UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

FINAL PAPER num. 3529

Evolving Minimal Addition Chain Exponentiation

Marin Vlastelica Pogančić

Zagreb, June 2014.

CONTENTS

1. Introduction **1**

2. Other Exponentiation Methods **2**

2.1. Binary Method for Exponentiation 2

2.2. Right-to-left Binary Method for Exponentiation 3

2.3. The Factor Method 4

3. Addition Chain Exponentiation **5**

3.1. Addition Chain Definition 5

3.2. Differential Addition Chains 11

3.3. Addition-subtraction Chains 12

3.4. Differential Addition-subtraction Chains 13

4. Optimizing Addition Chains **14**

4.1. Evolutionary Computational Framework 14

4.2. Genetic Algorithm Optimization 15

4.2.1. Selection Operator 15

4.2.2. Individual Representation 16

4.2.3. Genetic Operators 17

4.2.4. Algorithm Application 19

4.2.5. Addition Chain Fitness Function Variations 21

4.2.6. Optimized Parameter Values 24

4.2.7. Addition Chain Optimization Results 25

4.2.8. Differential Addition Chain Optimization Results 29

4.2.9. Differential Addition-subtraction Chain Optimization Results 33

4.2.10. Addition-subtraction Chain Optimization Results 37

4.3. Genetic Annealing Optimization 41

4.3.1. Genetic Annealing Algorithm 41

4.3.2. Algorithm Application 45
4.3.3. Addition Chain Optimization Results 45
4.3.4. Differential Addition Chain Optimization Results 49
4.3.5. Differential Addition-subtraction Chain Optimization Results 53
4.3.6. Addition-subtraction Chain Optimization Results 57
4.4. Genetic Optimization Results Analysis 61
4.5. Particle Swarm Optimization 62
4.5.1. Particle Representation 62
4.5.2. Algorithm Initialization 62
4.5.3. Fitness Function 63
4.5.4. Velocity Values 63
4.5.5. Particles and Velocities Updating 63
4.5.6. Optimization Results 65

5. Conclusion **66**

Bibliography **67**

1. Introduction

The central subject of this paper is the exploring of algorithms for finding optimal methods to compute large powers. One way of looking at this is the following, let us suppose we have an expression x^n which we want to compute where n is a natural number exponent, the most basic attempt would be:

$$\begin{aligned} x_1 = x_2 = ... = x_n \\ x_1 * x_2 * ... * x_n = x^n \end{aligned} \tag{1.1}$$

Of course this isn't the optimal way of computing the exponent, the problem has been in the spotlight of scientific studies for quite a while and the benefit of its solution has a huge impact in optimizing mathematical computation to the minimal number of operations. This optimization problem brings us to the subject of addition chains and differential addition chains which have proven to be the best methods for computing large powers with a minimal number of operations, one would ask himself why would anyone want to analyze a proven method of optimal calculation. The trick is this, there is no mathematically proved method for finding optimal addition chains so we are left to a certain quantity of mathematical uncertainty. Before we immerse ourselves in analyzing the methods for finding optimal addition chains through evolutionary computation we should go through some of the methods introduced in the past and prove that they are not optimal. Also, we need to keep in mind that we are working directly with the exponent, the base of the power doesn't really interest us when searching for the minimal number of operations in calculating powers.

2. Other Exponentiation Methods

2.1. Binary Method for Exponentiation

This is probably the most intuitive method for application in computer systems mainly because it finds its grounds in binary logic. Again, suppose we have to compute the power x^n and the only available operations we have are squaring and multiplying by x. Looking at the exponent the only operations needed are shifting left (which is equivalent to multiplying by 2) and adding 1. As we can see the operations play in favor of computer processors which are able to execute them with ease. Taking all of this into consideration the solution would look something like this:

$$SSSSXSXSSS \tag{2.1}$$

Where S denotes the squaring operation of the last result and X denotes the operation of multiplying the last result by x or to put it in needed perspective, the first operation is multiplying the exponent by 2 and the second is adding 1 to the exponent. This method is quite ancient, it appeared in Pingala's Hindu classic around 200 B.C. bit a clear discussion of how to compute 2^n was given by al-Uqlidisi of Damascus in A.D. 952. which is actually equivalent to what we did here.(Knuth, 2002)

2.2. Right-to-left Binary Method for Exponentiation

This method is a variation of the standard binary method of exponentiation, lets get the idea why does the earlier method need variation. The standard binary method starts from the base x and works its way up to x^n, a more careful reader would see that this method isn't really applicable because it's extremely impractical in calculation, basically there is no way of finding the minimal number of operations other than blind search, this is where the variation comes in, instead of computing from the beginning x, we start computing from the end x^n. The algorithm itself would look like this:

A1. [Initialize] Set $N \leftarrow n$, $Y \leftarrow 1$, $Z \leftarrow x$.

A2. [Halve N] (At this point, $x^n = Y\ Z^N$.) Set $N \leftarrow \lfloor N/2 \rfloor$, and at the same time determine whether N was even or odd. If N was even, skip to step 5.

A3. [Multiply] Set $Y \leftarrow Z * Y$.

A4. [$N = 0$?] if $N = 0$, the algorithm terminates, with Y as the answer.

A5. [Square Z] Set $Z \leftarrow Z * Z$, and return to step 2.

Algorithm A is a practical method for multiplication by hand, since it involves only the simple operations of doubling, halving and adding. It is often called the "Russian peasant method" of multiplication, since the Western visitors to Russia in the nineteenth century found the method in wide use there.

The number of multiplications required by Algorithm A is:

$$\lfloor \log n \rfloor * \nu(n) \tag{2.2}$$

Where $\nu(n)$ is the number of ones in the binary representation of n.

Is it optimal? Several authors have published statements (without proof) that the binary method actually gives the minimum possible number of multiplications. But that is not true. The smallest counterexample is $n = 15$ when the binary method needs six multiplications, yet we can calculate $y = x^3$ in two multiplications and $x^{15} = y^5$ in three more, achieving the desired result with only five multiplications. (Knuth, 2002)

2.3. The Factor Method

The factor method is based on a factorization of n. If $n = pq$, where p is the smallest prime factor of n and $q > 1$, we may calculate x^n by first calculating x^p and then raising this quantity to the qth power. If n is prime, we may calculate x^{n-1} and multiply by x. And, of course, if $n = 1$, we have x^n with no calculation at all. Repeated application of these rules gives a procedure for evaluating x^n, given any value of n. For example, if we want to calculate x^5, we first evaluate $y = x^5 = x^4 * x = (x^2)^2 * x$, then we form $y^{11} = y^{10} * y = (y^2)^5 * y$. The whole process takes eight multiplications, while the binary method would have required nine. The factor method is better than the binary method on the average, but there are cases ($n = 33$ is the smallest example) where the binary method excels.

3. Addition Chain Exponentiation

3.1. Addition Chain Definition

In the earlier chapter we have discussed other methods for calculating exponentiation. Stating some simple examples we have proven that those methods are not optimal methods for exponentiation, the search has brought us to addition chains which have proven to be the most economical way to compute x^n, this is a mathematical problem with an interesting history. Although we are concerned with the calculation of powers the problem can be easily reduced to addition, since the exponents are additive. Let's begin with a formal definition of addition chains, an addition chain for n is a sequence of integers

$$1 = a_0, a_1, a_2, ..., a_r = n \tag{3.1}$$

with a property that

$$a_i = a_j + a_k \text{ for some } k \leq j < i, \tag{3.2}$$

for all $i = 1, 2, ..., r$. Best way of looking at this is if we take a look at the structure of a computer processor which has an accumulator and is capable of three operations LDA, STA and ADD. The processor begins with the number 1 in its accumulator and computes the number n by adding together previous results. For the sake of understanding the following example of how addition chains help ease exponentiation is given, for the number a^{15} :

$$a^{15} = a^3[(a^3)^2]^2 \tag{3.3}$$

The main question is, how many multiplications takes in 3.3? Lets start with $b = a^3$, we have to bring a to the power of 3 and there is no other way than to multiply a three times consecutively, that is two multiplications to start with $((a * a) * a)$. Now we need to calculate $c = (a^3)^2$ and that is only one multiplication, we obtained a^3 earlier. Now we are only left with two multiplications $b * c^2$, c^2 is one multiplication and then

multiplying with b is the other. In the end our computation of 3.3 amounts to $r = 5$ multiplications.

The shortest length r, for which there exists an addition chain for n is denoted by $l(n)$. In the rest of this section we will analyze the function $l(n)$ and go through some proven theorems that describe it. The problem of calculating the function $l(n)$ was first raised by H. Dellac in 1894 and it hasn't been solved to this day. The rest of this section is heavily based on the mathematical proofs stated in (Knuth, 2002) where it is described in more detail.

