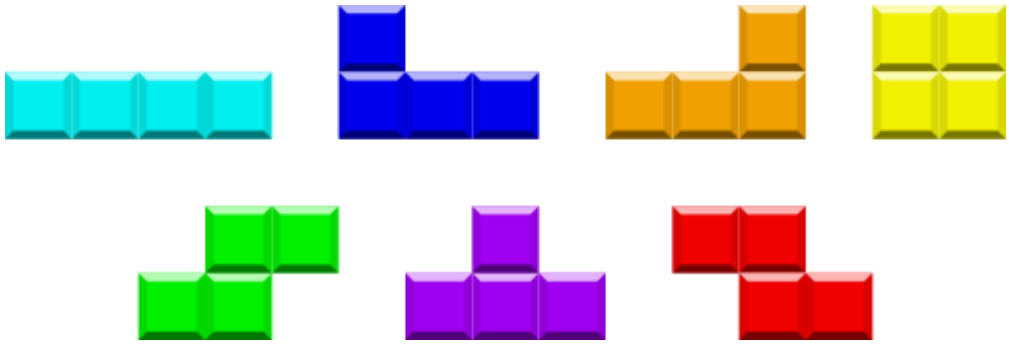


Project - The Tetris Master

Computer Models of Cognitive Processes

Simon Bergström & Robin Berntsson
MT3A

Abstract

This describes a project work in the course TNM066 Computer Models of Cognitive Processes at Linköpings Universitet. The goal was to create a program that could play the game Tetris. Since the knowledge in this area was low before the start of the project we didn't have many solid goals from the start. We wanted to learn about genetic algorithm which is the technique behind the artificial intelligence in this program.

Tetris is a game that can be controlled by certain rules. These rules get numerical values and those values make the program find the best move. The genetic algorithms job is to find the optimal value for these rules.

Table of Contents

Introduction

Genetic algorithms are good to solve e.g. global optimization problems and are useful in many different areas. Our goal was to solve a specific problem with help of this algorithm.

To create a controlling program that could play the game tetris was the problem that we chose to solve. The control or AI (Artificial Intelligence) program should make moves in the game that seemed logic most of the time.

To have full control of how this AI program should be designed, an own version of the Tetris game was programmed. Then the AI program for controlling the game was created.

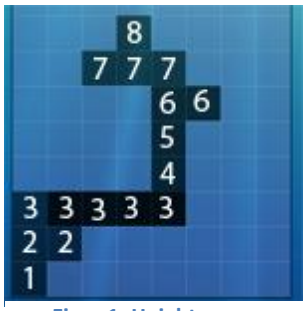
Theory

Tetris

Tetris is a tile-matching puzzle game that involves getting as high score as possible. In the start of the game you have an empty board and one tile at a time is raining down from the top of the board. To get score you should place these tiles in the board to you get a full row then the row gets deleted and you get points. The game is over when you have build so high with the tiles so one piece touches the roof of the board. Combo points are possible to get if you delete more than one line at the same time.

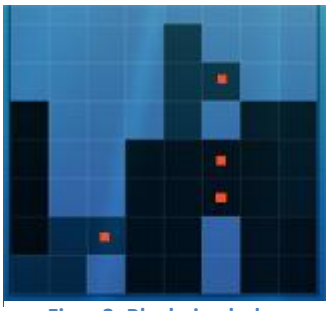
AI

To get the AI to play Tetris it has to know where to put the pieces; this is decided by evaluating every possible position for the current piece and calculating that positions score. The score is determined by seven factors: the height, touching other blocks, will it produce a block, will it produce a hole, number of cleared rows, is it touching a wall and is it touching the floor (figure 1-7) [3].
The dots in different colors (figure 2-4,6-7) shows which part of the tiles that is taking in to calculation according to the different factors.

