Solving Maximum Number of Run Using Genetic Algorithm

Solving Maximum Number of Run Using Genetic Algorithm

BY

KELVIN CHAN, M.Sc

A Thesis

Submitted to the School of Graduate Studies

In Partial Fulfillment of the Requirements

For the Degree

Master of Science

MCMASTER UNIVERSITY

MCMASTER UNIVERSITY

MASTER OF SCIENCE (2008)	McMaster University
(COMPUTER SCIENCE)	Hamilton, Ontario

TITLE:	Solving Maximum Number of Run Using Genetic Algorithm
AUTHOR:	CHAN, KELVIN
SUPERVISOR:	DR. IVAN BRUHA

NUMBER OF PAGES: V, 53

ABSTRACT

This thesis defends the use of genetic algorithms (GA) to solve the maximum number of repetitions in a binary string. Repetitions in strings have significant uses in many different fields, whether it is data-mining, pattern-matching, data compression or computational biology [4]. Main extended the definition of repetition, he realized that in some cases output could be reduced because of overlapping repetitions, that are simply rotations of one another [10]. As a result, he designed the notion of a run to capture the maximal leftmost repetition that is extended to the right as much as possible. Franek and Smyth independently computed the same number of maximum repetition for strings of length five to 35 using an exhaustive search method. Values greater than 35 were not computed because of the exponential increase in time required. Using GAs we are able to generate string with very large, if not the maximum, number of runs for any string length. The ability to generate strings with large runs is an advantage for learning more about the characteristics of these strings.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Dr. I. Bruha, my supervisor, for all his support. This thesis could not have been completed with his help and guidance. I would also like to give special thanks to Dr. F. Franek, his help throughout the thesis was important to its success.

Hamilton, Ontario, Canada Kelvin Chan

March 2008

TABLE OF CONTENTS

List of Figures

List of Tables

CHAPTER 1: INTRODUCTION

1.1 Overview

A genetic algorithm (GA) is a robust optimization technique which simulates the natural processes of evolution. It finds near-optimum solutions to very complex problems through the exploitation of a population of points in search spaces. As such, GAs differ from other methods like exhaustive search, in that, they work with coded parameters and obey stochastic transition rules. Although a random process in appearance, the search for better performing points is based on values obtained from an objective function defining the problem. In addition to having a sound mathematical foundation, GA's appeal to natural evolution and genetics makes it particularly attractive for solving a complex optimization problem.

This thesis defends the use of genetic algorithms (GAs) to solve the maximum number of repetitions in a binary string in three parts.

In the first part, Chapter 2, briefly supplies a background and an understanding of the terminology for the genetic operators. The diversity and types of optimization problems which GAs have been handling is shown through a quick glance at the advantages of GAs. The chapter concludes with an example of a simple GA thoroughly explained.

The second part presents the problem. It starts with a rigorous definition of strings and repetition being given. The idea of 'runs' are also introduced. Using the definition of strings and the idea of runs, previously done research on the problem are shown and explained.

The last part discusses how GAs are applied to the problem showing that:

- GAs can be used to solve the problem, which means it does converge on the maximum number of runs, and

- The current conjectures for maximum number of runs are not contradicted.

The thesis concludes with a summary of accomplishments. Results are reiterated and an outlook for the future research for the algorithm and the problem area are suggested.

1.2 Exhaustive Search

For discrete problems in which no efficient solution method is known, it might be necessary to systematically enumerate all possible candidates for the solution and to check whether each candidate satisfies the problem's statement. Such a trivial but very general problem-solving technique of exhaustive examination of all possibilities is known as exhaustive search, direct search, or the "brute force" method.

Brute-force search is simple to implement and will always find a solution if it exists. However, its cost is proportional to the number of candidate solutions, which, in many practical problems, tends to grow very quickly as the size of the problem increases. Therefore, brute-force search is typically used when the problem size is limited, or when there are problem-specific heuristics that can be used to reduce the set of candidate solutions to a manageable size.

The method is also used when the simplicity of implementation is more important than speed. This is the case, for example, in critical applications where any errors in the algorithm would have very serious consequences or when using a computer to prove a mathematical theorem. Brute-force search is also useful as "baseline" method when benchmarking other algorithms.

1.2.1 Implementing the brute-force search

In order to apply brute-force search to a specific class of problems, one must implement four procedures, *first*, *next*, *valid*, and *output*. These procedures take as a parameter, the data *P* for the particular instance of the problem that is to be solved, and do the following:

1. *first* (*P*): generate a first candidate solution for *P*.
2. *next* (*P*, *c*): generate the next candidate for *P* after the current one *c*.
3. *valid* (*P*, *c*): check whether candidate *c* is a solution for *P*.
4. *output* (*P*, *c*): use the solution *c* of *P* as appropriate to the application.

Figure 1.1: Brute-Force Search Algorithm

```
c = first(P)
while (c!=null) {
        if valid(P, c) then output(P,c)
        c = next(P,c)
}
```

The *next* procedure must also tell when there are no more candidates for the instance *P*, after the current one *c*. A convenient way to do that is to return a "null candidate". Likewise the *first* procedure should return null if there are no candidates at all for the instance *P*. The brute-force method is then expressed by the algorithm in Figure 1.1.

1.3 Simulated Annealing

Simulated annealing is a natural optimization method that simulates thermal annealing which is a process aimed at obtaining the perfect crystallizations by attaining a slow enough temperature reduction to give atoms the time to attain the lowest energy state. This was first presented by Kirkpatrik et al for solving hard combinatorial optimization problems [18]. The disadvantage of this probabilistic approach is a large amount of computation time for obtaining a near-optimal solution [19].

Simulated annealing is the process in which a substance is heated above its melting temperature and then gradually cooled to produce the crystalline lattice which minimizes its energy probability distribution. This crystalline lattice, composed of millions of atoms perfectly aligned, is a beautiful example of nature finding an optimal structure. However, if the temperature is decreased too quickly, flaws in the crystal can be locked in. The slow temperature decrease allows these flaws to "work themselves out" forming a much better crystal. The key to crystal formation is carefully controlling the rate of change of temperature.

1.4 Ant Colony

Another method of natural optimization is the ant colony optimization algorithms. This algorithm was proposed by Dorigo et al that had been aspired by the foraging behavior of ant colonies [20]. This algorithm tries to imitate the natural behavior of ants in finding the shortest distance between their nests and food sources. Ants exchange information about good routes through a chemical substance called pheromone.

In nature, ants start off wandering around randomly. Upon finding food, they return to their colony while laying down pheromone trails. If other ants find such a path, they are likely not to keep traveling at random, but instead follow the trail returning and reinforcing it if they eventually find food. Over time, the pheromone trail would start to evaporate thus reducing its attractive strength. The more time it takes for an ant to travel down the path and back again, the more time the pheromones have to evaporate. A short path, by comparison gets marched over faster, and thus the pheromone density remains high as it is laid on the path as fast as it can evaporate.

Thus, when one ant finds a good path from the colony to a food source, other ants are more likely to follow that path, and positive feedback eventually leaves all the ants following a single path. The idea of the ant colony algorithm is to mimic this behavior with "simulated ants" walking around the graph which represents the problem to be solved.

CHAPTER 2: GENETIC ALGORITHM

2.1 Introduction to Genetic Algorithms

Genetic algorithms (GAs) find their roots in Darwin's principles of evolution and in simple notions of genetics. Although it has been studied as early as the 1960s, modern genetic algorithms only started in 1975 with the publication of the book *Adaptation in Natural and Artificial System* by J.H. Holland [7]. In this chapter an explanation of this natural optimization algorithm is shown thoroughly using the same format as R.L. Haupts and S.E. Haupts in their book *Practical Genetic Algorithms* [6]. We first describe the genetic algorithms and then list the useful features of GAs. Finally we demonstrate an actual GA by using an example and explaining all its members.