We will now define two auxiliary functions for convenience in our subsequent discussion:

$$\lambda(n) = \lfloor \log n \rfloor \tag{3.4}$$

$$\nu(n) = \text{number of 1s in the binary respresentation of n.} \tag{3.5}$$

Thus $\lambda(17) = 4, \nu(17) = 2$. In terms of these functions, the binary addition chain for n requires exactly $\lambda(n) + \nu(n) - 1$ steps, this becomes

$$l(n) \leq \lambda(n) + \nu(n) - 1. \tag{3.6}$$

Earlier we have said in the definition of addition chains that for $1 \leq i \leq r$, $a_i = a_j + a_k$ has to hold for some j and k, $0 \leq k \leq ji$. If this relation holds for more than one pair (j, k), we let j be as large as possible. Let us say that step i of the addition chain is a doubling, if $j = k = i - 1$; then a_i has the maximum possible value $2a_{i-1}$ that can follow the ascending chain $1, a_1, ..., a_{i-1}$. If j (but not necessarily k) equals $i - 1$, let us say that step i is a star step. Finally let us say that step i is a small step if $\lambda(a_i) = \lambda(a_{i-1}$. Since $a_{i-1} < a_i \leq 2a_{i-1}$ the quantity $\lambda(a_i)$ is always equal to either $\lambda(a_{i-1})$ or $\lambda a_i - 1 + 1$; it follows that, in any chain, the length r is equal to $\lambda(n)$ plus the number of small steps.

Several elementary relations hold between these types of steps: Step 1 is always a doubling. A doubling is a star step, but never a small step. A doubling must be followed by a star step. Furthermore if step i is not a small step, then step $i+1$ is either a small step or a star step, or both; putting this another way, if step $i+1$ is neither small nor star, step i must have been small.

A star chain is an addition chain that involves only star steps. This means that each term a_i is the sum of a_{i-1} and a previous a_k. A simple processor we've discussed earlier makes use only of the operations STA and ADD, LDA is not needed since each new term of the sequence utilizes the preceding result in the accumulator. The minimum length of a star chain for n is denoted by $l^*(n)$, the following holds:

$$l(n) \leq l^*(n) \tag{3.7}$$

Theorem A. If the addition chain includes d doublings and $f = r - d$ nondoublings, then

$$n \leq 2^{d-1} F_{f+2} \tag{3.8}$$

Proof. By induction on $r = d + f$, we see that 3.6 is certainly true when $r = 1$. When $r > 1$, there are three cases: If step r is a doubling, then $\frac{1}{2}n = a_{r-1} \leq 2^{d-2} F_{f+3}$; hence (14) follows. If steps r and $r-1$ are both nondoublings, then $a_{r-1} \leq 2^{d-1} F_{f+1}$; hence $n = a_r \leq a_{r-1} + a_{r-2} \leq 2^{d-1}(F_{f+2} + F_{f+1}) = 2^{d-1} F_{f+3}$ by the definition of the Fibonacci sequence. Finally, if step r is a nondoubling but step $r-1$ is a doubling, then $a_{r-2} \leq 2^{d-2} F_{f+2}$ and $n = a_r \leq a_{r-1} + a_{r-2} = 3a_{r-2}$. Now $2F_{f+3} - 3F_{f+2} = F_{f+1} - F_f \geq 0$; hence $n \leq 2^{d-1} F_{f+3}$ in all cases.

The method of proof we have used shows that the inequality 3.8 is best possible under the stated assumptions. The addition chain

$$1, 2, ..., 2^{d-1}, 2^{d-1}F_3, 2^{d-1}F_4, ..., 2^{d-1}F_{f+3} \tag{3.9}$$

has d doublings and f nondoublings.

Corollary. If the addition chain includes f nondoublings and s small steps, then

$$s \leq f \leq 3.271s. \tag{3.10}$$

Proof. Obviously $s \leq f$. We have $2^{\lambda(n)} \leq n \leq 2^{d-1} F_{f+3} \leq 2^d \phi^f = 2^{\lambda(n)+s}(\phi/2)^f$, since $d + f = \lambda(n) + s$, and since $F_{f+3} \leq 2\phi^f$ when $f \geq 0$. Hence $0 \leq s \ln 2 + f \ln(\phi/2)$, and the corollary follows from the fact that $\ln 2 / \ln(2/\phi) \approx 3.2706$

Values of $l(n)$ for special n. It is easy to show by induction that $a_i \leq 2^i$, and therefore $\log n \leq r$ in any addition chain. Hence

$$l(n) \geq \lceil \log n \rceil \tag{3.11}$$

This lower bound, together with the upper bound 3.6 given by the binary method gives us the values

$$l(2^A) = A \tag{3.12}$$

$$l(2^A + 2^B) = A + 1, \text{ if } A > B. \tag{3.13}$$

In other words, the binary method is optimum when $\nu(n) \leq 2$. With some further calculation we can extend these formulas to the case $\nu(n) = 3$:

Theorem B.

$$l(2^A + 2^B + 2^C) = A + 2, \text{ if } A > B > C. \tag{3.14}$$

Proof. All addition chains with exactly one small step have one of the following six types (where all steps indicated by "..." represent doublings):

1. Type: $1, ..., 2^A, 2^A + 2^B, ..., 2^{A+C} + 2^{B+C}; A > B \geq 0, C \geq 0.$
2. Type: $1, ..., 2^A, 2^A + 2^B, 2^{A+1} + 2^B, ..., 2^{A+C+1} + 2^{B+C}; A > B \geq 0, C \geq 0.$
3. Type: $1, ..., 2^A, 2^A + 2^{A-1}, 2^{A+1} + 2^{A-1}, 2^{A+2}, ..., 2^{A+C}; A > 0, C \geq 2.$
4. Type: $1, ..., 2^A, 2^A + 2^{A-1}, 2^{A+1} + 2^A, 2^{A+2}, ..., 2^{A+C}; A > 0, C \geq 2.$
5. Type: $1, ..., 2^A, 2^A + 2^{A-1}, ..., 2^{A+C} + 2^{A+C-1}, 2^{A+C+1} + 2^{A+C-2}, ..., 2^{A+C+D+1} + 2^{A+C+D-2}; A > 0, C > 0, D \geq 0$
6. Type: $1, ..., 2^A, 2^A + 2^B, 2^{A+1}, ..., 2^{A+C}; A > B \geq 0, C \geq 1$

By the corollary to Theorem A, there are at most three nondoublings when there is one small step; this maximum occurs only in sequences of type 3. All of the above are star chains, except type 6 when B < A - 1.

The theorem now follows from the observation that $l(2^A + 2^B + 2^C) \leq A + 2$ and $l(2^A + 2^B + 2^C)$ must be greater than $A + 1$, since none of the six possible types have $\nu(n) > 2$.

The calculation of $l(2^A + 2^B + 2^C) \leq A + 2$ when $A > B > C > D$,is more involved. By the binary method it is at most $A + 3$, and by the proof of Theorem B it is at least $A + 2$. The value $A + 2$ is possible, since we know that the binary method is not optimal when $n = 15$ or $n = 23$. The complete behavior when $\nu(n) = 4$ can be determined.

Theorem C If $\nu(n) \geq 4$ then $l(n) \geq \lambda(n)+3$, except in the following circumstances when $A > B > C > D$ and $l(2^A + 2^B + 2^C + 2^D)$ equals $A + 2$:

1. Case: $A - B = C - D$. (Example: $n = 15$)
2. Case: $A - B = C - D + 1$. (Example: $n = 23$)
3. Case: $A - B = 3$, $C - D = 1$.(Example: $n = 39$)
4. Case: $A - B = 5$, $B - C = C - D = 1$. (Example: $n = 135$)

Proof. When $l(n) = \lambda(n) + 2$, there is an addition chain for n having just two small steps; such an addition chain starts out as one of the six types in the proof of Theorem B, followed by a small step, followed by a sequence of nonsmall steps. Let us say that n is "special" if $n = 2^A + 2^B + 2^C + 2^D$ for one of the four cases listed in the theorem. We can obtain chains of the required form for each special n, therefore it remains for us to prove that no chain with exactly two small steps contains any elements with $\nu(a_i) > 4$ except when a_i is special.(Knuth, 2002)

Theorem D

$$\lim_{n\to\infty} \frac{l^*(n)}{\lambda(n)} = \lim_{n\to\infty} \frac{l(n)}{\lambda(n)} = 1 \tag{3.15}$$

Proof. The addition chain for the 2^k-ary method is a star chain if we delete the second occurrence of any element that appears twice in the chain; for if a_i is the first element among $2d_0, 4d_0, ...$ of the second line that is not present in the first line, we have $a_i \leq 2(m-1)$; hence $a_i = (m-1)+a_j$ in the first line. By totaling up the length of the chain, we have

$$\lambda(n) \leq l(n) \leq l^*(n) < (1 + 1/k) \log n + 2^k \tag{3.16}$$

for all $k \geq 1$. The theorem follows if we choose, say, $k = \lfloor \frac{1}{2} lg \lambda(n) \rfloor$.(Knuth, 2002)

Theorem E Let ϵ be a positive real number. The number of addition chains such that

$$\lambda(n) = m, r \leq m + (1 - \epsilon)m/\lambda(m) \tag{3.17}$$

is less than α^m, for some $\alpha < 2$, for suitably large m. (In other words, the number of addition chains so short that 3.17 is satisfied is substantially less than the number of values of n such that $\lambda(n) = m$, when m is large).

