People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University M'Hamed BOUGARA – Boumerdes

Institute of Electrical and Electronic Engineering
Department of Electronics

Final Year Project Report Presented in Partial Fulfilment of the Requirements for the Degree of

MASTER

In **Electrical and Electronic Engineering**
Option: **Computer Engineering**

Title:

Playing Tetris Using Genetic Algorithms

Presented by:

- **BENARBA Abdelkarim**

Supervisor:

Dr. M. KHALIFA

Registration Number:………./2016

DEDICATION

I dedicate this modest work to my parents, my brothers and sister, and to all my family and friends with whom I share the good and the bad.

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Dr. Khalifa for the continuous support and to all my friends and colleagues who helped during the making of this project

ABSTRACT

This project discusses the training of a one-piece Tetris playing AI using the general optimization algorithms "genetic algorithms". The player AI is implemented with two evaluation functions (exponential and linear) optimizing a set of 10 features. This player and the genetic algorithm to train it are built using only C++11 standard library. Limited to 1000 moves, the two players resulting from the training using the exponential and linear evaluation functions had average results of 381 and 421 moves, respectively, and a respective average score of 2707 and 2874. The two methods gave good results compared to the time constrains, and in the case of this project their results are very close.

TABLE OF CONTENTS

DEDICATION ……………………………………………………………………………I

ACKNOWLEDGEMENTS ………………………………………………………..……...II

ABSTRACT …………………………………………………………………………….III

TABLE OF CONTENT ……………………………………………………………...IV

LIST OF FIGURES ……………………………………………………………………VI

LIST OF TABLES …………………………………………………………………...VI

GENERAL INTRODUCTION ………………………………………………………………1

CHAPTER ONE: Introduction to the problem of playing Tetris

1.1.Introduction ……………………………………………………………...……..…2

1.2.Tetris ………………………………………………………………………..……..2

1.2.1. Background ……………………………..……………………………..……2

1.2.2. Standards and gameplay …………….…………………………………..….3

1.2.3. Why Tetris? ………………….…………………………………………..……5

1.3.Game playing using AI ………………………………………………….………..5

1.4.Conclusion ……………………………………………………………….…………6

CHAPTER TWO: Genetic algorithms

2.1.Introduction ………………………………………...………………………7

2.2.Background …………………………………………………………………………7

2.3.A simple genetic algorithm ……………………………………………………………7

2.4.How do genetic algorithms work? ………………………………………………….8

2.5.GA operators …………………………………………………………………………..8

2.5.1. Selection ……………………………………..……….……………..9

2.5.1.1.Roulette Wheel Selection ……………………………………………………9
2.5.1.2.Elitism …………………………………………………………………11
2.5.1.3.Rank Selection ……………………………………………………………11
2.5.1.4.Tournament Selection ……………………………………………………12
2.5.1.5.Steady-State Selection ……………………………………………………12
2.5.2. Crossover ………………………………………………………………………13
2.5.3. Mutation ………………………………………………………………………16
2.6.Advantages and disadvantages of GA ………………………………………………17
2.7.Conclusion …………………………………………………………………………17

CHAPTER THREE: Design and implementation of the Tetris player

3.1.Introduction ……………………………………………………………………………18
3.2.Method …………………………………………………………………………………18
3.2.1. Player Design ………………………………………………………………………18
3.2.2. Genetic Algorithm Design …………………………………………………………21
3.3.Results …………………………………………………………………………………23
3.3.1. Exponential evaluation function ……………………………………………………23
3.3.2. Linear evaluation function …………………………………………………………24
3.4.Discussion ……………………………………………………………………………26
3.5.Conclusion ……………………………………………………………………………26

GENERAL CONCLUSION ……………………………………………………………27

REFERENCES ……………………………………………………………………………28

LIST OF FIGURES

Figure 1.1: Dmitry Pavlovsky, Alexey Pajitnov, and Tetris ……………………………………2

Figure 1.2: a standard Tetris grid ………………………………………………………………….3

Figure 1.3: All seven Tetriminos with their orientations ……………………………………….4

Figure 1.4: a Tetris player exploring all possible actions for the current piece ………………..6

Figure 2.1: a two point crossover between two chromosomes ……………………………..…13

Figure 2.2: mutation applied to gene in a genotype (chromosome) …………………………..16

Figure 3.1: Flowchart of the implemented Tetris player ………………………………………20

Figure 3.2: Flowchart of the genetic algorithm implemented …………………………………22

Figure 3.3: Plot of the scores of the population through the generations with exponential evaluation function ……………………………………………………………………………23

Figure 3.4: Plot of the scores of the population through the generations with linear evaluation function …………………………………………………………………………………..24

Figure 3.5: Snapshot of a run of the best linear player ……………………..……...………25

LIST OF TABLES

Table 3.1: comparison between the best players of the linear and exponential implementations ……………………………………………………………………………………………26

General introduction

Artificial intelligence (AI) is the intelligence exhibited by machines. In computer science, an ideal "intelligent" machine is a flexible rational agent that perceives its environment and takes actions that maximize its chance of success at an arbitrary goal.

Game playing has been a major topic of AI since the very beginning. Beside the attraction of the topic to people, it is also because of its close relation to "intelligence", and its well-defined states and rules.

There are perfect information games (such as Chess and Go) and imperfect information games (such as Bridge and games where dice are used). Given sufficient time and space, usually an optimum solution can be obtained for the former by exhaustive search, though not for the latter. However, for most interesting games, such a solution is usually too inefficient to be practically used. [1]

Of these imperfect information games, the interest of this project is to give a solution to Tetris. Tetris doesn't have an optimum solution [2]; however this project attempts to get a statically decent solution through training a Tetris player using genetic algorithm, which is a heuristic method.

This project is divided into three chapter described as follows:

Chapter one: this chapter presents an overview of the problem of playing Tetris.

Chapter two: this chapter presents the theory behind genetic algorithm, its different constituent parts, and how it works.

Chapter three: this chapter shows the design and implementation of the Tetris player and the genetic algorithm. Then it presents the results and discusses them.

The final part of this project is the conclusion, where the results of the implementation are judged.

CHAPTER ONE:

Introduction to the problem of playing Tetris

1.1. Introduction

This chapter's goal is to introduce the game of Tetris and its standards, and to explain how this game is played and show why Tetris is good for testing AI. Then it will present the relation of games and game-playing to the field of artificial intelligence and present a principle used in game-playing.

1.2.Tetris

1.2.1. Background

In 1985, Alexey Pajitnov and Dmitry Pavlovsky were computer engineers at the Computing Center of the Russian Academy of Sciences. Alexey and Dmitry were interested in developing and selling addictive computer games. They tested out several different games.

Alexey was inspired by the ancient Greek puzzle game, Pentaminos, which involved arranging puzzle pieces made of five squares. He thought of the idea of arranging Pentamino pieces as they fell in to a rectangular cup, but realized that the twelve different five-square shapes were too complex for a video game.Alexey switched to using seven "tetramino" pieces, each made of four squares.

In 1985.6, Alexey Pajitnov programmed the first version of Tetris on an Electronica 60.

