Tetris Learning Agent Project

CS3243 - Introduction to Artificial Intelligence

Group 11 - Alexander McManus (A0163606H), Carlo Borgo (A0164066E), Jacob Levy (A0164061N), Alec Kretch (A0153131X), Andres Ruiz (A0165056E)

The AI

The user agent uses a heuristic function to evaluate the utility of each possible move given the current state of the Tetris board and the next piece to be played. The move with the largest utility score is used. The features used to determine the utility are described below:

Lines Cleared: Number of lines removed after placing a piece
Holes: Total empty slots with at least one filled slot somewhere above it
Pile Height: The max height reached by all columns
Connected Holes: Holes where vertically adjacent holes are considered as one
Altitude Difference: The height difference between the tallest and shortest column
Sum of Well Depths: A well is found when both adjacent columns have a higher height; depth is the difference between the middle column and the shortest neighbor
Max Well Depth: The maximum well depth found on the entire field
Landing Height: The highest row reached by the most recently placed piece
Block Count: Total number of blocks on the field
Weighted block count: Each block is weighted based on its row, then summed
Row / Column transitions: A transition is defined as changing from a filled space to an empty one as the column/row is scanned top to bottom/left to right.
Eroded Block Count: the number of squares in the current piece being placed to be removed by clearing lines, multiplied by the number of lines cleared.

The final weights used were (corresponding with the above order): 0.3408865068737852, -1.4653636719630416, -0.05293885515293552, -0.034106140141515584, -0.31614303961958556, -0.3805273443398649, -0.23471267918083763, -0.5211479038608587, -0.7165155013131055, -0.08670015348094343, -0.6413714293907122, -1.1182510798482295, 0.12870709131961683

Each feature score is multiplied by a weight (found using the Particle Swarm Optimization Algorithm) and added together. The heuristic can be described by the following equation: $H(i) = \omega(k)*\Phi k(i)$; for state i and features k = 0, 1, …

After testing our learning algorithm, we found the having too few features was only able to net us about 2500 lines. After various tests, we arrived at this final set of features as they provided the best results. We concluded that having more features than needed should have less of a negative impact than having too few as the superfluous ones should level out to 0 by our learning algorithm. Other features were tried (such as holes on the sides of the board), but they had little impact. Due to hardware constraints, we limited our features to ensure some convergence.

Learning - Particle Swarm Optimization (PSO)

To learn the best weights for our heuristic, we implemented a variation of the Particle Swarm Optimization algorithm, detailed in [1]. PSO begins by initializing N particles to include their own solution vector of values randomized between [-1, 1). The solution vector represents the weights given to the decision algorithm. Each particle then plays a series of 10 games and finds the median fitness score based on the: number of rows cleared, average/max height, average/max number of column and row transitions. Fitness was evaluated using more than just rows cleared in order to improve the efficiency of the algorithm in the first few iterations. This allows us to choose between two particles that score similarly, but used different strategies (it is better to keep a low average height than the contrary, for example).

Once all 10*N games are complete, the PSO algorithm updates the solution vectors of all N particles using the equation: $x_i(t+1) = x_i(t) + v_i(t+1)$. x_i represents the solution vector of particle i at iteration t, and v_i represents the velocity vector of particle i at iteration t, defined by the following: $v_i(t+1) = \Phi * v(t) + c_1 * r_1(t) * (y(t)-x(t)) + c_2 * r_2(t) * (\hat{y}(t)-x(t))$.

Φ is the inertia vector, it is responsible for balancing the particle's exploration and exploitation. c_1 and c_2 are the acceleration vectors, and r_1 and r_2 are random values $\in (0,1)$ used to assist in the exploration of new solution vectors. $y(t)$ is the best value particle i has seen in its lifetime (part of the cognitive component). $\hat{y}(t)$ is the best value found by any of particle i's neighbours in their lifetimes (part of the social component). Neighbours are defined using the Von Neumann topology with random node distribution. A delta limit of ± 0.5 is set on each value in the particle's solution vector to prevent converging too fast to a possible local maximum.

