CITS4211 Artificial Intelligence

Tetris AI - Nathan Aplin and Ash Tyndall.

Submitted 20/05/2013.

Table of Contents

Analysis of Problem

In this assignment, we were asked to construct a "Tetris" artificial intelligence that attempts to output a list of Tetris optimal piece placements given a finite list of input pieces.

This AI is hoped to work on a general case of varying grid widths, tetromino buffer sizes, and a finite but arbitrarily large lists of pieces, but a suggested grid width (w) of 11, piece buffer (b) of 1 and a finite list of $\leq$ 1000 pieces is provided.

As described, the Tetris problem discussed is both simultaneously easier and harder than a conventional Tetris algorithm. By providing a list of the tetromino order ahead of time (unlike conventional Tetris, which usually has a 1 piece lookahead), a more "perfect" algorithm can be designed that would be able to better cope with a known future. Of course this simultaneously increases the potential solutions that can be considered by a large amount.

Fundamentally, a good solution to this problem would address the following aspects:

- Consideration of the utility of future actions, and the execution decisions that require planning.
- Utilization of data structures and algorithms that allow the quick searching of possible decisions.
- Modifiable utility function, permitting experimentation with different utility descriptions.
- Generalisation to arbitrary piece types and grid widths.
- Ability to cope with a variety of piece frequencies.

The problem itself can be divided into several discrete sub-problems that can be considered individually and then designed to inform each other. These are:

- Encoding and decoding – Converting the text input into objects and the objects into the required text output.
- Decision consideration – Have a data structure that allows the AI to consider the utility of different decisions.
- Decision execution – Have an algorithm that permits the AI to decide which action would be most advantageous to it, and also which actions would be legal Tetris moves.

In this report we present our solution for consideration, and analyse it from a theoretical and experimental standpoint.

Proposed Solution

The solution we propose takes a modular approach, which attempts to utilize the fact that decisions can be pregenerated ahead of time to attempt to play the optimal game. Our solution is written in Python.

The project is broadly split into two parts; the pregeneration step and the executive step.

Pregeneration Step

Because we know in advanced which tetrominoes are going to require placement, and the order in which they will do so, we elected to construct an placement decision tree ahead of time that contains all optimal (and some non-optimal) moves that can be used to formulate decisions about which placement to execute.

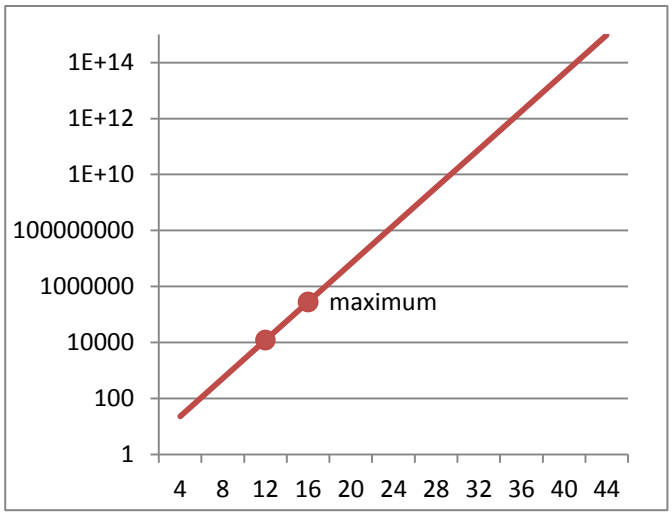

Figure 1: Grid area vs. number possible combinations

This placement decision tree (PDT) would consider a rectangular area of the board, and simulate moves that could occur within this rectangle, discarding those that are not possible, valid or optimal.[1]

Of course, to simulate a 4 x 11 (area 44) sized grid of Tetris move placements would be an intractable problem (Figure 1), producing far too many potential combinations. However, we can instead simulate several smaller rectangles and limit the placements of pieces within their boundaries.