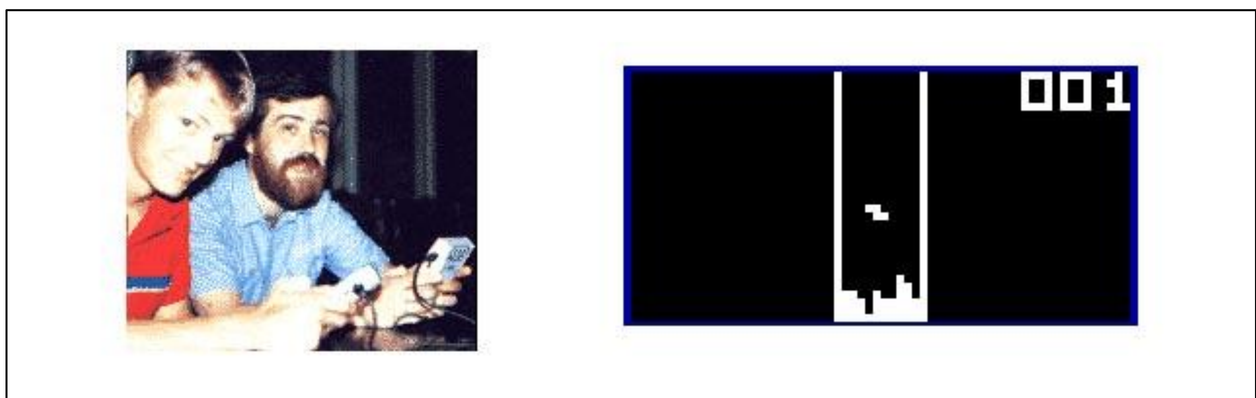

Figure 1.1: Dmitry Pavlovsky, Alexey Pajitnov, and Tetris. [3]

1.2.2. Standards and Gameplay:

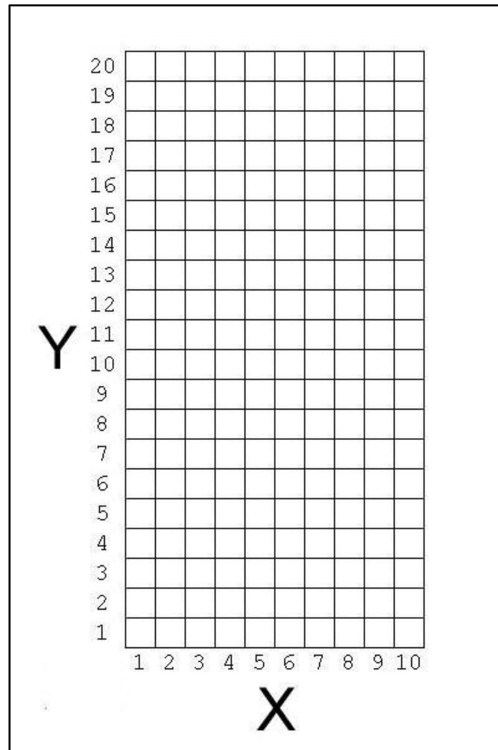

Figure 1.2: a standard Tetris grid

The main constituents of this game are the grid and the Tetriminos. As shown in figure 1.2, the grid is 10x20 a total of 200 cells. "Tetriminos" are game pieces shaped like Tetraminos, geometric shapes composed of four square blocks each. Random sequences of Tetriminos fall down the playing field (a rectangular vertical shaft, called the "well" or "matrix"). There are seven (7) standard Tetris pieces, with the following letter names: **{O, I, S, Z, L, J, T}**

The letter names are inspired by the shapes of the pieces.

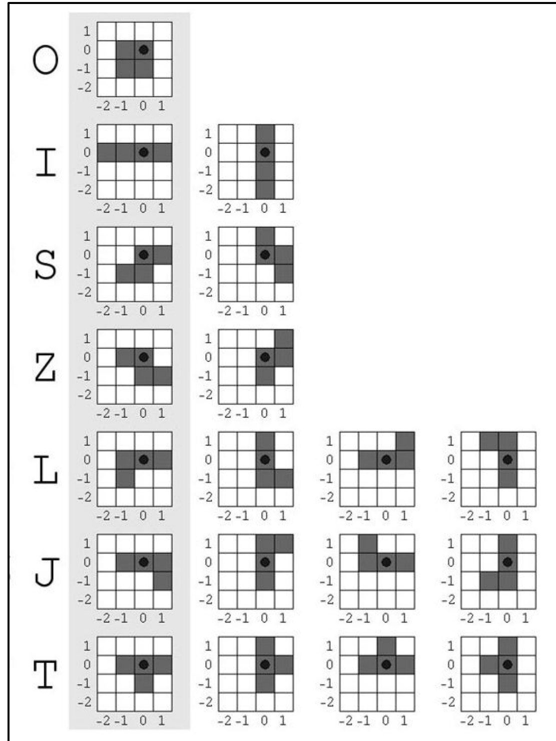

Figure 1.3: All seven Tetriminos with their orientations

The "**O**" (box) piece only has a single orientation, and does not change the locations of any of its occupied (full) cells in response to any counterclockwise rotation event.

The "**I**" (line) piece has two possible orientations, initially appearing in a horizontal orientation. The "**I**" piece alternates between the two orientations in response to successive counterclockwise rotation events.
The "**S**" and "**Z**" pieces each have two possible orientations.
These pieces each alternate between two orientations in response to successive counterclockwise rotation events.
The "**L**", "**J**", and "**T**" pieces each have four possible orientations, and these orientations are the results of simple rotations about center points on the shapes.

The objective of the game is to manipulate these Tetriminos, by moving each one sideways (if the player feels the need) and rotating it by 90 degree units, with the aim of creating a horizontal line of ten units without gaps. When such a line is created, it disappears, and any block above the deleted line will fall. When a certain number of lines are cleared, the game enters a new level. As

the game progresses, each level causes the Tetriminos to fall faster, and the game ends when the stack of Tetriminos reaches the top of the playing field and no new Tetriminos are able to enter. Some games also end after a finite number of levels or lines.

All of the Tetriminos are capable of single and double clears. *I*, *J*, and *L* are able to clear triples. Only the *I* Tetrimino has the capacity to clear four lines simultaneously, and this is referred to as a "Tetris".

Players lose a typical game of *Tetris* when they can no longer keep up with the increasing speed, and the Tetriminos stack up to the top of the playing field. This is commonly referred to as "*topping out*." there are implementations that have constant speed all through the game where the player loses because of the probability of making a mistake will build up over time until it becomes inevitable.

1.2.3. Why Tetris?

Tetris is a good example problem to try to solve because of the following reasons:

1. Has well defined rules
2. Treats the common problem of putting things into order out of a disorder.
3. Has a factor of randomness which simulates a big range of problems.
4. It is relatively easy to implement. And as a result gives an insight on how to solve more complex problems.

1.3. Game playing using AI

Games are represented in a wide spectrum; ranging from strategy games to role playing games from action games to puzzle games. These wide varieties of games are conceived to challenges the intelligence and reflexes of the human player. The AI field has been highly

interested in game playing since they provide a benchmark to compare artificial intelligence against human intelligence.

One of the popular solutions to playing games is to look at the possible future states of the game depending on the allowed actions of the player, and evaluate these states using an evaluation function to find out which is the best state. By doing that the player is capable to choose the actions that would lead to that state. Figure 1.4 shows a tetris player generating future states to analyze them.

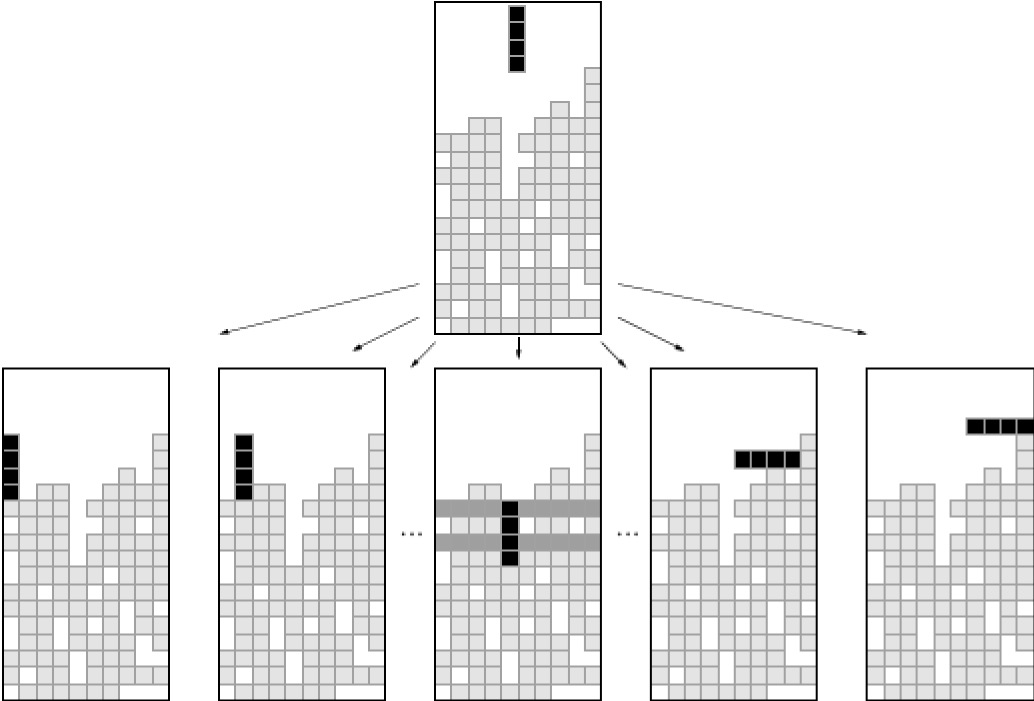

Figure 1.4: a Tetris player exploring all possible actions for the current piece [4]

1.4. Conclusion

This chapter introduced the game of Tetris and its standards, explained how this game is played, and showed why Tetris is a good for testing AI. Then it presented the relation of games and game-playing to the field of artificial intelligence and present a principle used in game-playing.

