The Count of Monte Carlo

Applying Monte Carlo Tree Search to Tetris

Kevin Holdcroft	Samir Khays	Yoshua Nava	Viktor Tuul
19930407-9131	19950822-5171	19901117-1411	19950530-2613
kevinhol@kth.se	khays@kth.se	yoshua@kth.se	tuul@kth.se

Abstract

Tetris is a classic video game and has cemented it's place in popular culture. Developing Artificially Intelligent players for video games provide insight as to what scenarios different methods perform best at. In this paper, we implemented a Monte Carlo Tree Search (MCTS) AI capable of playing Tetris. We compared this method to a shallow Breadth-First Search (BFS) method. The AI using MCTS was able to clear 27 lines in a single run, while the BFS was able to clear 40 with a search depth of 1 and 92 lines with a depth of 2. While MCTS was able to reasonably play the game, the depth of it's search provided worse results. It chooses random pieces to simulate the future states, and if these pieces do not appear, the block placement is non-optimal.

1 Introduction

Tetris is a classic videogame released in the mid-1980s. The game involves arranging 7 different types of blocks to create rows along the playing area. The blocks fall from the top of the map downwards and can only be rotated or moved left and right by the player. Each time a block is placed, a new, random block begins descending at the top of the screen. Depending on the variant of the game, one either has knowledge of type of the next immediate block or not.

The blocks fall into a grid that is 20 squares high and 10 wide. Whenever every square in a row is occupied, the row disappears and the squares above lower to occupy the disappeared row. When the stack of blocks reach the top of the grid, the game is over.

AI for Tetris already exist, typically either using breadth first search (BFS), or deep learning [1][2]. As the incoming blocks cannot be predicted, the block planner has to be capable of adapting to different patterns.

Since the number of states is discrete, tree-search methods seem like an obvious choice. BFS uses a heuristic function to evaluate the placement of the piece. Niko Böhm et al. introduced genetic algorithms in order to refine the weights on their heuristic functions, producing an AI capable of clearing 859,520 lines [3]. Eric Fritz used used similar methods to create a model that can clear over 100,000 lines within only an hour of training [4].

However, BFS methods are limited due to time constraints. As the levels in Tetris increase, the blocks fall faster, and the available search time decreases. Due to time constraints and the large number of future states, these methods typically are only able to search to a depth of one or two [5].

Deep learning methods require extensive training in order to perform well. The states are image based, which means they they are catered to one version of Tetris and might not perform well on other versions.

1.1 Contribution

In the paper, we used Bandit-Based Monte Carlo Tree Search (MCTS) as an agent to play Tetris. AI for Tetris already exist, typically either using breadth first search (BFS) and deep learning (CITE). MCTS performs well under time constraints, since while it will likely not return an optimal result, it will always return a *good* result [6]. As such, MCTS should be a resonable algorithm choice when designing an AI capable of playing Tetris.

To demonstrate the capabilities of MCTS, we designed an AI agent that is able to play Tetris to a reasonable capability. The results from MCTS

were compared to a model with a simple BFS with a search depth of one.

1.2 Outline

Common practises in Tetris AIs will be discussed in section 2, as well as introducing the background to MCTS. The theoretical methodology will be discussed in section 3. Section 3.1 will include details on how the heuristics were chosen, as well as their weights and other constants. Section 4 will show the results from simulations, and finally, a conclusion as well as a critical review of the method will be given in section 5.

All simulations and algorithms were run and written in Python. The version of game used is found here: https://github.com/jjengo/tetris

2 Related work

Tetris was invented in 1985 by Alexey Pajitnov et al., and has since been subject to research to perform a capable AI player. Various approaches have been used, but have typically fallen under the umbrella of tree search [7].

In 2003, Colin Fahey was developed an AI capable of clearing one million blocks in real-time [8]. He used a BFS search to a depth of max two, utilizing only three heuristics: the height of the stack, the flatness of the pile, and the number of empty covered by rows. What makes this especially impressive is that the algorithm was run on a separate computer than the game, utilizing a webcam pointing at a CRT screen.