Genetic algorithms are an optimization and search technique. In this chapter, GAs are defined and their mechanics explained using a simple genetic algorithm in Section 2.3. The power of this optimization technique resides with its method of exploiting and exploring the search space. A GA encodes a potential solution to a specific problem on a simple chromosome-like data structure known as a *chromosome* or an *individual*. At first, a GA randomly creates a population of individuals (potential solutions) and GA operators are applied to these individuals to find the best solution or evolve to a state that maximizes the "fitness" of the individuals.

GAs use vocabulary borrowed from natural genetics. So we need a bit of biological background on heredity at the cellular level. A *gene* is a basic unit of heredity. An organism's genes are carried on one of a pair of *chromosomes* in the form of *deoxyribonucleic acid* more commonly known in the short form *DNA*. Each cell of the organism contains the same number of chromosomes. Genes often occur with two functional forms in the chromosomes, each representing a different characteristic. Each of these forms is known as *allele*. For instance, a human may carry one allele for brown eyes in one chromosome and another for blue eyes in the other chromosome. The combination of alleles on the chromosomes determines the traits of the individual. The trait observed is the *phenotype*, but the combination of alleles is the *genotype*.

We thus talk about *individuals* which are chromosomes, genotypes, strings in a population, and genes which are features.

Each individual represents a potential solution to the problem; the meaning of a particular individual is defined externally by the user. An evolution process running on a population of chromosomes corresponds to a search through a space of potential solutions.

The population undergoes a simulated evolution. At each generation the relatively "good" solutions reproduce, while the relatively "bad" solutions die. To distinguish between different solutions, we use *a fitness* or *an objective function* which plays the role of an environment [12]. Usually, the evaluation function is either called *a fitness function* if the individual with a maximum evaluation is desired, or called *a cost function* if the individuals with a minimum evaluation are desired. We can easily change a maximizing problem into a minimizing problem by simply changing the sign of the evaluation function.

2.2 The Advantage of Genetic Algorithms

Genetic algorithms are a class of general purpose, domain independent, search methods which strike a remarkable balance between exploration and exploitation of the search space. GAs can be used in image processing, numerical function optimization, combinatorial optimization, machine learning [2][3]. GAs outperform other searching optimization methods when solving very complex problem such as combinatorial optimization problems and highly constrained engineering problems. GAs belong to the class of probabilistic algorithms, yet they are very different from random algorithms as they combine elements of directed and stochastic search. Because of this, GAs are also more robust than existing directed search methods.

Some advantages of GAs are [6]:

- Optimize with continuous or discrete variables,
- Does not require derivative information,
- Simultaneously searches from a wide sampling of cost surface,

- Deals with a large number of variables,
- Is well suited for parallel computers,
- Optimized variables with extremely complex cost surfaces (they can jump out of a local minimum),
- Provides not only a single solution but a list of optimum solutions,
- May encode the variables so that the optimization is done with the encoded variables, and
- Works with numerically generated data, experimental data, or analytical functions.

GAs have been quite successfully applied to optimization problems like wire routing, scheduling, adaptive control, game playing, cognitive modeling, transportation problems, traveling salesman problems, optimal control problems, etc.

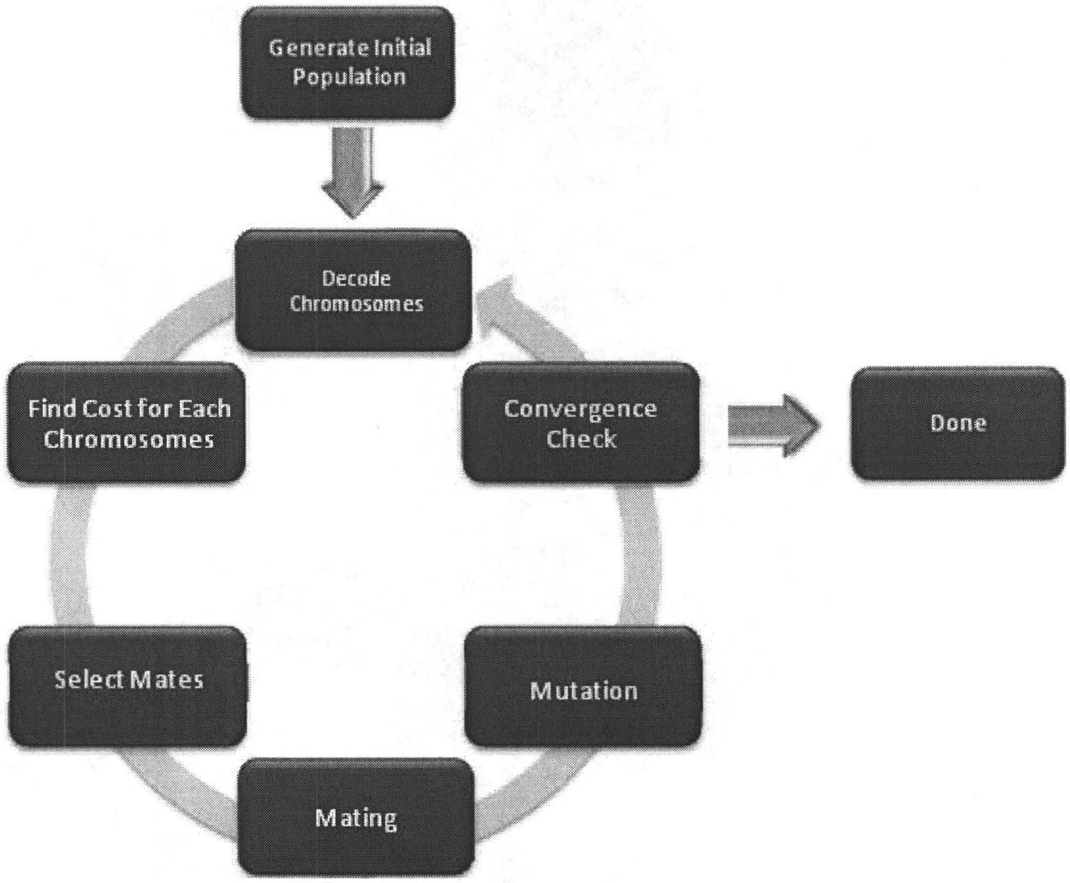
2.3 Simple Binary Example of Genetic Algorithms

A genetic algorithm (GA) for a particular problem must have the following five components:

1. A genetic representation for potential solutions to the problem,
2. A way to create an initial population of potential solutions,
3. An evaluation function that rates the solutions in terms of their "fitness",

4. Genetic operators that alter the composition of children, and
5. Values for various parameters that the genetic algorithm uses (population size, crossover rate, mutation rate, etc.)

Figure 2.1 Flowchart of a Genetic Algorithm



The process of a simple genetic algorithm is shown as the flowchart in Figure 2.1. We explain every process in this flowchart with an implementation of a simple binary genetic algorithm.

Let us use a binary genetic algorithm to design an example. Suppose we have a string of length ten that consists of 1's and 0's and we want to maximize the number of 1's in it. We would have a fitness function that is shown in Figure 2.2.

Figure 2.2 Sample Code for Fitness Function

```
int fitness(string genome) {
        for int i=1 to genome.width {
                if genome.gene(i) = 1 then
                        score = score + 1;
        }
        return score;
}
```

2.3.1 Representation

To solve this problem as a GA, we first consider the representation of the potential solution; from the definition of our problem it is clear that a ten bit binary string is a perfect representation of the solutions.

2.3.2 Evaluations Function

Since the problem wants the string to have the most 1's, we can define the evaluation function as the number of 1's in the string. The value of this function is not the value of the binary number, for this evaluation function fitness(10000000) = fitness(00000001) = 1, it does not care about the position of the 1's.

