Artificial Intelligence Gameplay Assignment:

Alex de Vryer and Ben Cox (Team 'DROP_TABLE_Students')

Evolution of Agent Approaches Throughout Development:

Our agent went through several evolutions before we decided on our final submission agent. Listed below are the details of the agents which we have considered throughout development but were not ultimately selected as our final agent:

Purely Greedy Agent:

The first agent which we developed was a Greedy agent which used our utility function to decide which move should be played given the current board state. Our utility function is explained in more detail below. Each time our agent had to place a piece, all possible moves for the current board state would be generated, and the utility function would then calculate the desirability score for each of the states. We would pick the move which had the highest utility value to play next. However, we wanted to explore using some adversarial search techniques so we quickly began developing more agents.

Minimax Player with Alpha-Beta Pruning:

Soon after developing our Greedy agent, we decided to develop an agent which utilised Minimax search with Alpha-Beta pruning to make more informed move decisions. We initially set the agent to have a 2-ply lookahead.

However, we found that there were several problems with this approach, namely:

1. Our agent did not make the best decisions as a 2-ply lookahead was not sufficient enough for our agent to pick the best move to play.
2. The agent took a very long time to deliberate whilst finding the best move to play. Tetress can have a very high branching factor; we found that some board states had upwards of 1000 possible moves which could be chosen by our agent during deliberation. As a result, the evaluation of all of these states using Minimax took a very long time, meaning our agent regularly ran out of time whilst deliberating.

Minimax/Alpha-Beta with Limited Branching Factor:

To try and overcome the issues with agent deliberation time mentioned previously, we decided to limit the branching factor of the Minimax search tree to a maximum of 10 moves for each node expansion. To choose the 10 moves for MIN and MAX, we used our utility function to determine the 10 moves with the highest utility value for MAX to play as well as the 10 moves with the lowest value for MIN to play for the current board state.

Whilst this improvement did speed up our agent's deliberation time significantly, we found that our agent was not making the best move choices due to the limitations of a 2-ply lookahead.

Minimax/Alpha-Beta with Varying Branching Factor and Lookahead Depth:

To help our agent make more optimal move decisions, we decided to vary the lookahead depth and branching factor of Minimax depending on the number of piece placements available to our agent in the current board state. Our variation strategy is below:

- If we have more than 50 moves available, use 2-ply lookahead for 10 best moves
- If we have between 26 and 50 moves available, use 4-ply lookahead for 5 best moves

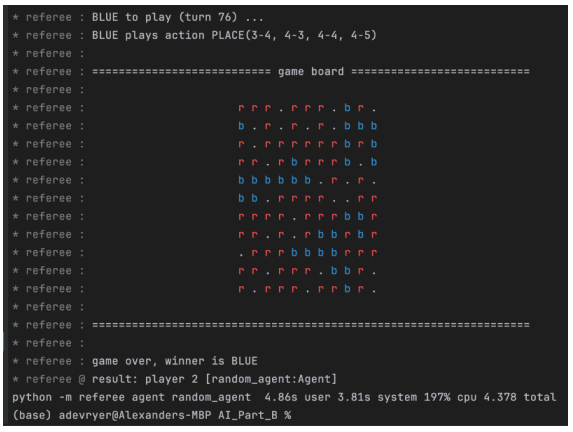
- If we have between 2 and 25 moves available, use 6-ply lookahead for 3 best moves
- If we have only 1 move available, just play that move immediately

Upon testing, we found that this helped our agent make more optimal move placements than it did previously. Our agent would search deeper in instances where more informed move decisions were required (i.e. less moves were possible), ensuring that the consequences of each move were well considered and allowing the best move to be selected for the given state. In instances where our agent had many moves available, less careful deliberation was required, allowing us to use a quicker search strategy with a lower lookahead. However, our agent's deliberation time was still too slow on Gradescope to play a full game which required many move placements.

Combined Greedy and Minimax/Alpha-Beta Agent:
One of our final Tetress agents was a combined Greedy-Minimax agent which chooses a different move selection strategy based on the number of available moves in a given state.

- If we have more than 40 available moves which our agent can play, we select the move with the highest utility value (i.e. Greedy move selection).
- If we have up to 40 moves available, we will do Minimax with Alpha-Beta pruning with a 4-ply lookahead on the four best moves available to our agent as determined by our utility function.

