Ian Quinn, Jared Tulayan, Shivam Verma
Assignment 2 - Report

Section 1 - The Interface

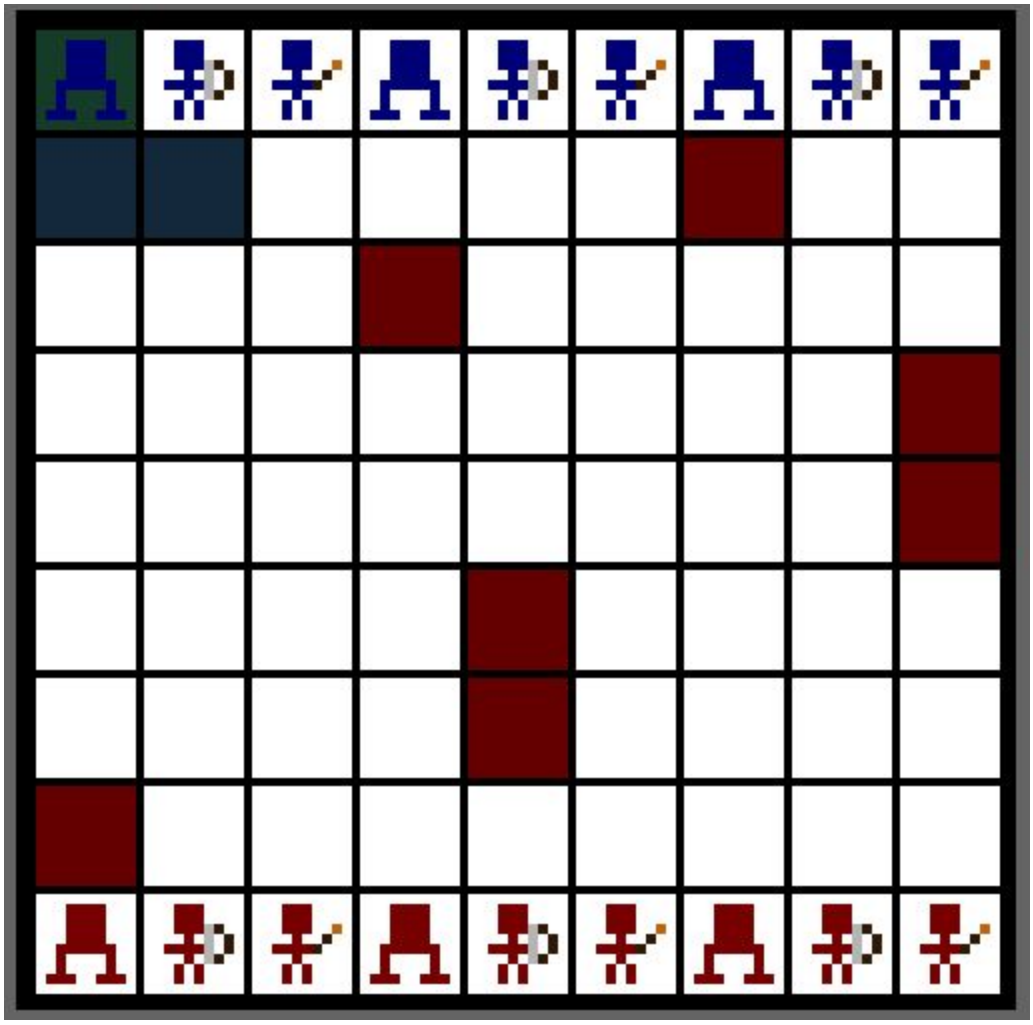

Figure 1 - The GUI board we have where the user can choose a possible move (where the gray squares outline possible moves). The red squares are pits.

Section 2 - Metric of Evaluation

The metric of evaluation chosen was the difference in the number of pieces possessed by both sides. So, if the AI has more pieces, the difference will be negative and if the Human Player has more pieces, the difference will be positive.

