);
801.     boardMoveTable[6].moves.push_back(1);
802.     boardMoveTable[6].moves.push_back(2);
803.     boardMoveTable[6].moves.push_back(9);
804.     boardMoveTable[6].moves.push_back(10);
805.
806.     boardMoveTable[7].jumps.push_back(14);
```

```
807.     boardMoveTable[7].removals.push_back(10);
808.     boardMoveTable[7].jumps.push_back(16);
809.     boardMoveTable[7].removals.push_back(11);
810.     boardMoveTable[7].moves.push_back(2);
811.     boardMoveTable[7].moves.push_back(3);
812.     boardMoveTable[7].moves.push_back(10);
813.     boardMoveTable[7].moves.push_back(11);
814.
815.     boardMoveTable[8].jumps.push_back(15);
816.     boardMoveTable[8].removals.push_back(11);
817.     boardMoveTable[8].moves.push_back(3);
818.     boardMoveTable[8].moves.push_back(4);
819.     boardMoveTable[8].moves.push_back(11);
820.     boardMoveTable[8].moves.push_back(12);
821.
822.     boardMoveTable[9].jumps.push_back(2);
823.     boardMoveTable[9].removals.push_back(6);
824.     boardMoveTable[9].jumps.push_back(18);
825.     boardMoveTable[9].removals.push_back(14);
826.     boardMoveTable[9].moves.push_back(5);
827.     boardMoveTable[9].moves.push_back(6);
828.     boardMoveTable[9].moves.push_back(13);
829.     boardMoveTable[9].moves.push_back(14);
830.
831.     boardMoveTable[10].jumps.push_back(1);
832.     boardMoveTable[10].removals.push_back(6);
833.     boardMoveTable[10].jumps.push_back(3);
834.     boardMoveTable[10].removals.push_back(7);
835.     boardMoveTable[10].jumps.push_back(17);
836.     boardMoveTable[10].removals.push_back(14);
837.     boardMoveTable[10].jumps.push_back(19);
838.     boardMoveTable[10].removals.push_back(15);
839.     boardMoveTable[10].moves.push_back(6);
840.     boardMoveTable[10].moves.push_back(7);
841.     boardMoveTable[10].moves.push_back(14);
842.     boardMoveTable[10].moves.push_back(15);
```

```
843.  
844.     boardMoveTable[11].jumps.push_back(2);  
845.     boardMoveTable[11].removals.push_back(7);  
846.     boardMoveTable[11].jumps.push_back(4);  
847.     boardMoveTable[11].removals.push_back(8);  
848.     boardMoveTable[11].jumps.push_back(18);  
849.     boardMoveTable[11].removals.push_back(15);  
850.     boardMoveTable[11].jumps.push_back(20);  
851.     boardMoveTable[11].removals.push_back(16);  
852.     boardMoveTable[11].moves.push_back(7);  
853.     boardMoveTable[11].moves.push_back(8);  
854.     boardMoveTable[11].moves.push_back(15);  
855.     boardMoveTable[11].moves.push_back(16);  
856.  
857.     boardMoveTable[12].jumps.push_back(3);  
858.     boardMoveTable[12].removals.push_back(8);  
859.     boardMoveTable[12].jumps.push_back(19);  
860.     boardMoveTable[12].removals.push_back(16);  
861.     boardMoveTable[12].moves.push_back(8);  
862.     boardMoveTable[12].moves.push_back(16);  
863.  
864.     boardMoveTable[13].jumps.push_back(6);  
865.     boardMoveTable[13].removals.push_back(9);  
866.     boardMoveTable[13].jumps.push_back(22);  
867.     boardMoveTable[13].removals.push_back(17);  
868.     boardMoveTable[13].moves.push_back(9);  
869.     boardMoveTable[13].moves.push_back(17);  
870.  
871.     boardMoveTable[14].jumps.push_back(5);  
872.     boardMoveTable[14].removals.push_back(9);  
873.     boardMoveTable[14].jumps.push_back(7);  
874.     boardMoveTable[14].removals.push_back(10);  
875.     boardMoveTable[14].jumps.push_back(21);  
876.     boardMoveTable[14].removals.push_back(17);  
877.     boardMoveTable[14].jumps.push_back(23);  
878.     boardMoveTable[14].removals.push_back(18);
```

```
879.     boardMoveTable[14].moves.push_back(9);
880.     boardMoveTable[14].moves.push_back(10);
881.     boardMoveTable[14].moves.push_back(17);
882.     boardMoveTable[14].moves.push_back(18);
883.
884.     boardMoveTable[15].jumps.push_back(6);
885.     boardMoveTable[15].removals.push_back(10);
886.     boardMoveTable[15].jumps.push_back(8);
887.     boardMoveTable[15].removals.push_back(11);
888.     boardMoveTable[15].jumps.push_back(22);
889.     boardMoveTable[15].removals.push_back(18);
890.     boardMoveTable[15].jumps.push_back(24);
891.     boardMoveTable[15].removals.push_back(19);
892.     boardMoveTable[15].moves.push_back(10);
893.     boardMoveTable[15].moves.push_back(11);
894.     boardMoveTable[15].moves.push_back(18);
895.     boardMoveTable[15].moves.push_back(19);
896.
897.     boardMoveTable[16].jumps.push_back(7);
898.     boardMoveTable[16].removals.push_back(11);
899.     boardMoveTable[16].jumps.push_back(23);
900.     boardMoveTable[16].removals.push_back(19);
901.     boardMoveTable[16].moves.push_back(11);
902.     boardMoveTable[16].moves.push_back(12);
903.     boardMoveTable[16].moves.push_back(19);
904.     boardMoveTable[16].moves.push_back(20);
905.
906.     boardMoveTable[17].jumps.push_back(10);
907.     boardMoveTable[17].removals.push_back(14);
908.     boardMoveTable[17].jumps.push_back(26);
909.     boardMoveTable[17].removals.push_back(22);
910.     boardMoveTable[17].moves.push_back(13);
911.     boardMoveTable[17].moves.push_back(14);
912.     boardMoveTable[17].moves.push_back(21);
913.     boardMoveTable[17].moves.push_back(22);
914.
```

```
915.     boardMoveTable[18].jumps.push_back(9);
916.     boardMoveTable[18].removals.push_back(14);
917.     boardMoveTable[18].jumps.push_back(11);
918.     boardMoveTable[18].removals.push_back(15);
919.     boardMoveTable[18].jumps.push_back(25);
920.     boardMoveTable[18].removals.push_back(22);
921.     boardMoveTable[18].jumps.push_back(27);
922.     boardMoveTable[18].removals.push_back(23);
923.     boardMoveTable[18].moves.push_back(14);
924.     boardMoveTable[18].moves.push_back(15);
925.     boardMoveTable[18].moves.push_back(22);
926.     boardMoveTable[18].moves.push_back(23);
927.
928.     boardMoveTable[19].jumps.push_back(10);
929.     boardMoveTable[19].removals.push_back(15);
930.     boardMoveTable[19].jumps.push_back(12);
931.     boardMoveTable[19].removals.push_back(16);
932.     boardMoveTable[19].jumps.push_back(26);
933.     boardMoveTable[19].removals.push_back(23);
934.     boardMoveTable[19].jumps.push_back(28);
935.     boardMoveTable[19].removals.push_back(24);
936.     boardMoveTable[19].moves.push_back(15);
937.     boardMoveTable[19].moves.push_back(16);
938.     boardMoveTable[19].moves.push_back(23);
939.     boardMoveTable[19].moves.push_back(24);
940.
941.     boardMoveTable[20].jumps.push_back(11);
942.     boardMoveTable[20].removals.push_back(16);
943.     boardMoveTable[20].jumps.push_back(27);
944.     boardMoveTable[20].removals.push_back(24);
945.     boardMoveTable[20].moves.push_back(16);
946.     boardMoveTable[20].moves.push_back(24);
947.
948.     boardMoveTable[21].jumps.push_back(14);
949.     boardMoveTable[21].removals.push_back(17);
950.     boardMoveTable[21].jumps.push_back(30);
```

```
951.     boardMoveTable[21].removals.push_back(25);
952.     boardMoveTable[21].moves.push_back(17);
953.     boardMoveTable[21].moves.push_back(25);
954.
955.     boardMoveTable[22].jumps.push_back(13);
956.     boardMoveTable[22].removals.push_back(17);
957.     boardMoveTable[22].jumps.push_back(15);
958.     boardMoveTable[22].removals.push_back(18);
959.     boardMoveTable[22].jumps.push_back(29);
960.     boardMoveTable[22].removals.push_back(25);
961.     boardMoveTable[22].jumps.push_back(31);
962.     boardMoveTable[22].removals.push_back(26);
963.     boardMoveTable[22].moves.push_back(17);
964.     boardMoveTable[22].moves.push_back(18);
965.     boardMoveTable[22].moves.push_back(25);
966.     boardMoveTable[22].moves.push_back(26);
967.
968.     boardMoveTable[23].jumps.push_back(14);
969.     boardMoveTable[23].removals.push_back(18);
970.     boardMoveTable[23].jumps.push_back(16);
971.     boardMoveTable[23].removals.push_back(19);
972.     boardMoveTable[23].jumps.push_back(30);
973.     boardMoveTable[23].removals.push_back(26);
974.     boardMoveTable[23].jumps.push_back(32);
975.     boardMoveTable[23].removals.push_back(27);
976.     boardMoveTable[23].moves.push_back(18);
977.     boardMoveTable[23].moves.push_back(19);
978.     boardMoveTable[23].moves.push_back(26);
979.     boardMoveTable[23].moves.push_back(27);
980.
981.     boardMoveTable[24].jumps.push_back(15);
982.     boardMoveTable[24].removals.push_back(19);
983.     boardMoveTable[24].jumps.push_back(31);
984.     boardMoveTable[24].removals.push_back(27);
985.     boardMoveTable[24].moves.push_back(19);
986.     boardMoveTable[24].moves.push_back(20);
```

```
987.     boardMoveTable[24].moves.push_back(27);
988.     boardMoveTable[24].moves.push_back(28);
989.
990.     boardMoveTable[25].jumps.push_back(18);
991.     boardMoveTable[25].removals.push_back(22);
992.     boardMoveTable[25].moves.push_back(21);
993.     boardMoveTable[25].moves.push_back(22);
994.     boardMoveTable[25].moves.push_back(29);
995.     boardMoveTable[25].moves.push_back(30);
996.
997.     boardMoveTable[26].jumps.push_back(17);
998.     boardMoveTable[26].removals.push_back(22);
999.     boardMoveTable[26].jumps.push_back(19);
1000.    boardMoveTable[26].removals.push_back(23);
1001.    boardMoveTable[26].moves.push_back(22);
1002.    boardMoveTable[26].moves.push_back(23);
1003.    boardMoveTable[26].moves.push_back(30);
1004.    boardMoveTable[26].moves.push_back(31);
1005.
1006.    boardMoveTable[27].jumps.push_back(18);
1007.    boardMoveTable[27].removals.push_back(23);
1008.    boardMoveTable[27].jumps.push_back(20);
1009.    boardMoveTable[27].removals.push_back(24);
1010.    boardMoveTable[27].moves.push_back(23);
1011.    boardMoveTable[27].moves.push_back(24);
1012.    boardMoveTable[27].moves.push_back(31);
1013.    boardMoveTable[27].moves.push_back(32);
1014.
1015.    boardMoveTable[28].jumps.push_back(19);
1016.    boardMoveTable[28].removals.push_back(24);
1017.    boardMoveTable[28].moves.push_back(24);
1018.    boardMoveTable[28].moves.push_back(32);
1019.
1020.    boardMoveTable[29].jumps.push_back(22);
1021.    boardMoveTable[29].removals.push_back(25);
1022.    boardMoveTable[29].moves.push_back(25);
```

```

1023.
1024.        boardMoveTable[30].jumps.push_back(21);
1025.        boardMoveTable[30].removals.push_back(25);
1026.        boardMoveTable[30].jumps.push_back(23);
1027.        boardMoveTable[30].removals.push_back(26);
1028.        boardMoveTable[30].moves.push_back(25);
1029.        boardMoveTable[30].moves.push_back(26);
1030.
1031.        boardMoveTable[31].jumps.push_back(22);
1032.        boardMoveTable[31].removals.push_back(26);
1033.        boardMoveTable[31].jumps.push_back(24);
1034.        boardMoveTable[31].removals.push_back(27);
1035.        boardMoveTable[31].moves.push_back(26);
1036.        boardMoveTable[31].moves.push_back(27);
1037.
1038.        boardMoveTable[32].jumps.push_back(23);
1039.        boardMoveTable[32].removals.push_back(27);
1040.        boardMoveTable[32].moves.push_back(27);
1041.        boardMoveTable[32].moves.push_back(28);
1042.    }
1043.}

```

Player.hpp

```

1. #ifndef PLAYER_H
2. #define PLAYER_H
3.
4. #include "Pieces.hpp"
5. #include "Board.hpp"
6.
7. /**
8.  * Header definition for class Player.
9. *
10. * The Player class defines a virtual player, in the case of this program an AI.

```

```
11. * No human players are used in this game - meaning no manual input is sought from the user.
12. * Project 2, Requirement #7: "Your program should play with a computer to a computer."
13. *
14. */
15.
16.class Player
17.{
18.
19.private:
20.    Color color;          // represents player's color in the game, either RED or BLACK
21.    int numPieces;        // how many pieces does Player have left
22.    int numPiecesTaken;  // Player's current score based on captured enemy pieces
23.    int numTurnsTaken;   // counter for Player's turns taken
24.    bool didPlayerMove; // EndGame Condition
25.
26.    int depth, evalVersion;
27.
28.public:
29.    Player(); // constructor
30.    ~Player(); // destructor
31.
32.    bool isMinimax; // if false use AB Prune, if true use Minimax. Allows control over alg player uses
33.
34.    Player(bool minMax, Color color, int depth, int evalVersion); // overloaded constructor to set player color, which
   is IMMUTABLE
35.
36.    int takeTurn(Board &state);
37.    int getNumPieces();
38.    int getNumPiecesTaken();
39.    int getNumTurns();
40.    bool getDidPlayerMove();
41.    Color getColor();
42.
43.    void decreaseNumPieces(int numPiecesToDecreaseCount);
44.    void increaseNumPiecesTaken(int numPiecesToIncreaseScore);
45.
```

```
46. static void printMove(Board::Move, Color color, bool willPrintAlways);
47.
48. int minimaxExpandedNodes; // how many nodes we expand
49. int minimaxLeafNodes; // how many nodes we expand
50. int absearchExpandedNodes; // how many nodes we expand
51. int absearchLeafNodes; // how many nodes we expand
52.
53. int getMinimaxTotalNodes()
54. {
55.     return minimaxExpandedNodes + minimaxLeafNodes;
56. }
57.
58. int getAbSearchTotalNodes()
59. {
60.     return absearchExpandedNodes + absearchLeafNodes;
61. }
62.};
63.
64.#endif // !PLAYER_H
```

Player.cpp

```
1. #include "Player.hpp"
2. #include "Algorithm.hpp"
3.
4. #include <iostream>
5.
6. Player::Player()
7. {
8. }
9. Player::~Player()
10.{
```

```

11.}
12.
13. Player::Player(bool minMaxState, Color color, int depth, int evalVersion)
14.{
15.    this->color = color;
16.    numPieces = 12;      // how many pieces does Player have left
17.    numPiecesTaken = 0; // Player's current score based on captured enemy pieces
18.    numTurnsTaken = 0; // counter for Player's turns taken
19.    isMinimax = minMaxState;
20.    this->depth = depth;
21.    this->evalVersion = evalVersion;
22.
23.    this->minimaxExpandedNodes = 0;
24.    this->minimaxLeafNodes = 0;
25.    this->absearchExpandedNodes = 0;
26.    this->absearchLeafNodes = 0;
27.}
28.
29. int Player::takeTurn(Board &state)
30.{
31.    Algorithm::Result result;
32.    Algorithm *algorithm = new Algorithm(evalVersion, depth, *this);
33.
34.    if (isMinimax)
35.    {
36.        result = algorithm->minimax_a_b(state, this->depth, this->color, 9000000, -8000000);
37.        this->minimaxExpandedNodes += algorithm->minimaxExpandedNodes;
38.        this->minimaxLeafNodes += algorithm->minimaxLeafNodes;
39.    }
40.    else
41.    {
42.        result = algorithm->alphaBetaSearch(state);
43.        this->absearchExpandedNodes += algorithm->absearchExpandedNodes;
44.        this->absearchLeafNodes += algorithm->absearchLeafNodes;
45.    }
46.

```

```

47.     if (result.bestMove.destinationSquare.size() == 0)
48.     {
49.         didPlayerMove = false; // Player did not make a turn
50.     }
51.     else
52.     {
53.         state = state.updateBoard(result.bestMove, this->color);
54.         printMove(result.bestMove, this->color, true);
55.         numTurnsTaken++; // increment Player's own turn counter
56.         didPlayerMove = true; // return true as player did make a turn
57.         state.printBoard();
58.     }
59.
60.     // return how many pieces the player took during their turn
61.     return result.bestMove.removalSquare.size();
62. }
63.
64. int Player::getNumPieces()
65. {
66.     return numPieces;
67. }
68.
69. int Player::getNumPiecesTaken()
70. {
71.     return numPiecesTaken;
72. }
73.
74. int Player::getNumTurns()
75. {
76.     return numTurnsTaken;
77. }
78.
79. bool Player::getDidPlayerMove()
80. {
81.     return didPlayerMove;
82. }

```

```
83.
84. Color Player::getColor()
85. {
86.     return color;
87. }
88.
89. void Player::decreaseNumPieces(int numPiecesToDecreaseCount)
90. {
91.     numPieces -= numPiecesToDecreaseCount;
92. }
93.
94. void Player::increaseNumPiecesTaken(int numPiecesToIncreaseScore)
95. {
96.     numPiecesTaken += numPiecesToIncreaseScore;
97. }
98.
99. void Player::printMove(Board::Move move, Color color, bool alwaysPrint)
100.    {
101.        std::string colorStr;
102.        if (color == Color::BLACK)
103.            colorStr = "Black";
104.        else
105.            colorStr = "Red";
106.
107.        int conditionalPrint;
108.        if (alwaysPrint)
109.            conditionalPrint = 1;
110.        else
111.            conditionalPrint = Pieces::outputDebugData;
112.
113.        if (conditionalPrint)
114.        {
115.            std::cout << colorStr << " moves from " << move.startSquare << " to destination (in sequence): ";
116.            for (int i = 0; i < move.destinationSquare.size(); i++)
117.                std::cout << "dest: " << move.destinationSquare.at(i) << std::endl;
118.        }
```

```
119.      }
120.
```

Game.hpp

```
1. #ifndef GAME_H
2. #define GAME_H
3.
4. #include "Player.hpp"
5. #include "Board.hpp"
6.
7. /**
8.  * Header definition for class Game.
9. *
10. * The Game class represents a checkers Game.
11. *
12. * A game consists of a board, two players (red and black), and
13. * 12 checkers pieces for each player.
14. *
15. * A game ends when either player loses all their pieces or is blocked and
16. * has no further moves available.
17. *
18. * The startGame() function will trigger the process and no input from the user is required.
19. * Red and Black players will be created. Each player will be given 12 checker pieces.
20. * The pieces will be appropriately placed on the board.
21. * Each player will execute their strategy.
22. *
23.
24. */
25.
26.class Game
27.{
```

```
28.
29.private:
```

```

30.     Board state;
31.     Player redPlayer;
32.     Player blackPlayer;
33.     const int MAX_ALLOWED_TURNS = 80;
34.     void printNodes(Player player, std::string colorText);
35.
36. public:
37.     Game(); // constructor
38.     ~Game(); // destructor
39.     Game(bool, int, bool, int, int); // player1 algo, eval version, player2 algo, eval version, depth
40.
41.     enum class GameOver
42.     {
43.         BLACK_WINS = 0,
44.         RED_WINS = 1,
45.         DRAW = 2,
46.         NOT_DONE = 3
47.     };
48.
49.     // The only initialization function needed, as the game will
50.     // be played automatically by 2 AI players (MIN and MAX).
51.     // while gameOver == NOT_DONE keep playing
52.     GameOver startGame();
53.
54.     // simple function to invert the enum value, thus determine who's turn is it next.
55.     // E.g., if currentPlayer is RED (1), function returns BLACK (-1)
56.     Color changePlayer(Color currentPlayer);
57.
58.     GameOver gameOver(); // Have end game conditions been met?
59.     bool doesRedWin();
60.     bool doesBlackWin();
61.     bool isItADraw();
62. };
63.
64. #endif // !GAME_H
65.

```

Game.cpp

```
1. #include "Game.hpp"
2.
3. #include <iostream>
4.
5. Game::Game()
6. {
7. }
8.
9. Game::~Game()
10.{}
11.}
12.
13. Game::Game(bool player1MinMax, int evalVersionP1, bool player2MinMax, int evalVersionP2, int depth)
14.{
15.    state = Board();
16.    state.InitializeMoveTable();
17.    redPlayer = Player(player1MinMax, Color::RED, depth, evalVersionP1);
18.    blackPlayer = Player(player2MinMax, Color::BLACK, depth, evalVersionP2);
19.}
20.
21. Game::GameOver Game::startGame()
22.{
23.    int piecesTaken;
24.
25.    while (true)
26.    {
27.        std::cout << "\n\nRound " << blackPlayer.getNumTurns() + 1 << " Black's Move..." << std::endl;
28.    }
}
```

```

29.     piecesTaken = blackPlayer.takeTurn(state);
30.     blackPlayer.increaseNumPiecesTaken(piecesTaken);
31.     redPlayer.decreaseNumPieces(piecesTaken);
32.
33.     if (piecesTaken > 0)
34.         std::cout << "  BLACK player took " << piecesTaken << " piece(s)." << std::endl;
35.
36.     if (doesBlackWin())
37.     {
38.         printNodes(redPlayer, "RED");
39.         printNodes(blackPlayer, "BLACK");
40.         return GameOver::BLACK_WINS;
41.     }
42.
43.     if (doesRedWin())
44.     {
45.         printNodes(redPlayer, "RED");
46.         printNodes(blackPlayer, "BLACK");
47.         return GameOver::RED_WINS;
48.     }
49.
50.     if (isItADraw())
51.     {
52.         printNodes(redPlayer, "RED");
53.         printNodes(blackPlayer, "BLACK");
54.         return GameOver::DRAW;
55.     }
56.
57.     std::cout << "\n\nRound " << redPlayer.getNumTurns() + 1 << " Red's Move..." << std::endl;
58.
59.     piecesTaken = redPlayer.takeTurn(state);
60.     redPlayer.increaseNumPiecesTaken(piecesTaken);
61.     blackPlayer.decreaseNumPieces(piecesTaken);
62.
63.     if (piecesTaken > 0)
64.         std::cout << "  RED player took " << piecesTaken << " piece(s)." << std::endl;

```

```

65.
66.    if (doesBlackWin())
67.    {
68.        printNodes(redPlayer, "RED");
69.        printNodes(blackPlayer, "BLACK");
70.        return GameOver::BLACK_WINS;
71.    }
72.
73.    if (doesRedWin())
74.    {
75.        printNodes(redPlayer, "RED");
76.        printNodes(blackPlayer, "BLACK");
77.        return GameOver::RED_WINS;
78.    }
79.
80.    if (isItADraw())
81.    {
82.        printNodes(redPlayer, "RED");
83.        printNodes(blackPlayer, "BLACK");
84.        return GameOver::DRAW;
85.    }
86. }
87. }
88.
89. void Game::printNodes(Player player, std::string colorText)
90. {
91.     std::cout << std::endl;
92.     if (player.isMinimax)
93.     {
94.         std::cout << colorText << " Leaf Nodes: " << player.minimaxLeafNodes << std::endl;
95.         std::cout << colorText << " Expanded Nodes: " << player.minimaxExpandedNodes << std::endl;
96.         std::cout << colorText << " Total Nodes: " << player.getMinimaxTotalNodes() << std::endl;
97.     }
98.     else
99.     {
100.         std::cout << colorText << " Leaf Nodes: " << player.absearchLeafNodes << std::endl;

```

```
101.         std::cout << colorText << " Expanded Nodes: " << player.absearchExpandedNodes << std::endl;
102.         std::cout << colorText << " Total Nodes: " << player.getAbSearchTotalNodes() << std::endl;
103.     }
104.     std::cout << std::endl;
105. }
106.
107. Color Game::changePlayer(Color currentPlayer)
108. {
109.     if (currentPlayer == Color::BLACK)
110.         return Color::RED;
111.     else
112.         return Color::BLACK;
113. }
114.
115. bool Game::doesBlackWin()
116. {
117.     std::vector<Board::Move> redMoves = state.moveGen(Color::RED);
118.     return (redMoves.size() == 0);
119. }
120.
121. bool Game::doesRedWin()
122. {
123.     std::vector<Board::Move> blackMoves = state.moveGen(Color::BLACK);
124.     return (blackMoves.size() == 0);
125. }
126.
127. bool Game::isItADraw()
128. {
129.     if (redPlayer.getNumTurns() >= MAX_ALLOWED_TURNS ||
130.         blackPlayer.getNumTurns() >= MAX_ALLOWED_TURNS)
131.         return true;
132.     return false;
133. }
```

Algorithm.hpp

```
1. #ifndef ALGORITHM_H
2. #define ALGORITHM_H
3.
4. #include <vector>
5. #include <string>
6.
7. #include "Player.hpp"
8. #include "Board.hpp"
9.
10. /**
11.  * Header definition for class Algorithm.
12.  * @author multiple - David, Boris, and Randy
13.  *
14.  * This class will encapsulate the algorithmic approach the AI uses to play Checkers.
15. *
```

```

16. * There are only two major algorithms supported.
17. *   1) Minimax - a depth first, depth limited search procedure. From the Richard and Knight book.
18. *       The minimax function has a heuristic value for leaf nodes (end nodes and nodes at the maximum permitted depth
   ).
19. *       Intermediate nodes get their value from a child/successor leaf node.
20. *   2) Alpha-
   Beta Pruning - a search algorithm that decreases the number of nodes evaluated by minimax in it's search tree.
21. *       We stop evaluating a possible move when at least one option is found to be worse than a previously examined m
   ove.
22. *       NOTE: It should return the SAME result as minimax, just "prunes" branches that will not affect the final outc
   ome,
23. *       thus improving performance.
24. *
25. * Three evaluation functions will be used in conjunction with the two search algorithms.
26. *
27. */
28.
29.class Algorithm
30.{
31.
32.public:
33.    Algorithm(); // constructor
34.    ~Algorithm(); // destructor
35.
36.    Algorithm(int evalVersion, int maxDepth, Player callingPlayer);
37.
38.    struct Result
39.    {
40.        int value;
41.        Board::Move bestMove;
42.    };
43.
44.    int minimaxExpandedNodes; // how many nodes we expand
45.    int minimaxLeafNodes; // how many nodes we expand
46.    int absearchExpandedNodes; // how many nodes we expand
47.    int absearchLeafNodes; // how many nodes we expand

```

