AI Block Battle

Final Report

Carter Tiernan
27151114
May 5, 2016

1. Introduction

1.1. Classic Tetris

The fundamental aspects of Tetris have stayed relatively constant since its initial release in 1984 by video game developer Vladimir Pokhilko. It is played on a rectangular field built of cells that all begin empty. The game is composed of rounds, each of which starts when one of seven game pieces, called a "Tromino", is randomly generated at the top of the game field. Each Tromino has a different shape and can be classified by the Roman letter it most highly resembles: "I", "O", "J"," L", "S", "Z", and "T" (**Figure 1**). During a round the player can rotate and move the Tromino around the board; however, movement is restricted to left and right and down meaning a Tromino cannot traverse up the game field. The player's objective is to stack the sequentially generated Trominoes to create complete rows of filled cells on the board. When a row is completely filled it will break and all cells above it will fall down to occupy the newly emptied row (**Figure 2**). The game is over when the stack of Trominoes reaches the top of the field.

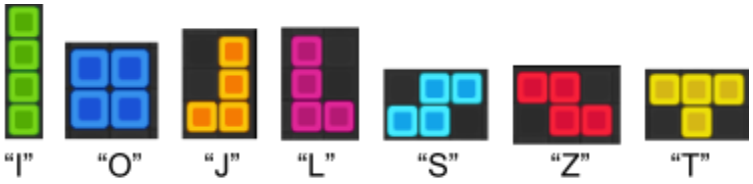


Figure 1
The seven Trominoes

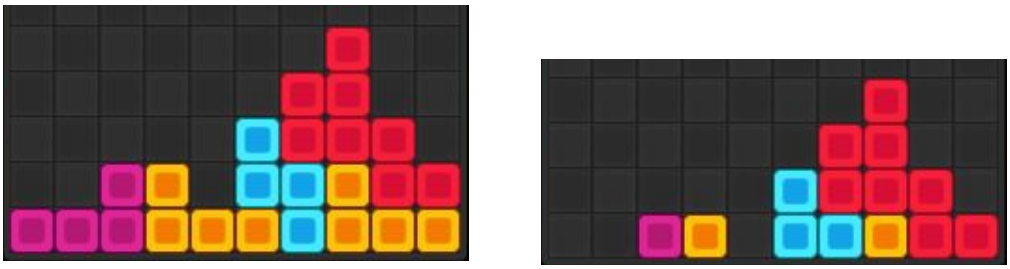


Figure 2
Completing a line

1.2. Nintendo's Two Player Tetris

One of the most influential alternatives to classic Tetris was developed by Nintendo in 1989 and featured a competitive play mode. Competitive play mode added many new features to the game, most centrally was it being a two player version of Tetris. During competitive play, two players simultaneously stack the same sequence of Trominoes following very similar fundamentals as classic Tetris. Competitive play begins to diverge from classic Tetris with the addition of features such as garbage that spawn on the bottom of a player's field when their opponent gains points and

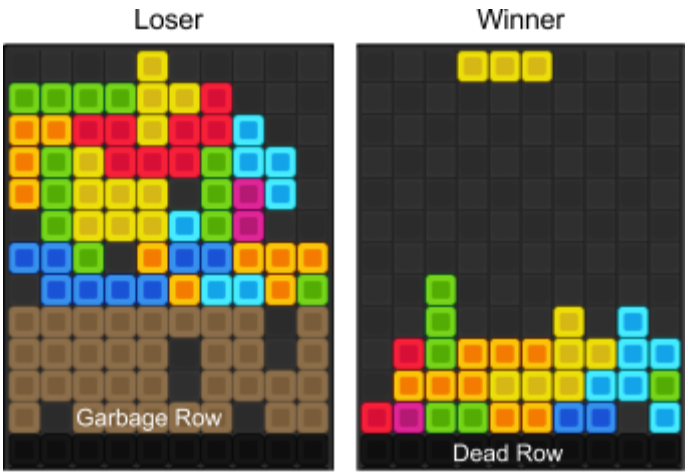


Figure 3
An example of fields at the end of two player Tetris, the left is player 1, who lost the game, and the right player 2, who won the game.

dead rows that spawn as time passes during the game. As in classic Tetris, when one player's stack of Trominoes reaches the top of their field they lose the game and the other player wins (**Figure 3**).

