Project Part B - Tetress Agent

Our Approach to Tetress

We have used minimax search to select an action for our game-playing agent. We first considered MCTS as an approach since the lecture example on AlphaGO made it appear to be an attractive search strategy. However, after reading about the potential downsides and difficulties of implementing it in this context, we decided against it. Using the advice of a tutor, we also implemented a-b pruning as described in the lectures with our minimax search since we were told it was a simple addition to the search function.

Our minimax function takes the standard approach of recursively calling a min_value and max_value function. Although, we also implemented two set data types, to make sure we don't expand the same board twice, creating a hashing function for efficiency. In the future, with more time and expertise, we would consider a more efficient way of move ordering our expanded nodes to increase minimax efficiency and a tree-data structure for searching and saving child nodes.

Further, we decided that our minimax algorithm would only be used for the middle set of turns in the game. We modeled our game playing off the progression of a chess game, which has an opening, midgame and endgame. For the opening, we derived the idea of 'book moves' from chess. This is a set of predetermined moves that we use for the first five moves of the game, which we decided on using our evaluation function and machine learning. We applied our evaluation function to every possible opening move to decide on a set, and then implemented logic to place an array of book moves, iterating through until one is playable, and if none are playable, calling our minimax search. One of the key benefits of using this book moves approach results from the fact that the highest branching factor occurs at the opening of the game when there is the most possible playable space. Thus, by using predetermined moves, we avoid a possibly extremely inefficient period of minimax search, thus heavily speeding up our program. In a similar vain, we also decided to reduce the depth of our minimax search based on the number of turns taken, going from 3 to 10 layers as the game progresses.

From reading Reference 3, we considered implementing an endgame search as well based on this chess research. This uses the notion of 'killer moves' such as checkmates in chess. However, we eventually decided to just use minimax for our search in the endgame due to the ambiguous nature of categorizing a game state as in endgame. Instead, we decided to make our evaluation function a function of the number of moves played, the time remaining, and space remaining in the board, resulting in an evolving minimax search through the game.

Project Part B - Tetress Agent

Our Evaluation Function

Our evaluation function is a combination of many smaller weighted components, an idea that we gathered partly from the TDLead algorithm learnt in class. We consider 4 factors in determining a board's score. These include the ratio of the players squares compared to the opponents squares, the players branching factor on their next move, the opponents branching factor on their next move, and the amount of rows and columns covered by the players pieces.

The ratio of player squares vs opponent squares was our highest weighted component. This is just a good general measure of how good a board is. If the player can remove the opponents squares by placing their own squares and not losing many of their own squares, this will be considered a good move. Furthermore, in an end game scenario, this is very important, as having more tiles on the board wins you the game.

The branching factors for both players were also highly weighted. This is the fundamental idea of the game, as we want to increase our branching factor when we can, allowing us to place more pieces. At the same time, we want to decrease the opponents branching factor to as close to zero as possible, as this is one of the win conditions for the game.

Lastly, we had the span of our pieces across the board. This was calculated greedily by finding the least amount of row or column clears required to fully delete all tiles of a colour. Although this doesn't carry as much weight as the other factors, it can still be important. Early game, a good spread across the board is useful, as this is when the player is in most danger of just having all of their pieces deleted if they are not careful. We calculated this greedily to save complexity, as this replicates the popular minimum cover problem, which is NP-Hard.
Each of the 4 factors were normalized and then multiplied by a coefficient. The coefficients were selected using our own game knowledge of good positions on the board.

Our Use of Machine Learning

We utilized the python tensorflow library for machine learning to determine the constants for our weighted evaluation function. This was done by playing our random agent against our minimax agent and determining which evaluation constants were most conducive to our agent winning. We also used the same technique to determine the book moves as mentioned. We essentially put a cap on the number of book moves that we wanted and then aimed to figure out the best opening of our player by the percentage of wins it received when using our minimax approach when playing against an agent that played pieces randomly.

Performance Evaluation

To judge the performance of our player, we took our working expansion module and built a new agent on top of it that could play against our original minimax agent. This agent simply played random moves instead of using minimax to choose a piece. This was extremely helpful for us in order to tweak our agent in order to achieve the maximum win rate. It also allowed us to use machine learning so that we could decide on code changes based on win percentage against this agent.

Unique Aspects of Our Agent

The two most significantly unique aspects of our code were our expansion methodology and our Bitboard data structure.