2.3.3 Initial Population

The initialization process is very simple; we create a population of chromosomes, where each chromosome is a binary vector of eight bits. All ten bits for each chromosome are initialized randomly. In this example we set the population at four, and we keep the population size fixed, during the whole process.

Our initial population is:

S_1 = 0100101101

S_2 = 1001011111

S_3 = 0101101010

S_4 = 1111100100

The fitness function fitness() evaluates them as following:

fitness(S_1) = 5

fitness(S_2) = 7

fitness(S_3) = 5

fitness(S_4) = 6

The chromosome S_2 is the best of the four chromosomes since it has the highest fitness value among them.

2.3.4 Selection

In this step we choose better individuals for reproduction, so hopefully they will produce an even better offspring. The new offspring will replace the "unfit" individuals. There are many different techniques that a genetic algorithm can use to select the individuals to be copied over to the next generation, but listed below are some of the most common methods. Some of the methods are mutually exclusive but some can be used in combination as it will be shown in our research [8].

1. Elitist Selection

   The most fitted members of each generation are guaranteed to be selected. Most GAs do not use pure elitism but instead use a modified version of elitist selection where the single or a few best are copied into the next generation just in case nothing better is generated.

2. Rank Selection

   Each individual in the population is assigned a numerical rank based on its fitness and selection is based on this ranking rather than absolute difference in fitness. The advantage of this method is that it can prevent very fit individuals from gaining dominance early at the expense of less fit ones, which would reduce the population's genetic diversity and get stuck in a local minima, which might hinder attempts to find an acceptable solution.

3. Tournament Selection

   Individuals are chosen at random from the population and are placed into competition against each other. The best individuals, the individuals with the higher fitness, from the population are chosen to be reproduced.

4. Roulette Wheel Selection

   A form of fitness-proportionate selection in which the chance of an individual's being selected is proportional to the amount by which its fitness is greater or less than its competitors' fitness. Conceptually, this can be represented as a game of roulette, each individual gets a slice of the wheel, but more fit ones get larger slices than less fit ones. The individual is chosen by whichever the wheel lands on each time.

5. Steady-State Selection

   The offspring of the individuals selected from each generation goes back into the pre-existing gene pool, replacing some of the less fit members of the previous generation. Some individuals are retained between generations, so that there is a guarantee that good potential solutions are kept in the population.

6. Parallel Migration

   A selection method with multiple, independent population. Each population evolves independently but each generation some individuals migrate from one population to another. The migration algorithm is deterministic; each population migrates a fixed number of its best individuals to its neighbor. The master population is updated after each generation with the best individuals from each population.

2.3.5 Crossover

Crossover is the creation of one or more offspring from the parents selected in the pairing process. In single point crossover, a crossover point is selected randomly between the first and the last bits of the parent's chromosomes. Table 2.1 shows the pairing and crossover process for the problem at hand, where "|" is the crossover point.

Table 2.1 Initial Population with the first generation

Chromosomes	Parent Chromosomes	Offspring
S_1	01001 | 01101	01001 | 11111
S_2	10010 | 11111	10010 | 01101
S_3	0101101 | 010	0101101 | 100
S_4	1111100 | 100	1111100 | 010

The first parent passes its binary code to the left of the crossover point to the first offspring. In the same manner, the second parent passes its binary code to the left of the same crossover point to the second offspring. Next, the binary code of the right of the crossover point of the first parent goes to the second offspring and second parent passes its code to the first offspring. Consequently the offspring contains portions of the binary codes of both parents.

2.3.6 Mutation

Mutation is performed on a bit-by-bit basis. It randomly alters the bits in the chromosome with the small probability P_m. Every bit, in all chromosomes in the whole population, has an equal probability to undergo mutation, i.e. change from 0 to 1 or vice-versa. Increasing the number of mutation increases the algorithm's freedom to explore outside the current region of variable space. We proceed in the following way.

For each chromosome in the current population and for each bit within the chromosome:

- Generate a random number r from the range [0...1]
- If $r < P_m$, mutate the bit.

Let us set $P_m = 0.1$, now the population size is 4, every chromosome has ten bits, 0.1x4x10 = 4, so four bits will be mutated. The mutated bits in Table 2.2 are shown in bold, italics.

An alternative way of mutation is to think of a whole population as a two dimensional array pop[n_{pop}, n_{bits}], where n_{pop} is the population size, n_{bits} is the length of each chromosome, in which each row represents a chromosome, and each column represents a gene. Thus a random number generator creates pairs of random integers (m_{row}, m_{col}) that correspond to the rows and columns of the mutated bits.

Table 2.2 Population after Crossover and Mutation in the second generation

Population after Mating	Population After Mutation	New Fitness Value
0 1 0 0 1 1 1 1 1 1	0 1 0 0 **0** 1 1 1 1 1	6
1 0 0 1 0 0 1 1 0 1	0 1 0 1 1 0 1 1 0 0	5
0 1 0 1 1 0 1 1 0 0	1 0 **1** 1 0 0 1 1 0 1	6
1 1 1 1 1 0 0 0 1 0	1 1 **0** 1 1 0 0 0 1 **1**	6

We can use the following computer code to find the rows and columns of the mutated bits.

n_{mut} = round (N_{pop} * N_{bits} * P_m) //number of mutations

n_{row} = round (rand(1, P_m) * N_{pop} + 1) //row of the bits to be mutated

n_{col} = round (rand(1,P_m) * N_{bits}) //column of the bits to be mutated

n_{mut} = round (N_{pop} * N_{bits} * P_m) = round (4*10*0.1) = 4

So, mutations occur three times, the following pairs were randomly selected.

n_{row} = [1 3 4 4] n_{col} = [4 3 3 10]

2.3.7 The Next Generation

After the mutation takes place, the fitness associated with the offspring and mutated chromosomes are calculated in the third column in Tables 2.2 to 2.7. The process described is iterated. For example, the population at the end of the next generation is show in Table 2.3, Table 2.4, Table 2.5, Table 2.6, and Table 2.7, respectively. We took the population in Table 2.2 as the starting population of the second generation.

Table 2.3 Population after Crossover and Mutation in the third generation

Population after Mating	Population After Mutation	New Fitness Value
0 1 0 1 1 0 1 1 0 0	0 1 0 1 1 0 1 1 0 0	5
0 1 0 0 0 1 1 1 1 1	0 1 0 0 0 1 1 1 1 1	6
1 0 1 1 0 0 1 1 0 1	1 **1** 1 1 **1** 0 1 1 0 1	8
1 1 0 1 1 0 0 0 1 1	1 1 0 1 1 0 **1** 0 **0** 1	6

Table 2.4 Population after Crossover and Mutation in the fourth generation

Population after Mating	Population After Mutation	New Fitness Value
0 1 0 1 1 0 1 1 0 0	0 1 0 1 1 0 1 1 0 **1**	6
0 1 0 0 0 1 1 1 1 1	0 1 0 **1** 0 1 1 1 **0** 1	5
1 1 0 1 1 0 1 1 0 1	1 1 0 1 1 **1** 1 1 0 1	8
1 1 1 1 1 0 1 1 0 1	1 1 1 1 1 **1** 1 1 0 1	9

Table 2.5 Population after Crossover and Mutation in the fifth generation

Population after Mating	Population After Mutation	New Fitness Value
0 1 0 1 1 1 1 1 0 1	0 1 0 1 1 **0** 1 1 **1** 1	7
0 1 0 1 0 0 1 1 0 1	0 1 0 1 0 **1** 1 1 0 1	6
1 1 0 1 1 1 1 1 0 1	**0** 1 0 1 1 1 1 1 0 1	7
1 1 1 1 1 1 1 1 0 1	1 1 1 1 1 1 1 1 0 1	9