```

48.
49. // minimax algorithm returns the position of the best move
50. Result minimax_a_b(Board board, int depth, Color color, int useThresh, int passThresh);
51.
52. // AB Prune algorithm
53. Result alphaBetaSearch(Board state);
54.
55. void setEvalVersion(int evalVersion);
56. void setMaxDepth(int maxDepth);
57.
58.private:
59.     int numNodesGenerated;
60.     int evalVersion;
61.     int currentDepth, maxDepth;
62.     Player callingPlayer;
63.
64. // plausible move generator, returns a list of positions that can be made by player
65. std::vector<Board::Move> movegen(Board board, Color color);
66.
67. /* static evaluation functions return a number representing the
68. * goodness of Position from the standpoint of Player
69. * A helper function staticEval is used to determine which evalFunction to use
70. */
71. int evalFunctOne(Board state, Color color);
72. int evalFunctTwo(Board state, Color color);
73. int evalFunctThree(Board state, Color color);
74.
75. // wrapper function that will decide which of the actual three eval functions to call
76. int staticEval(Board state, Color color, int evalVersion);
77.
78. // if true, return the structure
79. bool deepEnough(int currentDepth);
80.
81. bool terminalTest(Board state, int depth); // terminal test for alpha-beta-search
82. Result maxValue(Board state, int depth, int alpha, int beta, Color color);
83. Result minValue(Board state, int depth, int alpha, int beta, Color color);

```

```
84.     int utility(Board state);
85.     std::vector<Board::Move> actions(Board state, Color color);
86.
87.     Color switchPlayerColor(Color color);
88.
89.     int passSign(int passthresh);
90. };
91.
92.#endif // !ALGORITHM_H
```

Algorithm.cpp

```
1. #include "Algorithm.hpp"
2.
3. #include <limits>
4. #include <stdexcept>
5. #include <iostream>
6.
7. Algorithm::Algorithm()
8. {
9.     this->minimaxLeafNodes = 0;
10.    this->minimaxExpandedNodes = 0;
11.    this->absearchLeafNodes = 0;
12.    this->absearchExpandedNodes = 0;
13. }
14.
15.Algorithm::~Algorithm()
16.{}
17.}
18.
19./*
20. * Overloaded constructor for Algorithm to set internal member variables
21. */
```

```

22. Algorithm::Algorithm(int evalVersion, int maxDepth, Player callingPlayer)
23. {
24.     this->evalVersion = evalVersion;
25.     this->maxDepth = maxDepth;
26.     this->callingPlayer = callingPlayer;
27.     this->minimaxLeafNodes = 0;
28.     this->minimaxExpandedNodes = 0;
29.     this->absearchLeafNodes = 0;
30.     this->absearchExpandedNodes = 0;
31. }
32.
33. /**
34. * movegen function gets a list of all possible moves for a player as a vector of moves.
35. * This function is essentially a wrapper, which calls the movegen function in class Board.
36. * @author Borislav Sabotinov
37. *
38. * @param Board board
39. * @param Player board
40. *
41. * @return vector<Board::Move> listofPossibleMoves
42. */
43. std::vector<Board::Move> Algorithm::movegen(Board board, Color color)
44. {
45.     return board.moveGen(color);
46. }
47.
48. /**
49. * Member Function | Algorithm | evalFunctionOne
50. *
51. * Summary : Evaluates the current board and produces a score for it based on
52. *           the current player of this ply. Note that it toggles back and forth
53. *           from maximizing to minimizing.
54. *
55. * @author : David Torrente
56. *
57. * @param State state : The state of the board to be used in evaluation.

```

```

58. *
59. * @param Color color      :           The player color that is considered the current player.
60. *
61. * @return int              :           Returns an evaluated score for the board.
62. *
63. */
64.int Algorithm::evalFuncOne(Board state, Color color)
65.{
66.    int gameTurn = callingPlayer.getNumTurns();
67.    int finalScore = 0;
68.    bool criticalPoint = false;
69.    Color opponentColor;
70.    int backRowDefense = 0;
71.    int opponentBackRowDefense = 0;
72.    int centerPiece;
73.    int centerControl = 0;
74.    // These are the weights for the opening series of turns.
75.    // They later adjust to better fit end game play.
76.    int manWeight = 40;
77.    int kingWeight = 90;
78.    int mobilityWeight = 5;
79.    int backRowDefenseWeight = 5;
80.    int centerControlWeight = 6;
81.    int playerScore = 0;
82.    int playerBonus = 0;
83.    int opponentManWeight = 40;
84.    int opponentKingWeight = 90;
85.    int opponentMobilityWeight = 5;
86.    int opponentBackRowDefenseWeight = 2;
87.    int opponentScore = 0;
88.    int opponentBonus = 0;
89.    std::vector<Board::Move> playerMoves;
90.    std::vector<Board::Move> opponentMoves;
91.    //=====
92.    // Set up a few items based on the player color. Opponent and player
93.    // defense row.

```

```

94. //=====
95. if (Color::RED == color)
96. {
97.     opponentColor = Color::BLACK;
98.     // Calculate back row defense
99.     for (int redIter = 1; redIter <= 4; redIter++)
100.    {
101.        if (state.getPieceInSquare(redIter, color) == 1)
102.        {
103.            backRowDefense++;
104.        }
105.        if (state.getPieceInSquare(redIter + 28, opponentColor) == 1)
106.        {
107.            opponentBackRowDefense++;
108.        }
109.    }
110. }
111. else
112. {
113.     opponentColor = Color::RED;
114.     for (int blackIter = 29; blackIter <= 32; blackIter++)
115.     {
116.         if (state.getPieceInSquare(blackIter, color) == 1)
117.         {
118.             backRowDefense++;
119.         }
120.         if (state.getPieceInSquare(blackIter - 28, opponentColor) == 1)
121.         {
122.             opponentBackRowDefense++;
123.         }
124.     }
125. }
126. int manCount = state.getNumRegularPieces(color);
127. int kingCount = state.getNumKingPieces(color);
128. int opponentManCount = state.getNumRegularPieces(opponentColor);
129. int opponentKingCount = state.getNumKingPieces(opponentColor);

```

```

130. //=====
131. // Check for the two obvious ones, a winning state or a losing state.
132. // Used to simply exit out of the eval function without calculating
133. // a bunch of extra values.
134. //=====
135. playerMoves = state.moveGen(color);
136. int mobility = playerMoves.size();
137. // This is a losing state, avoid it.
138. // Note that vs. other evaluation functions, this may not happen.
139. // This means that while it sees a losing state, the other eval
140. // may take a different path and void out these results.
141. if (!criticalPoint && mobility == 0)
142. {
143.     opponentBonus = opponentBonus + 6000000;
144.     criticalPoint = true;
145. }
146. opponentMoves = state.moveGen(opponentColor);
147. int opponentMobility = opponentMoves.size();
148. // This is a winning state, try to get it.
149. // Note that vs. other evaluation functions, this may not happen.
150. // This means that while it sees a losing state, the other eval
151. // may take a different path and void out these results.
152. int totalJumped = 0;
153. int totalToJump = opponentKingCount + opponentManCount;
154. for (int moveIter = 0; !criticalPoint && moveIter < playerMoves.size(); moveIter++)
155. {
156.     totalJumped = playerMoves.at(moveIter).destinationSquare.size();
157.     if ((totalToJump - totalJumped) == 0)
158.     {
159.         playerBonus = playerBonus + 6000000;
160.         criticalPoint = true;
161.     }
162. }
163. if (!criticalPoint)
164. {
165.     //=====

```

```

166.     // Set some values based on the current turn to adjust play styles.
167.     // Games tend to run a little longer, so the turn numbers are set up
168.     // higher than one would expect.
169.     //=====
170.     if (gameTurn >= 40)
171.     {
172.         manWeight = 20;
173.         kingWeight = 80;
174.         mobilityWeight = 3;
175.         backRowDefenseWeight = 2;
176.         centerControlWeight = 2;
177.         opponentManWeight = 20;
178.         opponentKingWeight = 80;
179.         opponentMobilityWeight = 3;
180.         opponentBackRowDefenseWeight = 1;
181.     }
182.     else if (gameTurn >= 60)
183.     {
184.         manWeight = 10;
185.         kingWeight = 30;
186.         mobilityWeight = 3;
187.         backRowDefenseWeight = 0;
188.         centerControlWeight = 1;
189.         opponentManWeight = 40;
190.         opponentKingWeight = 90;
191.         opponentMobilityWeight = 1;
192.         opponentBackRowDefenseWeight = 0;
193.     }
194.     //=====
195.     // Check for pretty bad states, such as a large difference in the number
196.     // of pieces between players. This typically means multiple jumps.
197.     // Intended to avoid fine tuning specific calculations.
198.     //=====
199.     int totalPieceDisparityCount = ((manCount + kingCount) - (opponentManCount + opponentKingCount));
200.     // This means attempt to reduce the opponent
201.     // at almost any price. Even if it means losing

```

```

202.     // a few pieces.
203.     if (totalPieceDisparityCount >= 2)
204.     {
205.         playerBonus = playerBonus + 1000;
206.         opponentManWeight = 600;
207.         opponentKingWeight = 1600;
208.         manWeight = 1;
209.         kingWeight = 1;
210.     }
211.     // This is a larger disparity. Be even more aggressive.
212.     if (totalPieceDisparityCount >= 4)
213.     {
214.         playerBonus = playerBonus + 10000;
215.         opponentManWeight = 6000;
216.         opponentKingWeight = 16000;
217.         manWeight = 1;
218.         kingWeight = 1;
219.     }
220.     // This means that you are trailing in pieces.
221.     // The objective is to now set up a more defensive
222.     // mode of play. Typically it is bad to get here.
223.     else if (totalPieceDisparityCount <= -2)
224.     {
225.         opponentBonus = opponentBonus + 1000;
226.         manWeight = 60;
227.         kingWeight = 160;
228.     }
229.     // A difference in kings can lead to a loss.
230.     // This is a small adjustment to monitor this.
231.     if ((kingCount - opponentKingCount) >= 2)
232.     {
233.         playerBonus = playerBonus + 100;
234.     }
235.     else if (kingCount - opponentKingCount <= -2)
236.     {
237.         opponentBonus = opponentBonus + 100;

```

```

238. }
239. //=====
240. // Check for a specific location bonus.
241. // 14,15,17,18,19,22,23 are the center locations.
242. // Add a bonus if you occupy them.
243. //=====
244. for (int centerIter = 14; centerIter <= 23; centerIter++)
245. {
246.     centerPiece = state.getPieceInSquare(centerIter, color);
247.     // Double the control value if it is a king in that location.
248.     if (centerPiece == 1)
249.     {
250.         centerControl = centerControl + 1;
251.     }
252.     else if (centerPiece == 2)
253.     {
254.         centerControl = centerControl + 2;
255.     }
256.     // Skip the edges
257.     if (centerIter == 15 || centerIter == 19)
258.     {
259.         centerIter++;
260.         if (centerIter == 20)
261.         {
262.             centerIter++;
263.         }
264.     }
265. }
266. }
267. // Tally up the player overall score and then the opponent overall score.
268. playerScore =
269.     (manCount * manWeight) +
270.     (kingCount * kingWeight) +
271.     (mobility * mobilityWeight) +
272.     (backRowDefense * backRowDefenseWeight) +
273.     playerBonus;

```

```

274.     opponentScore =
275.         (opponentManCount * opponentManWeight) +
276.         (opponentKingCount * opponentKingWeight) +
277.         (opponentMobility * opponentMobilityWeight) +
278.         (opponentBackRowDefense * opponentBackRowDefenseWeight) +
279.         opponentBonus;
280.     finalScore = playerScore - opponentScore;
281.     // Clamp the final score based on the current scoring
282.     // system. Note that these values can later be adjusted
283.     // to use a wider range.
284.     if (finalScore <= -8000000)
285.     {
286.         finalScore = -7999999;
287.     }
288.     if (finalScore >= 9000000)
289.     {
290.         finalScore = 7999999;
291.     }
292.     return finalScore;
293. }
294.
295. /**
296. * Second evaluation function
297. * @author Randall Henderson
298. *
299. * @param Board State
300. * @param Color color
301. *
302. * @return an integer score of how good we think the state is
303. */
304. int Algorithm::evalFuncTwo(Board state, Color color)
305. {
306.     const int KING = 2,
307.             MAN = 1;
308.     std::string indentValue; // Tracking header in cout statements. Shows who is player being processed
309.     std::string playerColor; // Used for debugging output

```

```

310.    // these four variables are the number of pieces each player has on the board in this state.
311.    int numPieces = state.getNumRegularPieces(color),                                // how many regular pieces a
     re on the board
312.    numKingPieces = state.getNumKingPieces(color),                                // how many Kings are on the
     board
313.    numOpponentPieces = state.getNumRegularPieces(switchPlayerColor(color)),    // same as above
314.    numOpponentKingsPieces = state.getNumKingPieces(switchPlayerColor(color)), // except opponent values
315.    // Total Pieces on the board
316.    totalPieces = numPieces + numKingPieces + numOpponentPieces + numOpponentKingsPieces,
317.    // This places a value on the player's pieces. This weight can be modified to encourage defensive
318.    // piece preservation strategy
319.    pieceValue = 560,
320.    kingValue = 3000,
321.    // total value of the piece preservation portion of the equation
322.    preservePlayersPieces = numPieces * pieceValue + numKingPieces * kingValue,
323.    // total value of the opponents piece value
324.    preserveOpponentsPieces = numOpponentPieces * pieceValue + numOpponentKingsPieces * kingValue,
325.    // This places a value on taking opponent pieces, if a move removes an opponent piece this results in
326.    // a higher return value for the move
327.    opponentValue = 100,
328.    opponentKingValue = 1000,
329.    opponentPieces = (numOpponentPieces * opponentValue + numOpponentKingsPieces * opponentKingValue),
     // Value of all opponent's pieces
330.    reduceOpponentPieceValue = (opponentPieces - numOpponentKingsPieces * opponentKingValue) / opponentValue
     , // value of each man
331.    reduceOpponentKingValue = opponentPieces - reduceOpponentPieceValue,
     // value of each king
332.    crossOfPainValue = 50,
     //This is the square immediately above, below, to the right or left of the king
333.    // When in end game (less than 8 pieces) this value is used to alter strategy between taking pieces
     // and preserving pieces
334.    endGameAdjust = 1,
335.    playerPiece,
336.    opponentPiece,
337.    positionAdder = 0,
338.    opponentPositionAdder = 0,
339.

```

```

340.         playerScore,
341.         opponentScore,
342.         moveScore,
343.         currentTerminal = 0,
344.         opponentTerminal = 0;
345. // The opponent's board will be scored the same as the current player. This percentage allows for the
346. // adjustment in the impact the player's board has on the final value
347. double opponentEvaluationWeight = .55;
348. // the values Red squares. This encourages the Red men to move towards the opponents base line
349. // the two high values for Red's base line encourages a base defense strategy
350. // startup Board
351. int squareValuesForRed[] = {10, 1, 10, 1,
352.                             8, 5, 5, 2,
353.                             5, 3, 3, 5,
354.                             5, 3, 5, 3,
355.                             5, 8, 5, 8,
356.                             15, 8, 15, 8,
357.                             15, 25, 15, 25,
358.                             75, 75, 75, 75};
359. if (numOpponentPieces == 3) // only Kings left, encourage all men to move to the back row
360. {
361.     for (int i = 0; i < 32; ++i)
362.     {
363.         squareValuesForRed[i] = (i % 4) * 3;
364.     }
365. }
366. int squareValuesForBlack[32]; // this board will be initialized below
367. // black values are the opposite of Red
368. for (int i = 31; i >= 0; --i)
369.     squareValuesForBlack[31 - i] = squareValuesForRed[i];
370. // this array is used to encourage the King to move towards the center of the board
371. int squareValuesForKing[] = {1, 1, 1, 1,
372.                             3, 5, 5, 3,
373.                             7, 9, 9, 7,
374.                             8, 13, 13, 8,
375.                             8, 13, 13, 8,

```

```

376.                     7, 9, 9, 7,
377.                     3, 5, 5, 3,
378.                     1, 1, 1, 1};
379.     if (color == Color::RED)
380.     {
381.         playerColor = Pieces::ANSII_RED_START;
382.         playerColor.append("RED ");
383.         playerColor.append(Pieces::ANSII_END);
384.     }
385.     else
386.         playerColor = "BLACK ";
387.     indentValue = playerColor;
388.     if (Pieces::outputDebugData > 1)
389.         std::cout << indentValue << Pieces::ANSII_GREEN_COUT << " Evaluating Current Player: Pieces-> "
390.                         << Pieces::ANSII_END << numPieces << " Kings-> " << numKingPieces << " Opponent Pieces-> "
391.                         << numOpponentPieces << " Kings-> " << numOpponentKingsPieces << " total pieces-> "
392.                         << totalPieces << std::endl;
393.     if (numPieces + numKingPieces == 0) // if the current player is out of pieces this is a terminal state
394.     {
395.         if (Pieces::outputDebugData > 1)
396.             std::cout << indentValue << Pieces::ANSII_RED_COUT << "Current Player is in TERMINAL STATE!! "
397.                         << Pieces::ANSII_END << std::endl;
398.         currentTerminal = -877775; // returning a high value. the current player is about to lose
399.                                     // Avoid this move!
400.     }
401.     if (numOpponentPieces + numOpponentKingsPieces == 0) // if the current player is out of pieces this is a ter
  minal state
402.     {
403.         //if (Pieces::outputDebugData > 1)
404.         std::cout << indentValue << Pieces::ANSII_RED_COUT << "Opponent is in TERMINAL STATE!! "
405.                         << Pieces::ANSII_END << std::endl;
406.         opponentTerminal = 777775; // The opponent's is about to lose
407.                                     // Go for it if possible
408.     }
409.     // Piece preservation and the taking of opponent pieces are global values. In other words, if no pieces are
  lost

```

```

410.    // or taken, all moves that have this result will have the same evaluation value.
411.    // This loop looks at each player's piece and the board generates a value that will separate certain moves f
     rom
412.    // the rest.
413.    for (int i = 1; i <= 32; ++i)
414.    {
415.        // is a player or opponent in this square?  RV (0 = empty, 1 = man, 2 = king)
416.        playerPiece = state.getPieceInSquare(i, color);
417.        opponentPiece = state.getPieceInSquare(i, switchPlayerColor(color));
418.        // get value of current player's board
419.        if (playerPiece == KING) // piece is a King
420.        {
421.            // this encourages a king to move towards the center of the board
422.            positionAdder += squareValuesForKing[i - 1] * 10;
423.            // Cross of Pain Calculation
424.            // look for opponents in column + or - 1 or row + or - 8.  if true this is a good place to be
425.            // Current piece is not on the right edge!
426.            if ((i - 1) % 4 != 0 && state.getPieceInSquare((i - 1) % 4, switchPlayerColor(color)) != 0)
427.                positionAdder += crossOfPainValue;
428.            // Current piece is not on the left edge!  i.e. they are in the same row
429.            if (((i - 1) / 4 + 1) == (i / 4 + 1) && state.getPieceInSquare((i + 1) % 4, switchPlayerColor(color)
 ) != 0)
430.                positionAdder += crossOfPainValue;
431.            // Current piece is not on the top row!
432.            if (i - 8 > 0 && state.getPieceInSquare((i - 8), switchPlayerColor(color)) != 0)
433.                positionAdder += crossOfPainValue;
434.            // Current piece is not on the bottom row!
435.            if (i + 8 < 33 && state.getPieceInSquare((i + 8), switchPlayerColor(color)) != 0)
436.                positionAdder += crossOfPainValue;
437.        }
438.        else if (playerPiece == MAN) // Regular pieces are encouraged to move towards the opponent's back row.
439.        {
440.            if (color == Color::RED)
441.            {
442.                positionAdder += squareValuesForRed[i - 1];
443.            }

```

```

444.         else if (color == Color::BLACK)
445.     {
446.         positionAdder += squareValuesForBlack[i - 1];
447.     }
448. }
449. //Get value of opponent player's board using the same method as player
450. if (opponentPiece == KING) // piece is a King
451. {
452.     // this encourages a king to move towards the center of the board
453.     opponentPositionAdder += squareValuesForKing[i - 1] * 10;
454.     // Does the current player end up in the Cross of Pain Calculation
455.     // look for opponents in column + or - 1 or row + or - 8. if true this is a good place to be
456.     // Current piece is not on the right edge!
457.     if ((i - 1) % 4 != 0 && state.getPieceInSquare((i - 1) % 4, switchPlayerColor(color)) != 0)
458.         opponentPositionAdder += crossOfPainValue;
459.     // Current piece is not on the left edge!
460.     if (((i - 1) / 4 + 1) == (i / 4 + 1) && state.getPieceInSquare((i + 1) % 4, switchPlayerColor(color)
461. ) != 0)
462.         opponentPositionAdder += crossOfPainValue;
463.     // Current piece is not on the top row!
464.     if (i - 8 > 0 && state.getPieceInSquare((i - 8), switchPlayerColor(color)) != 0)
465.         opponentPositionAdder += crossOfPainValue;
466.     // Current piece is not on the bottom row!
467.     if (i + 8 < 33 && state.getPieceInSquare((i + 8), switchPlayerColor(color)) != 0)
468.         opponentPositionAdder += crossOfPainValue;
469. }
470. else if (playerPiece == MAN) // Opponent's Regular pieces are encouraged to move towards the back row.
471. {
472.     if (color == Color::RED)
473.     {
474.         opponentPositionAdder += squareValuesForRed[i - 1];
475.     }
476.     else if (color == Color::BLACK)
477.     {
478.         opponentPositionAdder += squareValuesForBlack[i - 1];
479.     }

```

```

479.         }
480.     }
481.     playerScore = preservePlayersPieces + positionAdder + currentTerminal;
482.     opponentScore = (preserveOpponentsPieces + opponentPositionAdder + reduceOpponentKingValue) * opponentEvaluationWeight + opponentTerminal;
483.     moveScore = playerScore - opponentScore;
484.     if (Pieces::outputDebugData > 1)
485.         std::cout << indentValue << Pieces::ANSII_GREEN_COUT << " Evaluated Move: moveScore-"
486.             > << Pieces::ANSII_END
487.             << moveScore << std::endl;
488.
489.     // Clamp the final score based on the current scoring
490.     // system. Note that these values can later be adjusted
491.     // to use a wider range.
492.     if (moveScore <= -8000000)
493.     {
494.         moveScore = -7999999;
495.     }
496.     if (moveScore >= 9000000)
497.     {
498.         moveScore = 7999999;
499.     }
500.     return moveScore;
501. }
502. /**
503. * Third evaluation function
504. * @author Borislav Sabotinov
505. *
506. * @param Board State
507. * @param Color color
508. *
509. * @return an integer score of how good we think the state is
510. */
511. int Algorithm::evalFuncThree(Board state, Color color)
512. {

```

```

513. /* Declaration of "boards" with a weight for each square
514. Two for regular pieces of each color and two for kings of each color
515. Encourage player to keep two pieces in the back for defense
516. leaving 10 pieces for offense. Advance to the center
517. but somewhat in waves as a cluster, to avoid suicidal pieces that expose themselves
518. Try to attack the opponent's "double corner" from where a kinged piece can escape faster
519. */
520. int squareValuesForRed[] = {7, 1, 7, 1,
521.                             1, 2, 2, 2,
522.                             1, 5, 5, 5,
523.                             1, 3, 3, 3,
524.                             1, 4, 4, 4,
525.                             1, 5, 250, 250,
526.                             1, 250, 500, 500,
527.                             50, 100, 100, 1000};
528.
529. int squareValuesForBlack[] = {1000, 100, 100, 50,
530.                             500, 500, 250, 1,
531.                             250, 250, 5, 1,
532.                             4, 4, 4, 1,
533.                             3, 3, 3, 1,
534.                             5, 5, 5, 1,
535.                             2, 2, 2, 1,
536.                             1, 7, 1, 7};
537.
538. // Kings preference for center, with some traversal lines
539. // to attempt and avoid a "back-and-forth" pattern
540. int squareValuesForKing[] = {1, 1, 1, 1,
541.                             1, 5, 5, 55,
542.                             5, 15, 45, 1,
543.                             1, 5, 35, 5,
544.                             5, 25, 25, 1,
545.                             1, 15, 5, 15,
546.                             5, 5, 5, 10,
547.                             1, 1, 1, 1};
548.

```

```

549.     std::string colorTxt = (color == Color::RED) ? " (RED is Friendly) " : " (BLACK is Friendly) ";
550.     // KING has 4 moves max, so value is 4; MAN has 2 moves max so values is 2
551.     const int KING = 2, MAN = 1, KING_VALUE = 4, MAN_VALUE = 2;
552.
553.     int numPlayerTotalPieces = state.getNumPlayerTotalPieces(color);
554.     int numEnemyTotalPieces = state.getNumPlayerTotalPieces(switchPlayerColor(color));
555.     int numPlayerTotalKings = state.getNumKingPieces(color);
556.     int numEnemyTotalKings = state.getNumKingPieces(switchPlayerColor(color));
557.     int numPlayerTotalMen = numPlayerTotalPieces - numPlayerTotalKings;
558.     int numEnemyTotalMen = numEnemyTotalPieces - numEnemyTotalKings;
559.
560.     int numKingsScore = numPlayerTotalKings * KING_VALUE;
561.     int numMenScore = numPlayerTotalMen * MAN_VALUE;
562.
563.     int diffInNumPieces = numPlayerTotalPieces - numEnemyTotalPieces;
564.     int diffInNumKings = numPlayerTotalKings - numEnemyTotalKings;
565.     int diffInNumMen = numPlayerTotalMen - numEnemyTotalMen;
566.
567.     // PIECE BONUS/PENALTY - UNCOMMENT TO ACTIVATE
568.     // if diff in kings is positive, score is amplified with a bonus
569.     // if diff in kings is negative, however, score is penalized accordingly (by adding a negative)
570.     // numKingsScore += (40 * diffInNumKings);
571.
572.     // if diff in men is positive, score is amplified with a bonus
573.     // if diff in men is negative, however, score is penalized accordingly (by adding a negative)
574.     // numMenScore += (20 * diffInNumMen);
575.     // END PIECE BONUS/PENALTY
576.
577.     int casualtyScore = 0, captureScore = 0, positionScore = 0, playerPiece = 0, enemyPiece = 0, advancementScore
      e = 0;
578.
579.     std::vector<Board::Move> playerMoves = state.moveGen(color);
580.     std::vector<Board::Move> enemyMoves = state.moveGen(switchPlayerColor(color));
581.
582.     Pieces playerPieces = state.getPlayerPieces(color);
583.     Pieces opponentPieces = state.getOpponentPieces(color);

```

