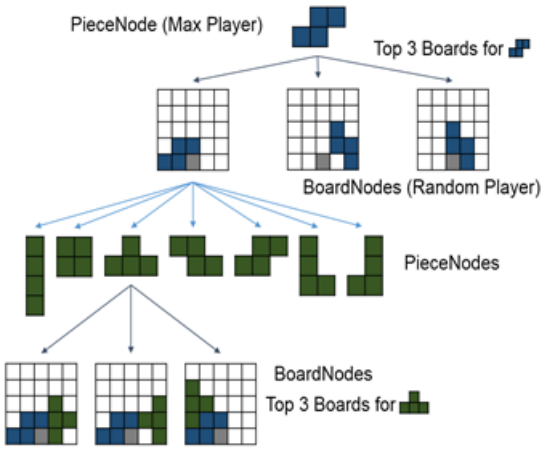
AlphaTetris

Group 20: Karen Ang Mei Yi (A0116603R), Leow Yijin (A0131891E), Ng Jing Siang (A0124321Y),
Teo Qi Xuan (A0124206W), Vincent Seng Boon Chin (A0121501E)

1 Introduction

Our Tetris agent focuses on searching as deep as possible whilst making use of beam search to limit the number of states that it has to consider. In order to scale up to big data, we have chosen to make use of parallel computing to speed up both search and learning aspects of our agent. We also make use of a column based bitboard to represent the board state which is much more memory efficient and allows us to exploit efficient bitwise operations. The agent alternates between a PieceNode and a BoardNode. The former expands into multiple BoardNodes and only a constant k number of top scored boards are kept for further expanding.

2 Agent Strategy



Looking ahead and keeping the best few boards for each piece.

2.1 Beam Search with ExpectiMax

Tetris is a game with a high branching factor. Therefore a look-ahead strategy is computationally expensive, even for just one layer. To overcome this problem, we employed beam search where we only keep the k best boards for each piece.

The scores are propagated back to the root node using the ExpectiMax policy where the PieceNode passes back the top scored board and the BoardNode passes back the average score of the 7 PieceNodes.

3 Learning

The heuristic function we have chosen is a linear weighted sum of features, namely, the number of complete lines, the aggregate height, landing height and bumpiness of the board, alternating blank and filled spaces along a row and a column, termed as row and column transition in the source code, the number of wells in the side and top, the difference between the height of the columns and the number of holes.

Since we do not have a corrective agent or learning data to conduct supervised learning, we chose to explore (1) Genetic Algorithms and (2) Particle Swarm Optimisation in training the weights.

3.1 Genetic Algorithm (GA)

We utilised a GA library known as JGAP to train the weights of each feature. Beginning with a population size of 1000 randomly crafted chromosomes, where the range of each weight ranged from 0 to 10, we evolve the population for 20 generations. The fitness function was defined as the aggregate empty space height (detailed in later part of the report). Evolution occurs using both crossover and mutation. The new chromosome is then subjected to mutation with some probability to maintain some diversity in the new generation of chromosomes, to breed even fitter chromosomes in the future generation.

Our results from the training is as follows:

ColTrans	9.994
Holes	9.430
Well	2.804
TopVariety	2.576
RowTrans	2.152
CompleteLines	1.721
AggregateHeight	1.410
MiniMaxTop	0.782
LandingHeight	0.441
SideBump	0.296
TopParity	0.109

Clearly, the most important features are column transition and holes while MiniMaxTop, landing height, side bump and top parity play a subdued role.