Table 2.6 Population after Crossover and Mutation in the sixth generation

Population after Mating	Population After Mutation	New Fitness Value
0 1 0 1 0 0 1 1 1 1	0 1 0 1 0 **1** 1 1 **0** 1	6
0 1 0 1 1 1 1 1 0 1	0 1 0 1 1 1 1 1 0 1	7
1 1 1 1 1 1 1 1 0 1	1 1 1 1 1 1 1 1 0 1	9
0 1 0 1 1 1 1 1 0 1	0 1 0 1 1 **0** 1 1 **1** 1	7

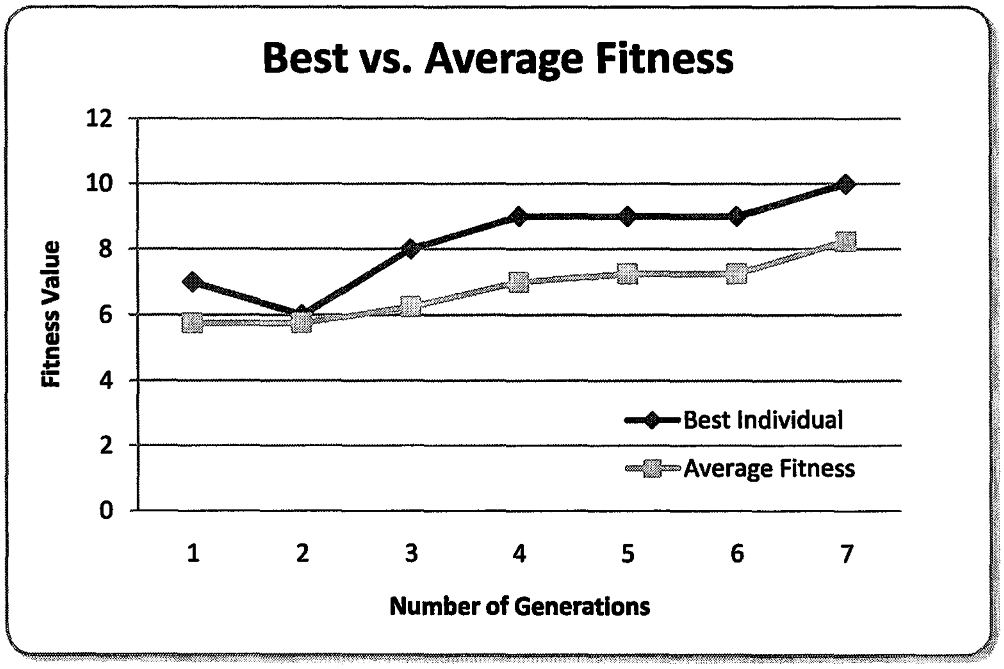
2.3.8 Convergence

The number of generations evolve depends on whether an acceptable solution is reached or a set number of iterations is exceeded. After a while, all the chromosomes and associated costs would become the same if it were not for the mutations. At this point of the algorithm it should be stopped.

Table 2.7 Population after Crossover and Mutation in the seventh generation

Population after Mating	Population After Mutation	New Fitness Value
0 1 0 1 1 1 1 1 0 1	0 1 0 1 1 1 1 1 0 1	7
0 1 0 1 0 1 1 1 0 1	0 1 0 1 **1** 1 1 1 0 1	7
0 1 0 1 1 1 1 1 0 1	**1** 1 0 1 1 1 1 1 **1** 1	9
1 1 1 1 1 0 1 1 1 1	1 1 1 1 1 **1** 1 1 1 1	10

Figure 2.3 Best vs. Average Individual



Most genetic algorithms keep track of the population statistics in the form of a population mean and minimum cost. As shown in Table 2.7, for our example after the seventh generations the global maximum fitness was found to be 10. This maximum fitness was found in

$$\underset{\substack{\text{Initial}\\ \text{Population}}}{4} \quad + \quad \underset{\substack{\text{Fitness Evaluations}\\ \text{per Generation}}}{4} \quad * \quad \underset{\substack{\text{Number of}\\ \text{Generations}}}{7} \quad = \quad 32$$

fitness functions evaluations or checking $28/(2^{10}) * 100 = 2.7\%$ of the population. Figure 2.3 shows a plot of the algorithm convergence in terms of the best and average fitness of each generation.

From Figure 2.3 we can see that at the end of the seventh generation, the algorithm is able to find the best fitness value of 10. The average fitness value is also increased from initial value of 5.75 to the final value of 8.25.

2.4 Three Evolutionary Approaches

Our experiments would use three different evolutionary approaches to demonstrate our results. First, we used crossover as a base algorithm to compare results to see if conjectures described in the previous section indeed hold true. This simple genetic algorithm uses the basic crossover technique to evolve the population of binary string. It uses non-overlapping populations, that means no string from previous population is past directly to the next population. This method was mainly used to confirm that genetic algorithms will in fact converge to generate a good solution for this problem. We also use it as a basis for comparing the other approaches that we will be testing.

The steady-state algorithm is used to test the conjecture that strings which are good will in turn create strings with better results. The steady-state compares this by keeping better percentage of the population into the next generation. If the conjecture holds then it should converge faster with better runs. Each generation the algorithm creates an entirely new population of individuals by selecting from the previous population then mating to produce the new offspring for the new population. This process continues until the algorithm's terminating criteria are met, in this case the number of generations.

Lastly, we used a parallel migrating technique with the previous steady-state algorithm. With the multiple populations the convergent should be even faster than just a normal steady-state algorithm. This approach has multiple, independent populations and was theorized to results in a faster conversion of the populations to the strings with rich runs because of its ability to do multiple evolutions every generation.

CHAPTER 3: THE PROBLEM

3.1 String

Strings are generally defined to be sequence of characters for example letters, numerals, symbols and punctuation marks. The simplest examples of strings are English words, which is a concatenation of the English alphabet [5]. Any string can be described as a sequence of elements drawn from a particular set. That set is called an *alphabet*. The members of the alphabet are referred to as the *letters*. Going back to the example of English it is evident that there are 26 letters in the alphabet. The size of the alphabet is referred to as its *cardinality*. We will be using a binary alphabet throughout this thesis to explain and demonstrate various ideas. A commonly used binary alphabet in computer science is known as a binary string. It is a combination of '0's and '1's, used to denote the two states of on and off.

Computer information storage and processed by computers is a combination for these two symbols. For instance, in a computer system, all the files, memory contents, I/O signals, all can be viewed as strings drawn from the set {0,1}. Here are some everyday examples of strings [17]:

- A text file, whose elements are ASCII characters.
- A stream of bits beamed from a space vehicle.

- A DNA sequence, composed by four letters A, C, G and T, standing for adenine, cytosine, guanine, and thiamin respectively.
- A computer program, expressed as words and separators (semicolon, colon, etc.)

An important characteristic of each string is its length, which is the number of characters in it. The length can be any natural number, zero or any positive integer. A particularly useful string for some programming applications is the empty string, which is a string containing no characters and thus having a length of zero. A substring is any contiguous sequence of characters in a string.

A string is usually expressed as one-dimensional arrays:

$$x\colon array[1 \ldots n]$$

In this case the length of string x is defined as n and denoted by $|x|$.

3.2 Representation

Repetitions in strings have significant uses in many different fields, whether it is data-mining, pattern-matching, data compression or computational biology [5]. They play an essential role in both practice and theory.

The simplest form of a repetition is a square. If there exists some integers p and i such that:

$$x[i \dots i + p - 1] = x[i + p \dots i + 2p - 1]$$

then, we say that $x[i \dots i + 2p - 1]$ is a square of period p [5].

We represent a repetition with a common notation as a triple (i, p, r) for the given string $x = x[1 \dots n]$. The positive integers i, p, and r are the position, the period and the exponent, respectively. For example given the string 101010001, the following repetitions exists (1,2,3), (2,2,2), and (6,1,3).