```

584.     Pieces *p_playerPieces = &playerPieces;
585.     Pieces *p_opponentPieces = &opponentPieces;
586.
587.     std::vector<Board::Move> playerJumpsForPiece;
588.     std::vector<Board::Move> opponentJumpsForPiece;
589.
590.     // helper values to quickly check if a piece is in a certain notable location
591.     // back rows for each color player to determine KING-ing
592.     // sides indicate limited moves
593.     const long long redBackRowGrp = (1LL << 1) | (1LL << 2) | (1LL << 3) | (1LL << 4);
594.     const long long blackBackRowGrp = (1LL << 29) | (1LL << 30) | (1LL << 31) | (1LL << 32);
595.     const long long sideColumnGrp = (1LL << 5) | (1LL << 13) | (1LL << 21) | (1LL << 12) | (1LL << 20) | (1LL <<
596.     28);
597.
598.     // CHECK TERMINAL STATE
599.     if (enemyMoves.size() == 0)
600.         return 7999999; // good for us if enemy has no moves left!
601.     else if (playerMoves.size() == 0)
602.         return -7999999; // bad for us if we're out of moves!
603.
604.     // MAIN LOOP FOR SCORING POSITION
605.     for (int piece = 0; piece < 32; piece++)
606.     {
607.         // BEGIN POSITION SCORE SECTION
608.         if (color == Color::RED)
609.         {
610.             playerPiece = state.getPieceInSquare(piece, color);
611.             if (playerPiece == MAN)
612.                 positionScore += (squareValuesForRed[piece] * MAN_VALUE);
613.             else if (playerPiece == KING)
614.                 positionScore += (squareValuesForKing[piece] * KING_VALUE);
615.         }
616.         else if (color == Color::BLACK)
617.         {
618.             playerPiece = state.getPieceInSquare(piece, color);

```

```

619.             positionScore += (squareValuesForBlack[piece] * MAN_VALUE);
620.         else if (playerPiece == KING)
621.             positionScore += (squareValuesForKing[piece] * KING_VALUE);
622.     }
623.     // END POSITION SCORE SECTION
624. }
625.
626. // Check our moves; 1000 points for a safe capture, 2000 points for a multi-jump
627. for (int i = 0; i < playerMoves.size(); i++)
628. {
629.     if (playerMoves.at(i).removalSquare.size() > 1)
630.     {
631.         if (Pieces::outputDebugData)
632.             std::cout << " INSIDE EVAL-
3: We " << colorTxt << " can capture multiple pieces on this state! "
633.                                         << "Start: " << playerMoves.at(i).startSquare << "End: " << playerMoves.at(i).destinationSquare.back()
634.                                         << " " << std::endl;
635.
636.         captureScore += 2000;
637.     }
638.     else if (playerMoves.at(i).removalSquare.size() == 1)
639.     {
640.
641.         int enemyCaptureSqr = playerMoves.at(i).removalSquare.back();
642.         int enemyCaptureType = state.getPieceInSquare(enemyCaptureSqr, switchPlayerColor(color)); // it's an
enemy piece, what is it's type?
643.
644.         if (enemyCaptureType == KING)
645.         {
646.             if (Pieces::outputDebugData)
647.                 std::cout << " INSIDE EVAL-3: We can capture enemy KING! " << colorTxt << std::endl;
648.
649.             captureScore += 400;
650.         }
651.         else if (enemyCaptureType == MAN)

```

```

652.         captureScore += 200;
653.
654.         int destSqr = playerMoves.at(i).destinationSquare.back();
655.         std::vector<int> adjMoves = state.boardMoveTable[destSqr].moves;
656.
657.         if (color == Color::RED)
658.         {
659.             for (int j = 0; j < adjMoves.size(); j++)
660.             {
661.                 if (adjMoves.at(j) > destSqr) // check enemy MEN and KING below
662.                 {
663.                     int enemyPiece = state.getPieceInSquare(adjMoves.at(j), switchPlayerColor(color));
664.                     if (enemyPiece == MAN || enemyPiece == KING)
665.                         captureScore -= 100; // not safe
666.                 }
667.                 else if (adjMoves.at(j) < destSqr) // we're red, anything above us can only capture if enemy
668.                     KING
669.                 {
670.                     int enemyPiece = state.getPieceInSquare(adjMoves.at(j), switchPlayerColor(color));
671.                     if (enemyPiece == KING)
672.                         captureScore -= 100; // not safe
673.                 }
674.                 else
675.                     captureScore += 1000; // we're safe to capture
676.             }
677.         }
678.         else
679.         {
680.             for (int j = 0; j < adjMoves.size(); j++)
681.             {
682.                 if (adjMoves.at(j) < destSqr) // check enemy MEN and KING above
683.                 {
684.                     int enemyPiece = state.getPieceInSquare(adjMoves.at(j), switchPlayerColor(color));
685.                     if (enemyPiece == MAN || enemyPiece == KING)
686.                         captureScore -= 100; // not safe
687.                 }
688.             }
689.         }
690.     }
691. }

```

```

687.             else if (adjMoves.at(j) > destSqr) // we're black, anything below us can only capture if enemy
688.                 y KING
689.             {
690.                 int enemyPiece = state.getPieceInSquare(adjMoves.at(j), switchPlayerColor(color));
691.                 if (enemyPiece == KING)
692.                     captureScore -= 100; // not safe
693.                 captureScore += 1000; // we're safe to capture
694.             }
695.         }
696.     }
697. }
698.
699. // BEGIN CASUALTY SECTION
700. for (int j = 0; j < enemyMoves.size(); j++)
701. {
702.     //std::cout << "we are here" << std::endl;
703.     if (enemyMoves.at(j).removalSquare.size() > 1)
704.     {
705.         if (Pieces::outputDebugData)
706.             std::cout << " INSIDE EVAL-
3: Enemy can capture multiple pieces, avoid!" << colorTxt << std::endl;
707.
708.         casualtyScore -= 40000; // we lose too much, really bad
709.     }
710.     else if (enemyMoves.at(j).removalSquare.size() == 1)
711.     {
712.         // friendly piece is captured, what is it's type?
713.         int capturedPieceType = state.getPieceInSquare(enemyMoves.at(j).removalSquare.at(0), color);
714.         if (capturedPieceType == KING)
715.         {
716.             if (Pieces::outputDebugData)
717.                 std::cout << " INSIDE EVAL-3: Enemy can capture a KING, avoid!" << colorTxt << std::endl;
718.
719.             casualtyScore -= 4000; // we lose a KING, a valuable piece
720.         }

```

```

721.     else if (capturedPieceType == MAN) // we lose one MAN
722.     {
723.         int opponentDestinationSqr = enemyMoves.at(j).destinationSquare.at(0);
724.         std::vector<int> adjMoves = state.boardMoveTable[opponentDestinationSqr].moves;
725.
726.         if (color == Color::RED)
727.         {
728.             // if we are RED, opponent is Black; if BLACK enemy lands on our back row, avoid at all cost
729.             // we are therefore trying to minimize the chance of an enemy getting a KING
730.             if ((1 << opponentDestinationSqr) & redBackRowGrp)
731.                 casualtyScore -= 5000;
732.             // opponent lands on their own back row; not so bad but we can't retaliate so avoid
733.             else if ((1 << opponentDestinationSqr) & blackBackRowGrp)
734.                 casualtyScore -= 2000;
735.             // opponent lands on one of the side squares, where we cannot retaliate. Avoid as well
736.             else if ((1 << opponentDestinationSqr) & sideColumnGrp)
737.                 casualtyScore -= 2000;
738.             else
739.             {
740.                 // if (diffInNumMen >= 1)
741.                 // {
742.                 for (int j = 0; j < adjMoves.size(); j++)
743.                 {
744.                     int ourPiece = state.getPieceInSquare(adjMoves.at(j), switchPlayerColor(color));
745.                     // check if we have a king below; we're red, only our king can go upwards
746.                     if (adjMoves.at(j) > opponentDestinationSqr)
747.                     {
748.                         if (ourPiece == KING)
749.                             captureScore += 1000; // can retaliate
750.                     }
751.                     // we're red, we can retaliate with MAN or KING if enemy is above
752.                     else if (adjMoves.at(j) < opponentDestinationSqr)
753.                     {
754.                         if (ourPiece == MAN || ourPiece == KING)
755.                             captureScore += 1000; // can retaliate
756.                     }

```

```

757.             else
758.                 captureScore += 0; // we cannot capture
759.             }
760.         // }
761.     }
762. }
763. else
764. {
    // if we are BLACK, opponent is Red; if RED enemy lands on our back row, avoid at all cost
    // we are therefore trying to minimize the chance of an enemy getting a KING
    if ((1 << opponentDestinationSqr) & blackBackRowGrp)
        casualtyScore -= 5000;
    // opponent lands on their own back row; not so bad but we can't retaliate so avoid
    else if ((1 << opponentDestinationSqr) & redBackRowGrp)
        casualtyScore -= 2000;
    // opponent lands on one of the side squares, where we cannot retaliate. Avoid as well
    else if ((1 << opponentDestinationSqr) & sideColumnGrp)
        casualtyScore -= 2000;
    // if we've gotten this far, we lose one MAN and opponent lands somewhere we can retaliate
    // We ask - Can we? If yes, do it if we have piece parity or an advantage of more pieces
    else
    {
        // if (numPlayerTotalPieces >= numEnemyTotalPieces)
        // {
        for (int j = 0; j < adjMoves.size(); j++)
        {
            int ourPiece = state.getPieceInSquare(adjMoves.at(j), switchPlayerColor(color));
            if (adjMoves.at(j) < opponentDestinationSqr) // check our KING above
            {
                if (ourPiece == KING)
                    captureScore += 1000; // can retaliate
            }
            // we're black, if enemy is below we can retaliate with MAN or KING
            else if (adjMoves.at(j) > opponentDestinationSqr)
            {
                if (ourPiece == MAN || ourPiece == KING)

```

```

793.                               captureScore += 1000; // can retaliate
794.                           }
795.                           captureScore += 0; // we cannot retaliate
796.                       }
797.                   }
798.               }
799.           }
800.       }
801.   }
802. }
803. //END CASUALTY SECTION
804.
805.     int compositeScore = numMenScore + numKingsScore + advancementScore + positionScore + captureScore + casualtyScore;
806.
807.     return compositeScore;
808. }
809.
810. /**
811. * wrapper function that will decide which of the actual three eval functions to call
812. * @author Borislav Sabotinov
813. *
814. * @param Board position
815. * @param Player player
816. * @param int evalVersion - used to determine which of the 3 eval functions to call
817. *
818. * @return a Result struct, which consists of a value and a move.
819. */
820. int Algorithm::staticEval(Board state, Color color, int evalVersion)
821. {
822.     int scoreOfGoodness;
823.
824.     switch (evalVersion)
825.     {
826.         case 1:
827.             scoreOfGoodness = evalFunctOne(state, color);

```

```

828.         break;
829.     case 2:
830.         scoreOfGoodness = evalFuncTwo(state, color);
831.         break;
832.     case 3:
833.         scoreOfGoodness = evalFuncThree(state, color);
834.         break;
835.     default:
836.     {
837.         scoreOfGoodness = 1; // default and debug value. Player takes first option everytime
838.         if (Pieces::outputDebugData > 1)
839.             std::cout << Pieces::ANSII_GREEN_COUT << " TEST/DEBUG EVALUATION BRANCH RETURN VALUE = 1 " << Pieces
840.             ::ANSII_END << std::endl;
841.         //throw std::runtime_error("Error: eval function # may only be 1, 2, or 3!");
842.     }
843. }
844. return scoreOfGoodness;
845. }
846.
847. /**
848. * deepEnough - Basic if check, if currentDepth >= maxDepth, true; otherwise false
849. *
850. * @author Randall Henderson
851. *
852. */
853. bool Algorithm::deepEnough(int currentDepth)
854. {
855.     if (currentDepth <= 0)
856.         return true;
857.     else
858.         return false;
859. }
860.
861. /**
862. * minimax algorithm returns the position of the best move

```

```

863.     * @author Randall Henderson
864.     *
865.     * @param Board board
866.     * @param int depth
867.     * @param Color color
868.     * @param int passThresh = 9000000
869.     * @param int useThresh = -8000000
870.     *
871.     * @return a Result struct, which consists of a value and a Move
872.     */
873.
874. Algorithm::Result Algorithm::minimax_a_b(Board state, int depth, Color color, int useThresh, int passThresh)
875. {
876.     Algorithm::Result result; // Return structure for MiniMaxAB
877.     std::string indentValue; // Tracking header in cout statements. Shows level in recursion
878.     std::string playerColor; // Used for debugging output
879.     Board::Move bestPath; // best move struct - starts as a null move
880.     int newValue;
881.
882.     if (color == Color::RED)
883.     {
884.         playerColor = Pieces::ANSII_RED_START;
885.         playerColor.append(" RED ");
886.         playerColor.append(Pieces::ANSII_END);
887.     }
888.     else
889.     {
890.         playerColor = "BLACK ";
891.     }
892.
893.     indentValue = playerColor;
894.
895.     indentValue.append("1.");
896.
897.     // Debug code for values passed into the function
898.     if (Pieces::outputDebugData > 0 && depth == 0)

```

```

899.         std::cout << indentValue << Pieces::ANSII_GREEN_COUT << "Entering MINIMAX_A_B Value: " << result.value
900.             << " Depth-> " << depth << " useThresh-> " << useThresh << " passThresh-> " << passThresh
901.                 << Pieces::ANSII_END << std::endl;
902.
903.         indentValue.append("2.");
904.
905.         if (deepEnough(depth)) // deep enough and Terminal could be combined. Separated for error tracking
906.     {
907.         result.value = staticEval(state, color, evalVersion);
908.
909.         if (Pieces::outputDebugData > 0)
910.             std::cout << indentValue << Pieces::ANSII_RED_COUT << "Deep Enough, Move Evaluated. Returning -> "
911.                             << result.value << " Bestmove destination size " << result.bestMove.destinationSquare.size
912.             () << Pieces::ANSII_END << std::endl;
913.
914.         minimaxLeafNodes++;
915.
916.         // to deal with odd depths and keep it in the right order
917.         if (callingPlayer.getColor() != color)
918.             result.value = -result.value;
919.
920.         return result;
921.     }
922.
923.     indentValue.append("3.");
924.
925.     std::vector<Board::Move> successors = movegen(state, color);
926.
927.     // Current Player has no moves. This is the equivalent to deep enough or terminal move
928.     if (successors.size() == 0)
929.     {
930.         result.value = staticEval(state, color, evalVersion);
931.
932.         if (Pieces::outputDebugData)
933.             std::cout << indentValue << Pieces::ANSII_RED_COUT << "Player has no moves. Returning -> "
934.                             << result.value << Pieces::ANSII_END << std::endl;

```

```

934.
935.        minimaxLeafNodes++;
936.
937.        // to deal with odd depths and keep it in the right order
938.        if (callingPlayer.getColor() != color)
939.            result.value = -result.value;
940.
941.        return result;
942.    }
943.
944.    for (int successorIndex = 0; successorIndex < successors.size(); successorIndex++)
945.    {
946.        //Create a board at the current iteration of successors
947.        Board tmpState = state.updateBoard(successors.at(successorIndex), color);
948.
949.        minimaxExpandedNodes++;
950.
951.        indentValue.append(">.");
952.
953.        if (Pieces::outputDebugData)
954.            std::cout << indentValue << Pieces::ANSII_BLUE_COUT << "Checking Moves: Move #-
955. > " << successorIndex + 1
956.             << " Passed in parameters: Start-
957. > " << successors.at(successorIndex).startSquare << " Move to-> "
958.                 << successors.at(successorIndex).destinationSquare.back() << " Depth-> " << depth
959.                 << Pieces::ANSII_END << std::endl;
960.
961.        // recursive call
962.        Result resultSucc = minimax_a_b(tmpState, depth - 1, switchPlayerColor(color), -passThresh, -useThresh);
963.
964.        if (Pieces::outputDebugData)
965.            std::cout << indentValue << Pieces::ANSII_GREEN_START << "Recursive Return: Just checked-> "
966.                << successors.at(successorIndex).startSquare << " -> "
967.                << successors.at(successorIndex).destinationSquare.back()
968.                << " New Value-> " << resultSucc.value << " Depth-> "
969.                << depth << " useThresh-> " << useThresh << " passThresh-> " << passThresh

```

```

968.             << Pieces::ANSII_END << std::endl;
969.
970.             newValue = -resultSucc.value;
971.
972.             if (newValue > passThresh) // Found the Best Move
973.             {
974.                 if (Pieces::outputDebugData > 0 && depth == this->maxDepth)
975.                     std::cout << indentValue << Pieces::ANSII_YELLOW_COUT << "New Best Move.  From-
976. > " << successors.at(successorIndex).startSquare
977.                               << " to-> " << successors.at(successorIndex).destinationSquare.back()
978.                               << " Change PassThresh  Old: " << passThresh
979.                               << " to  New: " << newValue << Pieces::ANSII_END << std::endl;
980.
981.                 passThresh = newValue;
982.                 bestPath = successors.at(successorIndex);
983.             }
984.
985.             if (passThresh >= useThresh) // Best move on the branch.  No need to look anymore
986.             {
987.                 if (Pieces::outputDebugData)
988.                     std::cout << indentValue << Pieces::ANSII_YELLOW_COUT << "AB-
989. CUTOFF!!  Best Move on the Branch.  PassThresh -> "
990.                               << Pieces::ANSII_END << passThresh << " UseThresh-
991. > " << useThresh << " Returning " << std::endl;
992.
993.                 result.value = passThresh;
994.                 result.bestMove = successors.at(successorIndex);
995.             }
996.
997.             result.value = passThresh;
998.             result.bestMove = bestPath;
999.             return result;
1000.         }

```

```

1001.
1002. /**
1003.  * Alpha Beta Search
1004.  * @author Borislav Sabotinov
1005. *
1006. * @param Board state
1007. *
1008. * @return a Result struct, which consists of a value and a Move
1009. */
1010. Algorithm::Result Algorithm::alphaBetaSearch(Board state)
1011. {
1012.     if (Pieces::outputDebugData)
1013.     {
1014.         if (callingPlayer.getColor() == Color::RED)
1015.             std::cout << "RED ";
1016.         else
1017.             std::cout << "BLACK ";
1018.         if (Pieces::outputDebugData > 2)
1019.             std::cout << "In alphaBetaSearch...." << std::endl;
1020.     }
1021.
1022.     int alpha = std::numeric_limits<int>::min(); // tracks best value for max, initialized to WORST case
1023.     int beta = std::numeric_limits<int>::max(); // tracks best value for min, initialized to WORST case
1024.
1025.     return maxValue(state, maxDepth, alpha, beta, callingPlayer.getColor());
1026. }
1027.
1028. /**
1029.  * maxValue - algorithm from Russel & Norvig, implemented to fit this program
1030.  * This function would determine the score of goodness for a state passed in to the MAX player
1031.  * There is indirect recursion as MAX calls MIN, which calls MAX again and so on.
1032.  * @author Borislav Sabotinov
1033. *
1034. * @param Board state
1035. * @param int alpha
1036. * @param int beta

```

```

1037.    *
1038.    * @return int utilityValue
1039.    */
1040. Algorithm::Result Algorithm::maxValue(Board state, int depth, int alpha, int beta, Color color)
1041. {
1042.     if (Pieces::outputDebugData)
1043.     {
1044.         if (color == Color::RED)
1045.             std::cout << "\nRED ";
1046.         else
1047.             std::cout << "\nBLACK ";
1048.         std::cout << "In maxValue()! Depth is " << depth << std::endl;
1049.     }
1050.
1051.     Algorithm::Result result;
1052.     Board::Move bestMove;
1053.
1054.     if (deepEnough(depth))
1055.     {
1056.         absearchLeafNodes++;
1057.         result.value = staticEval(state, color, evalVersion);
1058.         return result;
1059.     }
1060.
1061.     std::vector<Board::Move> listOfActions = actions(state, color);
1062.
1063.     // terminal check
1064.     if (listOfActions.size() == 0)
1065.     {
1066.         absearchLeafNodes++;
1067.         result.value = staticEval(state, color, evalVersion); // eval acts as utility funct
1068.         return result;
1069.     }
1070.
1071.     if (Pieces::outputDebugData)
1072.         std::cout << "Not yet at a terminal state...." << std::endl;

```

```

1073.
1074.     result.value = std::numeric_limits<int>::min();
1075.
1076.     for (int actionIndex = 0; actionIndex < listOfActions.size(); actionIndex++)
1077.     {
1078.         Player::printMove(listOfActions.at(actionIndex), color, false);
1079.         absearchExpandedNodes++;
1080.         Board tmpState = state.updateBoard(listOfActions.at(actionIndex), color);
1081.         Algorithm::Result minValueResult = minValue(tmpState, depth - 1, alpha, beta, switchPlayerColor(color));
1082.
1083.         if (minValueResult.value > result.value) // Best move located
1084.         {
1085.             result.value = minValueResult.value;
1086.             bestMove = listOfActions.at(actionIndex);
1087.         }
1088.
1089.         if (result.value >= beta) // no need to examine branch
1090.         {
1091.             result.bestMove = listOfActions.at(actionIndex);
1092.             return result;
1093.         }
1094.
1095.         alpha = std::max(alpha, result.value);
1096.     }
1097.
1098.     if (Pieces::outputDebugData)
1099.     {
1100.         std::cout << "alpha: " << alpha << " beta: " << beta << " val: " << result.value << " move start: " << result.bestMove.startSquare << std::endl;
1101.         for (int i = 0; i < result.bestMove.destinationSquare.size(); i++)
1102.             std::cout << "dest: " << result.bestMove.destinationSquare.at(i) << std::endl;
1103.     }
1104.
1105.     result.bestMove = bestMove;
1106.     return result;
1107. }

```

```

1108.
1109. /**
1110.  * minValue function - algorithm from Russel & Norvig, implemented to fit this program
1111.  * This function would determine the score of goodness for a state passed in to the MIN player
1112.  * There is indirect recursion as MAX calls MIN, which calls MAX again and so on.
1113.  * @author Borislav Sabotinov
1114. *
1115. * @param Board state
1116. * @param int alpha
1117. * @param int beta
1118. *
1119. * @return Result structure, which contains a value score and a Board::Move bestMove structure
1120. */
1121. Algorithm::Result Algorithm::minValue(Board state, int depth, int alpha, int beta, Color color)
1122. {
1123.     if (Pieces::outputDebugData)
1124.     {
1125.         if (color == Color::RED)
1126.             std::cout << "\nRED ";
1127.         else
1128.             std::cout << "\nBLACK ";
1129.         std::cout << "In minValue()! Depth is " << depth << std::endl;
1130.     }
1131.
1132.     Result result;
1133.     Board::Move bestMove;
1134.
1135.     if (deepEnough(depth))
1136.     {
1137.         absearchLeafNodes++;
1138.         result.value = staticEval(state, color, evalVersion);
1139.         return result;
1140.     }
1141.
1142.     std::vector<Board::Move> listOfActions = actions(state, color);
1143.

```

```

1144.     // terminal check
1145.     if (listOfActions.size() == 0)
1146.     {
1147.         absearchLeafNodes++;
1148.         result.value = staticEval(state, color, evalVersion); // eval acts as utility funct
1149.         return result;
1150.     }
1151.
1152.     if (Pieces::outputDebugData)
1153.         std::cout << "Not yet at a terminal state...." << std::endl;
1154.
1155.     result.value = std::numeric_limits<int>::max();
1156.
1157.     for (int actionIndex = 0; actionIndex < listOfActions.size(); actionIndex++)
1158.     {
1159.         Player::printMove(listOfActions.at(actionIndex), color, false);
1160.         absearchExpandedNodes++;
1161.         Board tmpState = state.updateBoard(listOfActions.at(actionIndex), color);
1162.         Result maxValueResult = maxValue(tmpState, depth - 1, alpha, beta, switchPlayerColor(color));
1163.
1164.         if (maxValueResult.value < result.value) // Best move located
1165.         {
1166.             result.value = maxValueResult.value;
1167.             bestMove = listOfActions.at(actionIndex);
1168.         }
1169.
1170.         if (result.value <= alpha)
1171.         {
1172.             result.bestMove = listOfActions.at(actionIndex);
1173.             return result;
1174.         }
1175.
1176.         beta = std::min(beta, result.value);
1177.     }
1178.
1179.     if (Pieces::outputDebugData)

```

```

1180.         {
1181.             std::cout << "alpha: " << alpha << " beta: " << beta << " val: " << result.value << " move start: " << r
1182.             result.bestMove.startSquare << std::endl;
1183.             for (int i = 0; i < result.bestMove.destinationSquare.size(); i++)
1184.                 std::cout << "dest: " << result.bestMove.destinationSquare.at(i) << std::endl;
1185.         }
1186.         result.bestMove = bestMove;
1187.         return result;
1188.     }
1189.
1190. /**
1191. * Helper function to switch a color, so if we are RED and we want our opponent, we would get BACK
1192. * @author Borislav Sabotinov
1193. *
1194. * @param Color color - the color we want to invert
1195. *
1196. * @return Color - the color opposite of what we passed in
1197. */
1198. Color Algorithm::switchPlayerColor(Color color)
1199. {
1200.     if (color == Color::RED)
1201.         return Color::BLACK;
1202.     else
1203.         return Color::RED;
1204. }
1205.
1206. /**
1207. * terminalTest function is used by alpha-beta-search to determine if
1208. * the move either ends the game or leads to a leaf node.
1209. * @author Borislav Sabotinov
1210. *
1211. * First checks if we're at maxDepth (i.e. at a leaf/terminal node).
1212. * If yes, return true. Otherwise, next we check if the move ends the game.
1213. * For example - if Red player ends up with zero pieces, they lost and the game is over.
1214. *

```

```

1215.     * @param Board state
1216.     *
1217.     * @return bool isTerminalState
1218.     */
1219. bool Algorithm::terminalTest(Board state, int depth)
1220. {
1221.     bool isTerminalState = false;
1222.     std::vector<Board::Move> redMoves = state.moveGen(Color::RED);
1223.     std::vector<Board::Move> blackMoves = state.moveGen(Color::BLACK);
1224.     if (Pieces::outputDebugData)
1225.         std::cout << "Red Moves " << redMoves.size() << " Black Moves " << blackMoves.size() << std::endl;
1226.
1227.     if (redMoves.size() == 0 || blackMoves.size() == 0)
1228.         isTerminalState = true;
1229.
1230.     return isTerminalState;
1231. }
1232.
1233. /**
1234. * Utility function to determine the best move
1235. * Essentially a wrapper function that calls staticEval
1236. * @author Borislav Sabotinov
1237. */
1238. int Algorithm::utility(Board state)
1239. {
1240.     //return staticEval(state, this->callingPlayer, this->evalVersion);
1241.     return 1;
1242. }
1243.
1244. /**
1245. * actions method determines the list of possible actions, or moves, a player can make
1246. * Essentially a wrapper function to call Board's movegen function.
1247. * @author Borislav Sabotinov
1248. *
1249. * @param Board state
1250. *

```

```
1251.     * @return vector<Board::Move> - a list of possible moves for a player of a given color
1252.     */
1253.     std::vector<Board::Move> Algorithm::actions(Board state, Color color)
1254.     {
1255.         return state.moveGen(color);
1256.     }
1257.
1258. /**
1259. * Set the evaluation function version - 1, 2, or 3
1260. */
1261. void Algorithm::setEvalVersion(int evalVersion)
1262. {
1263.     this->evalVersion = evalVersion;
1264. }
1265.
1266. /**
1267. * Set the max depth that will be used
1268. */
1269. void Algorithm::setMaxDepth(int maxDepth)
1270. {
1271.     this->maxDepth = maxDepth;
1272. }
1273.
1274. int Algorithm::passSign(int passThresh)
1275. {
1276.     if (passThresh < 0)
1277.         return -1;
1278.     else
1279.         return 1;
1280. }
```

Simulation.hpp