CHAPTER TWO:

Genetic algorithm

2.1.Introduction

Genetic algorithms are a very general optimization algorithm modeled on the process of natural selection. It is one of several evolutionary algorithms, and differs from its closest variant in that it incorporates the idea of sexual reproduction, or “genetic recombination”.

2.2.Background

In 1859 Charles Darwin wrote Origin of Species and changed the worlds of science and philosophy. And about a hundred years later, in 1954, Nils Aall Barricelli was virtually the first to emulate evolution on a computer. A few years later, in the 1960s, Ingo Rechenberg latched on to the idea of genetic algorithms and realizing that it was much wildly generalizable and that is was not restricted to biology. And hence genetic algorithms were popularized and as a tool for optimization. A great part of the advancement of genetic algorithms is attributed to John Henry Holland and his work in the 60s and 70s which is heavily referenced here in this work.

2.3.A simple genetic algorithm

Given a clearly defined problem to be solved and a bit string representation for candidate solutions, a simple GA works as follows:

1. Start with a randomly generated population of n l-variable chromosomes (candidate solutions to a problem).
2. Calculate the fitness $f(x)$ of each chromosome x in the population.
3. Repeat the following steps until n offspring have been created:
    a. Select a pair of parent chromosomes from the current population, the probability of selection being an increasing function of fitness. Selection is done "with replacement," meaning that the same chromosome can be selected more than once to become a parent.
    b. With probability pc (the "crossover probability" or "crossover rate"), cross over the pair at a randomly chosen point (chosen with uniform probability) to form two offspring. If no crossover takes place, form two offspring that are exact copies of their respective parents. (Note that here the crossover rate is defined to be the

probability that two parents will cross over in a single point. There are also "multi−point crossover" versions of the GA in which the crossover rate for a pair of parents is the number of points at which a crossover takes place.)

c. Mutate the two offspring at each locus with probability pm (the mutation probability or mutation rate), and place the resulting chromosomes in the new population. If n is odd, one new population member can be discarded at random.

4. Replace the current population with the new population.
5. If reached convergence, stop. Else go to step 2. [6]

2.4.How do genetic algorithms work?

Although genetic algorithms are simple to describe and program, their behavior can be complicated, and many open questions exist about how they work and for what types of problems they are best suited. For much work has been done on the theoretical foundations of GAs.

The traditional theory of GAs (first formulated in Holland 1975) assumes that, at a very general level of description, GAs work by discovering, emphasizing, and recombining good "building blocks"(also called “schemas”) of solutions in a highly parallel fashion. The idea here is that good solutions tend to be made up of good building blocks—combinations of bit values that confer higher fitness on the strings in which they are present.

2.5.GA Operators

The simplest form of genetic algorithms involves three types of operators: selection, crossover (single point), and mutation.

1. **Selection** This operator selects chromosomes in the population for reproduction. The fitter the chromosome, the more times it is likely to be selected to reproduce.
2. **Crossover** This operator randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two offspring. For example, the strings 10000100 and 11111111 could be crossed over after the third locus in each to produce the two offspring 10011111 and 11100100. The crossover operator roughly mimics biological recombination between two single−chromosome (haploid) organisms.

3. **Mutation** This operator randomly flips some of the bits in a chromosome. For example, the string 00000100 might be mutated in its second position to yield 01000100. Mutation can occur at each bit position in a string with some probability, usually very small (e.g., 0.001). [5]

2.5.1. SELECTION

After deciding on an encoding (i.e. how to represent the chromosomes), the second decision to make in using a genetic algorithm is how to perform selection—that is, how to choose the individuals in the population that will create offspring for the next generation and how many offspring each will create. The purpose of selection is, of course, to emphasize the fitter individuals in the population in hopes that their offspring will in turn have even higher fitness. Selection has to be balanced with variation from crossover and mutation (the "exploitation/exploration balance"): too−strong selection means that suboptimal highly fit individuals will take over the population, reducing the diversity needed for further change and progress; too−weak selection will result in too−slow evolution. As was the case for encodings, numerous selection schemes have been proposed in the GA literature. Below is described some of the most common methods. As was the case for encodings, these descriptions do not provide rigorous guidelines for which method should be used for which problem; this is still an open question for GAs.

2.5.1.1.Roulette Wheel Selection

Holland's original GA used fitness−proportionate selection, in which the "expected value" of an individual (i.e., the expected number of times an individual will be selected to reproduce) is that individual's fitness divided by the average fitness of the population. The most common method for implementing this is "roulette wheel" sampling: each individual is assigned a slice of a circular "roulette wheel," the size of the slice being proportional to the individual's fitness. The wheel is spun N times, where N is the number of individuals in the population. On each spin, the individual under the wheel's marker is selected to be in the pool of parents for the next generation. This method can be implemented as follows:

1. Sum the total expected value of individuals in the population. Call this sum T.
2. Repeat N times:

Choose a random integer r between 0 and T.

Loop through the individuals in the population, summing the expected values, until the sum is greater than or equal to r. The individual whose expected value puts the sum over this limit is the one selected.

This stochastic method statistically results in the expected number of offspring for each individual. However, with the relatively small populations typically used in GAs, the actual number of offspring allocated to an individual is often far from its expected value (an extremely unlikely series of spins of the roulette wheel could even allocate all offspring to the worst individual in the population). James Baker [7] proposed a different sampling method—"stochastic universal sampling" (SUS)—to minimize this "spread" (the range of possible actual values, given an expected value). Rather than spin the roulette wheel N times to select N parents, SUS spins the wheel once—but with N equally spaced pointers, which are used to select the N parents.

SUS does not solve the major problems with fitness−proportionate selection. Typically, early in the search the fitness variance in the population is high and a small number of individuals are much fitter than the others.

Under fitness−proportionate selection, they and their descendents will multiply quickly in the population, in effect preventing the GA from doing any further exploration. This is known as "premature convergence." In other words, fitness−proportionate selection early on often puts too much emphasis on "exploitation" of highly fit strings at the expense of exploration of other regions of the search space. Later in the search, when all individuals in the population are very similar (the fitness variance is low), there are no real fitness differences for selection to exploit, and evolution grinds to a near halt. Thus, the rate of evolution depends on the variance of fitnesses in the population. [5]

2.5.1.2.Elitism

"Elitism," first introduced by Kenneth De Jong [8], is an addition to many selection methods that forces the GA to retain some number of the best individuals at each generation. Such individuals can be lost if they are not selected to reproduce or if they are destroyed by crossover or mutation. Many researchers have found that elitism significantly improves the GA's performance.

2.5.1.3.Rank Selection

Rank selection is an alternative method whose purpose is also to prevent too−quick convergence. In the version proposed by Baker [9], the individuals in the population are ranked according to fitness, and the expected value of each individual depends on its rank rather than on its absolute fitness. There is no need to scale fitnesses in this case, since absolute differences in fitness are obscured. This discarding of absolute fitness information can have advantages (using absolute fitness can lead to convergence problems) and disadvantages (in some cases it might be important to know that one individual is far fitter than its nearest competitor). Ranking avoids giving the far largest share of offspring to a small group of highly fit individuals, and thus reduces the selection pressure when the fitness variance is high. It also keeps up selection pressure when the fitness variance is low: the ratio of expected values of individuals ranked i and i+1 will be the same whether their absolute fitness differences are high or low.