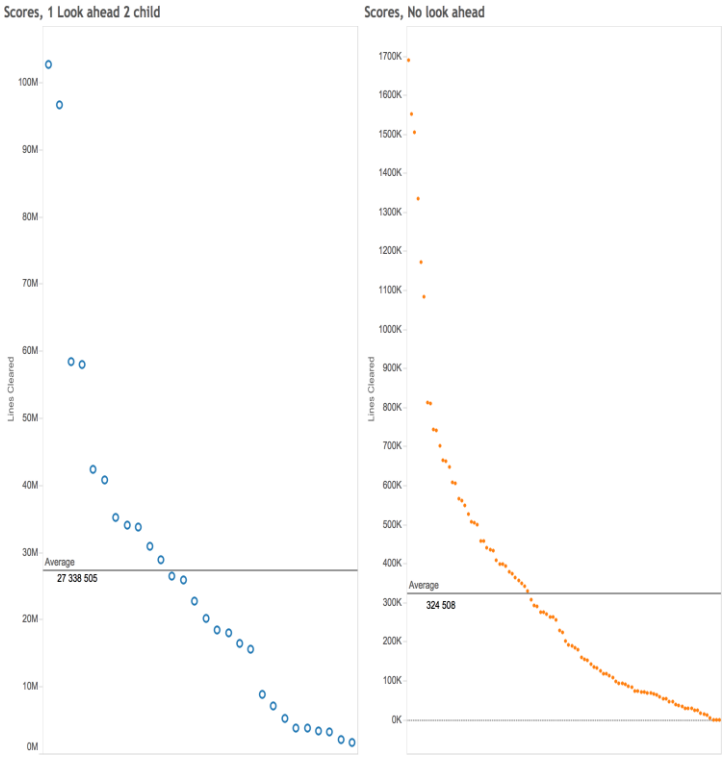
3.2 Particle Swarm Optimisation (PSO)

In researching online, we came across a method of training weights for the heuristic function which has produced good results. As a novel method for learning, we decided to explore the use of PSO. We made use of 4 basic features in the initial training using a library. However, we noticed that there appeared to be a limit that the agent seemed to be approaching for some set number of turns, despite the fact that the particles were

initialised to random positions. Moreover, the library used was quite limited, and we could not tweak various attributes such as starting positions, velocity along each feature as much as we would like to. Consequently, we have opted to utilise the genetic algorithm for training the weights instead.

4 Performance

4.1 Experimental Results and Discussion



Graph of scores obtained. Note the different axes.

Each point represents the score when the game terminates. Scores are sorted in descending order

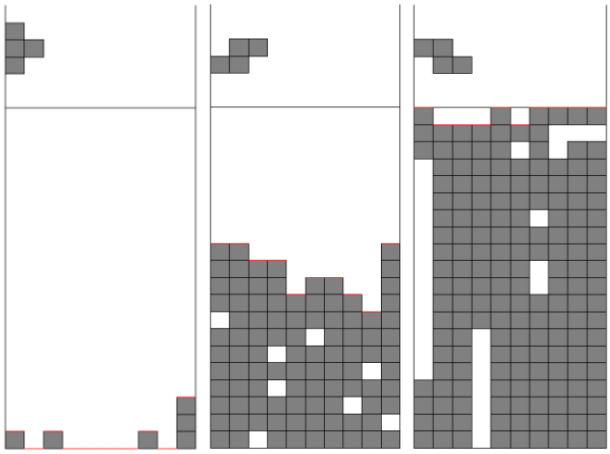
The agent played 100 games without look-ahead and 25 games using one look-ahead with two child using the weights obtained. For its worst performance, 1 look-ahead obtained a score of 2 million while its counterpart managed to clear only about 100 lines. From the scores above, it is evident that looking ahead helps the agent's average performance by about a factor of 100.

4.2 Novel and Significant Achievements

Our algorithm makes use of two distinguishing features: a 'reward' in training and a non-standard representation.

4.2.1 Notion of a Reward

We make use of a reward measure, which is given by the sum of the empty squares being maintained over the course of a game played by the AI. This reward measure is used to distinguish between heuristic functions which may produce similar number of lines cleared over a given number of turns. In such a case, it is evidently better if the AI consistently maintains a low height and as many empty squares as possible. This ensures the survival of the AI in the long-term. A subtle implication is that the best performing agent is one that clears the most number of lines.