Considering a Tetris grid of width 11, two 4x4 and one 4x3 rectangles will be simulated ahead of time to cover the entire width of the Tetris grid. These selections (denoted on Figure 1 as two round dots) sit at approximately the maximum simulation size possible on consumer hardware in a reasonable amount of time.

The pregeneration step code is contained primarily in the `generate_tree.py` file, which is designed to be a flexible and independent command line program to generate the placement decision tree. At its most basic, the pregeneration code takes two parameters (a height and a width), and a set of tetrominoes. It outputs a Python pickle file representing the resultant PDT to `tree.p`, which can then be imported into other Python scripts when necessary.

This part of the solution relies heavily on the advanced N-dimensional array data-type object supplied by the NumPy scientific computing library to represent Tetris grids as two dimensional arrays of Boolean values.

[1] Refer to the Definitions section.

Array Algorithms

NumPy provides a variety of logic functions (AND, OR, XOR, NOT) that can be performed on two arrays to provide another 2D solution array. These logic functions, because of their performance advantages (as they are written as a Python C extension) are used heavily as a way of calculating the possibility, validity and optimality of different tetromino combinations.

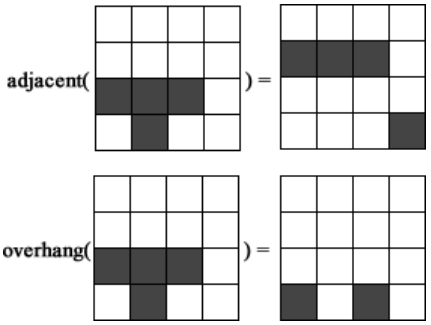

Figure 2: Adjacent and overhang functions

Calculating the PDT uses two different functions, known as `adjacent` and `overhang`, and a series of logical AND and OR operations to determine if a given placement is possible, valid and optimal.

The `adjacent` function (Figure 2) produces an array that contains True values on the highest blocks that are either directly above a piece, or the floor.

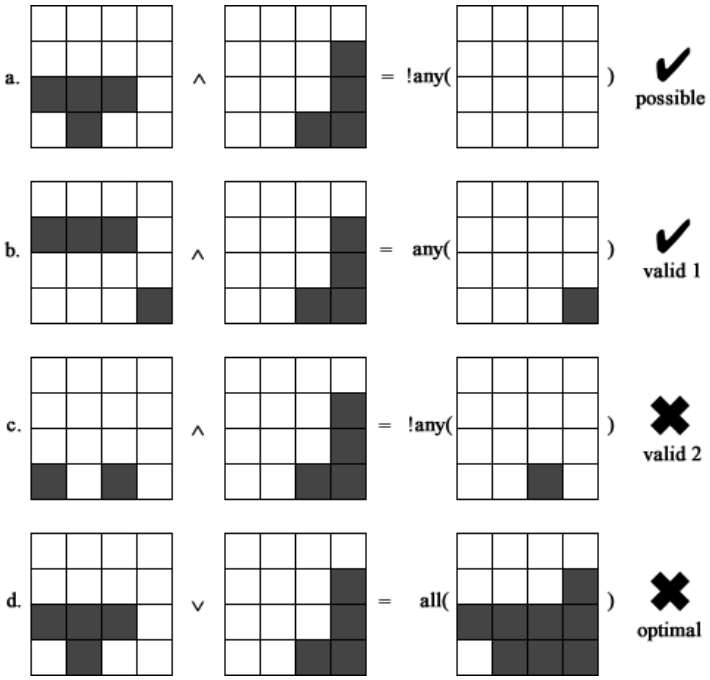

Figure 3: Computing possibility and validity

This array is useful, because (as demonstrated in Figure 3b) when ANDing a possible piece placement with this array, if any of the resultant blocks are True, then we can determine that the piece placement does not hover in midair, and is supported by other pieces as is required in Tetris.

The overhang function (also Figure 2) produces an array that contains True values on blocks that are determined to be shadowed (or overhung) by the given Tetris grid.