These updates continue until a predefined number of iterations has been reached, PSO then searches through all particles to identify the best median fitness score obtained across all games and outputs the solution vector associated with that iteration.

Observations and Limitations

Due to the AI's tendency to play unusually well/poorly in a fraction of the games, our agent must play many games and take the median as its reported fitness. A concern arose in finding a reasonable number of games to play at each iteration. We settled on 10 games per iteration as it allows us to ignore outliers while also not taking too long to execute.

The number of iterations to run PSO was another concession needed to be made. After each iteration, the agent improves its solution vectors for the heuristic, so it progressively clears more lines each game, increasing execution time. A balance had to be found between terminating the optimization after a reasonable amount of time and running it for a sufficient number of iterations to acquire an effective set of weights. Had the hardware constraints been relaxed, we would have removed an iteration cap and replaced it with a threshold of change for the current best solution vector: If the current best particle were to have little change in all of its weights, then we could conclude a maximum has been found.

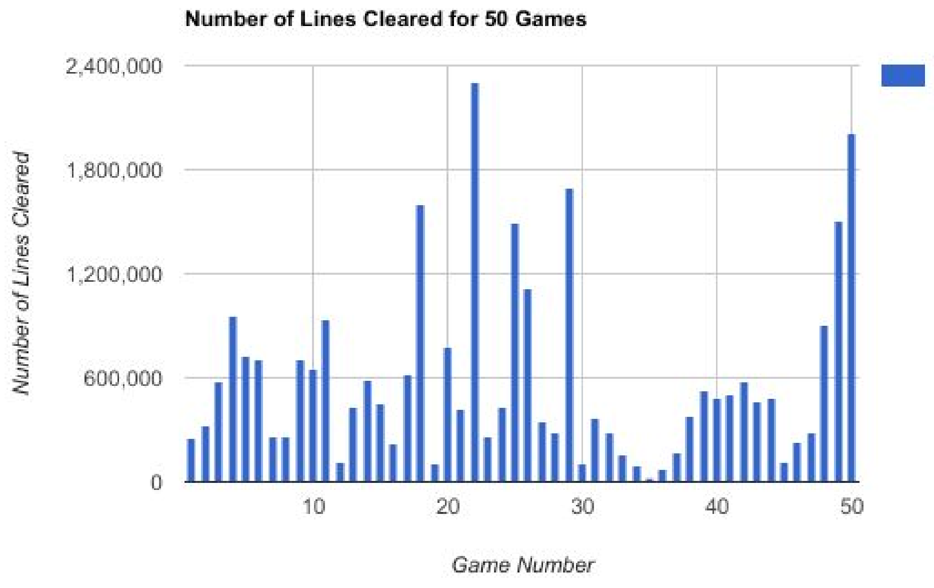
A limitation found in the AI itself is the absence of a lookahead piece (similar to how traditional Tetris is played, where the player is given both the current and next piece). Knowing the next piece would open many more play possibilities and offer an opportunity for better decision making. As this is not the case, the AI agent plays the best move at its *current* state, without considering any future moves. Its only concern is choosing the move that results in best new state.

[1] Langenhoven, Leo, Willem S. van Heerden, and Andries P. Engelbrecht. "Swarm Tetris: Applying Particle Swarm Optimization To Tetris". *IEEE Congress on Evolutionary Computation* (2010): n. pag. Web. 14 Apr. 2017.

Testing

We ran numerous tests on our code to tune the various parameters for both efficiency and optimality. We began by modifying three constants: the number of particles, number of threads, and number of games. We found the number of threads and games played should be close to 10; our hardware could not handle more than those specifications. Subsequent testing was dedicated to determining how many particles to use, we tested 25, 36, 49, 64, 81, and 100 particles.

