UNIVERSIDADE DO ALGARVE

Faculdade de Ciências e Tecnologia

Visualization of Genetic Algorithm Operation on Additive Decomposable Functions

Pedro Filipe Pereira Miquelina

Dissertação

Mestrado em Engenharia Informática

Trabalho efetuado sob a orientação de:

Prof. Dr. Fernando Lobo

2013

UNIVERSIDADE DO ALGARVE

Faculdade de Ciências e Tecnologia

Visualization of Genetic Algorithm Operation on Additive Decomposable Functions

Pedro Filipe Pereira Miquelina

Dissertação

Mestrado em Engenharia Informática

Trabalho efetuado sob a orientação de:

Prof. Dr. Fernando Lobo

2013

Visualization of Genetic Algorithm Operation on Additive Decomposable Functions

Declaração de autoria de trabalho

Declaro ser o autor deste trabalho, que é original e inédito. Autores e trabalhos consultados estão devidamente citados no texto e constam da listagem de referências incluída.

_____

Copyright

Abstract

In the past few years, there's been a growing interest in understanding genetic algorithm behavior and problem landscapes through visualization methods. This thesis tries to give an introduction to genetic algorithms and their theory, and describes a tool that attempts to better illustrate some of the important theoretical aspects behind a genetic algorithm, operating on additive decomposable functions, through the use of black and white images, in the form of an animation. Since these theoretical aspects can be hard to understand at first, this tool can be helpful for teaching purposes since it can provide different visualizations of a genetic algorithm working and provide real examples with images that can support some of the theory behind a genetic algorithm.

Keywords:

Genetic algorithms, building blocks, additive decomposable functions, onemax function, trap function, visualization.

Resumo

Inspirados pela seleção natural e pela genética, os algoritmos genéticos são métodos de otimização estocásticos utilizados para resolver problemas de otimização, tentando imitar a forma como a Natureza age.

Na teoria de seleção natural, os indivíduos mais fortes de uma população sobrevivem e propagam as suas características genéticas para as gerações seguintes enquanto os mais fracos tendem a desaparecer.

Na Natureza, uma população tem de se adaptar ao meio em que se encontra, mudando algumas características particulares ao longo de gerações através do uso da recombinação e da mutação.

Ao trabalhar com algoritmos genéticos, um problema pode ser visto como um meio onde uma população de soluções se encontra e evolui ao longo de gerações, através do uso de uma combinação de seleção, recombinação e mutação para solucionar o problema.

Podemos dizer que os algoritmos genéticos funcionam com uma população, isto é, um conjunto de soluções candidatas, que são normalmente inicializadas aleatoriamente, e que tem entre outros, dois importantes componentes, a seleção e a variação (recombinação e mutação).

Para além da seleção e da variação, os algoritmos genéticos possuem outros componentes, tais como, a representação, a inicialização, a função de aptidão, a substituição da população, entre outros. Estes componentes, individualmente são fáceis de perceber, mas quando combinados, a teoria por detrás do funcionamento de um algoritmo genético é mais difícil de compreender.

A teoria por detrás dos algoritmos genéticos baseia-se segundo John Henry Holland, no uso de esquemas. A teoria de esquemas de Holland, identifica as soluções como sendo parte de um conjunto de diversos padrões. Estes padrões, quanto menores e mais abrangentes, que possuam uma boa aptidão, são denominados de blocos construtores.

Os algoritmos genéticos processam blocos construtores que são a chave para encontrar uma boa solução. É necessário que os blocos construtores sejam bem misturados e que

cresçam em número. Para isto é essencial fornecer uma boa quantidade de blocos construtores na população inicial e o algoritmo genético deve saber escolher entre dois blocos construtores rivais.

Os problemas unimodais (possuem só um máximo local), são fáceis de resolver para os algoritmos genéticos porque é fácil decidir entre blocos construtores, enquanto para problemas multimodais é mais complicado decidir entre blocos construtores devido à existência de pelo menos um máximo global e um máximo local. Neste tipo de problemas, onde se englobam as funções enganosas utilizadas nesta tese, quanto mais afastado o máximo global estiver do máximo local e quanto menor for a diferença entre eles, mais difícil se torna para os algoritmos genéticos resolverem o problema.

Nos últimos anos tem existido um aumento na utilização de métodos visuais para tentar perceber melhor como funcionam os algoritmos genéticos e a teoria por detrás deles.

Esta tese tenta fazer uma introdução aos algoritmos genéticos e à sua teoria, e descreve uma ferramenta que tenta ilustrar alguns dos aspetos importantes da teoria de algoritmos genéticos aplicados a funções de decomposição, com recurso a imagens a preto e branco em forma de animações. Tendo em conta que estes aspetos teóricos podem ser difíceis de compreender ao início, esta ferramenta pode ser uma boa ajuda para fins de ensino, uma vez que pode fornecer diferentes visualizações de algoritmos genéticos a trabalhar e fornecer exemplos reais com imagens para apoiar a teoria por detrás de um algoritmo genético.

Palavras-Chave:

Algoritmos genéticos, blocos construtores, funções de decomposição aditivas, funções enganosas, visualização.

Acknowledgements

Firstly, I would like to thank my thesis supervisor, Prof. Dr. Fernando Lobo, for the talks we had, for listening, for giving me guidance and in troubled times for keeping me inspired to conclude this work.

I would like to show my gratitude to my parents for everything they did, do and always will for me, my brother João and our families.

I would like to thank the two lights of my life, my wife Andreia and my baby daughter Inês, for the hours they gave me to work on my Masters degree and also apologize for the time I didn't spent with you when working on my Masters degree. Without you I'm lost, I've no guide.

I also would like to thank everyone else that somehow helped me get through these difficult times, whether was just listening to me or just for being part of my life.

I would like to give an honor mention to my mother in law, Fernanda, for the hours of work you gave me.

Contents

Figure index

List of Acronyms

GA	Genetic Algorithm
BB	Building Block
B&W	Black and White
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
JSON	Javascript Object Notation
Ajax	Asynchronous JavaScript and XML

1. Introduction

During the creation of a small project for the Evolutionary Computation course, based on the visualization of randomly generated black and white images evolving to preloaded images, I was approached by the course teacher with the idea that it was possible to "trick" the Genetic Algorithm (GA) into going to the opposite result (black would be white and white would be black). After the theory studied in class and some debates on GAs, it was suggested that it would be a good asset to have more visualizations of GAs working, to better comprehend some important aspects behind the theory. With this need, this work was proposed.

In this thesis we'll start by introducing GAs and its components. Then we'll see some basic GA theory on how they work, what is schema theory, what are Building Blocks and we'll see two additive decomposable functions (onemax and trap functions). Afterwards we'll start focusing on the tool, referencing the technologies used and we'll see how the tool works.

2. Genetic Algorithms and their components

Inspired by natural selection and genetics, GAs are stochastic optimization methods that try to solve difficult optimization problems by mimicking Nature's processes.

In Darwin's theory survival of the fittest or natural selection [12], the stronger individuals of a population survive and propagate their characteristics to the next generations while the weaker tend to disappear.

In Nature, a population is to constantly adapt to the surrounding environment by changing particular characteristics (properties) over consecutive generations through the use of recombination and mutation.

In GAs, a problem can be viewed as an environment, where a population of solutions evolves over generations using the combination of selection, recombination and mutation to solve it.

So, a GA works with a population, a set of candidate solutions (individuals), that are usually randomly initialized and has, among others, two essential components: Selection and Variation (recombination and mutation).

2.1 Representation

Usually, the first thing to do when working with GAs, after identifying the problem, is to define a representation for the given problem [13]. Typically, a binary representation is used, but it can also be integers, real numbers, permutations or other complex types (trees, lists, arrays, among others). An example of a basic binary representation can be seen in fig. 1.

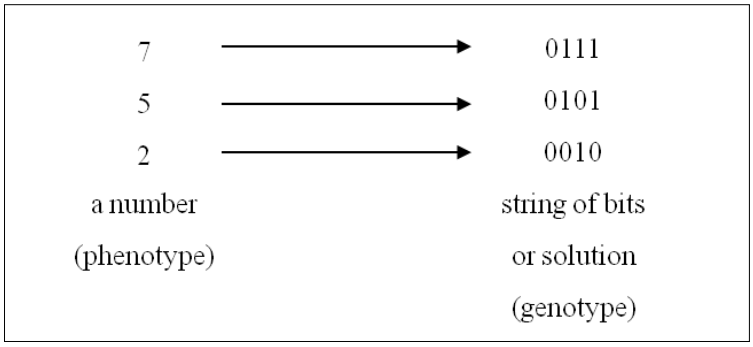


Fig 2.1 - Basic Binary Representation

2.2 Fitness Function

The fitness function is a way of quantifying the quality/value (fitness) of each given solution. This fitness is the quantitative information the algorithm uses to guide the search. Fig. 2 shows the previous basic binary representation with a fitness value given by the function $f(x)=x^2$.

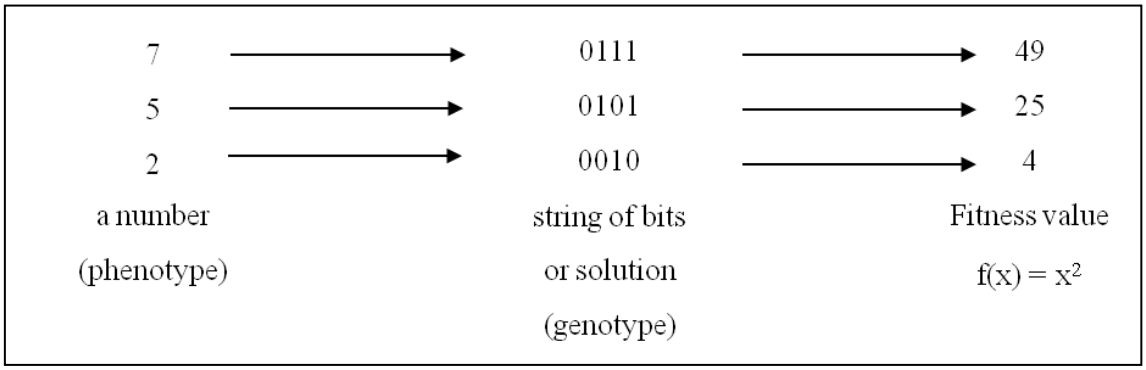


Fig. 2.2 - Basic Binary Representation with Fitness value

2.3 Initialization

The GA population is usually randomly initialized according to a uniform distribution, but can sometimes, according to specific problems, be initialized to previously known good solutions. In this work, we use a random initialization of the population.

For example, fig. 3 gives us eight bit solutions, where each bit has 50% chance of being 1 and 50% chance of being 0.

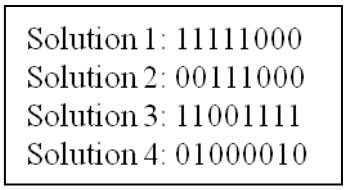


Fig. 2.3 - Four eight bit solutions randomly initialized

2.4 Selection

Selection works by simulating the survival of fittest, where stronger individuals have a better chance to survive, while the weaker tend to disappear. There are several selection methods, but they all fall under one of two categories: ordinal and proportionate. In ordinal selection, an individual has a chance to survive according directly to his fitness when compared to other individuals fitness in relative terms, while in proportionate selection, the individuals are selected according to the same probability distribution with a probability based on a proportion calculated by its fitness compared with the fitness of all other individuals. The main goal of selection is to make copies of the best solutions at the expanse of the worse ones.

