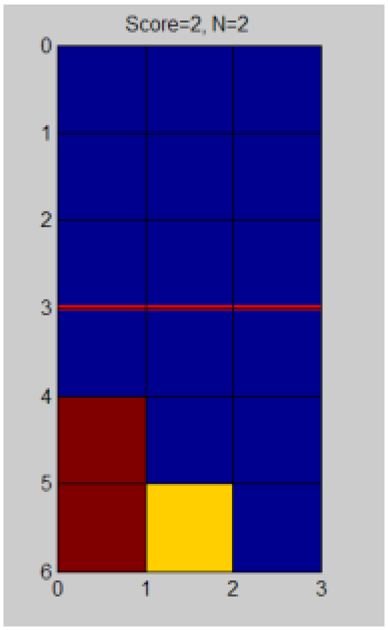
Tetris

Bryan DiLaura

ECEN 2703: Project #4

December, 2013



Introduction and Framing:

Tetris, released in 1984, is a popular video game, as well as being a very interesting basis of the third project for this class. A more simplified version of Tetris is being used for this assignment, utilizing only three different pieces (piece shapes will be described in more detail later), and a significantly smaller board (only 3x3). All of the other rules of Tetris still apply, once a row is completely filled with blocks, the row is eliminated, resulting in points. This project is an attempt to increase knowledge of dynamic programming and discrete outcomes, by attempting to create an optimal policy for this simplified game of Tetris.

In order to even begin to solve this problem, I had to frame it in a very similar way to the past two projects: as an optimal cost, routing problem. Tetris, having significantly more complexity than the node networks we were using before, was a little tricky to imagine in this way. The first thing that needed to be done, was how to set up the possible states of the system. With a 3x3 matrix representing the board, there are 9 possible spots where blocks can reside. Each space can either have a block in it, or not. This translates to 2^9 conceivable boards (with some of them not truly being possible, but they were kept in for simplicity's sake). Now, on top of this, there are 3 different pieces that can be played, so, there are 2^9*3 possible states. Finally, there is one last state, called the terminal state, which occurs when a piece goes beyond the top row, and loses the game. This means that there are 2^9*3+1 possible states that the game can be in. These states can be interpreted as nodes in the optimal routing problem.

Next, moving between states is defined as the new state is equal to the old state plus a decision at that state. For example, let's say the following state has occurred:

Current Board:

$$\begin{matrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{matrix}$$

Current Piece:

$$\begin{matrix} 1 \\ 1 \end{matrix}$$

There are many possible decisions that can be made using the current piece. The piece could be placed in any three of the columns, or could be rotated by ninety degrees and placed at the left or right of the board. Let's say for this example, the piece was going to be place vertically, on the left hand side of the board. The combination of this board, and decision on the piece results in the new board:

New Board:

$$\begin{matrix} \mathbf{1} & 0 & 0 \\ \mathbf{1} & 1 & 0 \\ 1 & 1 & 0 \end{matrix}$$

When talking about the optimal path routing, we were attempting to minimize the cost of working through a network of nodes. In this case, however, there is not a real cost to be minimized, rather there is a reward to be maximized: the number of rows eliminated. For example, in the situation describing the state transition, if the decision was made to place the block vertically on the right hand side a completely filled row would result. This would mean that row would be eliminated, and a score of 1 would be achieved. That decision would be the optimal policy for that board, as it eliminates a row. However, when looking at this problem, the rewards from not only the immediate decision need to be accounted for, but also the expected score moving forward in the game. To achieve this, the problem has to be framed looking at the 'destination' and moving backwards. This will result in the most optimal policy for playing the game.

This project is slightly different than the last project, in that the pieces are determined stochastically, meaning that the next piece coming in the game isn't necessarily known. This adds a layer of complexity, in that finding the most optimal placement for the current piece also has to do with the chances of what the next piece is going to be.

Code Structure:

For this project, two MATLAB functions were written. The function headers can be seen below:

[J, mu, rows] = tetris_policy_SN(N, P)

[decision] = tetris_play_SN(board, piece_num, iteration_num, mu)

Where:

- J is a 3D matrix with a 'rewards from this point' at each node, with each layer being associated with a piece
- mu is a 3D matrix describing the optimal decision that will get the game there, given the current piece (the layer of the matrix)
- rows is the expected number of rows eliminated
- N is the number of boards in the game (N-1 'stages' will be played)
- P is the probability matrix, describing the probabilities of each piece falling. This is of the format where rows is the current piece, and columns is the next piece. For example for the following probability matrix:

  $$\begin{matrix} 1/2 & 1/2 & 0 \\ 1/3 & 1/3 & 1/3 \\ 1 & 0 & 0 \end{matrix}$$

  If piece 1 is the current piece, there is a 1/2 chance piece 1 or 2 will be the next piece. If piece 2, then 1, 2, or 3 comes up with equal probability. If piece 3, piece 1 always comes next.