The linear ranking method proposed by Baker is as follows: Each individual in the population is ranked in increasing order of fitness, from 1 to N. The user chooses the expected value Max of the individual with rank N. The expected value of each individual iin the population at time t is given by

$$ExpVal(i,t) = Min + (Max - Min)\frac{rank\,(i,t)-1}{N-1} \;\ldots.\; (1)$$

where Min is the expected value of the individual with rank 1, and Max is the expected value of the individual with rank N.

At each generation the individuals in the population are ranked and assigned expected values according to equation 1. Baker recommended Max = 1.1 and showed that this scheme compared favorably to fitness proportionate selection on some selected test problems. Rank selection has a possible disadvantage: slowing down selection pressure means that the GA will in some cases be slower in finding highly fit individuals. However, in many cases the increased preservation of diversity that results from ranking leads to more successful search than the quick convergence

that can result from tournament selection. A variety of other ranking schemes (such as exponential rather than linear ranking) have also been tried. For any ranking method, once the expected values have assigned, the SUS method can be used to sample the population (i.e., choose parents).

2.5.1.4.Tournament Selection

The fitness−proportionate methods described above require two passes through the population at each generation: one pass to compute the mean fitness (and, for sigma scaling, the standard deviation) and one pass to compute the expected value of each individual. Rank scaling requires sorting the entire population by rank—a potentially time−consuming procedure. Tournament selection is similar to rank selection in terms of selection pressure, but it is computationally more efficient and more amenable to parallel implementation.

Two individuals are chosen at random from the population. A random number r is then chosen between 0 and 1. If $r < k$ (where k is a parameter, for example 0.75), the fitter of the two individuals is selected to be a parent; otherwise the less fit individual is selected. The two are then returned to the original population and can be selected again. An analysis of this method was presented by Goldberg and Deb [10].

2.5.1.5.Steady−State Selection

Most GAs described in the literature have been "generational"—at each generation the new population consists entirely of offspring formed by parents in the previous generation (though some of these offspring may be identical to their parents). In some schemes, such as the elitist schemes described above, successive generations overlap to some degree—some portion of the previous generation is retained in the new population. The fraction of new individuals at each generation has been called the "generation gap" (De Jong [8].) In steady−state selection, only a few individuals are replaced in each generation: usually a small number of the least fit individuals are replaced by offspring resulting from crossover and mutation of the fittest individuals. Steady−state GAs are often used in evolving rule−based systems (e.g., classifier systems; see Holland 1986) in which incremental learning (and remembering what has already been learned) is important and in which members of the population collectively (rather than individually) solve the problem at hand.

2.5.2. Crossover

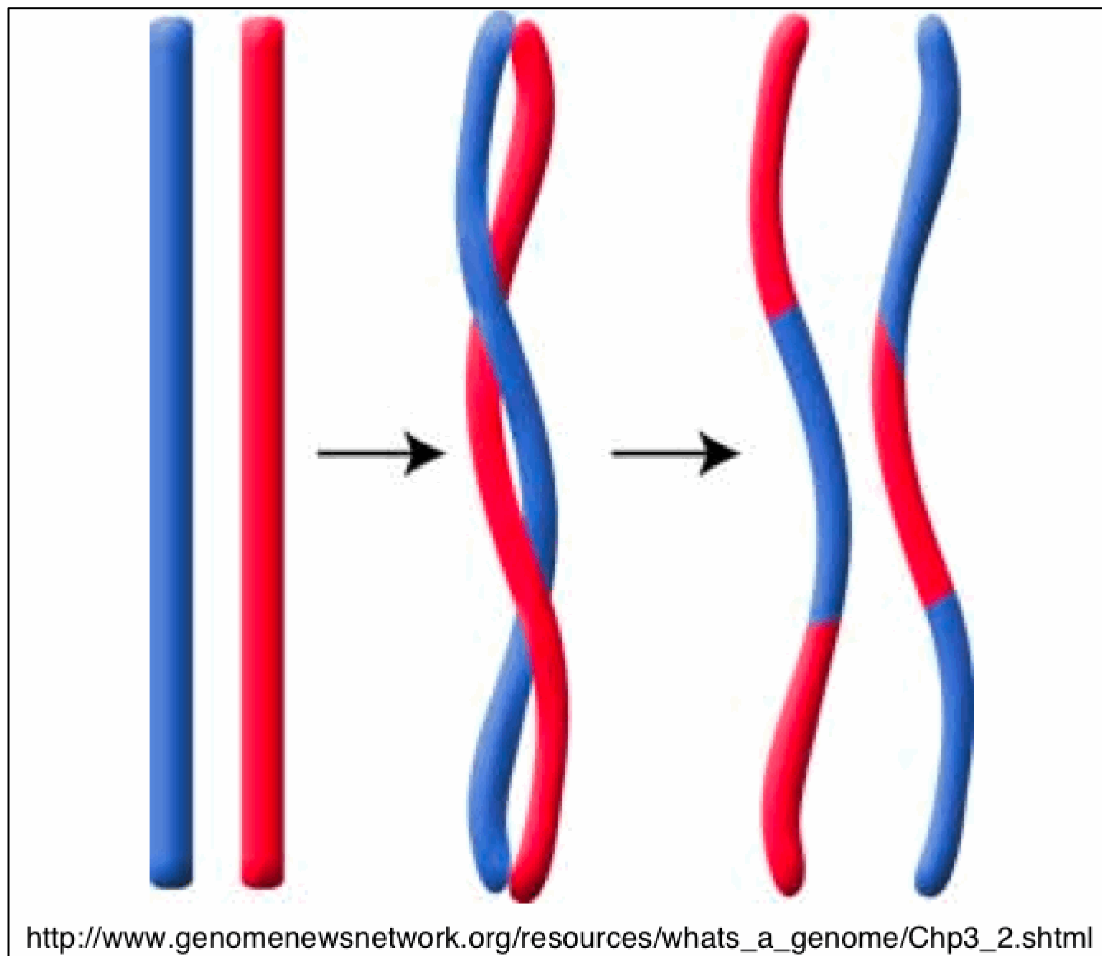

Figure 2.1: a two point crossover between two chromosomes [11]

It could be said that the main distinguishing feature of a GA is the use of crossover. Single−point crossover is the simplest form: a single crossover position is chosen at random and the parts of two parents after the crossover position are exchanged to form two offspring. The idea here is, of course, to recombine building blocks (schemas) on different strings. Single−point crossover has some shortcomings, though. For one thing, it cannot combine all possible schemas. For example, it cannot in general combine instances of 11*****1 and ****11** to form an instance of 11**11*1. Likewise, schemas with long defining lengths are likely to be destroyed under single−point crossover. Eshelman, Caruana, and Schaffer [12] call this "positional bias": the schemas that can be created or destroyed by a crossover depend strongly on the location of the bits in the chromosome. Single−point crossover assumes that short, low−order schemas are the functional building blocks of strings, but one generally does not know in advance what ordering

of bits will group functionally related bits together—this was the purpose of the inversion operator and other adaptive operators described above. Eshelman, Caruana, and Schaffer also point out that there may not be any way to put all functionally related bits close together on a string, since particular bits might be crucial in more than one schema. They point out further that the tendency of single−point crossover to keep short schemas intact can lead to the preservation of hitchhikers—bits that are not part of a desired schema but which, by being close on the string, hitchhike along with the beneficial schema as it reproduces. Many people have also noted that single point crossover treats some loci preferentially: the segments exchanged between the two parents always contain the endpoints of the strings.

To reduce positional bias and this "endpoint" effect, many GA practitioners use two−point crossover, in which two positions are chosen at random and the segments between them are exchanged. Two−point crossover is less likely to disrupt schemas with large defining lengths and can combine more schemas than single−point crossover. In addition, the segments that are exchanged do not necessarily contain the endpoints of the strings.

Again, there are schemas that two−point crossover cannot combine. GA practitioners have experimented with different numbers of crossover points (in one method, the number of crossover points for each pair of parents is chosen from a Poisson distribution whose mean is a function of the length of the chromosome). Some practitioners believe strongly in the superiority of "parameterized uniform crossover," in which an exchange happens at each bit position with probability p.

Parameterized uniform crossover has no positional bias—any schemas contained at different positions in the parents can potentially be recombined in the offspring. However, this lack of positional bias can prevent coadapted alleles from ever forming in the population, since parameterized uniform crossover can be highly disruptive of any schema.