3.3 Runs

3.3.1 Definition of Runs

Main extended the definition of repetition, he realized that in some cases output could be reduced because of overlapping repetitions, which are simply rotation of one another [12]. As a result he designed the notion of a run to capture the maximal leftmost repetition that is extended to the right as much as possible. As an example the string 101010001, previously described to have repetitions (1,2,3) can easily infer that the repetition (2,2,2) exists. Formally a run in a string x can be represented as 4-tuple (i, p, r, m) where (i, p, r) is a repetition as defined above and moreover [18]:

- The generator $x[i \dots i+p-1]$ is not a repetition (the maximality condition);
- The initial square part of the run $x[i \dots i+p-1] = x[i+p \dots i+2p-2]$ is left maximal i.e. $x[i-1 \dots i+p-2] \neq x[i+p-1 \dots i+2p-2]$ (the non left-extensibility condition);
- r is a maximal exponent, i.e. a maximal r such that $x[i \dots i+p-1] = x[i+p \dots i+2p-1] = \cdots = x[i+(r-1)p \dots i+rp-1]$;
- $m < p$ is the tail of run, i.e. a maximal m such that $x[i+rp \dots p+rp+m]$ is a proper prefix of the generator (the non right-extensibility condition).

It can also be written as:

$$x[i \dots i+|u|r+|u'|-1] = u^r u'$$

where i is the starting position, u is the generator, $|u|$ is the period, $|u'|$ is the tail (and hence u' prefix of u), r is the exponent (often referred to as the power) [20].

As an example, the runs in the string 11010010010 is:

- Period 1: $\underline{11}01\underline{00}1\underline{00}10$: (1,1,2,0), (5,1,2,0), (8,1,2,0)
- Period 2: $1\underline{1010}010010$: (2,2,2,0)
- Period 3: $11\underline{01001001}1$: (3,3,2,2)

3.3.2 The Maximal-Number of Runs Function

Let $R(x)$ denote the number of runs in a string x, then we define the maximal-number-of-runs function $\rho(n)$ by:

$$\rho(n) = \max\{R(x): |x| = n\}, \text{ where } |x| \text{ is the length of the string } x.$$

We shall call the function $\rho(n)$, the max-run function for short. Not much is known about the max-run function, but the following properties are shown to be true by Franek and Yang [5]:

- For any n, $\rho(n+1) \geq \rho(n)$.
- For any n, $\rho(n+2) \geq \rho(n) + 1$.
- For any n, $\rho(n+1) \leq \rho(n) + \left\lfloor\frac{n}{2}\right\rfloor, where \left\lfloor\frac{n}{2}\right\rfloor\ is\ \frac{n}{2}\ rounded\ down$
- For some n, $\rho(n+1) = \rho(n)$.
- For some n, $\rho(n+1) \geq \rho(n) + 2$.

Smyth et al also presented a set of conjectures about $\rho(n)$ [18]:

- $\rho(n) < n$.
- $\rho(n-1) \leq \rho(n) \leq \rho(n-1) + 2$.
- $\rho(n)$ is attained by a cube-free binary string.

Up until now none of these simple conjectures has been proven or refuted.

3.4 Current Theories and Proofs

In the early 1980s, there were three different repetition algorithms proposed [1,9,11], all of them executed in $O(n \log n)$ in the worst case. It was shown in 2000 by Kolpakov and Kucherov that the number of runs was linear in the length of the input string [15].

Franek and Smyth independently computed the same values for $n = 5, \ldots, 35$ giving all run-maximal strings, which is shown in *Appendix A*. The exhaustive searches ended at string lengths of 35 because of the enormous computational time required for a brute-force method search.

Although computational values require linear time to calculate, there have been many who has been trying to prove the conjecture $\rho(n) < n$ theoretically. "Recently, there has been a flurry of results concerning the upper bound of $\rho(n)$: first Rytter set the upper bound of $\rho(n) to$ $5n$, which was subsequently improved by Puglisi, Simpson, and Smyth to $3.48n$ and also by Rytter himself to $3.44n$. Recently, Crochemore and Ilie pushed the upper bound down to $1.6n$, indicating that a further computer analysis can obtain an upper bound as low as $1.18n$." [5]

Furthermore, evidence exists such that for an infinite family of strings of increasing lengths [5]:

$$\lim_{n\to\infty}\left(\frac{\rho(n)}{n}\right) = \frac{3}{1+\sqrt{5}}$$

which creates a lower bound. This follows that the most recent theoretical bound is:

$$\frac{3}{1+\sqrt{5}}n < \rho(n) < 1.18n$$

In the following chapter, the use of genetic algorithms will provide a quicker method to generate strings with length greater than 35 to further support the conjectures made in the previous sections.

CHAPTER 4: THE IMPLEMENTATIONS AND EXPERIMENTAL RESULTS

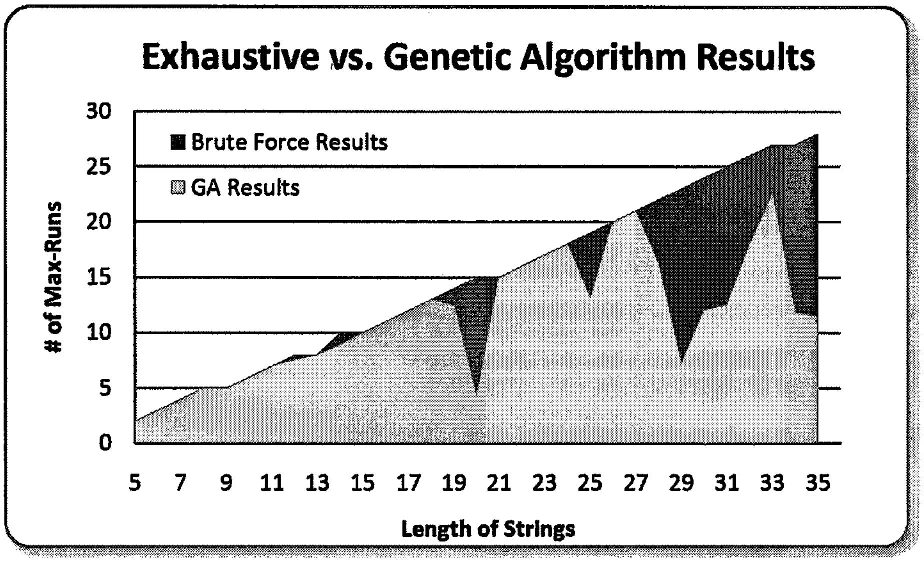
4.1 Implementation vs. Existing Information

4.1.1 Comparisons

The performance evaluation of the max-run problem is based on the algorithm's ability to find the optimal solution for a given string length. Previously, results have been calculated and verified independently by various different studies, as stated in the previous chapter. The results computed were of string length 5 to 35 using an exhaustive search method. Results were computed up to string length of 35 and creased because of the exponential growth in run-time required to calculate the different combinations of strings. We hypothesis that by using genetic algorithms to solve this problem we would be able to get a good solution because of its ability to traverse large populations of solutions while converging on a good solution, and its ability to jump out of local minima. Using the previously computed results as a base we compared our genetic algorithm.

Our first generation of genetic algorithm has been created using a simple crossover technique to observe the characteristics of its results. The population used in our research is a set of binary string, strings with only zeros and ones. We used a set number of generations as our termination condition because we wanted to confirm that the population of strings will in fact converge to a set of strings with the maximum number of runs. The fitness function used in our case is an algorithm that, obviously, calculates the number of runs in any given string.

Figure 4.1 Exhaustive vs. Genetic Algorithm Search Results



To prove that genetic algorithms do in fact generate a string with maximum number of runs, we used the above simple genetic algorithm to generate a set of solutions for the strings of length 5 to 35. Comparing those results with the results that were found from independent research previously done, we found that it does indeed generate a set of max-runs.