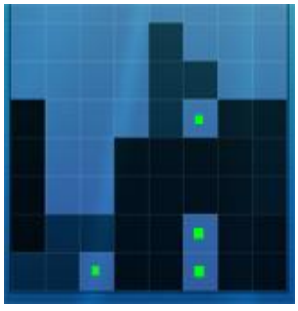


Figur 1. Height score

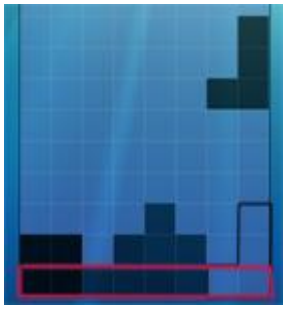
Figur 2. Touching other blocks



Figur 3. Blockning holes



Figur 4. Holes



Figur 5. Clearing row

Figur 6. Touching wall

Figur 7. Touching floor

$$Score = A * Height + B * Holes + C * Blocks + D * ClearedRows + E * Wall + F * OtherBlocks + G * Floor$$

(Eq.1)

The values of { A,B,C,D,E,F,G } in Eq.1 tells the AI how important the corresponding factor is. After calculating the scores for all positions the AI places the block at the position with highest score.

Genetic algorithm

As a human it is impossible to know what values the previous mentioned constants should have to optimize the AI. A way to solve this problem is to use genetic algorithms (GA). GA's is a search heuristic that mimics the process of natural evolution. It utilizes the fact that evolution is an optimizing process. The basic of GA's is a population consisting of individuals. In our case an individual represent a set of the previous mentioned constants (figure 8).

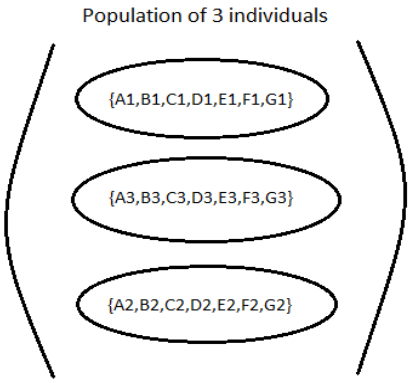


Figure 8

The constants can be seen as a setup of chromosomes and will be called this further on. Just like in nature only the strongest individuals can survive and pass its genome to the next generation. Each individual passes a fitness test and gets a fitness rank. In our case the fitness test is how high score the Tetris AI can get. The best individuals are allowed to reproduce and create the next generation. If done right each generation will be closer to the optimal solution then the previous (figure 9).

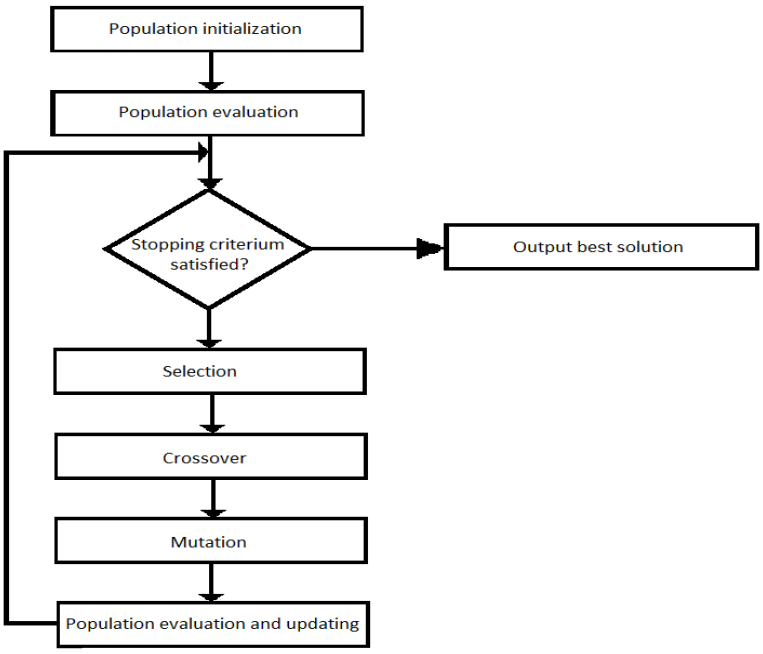


Figure 9. GA flow chart

Population

The size of the population depends on the amount of chromosomes. To be able to have a large diversity it is good to choose the size if the population to the square of the number of chromosomes [4]. There are different ways to choose the initial values of the chromosomes. One method is to pick random numbers; this is a good method if there is no knowledge about the optimal values. If it is known in what region the optimal numbers is, choosing the numbers in that area may be the best solution.

