Improvement on a Tetris Agent

I. Introduction

Tetris is a popular video game developed by Alexey Pajitnov in 1984, that uses tetraminos, geometric shapes that are composed of four squares. The objective is to move each one tetraminos and rotating it by 90 degree units, to create a horizontal line of ten blocks without gaps, each created line thus increasing your score. Any filled row is cleared and the blocks above it drop into the gaps. The game is finished once the screen is filled up, and one of the tetraminos goes over the top border.

![Fig1: An example of a Tetris board](image)

Because of the popularity of this game, many people have tried to develop a Tetris Agent that could maximize the cleared line number in one play on average. For this project, we make the assumption that the algorithm only knows the current piece and has no idea about the following tetraminos in the sequence and that no time constraint are given hence the game stop when the screen is filled. In this project, we implement a Tetris game program and the AI agent using the EI-Tetris algorithm[1], which is an improved Tetris algorithm based on the famous Pierre Dellacherie’s Tetris algorithm[2]. In our implementation, we use the same feature of the evaluation function as in EI-Tetris and optimize the weights by using Particle Swarm Optimization[8] (PSO).

II. Related Works

Many people have analyzed Tetris and designed some Tetris playing agent. Demaine et al.[3] shows that the solution of Tetris is NP-complete. In 2003, Pierre Dellacherie developed a one-piece algorithm which clears 650,000 lines in average[2] based on an hand-coded heuristic. Some more sophisticated approaches were implemented: Böhm et al.[4] presented a genetic algorithm that performs on average just below Dellacherie’s algorithm, Driessens[5] applied reinforcement learning to create Tetris playing agents and Szita and Lőrincz[6] and Chen et al.[7] improved this idea by using cross-entropy learning method and applying Ant Colony Optimization method. But, even if compared to the hand-coded algorithm[2], learning methods have more flexibility and are capable of adjusting their policies, these algorithms are still outperformed. Recently, Islam El-Ashi[1] improved Pierre Dellacherie’s algorithm by using particle swarm optimization[8] to compute the evaluation function.

III. Approach

Our implementation of the Tetris game is based on a graphical implementation of a Stanford Tetris Project by Nick Parlante[9]; it enables the user to play a standard game of Tetris, to change the speed of the pieces and to let the brain, the AI agent, play the game, as seen in the following figure.

![Fig3: Example of our broad game](image)
Improvement on a Tetris Agent

We use the same features as EI-Tetris algorithm used:
- The height where the piece is put (= the height of the column + (the height of the piece / 2))
- The number of rows eliminated.
- The number of row transitions: when an empty cell is adjacent to a filled cell on the same row.
- The number of column transitions: when an empty cell is adjacent to a filled cell on the same column.
- The number of holes: an empty cell that has at least one filled cell above it in the same column.
- The well sums: a succession of empty cells such that their left cells and right cells are both filled.

In our project, we are using PSO to optimize the weight of each feature in Tetris, hence we are using a swarm of 25 particles where:
- the position of each particle is a set of weights of the features in the evaluation function,
- the velocities of each particle are the variation between two iterations.

The position of each particles is evaluated in each iteration by using the number of rows cleared return by our Game Simulator with the given position as the set of weights.

The positions and velocities are generated randomly at the first then, at each iteration, the algorithm find the best weights (i.e. the position where the number of cleared rows is the biggest) of the entire swarm and update the global best position. The velocities of each particle are then adjusted based on current position and the global best position by using PSO rules[8]. After several iterations, the swarm reaches an approximate global optimal position which is the optimal weights of features. We return the global best position after five iteration as an approximation of this optimal weights.

IV. Evaluation

The weights of the features are computed by setting the number of particles to 25; the result is the global best position of the Particle Swarm Optimization after five iterations. The following table compares our own results and the optimal weights obtained by El-Ashi in [1] which also use the PSO.

<table>
<thead>
<tr>
<th>Features</th>
<th>Weights computed by El-Ashi</th>
<th>Results of our computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Height</td>
<td>-4.5001588</td>
<td>-3.3200740</td>
</tr>
<tr>
<td>2. Number of eliminated rows</td>
<td>3.4181268</td>
<td>2.70317569</td>
</tr>
<tr>
<td>3. Number of row transitions</td>
<td>-3.2178882</td>
<td>-2.7157289</td>
</tr>
<tr>
<td>4. Number of column transitions</td>
<td>-9.3486953</td>
<td>-5.1061407</td>
</tr>
<tr>
<td>5. Number of holes</td>
<td>-7.8992654</td>
<td>-6.9380080</td>
</tr>
<tr>
<td>6. Well sums</td>
<td>-3.3855972</td>
<td>-2.4075407</td>
</tr>
</tbody>
</table>

Table1: Computed weights of the features
The performance of the Tetris agent is evaluated by the number of cleared rows, given for ten games on a standard 10*20 board. The results are compared with the El-Tetris algorithm[1] and the Dellacherie’s algorithm[2] in the following figure that have been computed using the mdptetris[11] implementation in C. Figure 4 gives the results of 10 games using this three different algorithms.

![Figure 4: Number of cleared rows for ten standard size game](image)

Table 2: Number of average cleared rows

<table>
<thead>
<tr>
<th>Method</th>
<th>Average Cleared Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our result</td>
<td>13,999,235</td>
</tr>
<tr>
<td>Dellacherie[2]</td>
<td>5,024,731</td>
</tr>
<tr>
<td>EI-Tetris[1]</td>
<td>16,047,595</td>
</tr>
</tbody>
</table>

IV. Discussion

We have chosen to develop a version of the El-Tetris algorithm in Java to avoid the implementation of the Particle Swarm Optimization and be able to use the JSwarm library for our work. Nevertheless this choice had an important repercussion on the running time of our algorithm.

Hence our implementation of the AI agent can take about 30 seconds to play 100,000 pieces. To obtain weights in a reasonable time, we can only calculate less than 6 iterations in PSO because it take many hours for each iteration after the 4th iteration. The time cost also limit the number of particles in our PSO. Hence our result are only an approximation of the optimal value that could be reach by the PSO. But the difference between the weights we obtained and the once computed by El Ashi in the Table 1 is also due to the random initialization.

The Figure 4 shows the good results we obtained for some of the 10 standard games, with in the best case about 40 millions rows cleared by our AI Agent.

Moreover our average results is close to EI-Tetris algorithm. This running time is the principal cause that the result of our algorithm is not as good as the El-Tetris algorithm as seen in Table 2. However, we still obtained relevant results for the Tetris agent that outperformed the Dellacherie algorithm, which used similar features but where the weights are given by human experience.

A first improvement of our algorithm would be the development of a more efficient function to determine the impact of the weights on the evaluation function, instead of using the cleared rows of an end game.

Another direction to save computational time could be the parallelization of the algorithm to be computed simultaneously on different cores or computers.

Another additional work would be the addition of some more features to the El-Tetris algorithm, like:
- The altitude difference: the difference between the highest occupied and lowest free cell that are directly reachable from the top or,
- The maximum well depth.

It could be interesting to measure the impact of each new feature on the algorithm and evaluate if the evaluation function could have been improve or not. We initially wanted to implement that but the computational time is here again a restriction and even, if we add some features, we may not going to be able to compare properly the performance to the other algorithms.

References