# Artificial Intelligence:
## The Application of AI in Tetris

Prepared By: James Roose - 100852693
Prepared For: Dr. John Oommen
Date: April 19, 2015

**1. INTRODUCTION**
**1.1 Problem Description...…….3**
**1.2 Motivation……...……...……...3**
**2. SYSTEM DESIGN**
**2.1 AI Techniques……..…….4**
**2.2 Design Choices……..5**
**3. RESULTS**
**3.1 Result Summary…….……..9**
**3.2 Possible Enhancements……....11**
**4. REFERENCES**
**4.1 Reference List……...12**
**5. APPENDIX**
**5.1 Running the Code...……...……...13**

# INTRODUCTION

## 1.1 Problem Description

Tetris is a simple puzzle game, which involves manipulating blocks known as ‘Tetrominos’. These Tetrominos fall into the playing field at an increasing rate, and can be shifted right, left, or rotated to better fit the playing field. Once an entire row of the playing field has been filled with Tetrominos, it is removed from the field and the rows above it shift downwards. The game is lost when the rows of the playing field are stacked too high and reach the top of the screen. The objective of the game is to manipulate the Tetrominos you are given to complete rows in the playing field, thereby lowering the height of the playing field and preventing a game over for as long as possible.

The goal of this project is to implement Artificial Intelligence (AI) into the classic arcade game, Tetris. The ultimate goal for the AI is survival. It should be able to search the game space for the optimal location to place a Tetromino, such that a game over is prevented for as long as possible.

## 1.2 Motivation

Tetris is one of my favorite arcade games of all time. As a result, I have a good understanding of the problem domain and am largely interested in the problem. My own personal interest in the problem is my greatest motivation for this project. Additionally, Tetris provides a problem space that I feel can be broken down into components that and AI agent can understand, while still providing a sufficiently difficult problem. Lastly, this project has the potential to touch on many aspects of AI and while leaving room for future extensions.

## SYSTEM DESIGN

### 2.1 AI Techniques

The fundamental algorithm that is used is the Breadth-First Search (BFS) algorithm. This algorithm can be used to search the game space for the optimal location of a Tetromino in the game field. Additionally, Depth-First Search (DFS) was implemented in order to contrast the time complexity and performance of the two search algorithms.

Originally, I had also intended to implement the A* Search, which would make use of several heuristics in order to find an optimal solution. However, as the project progressed, it became clear that the BFS and DFS algorithm already needed to utilize these heuristics in order to find an optimal solution. Additionally, regardless of the order in which nodes are processed, all of the nodes must be taken into account in order to find an optimal solution. As a result, it became clear that the implementation of the A* Search algorithm would be redundant, as it would not limit the search space in any way or improve the performance of the system. In light of this, however, heuristics are still utilized in the implementation of BFS and DFS.

In Tetris, when placing a Tetromino, a window in the top right corner will display what the next Tetromino to be placed in the game field will be. Players use this information to place their current Tetromino in such a way that their next move will have a more optimal location in the playing field. The AI implemented in this project can replicate this play style simple by running the BFS or DFS algorithms twice. The first run will find all valid moves for the first Tetromino and the second run will determine all valid moves for the second Tetromino given each possible placement of the first Tetromino. The solutions would then be scored based on the predefined heuristics.

Lastly, a Genetic Algorithm (GA) or Particle Swarm Optimization Algorithm (PSO) was proposed as another means to find the optimal solution. Unfortunately, time did not allow for these features to be implemented. However, a PSO algorithm was used in another Tetris application to determine the optimal weights of several heuristics. Those results are included in this paper and will be detailed below.

## 2.2 Design Choices

The first major design choice that was necessary to make was how to implement the Tetris application that would be used by the AI. Before any work could be started on the actual AI, a Tetris game needed to be implemented. To solve this problem, a modified version of the T3TR0S game was implemented. T3TR0S is an open source implementation of Tetris that was written in Clojurescript and designed to celebrate the 30th anniversary of Tetris. However, the original implementation was far too complex for the purposes of this project, so a bare bones implementation was used and heavily modified to support AI interactions.

The next major design choice to be made was the implementation of a game loop for the system. The original T3TR0S game was heavily event driven, where events would mostly be triggered by user input. This had to be changed because there is no user in the scope of this project. Instead, a game loop was implemented which followed the algorithm detailed below:

1. If the game is running (we haven't hit a game over), start the loop
2. Retrieve the current state of the game
3. Run a search algorithm (BFS or DFS) to find a solution for the current state
4. Recursively apply the steps of the solution to the game space to place the Tetromino in the desired location
5. Update the game space and repeat from step 1

This allowed the game to be played in a sequential manner, in which the AI is given time to calculate a solution and can return a set of directions to be applied sequentially in order to place each Tetromino. However, it is suspected that the transformation from an event-driven system to a system governed by control flow is responsible for some of the problems that will be discussed later in this report.