- decision is the optimal decision at that node

- piece_num is the current piece (1, 2, or 3)
- iteration_num is the current number of turns in the game so far

The Tetris policy function is where the meat of the calculations are happening, so my explanation of will focus there.

The first thing that had to be figured out, was how to generate all of the possible states. This was done by a trick using binary. All of the 512 (2^9) possible states can be represented through binary numbers, nine digits long. Binary numbers 000 000 000 through 111 111 111 were generated, and then converted into 3x3 matrices. These are all of the possible boards. Boards could also be converted back into a decimal number (which is how I referred to the nodes in my code) and vice versa using this trick. The 513th state (the terminal state) was added by hand.

Next, the full sequence of pieces was created using a for loop and a simple counter, iterating through sequence until the end was reached, then starting at the beginning. This was done until N was reached.

A board – decision evaluation function was written next. This function took in the current board state, and a theoretical decision, and evaluated what the outcome of this decision would be, including if the terminal state was reached, the number of rows eliminated, and what the new state of the board would be. This was done by adding the two matrices together, and seeing if there was any overlap (if any 2's were present). If that was the case, the decision was shifted up one block, and it was tried again. This was done until either the terminal was reached, or an allowed position was found. The result of this addition was then checked to see if there were any rows that were filled (and should be eliminated). If there were, they were eliminated, the above rows were moved down, and a counter for the rows eliminated with that decision was incremented.

Now that all of these helper functions were created, the real workhorse of the policy function could be written. This is a nested for loop structure, walking its way through the stages (starting at the last one, and moving backwards), in each stage, evaluating every node, and at each node, evaluating for the three different possible pieces. Depending on what the current piece is, for each node, up to 8 orientations would need to be evaluated against the current board (the node). When this evaluation was done, the number of rows for each decision, was added to the expected rewards would be in the resulting board (in the J matrix, of the next stage). The expected return was the value of the J matrix, multiplied by the probability of getting that piece. Whichever decision maximizes immediate and future rewards, is stored in the mu matrix. The number of rows that would be eliminated, plus the expected eliminations, is put into the J matrix. This is done for every node, in every stage.

Finally, the number of rows that will be eliminated is in the first node, of the first stage (the empty board) in the J matrix, of all three layers. In order to come up with a single, expected number of rows to be eliminated, these three values were multiplied by 1/3, and then added together. This was done because the probability matrix doesn't say anything about what the first piece will be, so I assumed there would be a probability of 1/3 of it happening. With that, the policy function was complete[1].

For the play function, because the optimal decision was imbedded into the mu matrix, it is only a matter of indexing the correct node in the matrix. The layer is determined by the piece number, the row is determined by the iteration_num, and the comes from the board, which is converted from a board to a decimal number (as described earlier).

When both of these functions are used together, first calling policy for the specified sequence and N, then calling play for each member of the sequence (using the mu matrix from policy), Tetris can come to life, and be played.

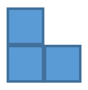
Code Analysis:

In order to show that my code is working properly, I will analyze a few piece sequences by hand, saying what the number of rows that should be, if played optimally, and then run my code to see if it gives the same response. I will be using the probability matrix as follows, to give a deterministic sequence of pieces:

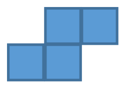
$$P = \begin{matrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{matrix}$$

This will give the sequence 1,2,3,1,2,3… and the starting piece can be changed as a variable. The pieces are defined below:

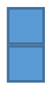
Piece 1:



Piece 2:



Piece 3:



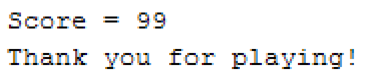
This sequence has the property of repeating in a convenient way. When I analyzed the pattern (starting with piece 1) I found that the best policy was to do the following sequence (I put them in matrices for easier description relative to the board):

[1] *It is worth noting, that my policy function, that is very slow. The code could potentially be optimized further by, for example, cutting out impossible boards, using more efficient code structure, vectorizing the code, or rewriting built-in functions for speed, however, in the interests of time, it was left in its partially-optimized state. I added in a progress bar to visually give the user feedback that it is working, and some idea of how long it would take to calculate.*