Given these (and the many other variants of crossover found in the GA literature), which one should you use?

There is no simple answer; the success or failure of a particular crossover operator depends in complicated ways on the particular fitness function, encoding, and other details of the GA. It is still a very important open problem to fully understand these interactions. There are many papers in the GA literature quantifying aspects of various crossover operators (positional bias, disruption potential, ability to create different schemas in one step, and so on), but these do not

give definitive guidance on when to use which type of crossover. There are also many papers in which the usefulness of different types of crossover is empirically compared, but all these studies rely on particular small suites of test functions, and different studies produce conflicting results.
Again, it is hard to glean general conclusions. It is common in recent GA applications to use either two−point crossover or parameterized uniform crossover with p.
For the most part, the comments and references above deal with crossover in the context of bit−string encodings, though some of them apply to other types of encodings as well. Some types of encodings require specially defined crossover and mutation operators—for example, the tree encoding used in genetic programming, or encodings for problems like the Traveling Salesman problem (in which the task is to find a correct ordering for a collection of objects).
Most of the comments above also assume that crossover's ability to recombine highly fit schemas is the reason it should be useful. Given some of the challenges we have seen to the relevance of schemas as an analysis tool for understanding GAs, one might ask if we should not consider the possibility that crossover is actually useful for some entirely different reason (e.g., it is in essence a "macro−mutation" operator that simply allows for large jumps in the search space). This question must left as an open area of GA research for interested readers to explore. (Terry Jones [13] has performed some interesting, though preliminary, experiments attempting to tease out the different possible roles of crossover in GAs.) Its answer might also shed light on the question of why recombination is useful for real organisms (if indeed it is)—a controversial and still open question in evolutionary biology.

2.5.3. Mutation

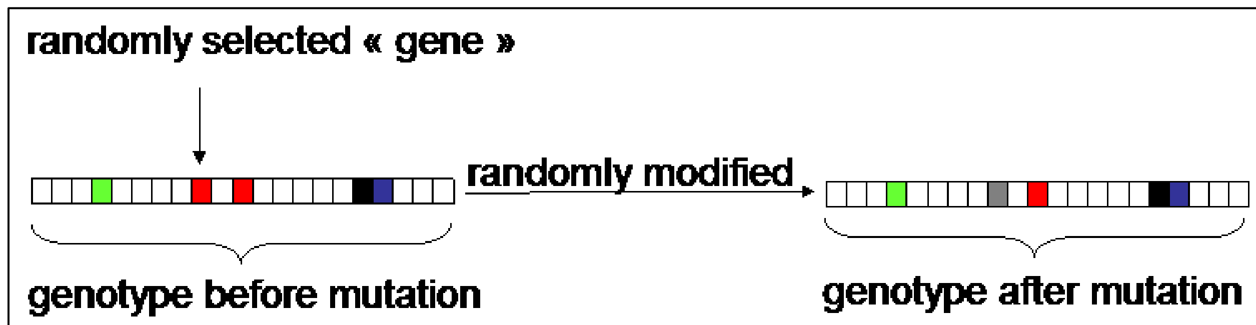

Figure 2.2: mutation applied to gene in a genotype (chromosome) [14]

A common view in the GA community, dating back to Holland's book Adaptation in Natural and Artificial Systems, is that crossover is the major instrument of variation and innovation in GAs, with mutation insuring the population against permanent fixation at any particular locus and thus playing more of a background role.

This differs from the traditional positions of other evolutionary computation methods, such as evolutionary programming and early versions of evolution strategies, in which random mutation is the only source of variation. (Later versions of evolution strategies have included a form of crossover.)

However, the appreciation of the role of mutation is growing as the GA community attempts to understand how GAs solve complex problems. Some comparative studies have been performed on the power of mutation versus crossover; for example, Spears [15] formally verified the intuitive idea that, while mutation and crossover have the same ability for "disruption" of existing schemas, crossover is a more robust "constructor" of new schemas. Mühlenbein [16], on the other hand, argues that in many cases a hill−climbing strategy will work better than a GA with crossover and that "the power of mutation has been underestimated in traditional genetic algorithms." As can be seen in the Royal Road experiments, it is not a choice between crossover and mutation but rather the balance among crossover, mutation, and selection that is all important. The correct balance also depends on details of the fitness function and the encoding. Furthermore, crossover and mutation vary in relative usefulness over the course of a run. Precisely how all this happens still needs to be elucidated. In my opinion, the most promising prospect for producing the right balances over the course of a run is to find ways for the GA to adapt its own mutation and crossover rates during a search. Some attempts at this will be described below.

2.6.Advantages and disadvantages of GA:

2.6.1. Advantages:

- Simple algorithm and easy implementation.
- Derivative-free technique.
- Capable to escape from local minima.
- Does not require a good initialization.
- Flexibility to hybridize with other techniques.

2.6.2. Disadvantages:

- Can take long time to convergence
- No guarantee of finding global maxima.
- It needs manual tuning of all the parameters, like mutation rate.
- Apart from the genetic parameters of the GA, other things like the fitness function, choice of genetic encoding, genotype to phenotype mapping, etc., are also important in the efficacy of the system.

2.7. Conclusion

This chapter discussed the general optimization known as “genetic algorithm”. It described briefly the origin of the genetic algorithms. Then, a basic genetic algorithm’s process flow is described. After that, a theory why genetic algorithm works was put forward. Following that, the components of genetic algorithm are described: selection and its different methods, crossover and how it can be applied, and mutation and its importance in the algorithm. Finally it presented the advantages and disadvantages of GAs.

CHAPTER THREE:

Implementation of the Tetris player

3.1. Introduction

The purpose of this chapter is to design a player that can play Tetris. This player is to be designed with two different evaluation functions (linear and exponential). To train this player an implementation of genetic algorithm is to be designed to maximize its efficiency and to minimize resources (like time) while doing that. The player and the GA are to be implemented with C++ (2011 standards). The results of this implementation should provide a tetris player capable of giving noticeable results.

3.2.Method

3.2.1. Player Design:

Decisions made to the different parts of the player are discussed in the following:

- **Speed of the game:** due to the fact that a computer can find a solution to falling tetraminos in an insignificant amount of time compared to a human player. A game is considered, where the speed is constant and the player can make it in time while the tetramino is still at its birth place.
- **One piece or two pieces:** this player is assumed to know only the current Tetraminos (one piece). This approach is taken given the fact using a two piece implementation can take 34 times (e.g. L Tetrimino has 34 possible placements) more computations and thus it would possibly take 34 times longer to execute a move.
- **Max moves limit:** due to the algorithm slowing down when the player gets better. A cap of 1000 moves is put to the player.
- **The Tetraminos:** although most implementation discussed in chapter 1 implement a Tetrimino as a 4x4 or 5x5 array of the different orientations of each Tetrimino. Each time collision is checked for it takes 16 or 25 checks. The algorithm can be enhanced by using structures of four pairs of coordinates to reduce the number of checks.
- **Birth of Tetraminos:** each Tetrimino in a specific orientation is born in the middle column (6th column in this implementation) of the board and in the row specified in the structure to not cause a collision on birth.
- **Scoring:** Reward each successful placement of a Tetrimino with 1pt, and 10pts for clearing 1 line, 30pts for 2 lines, 60pts for 3 lines, and 100pts for 4 lines. Here, making the player clear multiple lines at the same time is more emphasized.

- **Evaluation function**: Two instances of the evaluation function are tested:
  1. Linear: The value of each feature is multiplied with its corresponding weight (W) then everything is summed together.
  2. Exponential: with each feature value the corresponding displacement (D) is subtracted, then the corresponding exponent (E) is applied, and then we multiply by the weight (W).
- **The features**: These features are used to evaluate the grid resulting from the possible move:
  1. Cleared lines: The number of full horizontal rows on the game board
  2. Maximum Altitude: The height of the tallest column on the game board
  3. Deepest well: The depth of the deepest hole (a width-1 hole with filled spots on both sides)
  4. Roughness: The sum of the difference in height of adjacent columns
  5. Connected Holes: Each empty cell that has at least one filled cell directly above it
  6. Blocks: Sum of all filled cells on the board
  7. Weighted Blocks: Sum of all filled cells on the board, but each cell has the weight of the row it is in
  8. Highest Hole: Highest empty cell that has at least one filled cell above it
  9. Holes: Each empty cell that has at least one filled cell above it
  10. Highest Hole Depth: How much filled spots are above the highest hole
- **Execution flow of the player:** The execution flow is shown in figure 3.1.