```
1. #ifndef SIMULATION_H
2. #define SIMULATION_H
3.
4. #include "Board.hpp"
5. #include "Game.hpp"
6.
7. /**
8.  * Header definition for class Simulation.
9.  * @author Borislav Sabotinov
10. *
11. * This class is responsible for managing the series of Games that AI players will play for Project Two.
12. * It persists during the execution of the program and keeps track of the number of games played.
13. * It also allows to aggregate and print simulation analysis details.
14. *
15. * There are THREE (3) evaluation functions, one for each team member; two algorithms (minimax and Alpha-Beta).
16. * The simulation will execute each in turn.
17. *
18. * Fifteen runs with depth 2:
19. *   1. MinMax-A-B with Evl. Function #1 Verses Alpha-Beta with Evl. Function #1
20. *   2. MinMax-A-B with Evl. Function #1 Verses Alpha-Beta with Evl. Function #2
21. *   3. MinMax-A-B with Evl. Function #1 Verses Alpha-Beta with Evl. Function #3
22. *
23. *   4. MinMax-A-B with Evl. Function #2 Verses Alpha-Beta with Evl. Function #1
24. *   5. MinMax-A-B with Evl. Function #2 Verses Alpha-Beta with Evl. Function #2
25. *   6. MinMax-A-B with Evl. Function #2 Verses Alpha-Beta with Evl. Function #3
26. *
27. *   7. MinMax-A-B with Evl. Function #3 Verses Alpha-Beta with Evl. Function #1
28. *   8. MinMax-A-B with Evl. Function #3 Verses Alpha-Beta with Evl. Function #2
29. *   9. MinMax-A-B with Evl. Function #3 Verses Alpha-Beta with Evl. Function #3
30. *
31. *   10. MinMax-A-B with Evl. Function #1 Verses MinMax-A-B with Evl. Function #2
32. *   11. MinMax-A-B with Evl. Function #1 Verses MinMax-A-B with Evl. Function #3
33. *   12. MinMax-A-B with Evl. Function #2 Verses MinMax-A-B with Evl. Function #3
```

```

34. *
35. * 13. Alpha-Beta with Evl. Function #1 Verses Alpha-Beta with Evl. Function #2
36. * 14. Alpha-Beta with Evl. Function #1 Verses Alpha-Beta with Evl. Function #3
37. * 15. Alpha-Beta with Evl. Function #2 Verses Alpha-Beta with Evl. Function #3
38. *
39. * Fifteen runs with depth 4:
40. * 1. MinMax-A-B with Evl. Function #1 Verses Alpha-Beta with Evl. Function #1
41. * 2. MinMax-A-B with Evl. Function #1 Verses Alpha-Beta with Evl. Function #2
42. * 3. MinMax-A-B with Evl. Function #1 Verses Alpha-Beta with Evl. Function #3
43. *
44. * 4. MinMax-A-B with Evl. Function #2 Verses Alpha-Beta with Evl. Function #1
45. * 5. MinMax-A-B with Evl. Function #2 Verses Alpha-Beta with Evl. Function #2
46. * 6. MinMax-A-B with Evl. Function #2 Verses Alpha-Beta with Evl. Function #3
47. *
48. * 7. MinMax-A-B with Evl. Function #3 Verses Alpha-Beta with Evl. Function #1
49. * 8. MinMax-A-B with Evl. Function #3 Verses Alpha-Beta with Evl. Function #2
50. * 9. MinMax-A-B with Evl. Function #3 Verses Alpha-Beta with Evl. Function #3
51. *
52. * 10. MinMax-A-B with Evl. Function #1 Verses MinMax-A-B with Evl. Function #2
53. * 11. MinMax-A-B with Evl. Function #1 Verses MinMax-A-B with Evl. Function #3
54. * 12. MinMax-A-B with Evl. Function #2 Verses MinMax-A-B with Evl. Function #3
55. *
56. * 13. Alpha-Beta with Evl. Function #1 Verses Alpha-Beta with Evl. Function #2
57. * 14. Alpha-Beta with Evl. Function #1 Verses Alpha-Beta with Evl. Function #3
58. * 15. Alpha-Beta with Evl. Function #2 Verses Alpha-Beta with Evl. Function #3
59. *
60. * TOTAL 30 RUNS/GAMES WILL BE SIMULATED.
61. */
62.
63. class Simulation
64. {
65.
66. private:
67.     int numGamesPlayed;
68.
69.     // runs only games using Minimax algorithm

```

```

70. void runMinimaxOnly();
71. // runs only games using AB Prune algorithm
73. void runABPruneOnly();
74.
75. public:
76.     Simulation(); // constructor
77.     ~Simulation(); // destructor
78.
79.     // runs all games runs as delineated above
80.     void runFullSimulation();
81.
82.     // public method for specific simulations
83.     void runSpecificSimulation(int playerOneAlg, int playerOneEvalFunct, int playerTwoAlg, int PlayerTwoEvalFunct, int
84.     depth);
85.     // public method for player vs AI simulation
86.     void runPlayerVsAISimulation(int playerAlg, int playerEvalFunct, int depth);
87.
88.     // helper function to determine winner and break out of game loop
89.     static bool didSomeoneWin(Board board);
90.
91.     // returns a count of the number of games played in a simulation
92.     // each of the 3 run functions.
93.     int getNumGamesPlayed();
94.
95.     // creates a table with results for analysis.
96.     // how many nodes were created, etc.
97.     void generateAnalysisResults();
98.
99.     void printGameConfig(int redPlayerAlg, int redPlayerEvalFunct, int blackPlayerAlg, int blackPlayerEvalFunct, int d
100.     epth);
100.    void printGameResults(Game::GameOver endGameStatus);
101.
102.    // helper print methods
103.    static void printBlackWins();

```

```
104.     static void printRedWins();
105.     static void printDraw();
106. };
107.
108. #endif // !SIMULATION_H
```

Simulation.cpp

```
1. #include "Simulation.hpp"
2.
3. #include <iostream>
4. #include <stdexcept>
5.
6. /**
7.  * Simulation implementation
8.  * @author Borislav Sabotinov
9. *
10. * Responsible for driving the simulation based on user preference provided in Main
11. */
12.
13. Simulation::Simulation()
14. {
15.     this->numGamesPlayed = 0;
16. }
17.
18. Simulation::~Simulation()
19. {
20. }
21.
22. /**
23. * Runs all games runs as delineated in Simulation.hpp
24. * R = redundant game simulated
25. *
```

```

26.* 1. p1_alg: 1 p1_eval: 1 p2_alg: 1 p2_eval: 1 R
27.* 2. p1_alg: 1 p1_eval: 1 p2_alg: 1 p2_eval: 2
28.* 3. p1_alg: 1 p1_eval: 1 p2_alg: 1 p2_eval: 3
29.* 4. p1_alg: 1 p1_eval: 2 p2_alg: 1 p2_eval: 1 R
30.* 5. p1_alg: 1 p1_eval: 2 p2_alg: 1 p2_eval: 2 R
31.* 6. p1_alg: 1 p1_eval: 2 p2_alg: 1 p2_eval: 3
32.* 7. p1_alg: 1 p1_eval: 3 p2_alg: 1 p2_eval: 1 R
33.* 8. p1_alg: 1 p1_eval: 3 p2_alg: 1 p2_eval: 2 R
34.* 9. p1_alg: 1 p1_eval: 3 p2_alg: 1 p2_eval: 3
35.* 10. p1_alg: 1 p1_eval: 1 p2_alg: 2 p2_eval: 1
36.* 11. p1_alg: 1 p1_eval: 1 p2_alg: 2 p2_eval: 2
37.* 12. p1_alg: 1 p1_eval: 1 p2_alg: 2 p2_eval: 3
38.* 13. p1_alg: 1 p1_eval: 2 p2_alg: 2 p2_eval: 1
39.* 14. p1_alg: 1 p1_eval: 2 p2_alg: 2 p2_eval: 2
40.* 15. p1_alg: 1 p1_eval: 2 p2_alg: 2 p2_eval: 3
41.* 16. p1_alg: 1 p1_eval: 3 p2_alg: 2 p2_eval: 1
42.* 17. p1_alg: 1 p1_eval: 3 p2_alg: 2 p2_eval: 2
43.* 18. p1_alg: 1 p1_eval: 3 p2_alg: 2 p2_eval: 3
44.* 19. p1_alg: 2 p1_eval: 1 p2_alg: 2 p2_eval: 1 R
45.* 20. p1_alg: 2 p1_eval: 1 p2_alg: 2 p2_eval: 2 R
46.* 21. p1_alg: 2 p1_eval: 1 p2_alg: 2 p2_eval: 3
47.* 22. p1_alg: 2 p1_eval: 2 p2_alg: 2 p2_eval: 1
48.* 23. p1_alg: 2 p1_eval: 2 p2_alg: 2 p2_eval: 2
49.* 24. p1_alg: 2 p1_eval: 2 p2_alg: 2 p2_eval: 3
50.* 25. p1_alg: 2 p1_eval: 3 p2_alg: 2 p2_eval: 1 R
51.* 26. p1_alg: 2 p1_eval: 3 p2_alg: 2 p2_eval: 2
52.* 27. p1_alg: 2 p1_eval: 3 p2_alg: 2 p2_eval: 3 R
53./*
54.*/
55.void Simulation::runFullSimulation()
56.{
57.    std::cout << "\033[0;32mRunning a FULL simulation!\033[0m" << std::endl;
58.
59.    for (int depth = 2; depth <= 4; depth += 2)
60.    {
61.        std::cout << "Depth: " << depth << std::endl;

```

```

62.
63.     for (int p1_alg = 0; p1_alg < 2; p1_alg++)
64.     {
65.         for (int p2_alg = 0; p2_alg < 2; p2_alg++)
66.         {
67.             for (int p1_eval = 1; p1_eval < 4; p1_eval++)
68.             {
69.                 for (int p2_eval = 1; p2_eval < 4; p2_eval++)
70.                 {
71.                     // omit duplicates to save time - we only care about unique runs
72.                     // player 1 need not use alg 2. It's already covered by Player 2
73.                     if (p1_alg == 2 && p2_alg == 1)
74.                         continue;
75.
76.                     std::cout << "p1_alg: " << p1_alg << " p1_eval: " << p1_eval << " p2_alg: "
77.                                         << p2_alg << " p2_eval: " << p2_eval << std::endl;
78.
79.                     Game *game = new Game(p1_alg, p1_eval, p2_alg, p2_eval, depth);
80.                     Game::GameOver endGameStatus = game->startGame();
81.                     numGamesPlayed++;
82.                     printGameResults(endGameStatus);
83.                     delete game;
84.
85.                 } // p2_eval
86.             } // p1_eval
87.         } // p2_alg
88.     } // p1_alg
89. } // depth
90.}
91.
92./*
93. * Allows the user to run a specific, custom simulation based on their preference
94. * @param int redPlayerAlg - If 1, minimax; if 0, AB Prune
95. * @param int redPlayerEvalFunct - 1,2, 3, or 4
96. * @param int blackPlayerAlg - If 1, minimax; if 0, AB Prune
97. * @param int blackPlayerEvalFunct - 1,2, 3, or 4

```

```

98. * @param int depth - 2 to 15, preferably 2 or 4 as per project requirements
99. */
100. void Simulation::runSpecificSimulation(int redPlayerAlg, int redPlayerEvalFunct, int blackPlayerAlg, int blackPlayerEvalFunct, int depth)
101. {
102.     std::cout << Pieces::ANSII_GREEN_START << "Running a SINGLE game, specific simulation!" << Pieces::ANSII_END
103.     << std::endl;
104.     // Validate algorithm selections
105.     if ((redPlayerAlg < 0 || redPlayerAlg > 1) && (blackPlayerAlg < 0 || blackPlayerAlg > 1))
106.         throw std::runtime_error("Error: algorithm may only be 1 (minimax-a-b) or 0 (ab-prune)!");
107.
108.     // Validate evaluation function selections
109.     if ((redPlayerEvalFunct <= 0 || redPlayerEvalFunct > 4) && (blackPlayerEvalFunct <= 0 || blackPlayerEvalFunct > 4))
110.         throw std::runtime_error("Error: evalFunction may only be 1, 2, 3, or 4!");
111.
112.     // Validate depth
113.     if (depth <= 1 || depth > 15)
114.         throw std::runtime_error("Error: depth must be > 1 and <= 15. ");
115.
116.     Game *game = new Game(redPlayerAlg, redPlayerEvalFunct, blackPlayerAlg, blackPlayerEvalFunct, depth);
117.     Game::GameOver endGameStatus = game->startGame();
118.
119.     printGameResults(endGameStatus);
120.     printGameConfig(redPlayerAlg, redPlayerEvalFunct, blackPlayerAlg, blackPlayerEvalFunct, depth);
121.
122.     delete game;
123. }
124.
125. /**
126. * Helper function to print overall Game results, depending on who won
127. * @param Game::GameOver endGameStatus - the game will return to us who was the winner (if any)
128. */
129. void Simulation::printGameResults(Game::GameOver endGameStatus)
130. {

```

```

131.     if (endGameStatus == Game::GameOver::BLACK_WINS)
132.         printBlackWins();
133.     else if (endGameStatus == Game::GameOver::RED_WINS)
134.         printRedWins();
135.     else if (endGameStatus == Game::GameOver::DRAW)
136.         printDraw();
137.     else
138.         std::cout << "Oops, something went wrong!" << std::endl;
139.     }
140.
141. /**
142. * runPlayerVsAISimulation - play a game with a human against a computer player
143. * This is a fun option, it allows a person to test themselves against the AI
144. * @param int playerAlg - what alg will the AI use
145. * @param int playerEvalFunct - what eval function will AI use
146. * @param int depth - what depth will AI use
147. */
148. void Simulation::runPlayerVsAISimulation(int playerAlg, int playerEvalFunct, int depth)
149. {
150.
151.     Player computerPlayer = Player(playerAlg, Color::RED, depth, playerEvalFunct);
152.     bool gameOver = false;
153.     int moveSelection;
154.     Color computerPlayerColor = Color::RED;
155.     Color humanPlayerColor = Color::BLACK;
156.     Color currentPlayerColor = humanPlayerColor;
157.     Board board;
158.     board.printBoard();
159.
160.     while (!gameOver)
161.     {
162.         if (currentPlayerColor == humanPlayerColor) // BLACK
163.         {
164.             std::vector<Board::Move> blackMoves = board.moveGen(humanPlayerColor);
165.             // PRINT OUT BLACK'S MOVES
166.             std::cout << "Black's moves (b/B): ";

```

```

167.         for (int blackMoveIter = 0; blackMoveIter < blackMoves.size(); blackMoveIter++)
168.     {
169.         std::cout << "<" << blackMoveIter + 1 << "> " << blackMoves.at(blackMoveIter).startSquare;
170.         for (int destinationIter = 0; destinationIter < blackMoves.at(blackMoveIter).destinationSquare.s
171.             ize(); destinationIter++)
172.             {
173.                 std::cout << " to " << blackMoves.at(blackMoveIter).destinationSquare.at(destinationIter);
174.             }
175.             std::cout << ", ";
176.         std::cout << std::endl;
177.
178.         // GET HUMAN PLAYER MOVE
179.         bool isSelectionValid = false;
180.         while (!isSelectionValid)
181.         {
182.             std::cout << "Select BLACK (Human) move: ";
183.             std::cin >> moveSelection;
184.             if (moveSelection > blackMoves.size() || moveSelection < 0)
185.             {
186.                 std::cerr << "Out of range; please enter a valid choice!" << std::endl;
187.             }
188.             else
189.             {
190.                 board = board.updateBoard(blackMoves.at(moveSelection - 1), Color::BLACK);
191.                 currentPlayerColor = computerPlayerColor; // RED
192.                 isSelectionValid = true;
193.             }
194.         }
195.     }
196.     else if (currentPlayerColor == computerPlayerColor) // RED
197.     {
198.         // AI TAKES TURN AND PRINTS BOARD
199.         int numPiecesTakenByAI = computerPlayer.takeTurn(board);
200.         currentPlayerColor = humanPlayerColor; // BLACK
201.     }

```

```

202.
203.        // CHECK WIN-LOSS CONDITIONS
204.        gameOver = didSomeoneWin(board); // if true, game will end
205.    }
206.    board.printBoard(); // print final board after someone wins
207. }
208.
209. /**
210. * At the end of the game, print the game configuration the user provided for ease of reference
211. * @param int redPlayerAlg - If 1, minimax; if 0, AB Prune
212. * @param int redPlayerEvalFunct - 1,2, 3, or 4
213. * @param int blackPlayerAlg - If 1, minimax; if 0, AB Prune
214. * @param int blackPlayerEvalFunct - 1,2, 3, or 4
215. * @param int depth - 2 to 15, preferably 2 or 4 as per project requirements
216. */
217. void Simulation::printGameConfig(int redPlayerAlg, int redPlayerEvalFunct, int blackPlayerAlg, int blackPlayerEv
   alFunct, int depth)
218. {
219.     std::string algs[2] = {"Alpha-Beta-Search", "Minimax-Alpha-Beta"};
220.     std::cout << "Red player alg: " << algs[redPlayerAlg] << ", eval: " << redPlayerEvalFunct << std::endl;
221.     std::cout << "Black player alg: " << algs[blackPlayerAlg] << ", eval: " << blackPlayerEvalFunct << std::endl
   ;
222.     std::cout << "Depth: " << depth << std::endl;
223. }
224.
225. /**
226. * didSomeoneWin - returns true if one player won, to break out of game loops
227. * @param Board board
228. *
229. * @return true if someone won, otherwise false
230. */
231. bool Simulation::didSomeoneWin(Board board)
232. {
233.     bool isGameOver = false;
234.     std::vector<Board::Move> redMoves = board.moveGen(Color::RED);
235.     std::vector<Board::Move> blackMoves = board.moveGen(Color::BLACK);

```

```

236.
237.     if (blackMoves.size() == 0)
238.     {
239.         isGameOver = true;
240.         printRedWins();
241.     }
242.     else if (redMoves.size() == 0)
243.     {
244.         isGameOver = true;
245.         printBlackWins();
246.     }
247.
248.     return isGameOver;
249. }
250.
251. /**
252. * Helper function to display if Red wins
253. */
254. void Simulation::printRedWins()
255. {
256.     std::cout << "\nRED WINS!!!" << std::endl;
257.     std::cout << "RED Player: <(-_-')>" << std::endl;
258.     std::cout << "But most importantly, BLACK looooses (boooo!)" << std::endl;
259.     std::cout << "BLACK Player: (╯°□°)╯︵ ┻━┻" << std::endl;
260. }
261.
262. /**
263. * Helper function to display if Black wins
264. */
265. void Simulation::printBlackWins()
266. {
267.     std::cout << "\nBLACK WINS!!!" << std::endl;
268.     std::cout << "BLACK Player: <(-_-')>" << std::endl;
269.     std::cout << "But most importantly, RED looooses (boooo!)" << std::endl;
270.     std::cout << "RED Player: (╯°□°)╯︵ ┻━┻" << std::endl;

```

```
271. }
272.
273. /**
274.  * Helper function to display if we run out of permitted turns and get a draw
275. */
276. void Simulation::printDraw()
277. {
278.     std::cout << "DRAW!!!" << std::endl;
279.     std::cout << "Red - (•_•)▫ ▫( `Д' ▫) - Black" << std::endl;
280.     std::cout << "Mission FAILED...We'll get em next time!" << std::endl;
281. }
282.
283. /**
284.  * getNumGamesPlayed - returns a count of the number of games played in a simulation
285. *
286. */
287. int Simulation::getNumGamesPlayed()
288. {
289.     return numGamesPlayed;
290. }
291.
292. /**
293.  * generateAnalysisResults creates a table with results for analysis how many nodes were created, etc.
294. */
295. void Simulation::generateAnalysisResults()
296. {
297. }
```

main.cpp

```
1. #include <iostream>
2. #include <string.h> // used by strcmp method
3.
4. #include "Simulation.hpp"
5. #include "Game.hpp"
6. #include "Board.hpp"
7. #include "Player.hpp"
8. #include "Pieces.hpp"
9. #include "Algorithm.hpp"
10.
11. /**
12. * Main entry way into the application via main() method.
13. * @author Borislav Sabotinov
14. *
15. * The user can display a help menu.
16. * The user will be prompted to select how they wish to interact with the program.
17. * Available options are:
18. *   1. a full simulation,
19. *   2. partial (single game) simulation,
20. *   3. player vs. player, or
21. *   4. player vs. AI.
22. */
23.
24.// helper functions to make main() more readable and concise
25.void printWelcomeMsg();
26.void printHelpMenu();
27.void printMainMenuOptions();
28.void executeRunBasedOnUserInput(int userInput, bool &isValid);
29.void getCustomSimUserInput(int &computerPlayerAlg, int &computerPlayerEval, int &depth);
30.void getCustomSimUserInput(int &playerOneAlg, int &playerOneEvalFunct, int &playerTwoAlg, int &playerTwoEvalFunct, int &depth);
31.void runManualGame();
32.void goodbye();
```

```
33.
34. /**
35. * Main function, which serves as an entry point to the Checkers application.
36. * @author Borislav Sabotinov
37. *
38. * User may invoke a help menu by passing in either -h or -help as a CLI parameter when launching the program.
39. *
40. * @param int argc - count of the number of CLI arguments provided
41. * @param char* argv[] - char array of the CLI arguments
42. *
43. * @return EXIT_SUCCESS if the program completes successfully
44. */
45.int main(int argc, char *argv[])
46.{
47.    // display help menu
48.    if (argc >= 2)
49.    {
50.        std::string cliArg = argv[1];
51.
52.        if (cliArg == "-h" || cliArg == "-help")
53.        {
54.            printHelpMenu();
55.            return EXIT_SUCCESS;
56.        }
57.
58.        if (cliArg == "-nc") // disable color
59.        {
60.            // ANSI codes for colored text, to improve UI and readability
61.            Pieces::ANSII_BLUE_START = "";
62.            Pieces::ANSII_RED_START = "";
63.            Pieces::ANSII_RED_HIGH = "";
64.            Pieces::ANSII_END = "";
65.            Pieces::ANSII_GREEN_START = "";
66.            Pieces::ANSII_BLUE_COUT = "";
67.            Pieces::ANSII_RED_COUT = "";
68.            Pieces::ANSII_GREEN_COUT = "";
```

```

69.         Pieces::ANSII_YELLOW_COUT = "";
70.     }
71.     else if (cliArg == "-no") //
72.     {
73.         Pieces::outputDebugData = 0;
74.     }
75.     else if (cliArg == "-ncno")
76.     {
77.         // ANSI codes for colored text, to improve UI and readability
78.         Pieces::ANSII_BLUE_START = "";
79.         Pieces::ANSII_RED_START = "";
80.         Pieces::ANSII_RED_HIGH = "";
81.         Pieces::ANSII_END = "";
82.         Pieces::ANSII_GREEN_START = "";
83.         Pieces::ANSII_BLUE_COUT = "";
84.         Pieces::ANSII_RED_COUT = "";
85.         Pieces::ANSII_GREEN_COUT = "";
86.         Pieces::ANSII_YELLOW_COUT = "";
87.         Pieces::outputDebugData = 0;
88.     }
89. }
90.
91. printWelcomeMsg();
92. printMainMenuOptions();
93.
94. bool isValidInput = false;
95. while (!isValidInput)
96. {
97.     int userInput;
98.     std::cout << "Your choice " << Pieces::ANSII_BLUE_START << "(1, 2, 3, or 4)" << Pieces::ANSII_END << ": ";
99.     std::cin >> userInput;
100.
101.     executeRunBasedOnUserInput(userInput, isValidInput);
102. }
103.
104. goodbye();

```

```

105.
106.         return EXIT_SUCCESS;
107.     }
108.
109.     /**
110.      * Prints a welcome message to the console, along with the authors' names and emails.
111.      */
112.     void printWelcomeMsg()
113.     {
114.         std::cout << Pieces::ANSII_GREEN_START << "Welcome to the Checkers AI Program." << Pieces::ANSII_END << std::endl;
115.         std::cout << "Authors: " << Pieces::ANSII_RED_START << " David Torrente (dat54@txstate.edu), Randall Henders
116.             on (rrh93@txstate.edu), "
117.                 << "Borislav Sabotinov (bss64@txstate.edu)." << Pieces::ANSII_END << std::endl;
118.         std::cout << "Re-run this program with -h or -
119.             help CLI argument to see a help menu or refer to README for instructions."
120.                 << std::endl;
121.         std::cout << std::endl;
122.     }
123.
124.     /**
125.      * Prints a help message to the console if -h or -help are provided as CLI arguments when invoking the program
126.      */
127.     void printHelpMenu()
128.     {
129.         std::cout << "To use this program, please read the instructions below and re-launch." << std::endl;
130.         std::cout << "Additional details for building and execution are also available in the README file." << std::endl;
131.         std::cout << std::endl;
132.         std::cout << "Run with -nc for No Color, with -no for No Debug Output, or with -
133.             ncno for both No Color AND No Debug Output." << std::endl;
134.         std::cout << "When executing the program, you will be prompted to enter the algorithm and evaluation "
135.                 << "function for the simulation." << std::endl;

```

```

136.             << "evaluation functions sequentially."
137.             << "\nThis is the most common and preferred option." << std::endl;
138.     }
139.
140. /**
141. * Prints the main menu with option codes.
142. */
143. void printMainMenuOptions()
144. {
145.     std::cout << "NOTE: If 1 is selected below, you will NOT be prompted further for any eval function or algori
thm. "
146.             << "All will be simulated in order." << std::endl;
147.     std::cout << "For RED player, r = MAN and R = KING; for BLACK player, b = MAN and B = KING." << std::endl;
148.     std::cout << std::endl;
149.
150.     std::cout << "Choose a game mode below: " << std::endl;
151.     std::cout << "      1. Full Simulation (recommended)" << std::endl;
152.     std::cout << "      2. Single Custom Simulation" << std::endl;
153.     std::cout << "      3. Player vs Player (manual game)" << std::endl;
154.     std::cout << "      4. Player vs AI (will be asked to select AI playstyle)" << std::endl;
155.     std::cout << "      Ctrl + C to terminate program at any time." << std::endl;
156.     std::cout << std::endl;
157. }
158.
159. /**
160. * Given the user's choice in the main menu, execute the program accordingly.
161. */
162. void executeRunBasedOnUserInput(int userInput, bool &isValid)
163. {
164.     Simulation *simulation = new Simulation();
165.     switch (userInput)
166.     {
167.     case 1: // full sim
168.         isValid = true;
169.         simulation->runFullSimulation();
170.         std::cout << "# of Games Played: " << simulation->getNumGamesPlayed() << std::endl;

```