In Figure 4.1, we have shown the max-runs of string lengths from 5 to 35. The darker area is the results for the exhaustive search and the lighter grey area is the percentage that the genetic algorithm is able to get the correct solution. As we can see the results work well at the lower lengths because almost 100% of the time the genetic algorithm is able to find the right solution while the larger lengths slowly become worst and at string length of 35 only about half of the time does it result in the correct max-run.

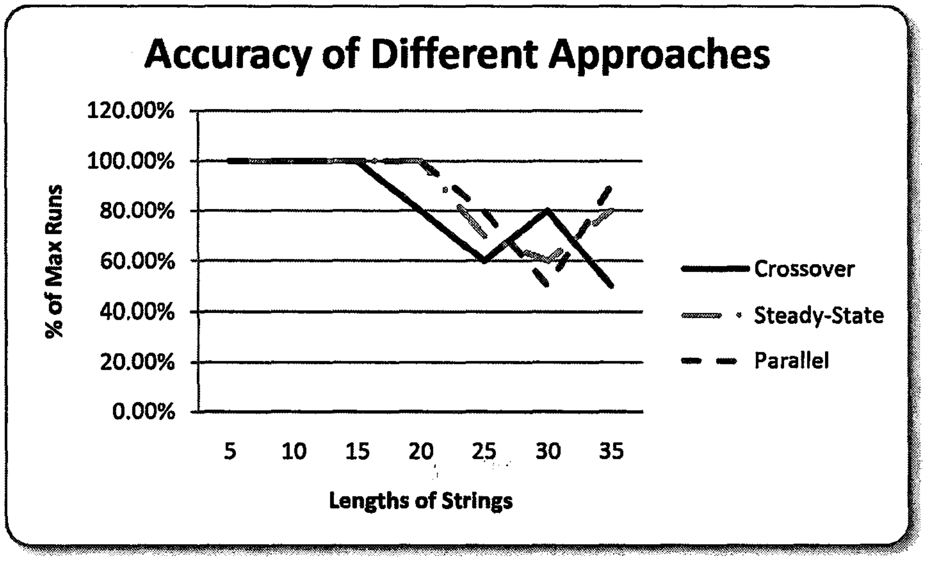
Although we have shown that the genetic algorithm actually does produce results that are comparable to using an exhaustive search method we have also shown that it does not always guarantee that the best solutions with the most runs every time will be generated. Evidence of this characteristic can be seen at the string length 20, in Figure 4.1 even though the results were good in the vicinity of that string length, the results did really poorly.

From the results in Figure 4.1 we can observe that genetic algorithms do indeed work to solve the max-run problem. It converges with certain probability. It is theorized that with different parameters much better solutions can be generated as it will be discussed in the following sections.

4.1.2 Results from Different Evolutionary Approaches

Although we used genetic algorithms to solve this problem, there are still many different combinations of evolutionary approaches as we discussed in Chapter 2. In our experiments we used three different types of approaches. The first was just the traditional crossover algorithm, the second approach we used was a steady-state algorithm in conjunction with the crossover technique and the last approach was integrating the previous technique with parallel migration.

Figure 4.2 Accuracy of the Three Different Evolutionary Approaches

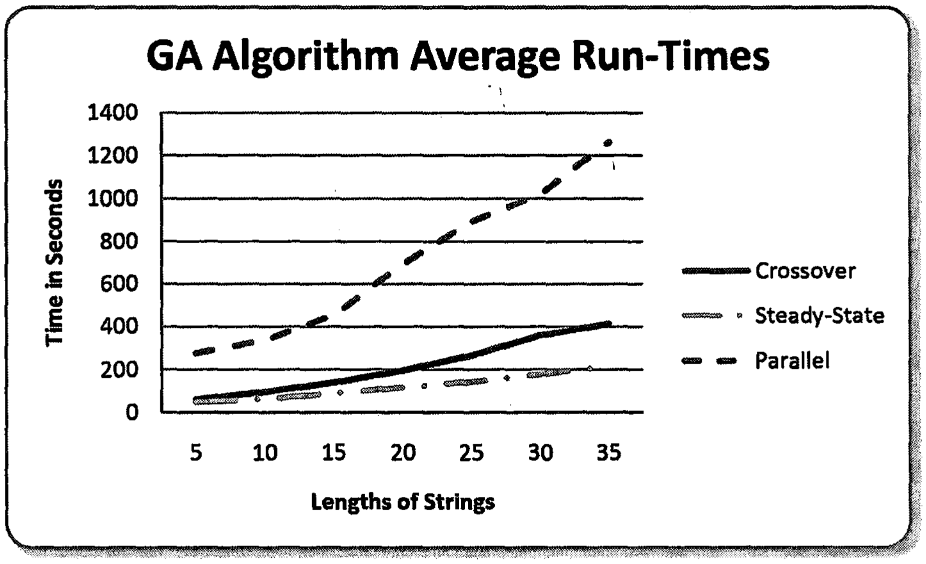


\

The three approaches have the following results shown in Figure 4.2 and 4.3, where 4.2 describes the accuracy of each type of algorithm used and 4.3 shows the run-time, which was done on a Intel® Core™ 2 CPU with 1.83 GHz and 1GB of RAM.

From Figure 4.2, we can see the solution between the three different approaches. Although the results look very similar we can first of all notice that the crossover approach starts to produce bad results much quicker than the other two. Besides that the steady-state and parallel migrating population approach have similar results. The similarity might be due to the fact that it uses the comparable steady-state algorithm.

Figure 4.3 Run-Time Comparisons of the Three Evolutionary Approaches



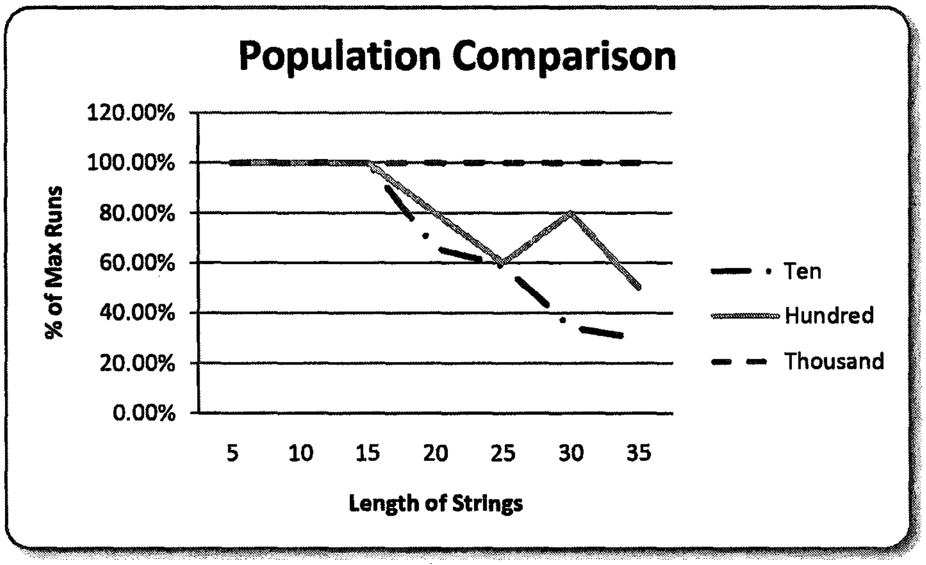
Both the steady-state and the parallel migrating population approach received similar percentages of getting the max-run but it took a lot longer to calculate one then the other as Figure 4.3 show. Although two of the three approaches seem to steadily increase in run-time as the string length increases the parallel migrating population increases in run-time exponentially faster. While genetic algorithms are much quicker than the exhaustive search, the exponential growth in the parallel algorithm suggests that just using the steady-state algorithm without parallel migration would be a preferable choice.

Just using the steady-state algorithm seems to be a good algorithm both generating a good set of solutions and in terms of run-time. In the following sections we look at other factors to further improve the chances of generating the maximum number of runs in a set length of string.