We found that this agent had a much shorter deliberation time than our previous agents because our agent did not use Minimax when it had a large set of moves to choose from. When we needed to more carefully deliberate when we had fewer moves, our agent ran Minimax to select moves more carefully, making a more optimal decision.



However, this implementation wasn't without its problems, namely its low performance against random agents. When playing 100 rounds against a random agent, we found that this agent only won 66 of those games. We found that in all the games which our agent lost, they ended with both players being unable to place any new moves (i.e. the board was full) as shown in the image to the right. We realised that because our Minimax function had limited branching factor, it will not evaluate some states which may allow the other agent to win as our utility function may not deem these states to be "optimal" ones to include in the Minimax tree.

Final Agent (Modified Greedy Agent with 2-ply Lookahead for "Death Moves"):
Our final agent selected for submission was a Greedy agent which chose moves to play using the utility function described later on in this report. However, our agent has several key modifications compared to usual Greedy agents to enhance its gameplay ability:

1. When evaluating the best move to play, our agent looks at a sequence of two states to determine which move is the best instead of just examining the final game state. Our agent considers both the game state at the start of our agent's turn as well as the state

after it plays one of the moves it can choose to play. This is described further in the Utility Function section.

2. Our Greedy agent also avoids states where “death moves” could possibly be played by the other player (i.e. a move which the other agent could play which will end the game and let the other agent win). In essence, our agent uses two utility functions; one picks states with maximum utility and the other picks states which prolong play. To prolong play, our agent:
    a. Finds all possible tetromino placements for the current board state and calculates the utility function values for all moves it could play,
    b. It then iterates through all the moves starting with the one with the highest utility function value first,
    c. It finds all possible tetrominoes the other player could play after we place the specified tetromino. It then checks if our agent can place a tetromino in that state after the other player has played (i.e. our agent checks two moves ahead to see if it can still play),
    d. If the other agent has one or more tetromino placement options which could end the game, the initial placement action is pruned and not considered by our agent.
    e. Our agent continues until it finds a tetromino it can place which will not allow the other player to end the game at all, even if this move does not have the highest utility function value. If this is deemed to be impossible for any available move, our agent just plays the move with the highest utility value.

We found that this additional evaluation allowed our agent to win more games against the random agent than our previous agents could (this is described further in the Performance Evaluation section). However, we found that there were still some games where our agent could lose as it was sometimes impossible to find moves which did not allow the other player to win. We hence attempted to implement a 4-ply lookahead to more rigorously detect and mitigate possible “death moves” earlier on, however this took too long to evaluate and was not included in our final agent. We also only choose to run this lookahead for states which have up to 400 possible moves, as states with more than 400 moves are unlikely to lead to a death state within two moves and it is not worth spending the computational power on such an unlikely event.

Our agent also has a much shorter deliberation time compared to previous agents. The longest deliberation time we observed during testing was approximately 6 seconds (this occurred in states which had nearly 400 possible moves and required some pruning). On average, our agent does not usually take longer than half a second to deliberate.

Our agent has a preset strategy for choosing its first move to play. We specified two “I” tetrominoes of coordinates ((2, 2), (2, 3), (2, 4), (2, 5)) and ((7, 6), (7, 7), (7, 8), (7, 9)) which our agent can choose from to be its first move of the game. We selected these tetrominoes as no matter where the other player decides to place their first piece, our agent will always be able to place one of these two tetrominoes on the board (if we are playing as Blue).

Furthermore, as discussed in more detail in the Utility Function section below, our agent’s primary goal is not to limit the branching factor of the other player; rather, we are choosing our moves to try and reduce the number of pieces the other player has compared to our agent, as well as limit the other agent’s ability to win games by reducing “death moves”. Therefore, our

agent does not gain much from trying to place a piece next to the other player's piece if we are playing as Blue in the first move.