```

171.         break;
172.     case 2: // one custom sim
173.         isValid = true;
174.         int playerOneAlg, playerOneEvalFunct, playerTwoAlg, playerTwoEvalFunct, depth;
175.         getCustomSimUserInput(playerOneAlg, playerOneEvalFunct, playerTwoAlg, playerTwoEvalFunct, depth);
176.         simulation-
    >runSpecificSimulation(playerOneAlg, playerOneEvalFunct, playerTwoAlg, playerTwoEvalFunct, depth);
177.         break;
178.     case 3: // player vs. player
179.         isValid = true;
180.         runManualGame();
181.         break;
182.     case 4: // player vs. ai
183.         isValid = true;
184.         getCustomSimUserInput(playerOneAlg, playerOneEvalFunct, depth);
185.         simulation->runPlayerVsAISimulation(playerOneAlg, playerOneEvalFunct, depth);
186.         break;
187.     default:
188.         std::cerr << "Invalid option selected! Valid choices are 1, 2, 3, and 4" << std::endl;
189.     }
190. }
191.
192. /**
193. * For a custom simulation, obtains user input and returns via parameters
194. * @param int &computerPlayerAlg
195. * @param int &computerPlayerEval
196. * @param int &depth
197. *
198. * @return int &computerPlayerAlg passed in by reference
199. * @return int &computerPlayerEval passed in by reference
200. * @return int &depth passed in by reference
201. */
202. void getCustomSimUserInput(int &computerPlayerAlg, int &computerPlayerEval, int &depth)
203. {
204.     std::cout << "Please select the type of simulation you wish to run by entering in it's number." << std::endl
    ;

```

```

205.         std::cout << "1. Run Minimax-A-B algorithm" << std::endl;
206.         std::cout << "0. Run Alpha-Beta-Search algorithm" << std::endl;
207.
208.         // PLAYER CHOICES
209.         std::cout << "Algorithm for RED - Player 1 " << Pieces::ANSII_BLUE_START << "(1 for minimax, 0 for ab-
   Search)" << Pieces::ANSII_END << ":";
210.         std::cin >> computerPlayerAlg;
211.         std::cout << std::endl;
212.         std::cout << "Evaluation for RED - Player 1 " << Pieces::ANSII_BLUE_START << "(1 (David's), 2 (Randy's), 3 (
   Boris'), 4 (returns 1st available move))" << Pieces::ANSII_END << ":" ;
213.         std::cin >> computerPlayerEval;
214.
215.         // DEPTH
216.         std::cout << "Enter the depth for the search tree " << Pieces::ANSII_BLUE_START << "(2 or 4 recommended; min
   = 2, max = 15)" << Pieces::ANSII_END << ":" ;
217.         std::cin >> depth;
218.     }
219.
220. /**
221. * getCustomSimUserInput is a helper function to obtain the algorithm and eval function for Player 1 and Player
222. * 2
223. * One game will be simulated only using this input.
224. */
225. void getCustomSimUserInput(int &playerOneAlg, int &playerOneEvalFunct, int &playerTwoAlg, int &playerTwoEvalFunc
   t, int &depth)
226. {
227.     std::cout << "Please select the type of simulation you wish to run by entering in it's number." << std::endl
   ;
228.     std::cout << "1. Run Minimax-A-B algorithm" << std::endl;
229.     std::cout << "0. Run Alpha-Beta-Search algorithm" << std::endl;
230.
231.     // PLAYER ONE CHOICES
232.     std::cout << "Algorithm for RED - Player 1 " << Pieces::ANSII_BLUE_START << "(1 for minimax, 0 for ab-
   Search)" << Pieces::ANSII_END << ":";
233.     std::cin >> playerOneAlg;
234.     std::cout << std::endl;

```

```

234.         std::cout << "Evaluation for RED - Player 1 " << Pieces::ANSII_BLUE_START << "(1 (David's), 2 (Randy's), 3 (Boris'), 4 (returns 1st available move))" << Pieces::ANSII_END << ":" ;
235.         std::cin >> playerOneEvalFunc;
236.
237.         // PLAYER TWO CHOICES
238.         std::cout << "Algorithm for BLACK - Player 2 " << Pieces::ANSII_BLUE_START << "(1 for minimax, 0 for ab-Search)" << Pieces::ANSII_END << ":" ;
239.         std::cin >> playerTwoAlg;
240.         std::cout << std::endl;
241.         std::cout << "Evaluation for BLACK - Player 2 " << Pieces::ANSII_BLUE_START << "(1 (David's), 2 (Randy's), 3 (Boris'), 4 (returns 1st available move))" << Pieces::ANSII_END << ":" ;
242.         std::cin >> playerTwoEvalFunc;
243.
244.         // DEPTH
245.         std::cout << "Enter the depth for the search tree " << Pieces::ANSII_BLUE_START << "(2 or 4 recommended; min = 2, max = 15)" << Pieces::ANSII_END << ":" ;
246.         std::cin >> depth;
247.     }
248.
249. /**
250. * runManualGame function provides a human user the ability to play checkers with another human.
251. * It is a manual, input based game where a player must enter the number of the turn they wish to execute.
252. * @author David Torrente
253. * @author Borislav Sabotinov
254. */
255. void runManualGame()
256. {
257.     bool gameOver = false;
258.     int moveSelection;
259.     Color currentPlayer = Color::BLACK;
260.     std::vector<Board::Move> redMoves;
261.     std::vector<Board::Move> blackMoves;
262.     Board board;
263.     board.printBoard();
264.
265.     while (!gameOver)

```

```

266. {
267.     std::vector<Board::Move> redMoves = board.moveGen(Color::RED);
268.     std::vector<Board::Move> blackMoves = board.moveGen(Color::BLACK);
269.     std::cout << std::endl;
270.     std::cout << "Red (r = MAN / R = KING) moves: ";
271.
272.     for (int redMoveIter = 0; redMoveIter < redMoves.size(); redMoveIter++)
273.     {
274.         std::cout << "(" << redMoveIter + 1 << ")" << redMoves.at(redMoveIter).startSquare;
275.         for (int destinationIter = 0; destinationIter < redMoves.at(redMoveIter).destinationSquare.size(); d
estinationIter++)
276.         {
277.             std::cout << " to " << redMoves.at(redMoveIter).destinationSquare.at(destinationIter);
278.         }
279.         std::cout << ", ";
280.     }
281.     std::cout << std::endl;
282.
283.     std::cout << "Black (b = MAN / B = KING) moves: ";
284.     for (int blackMoveIter = 0; blackMoveIter < blackMoves.size(); blackMoveIter++)
285.     {
286.         std::cout << "(" << blackMoveIter + 1 << ")" << blackMoves.at(blackMoveIter).startSquare;
287.         for (int destinationIter = 0; destinationIter < blackMoves.at(blackMoveIter).destinationSquare.size(
); destinationIter++)
288.         {
289.             std::cout << " to " << blackMoves.at(blackMoveIter).destinationSquare.at(destinationIter);
290.         }
291.         std::cout << ", ";
292.     }
293.     std::cout << std::endl;
294.
295.     bool isSelectionValid = false;
296.     if (currentPlayer == Color::RED)
297.     {
298.         while (!isSelectionValid)
299.     {

```

```

300.         std::cout << "Select RED move: ";
301.         std::cin >> moveSelection;
302.         if (moveSelection > redMoves.size() || moveSelection < 0)
303.         {
304.             std::cerr << "Out of range; please enter a valid choice!" << std::endl;
305.         }
306.         else
307.         {
308.             board = board.updateBoard(redMoves.at(moveSelection - 1), Color::RED);
309.             currentPlayer = Color::BLACK;
310.             isSelectionValid = true;
311.         }
312.     }
313. }
314. else // BLACK's turn
315. {
316.     isSelectionValid = false;
317.     while (!isSelectionValid)
318.     {
319.         std::cout << "Select BLACK move: ";
320.         std::cin >> moveSelection;
321.         if (moveSelection > blackMoves.size() || moveSelection < 0)
322.         {
323.             std::cerr << "Out of range; please enter a valid choice!" << std::endl;
324.         }
325.         else
326.         {
327.             board = board.updateBoard(blackMoves.at(moveSelection - 1), Color::BLACK);
328.             currentPlayer = Color::RED;
329.             isSelectionValid = true;
330.         }
331.     }
332. }
333.
334. board.printBoard();
335. gameOver = Simulation::didSomeoneWin(board);

```

