UWA

CITS4211

Artificial Intelligence

Project Report

20925931

Eric (Jun) Tan

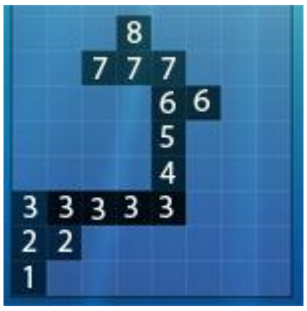
Project Analysis:

I will assume the reader has the knowledge of how to play Tetris (within the project specification rules), as to save space explaining the game rules. The end goal of my Tetris player is to minimise the maximum height of the squares on the final board. To achieve this, my AI program implements simple strategies and guidelines to follow when playing the game of Tetris.

Strategies:

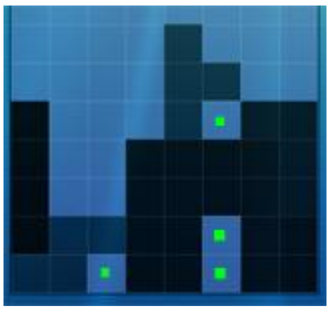
Minimize height

A clear concept of playing Tetris is to clear as many lines as possible, that is to keep the height of the blocks at a minimum at all times (for sake of the specification clearing multiple lines together will be omitted). To incorporate this into the analysis, increasing the total height of the board would be **penalized** and clearing lines would be **rewarded**.



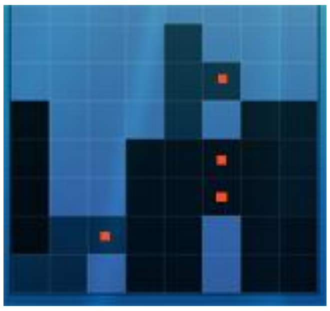
Packing blocks tightly together

Anyone who's played Tetris will instinctively know to clump tetrominoes together where they fit nicely and don't create any holes. The incentive for this is that to clear a row from the board, the entire row must be filled and a hole will be a huge detriment to allowing this. In terms of analysis, any pieces dropped that would create holes (spaces that are immediately blocked off from the top of the board by all left, right and above spaces) are **penalized**.



Clearing holes as fast as possible

A few holes here and there will be inevitable, especially when dealing with artificial intelligence (who will follow scripts and commands). Caused by perhaps specific ordering of pieces to be delivered or rare board cases, a hole (described above) will occur more often than not. The best way to deal with holes is to clear the rows above them as soon as possible. This is already described in the minimizing height strategy, so in this section of the analysis, we attempt to keep the amount of rows above holes to a minimum, which is to **penalize** blockades (any block that will be directly above a hole).



Program Analysis:

My program makes use of the above strategies and incorporates them as a (somewhat broken) passable Tetris AI. The algorithm used can be defined very simply:

1. Look at all possible current blocks (which include the next block to be delivered as well as any blocks in the buffer), and simulate every possible position and rotation of these pieces
2. Calculate a *score* (similar to heuristic estimates in A*), of each of the above positions
3. Move the block to the position and rotation that generates the highest score (in case of ties, choose the first block that generated that score)
4. Repeat for the next block until all blocks are done

To calculate the score of a position, I make use of a simple equation such as:

$$Score = x * total\ height + y * nHoles + z * nBlockades$$

where x, y and z are arbitrary weights of each factor, chosen within experimentation.

To begin the analysis, we know that originally we were going to penalize all height increases, hole creating and blockade placing, hence all the weights should be deemed negative. Initially I set x = -1, y = -0.5 and z = -0.1 as a very rough estimate as to how much my program should weigh the specific criteria and put emphasis on keeping the height at a minimum.

In a perfect scenario, where my program would function without issues, we would be able to experiment with these weights to generate the most efficient values for a given sample input and output case, and determine the most ideal values for them.

How my program plays Tetris:

I chose Java as my language of choice to write the program, as it is the language I have most experience with and have confidence with I/O functions as well as (board) game creating.