Bitboard

We found that the provided board structure was very slow for use when searching in Part A of the project. After some discussion, we knew that some numeric internal board representation would lead to the fastest comparison and cheapest storage, and most significantly, if we used binary values this would be the fastest since they are ⅛ the size of a normal integer (bit vs byte). However, a simple check showed we had three states for a square (red, blue or empty) and only two binary states (1 or 0), so we decided to make a trade-off, and in doing so created BitBoard.

The tradeoff was that we would store our board rep using binary, but using two 120 digit binary integers, one to represent the red pieces on the board and another for the blue pieces, meaning our board rep was now 240 bits, rather than 960 if we used one string of 120 integers. Due to our use of the bitboard class, we could not use the game's built-in Coord type, and so the Coords given to and returned from our search function had to be converted to and from 'indexes'.

Indexes represent our way of finding a square on the board in the bitboard binary strings. To convert a coord to an index, you first take the row value and multiply it by 11, and then take the column value and perform modulo 11 on it. The value 11 is used because it is the dimensions of the board. Thus, at every action and update call of our agent, we had to do conversions between both indexes and coords as well as the original board provided by the referee and our bitboard to update our internal state representation and return moves in the language of the referee (place actions).

The benefit of using a bitboard comes in the speed of operations. Our most used functions are set_tile for placing tiles on the board, get_tile for seeing if there is a tile in a coord/index, and copy, for copying a board. With a bitboard, these 3 operations all have a complexity of O(1). All of the bitboard operations leverage the incredibly fast speed of bitwise operations, allowing for very efficient calculations and modifications to the board.

Expanding logic

To go with our fast and light board representation, we had to have an efficient expansion method that used bitboard internally to reap its benefits. This was tested with a variety of different methods as we compared them to find the fastest. From Part A, we had an enum of all the Tetress pieces which we had iterated through to expand, but we soon realized that this was quite a naive approach

We then stumbled across two key ideas. The first was 'the diamond', where we realized that when expanding from a starting red square, the pieces placed must have all their squares within a 9 square wide diamond shape, representing the maximum reach of piece placement in all directions. We also realized that all the pieces and their orientations represented every possible combination of four squares when expanding. Thus we could throw out the idea of a piece altogether and rather first check the blocked squares (by opponent squares) in the diamond, and then expand outwards playing one square at a time in any allowed direction until we had played four squares that constituted a valid move.

We tried a variety of different methods when expanding out from one square, but landed on using a modified DFS in our final code. Our DFS expansion takes a tile on the board, and expands out in all directions until it reaches a depth of 4. If the DFS finds a tile at a depth of 4, this means that this shape created by the DFS is a valid tetromino. By doing this, we are able to completely remove the need for the Piece type. We chose DFS over BFS for one main reason. In a scenario where there is an empty gap of size 3 adjacent to the tile we are expanding off, BFS would waste time expanding into this. Whereas our DFS will try this path once, find that it fails, and then go and try all other possible paths.
The time complexity of our DFS is O(b^4), where b is the branching factor. However, in practice, it performs much better than this. When there is even just a few filled in tiles within the 4 tile range of the expanding tile, many possible shapes are removed from the possibilities, and many computations are

cut. The closer a tile is to the expanding tile, the more this computation time is cut. And because the nature of the game groups tiles next to each other, this means the games average performance is much better than this worst case performance. The worst case only ever occurs in the cases where there is 1 tile that we are expanding off, and this tile has no other tiles within 4 squares.

Supplementary Code

Throughout the development process we made a variety of different functions both to test functions that we had made, and as alternative methods that we didn't end up implementing. These included our random agent (discussed earlier) as well as testing functions to ensure our expand and minimax modules. These testing functions often checked that we had reached the desired depth, picked the correct child node, or expanded all possible nodes. We also implemented a variety of different sub functions for our evaluation function (discussed above) and moved between the combinations of these functions in order to achieve the best possible agent. Similarly, we created alternative functions for most of our modules during the drafting process before deciding on a final one. This included an alternative expanding method which used Uniform Cost search instead of a depth-first search that was eventually cut out due to inefficiency. Finally, all bitboard and utility function code was created separate from the original agent and then added in as classes in the program file.

References

1. https://www.chessprogramming.org/Opening_Book
2. https://www.chessprogramming.org/Bitboards
3. https://www.chessprogramming.org/Killer_Move