Utility Function:

Final Utility Function:

The final utility function used in our game returns an unnormalised weighted sum. However, compared to traditional utility functions which return the weighted sum of a given state, our utility function considers a sequence of two states: the current state (the state at the start of our agent's turn) as well as the future state of the game (one of the moves the agent can choose to play). Our utility function takes the following attributes into account:

1. Number of "open sides" which our agent's pieces have compared to the other agent (i.e. sides of pieces which are not touching another piece) multiplied by 10. This was done so that our player has several opportunities to place new pieces whilst trying to restrict the other player where possible.
2. Total number of pieces our agent has compared to the other agent multiplied by 125 so our agent chooses states which have a higher piece count than the other agent.
3. The total change in piece count each player has in this new state compared to the previous state (i.e. if a line is cleared, which agent ends up losing the most pieces). This is only considered if the total change in pieces for any agent is negative compared to the original state. This is multiplied by 300 to encourage line removal.
4. The number of lines in the current game state which have a large number of squares in them (8 or more pieces) minus the number of lines which have a low number of squares (2 or fewer pieces) multiplied by 5 to encourage our agent to spread its piece placements out during the first few game moves.

Our final utility function aims to make our agent into more of an "attacking" agent, where if it can clear a line to its advantage (i.e. the other agent loses more pieces), our agent will almost always choose this option due to the high weight of 250 chosen for this feature. Our agent also always prefers states which lead to us having a higher piece count than the other agent and also tries to restrict the movement/winning prospects of the other player where possible. We found that this type of agent performed very well against a random agent.

Prior Versions:

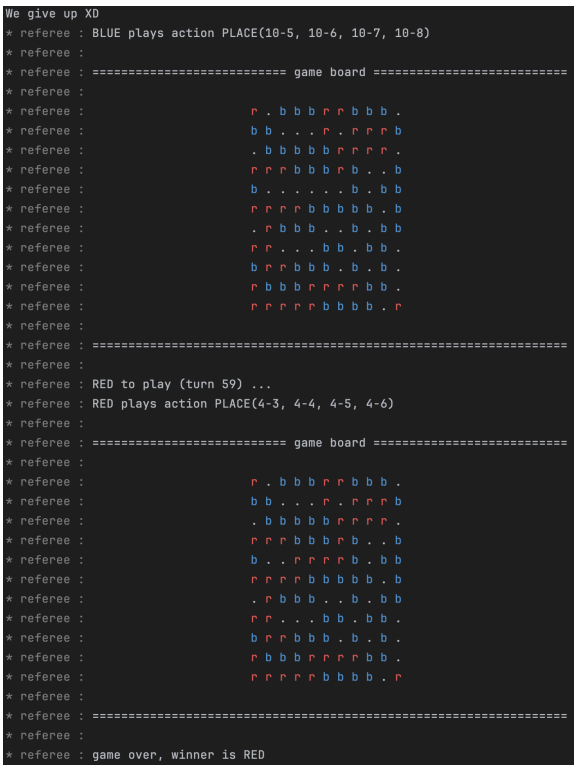
We went through several iterations of our utility function. Originally, we intended to include the number of possible piece placements for each player as a feature. However, it took a very long time to calculate for all possible piece placements in a given state. Ideally, this would have allowed our agent to perform much better as we could have chosen states which specifically limited the other player's moves, however the resource constraints made this infeasible.

We also originally included functionality to detect "holes" (i.e. gaps in the board of size three or less where no pieces could be placed) and avoid states which had several of these. This ensured that the game was less likely to end because no more pieces could be placed by either player (i.e. the board was full); a game would be won either by having more pieces or successfully limiting the opponent's placement ability. However, this also took too long for it to be included in the final version.

Weight Evaluations:
Our weights for (a) the number of pieces on the board, and (b) the total change in piece count between states have largely remained fixed throughout development. We wanted to ensure that our agent prioritised states where lines could be cleared and had a higher number of pieces, so we left these weights at their current high value.

However, for our other two weights (open sides and number of pieces in lines), we ran a small number of playoffs (approximately 20) against a random agent and (a) recorded the results, as well as (b) watched the game board state change over time. After the series of playoffs, we would change the two weights to better reflect the behaviour which we wanted to see in the game. For example, initially we had a higher weight of 10 (instead of 5) assigned to line lengths. However, we found that this had the side effect of creating more holes in the board, so the weight was steadily decreased to its current value.

Performance Evaluation:
Evaluation against Random Agent to Verify Agent Performance:
To evaluate whether or not our agent played better Tetress than a basic agent, our final agent played 100 rounds of Tetress against our random agent (50 rounds as Red, 50 rounds as Blue). We manually ran each game and tallied the results. Overall, our agent won 84 of the 100 games, with 43 of those being played as Red and 41 of those being played as Blue.



