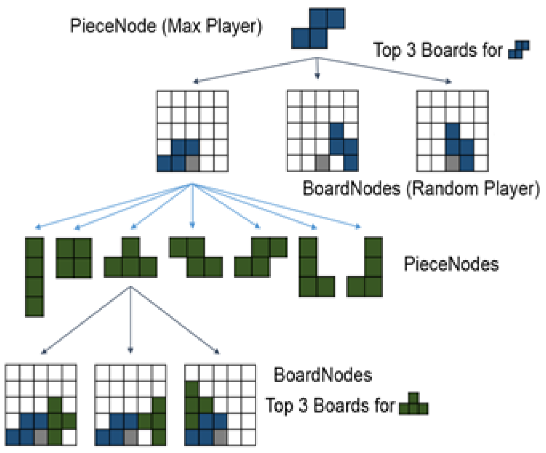
AlphaTetris

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 uses a beam search strategy with a weighted sum of heuristic features. The feature weights are determined by genetic algorithms and particle swarm optimisation, with a special valuing function. The search algorithm was parallelised to allow game and training performance to scale up to Big Data, and we used bitboards to model game state.

2 Agent Strategy



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

2.1 Beam Search with Chance Layers

The search tree layers alternate between PieceNodes and BoardNodes. The former expands into multiple BoardNodes and only a constant k (beam search width) number of top scored boards are kept for further expanding. The scores are recursively propagated back up to the root node using expecti-max policy.

2.2 Heuristic Features

The heuristic function we have chosen is a linear weighted sum of features. The features are:

1. **Aggregate height**: Sum of the column heights.
2. **Complete lines**: Number of lines eliminated by the taken move.
3. **Holes**: Number of blank cells below the highest filled cell in each column.
4. **Column transitions**: Number of alternating blank and filled cells along a column.
5. **Row transitions**: Number of alternating blank and filled cells along a row.
6. **Landing height:** The landing height of the new piece.
7. **Top parity**: If we fill the board with black and white squares in a checkered pattern, the top parity is the absolute difference between the number of white and black squares at the tip of each column.
8. **Top variety**: Number of unique 2 column height patterns. It increases the surface variety which reduces the chance of a forced hole.
9. **Min Max Top**: Difference between the highest and shortest column.
10. **Side bumps**: Sum of absolute height difference between the sides and their adjacent columns.
11. **Well**: A blank cell that is surrounded by a filled cell on its left and right.

3 Learning

Since we do not have a corrective agent or learning data to conduct supervised learning, we chose to use both Particle Swarm Optimisation and Genetic Algorithms in training the weights.

3.1 Particle Swarm Optimisation (PSO)

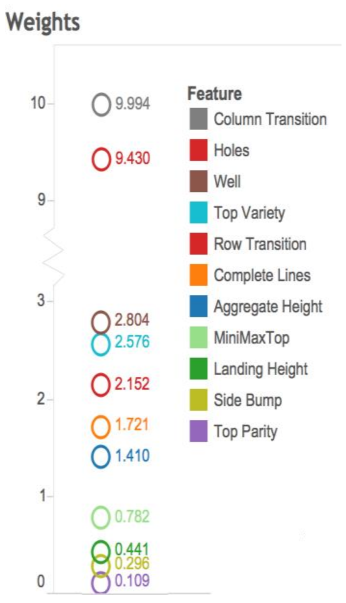
We used an open source PSO library (jswarm-pso) to obtain initial weight values for each feature. We started with 100 randomly initialised candidate particles as the initial population. The inertia parameter was reduced to accelerate convergence rate. The PSO was run for 30 iterations to get an “elite” set of values to seed the GA’s starting population

3.2 Genetic Algorithm (GA)

We used an open source GA library (JGAP) to train the final weights of each feature. We started with 1000 candidate chromosomes as the initial population, where 100 are configured with the “elite” results from the PSO, and 900 are configured randomly. The GA was run for 20 generations to get the final weights used for our agent. The fitness function was defined as the cumulative aggregate free space from the top of the board instead of lines cleared.

Our results are as follows:

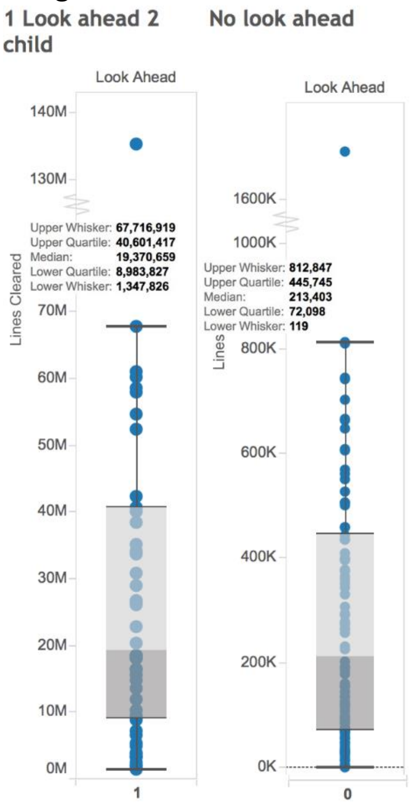
ColTrans	9.994
Holes	9.430
Well	2.804
TopVariety	2.576
RowTrans	2.152
CompleteLines	1.721
AggregateHeight	1.410
MinMaxTop	0.782
LandingHeight	0.441
SideBump	0.296
TopParity	0.109



Absolute values of weights shown to illustrate magnitude.