Similarly in 2003, Pierre Dellacherie developed a one-piece AI capable of clearing two million rows, with over 650,000 rows cleared on average [8]. Unfortunately, his heuristics and methods are not available.

In 2005, Niko Böhm et al. suggested combinging BFS with genetic algorithms in order to configure the weights on their heuristic functions. Their AI cleared 859,520 lines, which is less than Fahey's or Niko's algorithm [3]. However, their approach still proved that evolutionary algorithms are a reasonable approach for developing heuristics, and the decrease in line score could be due to different versions of the game.

Zhongjie Cai et all developed an AI for Tetris using bandit-based Monte-Carlo Tree Search in 2011 . The results from this paper did not directly state the number of lines cleared, but rather showed that a competent AI can be produced through MCTS. By using hasing functions, they expect their player to improve in time, but currently state that it's capabilities are above a benchmark player [5].

In 2013, Yiyuan Lee wrote a blog post on this development of a BFS Tetris AI, which uses a Genetic Algorithm in order to refine the scores. His setup cleared 2,183,277 over a two week period. While we did not use Genetic Algorithms and while this provides nothing new, our heuristic functions were inspired by his [9].

In 2017, Björn Dagerman combined Behavior Trees and Monte Carlo Tree Search in order to play a competitive version of Pac-Man. Behaviour Trees were to modify the weights for the heuristic function in MCTS [10]. We originally intended to use a similar approach, but as we began to develop a Behaviour Tree for Tetris, we realised that the heuristics would remain static regardless. The issue is that the state of the Tetris game never truly changes, since the goal remains the same regardless of the piece produced.

Alejandro Marzinotto describes how to use Behaviour Trees as a high-level logic to get a robot to grasp an object [11]. We originally intended to use Behaviour Trees as a path planner which allows the piece to fall. However, fall planner has very few nodes and doesn't need detailed logical processing. Similarly, we realised that the Tetris version we have accepts multiple key inputs per move, and as such developed a function capable of moving the piece to it's proper location within one in-game time-step.

3 Methods

We designed the path-planner using bandit-based Monte Carlo Tree Search.

A tree search method uses recursive nodes which represent possible states of the grid (referred to henceforth as "game-states"). By using some method of evaluating which is the best game-state to end up in, search methods choose a current action that will result in the best future state.

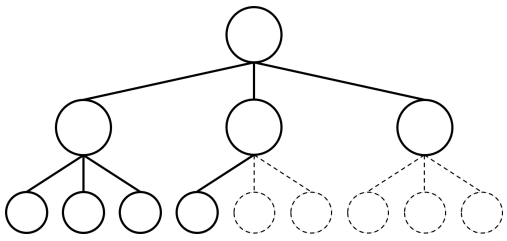


Figure 1: Example of Breadth-First Search. The search starts at the current game state, then searches the next possible game states for the best possible heuristic score. Then, it searches the future game states, the "children", for the best possible score.

Breadth First Search evaluates all possibilities for a nodes before predicting the moves one more step into the future. This method is decent for non-stochastic systems. However, when future variables are not known, BFS evaluates not only the possible actions, but each of the actions for each random action.

In the event of Tetris, BFS evaluates the best possible place to put the current piece in play. If the next is known, BFS finds the best possible place to put the next two pieces to optimise the value functions. However, when the piece is not known, the search must not only simulate all possible rotations and translations for each piece, but must do this for each possible block. This causes the AI to take exponentially more time, and is unrealistic for real-time play.

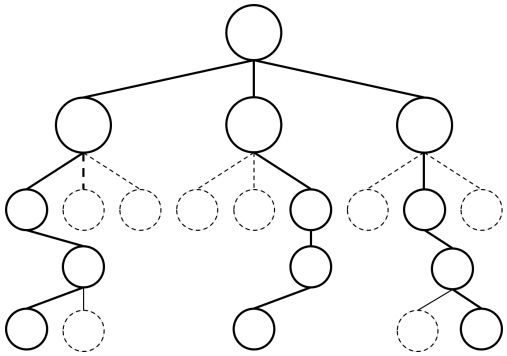


Figure 2: Example of Monte Carlo Tree Search. The search starts at the current game state, then searches a random number of the next possible game states for the best possible heuristic score. Then, it randomly searches future game states, the "children", for the best possible score. It explores deeper, but to less completeness.