Selection

There are numerous ways to select individuals which individual that is allowed to mate. The best method depends on the problem to be solved [5]. To find a good selection strategy it is often needed to test different ones and see which one is the best.

The different selection methods have different strengths and weaknesses. Fitness proportionate methods, such as roulette wheel selection approaches a good solution faster but sacrifices diversity in the population, therefore the chance to get stuck in a local maximum is high. On the opposite rank roulette wheel selection method gives chromosomes with low fitness higher chance to mate but approaches a good solution slower.

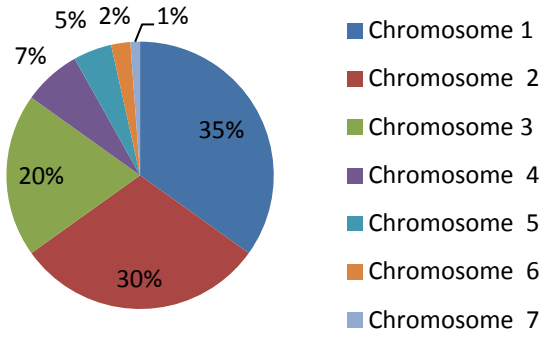


Figure 10. Roulette wheel selection

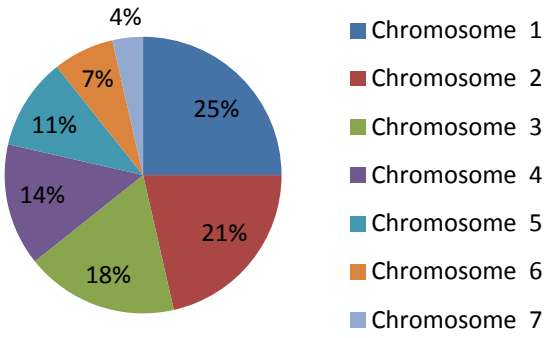


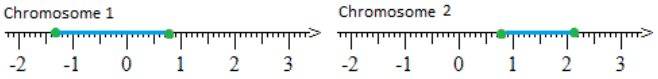
Figure 11. Ranked roulette wheel selection

The figures above (figure 10-11) show how the ranked roulette wheel selection method spreads the probability to mate more evenly.

To prevent the risk of losing the best individual elitism is often used (Figure 8). That is always allowing the individual with the best set of chromosomes to pass to the next generation unaltered.

Crossover

After the selection phase comes the crossover phase. This is the phase were two of the selected individuals reproduce two new ones. Such as *Selection* there is a wide range of different crossover methods. An easy method is to use a blend recombination; it takes the values of the parent's chromosomes and randomizes a value in between them (figure 12).



Figur 12. Crossover method

The green dots (figure 12) represent the value of one parent and the red dots the value for the other parent. The blue line between them is the range of possible values for the child. Each pair of parents produces two children so the population maintains the same size in every generation.

Mutation

The crossover algorithm mentioned above averages the value of the parents; this makes the algorithm prone to getting stuck with a bad set of chromosomes. To prevent this mutation is introduced. At every crossover there is a chance for a chromosome to mutate. If it mutates, it is assigned a random value.[6] It is important to choose the correct mutation rate as to high frequency may ruin the good solutions and a too low rate will make a too low of impact on the next generation.