Proof. We want to estimate the number of possible addition chains, and for this purpose our first goal is to get an improvement of Theorem A that enables us to deal more satisfactory with nondoublings, because the proof of the theorem is rather long for the more interested readers it can be found in (Knuth, 2002).

Corollary. The value of $l(n)$ is asymptotically $\lambda(n) + \lambda(n)/\lambda\lambda(n)$, for almost all n. More precisely, there is a function $f(n)$ such that $f(n) \to 0$ as $n \to \infty$, and

$$Pr(|l(n) - \lambda(n) - \lambda(n)/\lambda\lambda(n)| \geq f(n)\lambda(n)/\lambda\lambda(n))) = 0. \tag{3.18}$$

where Pr is the probability function for the occurrence of that event.(Knuth, 2002)

***Star chains.** Optimistic people find it reasonable to suppose that $l(n) = l^*(n)$; given an addition chain of minimal length $l(n)$, it appears hard to believe that we cannot find one of the same length that satisfies the (apparently mild) star condition. But in 1958 Walter Hansen proved the remarkable theorem that, for certain large values of n, the value of $l(n)$ is definitely less than $l^*(n)$, and he also proved several related theorems but for which we shall not go in deeper in this paper.(Knuth, 2002)

To sum up this section has gone through some math describing addition chains with which it can be concluded that this problem was and is pretty fascinating still which hopefully gives some motivation for an attempt to find the optimal addition chains for exponentiation.

3.2. Differential Addition Chains

In the earlier section we have explained what are addition chains and how they benefit the process of calculating large exponents. It is obligatory that we also mention differential addition chains.

Differential addition chains (also known as strong addition chains, Lucas chains, Chebyshev chains) are a special category of addition chains in which each sum is already accompanied by a difference or to put it delicately whenever a new chain element is formed by adding P and Q the difference $P - Q$ was already in the chain. An example of a differential addition chain would be the following:

$$0 \to 1 \to 2 \to 3 \to 4 \to 7 \to 11 \to 18 \to 29 \to 40 \to 51 \to 91 \tag{3.19}$$

of course it is clear that we have to observe the chain starting with 0 and working our way up because the numbers 1 and 2 have to be in the addition chain, to satisfy the differential addition chain rule when we add $1 + 1$ to get 2 the difference $1 - 1 = 0$ was already in the the chain. If we analyze this addition chain it is clear that all the elements satisfy that rule.(J.Bernstein, 2006)

1. $2 = 1 + 1, 1 - 1 = 0$
2. $3 = 2 + 1, 2 - 1 = 1$
3. $4 = 3 + 1, 3 - 1 = 2$
4. $7 = 4 + 3, 4 - 3 = 1$
5. $11 = 7 + 4, 7 - 4 = 3$
6. $18 = 11 + 7, 11 - 7 = 4$
7. $29 = 18 + 11, 18 - 11 = 7$
8. $40 = 29 + 11, 29 - 11 = 18$
9. $51 = 40 + 11, 40 - 11 = 29$
10. $91 = 51 + 40, 51 - 40 = 11$

3.3. Addition-subtraction Chains

Addition-subtraction chains are not strictly addition chains, we allow them more "freedom" so to speak. The definition of addition-subtraction chains is as follows: A sequence of numbers $a_0, a_1 ... a_n$ that satisfies

$$a_0 = 1, \text{ for } k > 0, \ a_k = a_i \pm a_j, \text{ for some } 0 \leq i, j < k \tag{3.20}$$

Logically, there are more addition-subtraction chains for one exponent that addition chains. Best to demonstrate addition-subtraction chains with the following example of the exponent 31:

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 31 \tag{3.21}$$

This addition-subtraction enables us to get to the exponent 32 in 4 additions and 2 subtractions which is shorter than the optimal addition chain for the exponent of length 7. Here is a more thorough analysis of the addition-subtraction chain in steps:

1. $2 = 1 + 1$
2. $4 = 2 + 2$
3. $8 = 4 + 4$
4. $16 = 8 + 8$
5. $32 = 16 + 16$
6. $31 = 32 - 1$

So we have come to the result where addition-subtraction chains seem to be a better solution for exponentiation that addition chains, but is that so? The problem with addition-subtraction chains is the division, which is more costly to a computer processor than multiplication. For this reason addition-subtraction chains are a rather less interesting method of exponentiation than addition chains.

3.4. Differential Addition-subtraction Chains

Earlier we have mentioned addition-subtraction chains as an alternative to exponentiation to addition chains so it is only fair to define what are differential addition-subtraction chains just for everything to be concise and not to confuse the reader.

We said that the characteristic of differential addition chains is that every addition is preceded by the subtraction of the addition members. Lets put that into context with the addition subtraction chain rule 3.20, in addition-subtraction chains we have addition and subtraction, so what happens when we subtract? The twist is that in differential addition-subtraction chains every addition is preceded by the subtraction result and every subtraction is preceded by the addition result. To put it simply there are two cases:

1. If we obtain a new member by addition: $P + P$, $P > Q$ then the subtraction result of $P - Q$ must already be in the chain.

2. If we obtain a new member by subtraction $P - Q$, $P > Q$ then the addition result of $P + Q$ must already be in the chain.

No examples are needed for this section because the understanding is assumed from the previous sections on differential addition chains and addition-subtraction chains.

4. Optimizing Addition Chains

In this section we shall try to find optimal addition chains using some methods in evolutionary computation, we will go through the results and analyze the difference in the process of generating the solutions in different algorithm settings, but firstly before digging in deeper in optimizing addition chains we should define the algorithms we are going to use in the process and the framework in which the algorithms will be implemented.

4.1. Evolutionary Computational Framework

It would be extremely impractical to write the algorithms from scratch in this case because this isn't the central theme of the paper, in this section describes a few things about the framework that was used to rather concentrate on the subject of addition chains and their analysis and not the very construction of the algorithms.

ECF (Evolutionary Computational Framework) is a framework for evolutionary computation designed on the Faculty of Electrical Engineering and Computing. The frameworks base programming language is C++, this enables it to excel in performance which is important in the case of evolutionary computation to produce the best possible results in the shortest period of time. It supports a various number of evolutionary algorithms and their variations, the process of optimization is reduced to defining an input file with the algorithm parameters and defining a fitness function that will guide the algorithms evaluation of an individual i.e. a solution. If a user wishes to speed up the process of optimization ECF also supports parallelization.

4.2. Genetic Algorithm Optimization

We shall embark on our journey of optimizing addition chains with maybe the most popular algorithm in evolutionary computation which was invented by John Holland (Čupić, 2013), the genetic algorithm. This algorithm draws its method from the very theory of evolution established by Charles Darwin. The basic idea is that we have a population of individuals that represent our solution which we advance through generations with mutation and crossover.

As it was mentioned mentioned earlier the genetic algorithm has its roots in evolution theory and is strongly dependent on the operators of mutation, crossover and selection which will be described in this section. Also, it is important to understand that there are many possibilities for the representation of an individual which will also be analyzed.

4.2.1. Selection Operator

The first operator that needs to be considered is the selection operator, as it was mentioned earlier we have a generation of individuals that has to be evaluated, after the evaluation the algorithm needs to make a selection of individuals that will progress to the next generation and participate in reproduction (the creation of new individuals), this is very important because we want to preserve the good individuals and their genetic information. The rest of this section will go through some methods of selection that are extensively used in the genetic algorithm more thoroughly explained in (Čupić, 2013).

Roulette Wheel Selection. The genetic algorithm that uses this kind of selection is called a generational genetic algorithm. This kind of selection generates a new population from the old one that has the same number of individuals as the previous population. The algorithm of roulette wheel selection can be simply states as follows:

Algorithm 1 Roulette wheel selection

1. Sum Calculate sum of all chromosome fitness values in the population - sum S

2. Select Generate random number from interval $(0, S)$ - r

3. Loop Go through the population and sum fitness values from 0 - sum s, When sum s is greater than r, stop and return the chromosome where the loop stopped.

Tournament selection. This kind of selection takes k individuals from the population, compares them and copies the best individual in the reproduction pool over which the genetic algorithms will take action in the next step after which we produce a new individual for the next generation. This is repeated till we generate a population that is the same size as the previous population

Steady-state selection. The central objective of this selection is to choose the bad solutions and replace them with new ones using the genetic operators. Firstly the bad solutions are erased from the population, after that we apply the genetic operators on the remaining good solutions. Because of this process there is a danger of producing solutions that are the same as previous solution so that should also be taken care of accordingly.

