CS3243: Introduction to Artificial Intelligence

Group 23
Marchal Nicolas (A0174733B) - Vandenberghe Loïc (A0174721H) - Lou Chun Kit Ryan (A0121352X) - Ooi Heng Seng (A0160376E) - Alex Fong Jie Wen (A0139019E)

1 Introduction

We will develop an AI agent that can play the popular game *Tetris* (1984). Our team has come up with an innovative idea by using simulated annealing to teach the computer how to play the game in a minimum amount of time. We then play to minimize the damage in case the next tetromino (Tetris piece) given is the worst possible.

2 Game Strategy

In Tetris, the number of states $\approx |S|2^{hw}$ where h and w are the height and width of the grid and S is the finite set of 7 different shapes allowed. In our project this corresponds to $7^{20 \cdot 10} \approx 10^{61}$. For a comparison, the earth "only" has around 10^{50} atoms.

The large amount of states doesn't allow us to search through all of them, so we use a heuristic function to reduce the search domain. This is known as *"Informed Search"*, which relies on the heuristic function to evaluate the board configuration (using key features) and chose the best move to play next.

Although there is no real opponent while playing Tetris, we can consider that an opponent is chosing the next tetrominos to play. We can therefore use *adversarial search*, more precisely the MINMAX algorithm, to anticipate undesired tetrominos. This consists of choosing the move, that guarantees the least damage if the random choice is the worst. As seen in section 4 this makes the agent perform more than 600% better! However, the MINMAX searching time blows up exponentially when we increase the depth of the search. To address this issue, we use $\alpha - \beta$ pruning algorithm. This reduces the search time by over 80%, as discussed in section 4.

3 Weight Learning

Setting the correct parameters and their associated weights is the key to this project. First of all, we have added the following features to the heuristic given in the project description

- **Sum of Heights** of each column : More lines cleared will have a smaller sum of heights. This heuristic encourages clearing lines even if the number of holes does not decrease.
- **Maximum Height Difference** : It discourages moves that increase the maximum height of the field, and encourages moves that raises the minimum height. While similar to ϕ_{11} to ϕ_{19}, this heuristic targets flattening the field, rather than flattening adjacent columns.
- **Grouped Holes** : This encourages grouped holes as opposed to separated holes, following the idea that grouped holes are easier to fill up with tetrominos.
- **Squares Above Holes** : It is the sum of 'number of squares above the lowest hole' for each column. This discourages moves that shift holes further from the surface of the field.

To decide the relative importance of each parameters of the Heuristic, we selected the weights by using reinforcment learning. Because the tetrominos are chosen randomly, the score performed by an agent is not constant over several games. We therefore choose to evaluate the performance of an agent using an average score over 20 to 30 games. In order to accelerate the process of learning, we choose to train the informed search agent instead of the MINMAX agent. We assumed that having a heuristic that performs well for an agent anticipating only one move (informed search) could then also be used for a MINMAX agent looking several step forward.

In the literature, neural networks and genetic algorithms (GA) seemed very popular for this task. After implementing these methods, we suffered from a long training time. GA is a slow learner because the entire population has to play in order to evaluate the performance. Our objective was to optimize the training time, so we decided to use a novel approach which is simulated annealing. Our implementation of simulation annealing was indeed faster than when we tried GA (see 4.1).

3.1 Our Contribution

We decided to use what we call *"Home Made"* simulated annealing. The specificity is that we have designed and coded all of our algorithm from scratch. The idea is to use only one agent and to randomly modify some of the weights of the heuristic to evaluate if the results are better. Although its learning performance are influenced by randomness it is the algorithm giving us the best result in the shortest amount of time compare to all our other tries. The pseudocode is given below :

Algorithm 1: *"Home Made"* Simulated Annealing

Input: Heuristic H, number of training iterations $maxIter$, maximum number of lines for training $maxTrain$, number of games to average n

for *i from 0 to maxIter* **do**
- $numberOfChange \leftarrow 5 \cdot [1 - \left\lfloor \frac{i}{maxIter} \right\rfloor]$
- $deltaParam \leftarrow e^{-\frac{i^{1.2}}{maxIter}}$
- $newH \leftarrow$ **newHeuristic**(H, $numberOfChange$, $deltaParam$)
- $newAvrg \leftarrow$ **playGame**($newH$, $maxTrain$, n)
- $P \leftarrow$ **acceptProba**($avrgScore$, $newAvrg$, i)
- **if** *newAvrgScore* $>$ *currentScore* ***or*** *with probability* P **then**
  - $H \leftarrow newH$; $currentScore \leftarrow newAvrg$

return H;

The number of weights chosen to be randomly modified ($numberOfChange$) decreases linearly. The idea is simple: at first we try to move 5 weights and hope to randomly get in the direction of a better solution. As we play more, we should converge towards a good solution, so we change less weights to simply adjust the model.