In this work, we use tournament selection of size *s*, that chooses *s* random solutions from the population, then from that subset of solutions, selects the best one. This procedure is repeated until the selected solutions are the same number of individuals of the population.

Imagine that the previous four randomly generated solutions in Fig. 3 have their respective fitness's as shown in fig.4.

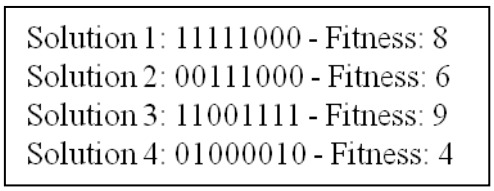


Fig. 2.4 - Four eight bit solutions with fitness

If we had a tournament of size 2, we could have the following random tournaments as shown in fig. 5.

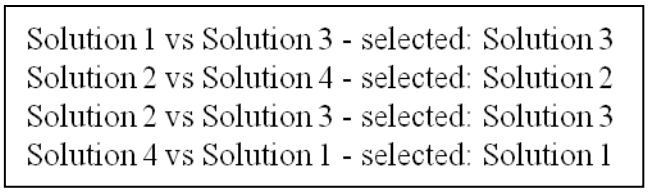


Fig. 2.5 - Selection procedure

In Fig. 6 we can see our selected individuals. We can notice that the best individual (solution with the highest fitness) in fig. 4, now has two copies, and the worst individual (solution with the lowest fitness) is gone.

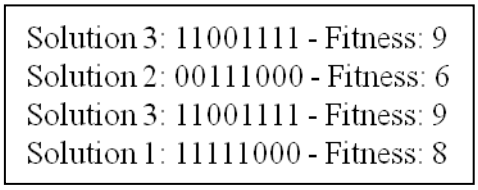


Fig. 2.6 - Tournament size 2 selected solutions

2.5 Variation - Recombination (Crossover)

Recombination or Crossover, consists on an exchange of properties between two or more individuals of a population to create new individuals. This operation is done probabilistically with a given probability P_c.

In this work, several types of crossover operators between two individuals are used, like one-point, two-point, k-point and uniform crossover. For example, one-point crossover selects one random position and exchanges the properties of the next positions as we can see in fig. 7.

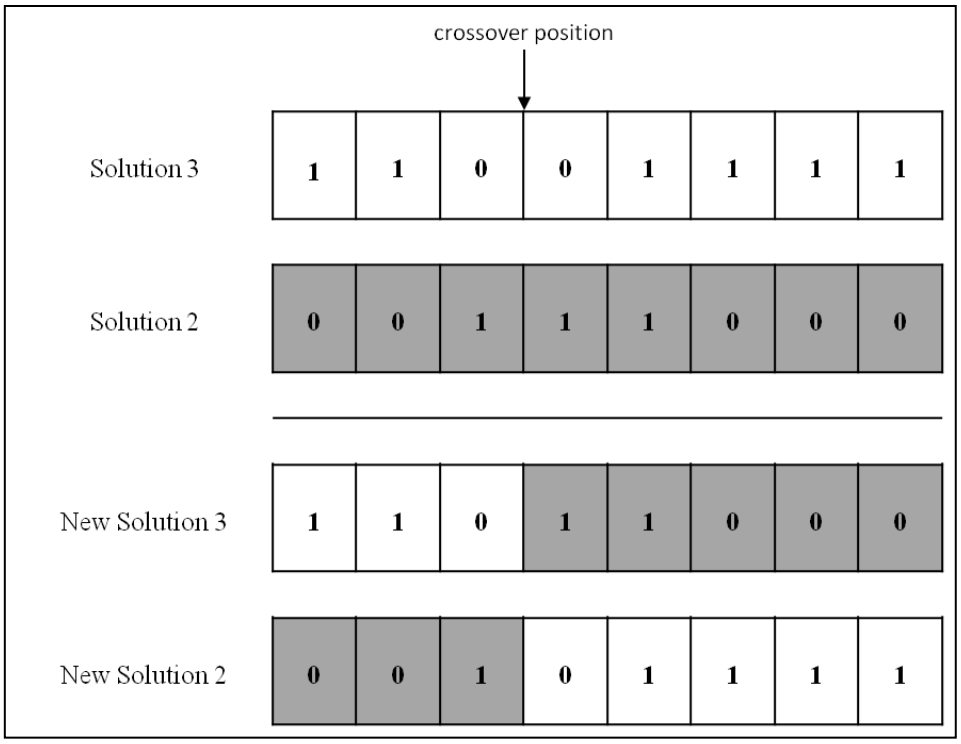


Fig. 2.7 - One-Point Crossover Example

In two-point crossover, 2 different random positions are chosen and the solutions exchange only the attributes between these 2 points as we can see in Fig. 8.

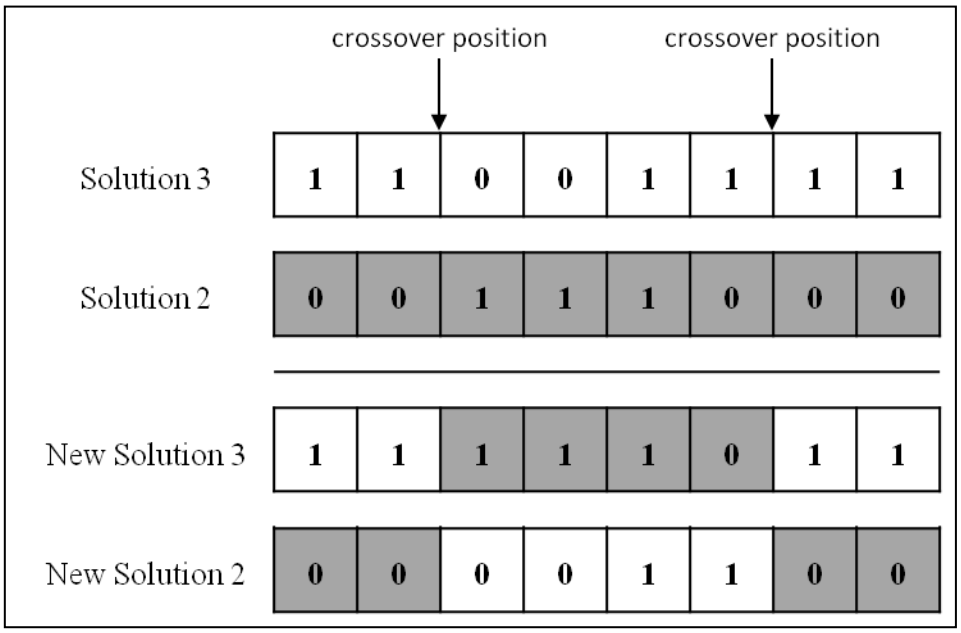


Fig. 2.8 - Two-Point Crossover Example

In k-point crossover, k different positions are chosen, and the solutions exchange attributes between each 2 points as we can see in fig. 9.

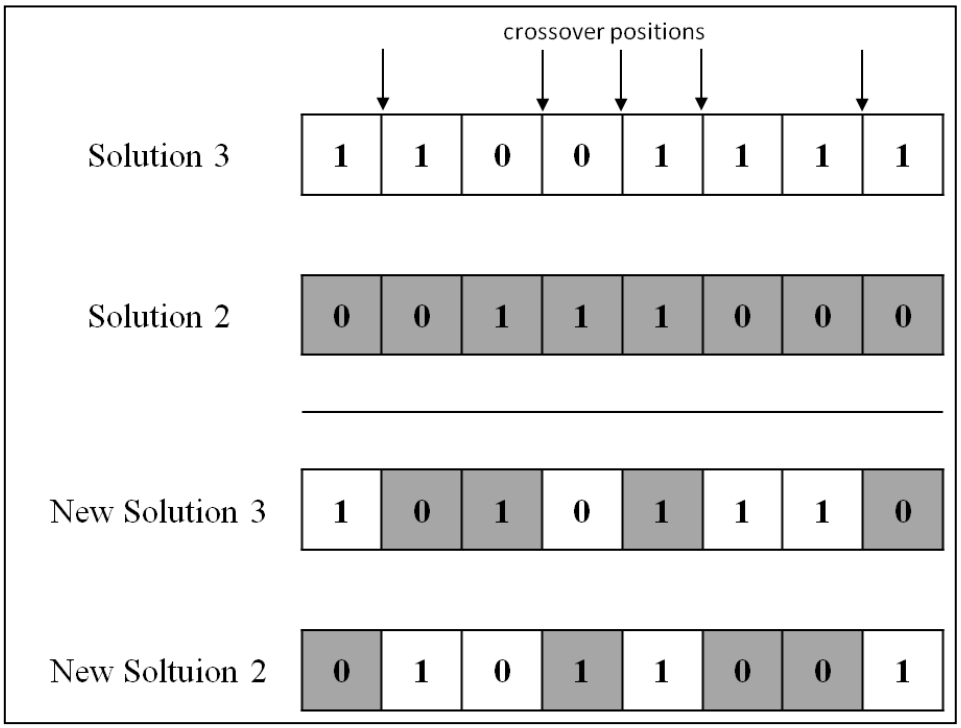


Fig. 2.9 - k-point crossover, k = 5

In uniform crossover, each bit of the solutions, has 50% probability of exchanging. An example of uniform crossover can be seen in fig. 10.

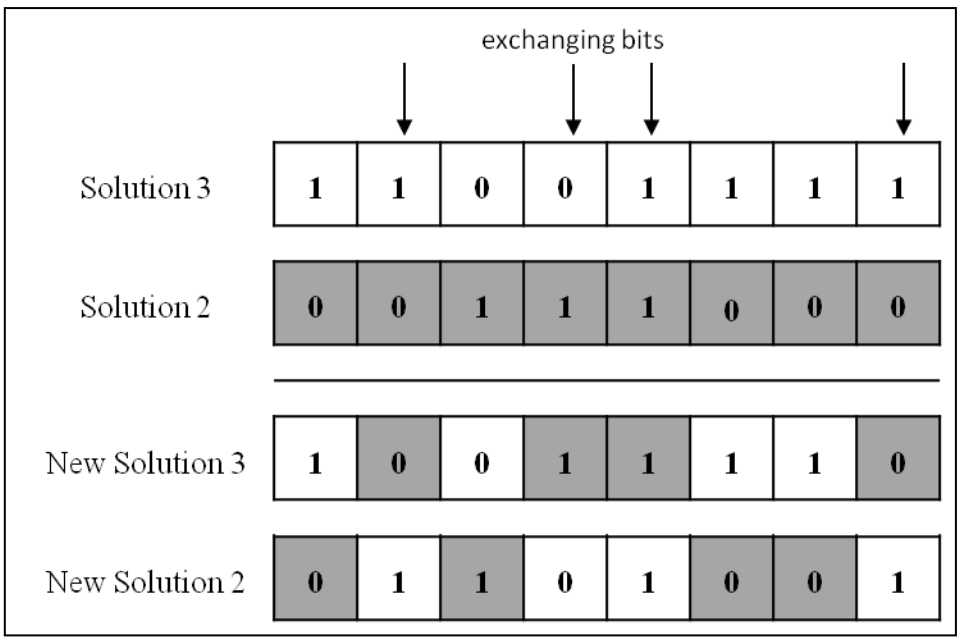


Fig. 2.10 - Uniform Crossover

In k-point crossover, when k is small compared to the string length, there is an implicit linkage between the positions that are close to each other, where they are most likely to stay together after the crossover, whereas in the uniform crossover, the bits are exchanged independently from each other. When k is very large, k-point crossover can actually be worse than uniform crossover in terms of linkage preservation. It is important to have this concept in mind for the next chapter, where we'll talk about the disruption factor that variation operators can cause.