Stage 1:			Stage 2:			(rows = 1)			Stage 3:			(rows = 3)		
0	0	0	0	**1**	**1**	0	0	0	0	0	0	0	0	0
0	0	**1**	**1**	**1**	1	0	1	1	**1**	1	1	0	0	0
0	**1**	**1**	0	1	1	0	1	1	**1**	1	1	0	0	0

This policy can then loop, assuming that the number of stages will be divisible by 3, there will be an equal number of rows eliminate to the number of stages. For this case (N=100), 99 stages will be played, so this sequence will be repeated 33 times, resulting in a score of 99.

When running my code, I got the following result:



```
Score = 99
Thank you for playing!
```

So, my code works for this example sequence! Great! However, something fairly interesting happened. The graphical analyzer gave a very different sequence than the policy described above, having a setup, a sequence repeated a bunch of times, and a finish, all of which were slightly different. The number of rows that it achieved was correct, however the policy simply looked different. This happens because the program choses the first one it comes across that maximizes the rows eliminated, not necessarily in the simplest way. Because of this, a more complex policy can result, but still give the same score.

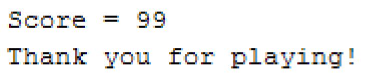
The next sequence that will be analyzed uses the same probability matrix, but starting with piece 2.

The optimal policy for this board is one row per stage. The by-hand sequence that gives this policy can be seen below:

Stage 1:			Stage 2:			(score = 1)			Stage 3:			(score = 3)		
0	0	0	**1**	**1**	0	0	0	0	0	0	0	0	0	0
0	**1**	**1**	**1**	1	1	1	1	0	1	1	**1**	0	0	0
1	**1**	0	1	1	0	1	1	0	1	1	**1**	0	0	0

Very similar to the last sequence, as long as the number of stages is divisible by 3, there should be 1 row elimination per stage (99 rows).

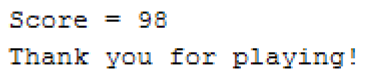
When the program is run the following is outputted:



```
Score = 99
Thank you for playing!
```

So the program is working correctly for this sequence too! Again, it didn't follow my manual policy, but it still got the correct answer.

With two examples both getting the optimal policy correct, it is fairly safe to assume that the code is working relatively well. One of the benefits of writing code to calculate the optimal policy is that it can calculate it for sequences that may be very difficult to analyze by hand. For example, I have no idea how to calculate the number of rows that should be eliminated by the sequence 3,1,2, but I can run it through my code, and get an answer that I can say with reasonable certainty is correct. When I ran this sequence through the program, I got the following output:



```
Score = 98
Thank you for playing!
```

The output of 98 rows is quite high, meaning that there is only 1 move used to setup, and the rest are getting one row per stage. This is nearly a perfect policy, so I trust its correctness.

After analyzing these three different sequences, when looking at the deterministic sequences, the function is working correctly.

Stochastic sequences

So far, there hasn't really been that much different from the last project. The main difference in this code, is that it can now handle a probabilistic determination of pieces falling, rather than a deterministic sequence of pieces. In order to test this functionality, I will be using the following probability matrix:

$$P = \begin{matrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{matrix}$$

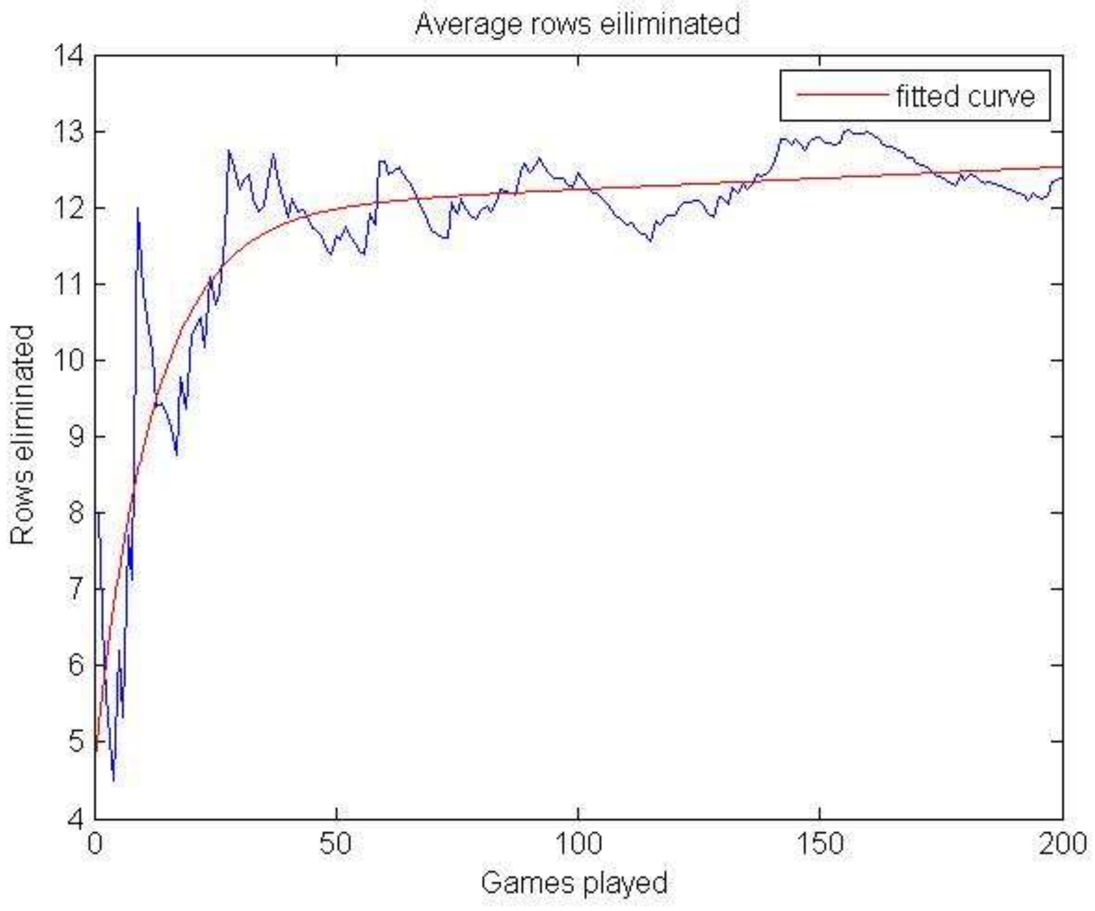
This means that each piece has an equal chance of coming up each turn.

When using the stochastic Tetris, and an N of 100, the expected number of rows to be eliminated are as follows (depending on the first piece):

- If the first piece is piece 1: 14.9 rows are expected to be eliminated.
- If the first piece is piece 2: 10.1 rows are expected to be eliminated.
- If the first piece is piece 3: 14.3 rows are expected to be eliminated.

This means that starting with piece one will give you a higher expected number of rows to be eliminated, and optimizes the cost to go in this case. If you take the expected value of these three (giving the expected value overall, given the fact that the first piece is randomly selected as well) we would get 13.1.