4.1.3 Population Size

For every GA, a very important part of the algorithm is the population. This represents the solution to the problem and can affect the outcome significantly. In this problem we have described that we will be using a set of binary strings. We have however not described the size of the population that we have used. The size of the population is basically the number of potential solution we have at each evolution of the algorithm. In Figure 4.4 shows the population size of ten, a hundred, and a thousand are shown. We have done a simple test to show that as in most cases when GAs are used, the bigger the population is, the better the solutions there will be. The simple reason behind this is that there are more potential solutions to the problem and there is less likelihood that the solution would be caught in local minima. As in every increase of the population there is a significant improvement to the chances of resulting in the max-run.

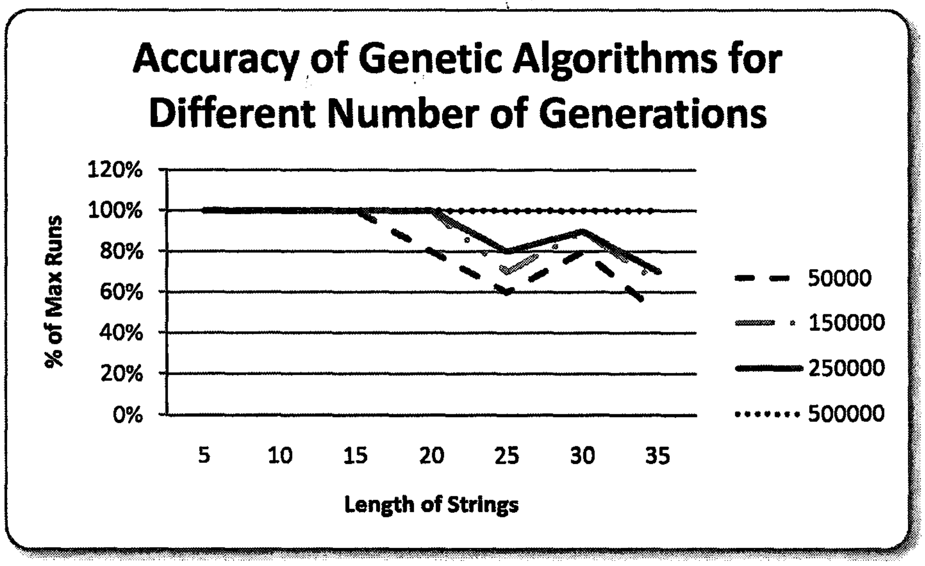
Figure 4.4 Comparing the Accuracy of Population Size of 10, 100, and 1000.



4.1.4 Termination Condition

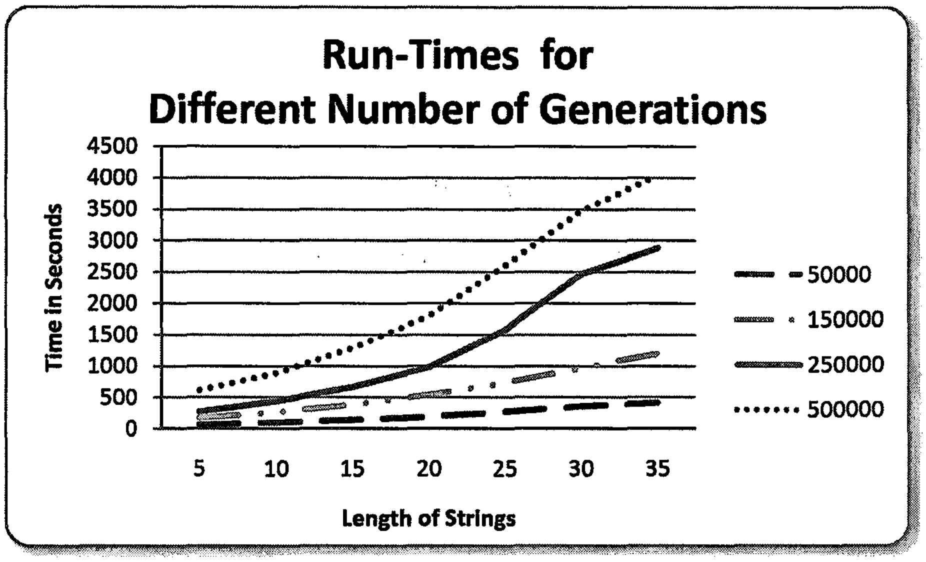
One of the most common, and necessary, variable parameter in a genetic algorithm is the terminating condition. In our research we used a set of fixed number of generations as the terminating condition, this has allowed us to observe the results and strengthen our conjectures. We compared the results to the original set of data, verified by the exhaustive search, to confirm our hypothesis that GAs do in fact converge. We know that from our initial results that a crossover GA with 50,000 generations and a population size of 100 does indeed converge to the right solutions, but does the number of generations change the results. It is commonly acknowledged since GAs are based on evolution the more generations there will be the better the results should be.

Figure 4.5 Comparing Accuracy of Genetic Algorithms for Generations of Size 50000, 150000, 250000, and 500000.



Using the same crossover GA we tried to experiment with different number of generations to see what the best results would be. From our initial observation we noticed that although GAs do not guarantee a max-run, the results do converge. The results in Figure 4.5 confirms that although not all of the population converges to the max-run as the number of generations grows the percentage of the results are significantly improved. Similar to increasing the size of the population, the more generations of evolutions there is, the more strings would be produced and the likelihood of max-run strings coming up would be better. It also shows that at some point if enough evolutions on a population is done that population would be optimal and produce a max-run.

Figure 4.6 Run-Times of Genetic Algorithms for Generations of Size 50000, 150000, 250000, and 500000.



The results prove that as the number of generations increase so does the percentage of max-run. This has caused the problem of the exponential increase in time required. Although the amount of time required at this stage is still minimal compared to doing exhaustive search, larger applications of this problem might require a lot more time.

4.2 Implementation for Unknown Values

The results from the previous section have shown that genetic algorithm can unquestionably be applied to this problem of generating strings that are rich with runs. It also has provided evidence to further support current conjectures about strings that are rich with runs.

The reason behind using GAs was the fact that exhaustive searches take too long and after string length 35 it was no longer feasible to exhaustively go through every combinations of strings. Our GAs solution provided an alternative way that will generate potential solutions for any length that is required.

Table 4.1 Initial Results.

String Length	*Best Max-Run*	*Strings With Max-Run*
50	40	11011001101100110110101101001011010110100101101011
51	41	100100110010011001000100110010011001***000***100110010011
52	42	110101101001011010110100101101001100***1000***100110010011
53	43	11010110100101101011010010100101101001010011001010010
54	45	001011010010110101101001011010110101101001011010110100
55	46	1101011010010110101101001011010010110101101001011010011
56	47	00101001011010010110101101001011010010110101101001011010
57	47	110110100101101001010010110100101101001010010110100101100
58	48	0101101001011010110100101101011010011010110100101101011010
59	50	00101101001011010110100101101011001011010110100101101011010
60	***49***	110100101101011010010110101100110110011011101100110110011011

We used the steady-state evolutionary GA, as stated before to be the best, to generate a set of solutions for string of length 50 to 60. We continued to use a population of a hundred and a hundred thousand generations as the terminating state. As stated in the previous chapters there are a few conjectures that have not been proven or refuted. The following are these conjectures that are used to examine our results.

- $\rho(n) < n$.
- $\rho(n-1) \leq \rho(n) \leq \rho(n-1) + 2$.
- $\rho(n)$ is attained by a cube-free binary string.

The results are shown in Table 4.1. From the initial results of the max-run we have seen that it does not satisfy some of the conjectures stated by Smyth et al. For example the strings at length 51 and 52 are not cube-free binary strings, they both contain a set of cubed zeros. These faults have been highlighted in the results. Also at length 60, clearly the max-run of 49 is not adequate. As it has been proven before we know that:

For any n, $\rho(n+1) \geq \rho(n)$.

