Steven Reed

Professor Jasim

CSC 4444

29 April 2025

Creating an Agent to Play the Game Tetris Using Informed Search

Problem Domain

The problem domain tackled in this project was creating an agent to play the game Tetris. Tetris is a single-player puzzle video game where the player manipulates falling geometric shapes, called tetrominoes, to fit them together and clear lines on a rectangular grid. Tetrominoes are each made of four square blocks, composed into seven distinct shapes (I, O, T, S, Z, J, L). The game is 10 cells wide and 20 cells tall. Each block in a tetromino is the size of exactly one cell of the game board.

In a continuous loop, tetrominoes slowly fall down the game board and then freeze when they attempt to fall into the bottom of the game board or into another frozen tetromino. While a tetromino is falling, the player can perform the following actions upon it: left, right, down, or rotate. Left, right, and down cause all parts of the tetromino to move one cell in the given direction. Rotate causes the tetromino to rotate 90 degrees clockwise. If the player's attempted action would cause the falling tetromino to collide with a frozen tetromino, the falling tetromino is not moved. Manipulating the falling tetromino is the only action the player can take.

The goal of the game is to rotate and move these falling tetrominoes, such that they fit together tightly when frozen. When a horizontal line of the game board is completely filled with frozen tetromino blocks, the line is cleared. Then, all tetrominoes above the cleared line(s) fall, and the player earns points proportional to the number of lines cleared. The game ends if the stack of frozen blocks reaches the top of the screen such that no more blocks can be placed.

The agent created for this project performs the task of creating viable Tetris action sequences for a given board and tetromino. By the inputting these actions in the place of human moves, the agent can essentially play a game of Tetris.

My motivation behind building this AI was that I really enjoy the game of Tetris and wanted to do a project with a manageable scope. I decided to work alone on this project, so choosing a topic I understood and could implement in a reasonable amount of time was crucial.

Methodology

To solve the problem of creating quality Tetris move sequences, a greedy informed search algorithm was utilized. An informed search algorithm is perfect for Tetris as there is essentially a single decision to make at any given time (where to place the falling tetromino) but it is difficult to map the ramifications of this decision. An informed search utilizes heuristics to estimate the quality of potential decisions. This allows for high quality decisions to be made in reasonable time in difficult to map environments.

Greedy informed search was chosen over A* search for two reasons. The first is the lack of backtracking in Tetris. In Tetris, once a move is made, there is no going back, and the outcome of that move cannot be changed. Therefore, it is logical to choose a search algorithm which only looks forward. The second reason is that the Tetris player has a relatively low amount of information about the future of the game. That is to say that the player only knows the current board layout, the currently falling tetromino, and the next tetromino. This lack of information is further compounded by the fact that the next piece is not factored into this search algorithm. Because of this lack of future information, choosing a good move does not involve

any planning ahead. Additionally, because of the mechanics of Tetris, choosing a move with high short term quality also creates the opportunity for high quality moves in the future. Greedy informed search is good at choosing immediate quality moves while only looking ahead.

All portions of this project were written using the python programming language, specifically version 3.12. The IDE used for this project was Pycharm by JetBrains. No plugins or settings need to be downloaded or configured on the host machine or IDE besides the libraries discussed below.

To create an agent that plays Tetris using Python, a working version of Tetris, implemented in Python, was needed. This implementation of Tetris was primarily created by following a YouTube tutorial by Coder Space called “Detailed Tetris Tutorial in Python”. In this tutorial, the pygame library was used to create a Tetris game in python. Pygame is a python library designed to facilitate the creation of games and multimedia applications. It has built-in functions to handle graphics, sound, input handling, timing, and collision detection as well as a strong user community and lots of tutorials. This Tetris implementation also utilized the python “random” library to choose random tetromino pieces. The implementation from this YouTube video was slightly modified to accept input from the search algorithm and to include an optional “bag” feature. When the “bag” feature is enabled, it is guaranteed that every seven tetrominoes dropped will contain all seven unique tetromino shapes.

The logic of the Tetris search algorithm is totally separate from this Tetris game implementation and was designed to be compatible with any implementation of Tetris. The Tetris search algorithm only requires two input parameters and utilizes a barebones Tetris implementation to simulate necessary Tetris functionality. The first parameter required is the current board state represented a list of lists where each sub list is a row of the board. The second required parameter is the shape of the current piece represented as an uppercase character. The search algorithm and its supporting code utilizes only the built-in functionalities of the Python 3.12 language. The “unittest” python library was utilized for testing the code of the search algorithm and its barebones Tetris implementation. Git was utilized for version control.