```
336.     }
337. }
338.
339. /**
340.  * Displays a compatible, pleasant graphic to the user as a goodbye
341. */
342. void goodbye()
343. {
344.     std::cout << std::endl;
345.     std::cout << Pieces::ANSII_BLUE_START << " _ _ _ _ " << Pieces::ANSII_END << std::endl;
346.     std::cout << Pieces::ANSII_BLUE_START << "/ _ / \W\W\W\W|_)\\/_/|_ | " << Pieces::ANSII_END << std::endl
347.     1;
348.     std::cout << Pieces::ANSII_BLUE_START << "\W\W|_)\\/_/\W\W/|_/_|_." << Pieces::ANSII_END << std::endl
349.     ;
350.     std::cout << Pieces::ANSII_BLUE_START << " " << Pieces::ANSII_END << std::endl;
351.     std::cout << std::endl;
352. }
```

Appendix B: Sample Output for 18 Games

Due to space constraints, complete output is available in individual files that accompany this project. Where possible, complete output with verbose printout is provided. These files, even the short ones with debug output disabled, are too long to include in this report. To provide an example – for Game 1 below, the short file is 3,397 lines long and the detailed file is 28,147 and 1.64 MB in size. The files are very readable, and the output is structured and well organized, so viewing the logs is preferable regardless.

This report will include a screenshot of the last state of the board and the game summary, which details how many nodes were expanded.

The naming convention for the output files is: `<alg>-<eval>-<alg>-<eval>-<depth>-<verbosity>.log`, where `alg` can be either `min` (for minimax-a-b) or `abs` (for alpha-beta-search), `eval` can be either 1, 2 or 3 (representing the evaluation functions), `depth` is either 2 or 4, and `verbosity` is either “`short`” or “`full`.”

A full game path is shown only for Game 1 and Game 10 for depths 2 and 4, respectively. This is due to size limitations – the output is simply too long to include in the report in full.

Game 1: (Minmax-A-B, EF-1) vs (A-B-Search, EF-1) – Depth 2

Refer to files min-1-abs-1-2-full and short.log for full output.

Round 80 Black's Move...
Black moves from 8 to destination (in sequence): dest: 4

	1	2	3	4	B
r	r	B			
5	6	7	8		
	9	10	11	12	
b	b				
13	14	15	16		
		R			
17	18	19	20		
21	22	23	24		
	25	26	27	28	
	R	R			
29	30	31	32		

RED Leaf Nodes: 2596
RED Expanded Nodes: 3295
RED Total Nodes: 5891

BLACK Leaf Nodes: 3004
BLACK Expanded Nodes: 3675
BLACK Total Nodes: 6679

DRAW!!!
Red - (я •_•)я ѿ('Д' ѿ) - Black
Mission FAILED...We'll get em next time!
Red player alg: Minimax-Alpha-Beta, eval: 1
Black player alg: Alpha-Beta-Search, eval: 1
Depth: 2

Figure 33: Minimax-1 vs ABS-1 Depth 2

Welcome to the Checkers AI Program.

Authors: David Torrente (dat54@txstate.edu), Randall Henderson (rrh93@txstate.edu), Borislav Sabotinov (bss64@txstate.edu).

Re-run this program with -h or -help CLI argument to see a help menu or refer to README for instructions.

NOTE: If 1 is selected below, you will NOT be prompted further for any eval function or algorithm. All will be simulated in order.
For RED player, r = MAN and R = KING; for BLACK player, b = MAN and B = KING.

Choose a game mode below:

1. Full Simulation (recommended)
 2. Single Custom Simulation
 3. Player vs Player (manual game)
 4. Player vs AI (will be asked to select AI playstyle)
- Ctrl + C to terminate program at any time.

Your choice (1, 2, 3, or 4): Please select the type of simulation you wish to run by entering in its number.

1. Run Minimax-A-B algorithm

0. Run Alpha-Beta-Search algorithm

Algorithm for RED - Player 1 (1 for minimax, 0 for ab-Search):

Evaluation for RED - Player 1 (1 (David's), 2 (Randy's), 3 (Boris'), 4 (returns 1st available move)): Algorithm for BLACK - Player 2 (1 for minimax, 0 for ab-Search):

Evaluation for BLACK - Player 2 (1 (David's), 2 (Randy's), 3 (Boris'), 4 (returns 1st available move)): Enter the depth for the search tree (2 or 4 recommended; min = 2, max = 15): Running a SINGLE game, specific simulation!

Round 1 Black's Move...

Black moves from 21 to destination (in sequence): dest: 17

		r		r		r		r		r	
	1		2		3		4				
r		r		r		r					
5		6		7		8					
	r		r		r		r				
	9		10		11		12				
	13		14		15		16				
	b										
	17		18		19		20				
		b		b		b					
	21		22		23		24				
	b		b		b		b				
	25		26		27		28				
	b		b		b		b				
	29		30		31		32				

Round 1 Red's Move...

Red moves from 12 to destination (in sequence): dest: 16

		r		r		r		r		r	
	1		2		3		4				
r		r		r		r					
5		6		7		8					
	r		r		r						
	9		10		11		12				
						r					
	13		14		15		16				

	b						
	17	18	19	20			
	b	b	b				
	21	22	23	24			
	b	b	b	b			
	25	26	27	28			
	b	b	b	b			
	29	30	31	32			

Round 2 Black's Move...

Black moves from 17 to destination (in sequence): dest: 13

	r	r	r	r	r	
	1	2	3	4		
r	r	r	r			
5	6	7	8			
r	r	r				
9	10	11	12			
b			r			
13	14	15	16			
17	18	19	20			
b	b	b	b			
21	22	23	24			
b	b	b	b			
25	26	27	28			
b	b	b	b			
29	30	31	32			

Round 2 Red's Move...

Red moves from 11 to destination (in sequence): dest: 15

	r	r	r	r	r	
	1	2	3	4		
r	r	r	r			
5	6	7	8			
r	r	r				
9	10	11	12			
b		r	r			
13	14	15	16			
17	18	19	20			
b	b	b	b			
21	22	23	24			
b	b	b	b			
25	26	27	28			
b	b	b	b			
29	30	31	32			

Round 3 Black's Move...

Black moves from 25 to destination (in sequence): dest: 21

	r	r	r	r	

	1	2	3	4	
r	r	r	r	r	
5	6	7	8		
r	r				
9	10	11	12		
b		r	r		
13	14	15	16		
17	18	19	20		
b	b	b	b		
21	22	23	24		
	b	b	b		
25	26	27	28		
b	b	b	b		
29	30	31	32		

Round 3 Red's Move...

Red moves from 10 to destination (in sequence): dest: 14

	r	r	r	r	
1	2	3	4		
r	r	r	r		
5	6	7	8		
r					
9	10	11	12		
b	r	r	r		
13	14	15	16		
17	18	19	20		
b	b	b	b		
21	22	23	24		
	b	b	b		
25	26	27	28		
b	b	b	b		
29	30	31	32		

Round 4 Black's Move...

Black moves from 22 to destination (in sequence): dest: 17

	r	r	r	r	
1	2	3	4		
r	r	r	r		
5	6	7	8		
r					
9	10	11	12		
b	r	r	r		
13	14	15	16		
b					
17	18	19	20		
b		b	b		
21	22	23	24		
	b	b	b		
25	26	27	28		
b	b	b	b		

|_29|____|_30|____|_31|____|_32|____|

Round 4 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	r		r		r		r
	1	2		3		4	
r		r			r		
5	6	7		8			
	r		r				
	9	10		11		12	
b		r		r		r	
13	14		15		16		
	b						
	17	18		19		20	
b			b		b		
21	22		23		24		
		b		b		b	
	25	26		27		28	
b		b		b		b	
29	30		31		32		

Round 5 Black's Move...

Black moves from 26 to destination (in sequence): dest: 22

	r		r		r		r
	1	2		3		4	
r		r			r		
5	6	7		8			
	r		r				
	9	10		11		12	
b		r		r		r	
13	14		15		16		
	b						
	17	18		19		20	
b		b		b		b	
21	22		23		24		
			b		b		
	25	26		27		28	
b		b		b		b	
29	30		31		32		

Round 5 Red's Move...

Red moves from 2 to destination (in sequence): dest: 7

	r			r		r
	1	2		3		4
r		r		r		
5	6	7		8		
	r		r			
	9	10		11		12
b		r		r		r
13	14		15		16	

	b						
17		18		19		20	
b	b	b	b	b	b	b	
21	22	23	24				
				b	b	b	
25	26	27		28			
b	b	b	b	b	b	b	
29	30	31	32				

Round 6 Black's Move...

Black moves from 24 to destination (in sequence): dest: 20

	r				r		r
	1	2		3		4	
r		r		r		r	
5	6	7		8			
	r	r					
	9	10		11		12	
b	r		r		r		
13	14	15		16			
	b					b	
	17	18		19		20	
b	b		b				
21	22	23		24			
			b		b		b
	25	26		27		28	
b	b		b		b		
29	30	31		32			

Round 6 Red's Move...

Red moves from 7 to destination (in sequence): dest: 11

	r				r		r	
—	—	1	—	2	—	3	—	4
r		r				r		
—	5	—	6	—	7	—	8	—
	r		r		r		r	
—	9	—	10	—	11	—	12	—
b		r		r		r		
—	13	—	14	—	15	—	16	—
	b						b	
—	17	—	18	—	19	—	20	—
b		b		b				
—	21	—	22	—	23	—	24	—
					b		b	
—	25	—	26	—	27	—	28	—
b		b		b		b		
—	29	—	30	—	31	—	32	—

Round 7 Black's Move...

Black moves from 27 to destination (in sequence): dest: 24

	1	2	3	4
r	r		r	
5	6	7	8	
	r	r	r	
	9	10	11	12
b	r	r	r	
13	14	15	16	
	b		b	
	17	18	19	20
b	b	b	b	
21	22	23	24	
			b	
	25	26	27	28
b	b	b	b	
29	30	31	32	

Round 7 Red's Move...

Red moves from 3 to destination (in sequence): dest: 7

	r				r
	1	2	3	4	
r	r	r	r		
5	6	7	8		
	r	r	r		
	9	10	11	12	
b	r	r	r		
13	14	15	16		
	b		b		
	17	18	19	20	
b	b	b	b		
21	22	23	24		
			b		
	25	26	27	28	
b	b	b	b		
29	30	31	32		

Round 8 Black's Move...

Black moves from 29 to destination (in sequence): dest: 25

	r				r
	1	2	3	4	
r	r	r	r		
5	6	7	8		
	r	r	r		
	9	10	11	12	
b	r	r	r		
13	14	15	16		
	b		b		
	17	18	19	20	
b	b	b	b		
21	22	23	24		
	b		b		
	25	26	27	28	
	b	b	b	b	
			b		
			25		

|_29|____|_30|____|_31|____|_32|____|

Round 8 Red's Move...

Red moves from 8 to destination (in sequence): dest: 12

	r						r
	1	2	3	4			
r	r	r					
5	6	7	8				
r	r	r	r				
9	10	11	12				
b	r	r	r				
13	14	15	16				
b			b				
17	18	19	20				
b	b	b	b				
21	22	23	24				
b			b				
25	26	27	28				
	b	b	b				
29	30	31	32				

Round 9 Black's Move...

Black moves from 30 to destination (in sequence): dest: 26

	r					r
	1	2	3	4		
r	r	r				
5	6	7	8			
r	r	r	r			
9	10	11	12			
b	r	r	r			
13	14	15	16			
b			b			
17	18	19	20			
b	b	b	b			
21	22	23	24			
b	b		b			
25	26	27	28			
	b	b	b			
29	30	31	32			

Round 9 Red's Move...

Red moves from 4 to destination (in sequence): dest: 8

	r						
	1	2	3	4			
r	r	r	r				
5	6	7	8				
r	r	r	r				
9	10	11	12				
b	r	r	r				
13	14	15	16				

		b							b	
		17		18		19		20		
b		b		b		b				
21		22		23		24				
	b	b					b			
	25		26		27		28			
			b		b					
29		30		31		32				

Round 10 Black's Move...

Black moves from 31 to destination (in sequence): dest: 27

		r								
		1		2		3		4		
r		r		r		r				
5		6		7		8				
	r	r		r		r		r		
	9		10		11		12			
b		r		r		r				
13		14		15		16				
	b						b			
	17		18		19		20			
b		b		b		b				
21		22		23		24				
	b	b		b		b		b		
	25		26		27		28			
					b					
29		30		31		32				

Round 10 Red's Move...

Red moves from 14 to destination (in sequence): dest: 18

		r								
		1		2		3		4		
r		r		r		r				
5		6		7		8				
	r	r		r		r		r		
	9		10		11		12			
b			r		r		r			
13		14		15		16				
	b		r			b				
	17		18		19		20			
b		b		b		b				
21		22		23		24				
	b	b		b		b		b		
	25		26		27		28			
					b					
29		30		31		32				

Round 11 Black's Move...

Black moves from 23 to destination (in sequence): dest: 14

		r								
		1		2		3		4		
r		r		r		r				
5		6		7		8				
	r	r		r		r		r		
	9		10		11		12			
b			r		r		r			
13		14		15		16				
	b		r			b				
	17		18		19		20			
b		b		b		b				
21		22		23		24				
	b	b		b		b		b		
	25		26		27		28			
					b					
29		30		31		32				

	1	2	3	4	
r	r	r	r	r	
5	6	7	8		
	r	r	r	r	
	9	10	11	12	
b	b	r	r		
13	14	15	16		
	b		b		
	17	18	19	20	
b	b		b		
21	22	23	24		
	b	b	b	b	
	25	26	27	28	
			b		
29	30	31	32		

BLACK player took 1 piece(s).

Round 11 Red's Move...

Red moves from 9 to destination (in sequence): dest: 18

	r					
	1	2	3	4		
r	r	r	r	r		
5	6	7	8			
	r	r	r			
	9	10	11	12		
b		r	r			
13	14	15	16			
	b	r		b		
	17	18	19	20		
b	b		b			
21	22	23	24			
	b	b	b	b		
	25	26	27	28		
			b			
29	30	31	32			

RED player took 1 piece(s).

Round 12 Black's Move...

Black moves from 26 to destination (in sequence): dest: 23

	r					
	1	2	3	4		
r	r	r	r	r		
5	6	7	8			
	r	r	r			
	9	10	11	12		
b		r	r			
13	14	15	16			
	b	r		b		
	17	18	19	20		
b	b	b	b	b		
21	22	23	24			
	b	b	b	b		

	25		26		27		28
				b			
	29		30		31		32

Round 12 Red's Move...

Red moves from 6 to destination (in sequence): dest: 9

	r						
	1	2		3		4	
r			r	r			
5	6	7		8			
	r	r	r	r			
	9	10		11		12	
b			r	r			
13	14	15		16			
	b	r		b			
	17	18		19		20	
b	b	b	b	b			
21	22	23		24			
	b		b	b			
	25	26		27		28	
				b			
29	30	31		32			

Round 13 Black's Move...

Black moves from 13 to destination (in sequence): dest: 6

	r						
	1	2		3		4	
r	b	r	r	r			
5	6	7		8			
	r	r	r	r			
	9	10		11		12	
		r	r	r			
13	14	15		16			
	b	r		b			
	17	18		19		20	
b	b	b	b	b			
21	22	23		24			
	b		b	b			
	25	26		27		28	
				b			
29	30	31		32			

BLACK player took 1 piece(s).

Round 13 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	r						
	1	2		3		4	
r	b	r	r	r			
5	6	7		8			
	r	r	r	r			

	9		10		11		12
					r		
13		14		15		16	
	b		r		r		b
	17		18		19		20
	b		b		b		
21		22		23		24	
	b			b		b	
	25		26		27		28
					b		
29		30		31		32	

Round 14 Black's Move...

Black moves from 22 to destination (in sequence) : dest: 15

	r						
	1		2		3		4
r		b		r		r	
5		6		7		8	
		r		r		r	
	9		10		11		12
			b		r		
13		14		15		16	
	b			r		b	
	17		18		19		20
	b			b		b	
21		22		23		24	
	b			b		b	
	25		26		27		28
					b		
29		30		31		32	

BLACK player took 1 piece(s).

Round 14 Red's Move...

Red moves from 19 to destination (in sequence): dest: 26

	r						
	1		2		3		4
r		b		r		r	
5		6		7		8	
		r		r		r	
	9		10		11		12
			b		r		
13		14		15		16	
	b			r		b	
	17		18		19		20
	b				b		
21		22		23		24	
	b		r		b		b
	25		26		27		28
					b		
29		30		31		32	

RED player took 1 piece(s).

Round 15 Black's Move...

Black moves from 6 to destination (in sequence): dest: 2

		r		B							
	1		2		3		4				
r			r		r						
5		6		7		8					
		r		r		r					
	9		10		11		12				
			b		r						
13		14		15		16					
	b					b					
	17		18		19		20				
b					b						
21		22		23		24					
	b		r		b		b				
	25		26		27		28				
						b					
29		30		31		32					

Round 15 Red's Move...

Red moves from 10 to destination (in sequence): dest: 19

		r		B							
	1		2		3		4				
r			r		r						
5		6		7		8					
			r		r						
	9		10		11		12				
					r						
13		14		15		16					
	b			r		b					
	17		18		19		20				
b				b							
21		22		23		24					
	b		r		b		b				
	25		26		27		28				
					b						
29		30		31		32					

RED player took 1 piece(s).

Round 16 Black's Move...

Black moves from 24 to destination (in sequence): dest: 15

		r		B							
	1		2		3		4				
r			r		r						
5		6		7		8					
			r		r						
	9		10		11		12				
			b		r						
13		14		15		16					
	b					b					
	17		18		19		20				
b				b							
21		22		23		24					
	b		r		b		b				
	25		26		27		28				
				b							
29		30		31		32					

	17		18		19		20
b							
21		22		23		24	
	b		r		b		b
	25		26		27		28
					b		
29		30		31		32	

BLACK player took 1 piece(s).

Round 16 Red's Move...

Red moves from 11 to destination (in sequence): dest: 18

	r	B				
	1	2	3	4		
r			r	r		
5	6	7	8		r	
	9	10	11	12		
	13	14	15	16		
b		r		b		
17	18	19	20			
b						
21	22	23	24			
b		r	b	b		
	25	26	27	28		
	29	30	31	32		

RED player took 1 piece(s).

Round 17 Black's Move...

Black moves from 2 to destination (in sequence): dest: 11

dest: 4

	r				B	
	1	2	3	4		
r						
5	6	7	8		r	
	9	10	11	12		
	13	14	15	16		
b		r		b		
17	18	19	20			
b						
21	22	23	24			
b		r	b	b		
	25	26	27	28		
29	30	31	32			

BLACK player took 2 piece(s).

Round 17 Red's Move...

Red moves from 26 to destination (in sequence): dest: 30

	r						B
1		2		3		4	
r							
5	6	7		8			
				r			
	9	10		11		12	
				r			
13	14	15		16			
b	r			b			
17	18	19		20			
b							
21	22	23		24			
b		b		b			
25	26	27		28			
	R			b			
29	30	31		32			

Round 18 Black's Move...

Black moves from 20 to destination (in sequence): dest: 11

	r						B
1		2		3		4	
r							
5	6	7		8			
			b		r		
	9	10		11		12	
				r			
13	14	15		16			
b	r			b			
17	18	19		20			
b							
21	22	23		24			
b		b		b			
25	26	27		28			
	R			b			
29	30	31		32			

BLACK player took 1 piece(s).

Round 18 Red's Move...

Red moves from 30 to destination (in sequence): dest: 26

	r						B
1		2		3		4	
r							
5	6	7		8			
			b		r		
	9	10		11		12	
				r			
13	14	15		16			
b	r			b			
17	18	19		20			
b							

_21		22		23		24	
	b		R		b		b
	25		26		27		28
						b	

Round 19 Black's Move...

Black moves from 25 to destination (in sequence): dest: 22

	r						B
	1		2		3		4
r							
5		6		7		8	
					b		r
	9		10		11		12
13		14		15		16	
	b		r				
	17		18		19		20
b		b					
21		22		23		24	
	r		R		b		b
	25		26		27		28
					b		
29		30		31		32	

Round 19 Red's Move...

Red moves from 18 to destination (in sequence): dest: 25

	r						B
	1		2		3		4
r							
5		6		7		8	
					b		r
	9		10		11		12
13		14		15		16	
	b						
	17		18		19		20
b							
21		22		23		24	
	r		R		b		b
	25		26		27		28
					b		
29		30		31		32	

RED player took 1 piece(s).

Round 20 Black's Move...

Black moves from 27 to destination (in sequence): dest: 23

	r						B
	1		2		3		4
r							

5		6		7		8	
				b		r	
	9		10		11		12
13		14		15		16	
	b						
	17		18		19		20
b			b				
21		22		23		24	
	r		R			b	
	25		26		27		28
				b			
29		30		31		32	

Round 20 Red's Move...

Red moves from 26 to destination (in sequence): dest: 19

	r						B
	1		2		3		4
r							
5		6		7		8	
				b		r	
	9		10		11		12
13		14		15		16	
	b			R			
	17		18		19		20
b							
21		22		23		24	
	r			b			
	25		26		27		28
				b			
29		30		31		32	

RED player took 1 piece(s).

Round 21 Black's Move...

Black moves from 28 to destination (in sequence): dest: 24

	r						B
	1		2		3		4
r							
5		6		7		8	
				b		r	
	9		10		11		12
13		14		15		16	
	b			R			
	17		18		19		20
b				b			
21		22		23		24	
	r			b			
	25		26		27		28
				b			
29		30		31		32	

Round 21 Red's Move...

Red moves from 19 to destination (in sequence): dest: 28

		r							B	
	1		2		3		4			
r										
5		6		7		8				
				b		r				
	9		10		11		12			
13		14		15		16				
	b									
	17		18		19		20			
b										
21		22		23		24				
	r						R			
	25		26		27		28			
					b					
29		30		31		32				
RED player took 1 piece(s).										

Round 22 Black's Move...

Black moves from 17 to destination (in sequence): dest: 14

		r							B	
	1		2		3		4			
r										
5		6		7		8				
				b		r				
	9		10		11		12			
13		14		15		16				
	b									
	17		18		19		20			
b										
21		22		23		24				
	r						R			
	25		26		27		28			
					b					
29		30		31		32				

Round 22 Red's Move...

Red moves from 25 to destination (in sequence): dest: 29

		r							B	
	1		2		3		4			
r										
5		6		7		8				
				b		r				
	9		10		11		12			
13		14		15		16				
	b									
	17		18		19		20			
b										
21		22		23		24				
	r						R			
	25		26		27		28			
					b					
29		30		31		32				

	17	18	19	20			
b							
21	22	23	24				
				R			
	25	26	27	b			
R							
29	30	31	32				

Round 23 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

		r						
	1		2		3		4	
r					B			
5		6		7		8		
				b		r		
	9		10		11		12	
		b						
13		14		15		16		
	17		18		19		20	
b								
21		22		23		24		
					R			
	25		26		27		28	
R					b			
29		30		31		32		

Round 23 Red's Move...

Red moves from 1 to destination (in sequence): dest: 6

	1		2		3		4	
r		r			B			
5		6		7		8		
				b		r		
	9		10		11		12	
		b						
13		14		15		16		
	17		18		19		20	
b								
21		22		23		24		
					R			
	25		26		27		28	
R					b			
29		30		31		32		

Round 24 Black's Move...

Black moves from 8 to destination (in sequence): dest: 3

					B			

	1	2	3	4
r	r			
5	6	7	8	
		b	r	
9	10	11	12	
	b			
13	14	15	16	
	17	18	19	20
b				
21	22	23	24	
				R
	25	26	27	28
R			b	
29	30	31	32	

Round 24 Red's Move...

Red moves from 28 to destination (in sequence): dest: 24

				B		
	1	2	3	4		
r	r					
5	6	7	8			
		b	r			
9	10	11	12			
	b					
13	14	15	16			
	17	18	19	20		
b				R		
21	22	23	24			
	25	26	27	28		
R			b			
29	30	31	32			

Round 25 Black's Move...

Black moves from 3 to destination (in sequence): dest: 7

	1	2	3	4		
r	r	B				
5	6	7	8			
		b	r			
9	10	11	12			
	b					
13	14	15	16			
	17	18	19	20		
b				R		
21	22	23	24			
	25	26	27	28		
R			b			
29	30	31	32			

|_29|____|_30|____|_31|____|_32|____|

Round 25 Red's Move...

Red moves from 29 to destination (in sequence): dest: 25

	1		2		3		4	
r		r		B				
5		6		7		8		
				b		r		
	9		10		11		12	
		b						
13		14		15		16		
	17		18		19		20	
b				R				
21		22		23		24		
	R							
25		26		27		28		
				b				
29		30		31		32		

Round 26 Black's Move...

Black moves from 7 to destination (in sequence): dest: 3

					B			
	1		2		3		4	
r		r						
5		6		7		8		
				b		r		
	9		10		11		12	
		b						
13		14		15		16		
	17		18		19		20	
b				R				
21		22		23		24		
	R							
25		26		27		28		
				b				
29		30		31		32		

Round 26 Red's Move...

Red moves from 12 to destination (in sequence): dest: 16

					B			
	1		2		3		4	
r		r						
5		6		7		8		
				b				
	9		10		11		12	
		b				r		
13		14		15		16		

	17	18	19	20			
b			R				
21	22	23	24				
	R						
	25	26	27	28			
			b				
29	30	31	32				

Round 27 Black's Move...

Black moves from 3 to destination (in sequence): dest: 7

	1	2	3	4			
r	r	B					
5	6	7	8				
	b						
	9	10	11	12			
	b		r				
13	14	15	16				
	17	18	19	20			
b			R				
21	22	23	24				
	R						
	25	26	27	28			
			b				
29	30	31	32				

Round 27 Red's Move...

Red moves from 25 to destination (in sequence): dest: 30

	1	2	3	4			
r	r	B					
5	6	7	8				
	b						
	9	10	11	12			
	b		r				
13	14	15	16				
	17	18	19	20			
b			R				
21	22	23	24				
	25	26	27	28			
	R		b				
29	30	31	32				

Round 28 Black's Move...

Black moves from 7 to destination (in sequence): dest: 3

			B				

	1	2	3	4
r	r			
5	6	7	8	
		b		
	9	10	11	12
	b		r	
13	14	15	16	
	17	18	19	20
b			R	
21	22	23	24	
	25	26	27	28
	R		b	
29	30	31	32	

Round 28 Red's Move...

Red moves from 16 to destination (in sequence): dest: 19

				B		
	1	2	3	4		
r	r					
5	6	7	8			
		b				
	9	10	11	12		
	b					
13	14	15	16			
			r			
	17	18	19	20		
b			R			
21	22	23	24			
	25	26	27	28		
	R		b			
29	30	31	32			

Round 29 Black's Move...

Black moves from 3 to destination (in sequence): dest: 7

	1	2	3	4		
r	r	B				
5	6	7	8			
		b				
	9	10	11	12		
	b					
13	14	15	16			
			r			
	17	18	19	20		
b			R			
21	22	23	24			
	25	26	27	28		
	R		b			
29	30	31	32			

|_29|____|_30|____|_31|____|_32|____|

Round 29 Red's Move...

Red moves from 19 to destination (in sequence): dest: 23

	1		2		3		4	
r		r		B				
5		6		7		8		
				b				
	9		10		11		12	
	b							
13		14		15		16		
17		18		19		20		
b			r		R			
21		22		23		24		
25		26		27		28		
		R			b			
29		30		31		32		

Round 30 Black's Move...

Black moves from 7 to destination (in sequence): dest: 3

					B			
	1		2		3		4	
r		r						
5		6		7		8		
				b				
	9		10		11		12	
	b							
13		14		15		16		
17		18		19		20		
b			r		R			
21		22		23		24		
25		26		27		28		
		R			b			
29		30		31		32		

Round 30 Red's Move...

Red moves from 23 to destination (in sequence): dest: 26

					B			
	1		2		3		4	
r		r						
5		6		7		8		
				b				
	9		10		11		12	
	b							
13		14		15		16		

	17	18	19	20				
b			R					
21	22	23	24					
	r							
	25	26	27	28				
	R		b					
29	30	31	32					

Round 31 Black's Move...

Black moves from 32 to destination (in sequence): dest: 27

					B			
	1	2	3	4				
r	r							
5	6	7	8					
	b							
	9	10	11	12				
	b							
13	14	15	16					
	17	18	19	20				
b			R					
21	22	23	24					
	r	b						
	25	26	27	28				
	R							
29	30	31	32					

Round 31 Red's Move...

Red moves from 24 to destination (in sequence): dest: 31

					B			
	1	2	3	4				
r	r							
5	6	7	8					
	b							
	9	10	11	12				
	b							
13	14	15	16					
	17	18	19	20				
b								
21	22	23	24					
	r							
	25	26	27	28				
	R	R						
29	30	31	32					

RED player took 1 piece(s).

Round 32 Black's Move...

Black moves from 3 to destination (in sequence): dest: 7

	1	2	3	4			
r	r	B					
5	6	7	8				
		b					
	9	10	11	12			
	b						
13	14	15	16				
	17	18	19	20			
b							
21	22	23	24				
		r					
	25	26	27	28			
	R	R					
29	30	31	32				

Round 32 Red's Move...

Red moves from 31 to destination (in sequence): dest: 27

	1	2	3	4			
r	r	B					
5	6	7	8				
		b					
	9	10	11	12			
	b						
13	14	15	16				
	17	18	19	20			
b							
21	22	23	24				
		r	R				
	25	26	27	28			
	R						
29	30	31	32				

Round 33 Black's Move...

Black moves from 11 to destination (in sequence): dest: 8

	1	2	3	4			
r	r	B	b				
5	6	7	8				
	9	10	11	12			
	b						
13	14	15	16				
	17	18	19	20			
b							
21	22	23	24				
		r	R				
	25	26	27	28			

		R					
29		30		31		32	

Round 33 Red's Move...

Red moves from 26 to destination (in sequence): dest: 31

	1		2		3		4
r		r		B		b	
5		6		7		8	
	9		10		11		12
		b					
13		14		15		16	
	17		18		19		20
b							
21		22		23		24	
				R			
	25		26		27		28
		R		R			
29		30		31		32	

Round 34 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
		b					
13		14		15		16	
	17		18		19		20
b							
21		22		23		24	
				R			
	25		26		27		28
		R		R			
29		30		31		32	

Round 34 Red's Move...

Red moves from 27 to destination (in sequence): dest: 23

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
		b					

13	14	15	16	
17	18	19	20	
b	R			
21	22	23	24	
25	26	27	28	
R	R			
29	30	31	32	

Round 35 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1	2	3	4		
r	r	B	B			
5	6	7	8			
9	10	11	12			
	b					
13	14	15	16			
17	18	19	20			
b	R					
21	22	23	24			
25	26	27	28			
R	R					
29	30	31	32			

Round 35 Red's Move...

Red moves from 23 to destination (in sequence): dest: 19

	1	2	3	4		
r	r	B	B			
5	6	7	8			
9	10	11	12			
	b					
13	14	15	16			
			R			
17	18	19	20			
b						
21	22	23	24			
25	26	27	28			
R	R					
29	30	31	32			

Round 36 Black's Move...

Black moves from 21 to destination (in sequence): dest: 17

	1	2	3	4			
r	r	B	B				
5	6	7	8				
	9	10	11	12			
	b						
13	14	15	16				
b		R					
17	18	19	20				
21	22	23	24				
	25	26	27	28			
	R	R					
29	30	31	32				

Round 36 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

	1	2	3	4			
r	r	B	B				
5	6	7	8				
	9	10	11	12			
	b	R					
13	14	15	16				
b							
17	18	19	20				
21	22	23	24				
	25	26	27	28			
	R	R					
29	30	31	32				

Round 37 Black's Move...

Black moves from 17 to destination (in sequence): dest: 13

	1	2	3	4			
r	r	B	B				
5	6	7	8				
	9	10	11	12			
b	b	R					
13	14	15	16				
	17	18	19	20			
21	22	23	24				
	25	26	27	28			

		R		R			
29		30		31		32	

Round 37 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1		2		3		4
r		r		B		B	
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 38 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 38 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b		R			

13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 39 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b	R		
13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 39 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b			
13	14	15	16	
			R	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 40 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b							
13		14		15		16			
				R					
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 40 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
				R					
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 41 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1		2		3		4		
r		r		B		B			
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
				R					
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		

		R		R			
29		30		31		32	

Round 41 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1		2		3		4
r		r		B		B	
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 42 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 42 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b		R			

13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 43 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b	R		
13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 43 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b			
13	14	15	16	
			R	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 44 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b							
13		14		15		16			
				R					
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 44 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 45 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1		2		3		4		
r		r		B		B			
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		

		R		R			
29		30		31		32	

Round 45 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1		2		3		4
r		r		B		B	
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 46 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 46 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b		R			

13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 47 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b	R		
13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 47 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b			
13	14	15	16	
			R	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 48 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

							B	
	1		2		3		4	
r		r		B				
5		6		7		8		
	9		10		11		12	
b		b						
13		14		15		16		
				R				
17		18		19		20		
21		22		23		24		
25		26		27		28		
		R		R				
29		30		31		32		

Round 48 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

							B	
	1		2		3		4	
r		r		B				
5		6		7		8		
	9		10		11		12	
b		b		R				
13		14		15		16		
17		18		19		20		
21		22		23		24		
25		26		27		28		
		R		R				
29		30		31		32		

Round 49 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1		2		3		4	
r		r		B		B		
5		6		7		8		
	9		10		11		12	
b		b		R				
13		14		15		16		
17		18		19		20		
21		22		23		24		
25		26		27		28		

		R		R			
29		30		31		32	

Round 49 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1		2		3		4
r		r		B		B	
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 50 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 50 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b		R			

13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 51 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b	R		
13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 51 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b			
13	14	15	16	
			R	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 52 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b							
13		14		15		16			
				R					
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 52 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 53 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1		2		3		4		
r		r		B		B			
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		

		R		R			
29		30		31		32	

Round 53 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1		2		3		4
r		r		B		B	
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 54 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 54 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b		R			

13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 55 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b	R		
13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 55 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b			
13	14	15	16	
			R	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 56 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b							
13		14		15		16			
				R					
17		18		19		20			
21		22		23		24			
	25		26		27		28		
		R		R					
29		30		31		32			

Round 56 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
				R					
17		18		19		20			
21		22		23		24			
	25		26		27		28		
		R		R					
29		30		31		32			

Round 57 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1		2		3		4		
r		r		B		B			
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
				R					
17		18		19		20			
21		22		23		24			
	25		26		27		28		

		R		R			
29		30		31		32	

Round 57 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1		2		3		4
r		r		B		B	
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 58 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 58 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b		R			

13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 59 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b	R		
13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 59 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b			
13	14	15	16	
			R	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 60 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b							
13		14		15		16			
				R					
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 60 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 61 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1		2		3		4		
r		r		B		B			
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		

		R		R			
29		30		31		32	

Round 61 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1		2		3		4
r		r		B		B	
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 62 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 62 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b		R			

13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 63 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b	R		
13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 63 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b			
13	14	15	16	
			R	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 64 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b							
13		14		15		16			
				R					
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 64 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 65 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1		2		3		4		
r		r		B		B			
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		

		R		R			
29		30		31		32	

Round 65 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1		2		3		4
r		r		B		B	
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 66 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 66 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b		R			

13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 67 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b	R		
13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 67 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b			
13	14	15	16	
			R	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 68 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b							
13		14		15		16			
				R					
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 68 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 69 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1		2		3		4		
r		r		B		B			
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		

		R		R			
29		30		31		32	

Round 69 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1		2		3		4
r		r		B		B	
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
				R		R	
29		30		31		32	

Round 70 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
				R		R	
29		30		31		32	

Round 70 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b		R			

13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 71 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b	R		
13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 71 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b			
13	14	15	16	
			R	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 72 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b							
13		14		15		16			
				R					
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 72 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 73 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1		2		3		4		
r		r		B		B			
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		

		R		R			
29		30		31		32	

Round 73 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1		2		3		4
r		r		B		B	
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
				R		R	
29		30		31		32	

Round 74 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
				R		R	
29		30		31		32	

Round 74 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b		R			

13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 75 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b	R		
13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 75 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b			
13	14	15	16	
			R	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 76 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b							
13		14		15		16			
				R					
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 76 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

								B	
	1		2		3		4		
r		r		B					
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
		R		R					
	29		30		31		32		

Round 77 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1		2		3		4		
r		r		B		B			
5		6		7		8			
	9		10		11		12		
b		b		R					
13		14		15		16			
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		

		R		R			
29		30		31		32	

Round 77 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1		2		3		4
r		r		B		B	
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 78 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b					
13		14		15		16	
				R			
	17		18		19		20
21		22		23		24	
	25		26		27		28
		R		R			
29		30		31		32	

Round 78 Red's Move...

Red moves from 19 to destination (in sequence): dest: 15

							B
	1		2		3		4
r		r		B			
5		6		7		8	
	9		10		11		12
b		b		R			

13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 79 Black's Move...

Black moves from 4 to destination (in sequence): dest: 8

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b	R		
13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 79 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1	2	3	4
r	r	B	B	
5	6	7	8	
9	10	11	12	
b	b			
13	14	15	16	
			R	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	R	R		
29	30	31	32	

Round 80 Black's Move...

Black moves from 8 to destination (in sequence): dest: 4

RED Leaf Nodes: 2596
RED Expanded Nodes: 3295
RED Total Nodes: 5891

BLACK Leaf Nodes: 3004
BLACK Expanded Nodes: 3675
BLACK Total Nodes: 6679

DRAW!!!
Red - (4 •_•)4 ♀(`Д' ♀) - Black
Mission FAILED...We'll get em next time!
Red player alg: Minimax-Alpha-Beta, eval: 1
Black player alg: Alpha-Beta-Search, eval: 1
Depth: 2

Game 2: (Minmax-A-B, EF-2) vs (A-B-Search, EF-2) – Depth 2

Refer to files min-2-abs-2-2-full and short.log for full output.

```
alpha: -885113 beta: 2147483647 val: -885113 move start: 11
Black moves from 11 to destination (in sequence): dest: 4
```

	B				B
	1	2	3	4	
	b	b			
	5	6	7	8	
		b		b	
	9	10	11	12	
	13	14	15	16	
				b	
	17	18	19	20	
b		b			
21	22	23	24		
			b		
25	26	27	28		
29	30	31	32		
BLACK player took 1 piece(s).					

```
RED Leaf Nodes: 253
RED Expanded Nodes: 345
RED Total Nodes: 598
```

```
BLACK Leaf Nodes: 196
BLACK Expanded Nodes: 295
BLACK Total Nodes: 491
```

```
BLACK WINS!!!
BLACK Player: «(-_-')»
But most importantly, RED looooses (boooo!)
RED Player: (J°□°) J ~ █
Red player alg: Minimax-Alpha-Beta, eval: 2
Black player alg: Alpha-Beta-Search, eval: 2
Depth: 2
```

Figure 34: Minimax-2 vs ABS-2 Depth 2

Game 3: (Minmax-A-B, EF-3) vs (A-B-Search, EF-3) – Depth 2

Refer to min-3-abs-3-2-short.log and min-3-abs-3-2-full.log files for complete output.

Round 80 Black's Move...

Black moves from 10 to destination (in sequence): dest: 7

	1	2	3	4
5	6	7	8	
9	10	11	12	
13	14	15	16	
17	18	19	20	
21	r	23	24	
r	r	R		
25	26	27	28	
29	30	31	32	

RED Leaf Nodes: 1158

RED Expanded Nodes: 1674

RED Total Nodes: 2832

BLACK Leaf Nodes: 1202

BLACK Expanded Nodes: 1533

BLACK Total Nodes: 2735

DRAW!!!

Red - (я •_•)я ѿ('Д' ѿ) - Black

Mission FAILED...We'll get em next time!

Red player alg: Minimax-Alpha-Beta, eval: 3

Black player alg: Alpha-Beta-Search, eval: 3

Depth: 2

Figure 35: Minimax-3 vs ABS-3 Depth 2

Game 4: (Minimax-A-B, EF-1) vs (Minimax-A-B, EF-2) – Depth 2

Refer to files min-1-min-2-2-full and short.log for full output.

```
Black moves from 17 to destination (in sequence): dest: 13
+---+---+---+---+---+---+
|   | 1 |   | 2 |   | 3 |   | 4 |
+---+---+---+---+---+---+
| 5 |   | 6 |   | 7 |   | 8 |   |
|   |   | B |   | B |   |   |
| 9 |   | 10|   | 11|   | 12|   |
| B |   | B |   | B |   |   |
| 13|   | 14|   | 15|   | 16|   |
|   |   | B |   | B |   |   |
| 17|   | 18|   | 19|   | 20|   |
|   |   |   |   |   |   | b |
| 21|   | 22|   | 23|   | 24|   |
|   | R |   |   |   |   |
| 25|   | 26|   | 27|   | 28|   |
|   |   |   |   |   |   | b |
| 29|   | 30|   | 31|   | 32|   |
+---+---+---+---+---+---+
RED Leaf Nodes: 1648
RED Expanded Nodes: 1968
RED Total Nodes: 3616

BLACK Leaf Nodes: 1620
BLACK Expanded Nodes: 2589
BLACK Total Nodes: 4209

DRAW!!!
Red - (я •_•)я ღ( `Д' ღ) - Black
Mission FAILED...We'll get em next time!
Red player alg: Minimax-Alpha-Beta, eval: 1
Black player alg: Minimax-Alpha-Beta, eval: 2
Depth: 2
```

Figure 36: Minimax-1 vs Minimax-2 Depth 2

Game 5: (Minmax-A-B, EF-1) vs (Minimax-A-B, EF-3) – Depth 2

Refer to files min-1-min-3-2-full and short.log for full output.

Black moves from 5 to destination (in sequence): dest: 1

	B						
	1		2		3		4
			R				
	5		6		7		8
	9		10		11		12
	13		14		15		16
		R					
	17		18		19		20
	21		22		23		24
			R		R		
	25		26		27		28
b							
29		30		31		32	

RED Leaf Nodes: 1604
RED Expanded Nodes: 2511
RED Total Nodes: 4115

BLACK Leaf Nodes: 1640
BLACK Expanded Nodes: 1950
BLACK Total Nodes: 3590

DRAW!!!
Red - (я •_•)я ѿ('Д' ѿ) - Black
Mission FAILED...We'll get em next time!
Red player alg: Minimax-Alpha-Beta, eval: 1
Black player alg: Minimax-Alpha-Beta, eval: 3
Depth: 2

Figure 37: Minimax-1 vs Minimax-3 Depth 2

Game 6: (Minmax-A-B, EF-2) vs (Minimax-A-B, EF-3) – Depth 2

Refer to files min-2-min-3-2-full and short.log for full output.

```
Round 80 Black's Move...
Black moves from 7 to destination (in sequence): dest: 2
```

	r	B			
	1	2	3	4	
b					
5	6	7	8		r
9	10	11	12		
13	14	15	16		
	R				
17	18	19	20		
21	22	23	24		
25	26	27	28		
29	30	31	32		

```
RED Leaf Nodes: 1103
RED Expanded Nodes: 1554
RED Total Nodes: 2657
```

```
BLACK Leaf Nodes: 821
BLACK Expanded Nodes: 1141
BLACK Total Nodes: 1962
```

```
DRAW!!!
Red - (я •_•)я ы( 'Д' ы) - Black
Mission FAILED...We'll get em next time!
Red player alg: Minimax-Alpha-Beta, eval: 2
Black player alg: Minimax-Alpha-Beta, eval: 3
Depth: 2
```

Figure 38: Minimax-2 vs Minimax-3 Depth 2

Game 7: (A-B-Search, EF-1) vs (A-B-Search, EF-2) – Depth 2

Refer to files abs-1-abs-2-2-full and short.log for full output.

```
alpha: 20702 beta: 2147483647 val: 20702 move start: 11
Black moves from 17 to destination (in sequence): dest: 13
```

	1	2	3	4
5	6	7	8	
	B	B		
9	10	11	12	
B	B	B		
13	14	15	16	
	B	B		
17	18	19	20	
			b	
21	22	23	24	
R				
25	26	27	28	
			b	
29	30	31	32	

```
RED Leaf Nodes: 1648
RED Expanded Nodes: 1968
RED Total Nodes: 3616
```

```
BLACK Leaf Nodes: 1620
BLACK Expanded Nodes: 2589
BLACK Total Nodes: 4209
```

```
DRAW!!!
Red - (я •_•)я ы( 'Д' ы) - Black
Mission FAILED...We'll get em next time!
Red player alg: Alpha-Beta-Search, eval: 1
Black player alg: Alpha-Beta-Search, eval: 2
Depth: 2
```

Figure 39: ABS-1 vs ABS-2 Depth 2

Game 8: (A-B-Search, EF-1) vs (A-B-Search, EF-3) – Depth 2

Refer to files abs-1-abs-3-2-full and short.log for full output.

```
Round 80 Black's Move...
Black moves from 5 to destination (in sequence): dest: 1
```

	B						
	1	2	3	4			
		R					
5	6	7	8				
	9	10	11	12			
	13	14	15	16			
	R						
	17	18	19	20			
	21	22	23	24			
		R	R				
	25	26	27	28			
b							
29	30	31	32				

```
RED Leaf Nodes: 1604
RED Expanded Nodes: 2511
RED Total Nodes: 4115
```

```
BLACK Leaf Nodes: 1640
BLACK Expanded Nodes: 1950
BLACK Total Nodes: 3590
```

```
DRAW!!!
Red - (я •_•)я ы( 'Д' ы) - Black
Mission FAILED...We'll get em next time!
Red player alg: Alpha-Beta-Search, eval: 1
Black player alg: Alpha-Beta-Search, eval: 3
Depth: 2
```

Figure 40: ABS-1 vs ABS-3 Depth 2

Game 9: (A-B-Search, EF-2) vs (A-B-Search, EF-3) – Depth 2

Refer to files abs-2-abs-3-2-full and short.log for full output.