This further array is useful to determine if a given permutation is valid, as a logical AND of a piece placement and this array should not contain any True values if the placement is valid (Figure 3c). Using this we can ensure that pieces are placed in the correct order.

Finally, we can use a simple logical AND of the current board and a new piece to determine if a placement is possible (Figure 3a) and a simple logical OR of the current board and a new piece to determine if a board state could be considered optimal (Figure 3d).

Tree Structure

Once we have developed a variety of algorithms that are able to provide tuples of "good" moves from a variety of different Tetris placement arrays, we need to restructure the result set so that it is quickly traversable to make decisions related to the current Tetris game.

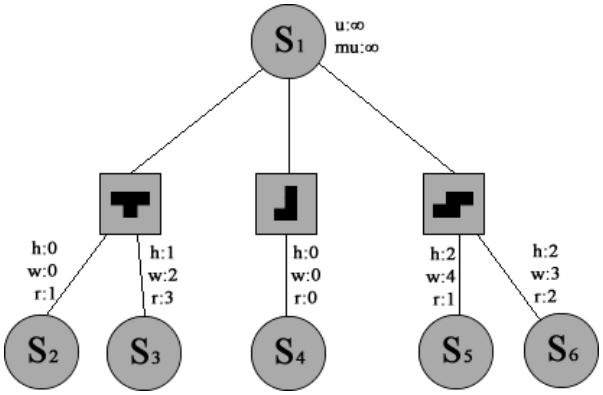

Figure 4: Structure of the PDT

The structure of the placement decision tree that we elected to use is displayed in Figure 4. Circular nodes are known as state nodes, and they contain both the utility of the current state of the game, as well as the maximum utility of any child nodes that exist under it.

State nodes are associated with a set of actions sets (represented by squares) that contain a set of actions (that is a placement indicating height, width and rotation) related to a given piece. These actions further index into either another state (thus continuing the game) or indicate that this is a terminal node and there are no further actions along this path.

Utility is calculated for each state as the state node is initialized with a board structure (it calls the `Board.utility` function in `tree.py`), and terminal states have a max utility that is equal to their own utility. Non-terminal nodes have their max utility calculated by a simple algorithm that traverses from terminal nodes to the root, modifying the maximum utility of each node it passes if it finds a better max utility in the child nodes it traverses.

The most important factor of the utility function is that the utility of states that are considered optimal is always $+\infty$, which will cause the root state node (a blank board) to always have a max utility and utility of positive infinity, provided there exists at least one optimal move in the tree.

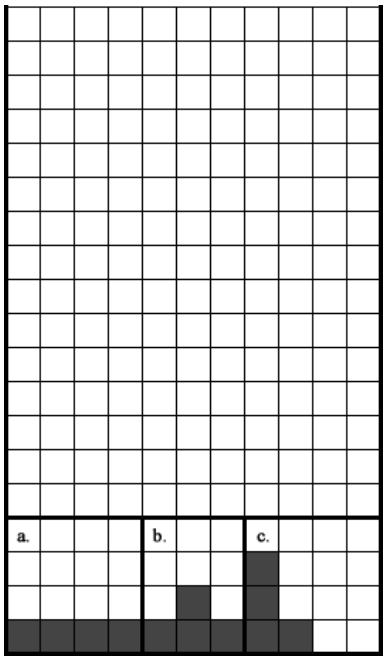

Figure 5: w=11 grid with 3 minigames

Executive Step

The executive step is the portion of the solution that is tasked with using the placement decision trees that are generated ahead of time to make decisions regarding the placement of the provided pieces.

It takes command line parameters regarding the Tetris piece buffer size, the method of traversing the Tetris tree, the input and output files, the width, buffer size, and debugging output.

We pregenerate several differently sized PDTs to create a solution that is more flexible to different grid widths. These sizes are 2x4, 3x4 and 4x4. Figure 5 demonstrates how a grid width can be broken down into separate independent "minigames" that are each played with their own copy of the appropriately sized PDT.