Section 3 - The Minimax Search + Alpha-Beta Pruning

```
def minimax(board, depth, is_major, h = h_disable, a = -inf, b = inf):
    b_metric = evaluate(board)
    b_moves = board.generate_moves(is_major)
    ret = (0, None)

    if depth == 0 or abs(b_metric) == inf:
        ret = (h(board = board), None)
    else:
        moves_queue = PriorityQueue()
        sign = 1 if is_major else -1
        value = inf * -sign
        mm = board.create_memento()

        for m in b_moves:
            moves_queue.put((sign * h(move = m), m))

        while not moves_queue.empty():
            item = moves_queue.get()
            move = item[1]
            func = max if is_major else min

            print(move)

            # Perform move, evaluate
            board.move(move[0], move[1])
            value = func(value, minimax(board, depth - 1, not is_major, h, a, b)[0])
            board.restore(mm)

            if is_major:
                a = max(a, value)
                if a >= b:
                    ret = item
                    break
            else:
                b = min(b, value)
                if b <= a:
                    ret = item
                    break

    return ret
```

Under the comment “Perform move, evaluate” it shows that the recursive call to minimax progressively uses a lower depth (depth - 1), showing that the minimax searches out to that depth as it works itself out. The bottom-level state is where depth=0, and as shown, the value of b_metric (our evaluation metric) is also taken in at that point and is used to evaluate the state.

Alpha-Beta pruning takes place closer to the end of the function where the function breaks out of the while loop based on where the alpha/beta values are relative to each other. If it is the human player’s turn and the alpha value is >= the beta value, it breaks out of the loop and returns the optimal move. On the flip side, if it is the AI player’s turn, and the beta value is <= the alpha value, it breaks out of the loop and returns the optimal move.

Section 4 - The Heuristics

We created five heuristic functions to be used. They are:

H_moves - which obtains the number of viable moves that each player has

H_advantage - Which compares how many winning fights a player can have, so it compares the player’s number of wumpuses to the opponent’s mages, the player’s heroes to the opponent’s wumpuses, and the player’s mages to the opponent’s heroes

H_euclidean - Compares the Euclidean distances of the player’s winning fights, so the distances between the player’s wumpuses to the opponent’s mages, and so on.

H_manhattan - Compares the Manhattan distances of the player’s winning fights, so the distances between the player’s wumpuses to the opponent’s mages, and so on.

H_spacing - Tries to maximize the spacing between the player’s own pieces, so the player can cover more of the board

The average times for each heuristic to perform a move are:

	Moves	Advantage	Euclidean	Manhattan	Spacing
Average Time (sec)	8.84	2.78	19.42	19.19	15.00

So, the clear benefit of the advantage heuristic is that it takes the least amount of time. A benefit that the other 4 heuristics have over the advantage heuristic is that their resultant values are more variable over the course of the game, so they may better depict the true state of the game.

Section 5 - Putting it Together

```python
# Minimax with alpha-beta pruning
# board: Board state
# depth: Current depth to search
# is_major: Whether the player is the major player
# h: Heuristic function, defaults to h_disable
# a: Alpha
# b: Beta
# Returns the move and value best suited to transition into the most optimal state
def minimax(board, depth, is_major, h = h_disable, a = -inf, b = inf):
    b_metric = evaluate(board)
    b_moves = board.generate_moves(is_major)
    ret = (b_metric, None)

    if depth == 0 or abs(b_metric) == inf:
        pass
    else:
        moves_queue = PriorityQueue()
        sign = 1 if is_major else -1
        mm = board.create_memento()
        opt_value = -inf * sign
        opt_move = ()

        for m in b_moves:
            moves_queue.put((sign * h(board = board, major = is_major), m))
```

```python
    while not moves_queue.empty():
        item = moves_queue.get()
        move = item[1]

        # Perform move, evaluate
        board.move(move[0], move[1])
        c_minimax = minimax(board, depth - 1, not is_major, h, a, b)
        diff = c_minimax[0] - opt_value

        if diff * sign > 0:
            opt_value = c_minimax[0]
            opt_move = move

        board.restore(mm)

        if is_major:
            a = max(a, opt_value)
            if a >= b:
                break
        else:
            b = min(b, opt_value)
            if b <= a:
                break

    ret = (opt_value, opt_move)

print(ret)
return ret
```

Finally, we allow a heuristic to be passed into the minimax function, with the default being h_disable, which is basically equivalent to no heuristic. In addition, our final version of the minimax function shown above was optimized from the figure displayed in section 3. Here, in our final version, we are able to integrate the minimax algorithm with alpha-beta pruning as well as a chosen heuristic and the earlier specified evaluation metric.