2.6 Variation - Mutation

While recombination uses two or more individuals to create new solutions, mutation happens in each individual to create a new one by applying some kind of random change to the solution, based on a given probability P_m.

In this work, bit-flip mutation is used, where each bit of the solution has the same probability of being "flipped", which is usually small. For example, take the new solution 2 previously created by one-point crossover in Fig.7. If we set a P_m equal to 1/8, and we throw a 8 sided dice for each bit, the expected number of bits to flip is one. An example of bit flip mutation can be seen in fig. 11.

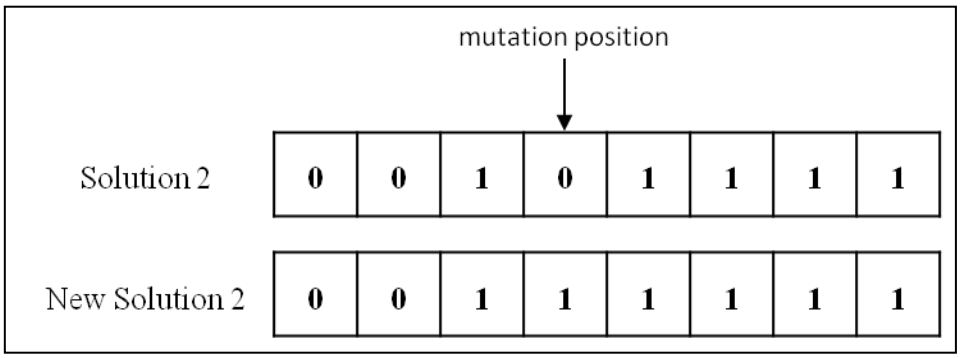


Fig. 2.11 - Mutation Example

2.7 Replacement

The role of replacement is to take the new individuals created by selection and variation, and incorporate some (steady-state replacement) or all (full replacement) in the original population.

In this work, full replacement is used, where the entire original population is replaced be the newly generated set of solutions.

2.8 Combining the components

Given a problem to be solved and combining the components shown above, a simple Genetic Algorithm works as follows:

```
Initialize a population with N random individuals
WHILE stopping conditions not met DO
        Select the best individuals using a selection operator
        Apply crossover to each pair with probability P_c
        Apply mutation to each bit with probability P_m
        Replace all or part of the existing population with the new solutions
END
```

Fig. 2.12 - Simple GA pseudo code

2.9 Stopping Conditions

The stopping conditions are the criterion by which the GA decides to continue or stop the search, being each criteria checked at each generation. There could be several different conditions such as, but not restricted to:

- Generation number - the GA stops when a specified number of generations is reached;
- Time based - the GA stops after running for a specified time period;
- Fitness threshold - the GA stops when a certain fitness value is found in a solution;
- Fitness convergence - the GA stops when the fitness of all solutions converged;
- Population convergence - the GA stops when all solutions have converged;

In this work, two stopping conditions are used. The GA stops if the population converges or if for 100 generations there is no improvement in the fitness.

2.10 Summary

In this chapter, we learned about the major components of a GA: representation, fitness function, initialization, selection, variation operators crossover and mutation, and replacement.

In this chapter we also saw that GAs work with a population that is firstly initialized and then combines and repeats several components until a stopping condition is met: First, the best individuals are chosen by applying selection to the population. Then the selected individuals are mixed by crossover, based on a given probability P_m, to generate new individuals. After, mutation is applied to them, according to a given probability P_m, to make small perturbations to the individuals.

Then the final step is to replace or update the original population with the newly generated individuals.

To know more about the basics of GAs please read chapter 3 of [11] and chapter 3 of [13].

3. Basic GA Theory

3.1 Schema Theory

The theoretical basis behind GAs, relies on the concept of schema (schemata in plural). A schema is a template that represents a set of solutions.

If a solution is a string built using symbols from an alphabet, we can say that schemata are strings made by that alphabet plus the character * (Alphabet U {*}). The * character works as a wildcard that can represent any given character from the existing alphabet.

For example, in a Binary alphabet, we have {0,1} and the * character can be either 1 or 0. Imagine that you have the following solution 110110110. This solution belongs to the schema 1******** and the schema *10******, among others, but doesn't belong to the schema *11*1**** because the 3rd position is different.

A schema has two important characteristics, the number of fixed positions (order) and the distance between the two outermost fixed positions (defining length).

The fitness of a schema is the average fitness of all the strings matching the schema. For instance, take H_1 as the following schema: 0***1**1* and H_2 as the following schema: *1**00*1*. H_1 has an order (o) of 3 and a defining length (δ) 7 and H_2 has an order(o) of 4 and a defining length (δ) of 6.

As we saw, a solution can be part of several schemata, so when the GA evaluates one solution, it's implicitly evaluating all of the schemata which that solution is a member of.

According to Holland [14], schemata that are short (low defining length) and low-order with above-average fitness (remember this for later) increase exponentially from generation to generation and below-average fitness schemata decay.

In the case of proportionate selection, one-point crossover and bit-flip mutation, this growth is given by the following inequality:

$$m(H,t+1) \geq m(H,t)\frac{\overline{f}(H,t)}{\overline{f}(t)}\left(1 - P_c\frac{\delta(\mathrm{H})}{l-1} - P_m o(H)\right) \quad (1)$$

where:

m(H,t) - number of instances of schema *H* at time t

$\overline{f}(H,t)$ - average fitness of all strings matching schema *H* at time *t*

$\overline{f}(t)$ - average fitness of the population at time *t*

δ(H) - defining length of schema *H*

o(H) - order of schema *H*

P_c - probability of crossover

P_m - probability of mutation

l - individual length

The previous inequality is known as Holland's *schema theorem*, which later Goldberg [15] , introduced the same formula independent of the GA operators:

$$m(H,t+1) \geq m(H,t)\frac{\overline{f}(H,t)}{\overline{f}(t)}\emptyset(H,t)[1-\epsilon(H,t)] \quad (2)$$

where:

$\emptyset(H,t)$ - Reproduction ratio of schema *H* at time *t* (due to selection)

$\epsilon(H,t)$ - Disruptor factor of schema *H* at time *t* (due to variation operators)

Growth rate:

$$\gamma = \emptyset(H,t)[1-\epsilon(H,t)] \quad (3)$$

So, if we want specific schemata to grow, we want $\gamma \geq 1$.

3.2 Building Blocks

As seen earlier, Holland's schema theorem suggested that GAs work by combining low order, short schemata with above average fitness. These schemata are called Building Blocks (BBs) [2].

Goldberg [15] found a good real life example to try to explain BBs:

"Just as a child creates magnificent fortresses through the arrangement of simple blocks of wood, so does a genetic algorithm seek near optimal performance through the juxtaposition of short, low order, high-performance schemata, or building blocks."

In the schema theory, we saw that a schema grows if its growth ratio is greater than 1.

$$\emptyset(H,t)[1-\epsilon(H,t)]>1 \quad (4)$$

For that to happen we can do two things, raise the selection pressure ($\emptyset(H,t)$) or decrease the disruptor factor ($[1-\epsilon(H,t)]$).

So to ensure that we have a good BB growth we cannot have very high disrupting factors or a very low selection pressure, but we'll see more in the next section.

3.2.1 Growing and mixing BBs

We need to understand that growing and mixing BBs [2] are different things. As we saw at the end of the last section, a schema grows if the growth ratio is greater than 1, and we can assure that in two ways, by raising the selection pressure or by lowering the disruption factor.

We want BBs to grow, and we can do it even with a very disruptive crossover, by simply raising the selection pressure or by controlling the amount of BB disruption by decreasing the probability that crossover happens (*Pc*). With this we get the following equation:

$$s(1-P_c)>1 \quad (5)$$

If we have a high selection pressure (*s*), quickly we'll have a uniform population, leading to the loss of diversity and then there will be no BBs exchange. Also, if we lower the crossover probability, we are lowering the number of BBs to exchange.

So to grow BBs, we have to combine good BBs from one solution with good BBs from another solution using crossover, where we want to minimize the BB disruption and maximize the BB exchange.

3.3 Additive Decomposable Functions

An additively decomposable function is a function which can be written as a sum of sub functions, each of which depends on only a small number of variables [6]. Next we present two popular examples of additive decomposable functions used in this work.

3.3.1 Onemax function

The onemax function, contains only one optimal solution, making it unimodal and easy for a GA to accomplish.

$$f_{onemax}(x) = \sum_{i=1}^{l} x_i \quad (6)$$

The previous equation gives us the onemax fitness function of a solution $x=(x_1,x_2,...,x_l)$ of size l. The function value is the sum of the x_i, with i *in* [1..l]. For binary strings this corresponds to counting the number of 1's.

In onemax we can say that BBs are of one bit size and they are not destroyed by crossover.

3.3.2 Trap function of size k

A trap function, also known as a deceptive function [1], can be easily build from the onemax function. It contains two optimal solutions, one local and one global, where the global solution is the best in the entire search space, and the local one, inferior to the best solution, is usually known as a deceptive solution.

In concatenated trap functions, the input solution is partitioned into m independent groups of k bits each. This means that in a solution of length l, there are $m=l/k$ traps.

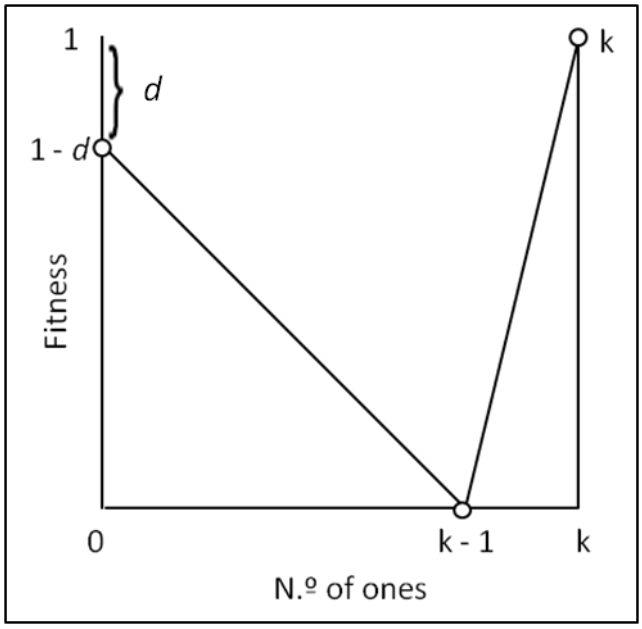


Fig. 3.1 - k-bit Trap Function

Fig. 13 gives us the representation of the *k*-bit trap function and has the following equation:

$$trap_k(x_1, x_2, \dots, x_k) = \begin{cases} 1 & if\ f_{onemax}(x_1, x_2, \dots, x_k) = k \\ 1 - d - \frac{f_{onemax}(x_1, x_2, \dots, x_k)}{k} & otherwise \end{cases} \quad (7)$$

With *d* being the difference between the local optima and the global optima.