Piece Placement

Because we are playing several different minigames simultaneously in our ideal solution, it becomes necessary to examine how specific pieces are allocated to different minigames, and the most optimal strategy for performing this task.

Our solution provides two algorithms for making piece decisions. The first algorithm, dubbed the naïve utility algorithm (activated by passing `--method 1`) is a control algorithm that uses a similar version of the utility function that classifies the tree nodes to play the game directly. This algorithm is naïve in that it does not have any lookahead capability as the tree does, however, it does have larger capacity for the handling of placing pieces in holes that being accessible from row erasure.

The chief difference between the utility function of this algorithm and the PDT is that this algorithm receives a large negative utility for each row that exists on the screen, to encourage it to play the game optimally. This encouragement is not necessary in the PDT model as it has a hardcoded dimensional limit.

The second algorithm uses a depth first search to traverse the PDT tree, allocating pieces to each minigames in a round-robin fashion. Because of this fact, this algorithm also doesn't employ piece lookahead. This is not the most efficient approach, however it is less complex to implement.

We also initially planned to employ an algorithm to consider the list of input moves and attempt to allocate pieces (as far as the buffer will allow) to minigames as optimally as possible, called prescient DFS. For instance, if we are provided with 10 pieces, the algorithm may determine that pieces 1, 3, 5 and 6 are most suited to the first minigames, and pieces 2, 4, 7, 8 are most suited to the second minigames, and finally pieces 9 and 10 are best in the third minigames.

This strategy would allow us to attempt to maximise our ability to predict what pieces are to come when making decisions around them. However, we elected to continue down this path due to initial testing results discussed later in this report.

Theoretical Analysis of Solution

In this section, we will analyse different code sections to determine their approximate theoretical efficiency.

Pregeneration Code

In the pregeneration code, code readability took priority over efficiency. As this step is computed ahead of time, it was not considered a high priority to make it as fast as possible.

The overall number of combination of pieces that is generated and passed into the possibility function is r^t where r is the number of rotations of all the given tetrominoes, and t is the number of tetrominoes that fit in a specified area. Then each one of these combinations is offset by all possible height and width values on the Tetris grid. Using this function, we can calculate that the complexity of the number of combinations is approximately

$$O(r^{\frac{a_g}{a_t}})$$

where a_g is the area of the grid, and a_t is the area of a tetromino.

Possibility

The possibility code's job is to iterate over every combination of piece rotations and offsets to determine which ones of them are possible. It does this by creating a multiprocessing pool that receives all possible combinations of the maximum amount of pieces that can fit in the given area, plus any fewer number of pieces that could fit.

The amount of possibilities that must be checked can be defined as the following function. This function increases rapidly with n and m, thus making the problem computationally difficult as the simulation grid increases in size.

$$f(n,m) = \sum_{i=1}^{m} \frac{n!}{(n-i)!\,(i!)}$$

$m =$ maximum number of pieces that can fit in area, $m \in \mathbb{N}$
$n =$ number of different positions possible in a given area, $n \in \mathbb{N}$

For each possibility, the pieces are logical ORed in any order, and each value is checked to determine if there are any True values (indicating a piece clash) with a logical AND of the current grid and the new piece. These logical operations have an estimated complexity of $O(wh)$ (where w and h are width and height) as it has to loop through the values of both arrays to compare them.

As the possibility code has to compute at maximum of four AND operations and four OR operations, we can estimate this step's complexity to be approximately $O(f(n,m))$.

Validity

The validity code's job is to iterate over every permutation of possible pieces (returned from the possibility function above) and determine which permutations are possible. It is less important that the validity code is as efficient as the possibility code, as the numbers it is required to consider are substantially less. The validity code similarly creates a multiprocessing pool to handle this task.