```
BLACK In maxValue()! Depth is 0
alpha: 1010 beta: 2147483647 val: 1010 move start: 9
Black moves from 7 to destination (in sequence): dest: 2

+-----+
| r | 1 | 2 | 3 | 4 |
+-----+
| b | 5 | 6 | 7 | 8 | r |
| 9 | 10 | 11 | 12 |
+-----+
| 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 |
| 29 | 30 | 31 | 32 |
+-----+  
  
RED Leaf Nodes: 1103
RED Expanded Nodes: 1554
RED Total Nodes: 2657  
  
BLACK Leaf Nodes: 821
BLACK Expanded Nodes: 1141
BLACK Total Nodes: 1962  
  
DRAW!!!
Red - (я •`•)я ы( 'Д' ы) - Black
Mission FAILED...We'll get em next time!
Red player alg: Alpha-Beta-Search, eval: 2
Black player alg: Alpha-Beta-Search, eval: 3
Depth: 2
```

Figure 41: ABS-2 vs ABS-3 Depth 2

Game 10: (Minmax-A-B, EF-1) vs (A-B-Search, EF-1) – Depth 4

Refer to files min-1-abs-1-4-full and short.log for full output.

```
Round 80 Black's Move...
Black moves from 1 to destination (in sequence): dest: 5

|   |   |   |   |   |   |   | |
|   | 1 |   | 2 |   | 3 |   | 4 |
| B |   |   | R |   |   | b |   |
| 5 |   | 6 |   | 7 |   | 8 |   |
|   | B |   |   |   |   |   |   |
| 9 |   | 10 |   | 11 |   | 12 |   |
| 13|   | 14 |   | 15 |   | 16 |   |
|   | 17 |   | 18 |   | 19 |   | 20 |
| 21|   | 22 |   | 23 |   | 24 |   |
|   | 25 |   | 26 |   | 27 |   | 28 |
| 29|   | 30 |   | 31 |   | 32 |   |
|   |   |   |   |   |   |   | R |
|   |   |   |   |   |   |   |   |

RED Leaf Nodes: 19380
RED Expanded Nodes: 26817
RED Total Nodes: 46197

BLACK Leaf Nodes: 18947
BLACK Expanded Nodes: 27852
BLACK Total Nodes: 46799

DRAW!!!
Red - (я •_•)я ы( 'Д' ы) - Black
Mission FAILED...We'll get em next time!
Red player alg: Minimax-Alpha-Beta, eval: 1
Black player alg: Alpha-Beta-Search, eval: 1
Depth: 4
```

Figure 42: Minimax-1 vs ABS-1 Depth 4

Welcome to the Checkers AI Program.

Authors: David Torrente (dat54@txstate.edu), Randall Henderson (rrh93@txstate.edu), Borislav Sabotinov (bss64@txstate.edu).

Re-run this program with -h or -help CLI argument to see a help menu or refer to README for instructions.

NOTE: If 1 is selected below, you will NOT be prompted further for any eval function or algorithm. All will be simulated in order.
For RED player, r = MAN and R = KING; for BLACK player, b = MAN and B = KING.

Choose a game mode below:

1. Full Simulation (recommended)
 2. Single Custom Simulation
 3. Player vs Player (manual game)
 4. Player vs AI (will be asked to select AI playstyle)
- Ctrl + C to terminate program at any time.

Your choice (1, 2, 3, or 4): Please select the type of simulation you wish to run by entering in it's number.

1. Run Minimax-A-B algorithm

0. Run Alpha-Beta-Search algorithm

Algorithm for RED - Player 1 (1 for minimax, 0 for ab-Search):

Evaluation for RED - Player 1 (1 (David's), 2 (Randy's), 3 (Boris'), 4 (returns 1st available move)): Algorithm for BLACK - Player 2 (1 for minimax, 0 for ab-Search):

Evaluation for BLACK - Player 2 (1 (David's), 2 (Randy's), 3 (Boris'), 4 (returns 1st available move)): Enter the depth for the search tree (2 or 4 recommended; min = 2, max = 15): Running a SINGLE game, specific simulation!

Round 1 Black's Move...

Black moves from 21 to destination (in sequence): dest: 17

		r		r		r		r		r	
	1		2		3		4				
r		r		r		r					
5		6		7		8					
	r		r		r		r				
	9		10		11		12				
13		14		15		16					
	b										
	17		18		19		20				
		b		b		b					
21		22		23		24					
	b		b		b		b				
	25		26		27		28				
b		b		b		b					
29		30		31		32					

Round 1 Red's Move...

Red moves from 9 to destination (in sequence): dest: 14

		r		r		r		r		r	
	1		2		3		4				
r		r		r		r					
5		6		7		8					
		r		r		r					
		9		10		11		12			
13		14		15		16					

	b						
	17	18	19	20			
	b	b	b				
	21	22	23	24			
	b	b	b	b			
	25	26	27	28			
	b	b	b	b			
	29	30	31	32			

Round 2 Black's Move...

Black moves from 25 to destination (in sequence): dest: 21

	r		r	r	r	
	1	2	3	4		
r	r	r	r			
5	6	7	8			
	r	r	r			
	9	10	11	12		
	r					
	13	14	15	16		
	b					
	17	18	19	20		
b	b	b	b			
	21	22	23	24		
		b	b	b		
	25	26	27	28		
b	b	b	b			
	29	30	31	32		

Round 2 Red's Move...

Red moves from 10 to destination (in sequence): dest: 15

	r		r	r	r	
	1	2	3	4		
r	r	r	r			
5	6	7	8			
		r	r			
	9	10	11	12		
	r	r				
	13	14	15	16		
	b					
	17	18	19	20		
b	b	b	b			
	21	22	23	24		
		b	b	b		
	25	26	27	28		
b	b	b	b			
	29	30	31	32		

Round 3 Black's Move...

Black moves from 17 to destination (in sequence): dest: 10

	r		r	r	r	
	17	18	19	20		
	b	b	b	b		
	21	22	23	24		
		b	b	b		
	25	26	27	28		
b	b	b	b			
	29	30	31	32		

	1	2	3	4
r	r	r	r	
5	6	7	8	
	b	r	r	
9	10	11	12	
		r		
13	14	15	16	
17	18	19	20	
b	b	b	b	
21	22	23	24	
	b	b	b	
25	26	27	28	
b	b	b	b	
29	30	31	32	

BLACK player took 1 piece(s).

Round 3 Red's Move...

Red moves from 7 to destination (in sequence): dest: 14

	r	r	r	r
1	r	2	3	4
r	r		r	
5	6	7	8	
		r	r	
9	10	11	12	
	r	r		
13	14	15	16	
17	18	19	20	
b	b	b	b	
21	22	23	24	
	b	b	b	
25	26	27	28	
b	b	b	b	
29	30	31	32	

RED player took 1 piece(s).

Round 4 Black's Move...

Black moves from 23 to destination (in sequence): dest: 18

	r	r	r	r
1	r	2	3	4
r	r		r	
5	6	7	8	
		r	r	
9	10	11	12	
	r	r		
13	14	15	16	
	b			
17	18	19	20	
b	b		b	
21	22	23	24	
	b	b	b	

	25		26		27		28
b		b		b		b	
29		30		31		32	

Round 4 Red's Move...

Red moves from 14 to destination (in sequence): dest: 23

	r		r		r		r
	1		2		3		4
r		r			r		
5		6		7		8	
				r		r	
	9		10		11		12
				r			
13		14		15		16	
	17		18		19		20
b		b		r		b	
21		22		23		24	
			b	b		b	
	25		26		27		28
b		b		b		b	
29		30		31		32	

RED player took 1 piece(s).

Round 5 Black's Move...

Black moves from 26 to destination (in sequence): dest: 19

dest: 10

	r		r		r		r
	1		2		3		4
r		r			r		
5		6		7		8	
		b		r		r	
	9		10		11		12
13		14		15		16	
	17		18		19		20
b		b			b		
21		22		23		24	
			b		b		
	25		26		27		28
b		b		b		b	
29		30		31		32	

BLACK player took 2 piece(s).

Round 5 Red's Move...

Red moves from 6 to destination (in sequence): dest: 15

	r		r		r		r
	1		2		3		4
r				r			

5		6		7		8	
				r		r	
	9		10		11		12
			r				
	13		14		15		16
	17		18		19		20
b		b			b		
21		22		23		24	
				b		b	
	25		26		27		28
b		b		b		b	
29		30		31		32	

RED player took 1 piece(s).

Round 6 Black's Move...

Black moves from 24 to destination (in sequence): dest: 19

		r		r		r		r
	1		2		3		4	
r					r			
5		6		7		8		
				r		r		
	9		10		11		12	
			r					
	13		14		15		16	
					b			
	17		18		19		20	
b		b						
21		22		23		24		
				b		b		
	25		26		27		28	
b		b		b		b		
29		30		31		32		

Round 6 Red's Move...

Red moves from 15 to destination (in sequence): dest: 24

		r		r		r		r
	1		2		3		4	
r					r			
5		6		7		8		
				r		r		
	9		10		11		12	
	13		14		15		16	
	17		18		19		20	
b		b			r			
21		22		23		24		
				b		b		
	25		26		27		28	
b		b		b		b		
29		30		31		32		

RED player took 1 piece(s).

Round 7 Black's Move...

Black moves from 28 to destination (in sequence): dest: 19

		r		r		r		r		r	
	1		2		3		4				
r						r					
5		6		7		8					
					r		r				
	9		10		11		12				
13		14		15		16					
					b						
17		18		19		20					
b		b									
21		22		23		24					
					b						
25		26		27		28					
b		b		b		b					
29		30		31		32					

BLACK player took 1 piece(s).

Round 7 Red's Move...

Red moves from 1 to destination (in sequence): dest: 6

			r		r		r		r		
	1		2		3		4				
r		r			r						
5		6		7		8					
					r		r				
	9		10		11		12				
13		14		15		16					
					b						
17		18		19		20					
b		b									
21		22		23		24					
					b						
25		26		27		28					
b		b		b		b					
29		30		31		32					

Round 8 Black's Move...

Black moves from 19 to destination (in sequence): dest: 15

			r		r		r		r		
	1		2		3		4				
r		r			r						
5		6		7		8					
					r		r				
	9		10		11		12				
				b							

13		14		15		16	
	17		18		19		20
b		b					
21		22		23		24	
				b			
	25		26		27		28
b		b		b		b	
29		30		31		32	

Round 8 Red's Move...

Red moves from 11 to destination (in sequence): dest: 18
dest: 25

			r		r		r
	1		2		3		4
r		r			r		
	5		6		7		8
						r	
	9		10		11		12
	13		14		15		16
	17		18		19		20
b							
	21		22		23		24
	r			b			
	25		26		27		28
b		b		b		b	
29		30		31		32	

RED player took 2 piece(s).

Round 9 Black's Move...

Black moves from 29 to destination (in sequence): dest: 22

			r		r		r
	1		2		3		4
r		r			r		
	5		6		7		8
						r	
	9		10		11		12
	13		14		15		16
	17		18		19		20
b		b					
	21		22		23		24
				b			
	25		26		27		28
	b		b		b		b
29		30		31		32	

BLACK player took 1 piece(s).

Round 9 Red's Move...

Red moves from 6 to destination (in sequence): dest: 10

			r	r	r
	1	2	3	4	
r			r		
5	6	7	8		
	r		r		
9	10	11	12		
13	14	15	16		
17	18	19	20		
b	b				
21	22	23	24		
		b			
25	26	27	28		
b	b	b	b		
29	30	31	32		

Round 10 Black's Move...

Black moves from 21 to destination (in sequence): dest: 17

			r	r	r
	1	2	3	4	
r			r		
5	6	7	8		
	r		r		
9	10	11	12		
13	14	15	16		
b					
17	18	19	20		
b					
21	22	23	24		
		b			
25	26	27	28		
b	b	b	b		
29	30	31	32		

Round 10 Red's Move...

Red moves from 10 to destination (in sequence): dest: 14

			r	r	r
	1	2	3	4	
r			r		
5	6	7	8		
			r		
9	10	11	12		
r					
13	14	15	16		
b					
17	18	19	20		
b					

21		22		23		24	
				b			
	25		26		27		28
	b		b		b		

Round 11 Black's Move...

Black moves from 17 to destination (in sequence): dest: 10

			r		r		r
	1		2		3		4
r					r		
5		6		7		8	
	b				r		
9		10		11		12	
13		14		15		16	
17		18		19		20	
	b						
21		22		23		24	
				b			
25		26		27		28	
	b		b		b		
29		30		31		32	

BLACK player took 1 piece(s).

Round 11 Red's Move...

Red moves from 12 to destination (in sequence): dest: 16

			r		r		r
	1		2		3		4
r					r		
5		6		7		8	
	b				r		
9		10		11		12	
					r		
13		14		15		16	
17		18		19		20	
	b						
21		22		23		24	
				b			
25		26		27		28	
	b		b		b		
29		30		31		32	

Round 12 Black's Move...

Black moves from 27 to destination (in sequence): dest: 23

			r		r		r
	1		2		3		4
r					r		

5		6		7		8	
		b					
	9		10		11		12
				r			
	13		14		15		16
	17		18		19		20
		b	b				
	21		22		23		24
	25		26		27		28
		b	b		b		
	29		30		31		32

Round 12 Red's Move...

Red moves from 8 to destination (in sequence): dest: 11

			r		r		r
	1		2		3		4
r							
	5		6		7		8
		b	r				
	9		10		11		12
				r			
	13		14		15		16
	17		18		19		20
		b	b				
	21		22		23		24
	25		26		27		28
		b	b		b		
	29		30		31		32

Round 13 Black's Move...

Black moves from 22 to destination (in sequence): dest: 18

			r		r		r
	1		2		3		4
r							
	5		6		7		8
		b	r				
	9		10		11		12
				r			
	13		14		15		16
		b					
	17		18		19		20
		b					
	21		22		23		24
	25		26		27		28
		b	b		b		
	29		30		31		32

Round 13 Red's Move...

Red moves from 16 to destination (in sequence): dest: 20

			r	r	r	
	1	2	3	4		
r						
5	6	7	8			
	b	r				
9	10	11	12			
13	14	15	16			
	b		r			
17	18	19	20			
	b					
21	22	23	24			
25	26	27	28			
	b	b	b			
29	30	31	32			

Round 14 Black's Move...

Black moves from 31 to destination (in sequence): dest: 27

			r	r	r	
	1	2	3	4		
r						
5	6	7	8			
	b	r				
9	10	11	12			
13	14	15	16			
	b		r			
17	18	19	20			
	b					
21	22	23	24			
			b			
25	26	27	28			
	b		b			
29	30	31	32			

Round 14 Red's Move...

Red moves from 11 to destination (in sequence): dest: 16

			r	r	r	
	1	2	3	4		
r						
5	6	7	8			
	b					
9	10	11	12			
13	14	15	16			
	b		r			
17	18	19	20			

				b			
21	—	22	—	23	—	24	—
				b			
—	25	—	26	—	27	—	28
		b			b		
29	—	30	—	31	—	32	—

Round 15 Black's Move...

Black moves from 10 to destination (in sequence): dest: 6

			r		r		r
1	—	2	—	3	—	4	—
r		b					
5	—	6	—	7	—	8	—
9	—	10	—	11	—	12	—
				r			
13	—	14	—	15	—	16	—
		b			r		
17	—	18	—	19	—	20	—
		b					
21	—	22	—	23	—	24	—
			b				
25	—	26	—	27	—	28	—
		b			b		
29	—	30	—	31	—	32	—

Round 15 Red's Move...

Red moves from 2 to destination (in sequence): dest: 9

				r		r	
1	—	2	—	3	—	4	—
r							
5	—	6	—	7	—	8	—
9	—	10	—	11	—	12	—
				r			
13	—	14	—	15	—	16	—
		b			r		
17	—	18	—	19	—	20	—
		b					
21	—	22	—	23	—	24	—
			b				
25	—	26	—	27	—	28	—
		b			b		
29	—	30	—	31	—	32	—

RED player took 1 piece(s).

Round 16 Black's Move...

Black moves from 18 to destination (in sequence): dest: 14

			r		r	
1	—	2	—	3	—	4

	r								
5		6		7		8			
	r								
	9		10		11		12		
		b			r				
13		14		15		16			
						r			
	17		18		19		20		
		b							
21		22		23		24			
			b						
	25		26		27		28		
		b			b				
29		30		31		32			

Round 16 Red's Move...

Red moves from 9 to destination (in sequence): dest: 18

					r		r		
	1		2		3		4		
r									
5		6		7		8			
	9		10		11		12		
						r			
13		14		15		16			
			r			r			
	17		18		19		20		
		b							
21		22		23		24			
			b						
	25		26		27		28		
		b			b				
29		30		31		32			

RED player took 1 piece(s).

Round 17 Black's Move...

Black moves from 23 to destination (in sequence): dest: 14

					r		r		
	1		2		3		4		
r									
5		6		7		8			
	9		10		11		12		
		b			r				
13		14		15		16			
			b			r			
	17		18		19		20		
21		22		23		24			
			b						
	25		26		27		28		
		b			b				
29		30		31		32			

|_29|____|_30|____|_31|____|_32|____|
BLACK player took 1 piece(s).

Round 17 Red's Move...

Red moves from 5 to destination (in sequence): dest: 9

						r		r
	1		2		3		4	
	5		6		7		8	
	r							
	9		10		11		12	
		b				r		
	13		14		15		16	
						r		
	17		18		19		20	
	21		22		23		24	
					b			
	25		26		27		28	
		b			b			
	29		30		31		32	

Round 18 Black's Move...

Black moves from 14 to destination (in sequence): dest: 5

						r		r
	1		2		3		4	
	b							
	5		6		7		8	
	9		10		11		12	
					r			
	13		14		15		16	
						r		
	17		18		19		20	
	21		22		23		24	
					b			
	25		26		27		28	
		b			b			
	29		30		31		32	

BLACK player took 1 piece(s).

Round 18 Red's Move...

Red moves from 3 to destination (in sequence): dest: 7

							r	
	1		2		3		4	
	b			r				
	5		6		7		8	
	9		10		11		12	

							r		
13		14		15		16			
							r		
	17		18		19		20		
21		22		23		24			
						b			
	25		26		27		28		
			b			b			
29		30		31		32			

Round 19 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

	B						r		
	1		2		3		4		
				r					
5		6		7		8			
	9		10		11		12		
						r			
13		14		15		16			
						r			
17		18		19		20			
21		22		23		24			
						b			
	25		26		27		28		
			b			b			
29		30		31		32			

Round 19 Red's Move...

Red moves from 4 to destination (in sequence): dest: 8

	B								
	1		2		3		4		
				r		r			
5		6		7		8			
	9		10		11		12		
						r			
13		14		15		16			
						r			
17		18		19		20			
21		22		23		24			
						b			
	25		26		27		28		
			b			b			
29		30		31		32			

Round 20 Black's Move...

Black moves from 27 to destination (in sequence): dest: 23

		B						
	1		2		3		4	
				r		r		
5		6		7		8		
	9		10		11		12	
						r		
13		14		15		16		
							r	
	17		18		19		20	
				b				
21		22		23		24		
	25		26		27		28	
		b			b			
29		30		31		32		

Round 20 Red's Move...

Red moves from 20 to destination (in sequence): dest: 24

		B						
	1		2		3		4	
				r		r		
5		6		7		8		
	9		10		11		12	
						r		
13		14		15		16		
							r	
	17		18		19		20	
				b		r		
21		22		23		24		
	25		26		27		28	
		b			b			
29		30		31		32		

Round 21 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4	
B				r		r		
5		6		7		8		
	9		10		11		12	
						r		
13		14		15		16		
							r	
	17		18		19		20	
				b		r		
21		22		23		24		

		25		26		27		28	
			b				b		
	29		30		31		32		

Round 21 Red's Move...

Red moves from 24 to destination (in sequence): dest: 27

		1		2		3		4	
	B				r		r		
	5		6		7		8		
		9		10		11		12	
							r		
	13		14		15		16		
		17		18		19		20	
					b				
	21		22		23		24		
						r			
		25		26		27		28	
			b			b			
	29		30		31		32		

Round 22 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

		B							
		1		2		3		4	
				r		r			
		5		6		7		8	
		9		10		11		12	
						r			
	13		14		15		16		
		17		18		19		20	
					b				
	21		22		23		24		
						r			
		25		26		27		28	
			b			b			
	29		30		31		32		

Round 22 Red's Move...

Red moves from 7 to destination (in sequence): dest: 11

		B							
		1		2		3		4	
					r				
		5		6		7		8	
						r			
		9		10		11		12	

							r		
13		14		15		16			
	17		18		19		20		
				b					
21		22		23		24			
					r				
	25		26		27		28		
			b			b			
29		30		31		32			

Round 23 Black's Move...

Black moves from 23 to destination (in sequence): dest: 18

	B								
	1		2		3		4		
					r				
5		6		7		8			
					r				
	9		10		11		12		
					r				
13		14		15		16			
			b						
	17		18		19		20		
21		22		23		24			
					r				
	25		26		27		28		
			b			b			
29		30		31		32			

Round 23 Red's Move...

Red moves from 8 to destination (in sequence): dest: 12

	B								
	1		2		3		4		
5		6		7		8			
					r		r		
	9		10		11		12		
					r				
13		14		15		16			
			b						
	17		18		19		20		
21		22		23		24			
					r				
	25		26		27		28		
			b			b			
29		30		31		32			

Round 24 Black's Move...

Black moves from 32 to destination (in sequence): dest: 23

	B						
	1	2	3	4			
	5	6	7	8			
	9	10	11	12			
	13	14	15	16			
		b					
	17	18	19	20			
	21	22	23	24			
	25	26	27	28			
	29	30	31	32			

BLACK player took 1 piece(s).

Round 24 Red's Move...

Red moves from 16 to destination (in sequence): dest: 20

	B						
	1	2	3	4			
	5	6	7	8			
	9	10	11	12			
	13	14	15	16			
		b		r			
	17	18	19	20			
	21	22	23	24			
	25	26	27	28			
	29	30	31	32			

Round 25 Black's Move...

Black moves from 18 to destination (in sequence): dest: 14

	B						
	1	2	3	4			
	5	6	7	8			
	9	10	11	12			
	b						
	13	14	15	16			
				r			
	17	18	19	20			
	21	22	23	24			

	25	26		27		28	
	b						
	29	30	31	32			

Round 25 Red's Move...

Red moves from 11 to destination (in sequence): dest: 15

	B						
	1	2	3	4			
	5	6	7	8			
				r			
	9	10	11	12			
	b	r					
	13	14	15	16			
				r			
	17	18	19	20			
		b					
	21	22	23	24			
	25	26	27	28			
	b						
	29	30	31	32			

Round 26 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1	2	3	4			
B							
	5	6	7	8			
				r			
	9	10	11	12			
	b	r					
	13	14	15	16			
				r			
	17	18	19	20			
		b					
	21	22	23	24			
	25	26	27	28			
	b						
	29	30	31	32			

Round 26 Red's Move...

Red moves from 15 to destination (in sequence): dest: 19

	1	2	3	4			
B							
	5	6	7	8			
				r			

	9	10	11	12
	b			
13	14	15	16	
		r		r
17	18	19	20	
		b		
21	22	23	24	
25	26	27	28	
	b			
29	30	31	32	

Round 27 Black's Move...

Black moves from 23 to destination (in sequence) : dest: 16

	1	2	3	4
B				
5	6	7	8	
			r	
9	10	11	12	
	b	b		
13	14	15	16	
			r	
17	18	19	20	
21	22	23	24	
25	26	27	28	
	b			
29	30	31	32	

BLACK player took 1 piece(s).

Round 27 Red's Move...

Red moves from 12 to destination (in sequence): dest: 19

	1	2	3	4
B				
5	6	7	8	
9	10	11	12	
	b			
13	14	15	16	
			r	r
17	18	19	20	
21	22	23	24	
25	26	27	28	
	b			
29	30	31	32	

RED player took 1 piece(s).

Round 28 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

	B							
	1	2	3	4				
	5	6	7	8				
	9	10	11	12				
	b							
	13	14	15	16				
			r	r				
	17	18	19	20				
	21	22	23	24				
	25	26	27	28				
	b							
	29	30	31	32				

Round 28 Red's Move...

Red moves from 19 to destination (in sequence): dest: 23

	B							
	1	2	3	4				
	5	6	7	8				
	9	10	11	12				
	b							
	13	14	15	16				
			r					
	17	18	19	20				
		r						
	21	22	23	24				
	25	26	27	28				
	b							
	29	30	31	32				

Round 29 Black's Move...

Black moves from 1 to destination (in sequence): dest: 6

	1	2	3	4				
	B							
	5	6	7	8				
	9	10	11	12				
	b							
	13	14	15	16				
			r					
	17	18	19	20				

				r			
21		22		23		24	
	25		26		27		28

Round 29 Red's Move...

Red moves from 23 to destination (in sequence): dest: 27

				1		2		3		4
				B						
	5		6		7		8			
	9		10		11		12			
			b							
	13		14		15		16			
							r			
	17		18		19		20			
	21		22		23		24			
						r				
	25		26		27		28			
			b							
	29		30		31		32			

Round 30 Black's Move...

Black moves from 30 to destination (in sequence): dest: 26

				1		2		3		4
				B						
	5		6		7		8			
	9		10		11		12			
			b							
	13		14		15		16			
							r			
	17		18		19		20			
	21		22		23		24			
			b		r					
	25		26		27		28			
	29		30		31		32			

Round 30 Red's Move...

Red moves from 20 to destination (in sequence): dest: 24

				1		2		3		4
				B						

5		6		7		8	
	9		10		11		12
		b					
	13		14		15		16
	17		18		19		20
					r		
	21		22		23		24
			b		r		
	25		26		27		28
	29		30		31		32

Round 31 Black's Move...

Black moves from 6 to destination (in sequence): dest: 1

	B						
	1		2		3		4
	5		6		7		8
	9		10		11		12
		b					
	13		14		15		16
	17		18		19		20
					r		
	21		22		23		24
			b		r		
	25		26		27		28
	29		30		31		32

Round 31 Red's Move...

Red moves from 24 to destination (in sequence): dest: 28

	B						
	1		2		3		4
	5		6		7		8
	9		10		11		12
		b					
	13		14		15		16
	17		18		19		20
					r		
	21		22		23		24
			b		r		r
	25		26		27		28
	29		30		31		32

Round 32 Black's Move...

Black moves from 1 to destination (in sequence): dest: 6

	1	B	2		3		4		
5		6		7		8			
	9		10		11		12		
		b							
	13		14		15		16		
	17		18		19		20		
	21		22		23		24		
			b		r		r		
	25		26		27		28		
	29		30		31		32		

Round 32 Red's Move...

Red moves from 27 to destination (in sequence): dest: 31

	1	B	2		3		4		
5		6		7		8			
	9		10		11		12		
		b							
	13		14		15		16		
	17		18		19		20		
	21		22		23		24		
			b			r			
	25		26		27		28		
				R					
	29		30		31		32		

Round 33 Black's Move...

Black moves from 26 to destination (in sequence): dest: 22

	1	B	2		3		4		
5		6		7		8			
	9		10		11		12		
		b							
	13		14		15		16		
	17		18		19		20		

		b						
21	—	22	—	23	—	24	—	
							r	
—	25	—	26	—	27	—	28	
				R				
29	—	30	—	31	—	32	—	

Round 33 Red's Move...

Red moves from 31 to destination (in sequence): dest: 26

		1		2		3		4
		B						
5	—	6	—	7	—	8	—	
9	—	10	—	11	—	12	—	
		b						
13	—	14	—	15	—	16	—	
17	—	18	—	19	—	20	—	
		b						
21	—	22	—	23	—	24	—	
			R				r	
—	25	—	26	—	27	—	28	
29	—	30	—	31	—	32	—	

Round 34 Black's Move...

Black moves from 22 to destination (in sequence): dest: 18

		1		2		3		4
		B						
5	—	6	—	7	—	8	—	
9	—	10	—	11	—	12	—	
		b						
13	—	14	—	15	—	16	—	
			b					
17	—	18	—	19	—	20	—	
21	—	22	—	23	—	24	—	
			R				r	
—	25	—	26	—	27	—	28	
29	—	30	—	31	—	32	—	

Round 34 Red's Move...

Red moves from 28 to destination (in sequence): dest: 32

		1		2		3		4
		B						

5		6		7		8	
	9		10		11		12
		b					
	13		14		15		16
			b				
	17		18		19		20
	21		22		23		24
			R				
	25		26		27		28
					R		
	29		30		31		32

Round 35 Black's Move...

Black moves from 18 to destination (in sequence): dest: 15

		1		2		3	
			B				
	5		6		7		8
	9		10		11		12
		b		b			
	13		14		15		16
	17		18		19		20
	21		22		23		24
			R				
	25		26		27		28
					R		
	29		30		31		32

Round 35 Red's Move...

Red moves from 26 to destination (in sequence): dest: 22

		1		2		3	
			B				
	5		6		7		8
	9		10		11		12
		b		b			
	13		14		15		16
	17		18		19		20
			R				
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 36 Black's Move...

Black moves from 6 to destination (in sequence): dest: 9

	1		2		3		4		
	5		6		7		8		
		B							
	9		10		11		12		
	13		14		15		16		
	17		18		19		20		
		R							
	21		22		23		24		
	25		26		27		28		
						R			
	29		30		31		32		

Round 36 Red's Move...

Red moves from 22 to destination (in sequence): dest: 17

	1		2		3		4		
	5		6		7		8		
		B							
	9		10		11		12		
	13		14		15		16		
		R							
	17		18		19		20		
	21		22		23		24		
	25		26		27		28		
						R			
	29		30		31		32		

Round 37 Black's Move...

Black moves from 14 to destination (in sequence): dest: 10

	1		2		3		4		
	5		6		7		8		
		B		b					
	9		10		11		12		
	13		14		15		16		
		R							
	17		18		19		20		

21		22		23		24	
	25		26		27		28
					R		
	29		30		31		32

Round 37 Red's Move...

Red moves from 17 to destination (in sequence): dest: 22

		1		2		3		4
	5		6		7		8	
		B		b				
		9		10		11		12
				b				
	13		14		15		16	
		17		18		19		20
	21		22		23		24	
		25		26		27		28
						R		
	29		30		31		32	

Round 38 Black's Move...

Black moves from 10 to destination (in sequence): dest: 6

		1		2		3		4
			b					
	5		6		7		8	
		B						
		9		10		11		12
				b				
	13		14		15		16	
		17		18		19		20
	21		22		23		24	
		25		26		27		28
						R		
	29		30		31		32	

Round 38 Red's Move...

Red moves from 22 to destination (in sequence): dest: 18

		1		2		3		4
			b					

5	6	7	8	
	B			
9	10	11	12	
		b		
13	14	15	16	
		R		
17	18	19	20	
21	22	23	24	
25	26	27	28	
29	30	31	32	

Round 39 Black's Move...

Black moves from 15 to destination (in sequence): dest: 11

1	2	3	4	
	b			
5	6	7	8	
	B		b	
9	10	11	12	
13	14	15	16	
	R			
17	18	19	20	
21	22	23	24	
25	26	27	28	
29	30	31	32	

Round 39 Red's Move...

Red moves from 18 to destination (in sequence): dest: 15

1	2	3	4	
	b			
5	6	7	8	
	B		b	
9	10	11	12	
13	14	15	16	
	R			
17	18	19	20	
21	22	23	24	
25	26	27	28	
29	30	31	32	

Round 40 Black's Move...

Black moves from 11 to destination (in sequence): dest: 8

	1		2		3		4
		b				b	
5		6		7		8	
	B						
	9		10		11		12
				R			
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 40 Red's Move...

Red moves from 15 to destination (in sequence): dest: 10

	1		2		3		4
	b				b		
5		6		7		8	
B		R					
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 41 Black's Move....

Black moves from 6 to destination (in sequence): dest: 1

	B							
	1		2		3		4	
	5		6		7		8	
	B		R					
	9		10		11		12	
	13		14		15		16	
	17		18		19		20	

21		22		23		24	
	25		26		27		28
					R		
	29		30		31		32

Round 41 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
			R		b		
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 42 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
	B			R		b	
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 42 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4
	B				b		

5		6		7		8	
	B		R				
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 43 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

	B						
	1		2		3		4
					b		
5		6		7		8	
	B		R				
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 43 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
				R		b	
5		6		7		8	
	B						
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 44 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
B			R		b		
5	6		7		8		
B							
9	10		11		12		
13	14		15		16		
17	18		19		20		
21	22		23		24		
25	26		27		28		
						R	
29	30		31		32		

Round 44 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

1	2	3	4					
B			b					
5	6	7	8					
B	R							
9	10	11	12					
13	14	15	16					
17	18	19	20					
21	22	23	24					
25	26	27	28					
			R					
29	30	31	32					

Round 45 Black's Move....

Black moves from 5 to destination (in sequence): dest: 1

	B							
	1		2		3		4	
						b		
	5		6		7		8	
	B		R					
	9		10		11		12	
	13		14		15		16	
	17		18		19		20	

21		22		23		24	
	25		26		27		28
					R		
	29		30		31		32

Round 45 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
			R		b		
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 46 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
	B			R		b	
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 46 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4
	B				b		

5		6		7		8	
	B		R				
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 47 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

	B						
	1		2		3		4
					b		
5		6		7		8	
	B		R				
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 47 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
				R		b	
5		6		7		8	
	B						
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 48 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

Round 48 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4
B					b		
5		6		7		8	
	B		R				
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 49 Black's Move....

Black moves from 5 to destination (in sequence): dest: 1

	B							
	1		2		3		4	
						b		
	5		6		7		8	
	B		R					
	9		10		11		12	
	13		14		15		16	
	17		18		19		20	

21		22		23		24	
	25		26		27		28
					R		
	29		30		31		32

Round 49 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
			R		b		
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 50 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
	B			R		b	
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 50 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4
	B				b		

5		6		7		8	
	B		R				
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 51 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

	B						
	1		2		3		4
					b		
5		6		7		8	
	B		R				
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 51 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
				R		b	
5		6		7		8	
	B						
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 52 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
B			R		b		
5	6		7		8		
B							
9	10		11		12		
13	14		15		16		
17	18		19		20		
21	22		23		24		
25	26		27		28		
29	30		31		32		R

Round 52 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4
B					b		
5	6		7		8		
B		R					
	9	10		11		12	
13	14		15		16		
	17	18		19		20	
21	22		23		24		
	25	26		27		28	
					R		
29	30		31		32		

Round 53 Black's Move....

Black moves from 5 to destination (in sequence): dest: 1

	B							
	1		2		3		4	
						b		
	5		6		7		8	
	B		R					
	9		10		11		12	
	13		14		15		16	
	17		18		19		20	

21		22		23		24	
	25		26		27		28
					R		
	29		30		31		32

Round 53 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
			R		b		
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 54 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
	B			R		b	
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 54 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4
	B				b		

5		6		7		8	
	B		R				
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 55 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

	B						
	1		2		3		4
					b		
5		6		7		8	
	B		R				
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 55 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
				R		b	
5		6		7		8	
	B						
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 56 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

Round 56 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4	
B					b			
5		6		7		8		
	B		R					
	9		10		11		12	
13		14		15		16		
	17		18		19		20	
21		22		23		24		
	25		26		27		28	
						R		
29		30		31		32		

Round 57 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

	B							
	1		2		3		4	
						b		
	5		6		7		8	
	B		R					
	9		10		11		12	
	13		14		15		16	
	17		18		19		20	

21		22		23		24	
	25		26		27		28
					R		
	29		30		31		32

Round 57 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
			R		b		
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 58 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
	B			R		b	
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 58 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4
	B				b		

5		6		7		8	
	B		R				
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 59 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

	B						
	1		2		3		4
					b		
5		6		7		8	
	B		R				
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 59 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
				R		b	
5		6		7		8	
	B						
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 60 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

Round 60 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4	
B					b			
5		6		7		8		
B		R						
	9		10		11		12	
13		14		15		16		
	17		18		19		20	
21		22		23		24		
	25		26		27		28	
						R		
29		30		31		32		

Round 61 Black's Move....

Black moves from 5 to destination (in sequence): dest: 1

	B							
	1		2		3		4	
						b		
	5		6		7		8	
	B		R					
	9		10		11		12	
	13		14		15		16	
	17		18		19		20	

21		22		23		24	
	25		26		27		28
					R		
	29		30		31		32

Round 61 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
			R		b		
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 62 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
	B			R		b	
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 62 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4
	B				b		

5	6	7	8	
B	R			
9	10	11	12	
13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	R
29	30	31	32	

Round 63 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

	B						
1		2	3		4		
				b			
5	6	7	8				
B	R						
9	10	11	12				
13	14	15	16				
17	18	19	20				
21	22	23	24				
25	26	27	28				
29	30	31	32				

Round 63 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
1		2	3		4		
				R	b		
5	6	7	8				
B							
9	10	11	12				
13	14	15	16				
17	18	19	20				
21	22	23	24				
25	26	27	28				
29	30	31	32				

Round 64 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

Round 64 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4	
B					b			
5		6		7		8		
B		R						
	9		10		11		12	
13		14		15		16		
	17		18		19		20	
21		22		23		24		
	25		26		27		28	
						R		
29		30		31		32		

Round 65 Black's Move....

Black moves from 5 to destination (in sequence): dest: 1

	B							
	1		2		3		4	
							b	
	5		6		7		8	
	B		R					
	9		10		11		12	
	13		14		15		16	
	17		18		19		20	

21		22		23		24	
	25		26		27		28
					R		
	29		30		31		32

Round 65 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
			R		b		
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 66 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
	B			R		b	
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 66 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4
	B				b		

5		6		7		8	
	B		R				
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 67 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

	B						
	1		2		3		4
					b		
5		6		7		8	
	B		R				
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 67 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
				R		b	
5		6		7		8	
	B						
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 68 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
B			R		b		
5	6		7		8		
B							
9	10		11		12		
13	14		15		16		
17	18		19		20		
21	22		23		24		
25	26		27		28		
29	30		31		32		R

Round 68 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4
B					b		
5	6		7		8		
B		R					
	9	10		11		12	
13	14		15		16		
	17	18		19		20	
21	22		23		24		
	25	26		27		28	
					R		
29	30		31		32		

Round 69 Black's Move....

Black moves from 5 to destination (in sequence): dest: 1

	B							
	1		2		3		4	
						b		
	5		6		7		8	
	B		R					
	9		10		11		12	
	13		14		15		16	
	17		18		19		20	

21		22		23		24	
	25		26		27		28
					R		
	29		30		31		32

Round 69 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
			R		b		
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 70 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
	B			R		b	
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 70 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4
	B				b		

5	6	7	8	
B	R			
9	10	11	12	
13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	R
29	30	31	32	

Round 71 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

	B						
1	2	3	4				
			b				
5	6	7	8				
B	R						
9	10	11	12				
13	14	15	16				
17	18	19	20				
21	22	23	24				
25	26	27	28				
29	30	31	32				

Round 71 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
1	2	3	4				
			R	b			
5	6	7	8				
B							
9	10	11	12				
13	14	15	16				
17	18	19	20				
21	22	23	24				
25	26	27	28				
29	30	31	32				

Round 72 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
B			R		b		
5	6		7		8		
B							
9	10		11		12		
13	14		15		16		
17	18		19		20		
21	22		23		24		
25	26		27		28		
29	30		31		32		

Round 72 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4	
B					b			
5		6		7		8		
	B		R					
	9		10		11		12	
13		14		15		16		
	17		18		19		20	
21		22		23		24		
	25		26		27		28	
						R		
29		30		31		32		

Round 73 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

	B							
	1		2		3		4	
						b		
	5		6		7		8	
	B		R					
	9		10		11		12	
	13		14		15		16	
	17		18		19		20	

21		22		23		24	
	25		26		27		28
					R		
	29		30		31		32

Round 73 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
			R		b		
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 74 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
	B			R		b	
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 74 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4
	B				b		

5	6	7	8	
B	R			
9	10	11	12	
13	14	15	16	
17	18	19	20	
21	22	23	24	
25	26	27	28	R
29	30	31	32	

Round 75 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

	B						
1	2	3	4				
			b				
5	6	7	8				
B	R						
9	10	11	12				
13	14	15	16				
17	18	19	20				
21	22	23	24				
25	26	27	28				
29	30	31	32				

Round 75 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
1	2	3	4				
			R	b			
5	6	7	8				
B							
9	10	11	12				
13	14	15	16				
17	18	19	20				
21	22	23	24				
25	26	27	28				
29	30	31	32				

Round 76 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
B			R		b		
5	6		7		8		
B							
	9	10		11		12	
13	14		15		16		
	17	18		19		20	
21	22		23		24		
	25	26		27		28	
					R		
29	30		31		32		

Round 76 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4	
B						b		
5		6		7		8		
	B		R					
	9		10		11		12	
13		14		15		16		
	17		18		19		20	
21		22		23		24		
	25		26		27		28	
						R		
29		30		31		32		

Round 77 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

	B							
	1		2		3		4	
						b		
	5		6		7		8	
	B		R					
	9		10		11		12	
	13		14		15		16	
	17		18		19		20	

21		22		23		24	
	25		26		27		28
					R		
	29		30		31		32

Round 77 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
			R		b		
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 78 Black's Move...

Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
	B			R		b	
	5		6		7		8
		B					
	9		10		11		12
	13		14		15		16
	17		18		19		20
	21		22		23		24
	25		26		27		28
					R		
	29		30		31		32

Round 78 Red's Move...

Red moves from 7 to destination (in sequence): dest: 10

	1		2		3		4
	B				b		

5		6		7		8	
	B		R				
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 79 Black's Move...

Black moves from 5 to destination (in sequence): dest: 1

	B						
	1		2		3		4
					b		
5		6		7		8	
	B		R				
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 79 Red's Move...

Red moves from 10 to destination (in sequence): dest: 7

	B						
	1		2		3		4
				R		b	
5		6		7		8	
	B						
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

Round 80 Black's Move...
Black moves from 1 to destination (in sequence): dest: 5

	1		2		3		4
B				R		b	
5		6		7		8	
B							
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
						R	
29		30		31		32	

RED Leaf Nodes: 19380
RED Expanded Nodes: 26817
RED Total Nodes: 46197

BLACK Leaf Nodes: 18947
BLACK Expanded Nodes: 27852
BLACK Total Nodes: 46799