The overall fitness function is the sum of all trap functions and is given by the following equation:

$$f_{trap_k}(x) = \sum_{i=0}^{m-1} trap_k(x_{i*k+1}, x_{i*k+2}, \dots, x_{i*k+k}) \quad (8)$$

Usually GAs have many difficulties in solving this function, because individuals are attracted to the deceptive solution, since the fitness of all the solutions are nearest to the local optimum instead of the global optimum.

In *k*-trap functions we can say that BBs are of *k* bit size and can be destroyed by crossover.

3.4 Population Sizing

The population size [3][4] in a GA is one important factor in determining the quality of convergence (success of the GA). If a GA works by growing BBs, the initial supply of BBs and the selection of the best BBs are very important.

Large populations show better results over smaller populations, but how to determine the right amount or how much larger a population has to be, to guarantee the success of the GA?

Although GAs are operationally simple, they are complex algorithms. To understand and design better, we need to see them as a divide and conquer approach. This is, we need to decompose the problem, into smaller decomposable sub-problems, solve the sub-problems, and integrate them as a whole solution.

3.4.1 Decomposing the problem

According to Goldberg, Deb and Clark [3], the decomposition of a GA consists of six points:

- Know what GAs are processing: BBs
- Solve problems tractable by BBs
- Supply enough BBs in the initial population
- Ensure the growth of good BBs
- Mix the BBs properly
- Decide well between competing BBs

The population sizing model used in this work is based on two of these points: The initial supply of BBs and the decision process between competing BBs.

3.4.2 Deciding between two BBs

Consider two BBs, H_1 and H_2, each with their mean fitness $f(H_1)$ and $f(H_2)$ and each with a fitness variance $\sigma^2(H_1)$ and $\sigma^2(H_2)$. Assuming that the fitness distributions are normal by the central limit theorem and that $f(H_1) > f(H_2)$, we'll have more solutions of H_1 than of H_2. However when competing in a one-on-one there's a chance that H_2 will be chosen by selection [4].

To calculate the probability of choosing wrong between two BBs (choosing H_2 instead of H_1), the following equation was set [3]:

$$p = \mathbb{N}\left(\frac{d}{\sqrt{\sigma^2(H_1)+\sigma^2(H_2)}}\right) \quad (9)$$

where:

$\mathbb{N}$ - Cumulative distribution function for a normal distribution of mean 0 and unit standard distribution.

d - $f(H_1) - f(H_2)$, signal difference between the fitness of each BB.

Following Golberg, Deb and Clark [3], we assume that the fitness function is the sum of *m* independent sub functions *fi*, each of size *k* , so the overall variance of the function can be calculated as:

$$\sigma_f^2 = \sum_{i=1}^{m} \sigma_{f_i}^2 \quad (10)$$

So, the average BB variance can be computed by $\sigma_{BB}^2 = \sigma_f^2/m$.

Since in a GA we are working with multiple BBs at the same time, we must take in consideration that for each BB competition, there is disturbance from the other $m' = m - 1$ BBs: $\sigma^2 = m'\sigma_{BB}^2$. So the probability of making the right decision between a single sample of each BB is given by the following equation:

$$p = \mathbb{N}\left(\frac{d}{\sqrt{2m\prime\sigma_{BB}}}\right) \quad (11)$$

3.4.3 Gambler's ruin model

The gambler's ruin model has been extensively studied and is a classic example of a random walk (mathematical tools that are used to predict the outcome of certain stochastic processes), with absorbing barriers.

In this model, the gambler has a starting capital (x_0) and plays against an opponent with a capital given by (n-x_0), where n is the maximum absorbing barrier (winning all the money) and 0 the minimum barrier (bankruptcy).

At each step of the game, the gambler has a probability p of increasing his capital by 1 and a probability $q=1-p$ of losing to the opponent.

In the next figure we can see 5 gambler's ruin walks, with p=50% and initial capital x_0=50 and total capital n=100.

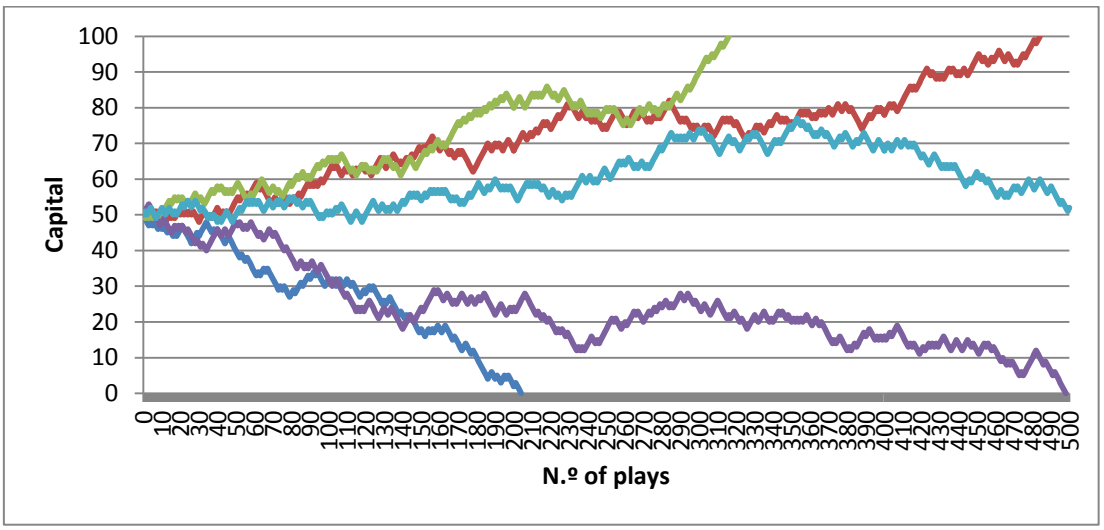


Fig. 3.2 - Gambler's Ruin Walks

Now, imagine the same idea applied to the decision process of the GA representing one Building Block. The initial capital x_0 represents the initial number of correct BBs in the population. n represents the convergence to right solutions and 0 to the wrong solutions. In a randomly initialized population, $x_0=n/2^k$, being k the order of the BB.

This model abandons the notion of generations in a GA and considers the decisions made one at a time until the all n BBs converged to the same value. Also, it assumes that the GA uses tournament selection [4].

From the random walk literature [16], the probability of a value being caught in the absorbing barrier n is given by:

$$P_n = \frac{1-(q/p)^{x_0}}{1-(q/p)^n} \quad (12)$$

As seen earlier the probability of choosing the correct BB, is higher than choosing the wrong BB ($p > q$). Also, x_0 tends to be small when compared with the population size and assuming that $1-(q/p)^n$ approaches 1, we can simplify the above equation to:

$$P_n \approx 1 - (\frac{1-p}{p})^{\frac{n}{2^k}} \quad (13)$$

If P_n is the probability of a value being caught in the absorbing barrier n, we can say that $\alpha=1-P_n$ is the probability of failure, this is, the probability of being caught in the absorbing barrier 0. Solving equation 13 for n, we get:

$$n = \frac{2^k \ln(\alpha)}{\ln(\frac{1-p}{p})} \quad (14)$$

Considering that p can be approximated to the first two terms of the power series expansion for the normal distribution as [17]:

$$p = \frac{1}{2} + \frac{1}{2}x \quad (15)$$

where: $$x = d/(\sigma_{BB}\sqrt{\pi m'}) \quad (16)$$

and replacing in equation 9, we get our final population equation:

$$n = -2^{k-1} \ln(\alpha) \frac{\sigma_{BB}\sqrt{\pi m'}}{d} \quad (17)$$

With this formula we can see that, if the average variance of BBs increase or the problem size increases the population size will increase. On the other hand if the signal difference increases or the probability of failure increase, the population size will decrease.

3.5 Summary

In this chapter we saw what GAs are processing, Building Blocks, short (low defining length) and low-order with above-average fitness schemata.

In this chapter, we also had a brief introduction to additive decomposable functions (ADF) and saw 2 examples of ADFs that we will use in the next chapter.

In this chapter, we also saw that the population size is a major factor for success of the algorithm, and that the optimal population size can be calculated based on the initial supply of BBs and the decision process between competing BBs.

4. Visualizing Onemax and Traps

4.1 Solution Encoding, visualizing an image

A black and white (B&W) image is composed by pixels that can be either black or white, and is defined by a width (*w*) and a height (*h*). For example, a 10 by 10 B&W image has 100 pixels.

Consider the following 10 by 1 B&W image:

Fig. 4.1 - 10*1 B&W Image

This image can be represented by a binary string where black represents 1 and white represents 0.

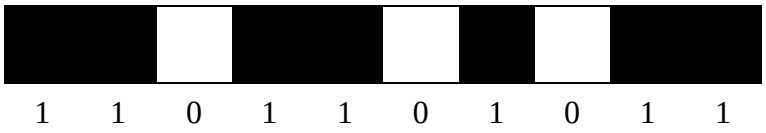


Fig. 4.2 - 10*1 B&W Image with Encoding

Having this in context, consider the next solution represented by a 100 bit string:

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

The string can be interpreted as the following B&W 10 by 10 image:

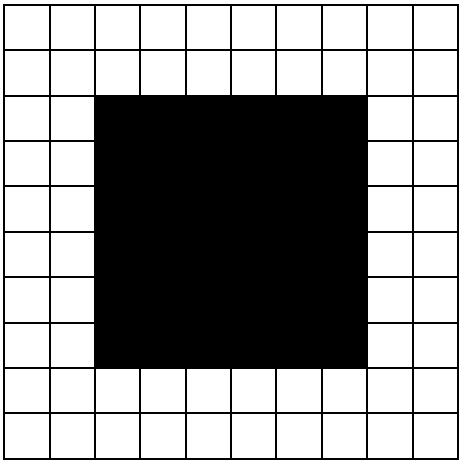


Fig. 4.3 - 10*10 B&W Image

4.2 Onemax as bitmap optimization

In onemax, the global optimum is the maximization of the function by having all the properties of a string equal to one. To adapt that concept to an image optimization, one has to do a small adaptation of the onemax function.

$$imageOptimization(x_i) = \begin{cases} 1 & if\ x_i = final_i \\ 0 & otherwise \end{cases} \quad (18)$$

If the property x_i of a solution equals the same property $final_i$ of the optimal image, we add 1, otherwise we add 0. With i in [1..l], the function value given by equation 19, is the sum of 1 for each property of the solution x=(x_1,x_2,...,x_l) that is equal to the correspondent property on the optimal solution $final$=($final_1$,$final_2$,...,$final_l$):

$$f_{imageOptimization}(x) = \sum_{i=1}^{l} imageOptmization(x_i) \quad (19)$$

For example, consider the solutions presented by fig. 18:

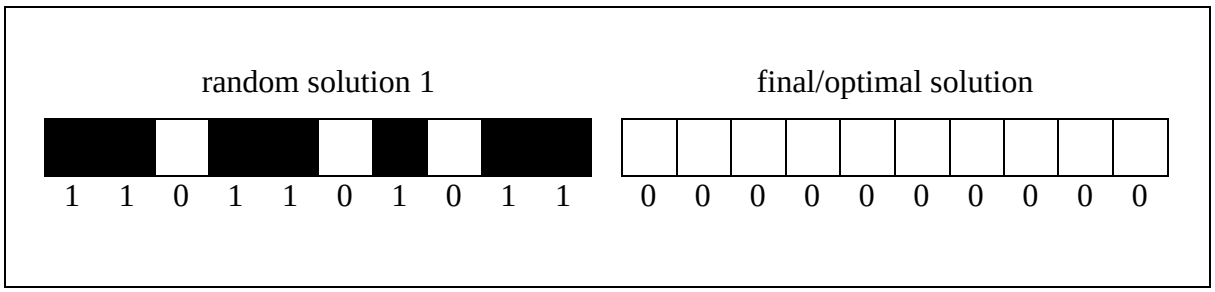


Fig. 4.4 - Visualization of two ten bit solutions

Calculating the fitness of the random solution 1 we'll have:

$f_{imageOptimization}$(1,1,0,1,1,0,1,0,1,1):

$$imageOptimization(1) = 0\ because\ (x_1 = 1) \neq\ (final_1 = 0)$$
$$imageOptimization(1) = 0\ because\ (x_2 = 1) \neq\ (final_2 = 0)$$
$$imageOptimization(0) = 1\ because\ (x_3 = 0) = (final_3 = 0)$$

$imageOptimization(1) = 0\ because\ (x_4 = 1) \neq (final_4 = 0)$
$imageOptimization(1) = 0\ because\ (x_5 = 1) \neq (final_5 = 0)$
$imageOptimization(0) = 1\ because\ (x_6 = 0) = (final_6 = 0)$
$imageOptimization(1) = 0\ because\ (x_7 = 1) \neq (final_7 = 0)$
$imageOptimization(0) = 1because\ (x_8 = 0) = (final_8 = 0)$
$imageOptimization(1) = 0\ because\ (x_9 = 1) \neq (final_9 = 0)$
$imageOptimization(1) = 0\ because\ (x_{10} = 1) \neq (final_{10} = 0)$

$$f_{imageOptimization}(1,1,0,1,0,1,0,1,1) = 0 + 0 + 1 + 0 + 0 + 1 + 0 + 1 + 0 + 0 = 3$$

The random solution 1 presented on fig.18 has a fitness value of 3.

Now consider the random solution 2 presented in fig. 19:

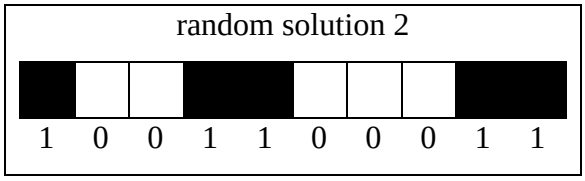


Fig. 4.5 - Vizualization of a random ten bit solution

Calculating $f_{imageOptimization}(1,0,0,1,1,0,0,0,1,1)$:

$imageOptimization(1) = 0\ because\ (x_1 = 1) \neq (final_1 = 0)$
$imageOptimization(0) = 1\ because\ (x_2 = 0) = (final_2 = 0)$
$imageOptimization(0) = 1\ because\ (x_3 = 0) = (final_3 = 0)$
$imageOptimization(1) = 0\ because\ (x_4 = 1) \neq (final_4 = 0)$
$imageOptimization(1) = 0\ because\ (x_5 = 1) \neq (final_5 = 0)$
$imageOptimization(0) = 1\ because\ (x_6 = 0) = (final_6 = 0)$
$imageOptimization(0) = 1\ because\ (x_7 = 0) = (final_7 = 0)$
$imageOptimization(0) = 1because\ (x_8 = 0) = (final_8 = 0)$
$imageOptimization(1) = 0\ because\ (x_9 = 1) \neq (final_9 = 0)$
$imageOptimization(1) = 0\ because\ (x_{10} = 1) \neq (final_{10} = 0)$

$$f_{imageOptimization}(1,1,0,1,0,1,0,1,1) = 0 + 1 + 1 + 0 + 0 + 1 + 1 + 1 + 0 + 0 = 5$$

Comparing the two random solutions fitness's we can see that the random solution 2 is better than random solution 1. Although this happens because random solution 2 has more equal individual proprieties, when using trap functions this isn't always true has we'll see in the next section.

4.3 From Onemax to Trap Function

In a trap function the concept used for onemax still applies, but instead of having to compare 1 property, we have to compare *k* properties at a time, in which we have to check how many are equal to the final image, and use the equation 19:

$$image_k size(x_j) = \begin{cases} 1 & if\ x_j = final_j \\ 1 - d - \frac{f_{imageOptimization}(x_j)}{k} & otherwise \end{cases} \quad (20)$$

With *j* in [1..l], x_j and $final_j$ represent a subgroup of *k* bits and *d* represents the difference between the global optima and the local optima. The overall fitness function is given by the sum of all trap functions:

$$f_{image_k size}(x) = \sum_{j=0}^{m-1} image_k size(x_j) \quad (21)$$

Where *m* represents the number of subgroups.

For example, consider the 10 bit solutions and a trap size of *k*=2 (*m*=5) presented by fig.20:

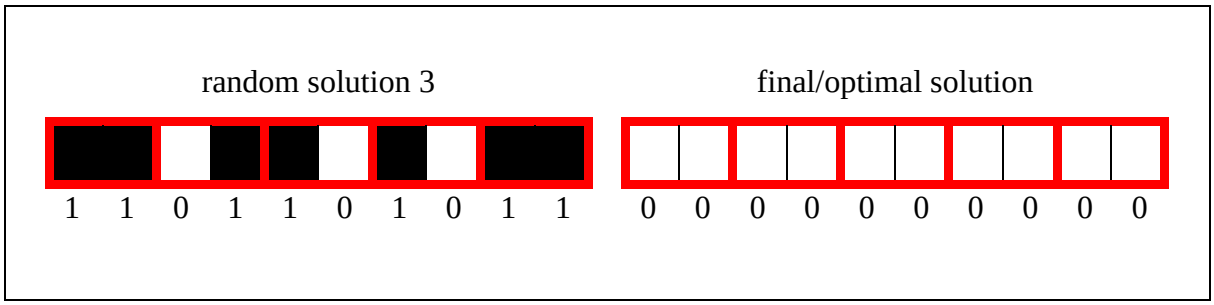


Fig. 4.6 - Visualization of two ten bit solutions with traps of size *k*=2

Assuming that difference between the optima is *d*=1/*k*, calculating the fitness of random solution 3, we'll have:

$f_{image_k size}(1,1,0,1,1,0,1,0,1,1)$:

$image_k size(1,1) = 1 - \frac{1}{2} - 0 = \frac{1}{2}$ $because$ $(x_1 = 1) \neq (final_1 = 0)$ and $(x_2 = 1) \neq (final_2 = 0)$

$image_k size(0,1) = 1 - \frac{1}{2} - \frac{1}{2} = 0$ $because$ $(x_3 = 0) = (final_3 = 0)$ and $(x_4 = 1) \neq (final_4 = 0)$

$image_k size(1,0) = 1 - \frac{1}{2} - \frac{1}{2} = 0$ $because$ $(x_5 = 1) \neq (final_5 = 0)$ and $(x_6 = 0) = (final_6 = 0)$

$image_k size(1,0) = 1 - \frac{1}{2} - \frac{1}{2} = 0$ $because$ $(x_7 = 1) \neq (final_7 = 0)$ and $(x_8 = 0) = (final_8 = 0)$

$image_k size(1,1) = 1 - \frac{1}{2} - 0 = \frac{1}{2}$ $because$ $(x_9 = 1) \neq (final_9 = 0)$ and $(x_{10} = 1) \neq (final_{10} = 0)$

$f_{image_k size}(1,1,0,1,1,0,1,0,1,1) = \frac{1}{2} + 0 + 0 + 0 + \frac{1}{2} = 1$

The random solution 3 presented in fig. 20 has a fitness value of 1.

Now consider the random solution 4 presented by fig. 21:

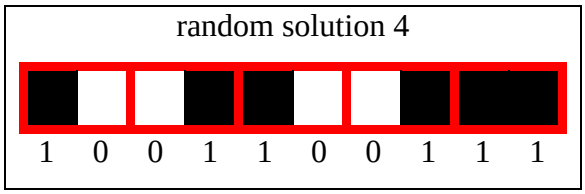


Fig. 4.7 - visualization of a random ten bit solution with traps of size k=2

Calculating the fitness value of random solution 4, we'll have:

$f_{image_k size}(1,0,0,1,1,0,0,1,1,1)$:

$image_k size(1,0) = 1 - \frac{1}{2} - 0 = 0$ $because$ $(x_1 = 1) \neq (final_1 = 0)$ and $(x_2 = 0) = (final_2 = 0)$

$image_k size(0,1) = 1 - \frac{1}{2} - \frac{1}{2} = 0$ $because$ $(x_3 = 0) = (final_3 = 0)$ and $(x_4 = 1) \neq (final_4 = 0)$

$image_k size(1,0) = 1 - \frac{1}{2} - \frac{1}{2} = 0$ $because$ $(x_5 = 1) \neq (final_5 = 0)$ and $(x_6 = 0) = (final_6 = 0)$

$image_k size(0,1) = 1 - \frac{1}{2} - \frac{1}{2} = 0$ $because$ $(x_7 = 0) = (final_7 = 0)$ and $(x_8 = 1) \neq (final_8 = 0)$

$image_k size(1,1) = 1 - \frac{1}{2} - 0 = \frac{1}{2}$ $because$ $(x_9 = 1) \neq (final_9 = 0)$ and $(x_{10} = 1) \neq (final_{10} = 0)$

$$f_{image_k size}(1,1,0,1,1,0,1,0,1,1) = 0 + 0 + 0 + 0 + \frac{1}{2} = \frac{1}{2}$$

Although random solution 4 is more similar to the optimal solution, since it has more equal individual properties than solution 3, when comparing the fitness of random solution 4 with the fitness of random solution 3, we can see the latter is better. With this example we can see how a trap function can be deceptive.

4.4 Summary

In this chapter we saw how a B&W image can be represented/encoded using a binary string. 1 represents black and 0 represents white. We saw how to calculate the fitness of a given solution using a similar approach to the onemax and how to calculate the fitness when using trap functions.

5. The Program

The program was built with the intention of demonstrating the previous GA theory, using black and white images examples, which are encoded as binary strings, to better understand how a GA works.