The amount of possibilities that must be checked can be defined with the following function:

$$g(m) = \sum_{i=1}^{m} i!\,c(i)$$

$c(i) =$ number of given combinations with i pieces

For each possibility, the Tetris grid a piece is placed on must have its `adjacent` and `overhang` array calculated. Both these functions iterate over each array element with a $O(wh)$ complexity, thus giving the validity step $O(g(m))$ complexity.

Tree Generation

The tree generation phase of the solution involves iterating over every valid permutation provided by the validity function, as well as iterating to the top of the tree from each terminal node to calculate the maximum utility values.

Thus the tree generation phase can be considered $O(n)$ where n is the total number of nodes provided by the permutation phase.

Executive Code

In the executive code, efficiency took priority over readability, as the executive code must attempt to meet recommended speed parameters. As discussed above, two different algorithms are used to play the Tetris game.

Naïve utility

The naïve utility algorithm is the less efficient of the two algorithms when it comes to execution time. It plays the game by looping over each piece input and calculating the optimal location for a piece, when considering utility values. To calculate the most optimal move, it must loop over every piece that can be placed on this turn (equal to the size of the buffer) in every possible rotation and position on the board.

The number of iterations required to do this can be described as:

$$h(p) = v_r(p)\, wb$$

$p =$ piece or pieces involves
$v_r(p) =$ average number of rotations the given pieces have
$b =$ maximum size of the tetromino buffer
$w =$ width of the Tetris grid

DFS traversal

Round-robin depth-first search algorithm, by virtue of utilizing tree traversal, is vastly more efficient when it comes to execution time. Because the tree is computed ahead of time, tree generation considerations can be excluded from these calculations. Depth first search efficiency is a well understood problem; its generalized time complexity is described as:

$$b^m$$

$b =$ maximum branching factor
$m =$ maximum depth of the state space

In our solution, we can easily demonstrate that:

$$i(g) = \max(\, \text{area}(g_0), \ldots, \text{area}(g_n)\,)$$

$$m = \frac{i(g_{all})}{a_t}$$

$g =$ set of minigames

as the tree is only as deep as the largest number of tetrominoes that can fit in the largest minigames. Similarly, we can describe the upper bound of the branching factor as:

$$b \leq n_p\, v_r(p_{all})\, i(g_{all})$$

$n_p =$ number of pieces

as the number of branches is equivalent to the number of possible further actions that can be taken at a given time.

Experimental Analysis of Solution

Pregeneration Code

Experimentation on the various facets of the pregeneration code with a variety of different grid areas has demonstrated that of the four stages, three of them grow linearly with grid size, while one of them grows exponentionally.

In each of the graphs displayed on this page, the blue line represents the size of the data set generated from the input, while the red line represents the size of the data set once processed.

For some scenarios, the simulation became intractable on consumer hardware, so in leiu of real data for those scenarios, dotten lines represent the predicted trends.

Figure 6 demonstrates the calculation of the different rotations and positions on the grid for different Tetris pieces. The input data is generated on the fly from all combinations of pieces, rotations, width offsets and height offsets.

The output data filters the input by pieces that fit wholly in the given grid. This is clearly a linear process.

Figure 7 demonstrates the calculation of what combinations of pieces are possible. The input data is all combinations of a maximum of m number[2] of the output of Figure 6. The output is those pieces that pass the "possible" tests as previously discussed. This graph is on a log scale and clearly indicates the exponential nature of this growth.

Finally, Figure 8 demonstrates the calculation of what combinations of pieces are valid. The input to this function is a factorial function, and is thus growing much faster than the output.