Elitism. There is a danger of losing a good solution after many iterations if the genetic operators deform it, therefore there is a need for a mechanism of protection for the best individuals, that mechanism is called elitism. As a result, the genetic algorithm with built-in elitism progresses asymptotically toward the global optimum.

4.2.2. Individual Representation

The representation of an individual is of great importance for the algorithm, it drives the way the fitness function will be constructed. For the purpose of this paper we will concentrate on a bit string representation of an individual, specifically for the genetic algorithm we refer to an individual as a *chromosome*, the reason for this is the very nature of the genetic algorithm, the chromosome consists of pieces of information we call *genes* which are the carriers of information in biology. The bit string representation is the simplest and the most used representation, it's also the most intuitive which we will see when we analyze the crossover and the mutation operators. It also needs to be mentioned that there are other representations of individuals such as the binary code representation or gray code representation or variations that will be left to a more interested readers research.

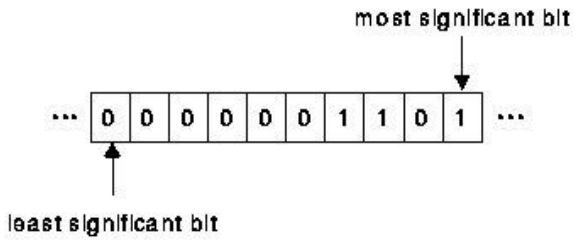

Figure 4.1: Bit string example

4.2.3. Genetic Operators

The genetic operators play a central role in producing new, interesting solutions in the population, through them we are grabbing solutions from the search space. The genetic operators in the genetic algorithm are *mutation* and *crossover*.

Mutation

Mutation is a genetic operator which causes the change of one or multiple genes, the result of the mutation is an altered individual.

The parameter that determines the probability of mutation p_m of one bit is a parameter of the algorithm. If the probability of the mutation converges to 1 then the algorithm becomes a random search algorithm. On the other hand, if it converges to 0 then the process will probably be stuck in a local optimum.(Golub, 2004)

The kind of mutation used in this paper is called *simple mutation* where every bit has the same probability of mutation p_m but of course there are different kinds of variations on the mutation operator.

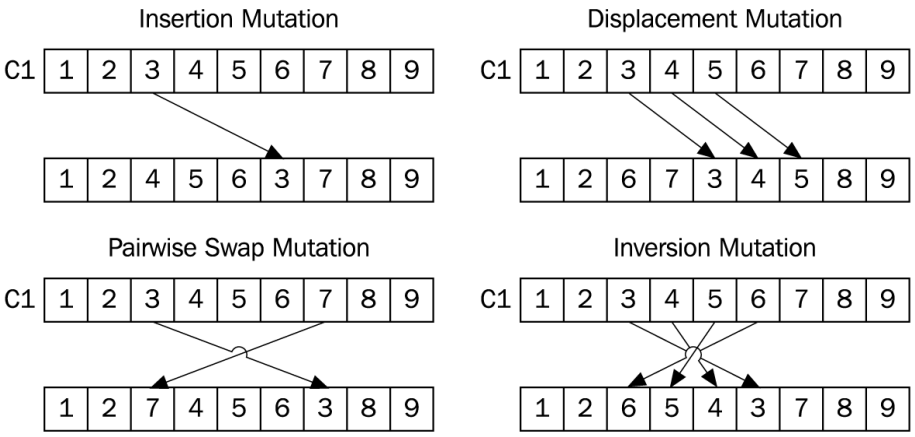

Figure 4.2: Mutation examples

Crossover

In the crossover process there are two important players, two individuals which are called *parents*. After crossover is applied one or more new individuals are generated which are called *children*. From the name of the actors in the operator it is logical to conclude that the whole point is that the children are created on the base of the parents genes, this way the children may represent better solutions than the parents.(Golub, 2004)

There are many different types of crossover but we will concentrate on the main one that will be used in this paper, *uniform crossover*. The uniform crossover operation can be easily represented with one formula. Let us say that A and B are parents and let R be a chromosome that is generated randomly, this would be the uniform crossover:

$$CHILD = AB + R(A \oplus B) \tag{4.1}$$

This crossover operator is also the fastest because of its use of logical operators which are directly supported by the computer architecture.

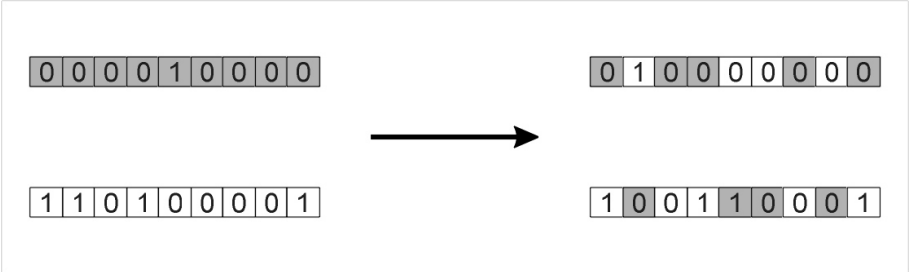

Figure 4.3: Example of crossover

Algorithm Parameters

Now that all the actors in the genetic algorithm have been analyzed it is obvious that the algorithm parameters have to be defined before running the algorithm on a optimization problem. Those parameters are mutation probability, size of the population, number of generations and the probability of crossover. The probability of crossover is typical for generational genetic algorithms, if we take a steady-state algorithm then it is redundant and is replaced by the number of individuals that will be eliminated.

Table 4.1: Genetic algorithm parameters example (Golub, 2004)

Parameter	Sign	Small population value	Large population value
Population size	PS	30	100
Crossover probability	p_c	0.9	0.6
Mutation Probability	p_m	0.01	0.001

4.2.4. Algorithm Application

Now that the mechanisms behind the genetic algorithm are explained the concrete solution for the problem of addition chains can be analyzed. Firstly we will begin with the explanation of the encoding that is being used, the fitness function and then we will analyze the results that are obtained when applying the genetic algorithm.

Individual Encoding

The encoding for our individual in the case of addition chains is as suggested in (Nedjah and de Macedo Mourelle). The basic idea is to store the addition chain as a bit string. Each position in the bit string represents a number that occurs in the addition chain or does not, depending on the value of the position (1 being for occurs in the chain and 0 being for doesn't occur).

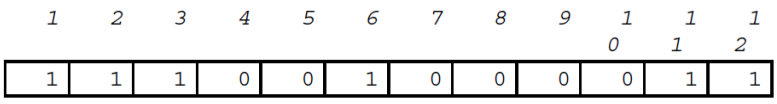

Figure 4.4: Addition chain encoding

It is obvious that this representation isn't memory efficient, suppose that we have to calculate large exponents like it is needed in real life application, every number from

1 to the exponent has to be represented by a position in the bit string so we end up with a bit string which is the same length as the exponent size. Disregarding memory-wise inefficiency it is also time consuming which will be seen when the fitness function is defined.

Fitness Function

The fitness function that will be used is fairly straightforward and follows the formal definition of addition chains and it is as follows:

Algorithm 2 Addition Chain Fitness Function

Input: bit string S
$fitness = 0$
for $i = 2; i < sizeof(S); i++$ **do**
  if $S[i]$ is set **then**
    $fitness+ = 1$
    for $j = 1; j \leq i; j++$ **do**
      for $k = 1; k \leq j; k++$ **do**
        if $k + j = i$ and $S[k]$ is set and $S[j]$ is set **then**
          The solution is valid
        end if
      end for
    end for
    if The solution isn't valid **then**
      $fitness+ = largePenalty$
    end if
  end if
end for

Basically the algorithm tells us that for every number in a supposed addition chain we have to loop through the previous numbers in the chain and test if there exists a pair of numbers that satisfy the addition chain definition i.e. there must be a k and a j for which $i = j + k + 1$, the one is there because the indexes start from 0 and our first number in the chain is 1. The extra looping can be avoided by breaking out of the loop which is checking if the position is valid when a specimen that confirms that is found.

4.2.5. Addition Chain Fitness Function Variations

If we want to compute the minimal differential addition chain the variation is fairly simple and it amounts to adding one more condition while evaluating the bit string individual in the fitness function.

Algorithm 3 Differential Addition Chain Fitness Function

Input: bit string S
$fitness = 0$
for $i = 2; i < sizeof(S); i++$ **do**
  if $S[i]$ is set **then**
    $fitness+ = 1$
    for $j = 1; j \leq i; j++$ **do**
      for $k = 1; k \leq j; k++$ **do**
        if $k + j = i$ and $S[k]$ is set and $S[j]$ is set and $S[j - k]$ is set **then**
          The solution is valid
        end if
      end for
    end for
    if The solution isn't valid **then**
      $fitness+ = largePenalty$
    end if
  end if
end for

As it can be seen the alteration is minimal, we simply add the test if the subtraction result of i and j exists in the if condition. Such alterations can be done for addition-subtraction chains and differential addition-subtraction chains as we can see in the algorithms 4 and 5.