1.3. Other Tetris Implementations

Additional features are commonly found in Tetris implementations. Combo points can be awarded to players that consecutively place Trominoes in point getting positions during multiple rounds in a row. Bonus points can be awarded for the amount of lines cleared during a round-- a player can clear up to four rows using an "I" Tromino. Moreover, bonus points can be awarded for performing complex moves to complete lines, such as a "T"-spins (**Figure 4**). One final example of additional features added to Tetris are powerful skip moves that allow players to skip placing a Tromino that would not fit in their field well.

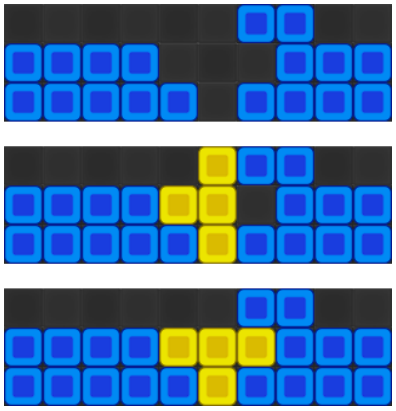


Figure 4
An example of a double T spin, rotating a "T" Tromino and clearing 2 lines.

1.4. Strategy

The fundamental strategy of Tetris is to fill up as many rows as possible while keeping the stack of Trominoes from reaching the top of the board. However, competitive play and the addition of more features to classic Tetris creates a complex and nontrivial game environment to strategize in.

1.4.1. Number of Possible Moves

Trominoes have a maximum of 4 rotations and 10 horizontal locations on the board, therefore planning ahead two Trominoes requires ranking a maximum of 1,600 board configurations.

$$(rotations\ of\ first\ Tromino\ \times possible\ locations\ of\ first\ Tromino\ )\ \times$$
$$(rotations\ of\ second\ Tromino\ \times possible\ locations\ of\ second\ Tromino\ )\ =$$
$$(4 \times 10\ )\ \times (\ 4 \times 10\ )\ =\ 1,600$$

However, this is largely dependant on the current and next Tromino, if the both Trominoes are “O”s there is only a total of 64 board configurations-- the lowest amount of board configuration of all other Tromino pairs.

$$(1 \times 8\ )\ \times (1 \times 8\ )\ =\ 64$$

Regardless, the many possibilities in which a player can stack their Trominoes leads to differing strategies upon where and how a Tromino is placed.

1.4.2. Examples

A simple strategy for a Tetris agent is to clear as many lines as possible. This could allow an agent to outlive an opponent by never taking risky moves to make higher point valued moves later. However, in some implementations clearing a single line does not give a player points. This makes the simple AI less effective because it will not take more risky moves to set up point getting field configurations.

Another, more aggressive, strategy would be to take advantage of bonus points and garbage rows by creating tall uniform stacks with a few deep valleys used to clear many rows with an “I” Tromino (**Figure 5**). By clearing multiple lines one’s opponent’s field will spawn garbage rows.

More complicated strategies include trying to create a stack with such figure as to allow “T”-Spins, or to intentionally not clear rows but set up more to be cleared later so combo points can be received.

Figure 5
Two valleys

2. Methods

In the most general sense, what an agent needs to do in order to play Tetris is output a list of commands describing actions to perform on the currently falling Tromino. The initial implementation of my agent created this list by randomly selecting actions from the set of valid actions, “left”, “right”, “turnRight”, “turnLeft”, “down”, “drop”, and always appended “drop” at the end.

In order to create a agent that made informed decisions I broke the planning of actions into distinct steps. The first step is to map all the possible board configurations that are created by every rotation and final position possible for the current and next Tromino. Then, for each board configuration a ranking is calculated and recorded along with the possible field. Lastly, my agent builds an action list describing how to move the current Tromino into the correct position.

2.1. Finding Board Configurations

2.1.1. Successful Implementation

My initial and most simple implementation of my agent took into account only the first Tromino. Additionally it used the method provided in the starter bot *projectPieceDown* to find valid Tromino positions. There were two main issues with my bot at this stage: its lack of long term planning, and unoptimized code. By only taking into account the first Tromino my bot could not plan very far ahead meaning it would make suboptimal plays due to its lack of knowledge of the next Tromino that would fall. However, the reason I only took into account one Tromino was that at this point my code was extremely unoptimized and took approximately seven to eight seconds to make a decision if both Trominoes were taken into account. One of the biggest problems of optimization was the *projectPieceDown* method (**Figure 6**). This method is used to see where a Tromino would be if it were dropped from its current location. It functions by testing if a Tromino fits from the top of the field all the way down to the bottom, or when it cannot fit anymore because it overlaps another Tromino.

```
def projectPieceDown(self, piece, offset):
    piecePositions = self.__offsetPiece(piece.positions(), offset)

    field = None
    for height in range(0, self.height-1):
```

Figure 6
A clipping of projectPieceDown, showing the for loop that tests all piece positions from the top to bottom of the field.