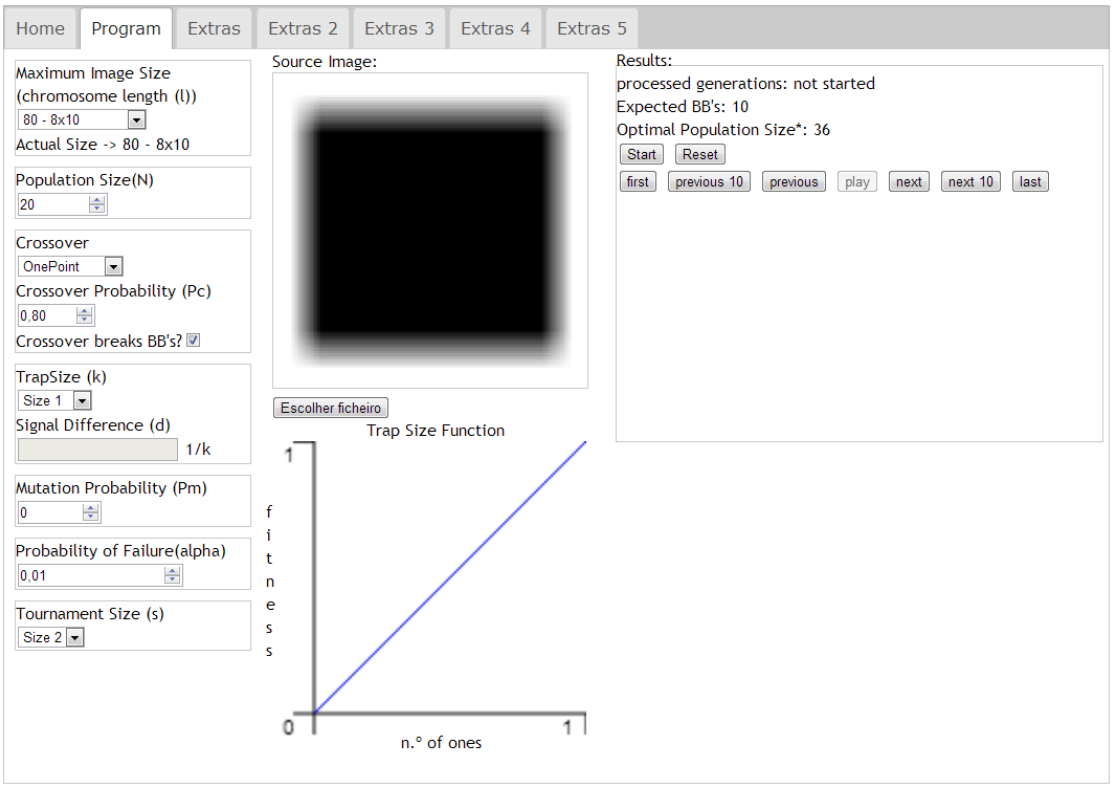


Fig. 5.1 - The Program

Fig.22 shows us the program that can be found at http://gas.byethost.com/#program

5.1 Used Technologies

5.1.1 HTML

HTML[8] - HyperText Markup Language is a W3C [7] standard. It is the core language of the World Wide Web and it's used to construct web pages. It's a simple language written in the form of elements consisting of tags enclosed in angle brackets (Hello World!), available in any operating system through the use of web browsers that read HTML documents, not displaying the tags but interpreting them, to compose readable web pages. HTML5, the lasted W3C Candidate Recommendation is used in this work.

5.1.2 CSS

CSS [9] - Cascading Style Sheets is the language for describing the presentation of Web pages, including colors, layout and fonts. It is independent of HTML and allows to adapt the presentation to different types of devices such as screens or printers.

5.1.3 Javascript

Javascript [10] is a scripting language, used to add functionalities to web pages, processed in the client-side. Its syntax was influenced by the language C, and the key design principles are taken from the Self and Scheme programming languages, with the support of object-oriented, imperative and functional programming styles.

5.1.4 JSON

JSON (JavaScript Object Notation) [19], is a lightweight data-interchange format that is easy for humans to read and write and also for machines to parse and generate. As the name says it is based on Javascript. It is a text format that is language independent. JSON is build on two structures: a collection of names/values that in different languages is realized as an object, record, struct, record, among other. An ordered list of values that can be considered by different languages as an array, vector, list or sequence.

5.1.5 Ajax

Ajax (Asynchronous JavaScript and XML) [20], is a way to use the existing standards, by exchanging data between the server and the client and updating parts of a webpage without the need to reload the whole page. In this work Ajax is used to load the preloaded results from existing files on the server.

5.2 Pedagogical tool

Teachers use examples routinely, either to illustrate an idea, visualize an abstract concept, or to help students understand complex procedures. If applied appropriately, examples can be powerful tools in a student learning process. Research shows that in the early phases of learning something new, students learn better and faster by studying working examples than by solving problems on their own [18].

This program tries to be a good source of GAs examples that can be used routinely in Evolutionary Computation courses or in other sources of learning and studying about GAs.

5.3 Program options

The program offers several different options, allowing to control several aspects/components of the GA. This allows to have a better perspective on how each aspect affects the performance of the GA and the final results.

Also, with some of the program options it is possible to apply the previous Gambler's Ruin Model to determine the optimal population size.

5.3.1 Maximum Image Size (*l*)

It is possible to choose from six different maximum image sizes (solution length), with each size having a simple pre-loaded image. The sizes vary from a minimum solution length of 80 bits, corresponding to a 8x10 image, to a maximum solution length of 14400 bits, corresponding to a 120x120 image. Fig. 23 shows this option and fig. 24 shows the pre-loaded image of this option.

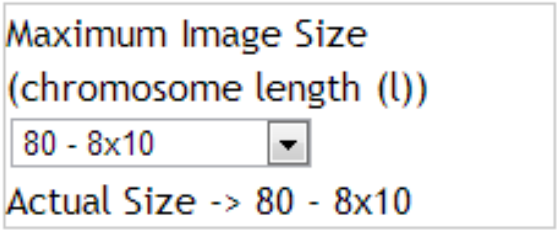


Fig. 5.2 - Maximum Image Size option

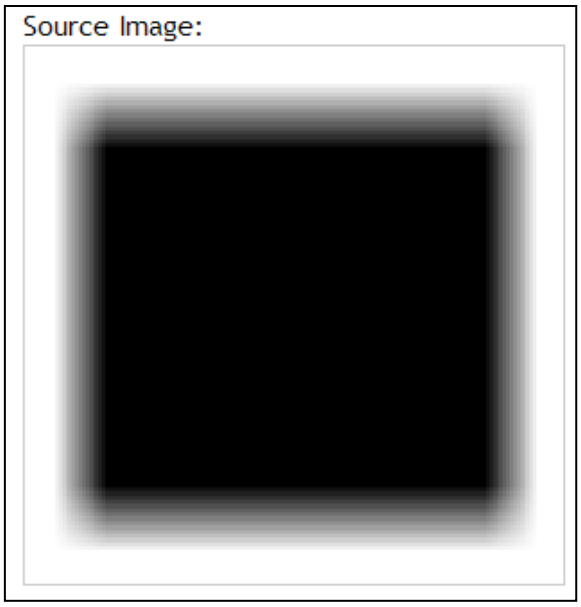


Fig. 5.3 - 8x10 Pre-loaded image

As we saw on the gambler's ruin population equation 17, if we increase the problem size, in this case the image size or solution length, the optimal population size will also increase.

5.3.2 Population Size (*n*)

The population can vary from 20 to 520000 according to the image size. If the maximum image size is 80bits, the highest population size allowed is 520000, if the maximum image size is 14400bits, the maximum population size is 3600. This limitation is due to javascript memory allocation. This option is shown in fig. 25.

Fig. 5.4 - Population Size (*n*) option

The initial population size will determine the expected supply of BBs in the initial population, which is one of the points in which the population sizing model is based on.

5.3.3 Crossover and Crossover Probability (*Pc*)

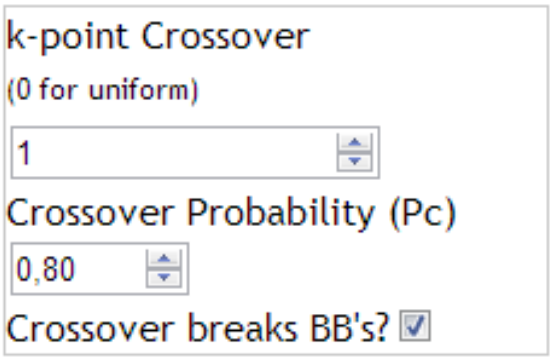


Fig. 5.5 - Crossover and Crossover Probability (*Pc*) option

In the crossover option shown by fig. 26, we can choose the type of crossover, either uniform crossover with the option 0, or *k*-point choosing the number of points. We can also choose the probability which we want the crossover to happen (*Pc*). There is also the option to choose if we want the crossover to break BBs (on by default).

The number of crossover points can vary if the option for breaking BBs is turned off. This is because if the crossover happens at the point of the trap size, the BBs will not break and the maximum number of crossover points will be the number of BBs (*m)* minus 1.

With this option we can control the part of the disruption factor mentioned in section 3.2.1

5.3.4 Trap Size (*k*) / Signal Difference(*d*)

In this option we can choose the size of the traps between 1,2,4,5,8,10 and 20.

Since the size of the trap must be a multiple of the length of the solution, as seen in section 3.3.2, the length of the solution may vary if the size of the trap changes .

The signal difference option controls the difference between the local optima and the global optima. By default, the value is set to 1/k, being selected either with 0 or 1. The

lower the value, the near the local optima is to the global optima and the higher the value, more apart the optima are. These 2 options are shown by fig. 27.
A image of a trap function is updated with these options as fig. 28 and 29 shows.

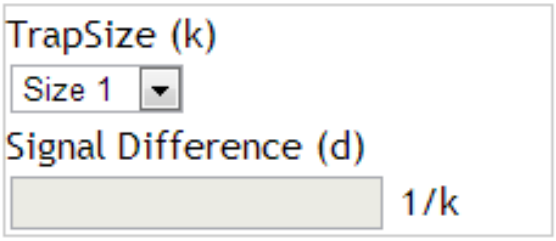


Fig. 5.6 - Trap Size (*k*) / Signal Difference (*d*) option

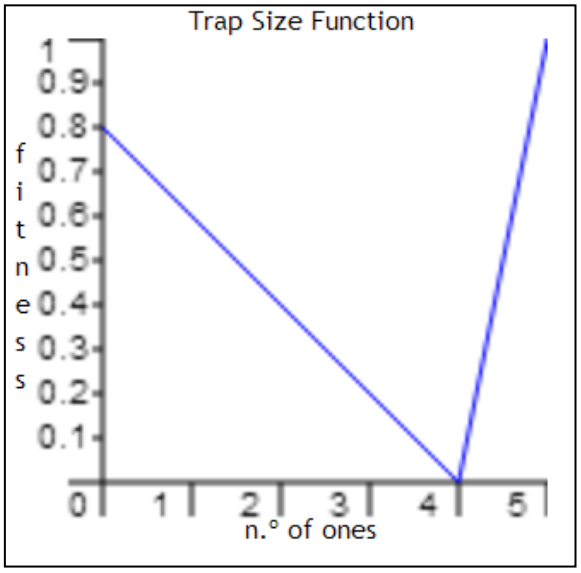


Fig. 5.7 - Trap Size Function with *k*=5 and *d*=1/*k*

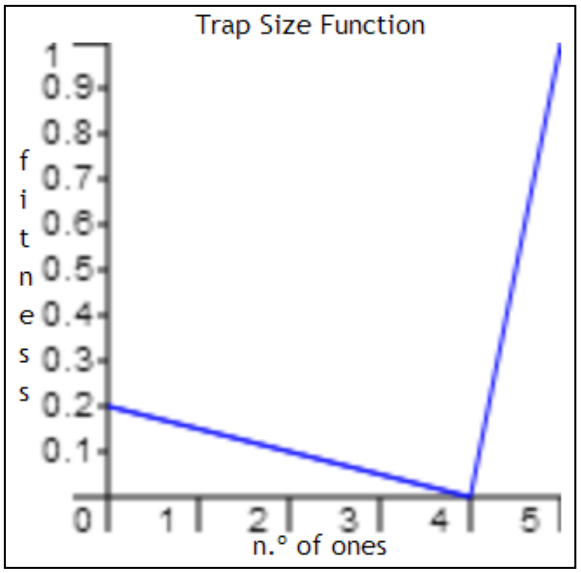


Fig. 5.8 - Trap Size Funtion with k=5 and d=0.8

The size of the trap k and the signal difference *d* are the options with more weight to the gambler's ruin population equation 17. The higher the trap size the higher the optimal population size. The higher the signal difference the lower the optimal population size.