The reward penalizes deeps wells as shown in the third diagram and encourages formations such as the one in the second diagram, which we have found to be better through empirical observation.

4.2.2 Bitboard

The standard Tetris board consists of 20 rows and 10 columns which is commonly represented as a boolean array. In java, a boolean has the same memory footprint as a 32-bit integer. Therefore a boolean array is not a very space efficient representation of the board. We implemented the state using bitboard where each column is represented by an integer and each bit on the integer corresponds to a position on the board. A set bit would indicate that the position is filled. The board can be represented by using just 10 integers which is 20 times more space efficient than using 200 booleans. We are also able to calculate some of the heuristic much more efficiently by using bitwise operations.

5 Scaling up to Big Data

When scaling up to big data, we consider two main factors - *Time* & *Space*. In the following sections, we will address how our algorithm is explicitly designed to handle these two criteria.

5.1 Space

In Artificial Intelligence, space tends to be a bigger problem than time (Low K. H., 2016). As such, we focus mainly on how we optimize in terms of space.

Based on our own observations, running our beam search algorithm using the given functions in State.java with the basic 7 pieces takes a whopping 3GB! Using the bitboard implementation that we described above, we are

able to reduce this to 4.11MB. Using some experimental runs, we obtain the following data:

Memory usage of Beam Search Algorithm using:

# of Tetris Pieces	State.java	BitBoardCol .java	% Usage Reduced
7	2.97GB	4.11MB	99.86%
14	3.90GB	7.23MB	99.81%

From this, we can observe two things. Firstly, BitBoardCol seems to provide a significant memory reduction of about 99.8% (wow!) regardless of the number of Tetris pieces, thus proving that our implementation does indeed save on space. Secondly, doubling the number of Tetris pieces raises the memory usage by a factor of less than 2. From this, we can guess that our implementation scales at most linearly with the number of Tetris pieces, that is, **O(n)**. We thus believe that it will be able to scale efficiently with the number of Tetris pieces.

5.2 Time

We also look into the efficiency of using a genetic algorithm, and how it scales with big data. The efficiency of a Genetic Algorithm is proportional to the population size and the number of generations it takes to converge(van Dijk, Thierens & de Berg, 2000). Thus, by varying these factors, we manage to obtain the following data. We also check if the algorithm manages to clear at least 1000 lines without dying as a check against premature convergence.

# Tetris Pieces	Pop. Size	Max generations allowed	Time to Converge	Lines Cleared >1000
7	100	5	17.78s	F
7	100	20	59.99s	T
7	1000	5	175.13s	F
7	1000	20	9 hours	T
14	100	20		T
14	1000	20		T

We made a few observations from the data.

Idk??

THIS NEEDS TO BE UPDATED

Time appears to scale linearly with generations and population size. At the same time, there seems to be an ideal balance of population size and generations that will allow for convergence in the shortest time possible.

When doubling the number of Tetris pieces, we discover that the time appears to scale about linearly as well. This is because of the playing of the game, rather than the genetic algorithm, as when we run the beam search using multithreading, the time speeds up significantly.

Extrapolating from our experimental results, we believe that running the multithreaded version on n Tetris pieces will take about (100n) seconds, which we believe to be tractable.

6 References

van Dijk, S., Thierens, D., & de Berg, M. (2000). Scalability and Efficiency of Genetic Algorithms for Geometrical Applications. In Schoenauer, M., Deb, K., Rudolph, G., Yao, X., Lutton, E., Merelo, J., & Schwefel, H., editors, *Parallel Problem Solving from Nature* PPSN VI (1st ed., p. 683). Berlin: Springer-Verlag Berlin Heidelberg.

Low K. H., B. (2016). *Lecture 3: Tree Search Algorithm*. Lecture, NUS LT19.

7 Appendix

DIAGRAM IF WE HAVE SPACE