Algorithm 4 Addition-subtraction Chain Fitness Function

Input: bit string S
$fitness = 0$
for $i = 2; i < sizeof(S); i++$ **do**
  if $S[i]$ is set **then**
    $fitness+ = 1$
    for $j = 1; j \leq i; j++$ **do**
      for $k = 1; k \leq i; k++$ **do**
        if ($k + j = i$ or $j - k = i$ and $S[k]$ is set and $S[j]$ is set **then**
          The solution is valid
        end if
      end for
    end for
    if The solution isn't valid **then**
      $fitness+ = largePenalty$
    end if
  end if
end for

Algorithm 5 Differential Addition-subtraction Chain Fitness Function

Input: bit string S
$fitness = 0$
for $i = 2; i < sizeof(S); i++$ **do**
  if $S[i]$ is set **then**
    $fitness+ = 1$
    for $j = 1; j \leq i; j++$ **do**
      for $k = 1; k \leq i; k++$ **do**
        if ($k + j = i$ and $S[j - k]$ is set or $j - k = i$ and $S[j + k]$ is set) and $S[k]$ is set and $S[j]$ is set **then**
          The solution is valid
        end if
      end for
    end for
    if The solution isn't valid **then**
      $fitness+ = largePenalty$
    end if
  end if
end for

The algorithms used for addition-subtraction were suggested in (Nedjah and de Macedo Mourelle), the question is are they efficient, this will be checked in the following sections.

4.2.6. Optimized Parameter Values

As it was said earlier, the parameters of the genetic algorithm are an important component in order to optimize the optimization process so to speak, the parameters set in this case were obtained through the optimization method specified in (Tomola-Fabro, 2013). The parameters that were optimized are mutation probability and population size, the optimal parameters obtained through this optimization method based on the exponents computed are:

Table 4.2: Fitness values for the optimization (Golub, 2004)

Exponent	Mutation probability	Population size
23	208-223	0.625-0.6875
55	208-223	0.4375-0.5
130	255-269	0.9375-1
250	294-300	0.625-0.75
768	121-128	0.78125-0.875

Because of the limited computational resources the experiments will be limited to 500 generations with a terminal stagnation of 100, which means that the experiment will end if there is no improvement over a period of 100 generations. All of that considered, for each exponent there will be 30 runs of one experiment setting and the results are obtained from the best run.

4.2.7. Addition Chain Optimization Results

Table 4.3: Fitness values for the optimization

Exponent	Minimal value	Max value	Average value	Standard deviation
23	6	54	7.35909	6.66985
55	8	65	13.2045	16.3243
130	9	142	17.6692	32.6997
250	12	516	34.71	75.112
768	18	788	89.0923	223.741

Table 4.4: The optimal chains obtained through optimization

Exponent	Chain	Length
23	1->2->3->5->10->13->23	7
55	1->2->3->5->10->11->22->33->55	9
130	1->2->4->6->10->20->30->50->80->130	10
250	1->2->4->8->10->18->19->26->45->53->106->125->250	14
768	1->2->3->5->10->11->22->32->54->65->68->130-> 136->168->204->298->434->564->768	19