The search algorithm works in three major steps. First, a function generates a list of possible board states which could occur when the current piece becomes frozen. Along with this it generates a list of player action sequences that could be taken to reach each generated possible board state. Then, all the possible board states are given a quality rating based on certain heuristics. Finally, the sequence of player actions needed to reach the highest quality board state is inputted into the Tetris game.

To generate the list of possible next board states, the current piece is dropped in every distinct rotation, from every horizontal position. Dropping in this sense means that the piece is repeatedly lowered on the game board until it collides with another piece. These generated board states are added to a list of possible next board states. These board states are generated inside of a barebones Tetris implementation called Small_Tetris that was created for the search algorithm. The pseudocode for this process can be found in the pseudocode section at the end of this paper and it assumes that an instance of the Small_Tetris class exists and has the current piece in its starting position at the top, middle of the game board.

To calculate the quality of the possible board states, four heuristics are utilized. These are: completed lines, aggregate height, number of holes, and bumpiness. These heuristics were found by doing some online research and are relatively common in Tetris agents. The original use of these heuristics seems to be a Github project from nine years ago called “Tetris Artificial Intelligence” by isaaclino.

Completed lines is the number of board rows which are full of tetromino blocks in the possible board state. Clearing lines is the primary goal of a Tetris player so this heuristic is heavily positively weighted in the quality calculation. Aggregate height is the sum of the heights of all columns. The height of a column is the distance of the topmost block in that column to the bottom of the column. The Tetris player becomes closer to losing the game as the aggregate

height of the board increases so this feature is slightly negatively weighted. The weight is slightly negative because increasing aggregate height is a fact of the game of Tetris, but it should be limited within reason. Number of holes is the number of empty board cells that have at least on block above them. Having a large number of holes is problematic so this heuristic is very negatively weighted. It is impossible to clear a line while it has a hole so holes should be avoided whenever possible. Bumpiness is the variation in the topmost blocks of each column. It is calculated by summing the absolute differences in heights between all adjacent column pairs. Having a high bumpiness means that the surface where blocks can fall to is very irregular which makes it difficult to fit pieces nicely and remove lines. As such, bumpiness is moderately negatively weighted in the value calculation because a smooth board is more important than a low aggregate height but less important than minimizing holes.

Once the quality of possible board states has been calculated, the index of the highest quality board state is found. Then, the corresponding move sequence needed to reach that board state is returned. This move sequence is then fed into the Tetris game in place of human keyboard input. Feeding search algorithm input occurs at the max rate the game can accept human input, which is once per frame. The AI is only triggered to make a new move sequence when a new tetromino is created so one list of input exists at a time. When this sequence is executed correctly, the AI can play Tetris and a human can watch its moves on the screen.

Outcome

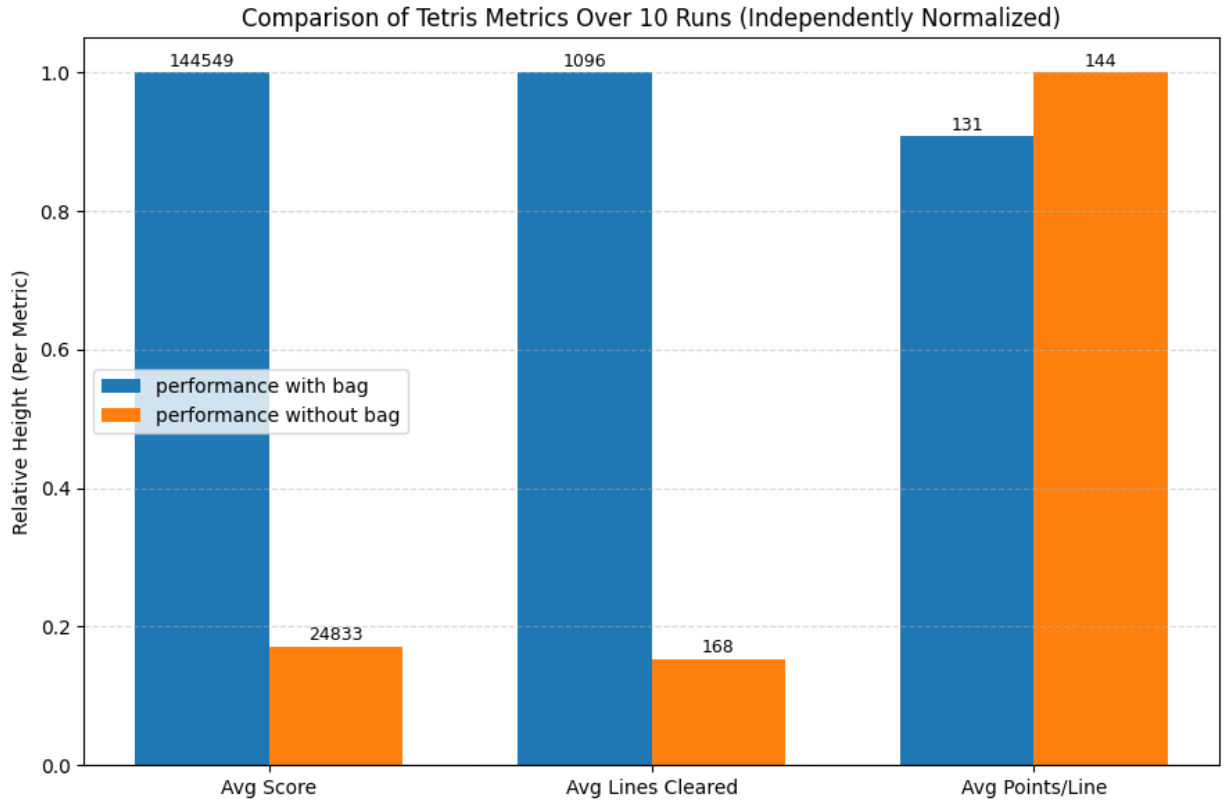
While it is difficult to assess the quality of individual moves generated by the AI, the sum of many generated moves seems to be of high quality, as the AI can achieve scores in the range of thousands to hundreds of thousands and clear lines effectively. There is quite a bit of variance in the scores the agent can achieve, as it is highly influenced by the sequence of pieces it is given. Generally, if the agent receives many of the same piece in a row, it will make poor decisions that often lead to death. With the bag feature enabled, this issue is mitigated and the agents scores, on average, much better than with the bag feature disabled. A graph of agent performance, comparing the average metrics of ten runs with the bag feature and ten runs without the bag feature can be seen below in the “figure” section.