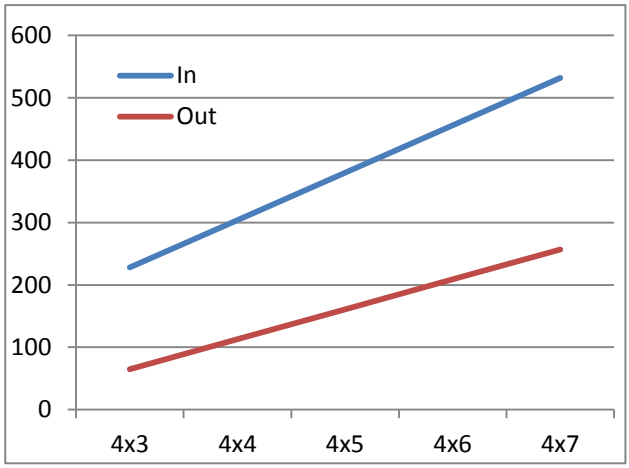

Figure 6: Combinations (linear scale)

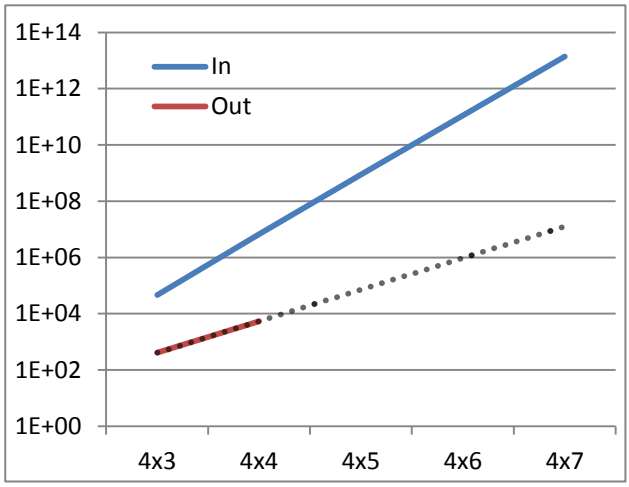

Figure 7: Possibilities (log scape)

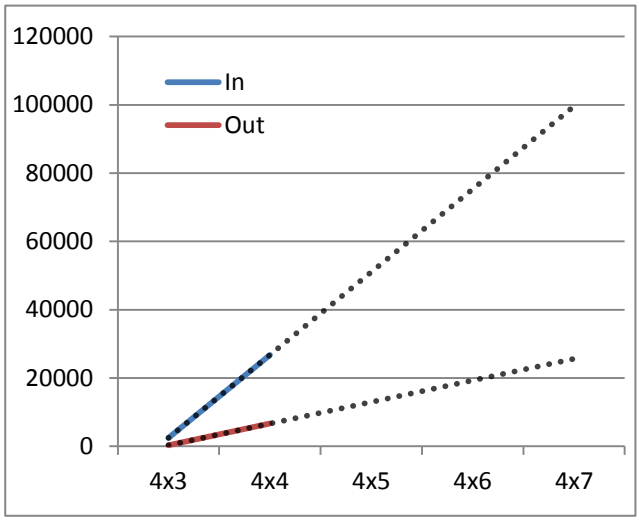

Figure 8: Valid possibilities (linear scale)

[2] For definition of m, see Possibility section in Theoretical Analysis of Solution