Once the environment was properly set up to accommodate the AI, the next major design choice to be made was how to implement the BFS and DFS search algorithms. The biggest difference between this project and the snake game from assignment one is that the snake game only has one goal state. The food is in a single location, so the ultimate goal is to reach that single location. On the other hand, Tetris allows for a variety of different possible solutions, placing pieces in a variety of locations with several different orientations. Therefore, there exists the additional challenge of determining which of the valid solutions is optimal.

To do this, it was necessary to come up with a 'rating scheme' that would determine which solutions are optimal. Six major heuristics would be taken into account when evaluating each solution. These heuristics are defined below, and examples are shown in Figure 1.0.

1. **Lines Cleared** – The number of rows cleared be placing the given Tetromino.
2. **Game Field Height** – The height of the game field, how high the game field has been piled and how close we are to a game over.
3. **Total Well Cells** – The number of well cells in the game board. A well cell is an empty cell in the game board that has non-empty cells below it and on both sides of it, but an empty cell above it.
4. **Total Column Holes** – The number of column holes in the game board. A column hole is an empty cell located directly beneath a non-empty cell.
5. **Column Transitions** – The number of transitions in a column is the number of times an empty cell is found to be adjacent to a non-empty cell within one column of the game board.
6. **Row Transitions** – The number of transitions in a row is the number of times an empty cell is found to be adjacent to a non-empty cell within one row of the game board.

A PSO algorithm was run on another Tetris application to obtain the optimal weights for each heuristic. A higher weight for a particular heuristic represents a higher preference for a play style revolving around that heuristic. These results were to be used in this project and are shown in Figure 1.1 below.

**Figure 1.0 – Heuristics for Rating Scheme**

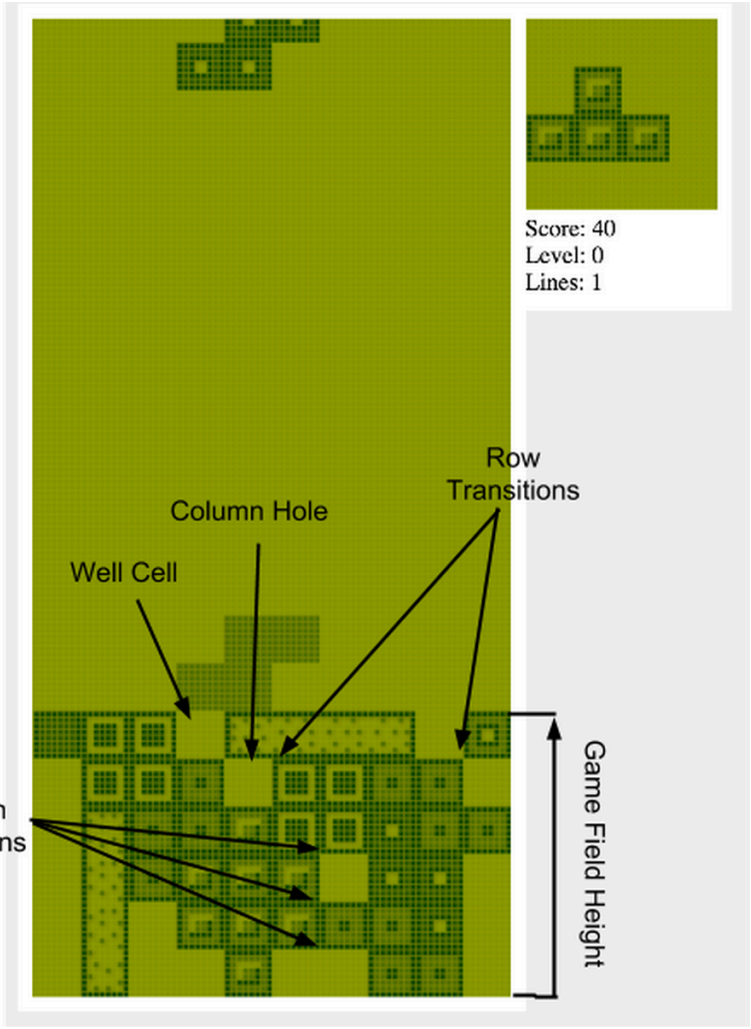


**Figure 1.1 – Heuristic Weights**

| **Heuristic** | **Weight** |
|---|---|
| Lines Cleared | 1.0000 |
| Game Field Height | 12.885 |
| Total Well Cells | 15.842 |
| Total Column Holes | 26.894 |
| Column Transitions | 27.616 |
| Row Transitions | 30.185 |

The last major design choice that was made was how to define the algorithm for the BFS and DFS searches in the context of Tetris. The following algorithm outlines the process of finding the optimal solution for the placement of a given Tetromino using the BFS or DFS algorithms (Step 3 in the game loop described above).

1. Add the spawn state to the node list.
2. Remove a node from the node list.
3. Obtain successor states by applying the transformation operations (shift left, shift right, move down, rotate).
4. If ‘move down’ is not among the valid successor states, then the current node is a ‘locked’ node; its solution can be considered for evaluation.
5. Add the successor states that have not already been visited to the node list.
6. If the current node is a ‘locked’ node, compare its solution to the current best solution. The better solution is the new best solution.
7. If the node list is not empty, repeat from step 2. Else, return the best solution.