Convergence Graphs

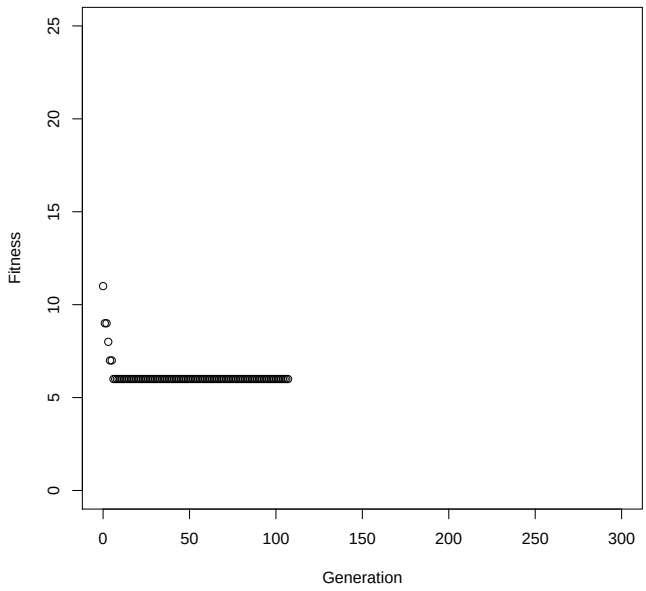

Figure 4.5: Algorithm convergence for exponent 23

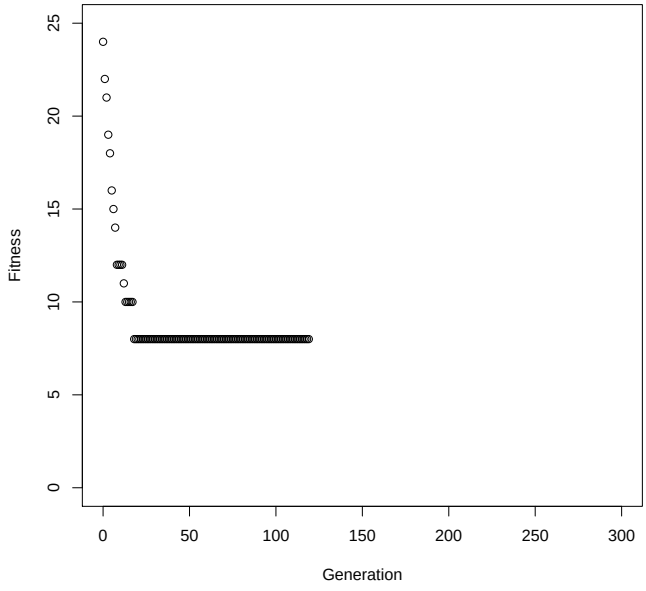

Figure 4.6: Algorithm convergence for exponent 55

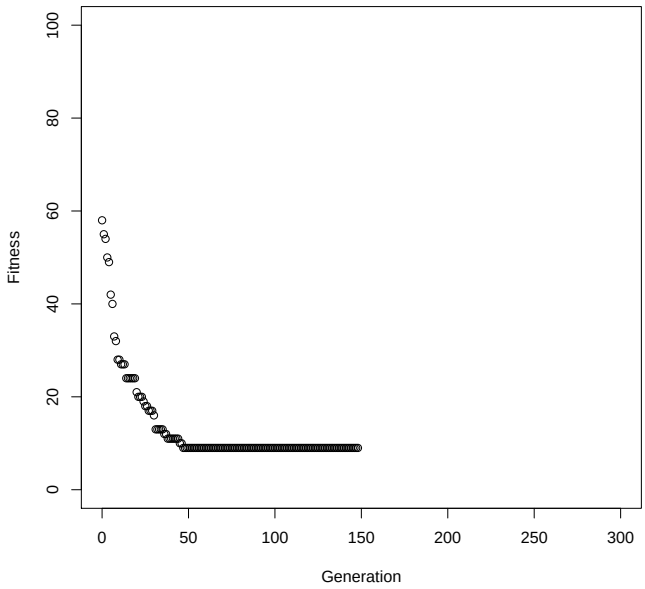

Figure 4.7: Algorithm convergence for exponent 130

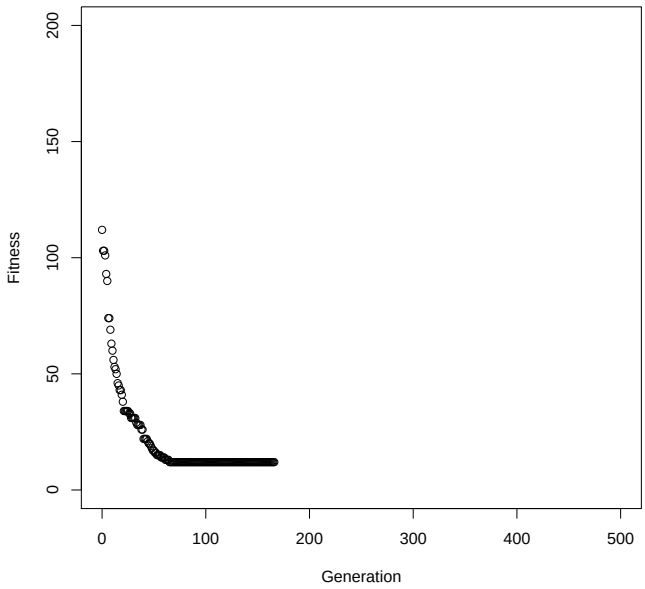

Figure 4.8: Algorithm convergence for exponent 250

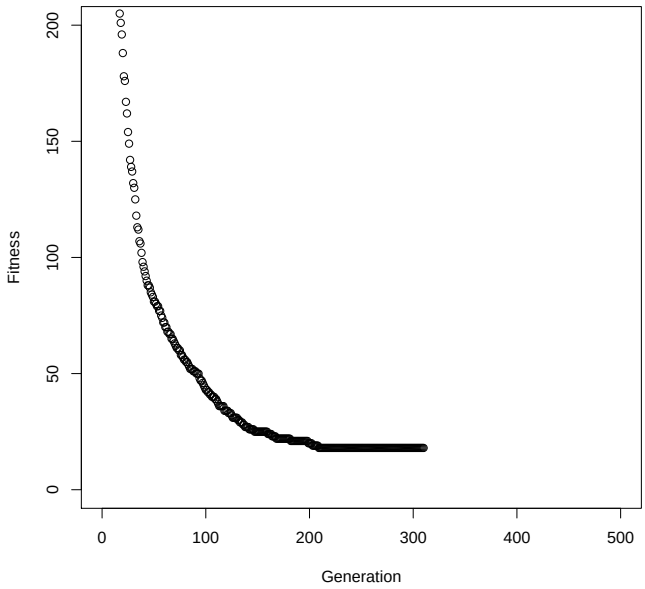

Figure 4.9: Algorithm convergence for exponent 768

4.2.8. Differential Addition Chain Optimization Results

Table 4.5: Fitness values for the optimization

Exponent	Minimal value	Max value	Average value	Standard deviation
23	7	102	9.54091	12.3138
55	8	175	15.4773	21.9311
130	9	270	19.6423	37.7292
250	15	267	39.37	74.5881
768	23	793	105.938	239.611

Table 4.6: The optimal chains obtained through optimization

Exponent	Chain	Length
23	0->1->2->3->4->5->9->14->23	9
55	0->1->2->3->5->6->11->22->33->55	10
130	0->1->2->3->5->10->15->25->40->65->130	11
250	0->1->2->3->5->8->11->13->14->15->25->30->35->55->70->125->250	17
768	0->1->2->3->5->6->7->11->13->16->17->19->22->34-> 38->41->54->75->76->108->109->184->292->476->768	24

Convergence Graphs

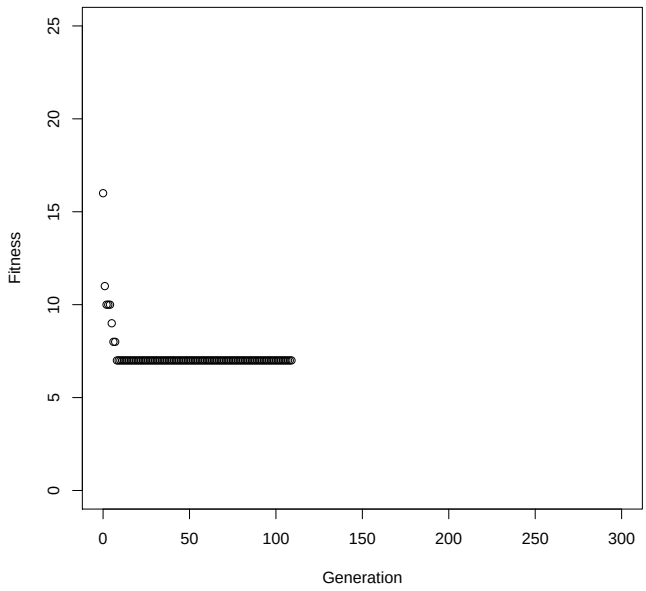

Figure 4.10: Algorithm convergence for exponent 23

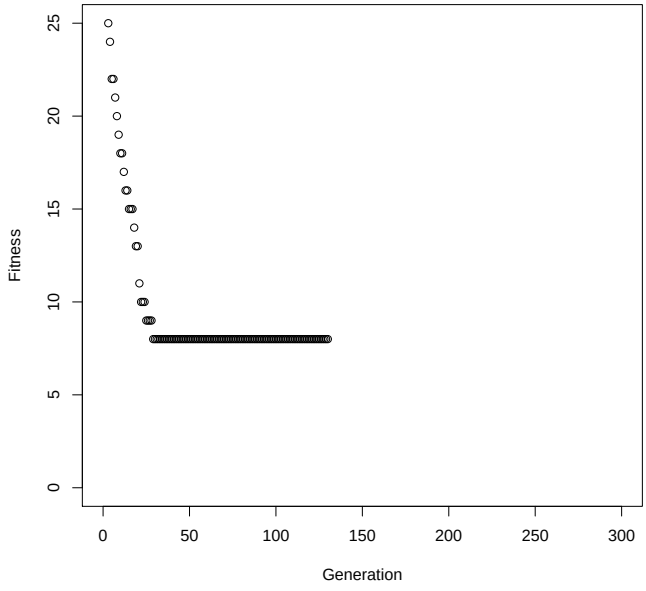

Figure 4.11: Algorithm convergence for exponent 55

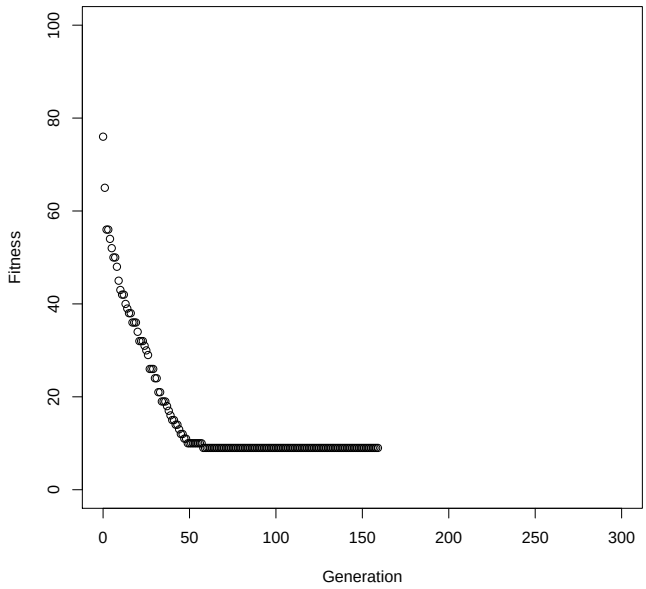

Figure 4.12: Algorithm convergence for exponent 130

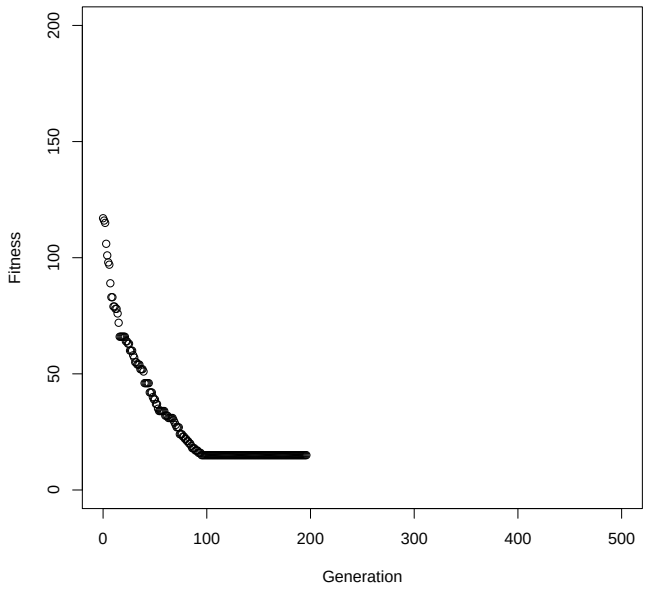

Figure 4.13: Algorithm convergence for exponent 250

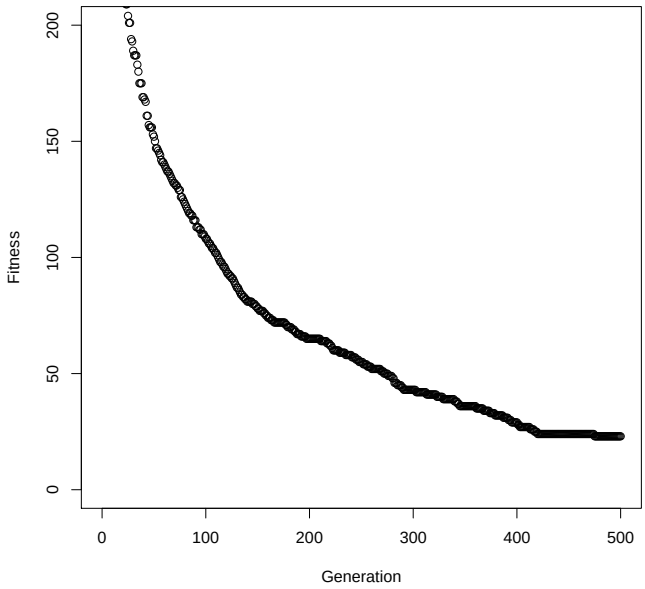

Figure 4.14: Algorithm convergence for exponent 768

4.2.9. Differential Addition-subtraction Chain Optimization Results

Table 4.7: Fitness values for the optimization

Exponent	Minimal value	Max value	Average value	Standard deviation
23	7	127	8.71818	10.2775
55	8	176	14.4409	20.9075
130	10	142	24.7269	41.6079
250	12	1015	39.69	104.155
768	26	796	97.0692	223.695