With the bag feature enabled, the agent scored over five times higher and cleared over five times as many lines than with the bag feature disabled. This shows that a decrease in random piece distribution was very beneficial. Interestingly, the runs with bag feature disabled scored, on average, slightly higher points per line. This is likely due to random chance with such a small sample size of ten runs. However, it could indicate that the lack of a bag causes the agent to clear more lines at once. Further testing with more runs in both conditions would have to be conducted to find the true cause.

One major is issue with this agent is that it tries to make impossible moves when it is close to losing. This causes it to unintentionally make very poor moves and end the game. An impossible move in this sense means that the frozen tetrominoes block the falling tetromino from making a move that would otherwise be possible. The agent cannot see that the move will be blocked because horizontal moves and rotations are not blocked during calculations as they are during real play. Because the agent cannot see what moves are impossible, it typically dies by creating a moderately high stack and then repeatedly attempting illegal moves until the game ends. If the agent were given some way of knowing when a move is illegal, it may be able to change its strategy to account for this and achieve higher performance. Another strategy to get around this issue would be to rethink how possible moves are generated so that it takes frozen tetrominoes blocking moves into account.

In conclusion, the outcome of this project was successful in creating an agent capable of playing the game Tetris using an informed search algorithm. The agent is not optimal and could be improved upon, but for a proof of concept it proved that search algorithms can be used to play Tetris.

Figure:



Pseudocode:

```
generate_move_sequence_helper_fuction(count_rotations, count_left, count_down) {
        # make empty move sequence list
        # append correct number of rotations command to move list
        # if count left is positive
                # append count_left many left commands to move list
        # if count left is negative
                # append negative count_left many right commands to move list
        # append count_down many down commands to move list
        # append this move list to all move sequences list
        # return move list
} function_end

generate_possible_board_states() {
        # save copy of original board state
        # loop for the number of unique rotational orientations the current piece has
                # move current piece left until it hits the wall and capture how many times it was
                moved
                        # drop current piece and capture how many down moves it took to land
                        # add tetromino blocks to game board array
                        # add current board state to possible game state list
                        # generate the moves sequence to reach this position and add it to list
                                # append the left command to the move sequence as many times
                                as the piece was moved left
                                # append the down command to the move sequence as many
                                times the piece was moved down
                        # loop
                                # reset board state to original
                                # move piece back to top while keeping its horizontal position
                                # move piece one right and decrease left move count by 1
                                # if any block in tetromino out is of bounds
                                   # break
                                # drop piece and capture how many down moves it took to drop
                                fully
                                # add tetromino blocks to game board array
                                # add current board state to possible game state list
                                # generate_move_sequence_helper_fuction()
                        # reset board state to original
                        # move piece back to starting position which is one left of starting pos
                        # rotate piece
                # return possible board states list
        } function_end
```