To start things off, I created a 2-dimensional array of Booleans which would hold the current game state of my Tetris board. A value of true would indicate the specific square is occupied with a block and a value of false meant it was empty. The size of my board would be dynamic and able to change at any time (conforming to the project specification). The other vital application necessary to being able to play Tetris is placing pieces onto the board legally (without breaking specified rules). My method `placePiece(int, int, int)` would take in the identity, rotation degree and horizontal position of a piece as parameters and place the piece on the board according to the rules specified in the project, delivered from the top of the board and falling until it reaches the bottom or another block. At the conclusion of every piece placed onto the board, it checks for any complete rows and removes them from the board, where all the rows above the removed row would fall down one row into their new position.

With the ground work of being able to play Tetris completed, I was ready to construct the AI to follow the strategies outlined and play the game. The actual code for the AI algorithm was not very complex at all, however does not perform completely optimally given that. Given the list of pieces to be delivered, it holds the specified strategies discussed above with relative ease, placing pieces in the way that each move will be the most optimal related to reducing height, holes and blockades. These specifications may lead to largely varied results. In certain situations, it can perform admirably, keeping the height of the grid below 4 lines when all the pieces line up nicely, however in many situations it may fall apart quite poorly and leave large gaps which it will fail to fill with pieces later on.

The two major concerns I've depicted from my algorithm are:

Lack of thinking ahead

Each piece is determined to be the 'best' piece to place judging solely on the pieces available, the next piece to be delivered or a piece in the buffer. Only out of these pieces is a decision derived with no knowledge of the 'next' piece to come straight after the current piece. An algorithm which takes into consideration one or two more pieces ahead will create many more desirable situations and avoid many holes and blockades.

No real incentive to place blocks next to each other

The way I've coded my program determines a 'hole' as a blank spot between 3 'filled' spots to the left, right and above it. My program (cleverly) avoids the problem of holes by creating larger holes, such as a hole of width 2 and height 2. According to my 'hole' finding code, it will miss this hole and ignore it, which will in result never become a cleared line.

I know these are fairly fixable problems and with a bit of time I would definitely revisit this algorithm and make adjustments. In an ideal program, I would simulate all combinations of both the current piece and the next piece to be filled in, as well as **reward** points for placing blocks next to currently existing blocks and walls/ground. The main reason I was not able to incorporate these into my design (apart from poorly managed self-time constraints) was that the method I was using to 'score' my piece positions only used the total Tetris board as a resource, with no knowledge of the last placed piece. From this constraint I could not find a way to determine whether the current block was placed next to existing blocks and/or walls without reconstructing the method.

Experiments:

Given my programs simplistic and flexible nature, it is able to process several hundred pieces per second with ease (however if I managed to implement the two solutions above it would be much closer to 5-10). All assumed variables can be altered at any point through the global static variables, including grid size, buffer size and the amount of pieces to be delivered. I find that a lot of success is achieved when increasing the buffer size (from say 1 to 5, which is common in many real-time Tetris games) greatly reduces the final height of my Tetris board. I've initially left the buffer size to 1 in-case of any inconsistencies arising from input files to output files, however I would definitely change the value to 7 when stress testing the program as it will still run with complete ease (a little bonus to having a simplified algorithm).

Tweaking the numbers of the weights in my score formula (described above) for a while yielded very interesting and slightly absurd results. I concluded that solely basing my scoring system off having the least height possible yielded far better results than weighing in all 3 variables (height, holes and blockades) thus in my code I have only made use of height although I've left the rest of my original code there (for show).

Extensive testing of input files, and changing of the variables results in rather flexible and defensive code, which I have yet to achieve to break. The program runs very fast and when variables are tweaked to their prime (buffer size to 7 or more), can play Tetris rather effectively. Clearly there will be defects and situations where a human would out-perform it but for a very simple design I am rather happy with how it can play.

The initial example input ran with my program produces a board of max height 3, the following output file will contain the following:

```
1 1 0
1 1 4
3 1 8
1 1 0
3 1 4
2 0 7
1 1 0
5 0 9
1 1 0
4 3 4
2 0 7
4 3 2
4 0 9
5 1 0
2 0 5
7 1 2
```

References:

All the images found on this report were taken from the following reference. Much of my algorithm used was inspired by the genetic algorithm used here, with slight altercations and much simplification. All code submitted by me was entirely written by me.

Coding a Tetris AI using a Genetic Algorithm | Lucky's Notes (2011)
http://luckytoilet.wordpress.com/2011/05/27/coding-a-tetris-ai-using-a-genetic-algorithm/