MCTS, in contrast, obtains an estimate of the best move by exploring random possible paths. Exploring random paths works by exploring deep and predicting what could be a decent move based on the stochastic future. Rather than searching through all possible pieces, it might only choose a few of them, and only a limited number of positions. Thus, based off of what *could* happen, it chooses the best move. The result is a method that tries to optimise how the board will look for a random piece.

3.1 Implementation

We employed an open source version of Tetris, that built upon Python standard libraries, numpy and pygame. The game encoded its current state as a matrix of integers, and was able to provide, at a given time step, information on the current and next pieces. The Tetris engine chosen was not particularly well optimised, but was robust, well designed, and documented.

3.1.1 Previous Approaches

We initially attempted to use a variant of Monte Carlo Tree Search called Blind-Bandit MCTS. Blind Bandit MCTS searches deep, using a random policy, in an attempt to find a "win" condition. Since you can never "win" at Tetris, we set the win condition to be if the heuristic functions were within a certain value. Thus, the tree would explore until it finds a win condition, and move the current piece in order to find this condition.

The advantage of the Blind-Bandit approach is that the search can be sped up rapidly by use of a hash-table [5]. Since you are searching deep, a

hash table save the results and improve the speed the game takes to explore a tree. As the game progresses, the hash-table expands, which in turn frees more time to allow for search.

The problem with this approach is that the the future pieces are random, and not previously determined. Thus, the pieces will move to a poor location which depends on a future piece. When that piece doesn't arrive, the previous piece remains in it's same poor position.

We still kept a hash table local to each node, to prevent revisiting states. For example, the square piece cannot be rotated, so there is no point to search through all possible rotations of it.

As a result, instead of the Blind-Bandit approach, we evaluated heuristics at each node. We originally based the validity of an action based off of the last heuristic of each branch. However, this meant that the any path with ideal piece drops will be favoured, and we end up with the same problem as the Blind-Bandit approach.

Instead of taking the last node as the heuristic value, we summed the heuristics from each child node and normalised it by the number of nodes. As such, instead of evaluating the best possible path, we are instead evaluating the robustness of the position of the piece for each child node. If an action gives a good heuristic, we know that it's random children and their random children all have good heuristics. This is opposed to taking an action down a path where end node is ideal, but deviation from the path will cause a bad piece placement.

In our last attempt at the problem, we validated "win" situations by conditioning on the height and number of holes of the tree root (the current state), the leaf (the most immediate choice), and the last playout (the future) reached in the simulation. This gave us the best results, but was, nonetheless, not capable of competing with the performance of simple Breadth-First Search.

It must be said that due to performance limitations in the game engine, we didn't attempt simulations with a depth higher than 3, neither we explored more than 30 nodes. Due to performance limitations, we didn't work on improving the performance of the engine, which could have allowed us to run more iterations of MCTS in less time, therefore increasing our information gain while planning.

3.1.2 Heuristics

We used three different heuristics to evaluate the quality of each node.

Heuristic	BFS Weight (d=1)	BFS Weight (d=2)	MCTS Weight
Number of Holes	-20	-2	-10
Aggregate Height	-90	-60	-0.5
Clear Line	0	0	0

The Number of Holes heuristic acts to prevent empty cells below another square. When a hole is below another piece, it prevents the rest of the row from being complete. The hole cannot be filled until the squares above it are removed, which may not happen if there is a buried hole preventing the block from disappearing. This metric is absolutely necessary, since it prevents an exponentially growing problem.

The Aggregate Height function sums the square the height of each column. This is the basic heuristic which defines the overall game - it gets worse as the height goes up. If Tetris was non-stochastic and if we have an unlimited tree, the AI should be able to get by on this heuristic alone. By making the heuristic the square of heights, rather than just a sum, each piece has more motivation to lay as flat as possible. See the following equation:

$$H_2 = \sum_{c=0}^{9} (column\_height_i)^2 \tag{1}$$

One would think that the Clear Line heuristic might be the most important of the three, since is the most basic and best goal out of the three. If the move results in one line being cleared, the heuristic returns a score of 1. If two lines are removed, it returns a score of 2, and so on. This is as best of a "goal" state as we can get for a game such as Tetris. However, we noticed that this heuristic was implicitly contained in the "Aggregate Height" heuristic, since future moves will lead to a state of lower overall height. While implemented, we opted to remove it and didn't noticed any impact on performance.