By using projectPieceDown every Tromino position from the top of the field down would be tested adding as many as 20 method calls and creating an entire 20 by 10 Tetris field every time my agent wanted to test a potential Tromino position. In order to reduce the repetition of this method call I had to create a more complex and accurate search algorithm for placing the Trominoes. I did this by exploiting features of the Trominoes, the current field and other game knowledge. The initial step was to stop finding the drop position of a Tromino by testing the top row down, so I implemented a method, *get_col_heights*, that calculated all the column heights so I could initially place the Tromino where it should fit-- on the top of the column it is in. Further optimizations of the heuristic such as horizontal limits describing where the Tromino could fit on the game field and xy coordinates defining where the lowest part of the Tromino was in relation to its center were useful

```
def projectPiece(self, piece, offset):
    piecePositions = self.__offsetPiece(piece.positions(), [offset[0],offset[1]])

    field = self.fitPiece(piecePositions, [0, 0])

    return field
```

```
# get the hieght of the column (to initially test at this hieght, not every height higher
col_to_check = piece_x + (self.memorized_field.width - piece_x_check_pos[1])
col_height = self.col_heights[col_to_check]

piece_y_lims = piece.get_y_pos_lims()
y_min = field.height + piece_y_lims[0] - col_height
```

Figure 7
The top code snippet is a new method *projectPiece*, it functions like *projectPieceDown* but does not loop. The location is determined using calculations such as the bottom code snippet.

during optimization. With these additions my agent could successfully guess where the Tromino would fit in a column within a few tries (**Figure 7**).

2.1.2. Unsuccessful Implementation

One major issue I had was trying to keep track of the current field and all the possible fields. This became especially important when I realized my bot was not accounting for completed lines, trash rows, or garbage rows. When using *projectPieceDown* it was not very important to account for these changes in the game field because the agent is going to drop the Tromino from the top of the board and therefore the height is not of much importance. However, when I started to do more complex calculations of possible field configurations I needed to keep track of these garbage and trash rows so my agent could correctly place the Tromino. My first attempt at a fix was to clear full lines on the memorized field myself, but this was extremely unreliable and turned out to cause more issues than it solved. I found similar things when trying to account for garbage and trash rows myself. Fortunately the fix was to simply re-clone the current field each round.

2.2. Ranking Board Configurations

Defining what a good field configuration in Tetris is difficult. Simply passing an agent the field configuration would constitute too large of a state space to reasonably acquire enough data to make decisions upon. Therefore features are extracted from the field as measures of its "goodness".

2.2.1. Features

i. Total Height

Total Height is the summed total of all the columns' height. A column's height is defined as its vertical distance, in cells, from the bottom of the field, including covered empty spaces.

ii. Completed Lines

Completed Lines is the total number of filled rows on the field.

iii. Point Getting Lines

Point Getting Lines computed as a discounted value of Completed Lines by 1.

iv. Bumpiness

Bumpiness is the summed total of the absolute difference of each column and its right neighbor's height.

v. Holes

Holes is the total area, in cells, of empty cell that are covered by a full cell.

vi. Valleys

Valleys is the count of columns that have a height of 3 or less compared to its neighbors.

vii. Has Valley

Has valley is a three value variable computed from the value of Valleys.

Valleys > 1	⇒	Has Valley = -1
Valleys = 0	⇒	Has Valley = 0
Valleys = 1	⇒	Has Valley = 1

2.2.2. Score

In order to rank possible board configurations, I used a composite score calculated as the weighted sum of a field's features.

$$Score = (totalHeight \times totalHeightWeight) + (pointGettingLines \times pointGettingLinesWeight) + (bumpiness \times bumpinessWeight) + (holes \times holesWeight) + (hasValley \times hasValleyWeight)$$

Features are extracted by analyzing each cell starting the bottom right of the field and traversing up each column sequentially. The weights are learned via a genetic algorithm discussed in section 2.4. When a board configurations score is computed I put its data onto a heap. I used a heap data structure to make retrieving the most highly ranked fields iefficient as the heap sorted on insertion by the score of the board configuration.