For the same reason, we decrease the magnitude of the change ($deltaParam$) using the exponential law $e^{-\frac{i^{1.2}}{maxIter}}$. This function has been empirically chosen by us after trying several different functions with MATLAB. The exponential function rapidly stops large changes but still maintains a minimum increase after a large number of tries. We then change $numberOfChange$ weights to get a new heuristic (see pseudocode of `newHeuristic()` in appendix A.2) and we compute the average over $n = 25$ games.

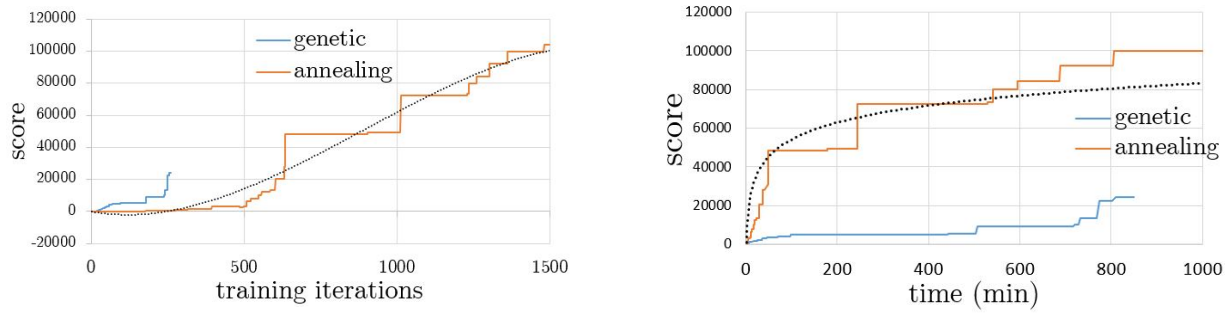
Moreover, we added an optimisation to compute the average. After hundreds of observations, we have seen that if after 5 games the average is less than $\frac{2}{3}$ of the current best score, it is very unlikely to return a better value. This happens almost 80% of the time. In this situation, we stop and return the average computed with only 5 games as an estimation. This is one of our biggest innovation as it decreases the training time by 65%.

Finally, we accept the a model if it performs better than the old one. To avoid getting stuck in local maxima, we also accept worse models with some probability P. The choice of the probability P must assure that we can escape local maxima without decreasing the performance. This probability depends on the relative error err (the larger the error, the lower the probability of accepting a new model as it is harmful). The probability must also decrease with the number of training iterations i already executed (at first there are high chances of local maxima but we should ultimately converge towards the best solution). We therefore used an exponential law as we had seen for simulated annealing in lecture (03-Informed Search - slide 41). We chose the law $e^{-err^{3.1} \cdot i^{1.2}}$. After many tries using MATLAB we selected this function that penalizes large errors. The pseudocode of `acceptProba()` as well as the behaviour of the probability P function for different relative errors is shown in appendix A.3.

4 Overall Performance

4.1 Time performance

As we previously said, we used simulated annealing as it was much quicker to train. When comparing simulated annealing and our version of genetic algorithms (GA), we see that although GA need less iterations to get better (figure 1a), it takes more time (figure 1b). This confirms that with the computing power at our disposal (personal computers) simulated annealing is our best option.



(a) Iterations.

(b) Time.

Figure 1: Training Time and Iterations for Simulated Annealing and Genetic Algorithm.

As we see in figure 1a,when we stopped the learning algorithm, the score still tended to increase with the number of training iterations. However, we are more interested in the training time as shown in figure 1b, where the trend line starts to plateau. Because the relative increase in performance also seems to drop over time (see figure 2) it motivates us not to leave the algorithm run for days. Our goal was to train the model in approximatively one night (12 hours), which is the reason why we stop the training algorithm after reaching an average of $\sim$ 100'000 lines even if better performance could be expected with more time.

In addition, we would like to emphasize that the learning efficiency is strongly influenced by chance. A lucky choice of parameter change can increase the average by more than 50% as seen in figure 2. The variation of the score with different random choices of weight variation is shown in figure 3. It is important to know that simulated annealing is supposed to find the optimal solution given an infinite time, which we do not have. Therefore, the learning algorithm has sometimes ended learning only a few thousand lines. In this report we present our best result.