Executive Code

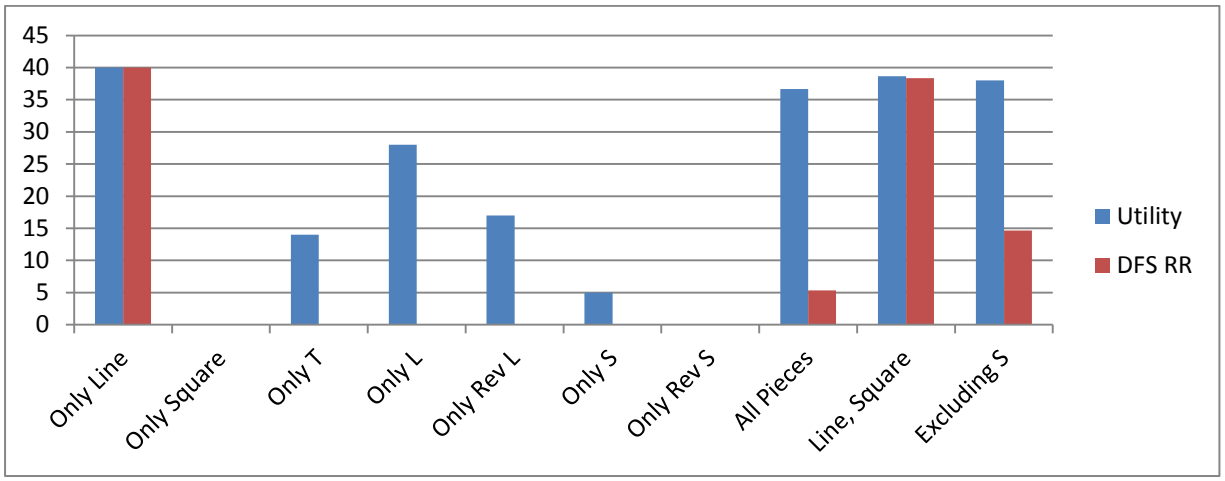

Figure 9: Comparison of rows cleared

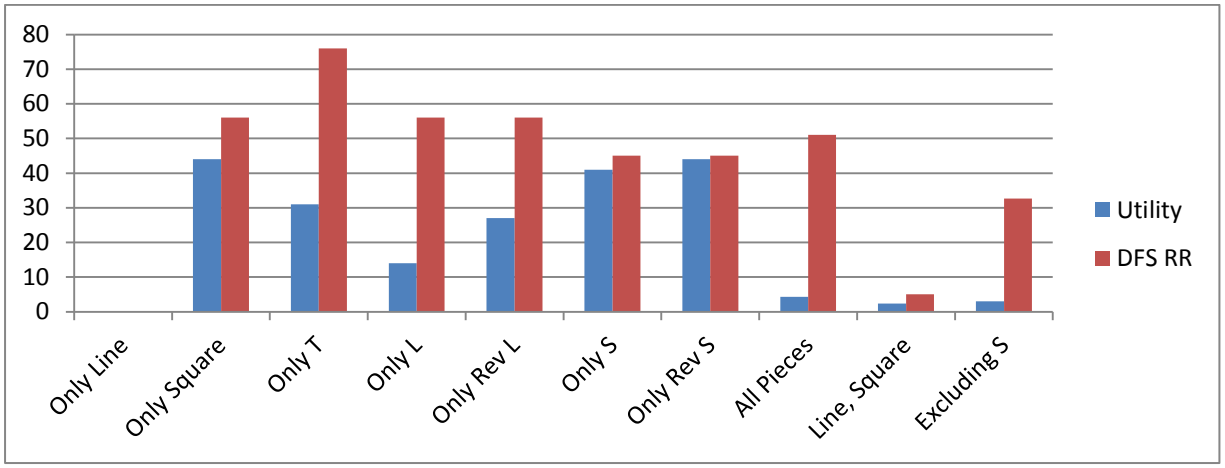

Figure 10: Comparison of final height

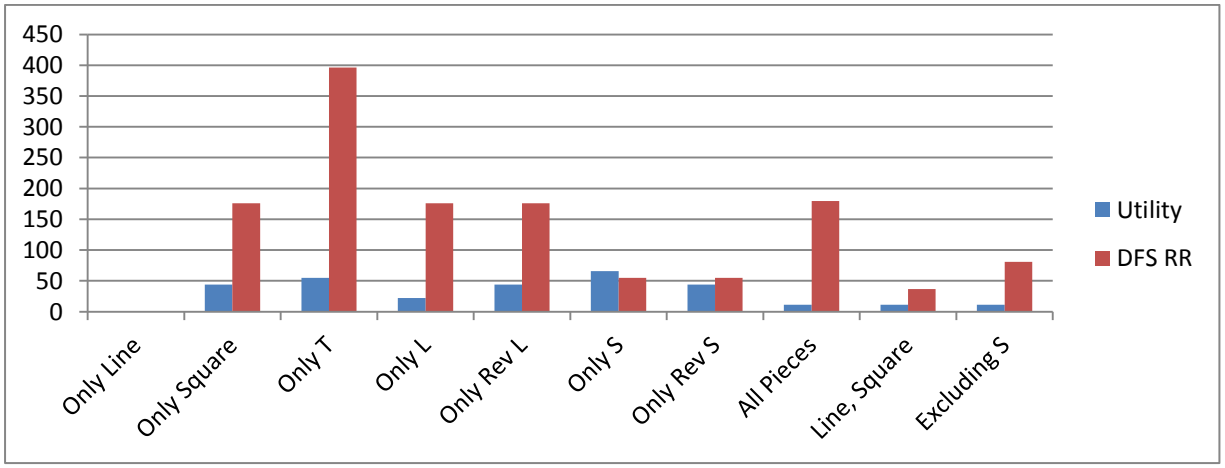

Figure 11: Comparison of number of holes

All graphs are averages of three different datasets with the same statistical distribution.

As these graphs clearly demonstrate, the naïve utility algorithm clearly outranks DFS RR in all aspects. We suspect that this is the case due to the fact that the inherent advantage of a PDT approach lies in the fact that its fast traversal can be used to perform piece lookahead. Without the utilisation of that functionality, the PDT is inferior in its ability to cope with row erasure and possible piece placements therein.

Reviewing the data, the utility function was clearly able to handle the curveballs inherent in giving the AI only one piece for the entire game. This is most likely because the utility function can make piece positions that require the creating of a hole, the erasing of a row and then the filling of a row.

Obviously however, neither function was able to cope with being presented with pieces that cannot be successfully tessellated with only themselves (e.g. S pieces).

It is interesting to note that the round-robin algorithm improves its performance significantly when the S pieces were excluded from the statistical distribution of pieces. As previously mentioned, this can probably also be attributed to the fact that S pieces require complex row erasure techniques to be efficiently tessellated with other pieces.

Prescient DFS

When examining prescient DFS, we elected to do some preliminary testing to determine if such an algorithm would bring improvement to our solution. Such an algorithm was intended to either return an optimum solution or no solution at all.

We found that for each loop (i.e. for each 11 pieces with $w = 11$) there was a 50% chance it would takes 10+ seconds to find no optimal solution, a 25% chance it quickly found no optimal solution, and 25% chance it quickly found an optimal solution.