2.2.3. Example

Figure 8
An example of a field's features. White lines are valleys, circles holes and the numbers represent column height.

Using **Figure 8** as an example of a possible board configuration, its features would be as follow.

i. The *Total Height* of the board configuration is *33*, the summed total of all 10 column heights listed as white number on the top of each column in **Figure 8**.
$$totalHeight = 33 = 1 + 4 + 4 + 4 + 4 + 4 + 4 + 5 + 3 + 0$$
ii. There are no completed lines on this board so *Completed Lines* is *0*
iii. *Point Getting Lines* is -1.
iv. The *Bumpiness* of this field is *9*
$$bumpiness = 9 = |1 - 4| + |4 - 4| + |4 - 4| + |4 - 4| + |4 - 4| + |4 - 4| + |4 - 4| + |4 - 5| + |5 - 3| + |3 - 0|$$
v. There are a total of four covered cells on this field, shown as circles in **Figure 8.** Therefore *Holes* is *4*.
vi. There are two valleys, marked as lines in **Figure 8**. Therefore *Valleys* is *2*

vii. *Has Valley* is *-1*.

2.3. Building Move List

Initially, when using projectPieceDown, building the correct list of moves to place the falling Tromino was trivial. Since projectPieceDown returns the position of the Tromino if it were to be dropped from its current position the agent only had to perform the correct number of rotations and horizontal movements then drop it to get the Tromino in the desired position.

When I began using projectPiece instead, column heights, and Tromino shape to decide where a Tromino could fit on the board, building a valid move list became a much more complex task. By using projectPiece my agent will find valid Tromino positions that are tucked in holes of the stack or positions that require rotations to be accessed, such as “T”-spins. To create these more complex lists of moves (**Figure 9**) it was extremely inefficient to do an directed downward search. Instead I created a graph abstraction of the Tetris board using best first search to quickly find a valid path, from final position to initial then translating the moves. I chose best first search because it is not extremely important that the Tromino takes the shortest path, as A* search would find, but instead that the Tromino finds a path fast. Best first search gave my agent this speed because most of the time moving upwards and directly toward the initial position is ideal in Tetris.

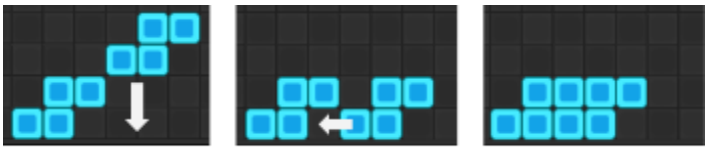


Figure 9
An example of a more complex move list: “down”, “down”, “left”, “drop”

2.3.1. TetrisGraph

Before explaining how TetrisGraph works it is important to know that every Tromino is placed within a 4 by 4 box I’ll refer to as a bounding box (**Figure 10**). This box is not visible to the agent and it only collides with other blocks and the edges of the board where the filled portion of the Tromino exist within it. This provides a useful abstraction of a Tromino that TetrisGraph takes advantage of in order to treat every Tromino uniformly, because if the bounding box fits in the game field the Tromino will as well.

Figure 10
A “L” and “Z” Tromino within the bounding box

TetrisGraph is fundamentally composed of nodes, each node representing a valid position the bounding box can be placed. All nodes are initialized with a set of valid actions, list of completed moves, and functionality to select the best action and perform an action (**Figure 11**). Ranking of the best action is calculated by minimizing the distance the Tromino is from its initial position. Horizontal and vertical distance are computed using Euclid's Algorithm, and rotational distance is computed as the difference

```
class Node:
    def __init__(self, pos, rot, moves, last_node):
        self.actions = ["turnLeft","turnRight","left","right","up"]

        self.posistion = pos
        self.rotatition = rot

        self.moves = moves

    def make_action(self, action_str, piece, field):

    def get_best_action(self, desired_pos, desired_rot):

    def is_dead(self):
        return len(self.actions) == 0
```

Figure 11
A overview of a TetrisGraph Node, notice up is listed instead of down. This is because TetrisGraph searched from the end position to the intitial position.

in the Trominoes rotation index. When a node performs an action it removes it from its move set, tests the new position to see if it is valid then returns a new node with an updated move list. If the action produced an invalid node the same node will be returned and the move list is not updated. If a node does not find any valid actions it runs out of moves to choose from and become “dead”. Dead nodes are not traversed to and search will begin from the node that found the previous node. If a node performs an action that returns the initial position of the Tromino the move list is translated into valid server commands and returned to my bot.