# RESULTS

## 3.1 Result Summary

Overall, I believe that the approach that was taken and described above was the right approach. However, it is unfortunate that the execution of the proposed approach did not go smoothly. I ran into many problems while trying to set up the environment for the AI and a fundamental logic error currently exists within the code. Nevertheless, conclusions can be drawn from the project, and these conclusions will be taken into account in the possible enhancements section below.

The hardest part of the project was transforming the original, event-driven, T3TR0S game into a bare bones and sequential Tetris game. These modifications were necessary to provide an environment that the AI could operate in, however, the game was not originally structured or intended to be played in this way. As a result, many problems related to control flow and current functionality began to arise. The AI could not use many of the important functions used in the original game because they directly impacted the game space, rather than the solution space the AI works with. Duplicate functions had to be created which provided the same functionality to the AI.

A currently existing problem with the application is the inability to clear completed rows. Once a row is completed, the animation for removing the line is launched and the line is removed from the game space. However, control flow is not given back to the game loop, and the game stops. An effort was made to return control flow to the game loop after the line removal sequence had finished, but it was not successful. Aside from this issue, the game loop is working as intended and properly manipulates the game space in accordance with the given solution.

Unfortunately, the search algorithms for BFS and DFS contain a major logic error. They work mostly as intended, however, the AI occasionally makes illegal moves. These issues stem from an error in determining which states are in a 'locked' state. A node should only be in a locked state if it is unable to move down any further in the game field. It is only these nodes that are taken into account when returning a solution. Therefore, there must be a logic issue in determining which nodes are in a 'locked' state. Aside from this issue, BFS and DFS appear to work as intended.

Surprisingly, however, BFS and DFS perform very differently. BFS makes illegal moves much more often than DFS, meanwhile DFS seems to perform decently overall. This conclusion is surprising because both algorithms perform on the same logic. The only difference between the two algorithms is the container in which they store nodes (Queue for BFS vs. Stack for DFS). This leads me to believe that DFS performs less illegal moves because it quickly explores the 'locked' nodes before most of the 'unlocked' nodes; meanwhile BFS explores most of the 'unlocked' nodes before it considers the 'locked' nodes. DFS quickly reaches a greater depth in the search (as implied by the name) and therefore quickly reaches the bottom of the game field; meanwhile BFS explores the breadth of the game field, reaching the deeper nodes at the bottom of the game field last. This gives DFS a chance to select a 'locked' node as the optimal solution before selecting 'unlocked' nodes.

The last reason as to why both searches perform poorly is due to the fact that the rating system proposed in section 2.2 above in not fully implemented. Due to time constraints and the inability to resolve currently existing problems, I was unable to fully implement this system. Currently, the AI only favours solutions that limit the height of

the game field. Had the rest of the system been fully implemented, it is likely that both search algorithms would perform better. Additionally, the weights could be altered, detailing which heuristics should be used and what kind of play style the AI should have.

### 3.2 Possible Enhancements

Needless to say, a lot can be done to enhance the current state of the project. It can be seen that there exists three major issues preventing the current implementation from working properly: a fundamental search error, an error in regards to control flow, and an incomplete implementation of heuristics. The system could be largely improved by fixing the BFS and DFS search algorithms to no longer make illegal moves. Once this is done, the remaining heuristics proposed in section 2.2 could be implemented to allow the AI to vary its play style. Once these issues are fixed, the issues governing control flow would need to be fixed before the AI could do any real gameplay. Without the ability to clear lines, the AI is doomed to fail. After implementing these fixes, the system would perform as expected and further functionality can then be added.

After the above core issues are addressed, the remaining aims of this project could be pursued. For instance, the AI could take into account its next Tetromino piece when placing its current piece on the board. This could be done by running the improved BFS or DFS searches twice before returning a solution.

Lastly, the GA or PSO algorithms could be attempted, in order to compare the performance of these vastly different algorithms to that of BFS and DFS.

## REFERENCES

### 4.1 Reference List


- Applying Artificial Intelligence to Nintendo Tetris. (n.d.). Retrieved April 19, 2015, from http://meatfighter.com/nintendotetrisai/#The_Algorithm

- Looney, E., Williams, S., Gutierrez, L., Oakman, C., Darnell, B., & Gambling, P. (n.d.). Imalooney/t3tr0s. Retrieved April 19, 2015, from https://github.com/imalooney/t3tr0s

- Williams, S. (n.d.). Shaunlebron/t3tr0s-slides. Retrieved April 19, 2015, from https://github.com/shaunlebron/t3tr0s-slides

# APPENDIX

## 5.1 Running the Code

1. Install Leiningen
2. Open up a terminal window and navigate to the AI-Tetris directory
3. Enter ‘lein cljsbuild auto’ into the terminal to run the auto-compiler
4. Play the game by opening public/index.html in your browser
5. The type of search used (BFS vs. DFS) can be changed by modifying the search-type defined on line 41 of game/core.cljs. Type :bfs is for BFS and :dfs is for DFS.