```
DRAW!!!
Red - (• _ •) ( ) - Black
Mission FAILED...We'll get em next time!
Red player alg: Minimax-Alpha-Beta, eval: 1
Black player alg: Alpha-Beta-Search, eval: 1
Depth: 4
```

Game 11: (Minmax-A-B, EF-2) vs (A-B-Search, EF-2) – Depth 4

Refer to files min-2-abs-2-4-full and short.log for full output.

```
BLACK Evaluated Move: moveScore-> -4365
alpha: -755 beta: 2147483647 val: -759 move start: -1485250512
alpha: -755 beta: 2147483647 val: -755 move start: 7
Black moves from 14 to destination (in sequence): dest: 10
```

	r						
	1	2	3	4			
	5	6	7	8			
	B	B					
	9	10	11	12			
r							
13	14	15	16				
			R				
	17	18	19	20			
b	R	R					
21	22	23	24				
	25	26	27	28			
	29	30	31	32			

```
RED Leaf Nodes: 28641
RED Expanded Nodes: 44847
RED Total Nodes: 73488
```

```
BLACK Leaf Nodes: 29419
BLACK Expanded Nodes: 39043
BLACK Total Nodes: 68462
```

```
DRAW!!!
Red - (я •_•)я ы( `Д' ы) - Black
Mission FAILED...We'll get em next time!
Red player alg: Minimax-Alpha-Beta, eval: 2
Black player alg: Alpha-Beta-Search, eval: 2
Depth: 4
```

Figure 43: Minimax-2 vs ABS-2 Depth 4

Game 12: (Minmax-A-B, EF-3) vs (A-B-Search, EF-3) – Depth 4

Refer to files min-3-abs-3-4-full and short.log for full output.

Round 22 Red's Move...
Red moves from 5 to destination (in sequence): dest: 14

			r			
1		2		3		4
5		6		7		8
					r	
9		10		11		12
		r			r	
13		14		15		16
17		18		19		20
			r			
21		22		23		24
					r	
25		26		27		28
29		30		31		32

RED player took 1 piece(s).

RED Leaf Nodes: 3052
RED Expanded Nodes: 4814
RED Total Nodes: 7866

BLACK Leaf Nodes: 4894
BLACK Expanded Nodes: 7203
BLACK Total Nodes: 12097

RED WINS!!!
RED Player: «(-_-')»
But most importantly, BLACK looooses (boooo!)
BLACK Player: (Jººººº) J ~ 
Red player alg: Minimax-Alpha-Beta, eval: 3
Black player alg: Alpha-Beta-Search, eval: 3
Depth: 4

Figure 44: Minimax-3 vs ABS-3 Depth 4

Game 13: (Minmax-A-B, EF-1) vs (Minimax-A-B, EF-2) – Depth 4

Refer to files min-1-min-2-4-full and short.log for full output.

```

Black moves from 14 to destination (in sequence): dest: 9

|---|---|---|---|---|---|
|---|1|---|2|---|3|---|4|---|
|---|5|---|6|---|7|---|8|---|
|---|B|---|---|---|---|---|---|
|---|9|---|10|---|11|---|12|---|
|---|---|---|B|---|---|---|---|
|---|13|---|14|---|15|---|16|---|
|---|---|b|---|---|---|---|---|
|---|17|---|18|---|19|---|20|---|
|---|---|---|r|---|---|---|---|
|---|21|---|22|---|23|---|24|---|
|---|---|---|---|R|---|---|b|---|
|---|25|---|26|---|27|---|28|---|
|---|---|b|---|---|b|---|---|
|---|29|---|30|---|31|---|32|---|
|---|---|---|---|---|---|---|---|
RED Leaf Nodes: 19139
RED Expanded Nodes: 25786
RED Total Nodes: 44925

BLACK Leaf Nodes: 14910
BLACK Expanded Nodes: 26728
BLACK Total Nodes: 41638

DRAW!!!
Red - (я •`_•)я  я( 'д' я) - Black
Mission FAILED...We'll get em next time!
Red player alg: Minimax-Alpha-Beta, eval: 1
Black player alg: Minimax-Alpha-Beta, eval: 2
Depth: 4

```

Figure 45: Minimax-1 vs Minimax-2 Depth 4

Game 14: (Minmax-A-B, EF-1) vs (Minimax-A-B, EF-3) – Depth 4

Refer to files min-1-min-3-4-full and short.log for full output.

Round 80 Black's Move...

Black moves from 10 to destination (in sequence): dest: 7

	1	2	3	4
b	b	B		
5	6	7	8	
	9	10	11	12
R				
13	14	15	16	
			r	
17	18	19		20
21	22	23	24	
		R	R	
25	26	27		28
29	30	31	32	

RED Leaf Nodes: 24683

RED Expanded Nodes: 37489

RED Total Nodes: 62172

BLACK Leaf Nodes: 38490

BLACK Expanded Nodes: 51299

BLACK Total Nodes: 89789

DRAW!!!

Red - (ж •_•)ж ('Д') - Black

Mission FAILED...We'll get em next time!

Red player alg: Minimax-Alpha-Beta, eval: 1

Black player alg: Minimax-Alpha-Beta, eval: 3

Depth: 4

Figure 46: Minimax-1 vs Minimax-3 Depth 4

Game 15: (Minmax-A-B, EF-2) vs (Minimax-A-B, EF-3) – Depth 4

Refer to files min-2-min-3-4-full and short.log for full output.

```
Black moves from 5 to destination (in sequence): dest: 1
```

	B						
	1	2	3	4			
5	6	7	8				
9	10	11	12				
		R					
13	14	15	16				
R	R	R					
17	18	19	20				
21	22	23	24				
25	26	27	28				
29	30	31	32				

```
RED Leaf Nodes: 23521
```

```
RED Expanded Nodes: 39900
```

```
RED Total Nodes: 63421
```

```
BLACK Leaf Nodes: 14272
```

```
BLACK Expanded Nodes: 18544
```

```
BLACK Total Nodes: 32816
```

```
DRAW!!!
```

```
Red - (я •'_•)я ы( 'Д' ы) - Black
```

```
Mission FAILED...We'll get em next time!
```

```
Red player alg: Minimax-Alpha-Beta, eval: 2
```

```
Black player alg: Minimax-Alpha-Beta, eval: 3
```

```
Depth: 4
```

Figure 47: Minimax-2 vs Minimax-3 Depth 4

Game 16: (A-B-Search, EF-1) vs (A-B-Search, EF-2) – Depth 4

Refer to files abs-1-abs-2-4-full and short.log for full output.

```
Round 80 Black's Move...
Black moves from 14 to destination (in sequence): dest: 9
```

	1	2	3	4
5	6	7	8	
B				
9	10	11	12	
13	14	15	16	
	b			
17	18	19	20	
21	22	23	24	
	r			
25	26	27	28	
	R		b	
29	30	31	32	
b		b		
30	31	32		

```
RED Leaf Nodes: 19139
```

```
RED Expanded Nodes: 25786
```

```
RED Total Nodes: 44925
```

```
BLACK Leaf Nodes: 14910
```

```
BLACK Expanded Nodes: 26728
```

```
BLACK Total Nodes: 41638
```

```
DRAW!!!
```

```
Red - (ж •`•)ж ыс ( 'Д' ы) - Black
```

```
Mission FAILED...We'll get em next time!
```

```
Red player alg: Alpha-Beta-Search, eval: 1
```

```
Black player alg: Alpha-Beta-Search, eval: 2
```

```
Depth: 4
```

Figure 48: ABS-1 vs ABS-2 Depth 4

Game 17: (A-B-Search, EF-1) vs (A-B-Search, EF-3) – Depth 4

Refer to files abs-1-abs-3-4-full and short.log for full output.

```

alpha: 1512 beta: 2147483647 val: 1512 move start: 8
Black moves from 10 to destination (in sequence): dest: 7

|   |   1   |   2   |   3   |   4   | | |
| b | b   | B   |   5   |   6   |   7   |   8   |
|   |   9   | 10  | 11  | 12  |
R   | 13  | 14  | 15  | 16  |
| 17  | 18  | 19  | 20  | | |
| 21  | 22  | 23  | 24  |
| 25  | 26  | R   | R   | 27  | 28  |
| 29  | 30  | 31  | 32  |

RED Leaf Nodes: 24683
RED Expanded Nodes: 37489
RED Total Nodes: 62172

BLACK Leaf Nodes: 38490
BLACK Expanded Nodes: 51299
BLACK Total Nodes: 89789

DRAW!!!
Red - (я •_•)я ы( 'Д' ы) - Black
Mission FAILED...We'll get em next time!
Red player alg: Alpha-Beta-Search, eval: 1
Black player alg: Alpha-Beta-Search, eval: 3
Depth: 4

```

Figure 49: ABS-1 vs ABS-3 Depth 4

Game 18: (A-B-Search, EF-2) vs (A-B-Search, EF-3) – Depth 4

Refer to files abs-2-abs-3-4-full and short.log for full output.

```
Round 80 Black's Move...
Black moves from 5 to destination (in sequence): dest: 1
```

	B					
	1	2	3	4		
5	6	7	8			
	9	10	11	12		
		R				
13	14	15	16			
	R	R	R			
17	18	19	20			
21	22	23	24			
	25	26	27	28		
29	30	31	32			

```
RED Leaf Nodes: 23521
RED Expanded Nodes: 39900
RED Total Nodes: 63421
```

```
BLACK Leaf Nodes: 14272
BLACK Expanded Nodes: 18544
BLACK Total Nodes: 32816
```

```
DRAW!!!
Red - (я •`_•)я  м( 'Д' м) - Black
Mission FAILED...We'll get em next time!
Red player alg: Alpha-Beta-Search, eval: 2
Black player alg: Alpha-Beta-Search, eval: 3
Depth: 4
```

Figure 50: ABS-2 vs ABS-3 Depth 4

Full Simulation – 72 Games!

Refer to Project2-fullSimulation-A04626934.log for short output of all 72 games – no text output but the full game path will be shown. Alternatively, expand the Project2-fullSimulationVerbose-A04626934.zip to see the full log for all 72 games with verbose output. Please note that this file is **very** large, over half a gigabyte, and may take some time to open.

```

Round 22 Red's Move...
Red moves from 5 to destination (in sequence): dest: 14

+---+---+---+---+---+---+
|   |   | r |   | 3 |   | 4 |
+---+---+---+---+---+---+
| 1 |   | 2 |   |   |   |   |
+---+---+---+---+---+---+
| 5 |   | 6 |   | 7 |   | 8 |   |
|   |   |   |   |   |   | r |
+---+---+---+---+---+---+
| 9 |   | 10 |   | 11 |   | 12 |   |
|   |   | r |   |   | r |   |
+---+---+---+---+---+---+
| 13 |   | 14 |   | 15 |   | 16 |   |
|   |   |   |   |   |   |   |
+---+---+---+---+---+---+
| 17 |   | 18 |   | 19 |   | 20 |   |
|   |   |   | r |   |   |   |
+---+---+---+---+---+---+
| 21 |   | 22 |   | 23 |   | 24 |   |
|   |   |   |   |   |   | r |
+---+---+---+---+---+---+
| 25 |   | 26 |   | 27 |   | 28 |   |
|   |   |   |   |   |   |   |
+---+---+---+---+---+---+
| 29 |   | 30 |   | 31 |   | 32 |   |
|   |   |   |   |   |   |   |
+---+---+---+---+---+---+
RED player took 1 piece(s).

RED Leaf Nodes: 3052
RED Expanded Nodes: 4814
RED Total Nodes: 7866

BLACK Leaf Nodes: 4894
BLACK Expanded Nodes: 7203
BLACK Total Nodes: 12097

RED WINS!!!
RED Player: <(-_-')>
But most importantly, BLACK looooses (booooo!)
BLACK Player: (J o o) J ~ L
# of Games Played: 72

```

Figure 51: Full Simulation of 72 total games!3

--- NOTHING FOLLOWS ---