2.4. Genetic Algorithm for Feature Weights

My first agent was derived from Yiyuan Lee’s (Near) Perfect Tetris AI and used the features he described, thus I was able to use the optimized feature weights he calculated. However, in my later implementations I added additional features and tweaked others to account for differences in the types of Tetris our agents were playing. Therefore I needed to learn updated weights that took into account this new information, I did so much like Yiyuan Lee and wrote my own genetic algorithm.

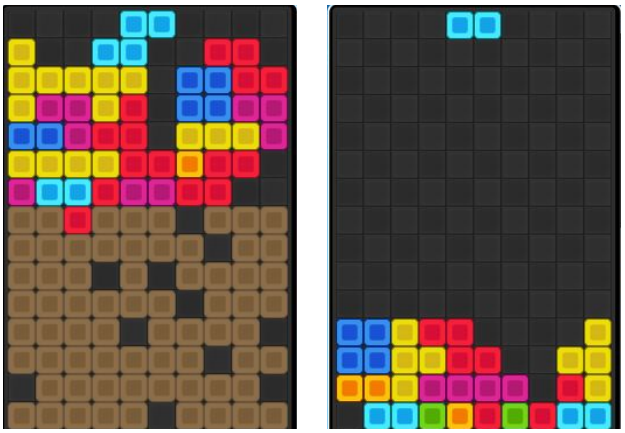


Figure 12
An example of how aggressive but tactical point getting can drown the opponent in garbage rows

One serious setback I faced when implementing the genetic algorithm was the difficulty to retrieve data about the agent's’ performance and the impossibility of truly testing my agent against competition during the training phase. When testing my agents I could read data about the game from text logs, but many aspects of the game are missing such as points earned each round and even what agent won the game. Along with time constraints, this forced me to choose the one readily accessible feature about the game I could, game length. I did so by recording the amount of rounds two of my agents played, assuming that agents that played longer would be more highly optimized. However, by using game length as my feature for fitness I missed a lot of data pertaining to the performance of my agent. For instance, knowing how many points an agent got per round would be very meaningful because in competitive play getting points generates garbage rows on the opponent's field. So faster games in which an agent’s strategy is to fill their opponents board with garbage rows can be argued as a higher performance strategy than endlessly clearing single lines, thus never sending garbage rows (**Figure 12**).

```
# generate a random index
# that is more likely to be lower than higher
idx = int(random.triangular(0,0,len(couples)))
```

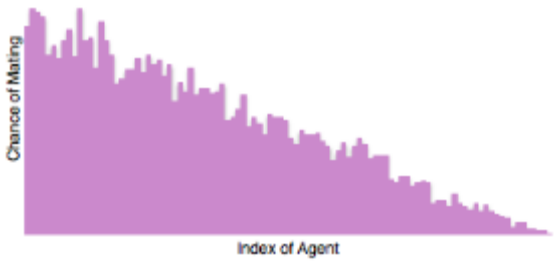


Figure 13
The call to get a parent index and an example of a triangular distribution

2.4.1. Botvelution

In order to learn feature weights a created a simple genetic algorithm I called Botvelution. I start the process by generating a small population of agents, each agent described as an array of randomly generated feature weights. After initializing the population the the agents are tested by selecting random couples to compete against each other in a game of Tetris. Once the game is complete the game log is scraped to record the length of the game. This process is repeated for every couple. After all couples have completed, the mating and mutation phase begin. Couples are ranked and ordered by length of game in descending order, and the top 10 performers are kept and survive through to the next generation. The rest of the population is generated by picking two parents from the ordered list using an index selected from a random triangular distribution (**Figure 13**). By using a triangular distribution lower indices are more likely to be chosen, and therefore better performing parents mate and pass on their optimized genes more frequently. When two agents mate, they swap a portion of their feature arrays with 90% probability, and otherwise do not swap any features. After the children have been generated, a mutation of each of the child's features occurs with 1% probability. Once all mutations are complete the 10 survivors and children are used as the new population and the testing phase is repeated.