5.3.5 Mutation Probability (*Pm*)

In this option, shown by fig. 30, we can control the probability of mutation, being the default value 0, which means that there is no mutation.

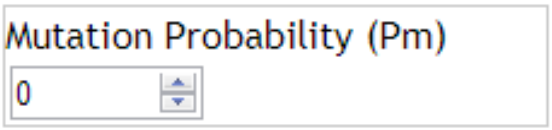


Fig. 5.9 - Mutation Probability (*Pm*) option

5.3.6 Probability of Failure (α)

With this option, shown by fig. 31, we can choose the probability of failure α, which tells us how many BBs are expected to converge incorrectly. For example, imagine we have a solution with a length l=1000 bits and a trap size k=2 ($m = l/k$ = 500 BBs). If we set a 1% (0,01) probability of failure, the expected number of BBs incorrectly solved is 5.

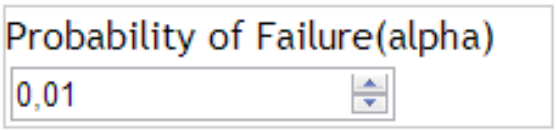


Fig. 5.10 - Probabilty of Failure (α) option

This option is part of the gambler's ruin equation 17, and the lower the value the higher the optimal population size.

5.3.7 Tournament Size (*s*)

In this option we can choose from three tournament sizes: size 2, size 4 and size 5.

This option, shown by fig.32, controls the selection pressure mentioned in section 3.2.1

Fig. 5.11 - Tournament Size (*s*) option

5.3.8 Image Loading

With this option it is possible to load an image from the computer or any other available source. The image is automatically transformed into a black and white image and it automatically rearranges itself when the maximum image size is changed. Fig. 33 shows an example of a loaded image.

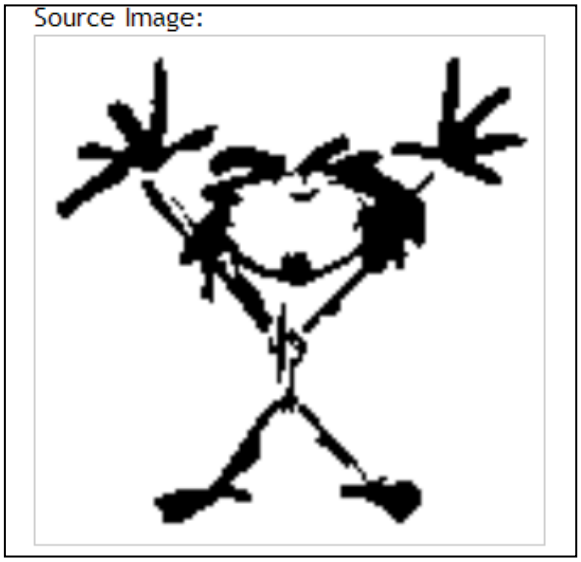


Fig. 5.12 - Loaded Image with size length: 14400 - 120x120

5.3.9 Results

In the results area we can find the number of processed generations if the program is running or it ended. We can also find the number of expected BBs in the population, given by the population size *n*. Next, we can see the optimal population size, calculated by the gambler's ruin equation 17 using the previously referred options. Fig. 34 shows us these results.

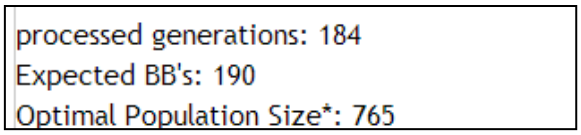


Fig. 5.13 - processed generations, expected BBs and optimal Population size

Afterwards we'll find a first set of buttons, as shown by fig. 35, that control the GA. It can start the GA, stop the GA, restart the GA after it's finished or reset the GA so the previous referred options are available to the user. This happens because when the program is running the options are unavailable.

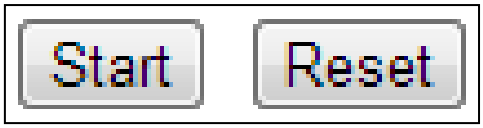


Fig. 5.14 - GA start, stop, restart and reset buttons

Next, fig. 36 shows a second set of buttons that control the animation, that is composed by the gambler's ruin random walk for each BB as seen in fig. 37, and the best solution of the current generation as seen in fig. 38. The first button shows generation 0, then previous buttons show 1 or 10 previous generations, the play button starts or stops the animation, the next buttons show 1 or 10 next generations and the last button, shows the last generation processed.

Fig. 5.15 - Animation control buttons

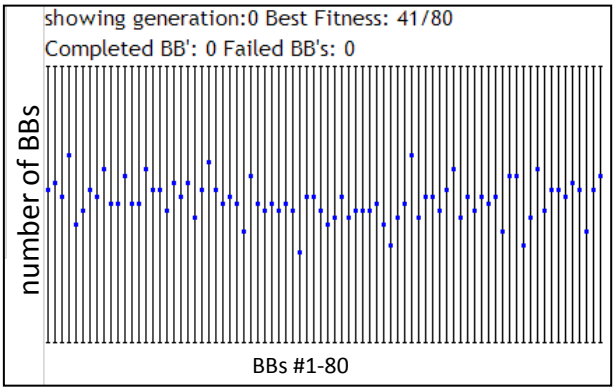


Fig. 5.16 - BBs Random walks at generation 0 (just initialized)

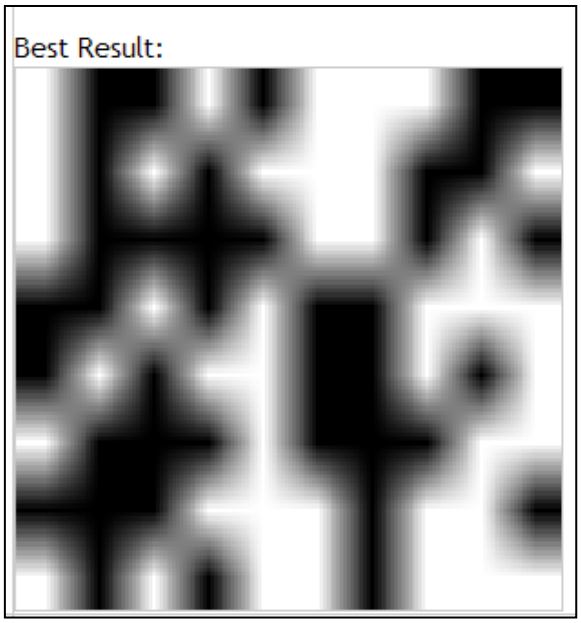


Fig. 5.17 - Best solution at generation 0 (just initialized)

As the next figure shows, we can see each BB random walk independently, by clicking the desired BB in the BBs Random walks.

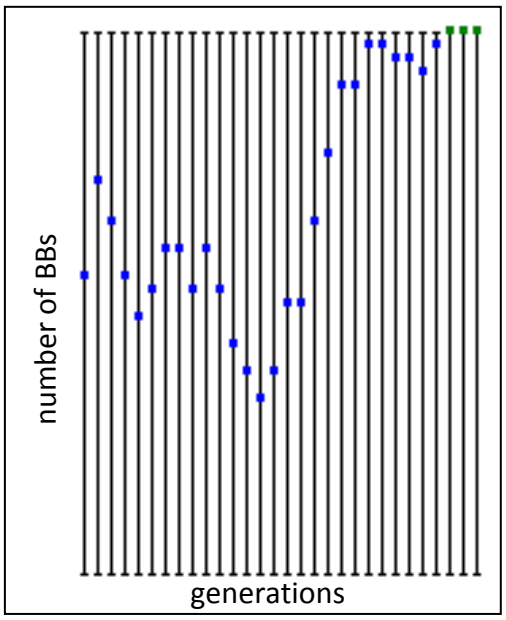
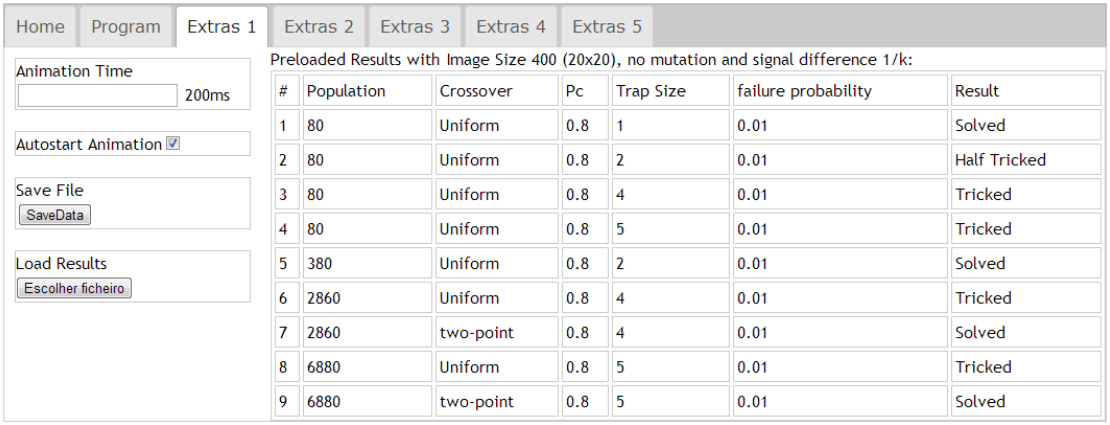


Fig. 5.18 - Random walk for 1 BB in a GA finished at generation 29

5.4 Extra Options

5.4.1 Extras 1



#	Population	Crossover	Pc	Trap Size	failure probability	Result
1	80	Uniform	0.8	1	0.01	Solved
2	80	Uniform	0.8	2	0.01	Half Tricked
3	80	Uniform	0.8	4	0.01	Tricked
4	80	Uniform	0.8	5	0.01	Tricked
5	380	Uniform	0.8	2	0.01	Solved
6	2860	Uniform	0.8	4	0.01	Tricked
7	2860	two-point	0.8	4	0.01	Solved
8	6880	Uniform	0.8	5	0.01	Tricked
9	6880	two-point	0.8	5	0.01	Solved

Fig. 5.19 - Extras 1

Fig. 40 shows the Extras 1 page and it can be visited at http://gas.byethost.com/#extras1

5.4.1.1 Animation Time

This extra option controls the animation time between each generation shown in the results area. It also controls the time between each play in the extra #5. The animation goes faster the smaller the value or slower the higher the value.

5.4.1.2 Save Results

This option allows the user to save the results obtained with the used options. The saved file is a text file containing JSON (JavaScript Object Notation).

5.4.1.3 Load Results

This option allows the user to load a previous saved result with the used options.

5.4.1.4 Preloaded Results

In this area we can find a table with 9 preloaded results with image size l=400, no mutation and signal difference d=1/k. The table shows the selected options used to obtain the preloaded result which are also loaded into the program to try and repeat the same result.

5.4.2 Extras 2 (Random initialization of a Population)

In this extra, it shows a random initialization of a population encoded in binary strings. We can choose the string length and the size of the population.