Table 4.8: The optimal chains obtained through optimization

Exponent	Chain	Length
23	0->1->2->4->5->10->15->25->30->55	10
55	0->1->2->3->5->10->15->25->30->55	10
130	0->1->2->3->4->7->11->18->29->47->65->130	12
250	0->1->2->4->6->10->16->22->38->44->54->98->152->250	14
768	0->1->2->3->5->7->8->11->12->14->15->17->20->28->29->40 ->43->45->47->52->54->59->87->92->146->238->384->768	28

Convergence Graphs

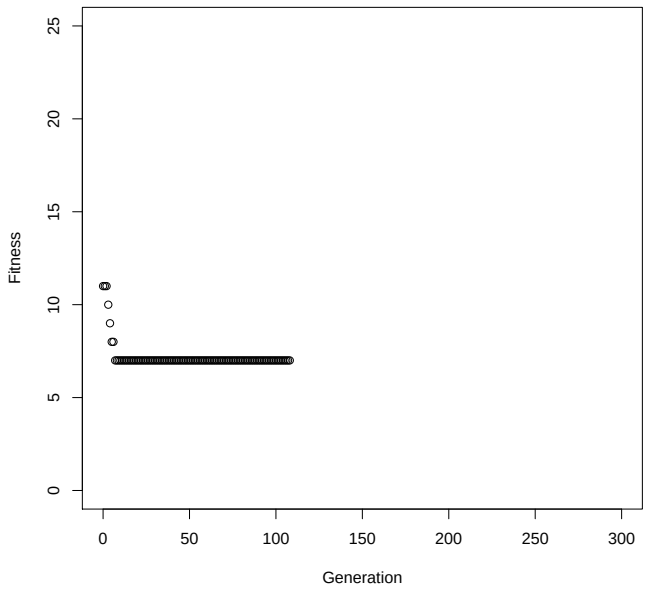

Figure 4.15: Algorithm convergence for exponent 23

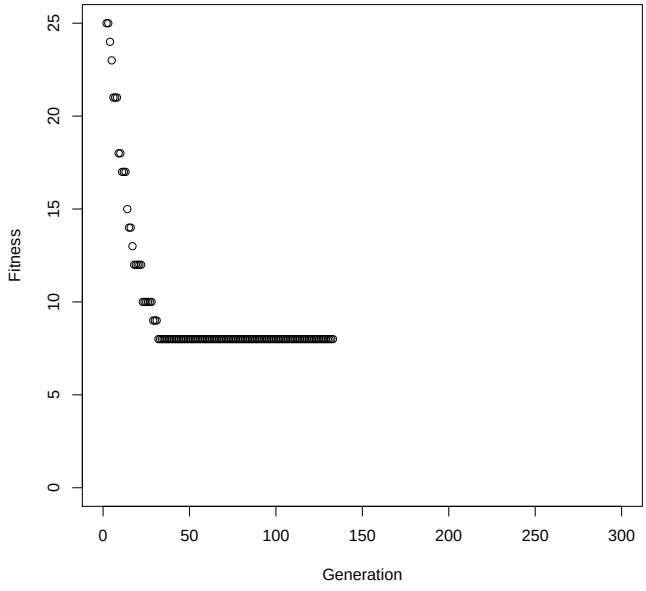

Figure 4.16: Algorithm convergence for exponent 55

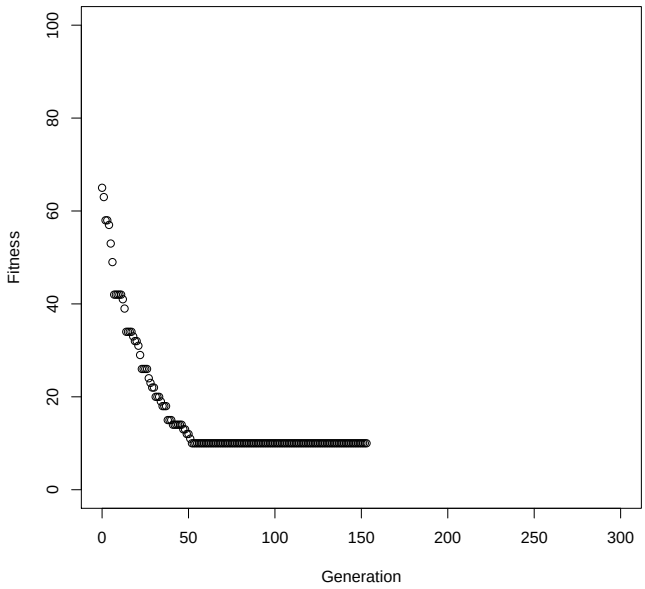

Figure 4.17: Algorithm convergence for exponent 130

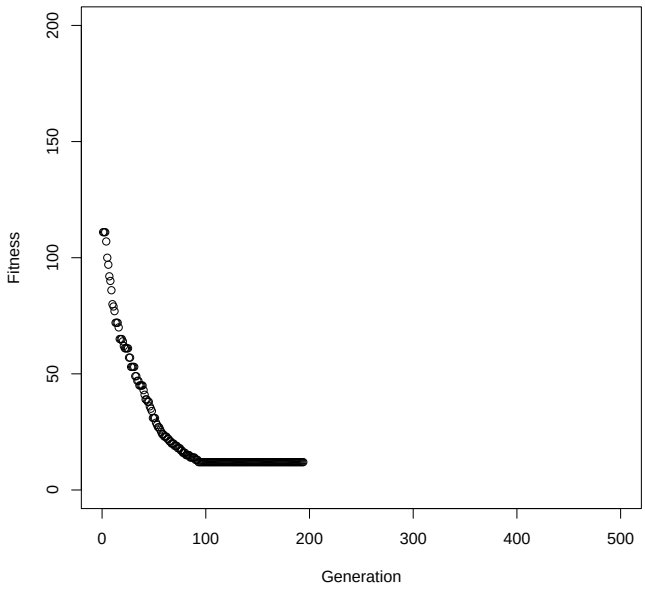

Figure 4.18: Algorithm convergence for exponent 250

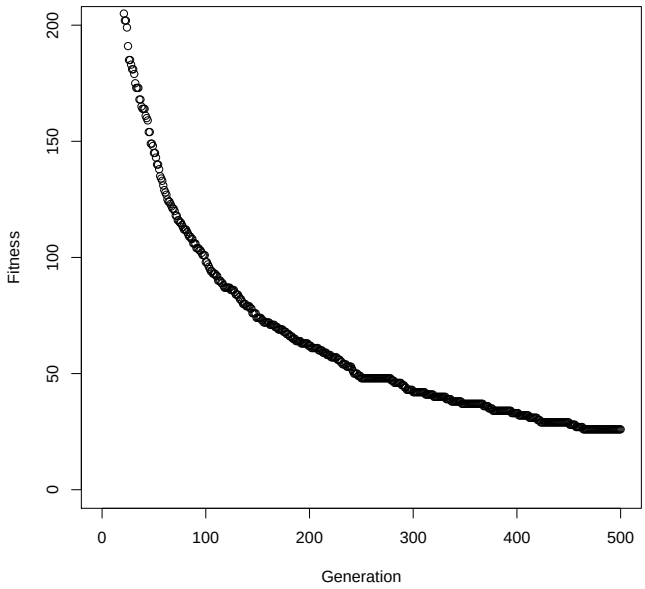

Figure 4.19: Algorithm convergence for exponent 768

4.2.10. Addition-subtraction Chain Optimization Results

Table 4.9: Fitness values for the optimization

Exponent	Minimal value	Max value	Average value	Standard deviation
23	6	53	8.2	8.21992
55	8	65	13.2045	16.3243
130	9	141	19.6962	35.9506
250	13	265	34.03	69.7326
768	21	791	80.3	205.955

Table 4.10: The optimal chains obtained through optimization

Exponent	Chain	Length
23	0->1->2->4->5->9->14->23	8
55	0->1->2->4->5->10->15->25->30->55	10
130	0->1->2->4->5->9->14->28->37->65->130	11
250	0->1->2->3->6->8->11->14->20->25->45->70->90->160->250	15
768	0->1->2->4->8->10->18->19->23->25->37->43->47->56->90->146->180->202->348->349->529->731->768	23