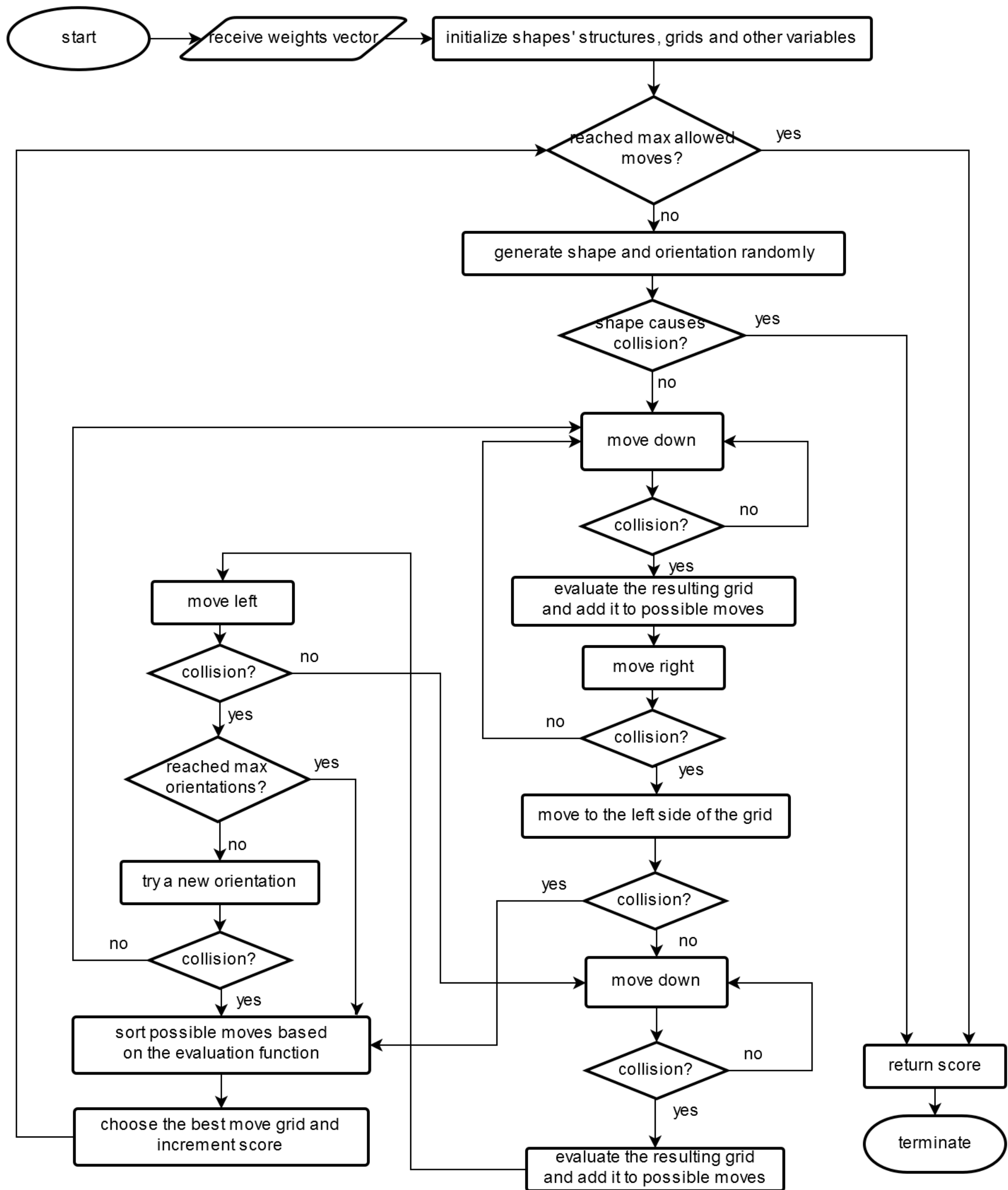

Figure 3.1: Flowchart of the implemented Tetris player

3.2.2. Genetic Algorithm Design

Throughout the stages of the GA some decisions were made. They are explained in the following:

- **Encoding of chromosomes:** three vectors are used to represent the displacement values (D), the exponent values (E) and the proportional weights (W).
- **Initial population:** each individual of the population is initialized randomly. The vector W is initialized between -100 and 100. The vector D is initialized between -20 and 20. The vector E is initialized between 0 and 5(any higher value will cause overflow).
- **Fitness function:** the fitness is based on the ability of the individual to play tetris. The tetris player function is called for each individual of the population and returns the score they achieved.
- **Selection:** steady state is applied as the selection method to enhance performance because it has constant time in selecting one individual. Also it has an easy adjustable parameter of how much to keep unchanged in the population. This parameter is set to 50 in this implementation to increase survival pressure and force the GA to converge rapidly. This parameter is referred to as *n* in figure 3.2.
- **Crossover:** a single point crossover method is used on each of the three vectors of an individual. This point is selected randomly. If we consider all three vectors as one vector this can translate to a crossover of three moving points (moving point of each vector) and two fixed points (the connection points between the vectors).
- **Mutation:** the rate of mutation is kept very low (0.1%), because mutation is mostly adaptive, where through long periods of times where the fitness requirements change. Another reason is that the size of the population is provides enough diversity.
- **The execution flow of the GA:** the execution flow is shown in figure 3.2