Thus $\rho(60) \geq \rho(59) = 50$.

With these results not satisfying our conjectures we decided to do some further testing.

4.3 Experimental Results

Upon further investigation, we decided to increase the number of generations previously used since it has been proven to increase the accuracy of generating the max-run. So using the same algorithm as the previous genetic algorithm we altered the terminating condition to check its convergences. We changed the terminating condition such that the algorithm checks and compares the best strings in the last 50 generations and compared it to the percentage changed, if there is no change then the genetic algorithm exits with the current population being the solution. This indeed created a set of better solution, the results in Table 4.2 generated by the new algorithm does conform to all the conjectures declared by Smyth et al.

Table 4.2 The Next Generations of Results.

String Length	*Best Max-Run*	*Strings With Max-Run*
50	41	00101001011010010100101101011010010110101101001011
51	42	110101101001011010110100101101001010010110100101101
52	43	0010110100101101001010010110100101101001010010110100
53	43	11010110100101101011010010100101101001010011001010010
54	45	001011010010110101101001011010110101101001011010110100
55	46	1101011010010110101101001011010010110101101001011010011
56	47	00101001011010010110101101001011010010110101101001011010
57	48	001010010110100101001011011010010110100101001011010010100
58	49	1101001010010110100101001011010010110100101001011010010100
59	50	00101101001011010110100101101011001011010110100101101011010
60	51	110101101001011010110100101100101101001011010110100101101011

Although there is no guarantee that these newly generated strings have the best max-runs values the likelihood is very high. We believe that these results are the best values because they were generated by a genetic algorithm and it conforms to the three conjectures Smyth et al states.

Table 4.3 Results for Future Research.

String Length	*Best Max-Run*	*String Length*	*Best Max-Run*
50	41	225	184
75	61	250	203
100	84	275	227
125	102	300	244
150	125	325	264
175	146	350	284
200	166		

With the genetic algorithm configured to a reasonable setting we calculated the following set of data for 50 to 350 at an interval of 25.

Previous research has mapped out the performance of strings up to 35 but with our algorithm, further research can be done to form more concrete conjectures about runs for binary strings.

CHAPTER 5: CONCLUSION

5.1 Conclusion

Genetic algorithms are versatile and can be applied to many different problems. It's important to understand that the functioning of such an algorithm does not guarantee success. These algorithms are nevertheless extremely efficient and are especially useful for solving problems with large sample spaces. In this thesis we have applied genetic algorithms to solve an interesting problem of generating strings with large quantity of runs.

The information we have on "Max Run Strings", which are strings with the maximum number of runs for its length, are very limited. We have discussed the details of the upper and lower bound of runs. We have also looked at the conjectures on these strings which were presented by Smyth et al about $\rho(n)$:

- $\rho(n) < n$.
- $\rho(n-1) \leq \rho(n) \leq \rho(n-1) + 2$.
- $\rho(n)$ is attained by a cube-free binary string.

With the information from previously done research by Franek and Yang, we were able to verify and conclude that our genetic algorithms does solve the problem. It also has behaviors we would expect this algorithm is suppose to have. Although this method of finding "Max Run Strings" does not always produce the desired results, it does provide at least a string with large amount of runs, which narrows the sample space that we need to traverse.

This thesis demonstrates genetic algorithms have the ability to be able to solve the problem. It also provides a means to generate strings with large amount of runs previously unable to. Finally and most importantly, we were able to verify conjectures made by Smyth et al and no contradictions to previously done research were found.

5.2 Future Work

In our work we have been able to generate a set of strings that conform to the conjectures previously made. This is only the beginning of using genetic algorithm to solve problems in stringology. There are more work that can be done in both stringology and genetic algorithm. In the field of genetic algorithm, the following areas are suggested for continued research ...

- Application of genetic algorithm on large sets of combinatory strings for purposes of data-mining.

- Experimentation for using genetic algorithm to generate strings with large runs needed for test cases of critical systems

Further study in stringology using genetic algorithms can also be done to achieve...

- A better upper and lower bound on the number of runs by using better genetic algorithms to have faster convergence and better solutions.

- More conjectures and how these strings are generated by using the generated strings we have currently produced.

- A better understanding if binary strings do indeed produce the maximum amount of runs. This can be done by using a fixed length string that is non-binary with the genetic algorithm to verify that the converging solutions with large runs are binary.

Bibliography

1. Apostolico, A. and Preparata, F. P. "Optimal off-line detection of repetitions in a string." Theoretical Computer Science Volume 22, pp.297-315, 1983.

2. Branke J. "Evolutionary Approaches to Dynamic Optimization Problems - Updated Survey." In: GECCO Workshop on Evolutionary Algorithms for Dynamic Optimization Problems, pp.27-30, 2001.

3. Bruha, I. and Kralik, P. "Embedding a Genetic Algorithm in Attribute-based Rule-Inducing Learning." Soft Computing (SOCO-99), Symposium ICSC (International Computer Science Conventions, Canada), Genova, Italy, pp. 631-635, 1999.

4. Crochemore, M. "An Optimal Algorithm for Computing the Repetitions in a Word." Volume 12, Number 5, pp. 244-250, 1981.

5. Dorigo, M. and Di Caro, G. "The Ant Colony Optimization Meta-Heruistic". WSES International Conference on Evolutionary Computation (EC'01). McGraw Hill, London, pp.11-32. 1999.

6. Franek, F., Simpson, R. J. and Smyth, W.F. "The Maximum Number of Runs in a String." 2003.

7. Franek, F. and Yang, Q. "An Asymptotic Lower Bound for the Maximal Number of Runs in a String." World Scientific Publishing Company Volume 19 Issue 1, pp. 195-203, 2008.

8. Haupt, R.L., and Haupt, S.E. Practical Genetic Algorithms, Second Edition, John Wiley & Sons, Inc., 2004.

9. Holland, J.H. "Adaptation in Natural and Artificial Systems." Ann Arbor: The University of Michigan Press, 1975.

10. Main, M. G. "Detecting leftmost maximal periodicities." Discrete Applied Maths, pp. 145-153, 1989.

11. Main, M.G. and Lorentz, Richard J. "An O(n log n) Algorithm for finding all repetitions in a string." Journal of Algorithms Volume 5, pp.422-432, 1984.

12. Marczyk, Adam. Genetic Algorithms and Evolutionary Computation, http://www.talkorigins.org/faqs/genalg/genalg.html, 2004.

13. Michalewicz, X. Genetic Algorithms + Data Structures = Evolution Programs Springer Verlag, New York, Third Edition, 1996.

14. Kirkpatrick, S. C.D. Gelatt, M. P. Vecchi. "Optimization by Simulated Annealing." Science Volume 220, No.4598, pp. 671-680. 1983.

15. Kolpakov, R. and Kucherov, G. "On maximal repetitions in words." Journal of Discrete Algorithms, pp.159-186, 2000.

16. Simon, J., Luling, R. and Diekmann R. "Applied Simulated Annealing." Lecture Notes in Economics and Mathematical Systems, Volume 396. Springer-Verlag, pp.17-44. 1993.

17. Smyth, B. Computing Patters in Strings Addsison Wesley, 2003.

18. Smyth, W.F.. "Repetitive perhaps, but certainly not boring." Theoretical Computer Science Volume 249, pp.289-303, 2000.

19. Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Brute-force_search. 2008.

20. Yang, Q. "Lower and Upper Bounds for Maximum Number of Runs." 2007.

Appendix A: Maximum Number of Runs from 5 to 35

String Length	*Maximum # of Runs*
5	2
6	3
7	4
8	5
9	5
10	6
11	7
12	8
13	8
14	10
15	10
16	11
17	12
18	13
19	14
22	16
21	15
20	15
23	17
24	18
25	19
26	20
27	21
28	22
29	23
30	24
31	25
32	26
33	27
34	27
35	28