4 Experimental Results

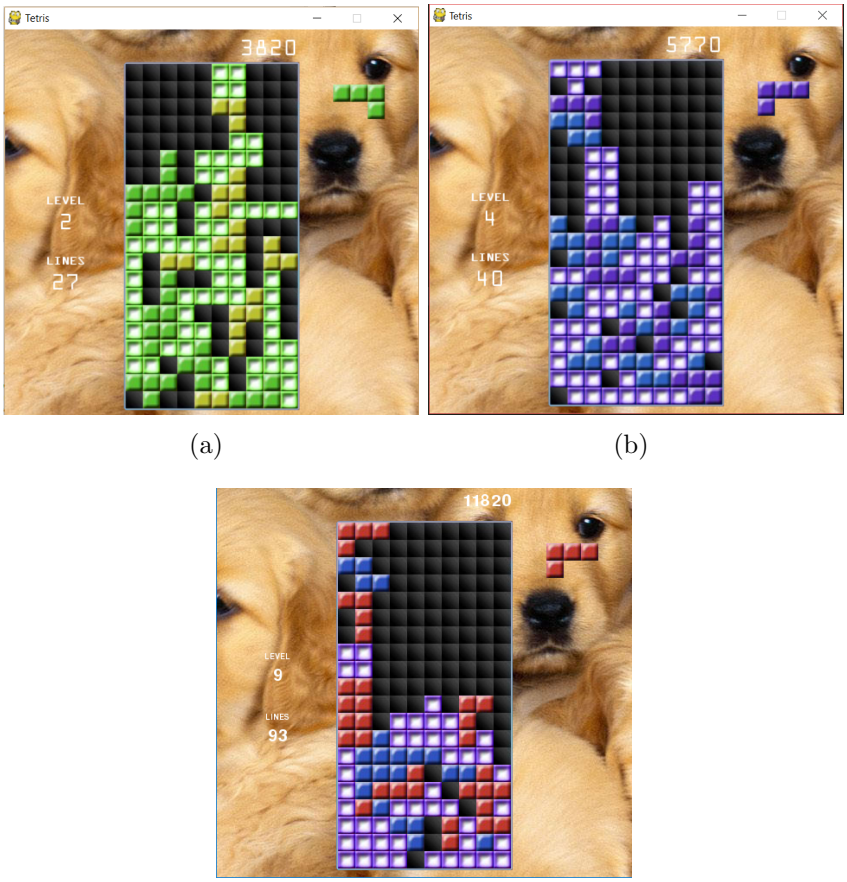
4.1 Experimental Setup

All simulations and algorithms were run and written in Python 2.7. We accessed the game objects only to obtain information about the game's state, to move the players, and to change the background image to that of puppies. The version of game used is found here: https://github.com/jjengo/tetris

Unfortunately, we were unable to test in real-time on this paper. In order to simulate real-time, we placed time constraints on the MCTS function, such that actions would be chosen in a reasonable time.

4.2 Experiment

We ran several iterations of each MCTS and BFS, and saved the highest results from each. The results are shown in figure 3.



(a) (b)

(c)

Figure 3: Best results for corresponding tree searches. (a) shows the final screen when running MCTS. (b) shows the final results when running BFS with a depth of 1. (c) is the results when running BFS with depth of 2

BFS returned the best results, followed by Monte Carlo. The summary is found in table 1

Method	Lines Cleared	Score
BFS (depth 2)	93	11820
BFS (depth 1)	40	5770
MCTS	27	3820

Table 1: Final Results

4.3 Discussion of Results

We believe that BFS works better, since the the actions do not overfit to the future states. When designing the heuristics, we try to take into account everything that might result in a good piece placement. There should be no holes, the grid should not be high, and the piece should try to clear lines. These effectively make a decent choice of to how to effectively place the piece to produce the best possible immediate grid. Since the grid always attempts to look as good as immediately possible, the grid will work well for future states as well.