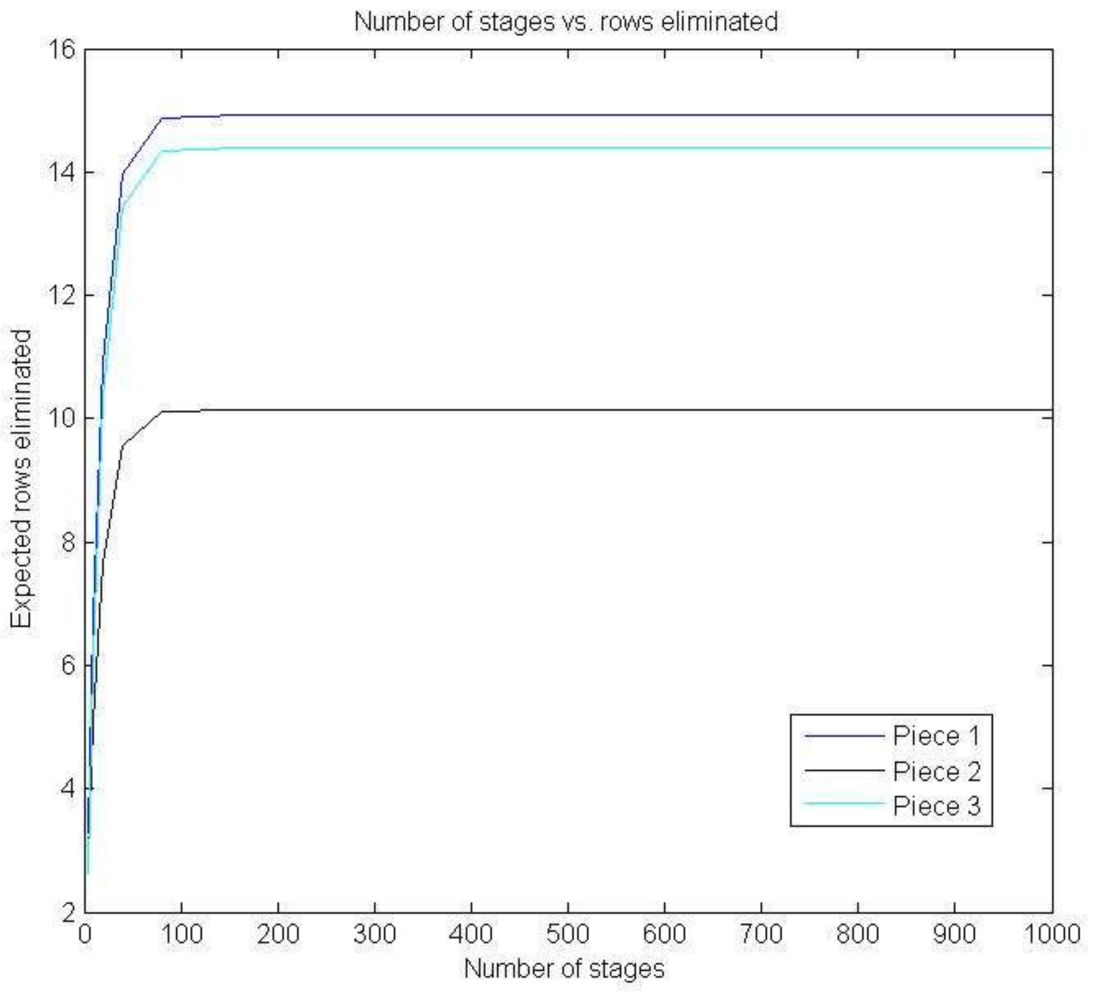
These numbers are expected values, which means that if the game was played a large number of times, the average number of rows eliminated would approach these values. In order to test this, I ran my code 200 times, and checked to see if the average went as expected (approaching 13.1, the expected number of rows eliminated given randomly selected starting pieces). I graphed the results, and put in a best fit line for the curve, seen below:



As can be seen from the best fit line the average number of rows eliminated does exponentially approach around 13. If this were to be done upwards of a thousand times, I'm certain that the curve would asymptotically approach the expected value. Notice how the average number of rows eliminated fluctuates wildly. This is due to the variability in the chances of getting pieces. Some games will score higher by chance, and others will score lower by chance. Just by sheer probability, the average will vary.

This graph matches up with my expectations of this relation. It makes sense, as after a certain amount of time, you will get an unlucky draw for pieces, and the game will end. With more iterations of the game, it should even out to some number of rows that can be eliminated based on the probability.

The number of rows eliminated seems to be correlated to the number of games that are played. Below is a graph that shows this relation:



As can be seen in the graph, as the number of games that are played increases, the expected number of eliminations increases, and then eventually levels out. This makes sense, as once again, eventually an unlucky piece sequence will happen, and just by pure statistics, you will lose.

Man vs. Machine

The ultimate goal of writing a program like this is that it would be able to beat a human at the game. To test this, I played 10 games of Tetris, and recorded the number of rows I eliminated. These trials can be seen below:

Game #	1	2	3	4	5	6	7	8	9	10	Average
Score	17	14	7	4	3	3	15	21	6	2	9.2

As much as I hate to admit it, the computer beat me. My average was slightly lower than the average the computer was able to do, and if I were to do more trials, I would guess that I would continue to be below the computer's expected eliminations. Unlike me, computers don't make any mistakes. However, my failure is not entirely in vain, as it means that the program is working as it should.

Conclusions

By framing Tetris as a shortest-path-routing problem, I was able to write a function that is capable of finding the optimal policy for placing stochastically selected pieces. This code was analyzed, and found to be working with exceptional precision. Now that the code is written, a sequence of any combination of pieces, order, or length could be evaluated, and the optimal policy could be obtained.

One rather large disadvantage of my function in its current state is that it is unreasonably slow. Finding the expected number of eliminations for 1000 stages took literally over an hour. Given more time, I would try to optimize the code, allowing more stages, or even in the further future, larger boards and more pieces, to be analyzed.