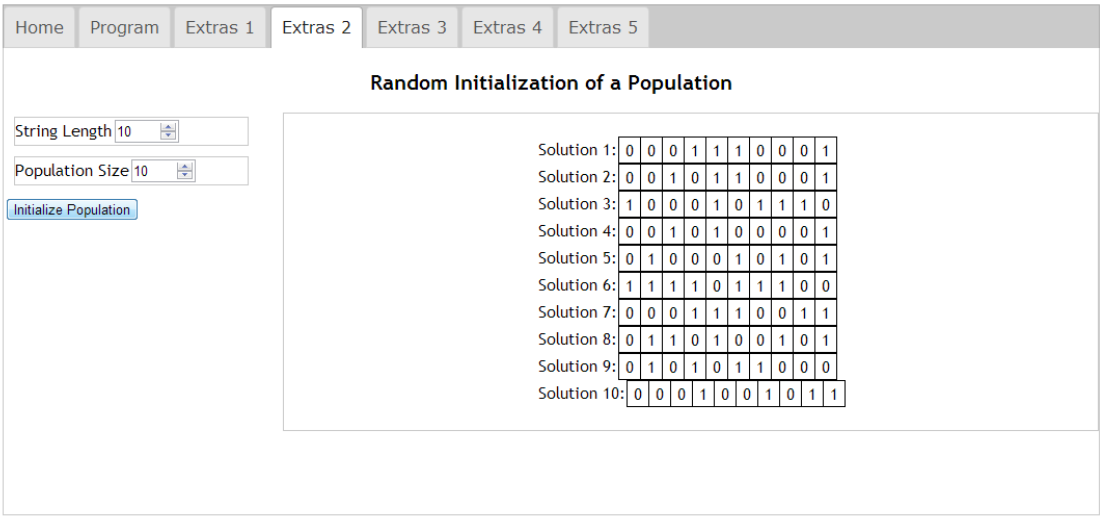


Fig. 5.20 - Extras 2

Fig. 41 shows the Extras 2 page and it can be visited at http://gas.byethost.com/#extras2

5.4.3 Extras 3 (Crossover Example)

In Extras 3, we can find random examples on different crossover operators on binary strings. One-point, Two Point, k-point and uniform crossover.

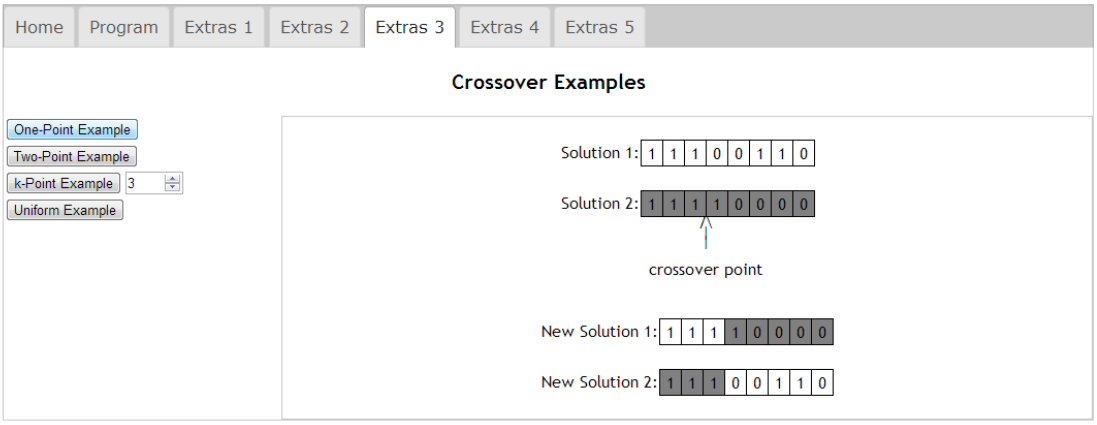


Fig. 5.21 - Extras 3

Fig. 42 shows the Extras 3 page and it can be visited at http://gas.byethost.com/#extras3

5.4.1 Extras 4 (Mutation Example)

In this extra we can find bit-flip mutation example on binary strings. We can choose the probability of mutation between 0 to 1.

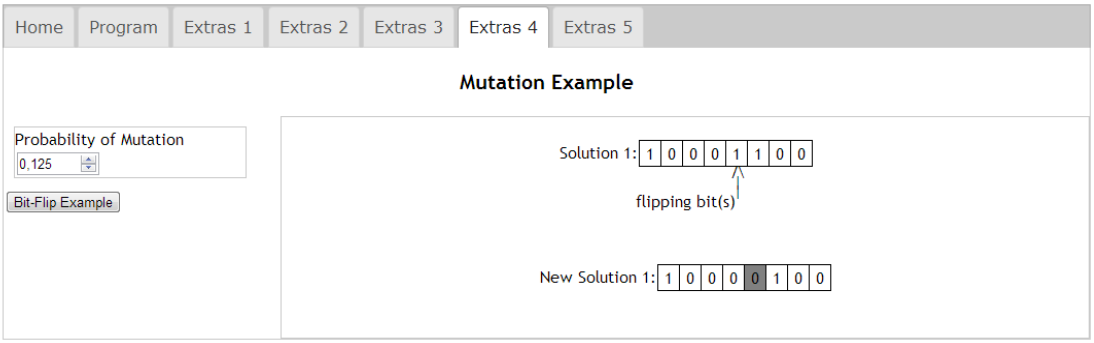


Fig. 5.22 - Extras 4

Fig. 43 shows the Extras 4 page and can be visited at http://gas.byethost.com/#extras4

5.4.1 Extras 5 (Gambler's Ruin/Random Walk)

In Extras 5 we can see the gambler's ruin problem in the form of a real animation showing the current capital of the gambler at a determined step time. We can choose the probability of winning and the starting capital of the gambler. The animation time can be controlled by the option in Extra 1.

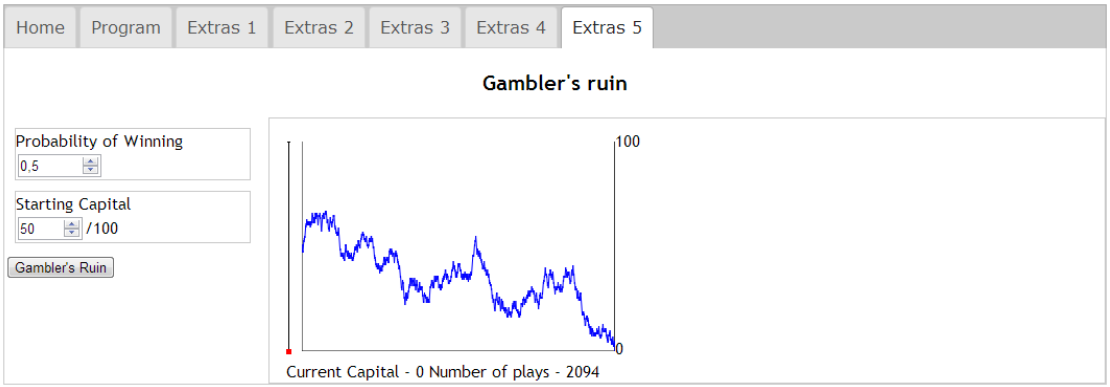


Fig. 5.23 - Extras 5

Fig. 44 shows the Extras 5 page and it can be visited at http://gas.byethost.com/#extras5

6. Summary and Conclusions

This thesis started by giving an introduction to Genetic Algorithms and their components. These components individually are easy to understand, but when combined to form a GA, the theoretical aspects behind it are harder to comprehend.

The thesis then tries to give a basis to the theory behind the GA, where we learned about the schema theory, Building Blocks, additive decomposable functions and population sizing with the gambler's ruin model. Later we learned more about 2 additive decomposable functions, onemax, where the fitness function is given by the sum of the value of each bit and trap functions of order *k*, where the fitness function is given by the value of the concatenated traps. It was easy to understand that the onemax was a simple unimodal problem to solve and that trap functions were a little harder due to the existence of 2 optimas (global and local) and the distance between them.

The thesis then provided an analogy to visualize the onemax and the trap functions as black and white images, where the binary strings are visualized with 1 being the black color and 0 the white color. With these concepts, the program was built and the technologies used were referred.

The thesis then mentions the importance of examples in the learning process and how it helps to better and faster learning.

To conclude the thesis tries to show in paper the program and its components.

The program should be a good source for understanding how a GA works, to be used on courses or other didactic subjects on evolutionary computation.

Bibliography

[1] K. Deb, "Deceptive landscapes" in Handbook of Evolutionary Computation (T. Bäck, D. B. Fogel, and Z. Michalewicz, eds.), pp. B2.7:1-5, Bristol, New York: Institute of Physics, Publishing and Oxford University Press, 1997.

[2] D. Thierens and D. E. Goldberg, "Mixing in genetic algorithms," in Proceedings of the Fifth International Conference on Genetic Algorithms (S. Forrest, ed.), (San Mateo, CA), pp. 38-45, Morgan Kaufmann, 1993.

[3] D. E. Goldberg, K. Deb, and J. H. Clark, "Genetic algorithms, noise, and the sizing of populations," Complex Systems, vol. 6, pp. 333-362, 1992.

[4] G. Harik, E. Cantú-Paz, D. E. Goldberg, and B. L. Miller, "The gambler's ruin problem, genetic algorithms, and the sizing of populations," Evolutionary Computation, vol. 7, no. 3, pp. 231-253, 1999.

[5] K. Sastry, D. Golberg, G. Kendall, "Search Methodologies - Introduction Tutorials in Optimization and Decision Support Techniques", Chapter 4 pp. 97-126, 2005.

[6] R.K. Thompson, A. H. Wrigth, "Additively Decomposable Fitness Functions", Evolutionary Computation, IEEE Transactions (Volume: 4, Issue: 4), pp. 373-379, 2000.

[7] About W3C, http://www.w3.org/Consortium.

[8] R. Berjon, T. Leithead, E.D. Navara, E. O'Connor, S. Pfeiffer, I. Hickson, "HTML5 - A vocabulary and associated APIs for HTML and XHTML", http://www.w3.org/TR/html5, December 2012.

[9] E. J. Etemad, "Cascading Style Sheets (CSS) Snapshot 2010", http://www.w3.org/TR/css-2010/, May 2011.

[10] B. Eich, "Javascript", http://en.wikipedia.org/wiki/JavaScript.

[11] M. Pelikan, "Genetic Algorithms", Missouri Estimation of Distribution Algorithms Laboratory (MEDAL), Department of Mathematics and Computer Science, University of Missouri-St. Louis, Report No. 2010007, 2010.

[12] C. Darwin, "On the origin of species", John Murray, 1859.

[13] A. E. Eiben, J. E. Smith, "Introduction to Evolutionary Computing", Springer, 2003.

[14] J. Holland, "*Adaptation in Natural and Artificial Systems"*, The MIT Press, 1975.

[15] D. E. Goldberg, “Genetic Algorithms in Search, Optimization and Machine Learning”, Addison Wesley, 1989.

[16] W. Feller, " An Introduction to Probability Theory and its Applications (Vol. 1)", 1968.

[17] M. Abromowitz, I. Stegun, "Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables", National Bureau of Standards, 1964.

[18] R. Mayer, "Learning and Instruction (2nd edition)", Upper Saddle River, NJ: Pearson Education, Inc.

[19] Standard ECMA-262 3rd Edition - December 1999, http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf, http://www.json.org/.