Of interest is the fact that the weight values for features column transitions and holes dwarf the weight values for the other features.

4 Agent Performance



Graph of scores obtained. Note the different axis magnitudes.

The agent played without look ahead (0 search tree depth) and 1 look ahead with 2 beam width. The 1 look-ahead version is used in the submission. Online sources claim that the best agents routinely achieve tens of millions of lines, and since our agent's performance is in the same order of magnitude, we conclude that our agent's strategy is highly effective.

Comparing mean performance, 1 look-ahead averages a score of ~20M lines while 0 look-ahead averages ~200K lines cleared. From this we can assume that the performance gain from increasing search tree depth is approximately 100x.

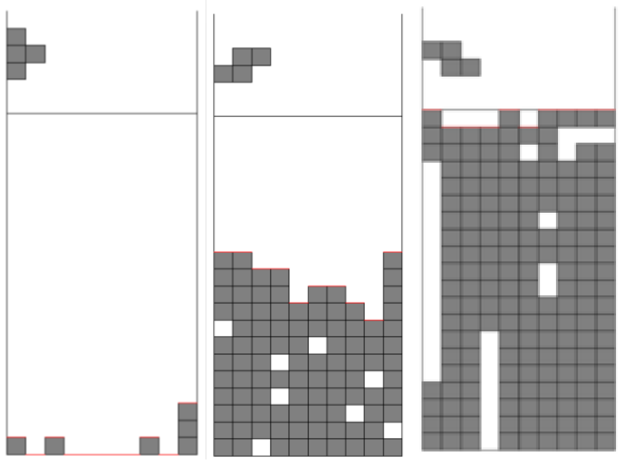
At 2+ search tree depth, the time taken to get statistical data becomes intractable.

5 Novel and Significant Achievements

Our algorithm makes use of two distinguishing features:

A highly customised fitness function and a non-standard representation of the game board.

5.1 Custom Fitness Function for Learning



An agent can collect more reward if it maintains less risky states such as the diagram on the left.

As the candidates get stronger, the time taken to run full games for subsequent iterations of learning increases drastically. To increase learning efficiency we decided to cap each candidate at 10,000 moves played before moving on to the next iteration. The strength of our candidates meant that all of them hit the move cap. However, because each move adds 4 squares to the board and each complete line is 10 squares, the maximum lines cleared at 10,000 moves is 4000, and the minimum 3979. Since it is pointless differentiating ~1000 candidates with only 19 distinct values, we had to come up with our own custom fitness measure.

Our new fitness measure is defined as the accumulated sum of the empty cells above each column every turn. With this fitness measure, candidates that maintain "safer" states over the entire course of play are selected over those that allow the board to fill up more often. The end result is that our agent is optimised for safer and more consistent long-term board maintenance, which may be hard to achieve without a fitness measure that takes into account the play sequence. In other words, our fitness measure trains the agent to implicitly act in a sequential rather than an episodic way.

5.2 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, allowing for a very space inefficient representation of the board.

We implemented the state using a bitboard where each column is represented by an integer and each bit on the integer corresponds to a square on the board. 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 the heuristics much more efficiently using bitwise operations.

6 Scaling up to Big Data

When scaling up to big data, we consider two main factors - *Time* & *Space*. As each chromosome is evaluated with a Tetris game per generation, it is in our interest to minimize the time and space usage in our Beam Search Algorithm.

6.1 Space

In AI, 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.

To test the amount of space saved using our column bitboard, we implemented a board representation similar to the one being used in State.java. We then ran our GA for 100 games and got the results as seen on the right.

Average Memory Usage of Beam Search Algorithm using:

Beam Width	# of look-aheads	State.java	BitBoardCol.java	% Usage Reduced
8	0	11.85MB	7.94MB	32.99%
8	1	194.04 MB	63.18MB	67.44%
3	2	211.8MB	73.39MB	65.35%
4	2	OutOf Memory Error	172.04MB	NA

From this, we can observe that our BitBoardCol implementation provides a sizable memory reduction of ~66%.

Theoretically, this space usage reduction also improves time efficiency by placing less load on the garbage collector and increasing speed gains from cache locality.

6.2 Time

The GA library we used already supports distributed programming out of the box on the population level (Limit is one chromosome per cluster/machine), so we chose to multithread and enhance the evaluation phase on the chromosomal level for the local machine. We multithreaded the beam search implementation and compared it with the basic implementation to test the speedup.

As shown below, the multithreaded version running on 8 cores provides a constant speedup of ~4x, and running on 4 cores provides a speedup of ~2x, so we can extrapolate the speedup to be a function on the number of available cores C: ~0.5*C.

CPU cores	Population Size	Max generations allowed	Time taken (Single) / s	Time taken (Multi) / s	Speedup
4	1000	5	5210	2255	2.31x
4		20	31991	16239	1.97x
8		5	4917	1030	4.77x
8		20	32311	7671	4.21x

7 References

Cingolani, P. *JSwarm-PSO: Swarm optimization package*. Retrieved 10 April 2016, from http://jswarm-pso.sourceforge.net/

Deng, H., Wang, L., Wang, F., & Lei, J. (2009). *Artificial intelligence and computational intelligence* (p. 361). Berlin: Springer.

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

Meffert, K. *JGAP: Java Genetic Algorithms Package*. Jgap.sourceforge.net. Retrieved 10 April 2016, from http://jgap.sourceforge.net/

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.