Throughout the playoffs, we noticed a few things:

1. In all the games which we lost, they ended with our agent having no possible move options which would remove the opportunity for the other agent to play a “death move” in the given state (e.g. see the first image to the right where our final agent (Blue) lost). As explained previously, this could be mitigated by increasing our agent’s lookahead, however this took too much time to evaluate.
2. If the game was to be played up to 150 moves, our agent would win as our utility function prioritises removing the other player’s pieces.

After the series of playoffs, we deemed our agent’s performance was highly satisfactory against a random agent, especially compared to our previous combined Greedy-Minimax agent which won 66 games. Even though our final agent did not necessarily implement any adversarial search methods, it delivered a more satisfactory performance than our other agents which did.

Other Aspects and Supporting Work:
We have included several additional features to our game which were not specifically covered or mentioned in class. In addition to the multiple algorithmic enhancements described above, we have some more aspects which have not yet been mentioned in this report:

Caching of Utility Function Value using a Transposition Table:
Throughout gameplay, we are very likely to have to calculate the utility value for a particular board sequence more than once. To avoid having to recalculate this value each time and to satisfy the time resource constraints imposed on our agent, we decided to cache the value of the utility function so it can be quickly retrieved if it has already been calculated using a transposition table (the hashing method used for game states is discussed in more detail later). In our agent, the transposition table takes the form of a dictionary within the Game class; the key is the board hash and the value is the result of the utility function (Laramée, 2000).

We also experimented with caching the possible actions which can be played by each player in a particular state to speed up play. However, we decided against including this in the final version due to memory constraints as our agent's memory usage exceeded 250MB memory whilst playing games which ran for a high number of moves.

Zobrist Hashing:
To be able to cache the utility values in the transposition table, we have implemented Zobrist hashing in our agent. Zobrist hashing is a hashing method which can be used by agents playing many different board games which require use of transposition tables, and has been widely used in Chess and Go agents (the code written for Zobrist Hashing in Tetress can be observed to the right) (*Zobrist Hashing*, 2021).

```
import random

from referee.game import PlayerColor, Coord, BOARD_N

NUM_PLAYERS = 2


6 usages  Alex de Vryer *
def init_board():
    # create an 11 * 11 * 2 array filled with random 16-bit numbers
    hash_table = [[[random.randint( a: 0, pow(2, 16)) for k in range(NUM_PLAYERS)] for j in range(BOARD_N)]
                  for i in range(BOARD_N)]
    return hash_table


1 usage  Alex de Vryer *
def hash_index(player):
    # return the index in the array dimension of size 2 for the specific player
    if player == PlayerColor.RED:
        return 0
    elif player == PlayerColor.BLUE:
        return 1
    else:
        return -1


9 usages  Alex de Vryer *
def board_hash(state: dict[Coord, PlayerColor], hash_table):
    # return the hash of the specified board state
    h = 0
    for i in range(BOARD_N):
        for j in range(BOARD_N):
            if Coord(i, j) in state.keys():
                # find the 16-bit number for this specific entry and XOR this with the current hash value
                player_index = hash_index(state[Coord(i, j)])
                h ^= hash_table[i][j][player_index]
    return h
```

Firstly, we create a three-dimensional array of size BOARD_N * BOARD_N * NUM_PLAYERS (11 * 11 * 2) and store a random 16-bit number in each array entry. We also assign an index for each player for the NUM_PLAYERS dimension of the array (in our agent, red players have index 0 and blue players have index 1).

After creating the initialisation vector, we check every square on the board to see if it is occupied with a player's square or not. If it is, we take the random 16-bit number in the entry of the three-dimensional array which corresponds to the current position on the board as well as the player's colour and bitwise XOR that number with the overall value so far (initially 0). We continue until all coordinates on the board have been checked (Laramée, 2000).

At the end, a unique hash of any given board sequence is created which is then used by the utility transposition table for caching purposes.

Sources Used:

Laramée, F. D. (2000, June 11). *Chess Programming Part II: Data Structures*. GameDev. Retrieved from https://www.gamedev.net/tutorials/programming/artificial-intelligence/chess-programming-part-ii-data-structures-r1046/

Minimax algorithm in game theory: Set 5 (Zobrist Hashing). GeeksforGeeks. (2023, April 26). https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-5-zobrist-hashing/

Zobrist Hashing. Zobrist Hashing - Chess Programming Wiki. (2021). Retrieved from https://www.chessprogramming.org/Zobrist_Hashing