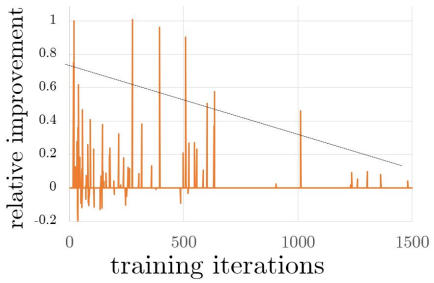


Figure 2: Learning Rhythm.

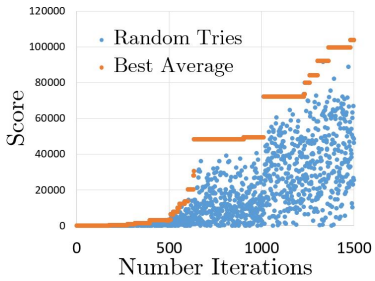


Figure 3: Random Scores.

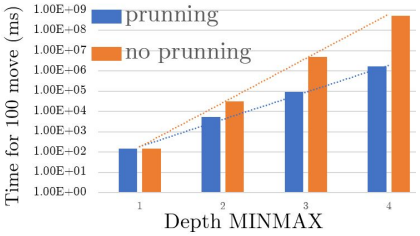
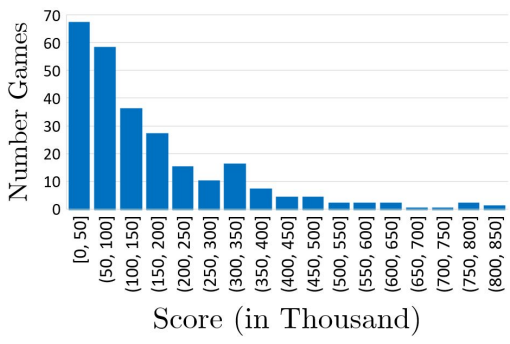


Figure 4: Prunning.

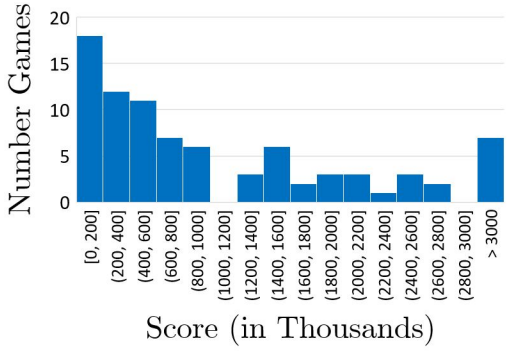
Moreover, we evaluated the time taken to do 100 moves using MINMAX algorithm (figure 4). We start by evaluating the time without pruning and we see that as the depth increases the time increases exponentially. Using a logarithmic scale, we get a linear relationship which confirms that the time complexity is exponential cst^d with the depth d. The cst is the slope shown in figure 4 and is around 150 without pruning. On figure 4, we also analysed the impact of $\alpha - \beta$ prunning for MINMAX. The time complexity is now in the order of 20^d. Although this is unfortunately still exponential, the time is reduced by $\sim$ **86%** (for $d = 2$)! This allows a depth of 2 to be quick enough to play a thousand move in less than a minute and it can play with a depth of 3 at the speed of a human ($\approx$ 1 move/s).

4.2 Score performance

We see the performance of our informed search player in figure 5a. We first notice that the performance varies a lot from one game to another because of the randomness in the selection of tetrominos. After playing 250 games, we get an average of 149'000 and a median of 100'500. These score are improved by using the MINMAX algorithm with a depth of two (figure 5b). By playing 80 games we get an average of **1'120'000** (+650%) and a median of **700'000** (+600%). Moreover, we played 58% above 500'000, 36% above 1'000'000 and 8% above 3'000'000. Our best score reached 5'744'006.



(a) No MINMAX.



(b) MINMAX.

Figure 5: Playing Performance.

4.3 Additional Improvements

As a final time improvment, we used parallel computing to fully exploit our hardware. This allows us to evaluate simultaneously different states for the heuristic and reduces the time to play by around **27%** for simple inform search. However, it can not be used for the MINMAX algorithm as it is not compatible with pruning. As pruning reduces the time by 86% for a depth of 2 (and 98% for a depth of 3), it is more efficient to use it rather than parallel computing.

Secondly parallel computing can be used to play 4 games simultaneously when evaluating the average performance. This only resulted in a **13%** decrease in learning time when run with a 2 cores processor. The gain is lower than expected as the PC was already working at full capacity. However, the gain would be much higher with more powerful computers and could help scale our algorithm to big data.

Additional algorithms such as Particle Swarm Optimization (PSO), neural networks and Temporal Difference Reinforcement Learning (TD) were unsuccessfully attempted. Genetic Algorithm was successfully implemented but abandoned because of too long training time.