We discovered that as the number of particles increased (with all else staying equal), we received worse results. This was due to the particles not receiving enough time to converge to a maximum. 25 particles created the greatest average fitness score since the lower number of particles could more easily converge to a local maximum (we cannot guarantee a global maximum, but our algorithm is designed to avoid local maxima as much as possible). We settled on 50 iterations to use with the other variables as this gave us weights that consistently produced high scores and did not require too long to execute. Once the optimal weights were found, we ran 50 games of Tetris and scored an average of 587,825.



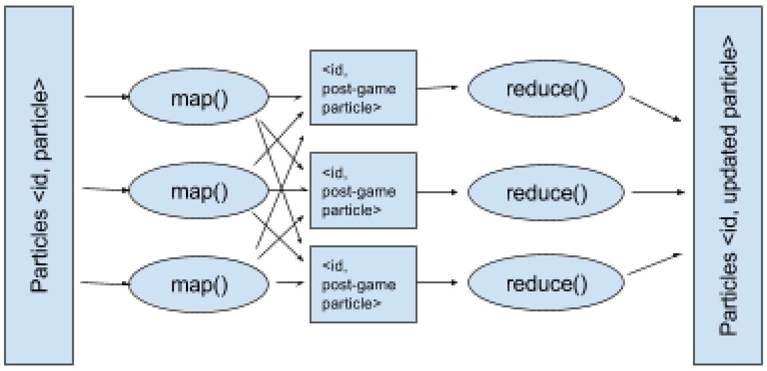
Threading and Big Data

Particle Swarm Optimization is easy to implement using parallel/concurrent computing for Big Data since it relies on discrete and encapsulated objects (the particles and their individual games). We can distribute each object across a cluster of machines and do parallel processing of the data. Multiple frameworks and techniques exist for this purpose, however the most practical is MapReduce.

In our proposed version of the algorithm using MapReduce, each iteration would be equivalent to one application of the map() and reduce() functions. Key-value pairs are <particle id, particle>, <particle id, particle post-game> and <particle-id, updated particle>.

[1] Langenhoven, Leo, Willem S. van Heerden, and Andries P. Engelbrecht. "Swarm Tetris: Applying Particle Swarm Optimization To Tetris". *IEEE Congress on Evolutionary Computation* (2010): n. pag. Web. 14 Apr. 2017.

The map() function runs the tetris simulation using the weights stored in the particles and outputs particles with postgame fitness scores. These are then handled by the reduce() function, which updates the particles by considering its performance as well as its neighbours'. We summarize in the following diagram (omitting the sort step):



We used the Java concurrency package 'java.util.concurrent'. A job is created for each game to be played by a particle, and then waits to be assigned to one of the available threads. The number of threads can easily be increased to utilize more threads/cores in powerful machines or computing clusters.

The main hotspot in the code is the utility calculation, which is very parallelizable. The agent was run on a quad-core computer with the number of threads set between 1 and 3. The average speedup was approximately +0.45 per thread compared to sequential execution.

Considering Amdahl's law: $S_{latency} = \frac{1}{(1-p)+\frac{p}{s}}$ we can use the S_{latency} result previously obtained. If we consider s = 2, then p = 0.62. This means that theoretically the proportion of execution time that benefits from threads is 62%. This is relatively close to the amount of CPU time consumed by thread function calls (~ 65%) when analyzed with a tool such as VisualVM. The learning agent was also ran on the cloud application Heroku and we noticed significant improvements in runtime when compared to execution on our laptops.

Novel Contributions

Our learning algorithm was designed specifically to avoid converging to local maxima, which unfortunately requires a much greater workload. We worked around this by using a non-standard fitness evaluation (incorporating the AI's performance throughout the game, rather than just the end result) as well as by limiting a particle's movement at each iteration and encouraging exploration through random variables. Furthermore, we included many different features in the heuristic to create a more robust algorithm, unfortunately at the cost of convergence speed.

[1] Langenhoven, Leo, Willem S. van Heerden, and Andries P. Engelbrecht. "Swarm Tetris: Applying Particle Swarm Optimization To Tetris". *IEEE Congress on Evolutionary Computation* (2010): n. pag. Web. 14 Apr. 2017.