This is synonymous as to how humans play. They rarely focus on long-term planning, but rather try to place the immediate piece in the best possible location. At the high levels of Tetris, there is no time for long-term planning, so players instead use immediate decision making. As such BFS plays with the same logic as a human player, and as such is as effective.

However, MCTS has a habit of "overfitting" the actions to future states. Since random nodes are chosen, the best action is biased towards nodes that have a specific piece. For example, if MCTS explores a path that has two straight pieces in a row appear (considered to be good luck), MCTS will return good heuristics for that branch, and will likely choose and action that anticipates those blocks.

Tetris is an interesting game, in that one block can really change the entire game. Typically, nine of the columns end up growing, with one column left open. Then, when a line piece appears, it can remove four blocks at a time. However, if this block does not appear, the map can build up rapidly. The issue with MCTS is that it anticipates that this or another piece will always come, and plans the game accordingly. When a piece doesn't appear, it can cause holes regardless of where it is placed.

However, BFS with a search depth of 1 does not plan into the future, and as such does not rely on obtaining a future piece. At every instance, it will focus on creating the best possible gameboard. The heuristics are catered towards creating the best possible gameboard, which in turn provides BFS with enough information to play Tetris effectively.

5 Summary and Conclusions

In this paper, we developed an AI that can play Tetris to a reasonable competency using Bandit-Based Monte Carlo Tree Search (MCTS). MCTS explores random possibilities to a depth deeper than possible with breadth-first tree search (BFS). The AI was capable of clearing 27 lines without human intervention.

While these numbers aren't huge, it shows that MCTS can perform as a competent AI for discrete non-deterministic games. We could likely improve the results by implementing Genetic Algorithms to improve our weights and refining the heuristic functions. Similarly, we could save the hashing results between games, which would cause MCTS to improve with more games played. Our current implementation is still rather rudimentary, and has a large room for improvement.

Surprisingly, BFS outperformed MCTS. We reason that MCTS becomes biased by having random paths with ideal piece drops, which will then move the current piece to a bad position in anticipation of those future pieces. Since the game is stochastic, the pieces never come, and MCTS is left with in worse game-state.

We anticipate the MCTS will be ideal in scenarios where there is a clear benefit to exploring deep, rather than shallow. Situations with few actions are preferable, and ones where actions can be undone. For example, moving a character across a grid (like in Pac-Man) would be an ideal scenario. However, with Tetris, MCTS does not provide any clear advantage.

References

[1] Amine Boumaza. How to design good tetris players. 2013.

[2] Tetris ai wikipedia. http://tetris.wikia.com/wiki/Tetris_AI.

[3] Stefan Mandl Niko Böhm, Gabriella Kókai. An evolutionary approach to tetris. 2005.

[4] Eric Fritz. hard-drop. https://github.com/efritz/hard-drop. (Claim found at: https://www.reddit.com/r/gamedev/comments/24nubs/how_i_made_a_tetris_ai/ch8z3q7/?st=j8p1qjw5&sh=4846da42).

[5] Zhongjie Cai, Dapeng Zhang, and Bernhard Nebel. Playing Tetris Using Bandit-Based Monte-Carlo Planning, 2011.

[6] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML*, volume 6, pages 282–293. Springer, 2006.

[7] Bruno Scherrer Christophe Thiery. Building controllers for tetris. international computer. https://hal.inria.fr/inria-00418954/document, 2012.

[8] Colin Fahey. Tetris. http://www.colinfahey.com/tetris/, 2003.

[9] Y. Lee. Tetris AI – The (Near) Perfect Bot. https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/, 2013.

[10] Björn Dagerman. High-level decision making in adversarial environments using behavior trees and monte carlo tree search, 2017.

[11] Alejandro Marzinotto. *Flexible Robot to Object Interactions Through Rigid and Deformable Cages*. PhD thesis, Computer Science and Engineering, 2017.