Joel Johnson, Wilder Perera, Matt Mulhall, Rob Rispoli
December 13th, 2019
Konane AI Project
CSE 4705

We have 6 functional python files included in our main source code. We have board.py, connect.py, game.py, konane_driver.py, minimax.py, and move.py. Our board.py file simply holds a board state and is able to change that board state either by making moves or removing pieces.

The connect.py is an application programming interface (API) used to interact with reading and writing server messages to the Konane server through telnet. The API built includes error handling to account for timeouts as well as disconnections and faulty server specifications. The program uses the python package telnetlib which you can see here. Using several different API calls to telnetlib, we are able to handle responses from the server with regular expressions or exact strings. Telnetlib allows us to specify timeouts for all read functions. An example of the initial prompt reads the ?Username response from the server and then has the user write the username back to the server. Here the timeout is specified to 10 seconds.

```
self.cnx.read_until(b"?Username:", 10)
self.cnx.write((self.username + "\r\n").encode('utf-8'))
```

All of the turns and removals are handled through this API as well as game termination due to opponents or yourself winning the game. At the end of the game we handle closing the connection with a quick two line function.

```
    """ close telent connection """
    def close(self):
        print("Closing connection...\n")
        self.cnx.close()
```

The Konane driver file is the connection between all of our code. The konane driver leverages the connection API, the game class, board class, and minimax functionalities to receive, transmit and process moves to and from the server. While running the game, the konane driver takes opponent moves and applies them to the game, and generates AI moves and applies them to the game. When the game terminates the konane driver closes the connection and relays the end of game message to us.

Our minimax.py holds our minimax implementation. This includes a plain generic minimax implementation that calls upon a static evaluation function located in game.py to get

the utility for a board state. It also call on gen_successors in game.py to get the possible moves from a board state.

In game.py we play the actual game and run minimax with our heuristic which is in the functions static_eval. In game.py we also generate all the successor board states and use these in our minimax algorithm. Game.py is responsible for setting what piece type the AI is and how to start the game by removing pieces. Game.py also contains many of the functions necessary to run the minimax AI such as get_moves, gen_successors, and static_eval. Game.py also holds our current board object that is used to determine next moves and is printed after every move.

```
joels-mac:src Joel$ python3 konane_driver.py
Please enter your username:
9
Please enter your password:
9

Successfully connected to Konane server at artemis.engr.uconn.edu...
Username: 9
Password: 9
Enter an opponent number: 8
Game:1661
Player:1
17 X O X O X O X O X O X O X O X O X O
16 O X O X O X O X O X O X O X O X O X
15 X O X O X O X O X O X O X O X O X O
14 O X O X O X O X O X O X O X O X O X
13 X O X O X O X O X O X O X O X O X O
12 O X O X O X O X O X O X O X O X O X
11 X O X O X O X O X O X O X O X O X O
10 O X O X O X O X O X O X O X O X O X
9  X O X O X O X O X O X O X O X O X O
8  O X O X O X O X O X O X O X O X O X
7  X O X O X O X O X O X O X O X O X O
6  O X O X O X O X O X O X O X O X O X
5  X O X O X O X O X O X O X O X O X O
4  O X O X O X O X O X O X O X O X O X
3  X O X O X O X O X O X O X O X O X O
2  O X O X O X O X O X O X O X O X O X
1  X O X O X O X O X O X O X O X O X O
0    X O X O X O X O X O X O X O X O X
   a b c d e f g h i j k l m n o p q r
```

Image 1: Example output when our program begins a game on the server.

Originally our evaluation of game states was based on multiple factors including: number of opponent moves, number of our moves, adjacent opponent pieces, number of our pieces around the perimeter and the number of opponent pieces around the perimeter. Over several thousand tests, we reached the conclusion that these factors don't necessarily correlate together into winning the game. We tried to convert Konane strategies into heuristics but were unable to

find reliable information on best Konane strategies. Because of this, we chose to implement a basic heuristic that simply tries to minimize opponent moves, and chose to optimize our minimax to be able to go to further depths. This solution has yielded the most robust players, beating out our other versions 100% of the time.

How our Program Progressed over Time

We started our program off with simple guessing of valid moves. This meant that over time our AI could potentially get to winning states just by luck. Konane is an interesting game, as you can win by simply randomly doing valid moves, as long as your opponent does not have a higher level strategy. From the get go, we decided to try many variations of heuristics. We first started off with a naive heuristic that prioritized maximizing the difference between our moves and our opponents move, which is not a bad heuristic. We used guessing as a baseline, and continually tested them against one another in batches of 50. The second heuristic was marginally better than guessing at around 65% win rate, which we did not think was enough.

We then moved on to testing against the 65% heuristic, since its baseline was better than guessing. Thinking that adding factors would improve our score, we began by implementing heuristics that tried to minimize and maximize the following factors: perimeter pieces of ourselves and our opponents, and adjacent opponent pieces. We also experimented with amplifying and dampening these factors over time using the amount of moves as a scalar. This effectively meant we could develop an early game or end game strategy for the AI. An example of this is the following heuristic: amount of our moves - amount of opponent moves + our perimeter pieces * amount of moves - opponent perimeter pieces / amount of moves. The scalars provide a strategic advantage, only using them where we felt they were most effective. Despite our high hopes for the heuristics, they simply did not perform well against our baseline.

After the frustration of a non-improving heuristic performance, we decided to go back to the basics. Originally we were trying to figure out what are the most impactful strategies of the game, but we realized that we wouldn't be able to figure those out without having played the game for a considerable amount of time. We knew that there was only a loose correlation between the previous attempts at heuristics and winning game states, so we left those behind. To get our heuristic, we simply used the fact that the game ends when our opponent has no moves left, using the difference of moves was simply not related to the end state of the game. In Konane, you can win with only having one move left, that makes your opponent have 0 moves, employing a differential heuristic is just adding more unrelated factors. In light of these realizations, we decided to simply make the following heuristic: -1 * amount of opponent moves. This heuristic, very simply, prioritizes game states where our opponent has as few moves as possible. In testing, this heuristic absolutely demolished all previous heuristics with quasi-determinism, every time losing 0 games. This was not all that puzzling, considering the other heuristics were looking for correlations that didn't exist.

After a multitude of testing we also introduced another feature that prioritizes deeper minimax searches in the beginning and end of the game. We did this to balance performance and winning, as with a maximum minimax depth, our program would never terminate fast enough. Using this strategy, we could maximize the benefit of more depths of game states, as we did not feel the middle of the game had as large of an effect as the beginning set up and last 50 or so moves. The greatest way to explain how we developed our system is testing. We simply tested for hours until we got a heuristic that we were satisfied with, always pitting the newest against the previous best.