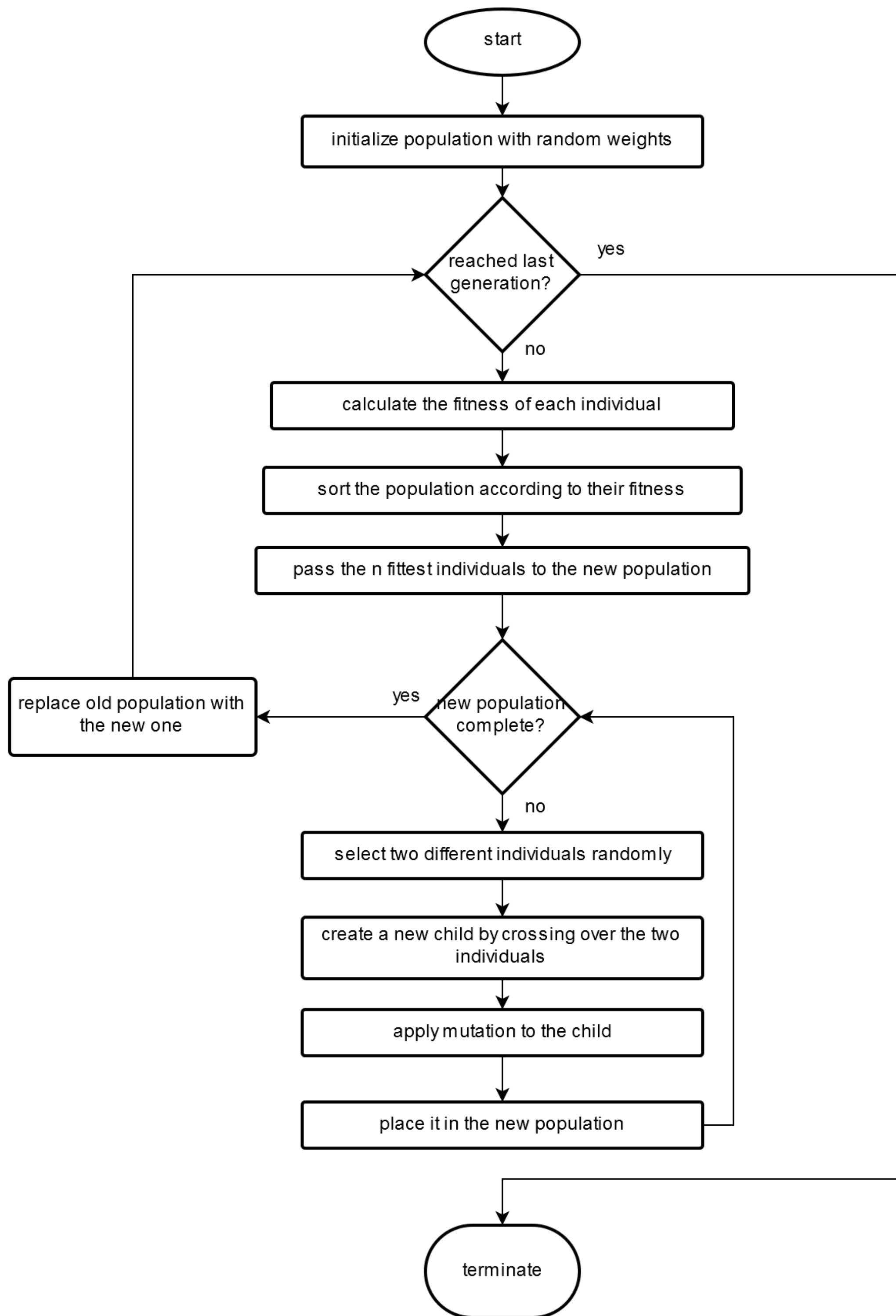

Figure 3.2: Flowchart of the genetic algorithm implemented

3.3. Results

3.3.1. Exponential evaluation function

The genetic algorithm is applied to the player with the exponential evaluation function as shown in figure 3.3.

This run of the genetic algorithm took 19 minutes and 33 seconds (1173seconds).

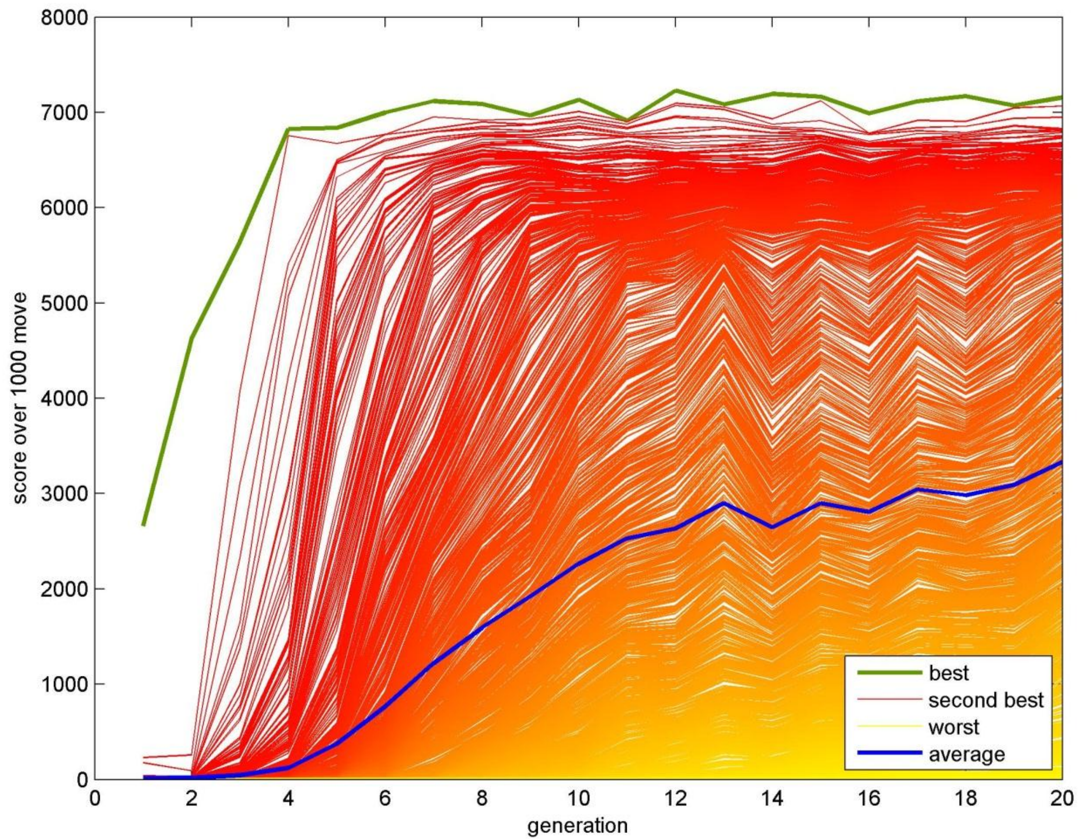

Figure 3.3: Plot of the scores of the population through the generations with exponential evaluation function

3.3.2. Linear evaluation function

The genetic algorithm is applied to the player with the linear evaluation function as shown in figure 3.4.

This run of the genetic algorithm took 23 minutes and 22 seconds (1403seconds).

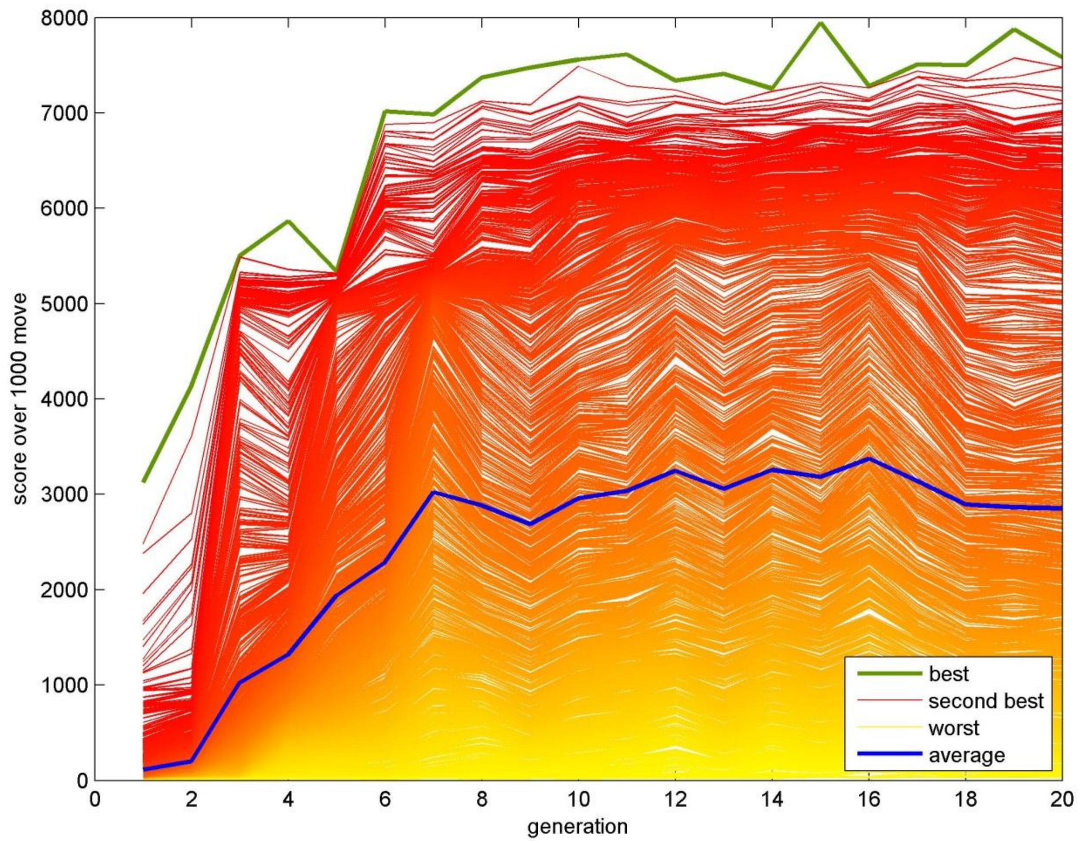

Figure 3.4: Plot of the scores of the population through the generations with linear evaluation function