Design

The program is implemented in Java. To create an environment which is easy to understand and simplifies for further development the design of the program is as follows [2]:

The program is divided in five different classes with a certain hierarchy (figure 13). The Game is divided in three classes named Game, Board and Pieces. They have all information that is needed to

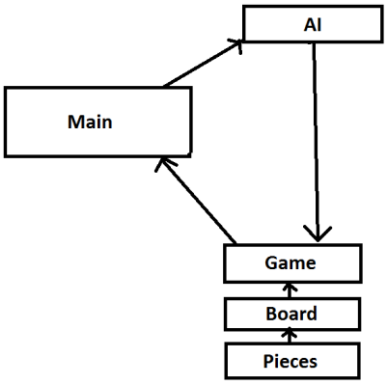


Figure 13

play the game with the keyboard. Then we have an AI class that handles the computer movement of the pieces and the genetic algorithm (fig 9). The main class contains the game loop and links the AI with the game.

Method & result

The following values of the different variables that have been used in this project are:

Variable	Value
Number of generations	100
Size of population	50
Mutations rate	0, 01 (1 %)
Number of chromosomes	7

The initial values of the chromosomes were initialized with values between -5 and 5. For the selection the ranked roulette selection method was used and for the crossover the blend recombination.

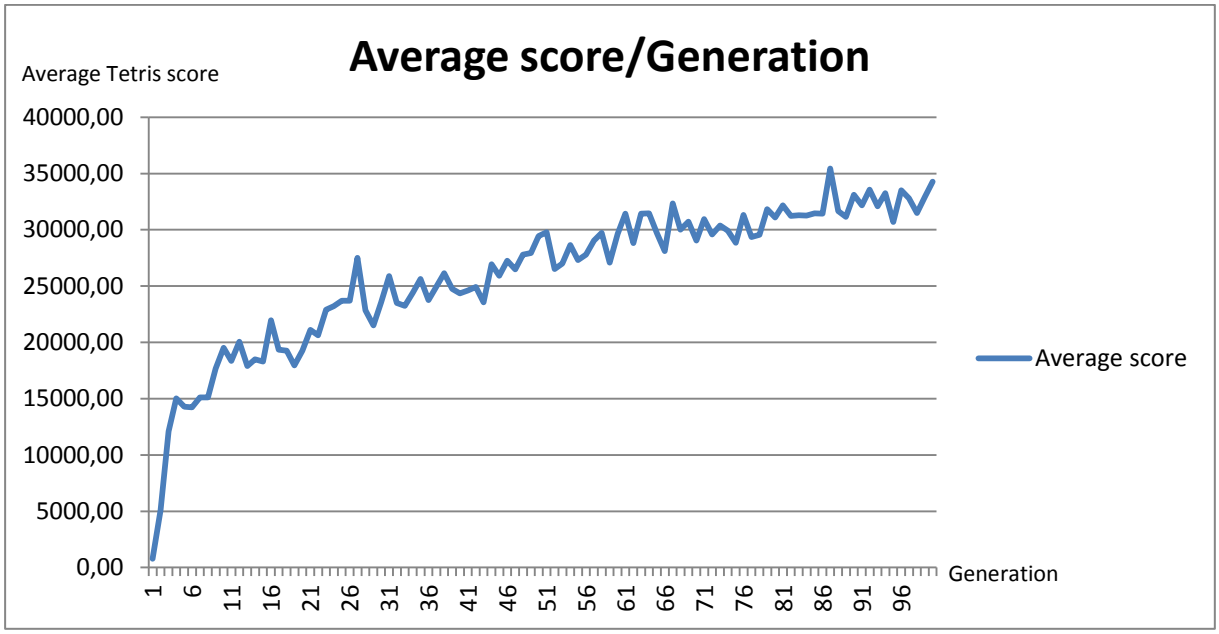


Figure 14

Figure 14 shows us the average score in the Tetris games for every generation and how it increases for every generation that calculates. It's also quite clear that the value converges against a certain maximum value for this method.