After running my genetic algorithm, the best performing agents could compete against themselves for approximately 40 rounds. Their parameters were slightly different from each other but the most optimal parameters I found are approximately:

Total Height	*-2.94*
Point Gettings Lines	*1.88*
Holes	*-1.1*
Bumpiness	*-0.54*
Has Valley	*1.44*

3. Results

3.1. Performance

3.1.1. Winning the Tournament

Due to the competitive environment of the AI Games the best measure to rank performance is to win games in the competition. Unfortunately, this is a difficult measure to get data on because online games are very infrequent and only one's own bot's game logs are available to them. This makes there be very little real data to train from, so I had to turn to local instances of the server competing against myself. As stated earlier this is an unfortunate reality because it limits the amount of data to analyze and does not accurately simulate the true tournament environment.

I found that one of my best performing bots was my initial implementation that used the projectPieceDown method and only accounted for the currently falling Tromino. I think this is because with a simple bot one is able to fully understand its decisions and optimize it quickly. Additionally, because I was using pre tested features and weights I was confident that the scoring function was accurate. As I began to complicate my bots decision making it also became more complicated to know exactly what it was doing. For instance, at one point my bot was not checking the first column as a valid place to put Trominoes. At first I thought this was because it wanted to put a "I" Tromino there and get clear four lines, but I realized otherwise when debugging. Also, by using a more complicated path finding method I risked having errors and inefficiencies in my code that can cause my bot to not find a move list, find false positive valid move list, or not find a valid move list that exists. All of these complications can cause my agent to make mistakes or wrong decisions, either placing a Tromino incorrectly or missing a good move and therefore its competitive advantage.

One large blocking point I had was trying to optimize the feature weights. On top of all the points of difficulty already mentioned, the algorithm itself is not optimized either. I do not know what an ideal population size is, ranges for each feature, how to more accurately score the parents, implement mutation rate annealing, or other done any other optimizations. Ideally I would like to research which features are useful, which are not, and find new helpful features I have not thought of yet, but the time scope of this project did not allow me to do so.

My agents performance in this measure was not fantastic. I am ranked 200+ on the global scale, but 2nd within our institution (**Figure 14**). Many of the extremely high performing bots have gone through many iterations and are written in more optimizable languages, so I do not think my overall performance is inadequate, although I would like to have had more time to see if I could have made my bot better.

Figure 14
My bots rankings and information on The AI Games

3.1.2. Finding the Best Move

I spent a large amount of time on this project working on optimizing my bots algorithms so it could look two Trominoes ahead and make decisions in the allotted time of half a second. Therefore a more optimistic measure of my bots performance is ability to analyze the game.

My initial implementation took 7 to 8 seconds to make a decision when accounting for two Trominoes. When optimizing my agent I used quick calculations I had available to me such as column height, and prior knowledge such a Tromino's location within the bounding box. By using column height and the offset of a Tromino in the bounding box I can find the lowest possible vertical position such a Tromino could be in that column (**Figure 15**). With this calculation I was able to try the most likely position for the piece to fit, then try positions above it. This is much more efficient than testing every position from the top of the board until the position is invalid. This optimization resulted in my agent only testing an

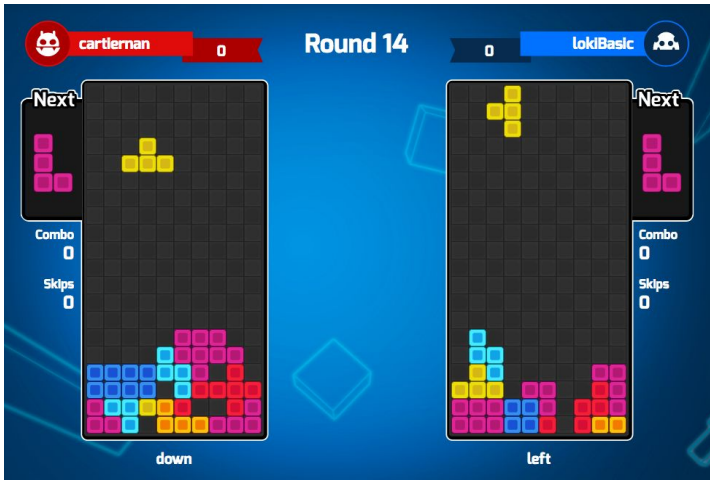
```
# lowest y pos = height of field - column hieght + offset of piece
y_min = field.height - col_height + piece_y_lims[0]
```

Figure 15
A sample calculation used to find the most likely position a Tromino will fit in a column

average of one to three fields before finding the valid position of a Tromino. With such a reduction in this repetitive calculation my agent can test all 1,600 boards within half a second.