5 Conclusion

Our team has used an innovative approach to speed up the learning of the heuristic function. We also took an interesting approach by considering the random selection of tetrominos as an opponent. We used the minmax algorithm to boost our performance, and made sure to use pruning to decrease the search time.

Our code has been written entirely by our team, built on pseudo codes given in lectures. Developing our own idea based on our intuition and the course material is one of the specificity of our project.

Overall, we reduced the learning time by 70% (reduce average + parallel games) and the playing time with MINMAX (depth of 2) by 86% (pruning). We also made it possible to play using a depth of 3 at a human speed ($<$ 1s/move) by reducing the search time by 98% (pruning). On average we clear 1'200'000 lines.

Appendices

A Pseudocodes

A.1 playGame()

Algorithm 2: playGame()

Input: Heuristic H, maximum number of lines for training $maxTrain$, number of games to average n, current score of best heuristic $avrgScore$

$sum = 0$;
for *nbrGames from* 0 *to* $maxTrain$ **do**
    play a game ;
    if *lose before reaching* $maxTrain$ **then** score = game result;
    else stop playing at $maxTrain$; score = $maxTrain$;
    $sum \leftarrow sum + score$;
    $newAvrg = \frac{sum}{nbrGames}$;
    if $avrgScore > 400$ *AND after five games :* $newAvrg < \frac{2}{3} avrgScore$ **then**
        return $newAvrg$
return $newAvrg$;

A.2 newHeuristic()

Algorithm 3: newHeuristic()

Input: original heuristic H, number of parameters to change $numberParam$, magnitude of the change $deltaParam$

for *numberIter from 0 to* $numberParam$ **do**
    randomly select 1 weight $weight_{selected}$ in the heuristic H ;
    Randomly choose the sign $\pm$ for $deltaParam$;
    $weight_{selected} \leftarrow weight_{selected} \pm deltaParam$;
$newH = H$;
return $newH$;

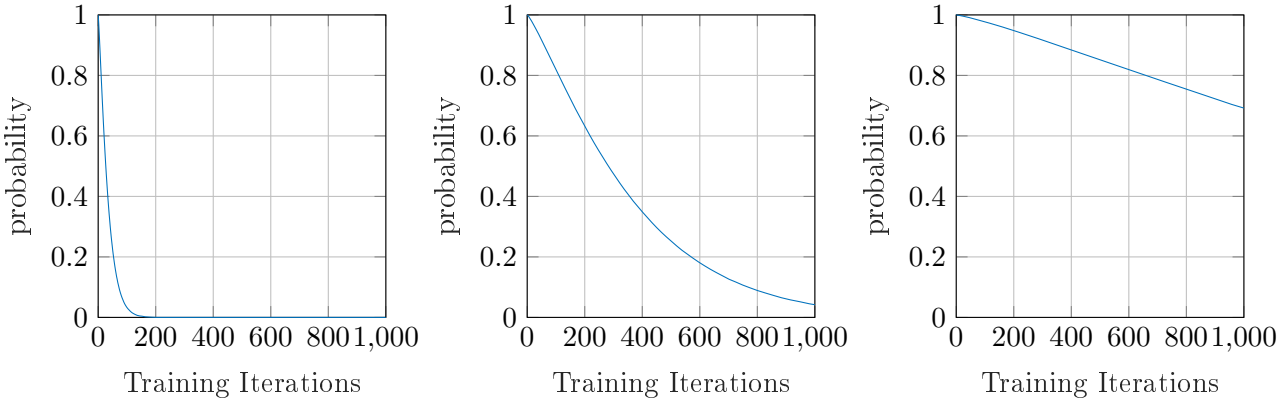
A.3 acceptProba()

Algorithm 4: acceptProba()

Input: score of original heuristic $avrgScore$, score of new heuristic $newAvrg$, number of training iteration done so far $iterSoFar$

\# compute the relative error of the new heuristic
$error = \frac{avrgScore - newAvrg + 1}{avrgScore}$;
\# use exponentially decreasing law
$proba = e^{-error^{3.1} \cdot iterSoFar^{1.2}}$;
\# set maximum probability of 50%
if $proba > 0.5$ **then** $proba = 0.5$;
return $proba$;

In figure 6, it is clearly shown how the probability is closely linked to the number of games and the error. As the learner plays more and more games, the probability of accepting worst results drops. Furthermore, we see that this probability drops much more when the error is higher. Moreover, at the beginning there is a high chance to switch to a worst model, but in the end the probability of changing is reduced (as we should converge to a global minimum). For large numbers of iterations, the probability of accepting a worst model is only high if the error is small (see 6c).



(a) error of 25 %. (b) error of 10 %. (c) error of 5 %.

Figure 6: probability function for different errors.