Team Name: Genuine Intelligence

Phillip Stewart

[phillipwstewart@gmail.com](mailto:phillipwstewart@gmail.com)

**AI Assignment #1: Chess KR-K Endgame**

I completed this project on my own. I wrote the entire C++ code myself, using only the C++ standard library. I used some extra files (Makefile, in, box.py, …) to develop quickly; these files are not required to run the program, yet I have included them for completeness (I also left my .git directory in the archive, as it documents my development process).

To compile the program, you may use the provided makefile. The command, "make main", will create the main program for running the game. The other programs created by running make were for testing. To run the program, run "./main" from a terminal, and the text output will provide instructions and print a simple board in ascii. To simply test the program with the case in testCase.txt, run "./main <in". You may also compile the program with any C++ compiler of choice, following the examples in the makefile. As the program uses stdin and stdout, you will need some console or terminal to run it. I compiled the programs provided with this package on Mac OSX 10.10.2 with g++, which invokes clang through XCode; as such, these programs most likely will not run correctly on linux or other versions of OSX, and certainly will not run on Windows.

I designed and implemented a heuristic function for both players X and Y. I went through many versions as I revised and tested my progress. The two functions defined in heuristic.cpp, named heuristicX and heuristicY evaluate the board and return an integer value indicating the relative value of that board state for the given player. The value returned ranges from ~0 for loss states to ~8000 for very desirable states. There are special cases which return 32768 and 65536 (the values 215 and 216, chosen somewhat arbitrarily). Those instances are when player Y can capture the Rook, when player X checks the king and forces it to move toward the edge, and when player X puts Y in checkmate. I will describe the final versions of the heuristics I used below.

The heuristic for player Y starts with a simple idea, but has complex additions to stay competitive near checkmate. Player Y should try to keep his king near the center. So I used the distance from center (a sort of x2 + y2 distance) and devised a simple function to yield base values of around 7,000 in the center down to 2,000 in the corners. This base value is the "distance factor". Then, I check whether the Rook is blocking the king from advancing toward the center. If the Rook is directly blocking the row towards center, there is a bonus for advancing toward the Rook. This value is the "rook factor". In some cases, such as when the Rook is directly blocking a row, it helps to know where the other King is; If the King is in a troublesome spot, a certain value is deducted. If the King is in a spot which makes it useless, a small value is added. This value is the "king factor". The heuristic returned is the addition of the distance, rook, and king factors. As mentioned above, if the king captures the Rook, a very large value is returned (215).

The heuristic for player X is a bit more complicated. The base value for this heuristic is also called the "rook factor". It ranges from ~0 to ~7,000 based on how closely toward the edge the opposing king is trapped. The Rook essentially adds 1,000 points to the heuristic for each row it moves toward the edge while pinning the king. Before this calculation however, I check for specific conditions; A direct force towards the edge returns 215, and a checkmate returns 216. If the Rook is adjacent to the opposing king, a value of 0 is returned, unless the supporting King is protecting the Rook, in which case a "protection factor" is added (which increases based on King position). The "king factor" increases as the supporting King approaches its target position, which is opposite the other king over the Rook's blocking line. If the King and Rook position themselves as such, they can force the opposing king toward the edge, so a lot of tricky calculation goes into the King's target factor. A final factor that comes into play is whether the Rook is positioned on an edge. This factor is used to ensure the Rook stays far from the opposing king on the blocking line. Thus, the heuristic is the addition or the Rook, King, protection, and edge factors.

I first devised and implemented simple move algorithms, which sort the heuristic values for all possible moves and return the next move with the highest heuristic value. I improved the heuristic functions while testing with these move functions, so the heuristics are well suited to play without a search function. I later implemented a version of minimax that replaces the minimizing branches with the basic move function for the opponent, so that the opposing player automatically selects their next best move. The values at depth are passed up the tree, and the move that maximizes its recursive branch is chosen. This search method was not always better than simply selecting the next best move (basic hill-climbing). So I tried variations on minimax, including an additive minimax, which adds the next best branch's heuristic up the tree; I also tried an expected value maximizing search function which was based on the probability that the opponent would react with certain move and the heuristic values from deeper branches were multiplied up the tree. I tested and customized this search function at different depths, but it never performed better than basic hill-climbing. In the end I settled on basic move selection for player X (highest ranked next move), and additive maximizing at a depth of 2 full turns ahead for player Y. There is also a mechanism in place for player X to avoid repetition of the same moves, however, no such mechanism was necessary for player Y, as they are trying to prolong the play indefinitely.

I have included the game summaries, which are automatically saved to file on completion of a game, for the two given test cases below. When the program is run, a more verbose output is sent to stdout, where the board and selected move are printed on the execution of each player's move. I have included in my code a few defined constants that determine the level of verbosity of output. The summaries below simply show the initial board state and the player moves in portable-game-notation (the standard for chess programs).

Output from running test-case 1:

Initial board:

-- -- -- -- -- -- -- --

8| | | | | | k| | |

-- -- -- -- -- -- -- --

7| | | | | | | | |

-- -- -- -- -- -- -- --

6| | | | | K| | | |

-- -- -- -- -- -- -- --

5| | | | | | | | R|

-- -- -- -- -- -- -- --

4| | | | | | | | |

-- -- -- -- -- -- -- --

3| | | | | | | | |

-- -- -- -- -- -- -- --

2| | | | | | | | |

-- -- -- -- -- -- -- --

1| | | | | | | | |

-- -- -- -- -- -- -- --

a b c d e f g h

Game summary:

1. Rh7 kg8

2. Ra7 kf8

3. Kf6 ke8

4. Ke6 kd8

5. Rh7 kc8

6. Kd6 kb8

7. Kc6 ka8

8. Kb6 kb8

9. Rh8# {Checkmate. Player X wins.}

Output from running test-case 2:

Initial board:

-- -- -- -- -- -- -- --

8| | | | | | | | |

-- -- -- -- -- -- -- --

7| | | | k| | | | |

-- -- -- -- -- -- -- --

6| | | | | R| | | |

-- -- -- -- -- -- -- --

5| | | | | | K| | |

-- -- -- -- -- -- -- --

4| | | | | | | | |

-- -- -- -- -- -- -- --

3| | | | | | | | |

-- -- -- -- -- -- -- --

2| | | | | | | | |

-- -- -- -- -- -- -- --

1| | | | | | | | |

-- -- -- -- -- -- -- --

a b c d e f g h

Game summary:

1. Rh6 ke7

2. Ke5 kd7

3. Kd5 kc7

4. Kc5 kd7

5. Ra6 ke7

6. Kd5 kf7

7. Ke5 ke8

8. Ra7 kd8

9. Kd6 ke8

10. Ke6 kd8

11. Rh7 kc8

12. Kd6 kb8

13. Kc6 ka8

14. Kb6 kb8

15. Rh8# {Checkmate. Player X wins.}