3.2. Details

3.3. Bot Competing in the AI Games Block Battle



3.4. Post on AI Games Discussion Board

http://theaigames.com/discussions/ai-block-battle/5729050e5d203ce53dcc86c7/my-experience-for-those-who-are-interested/1/show

index → AI Block Battle → My Experience: For Those who are Interested

1

Created 1 second ago

ctiernan

I competed in the Block Battle for my final project in an AI class I am taking at University.

My bot, cartiernan, is programed in python3. It is not the most competitive bot by far, but it does have some optimizations to look two Trominoes in advance. It ranks board configurations by using a heuristic based off features and weights generated via a genetic algorithm.

If you want to look at/download my source code or read more on my experience feel free to checkout my GitLab account.

https://gitlab.com/ctiernan/Block-Bot

Best of luck in the competition, Carter

edit

3.5. Gitlab Code Repository Link

https://gitlab.com/ctiernan/Block-Bot

4. Discussion and Conclusion

The AI Games is a growing and enjoyable platform to apply AI concepts to various games. It is relatively easy to understand, has well documented rules and good organization. However, the issues I have with it is the inaccessibility of real game data from the competitions and also working around the java server engine is difficult in Python. With more time I could have accomplished a lot more, I found myself spending a lot of my time working to cater with the engine instead of implementing my agent. Thus if I had more time I would have been able to figure out the API and build a good set of tools to extract the information I needed to create a competitive agent. Additionally optimizing parameters would be viable with a longer period of time.

4.1. Current Implementation

I found one of the most intriguing parts of developing my agent was pathfinding. I spent a lot of time working to enhance the pathfinding of my agent so it was efficient enough to account for two Trominoes as well as flexible enough to find covered positions or “T”-spin positions. Although imperfect, the functionality to find possible Tromino locations is stable. The move list building through TetrisGraph is much less mature and still could be optimized. For instance, checking if a node was visited before would reduce redundant searches. Im sure if I were to bug fix my pathfinding my bot would perform better because it would not make mistakes such as misplacing blocks that quickly cause a Tetris game to be lost.

Additionally, the features of my agent need optimization. I would need to figure out which features are meaningful and which are not, possibly adding and removing them. Once I was confident I had a good set of features I would run multiple genetic algorithms with varying parameters to figure out what truly are the optimal feature weights my agent should use. I think if I were able to do both these optimizations my bot could compete at a much higher level.

One last thought I wanted to test on my agent was to have a set of preferred moves, much like Google caching frequent searches. If a “T”-Spin is possible, or 3 or 4 lines clearable my agent should not test other possibilities just do the extremely beneficial move. More over there could be “diss”-preferred moves such as covering a valley, or blocking a good move next round. This would also add complication upon how to rank such a prefered moves to each other.

4.2. Different Implementation

I think it would be very interesting to apply reinforcement learning and neural networks to Tetris. I have read some papers on the subject and although many early implementations were not promising newer techniques have shown some promise. I chose not to do so for The AI Games Block Battle because it seems that only very sophisticated and modern implementations can compete competitively and the

difficulty of creating such a system was out of the scope of this project. Additionally, the platform that The AI Games provides is adequate but very difficult to get game data from. If I were to apply reinforcement learning or neural networks to Tetris it would be on a different platform that ran the games faster and provided more easy access to in game data. Additionally, I think it would be interesting to start with single player Tetris to reduce the variability of the game to get a good grasp on it prior to competing.

5. Resources

Reinforcement Learning and Neural Networks for Tetris

- Nicholas Lundgaard, Brian McKee
- http://www.mcgovern-fagg.org/amy/courses/cs5033_fall2007/Lundgaard_McKee.pdf

Tetris AI – The (Near) Perfect Bot

- Yiyuan Lee
- https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/

AI Games Rules for Block Battle

- http://theaigames.com/competitions/ai-block-battle/rules

General Tetris Information From Wikipedia

- https://en.wikipedia.org/wiki/Tetris