To work effectively, the algorithm would require the input of another algorithm 75% of the time, and a complex multiprocessing element to allow graceful timeouts. Additionally, suboptimal moves would create gaps that the algorithm will need to manage. However, due to the design of the PDT tree and line erasure, the successful recovery and filling of these gaps would not be effectively handled.

Conclusion

In conclusion, we designed and implemented a solution that demonstrated two different approaches to designing an Artificial Intelligence to play Tetris. While theoretical testing indicated that a placement decision tree approach would yield superior performance results, practical difficulties in generating a robust tree resulting in an approach that was experimentally inferior to a more basic utility algorithm.

The inferiority of the PDT was not realized at an early enough stage of the project to permit a complete redesign, which is something to keep in mind for future projects. Further preliminary testing was needed to establish the algorithm's effectiveness before committing significant development time to the approach.

Definitions

- **Possible** – a combination of Tetris moves is considered possible if the tetrominoes involved do not inhabit the same space on the Tetris grid.
- **Valid** – a permutation of Tetris moves is considered valid if in a given ordering of tetromino placements, no tetromino is unsupported (i.e. it hovers in midair) in its intended destination and no tetromino is obstructed from reaching its intended destination by other tetrominoes (i.e. when falling it would hit another tetromino before reaching its final place).
- **Optimal** – a permutation of Tetris moves is considered optimal (and having a $+\infty$ utility) if it is possible, valid and entirely fills the specified rectangle without leaving any empty space.
- **Piece lookahead** – the act of peering into the future pieces to allow planning.