Discussion

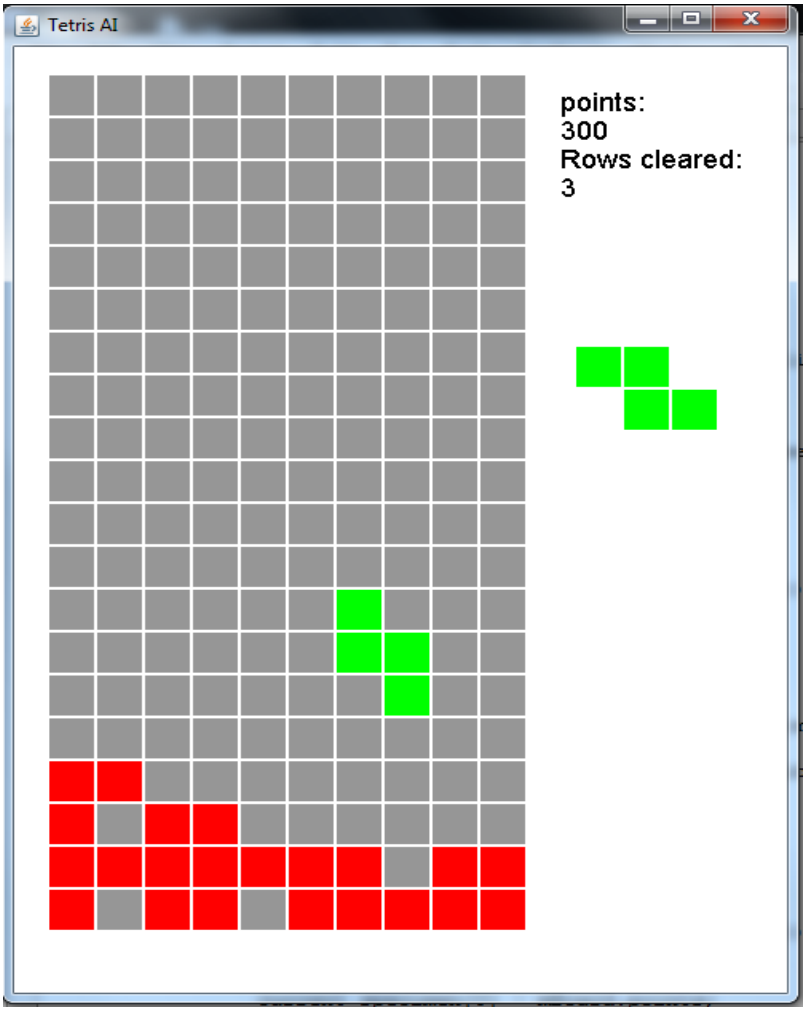
The result we got was good. We found values for the AI to make him play as a good player. We have not tried all the different methods and values of certain variables that maybe could increased the result.

100 generations is not enough to get a “perfect” result, but we choose to run with it because as the AI becomes better at the game every turn takes longer and after a while each generation takes really long time to run. 100 generation takes about 10 minutes to execute with a regular computer but if you increase to 1000 generation it will take about 8-9 hours to execute.

One thing to have in mind when interpreting this result is that Tetris is a random game. The best AI’s does not always get the best result because sometimes they get unlucky and get a hard game with many difficult tiles.

It’s important to see that this AI problem is not just all about finding the best methods for the GA. How the game is controlled is just as important and adding more chromosomes to control the program may increase the AI player by far.

From genetic algorithms we have learned that it could be a good method to use in many different optimization problems where there is a complex search space. The problem of creating a good Tetris player can for example been seen as an optimization problem for packing in boxes.



References

[1] https://wiki.engr.illinois.edu/display/cs598lvkfa10/Tetris+Game

[2] http://javilop.com/gamedev/tetris-tutorial-in-c-platform-independent-focused-in-game-logic-for-beginners/

[3] http://luckytoilet.wordpress.com/2011/05/27/coding-a-tetris-ai-using-a-genetic-algorithm/

[4] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1688360

[5] http://www.ijest.info/docs/IJEST11-03-05-190.pdf

[6] http://www.obitko.com/tutorials/genetic-algorithms/crossover-mutation.php