Squeeze-IT AI Agent Documentation

Austin Maddox, Katrina Bueno, and Cody West

I. Design

Components of the game include:

1. Game Master (Internal)
    a. A global 2D array to represent the 8x8 Squeeze-It board.
        i. The first row is populated with the black pieces (‘B’) and the last row is populated with the white pieces (‘W’). The empty cells are represented by a string containing a single space (‘ ‘).
        ii. Functions are included to form and update the board.
    b. A 4-tuple to represent a player’s move.
        i. The first two elements represent the initial coordinates of the piece and the last two represents its destination.
        ii. There are functions that interpret the tuples to ensure the legality of a player’s move.
    c. Functions to execute moves and communicate with the agent.
        i. These call the functions that will make the move or get the next move if one of the players is an agent. Most of the functionality results from the following function.
            1. move()
                a. Move is called once at the bottom of squeeze_it_GUI.py and then subsequently schedules itself to run 50 milliseconds later before terminating
                b. The function checks the value of the current player’s heuristic and runs on two cases
                    i. player: the current player is a human that needs to make a move. The function checks to see if the player has provided a move and passes if they have not or implements the move if it is valid.
                    ii. Any other value: any other value is considered to be a heuristic name, and is passed to the make_move() function, which is responsible for running the minimax. The new game state is returned, replacing the old game state.
                c. Upon a successful move in either case, the current player is changed and the turn count is incremented

2. GUI (External)
    a. The external design involves the creation of a GUI using the Tkinter Python module. The code works primarily on the principle of checking the internal structure of the code (i.e., the game master) every 50 milliseconds and updating the GUI to reflect these changes. This is primarily accomplished through the following function.
        i. update_GUI()
            1. This function goes through every part of the GUI, from the board to the labels, and updates each piece with the most current version of the values they represent. Furthermore, the function checks to see if the game is over, updating the GUI to indicate the winner if so.
            2. The function also provides input validation for human players, indicating their currently selected space by highlighting it.

II. Heuristic Evaluation Functions

The implemented heuristic evaluation functions are:

1. Simple
2. Defensive
3. Aggressive
4. Center

A. Simple Heuristics

Implemented as a function to check if the AI has more pieces than the other player. If the AI does have more pieces that the opponent the AI will receive a positive counter, otherwise it will receive a negative counter

B. Defensive Heuristics

The defensive heuristics first obtains an initial counter from the simple heuristics function, then runs its own checks to get a different counter value. These heuristics encourage behaviors such as staying close to edges, and diagonals. The defensive strategies discourage the AI from having pieces next to each other, and only having a single space in between its pieces.

C. Aggressive Heuristics

The aggressive heuristics works to encourage a close proximity to the opponent, and capturing opponent pieces. The aggressive strategies discourage the AI from having the same number of pieces as the opponent. There is a check that allows this heuristics to determine if making an aggressive move or defensive move would be a better option.

D. Center Heuristics

The center heuristics encourages the AI to keep its pieces close to the center. Center heuristic is initialized with the simple heuristic function

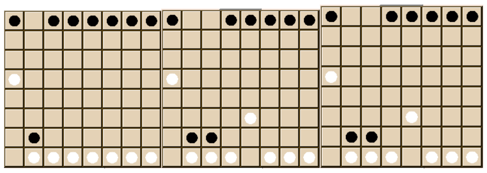
III. Search Strategies

A. The Search component of this project utilizes the minimax algorithm with alpha-beta pruning. This algorithm was an obvious choice for a game like Squeeze-it, as iterating through all moves and finding the one that performs best in the long run is a surefire way to ensure the AI does not make any mistakes that could cost the game. Beyond a depth of 2 however, minimax can take an incredible amount of time to run, initially clocking in at 15 seconds with a depth of 3 and the simple heuristic. The alpha-beta pruning, as expected, helps considerably with this problem, reducing the runtime under identical depth and heuristic by over 14 seconds.

B. One of the principal limitations of the structure of minimax is the tendency to pick the first move it comes across. Oftentimes, barring an incredible mistake on the part of the opponent, almost all game states on the board have the similar heuristic values. When this happens, the AI prioritizes the top left corner, regardless of color, as that is the corner it visits first when iterating through the game. Our best method of combat against this is to add many different modifiers to the heuristics besides just modifiers for number of pieces (i.e., moving beyond the simple heuristic defined above). The more reasons we give the minimax to pick different spaces for different reasons, the more intelligent the behavior of the algorithm will appear.

C. Despite the implementation of alpha-beta pruning, runtime does tend to be a limiting factor on minimax. When moving to a depth of 4, runtime jumps up to the previously seen 15 seconds, even with pruning. However, because this game is not incredibly complicated and players often have simple countermoves for basic attacks, going beyond a depth of 3 does not seem necessary.

D. One particular intriguing action regularly employed by the AI is the use of pieces on the same row to cover one another in the case of a squeeze. For example, in the following series of states, white moves from (4, 7) to (4, 5) in an attempt to take the black piece on (1, 6) via a squeeze.



However we can see that black responds with a move from (2, 0) to (2, 6). Now, if white takes either of the black pieces available to it, black will respond with a reverse-squeeze of the two capturing white pieces, resulting in a +1 trade for black.

E. This leads into one of the most important effects of minimax: it knows for a fact when trades will work out in its favor. In the previous example, black moved a piece into harm's way with the explicit knowledge that if that piece were to be taken, a superior capture opportunity would be present. While this fact may not be useful against other minimax agents, as they know the capture is a trap, a less intelligent non-minimax agent or, better yet, a human, are more likely to fall for the bait.

IV. Average Response Time

I7-9700 (Windows 10 v2004 Python 3.7.3)

	Simple	Aggressive	Defensive	Center	Overall
Avg Response Time (ms)	307.261	1056.217	1511.565	1031	1023.954
Std. Dev.	83.765	871.724	772.197	832.432	443.992

Xeon E5-1680v2 (MacOS 11.0, Python 3.8.3)

	Simple	Aggressive	Defensive	Center	Overall
Avg Response Time (ms)	462.304	1710.174	2253.87	1330	1550.139
Std. Dev.	88.392	1313.658	1246.207	485.034	698.215

I7-7820HQ (MacOS 11.0, Python 3.9.0)

	Simple	Aggressive	Defensive	Center	Overall
Avg Response Time (ms)	334.979	1323.367	940.245	694.489	823.27
Std. Dev.	84.832	735.319	663.676	335.059	415.849