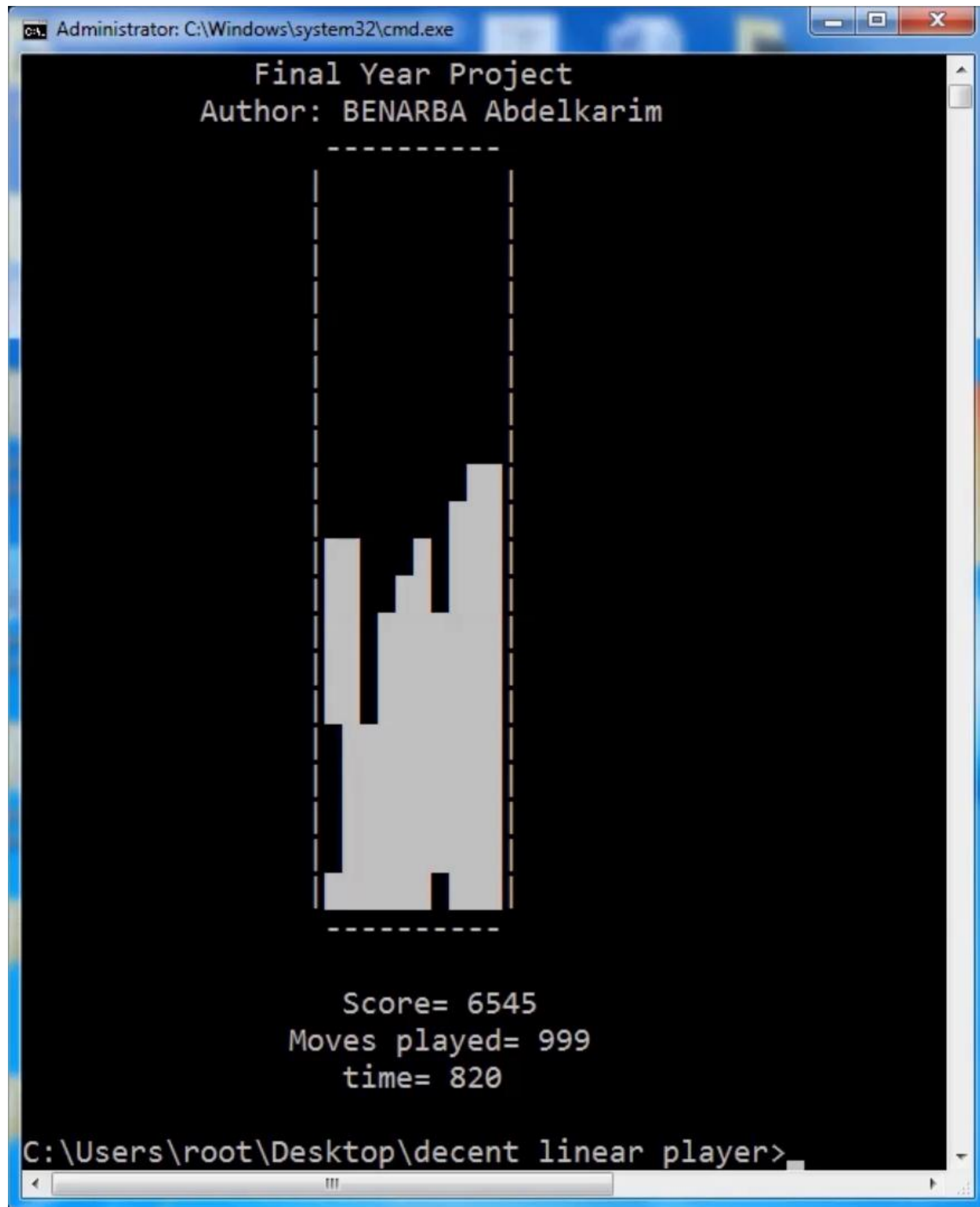

Figure 3.5: Snapshot of a run of the best linear player

3.4. Discussion

Table 3.1: comparison between the best players of the linear and exponential implementations

Evaluation function	Average score	Best score	Average moves	Best moves
Linear	2874	7145	421	1000
Exponential	2707	7532	381	1000

By comparing figure 3.3 and figure 3.4, the two methods seem to give similar results to some extent. But there is a small difference between them: while they both seem limited with the barrier of 1000 moves (the cap over the plot). Figure 3.3 shows a more abrupt cutoff than figure 3.4 showing that it has more potential for growth. That higher potential of growth would keep an extreme survival pressure pushing for more fitness. Another comparison is that the first figure's average plot seems to continue to grow after the 20th generation on the contrary to the second figure where the average plot seems to have converged. As shown in table 3.1, the linear best player has a slight advantage over the exponential player in regards to the average score and moves, while it has a slight disadvantage when it comes to the best score. But, in general, they are very insignificant differences.

3.5. Conclusion

In this chapter, a Tetris player was designed through following Tetris standards and by making some design decisions to improve its performance and to improve its training efficiency. As for training the player, a genetic algorithm was designed taking in consideration two points: the limitation of time and the objective of getting a player with some degree of efficiency. Then these two designs where implemented using C++ language (2011 standards). After that, the results of this implementation were shown using Matlab and then discussed and compared in the following section.

General conclusion

In this project, a solution to the problem of playing Tetris was presented. A Tetris player was implemented and trained with a genetic algorithm. The results of this work show that genetic algorithms are an effective way of training a Tetris player.

The resulting two Tetris players have very close performance; the player with the exponential evaluation function had an average of 381 and a best of 1000 moves, and an average of 2707 and a best of 7532 score wise. While the player with the linear evaluation function had an average of 421 and a best of 1000 moves, and an average of 2874 and a best of 7145 score wise.

From the results, it is evident that the two methods are limited by the cap of 1000 moves. Both the exponential and linear evaluation functions have more potential if the cap is removed. While it's clear that given the conditions of our implementation limit both methods, there is an apparent slight difference in the potential that these methods have beyond these limits, with the exponential implementation having more advantage beyond the cap.

Due to time constrains and the immensity of the subject, lots of aspects could not be treated in this project. In future work, the following points can be addressed:

- Implementing a two-piece player
- Implementing a player that can predict beyond the piece it sees
- Improving the player time performance wise
- Improving the genetic algorithm by finding better parameters for its operators
- Using other methods such as Neural Networks to train the Tetris player

REFERENCES

[1] Pei Wang, "CIS 3203. Introduction to Artificial Intelligence", Temple University, 2014.

[2] E. D. Demaine, S. Hohenberger, and D. Liben-Nowell. Tetris is hard, even to approximate. 9th COCOON, 2003.

[3] Colin P. Fahey, Tetris AI, 2003, URL http://www.colinfahey.com/

[4] Amine Boumaza. How to design good Tetris players, HAL, 2013.

[5] Randy L. Haupt and Sue Ellen Haupt, PRACTICAL GENETIC ALGORITHMS, Hoboken, New Jersey: John Wiley & Sons, Inc., 2004.

[6] Melanie Mitchell, AN INTRODUCTION TO GENETIC ALGORITHS, MIT Press, 1999.

[7] JE Baker, Reducing bias and inefficiency in the selection algorithm, Vanderbilt University, 1987.

[8] KA De Jong, Analysis of the behavior of a class of genetic adaptive systems, University of Michigan, 1975.

[9] JE Baker, Adaptive selection methods for genetic algorithms, Vanderbilt university, 1985.

[10] DE Goldberg, K Deb, A comparative analysis of selection schemes used in genetic algorithms, Foundations of genetic algorithms, 1991

[11] Jessen Havill, introduction to computational problem solving, Denison University, 2013.

[12] LJ Eshelman, RA Caruana, JD Schaffer, Biases in the crossover landscape, Proceedings of the third international conference on Genetic algorithms, 1989.

[13] Terry Jones, Crossover, Macromutation, and Population-based Search, Santa Fe Institute, 1995.

[14] Gerard YAHIAOUI and Pierre DA SILVA DIAS, "Genetic algorithms: tutorial", NEXYAD.

[15] WM Spears, An Overview of Evolutionary Computation, Springer, 1993.

[16] H Mühlenbein, How Genetic Algorithms Really Work: Mutation and Hillclimbing, PPSN, 1992.

[17] H. Burgiel. How to lose at Tetris. Mathematical Gazette, 1997.

[18] Max Bergmark, Tetris: A Heuristic Study, Royal Institute of Technology Stockholm, Sweden, 2015.

[19] Jason Lewis, Playing Tetris with Genetic Algorithms, Stanford, 2015 .

[20] J. Brzustowski, Can you win at Tetris? Department of Mathematics, University of British Columbia, 1992.

[21] Flom, L., Robinson, C.Using a Genetic Algorithm to Weight an Evaluation Function for Tetris. Colorado State University. 2004.