Convergence Graphs

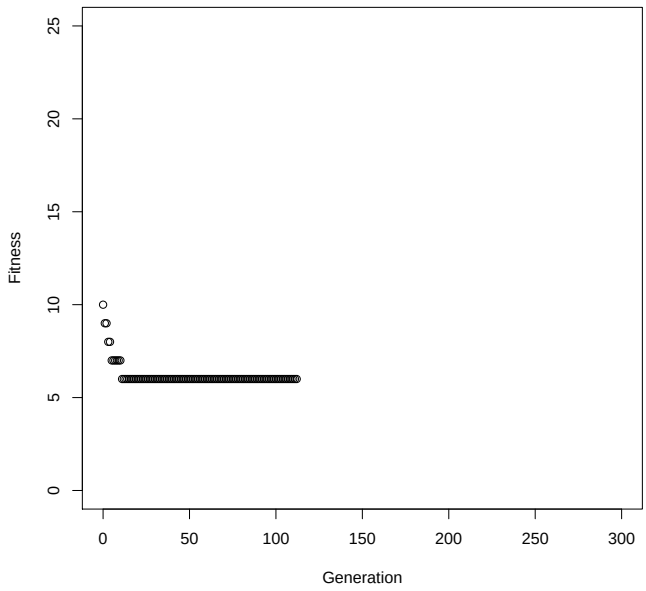

Figure 4.20: Algorithm convergence for exponent 23

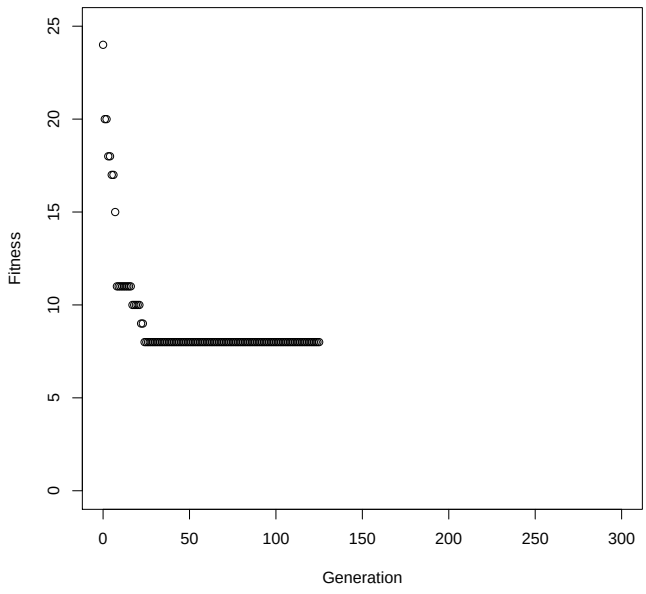

Figure 4.21: Algorithm convergence for exponent 55

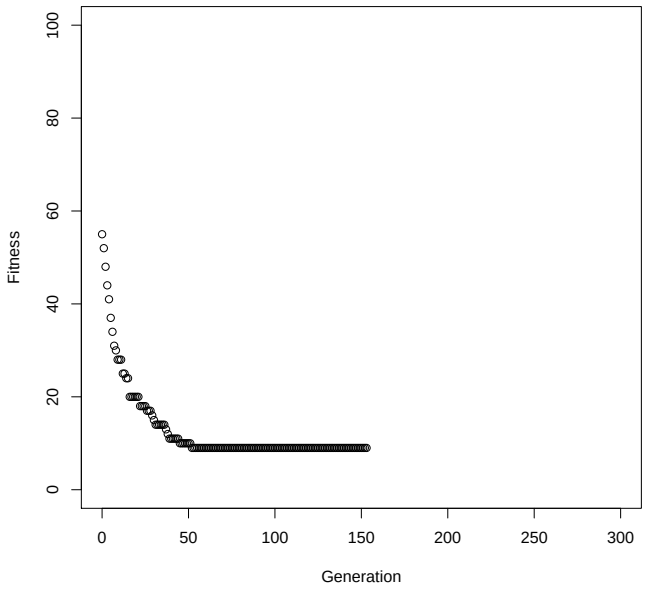

Figure 4.22: Algorithm convergence for exponent 130

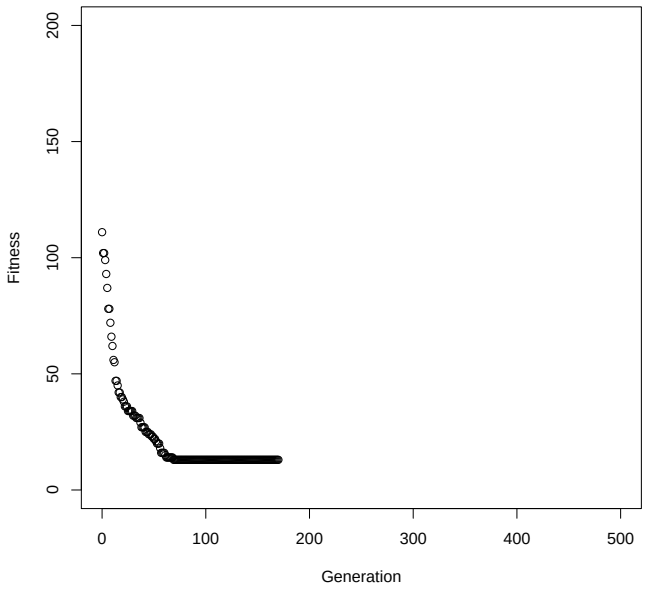

Figure 4.23: Algorithm convergence for exponent 250

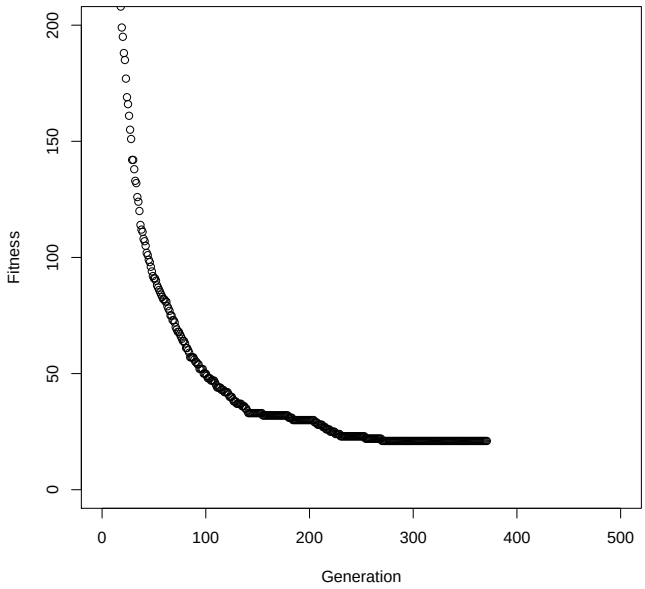

Figure 4.24: Algorithm convergence for exponent 768

4.3. Genetic Annealing Optimization

After seeing the results of the genetic algorithm we can test out more advanced options like the genetic annealing algorithm . Genetic annealing is a combination of the genetic algorithm and simulated annealing, the reason for combining the two is that the genetic algorithm, although a global search strategy doesn't guarantee the escape from local minima, the simulated annealing algorithm does.(Bertsimas and Tsitsiklis)

4.3.1. Genetic Annealing Algorithm

Because the genetic algorithm has been explained this section won't dig in deeper in the genetic part of the algorithm, the thing that needs to be explained is the simulated annealing algorithm which is an additional tool for the genetic algorithm in this case, when the simulated annealing process is clear then the genetic annealing process will be much easier to understand. The genetic annealing algorithm is a hybrid, random-search technique that uses the best of both worlds, that is, the genetic algorithm and the simulated annealing algorithm. Like the genetic algorithm it creates new solutions by exchanging genetic material between members of a population but instead of holding a tournament to decide which individuals will progress to the next generation the genetic annealing algorithm uses an adaptive, thermodynamic criterion based on a simple feedback scheme.

Simulated annealing

The simulated annealing algorithm is a probabilistic method for finding the global minimum of a cost function that may possess several local minima. It works by emulating the physical process whereby a solid is slowly cooled so that when eventually its structure is "frozen", this happens at a minimum energy configuration. It was originally inspired from the process of annealing in metal work. Annealing involves heating and cooling of a material to alter its physical properties due to the changes in its internal structure. As the metal cools its new structure becomes fixed, consequently causing the metal to retain its newly obtained properties. In simulated annealing we keep a temperature variable to simulate this heating process. We initially set the temperature high and then allow it to slowly 'cool' in the process. While this temperature variable is high the algorithm will be allowed, with more frequency, to accept solutions that are worse than our current solution. This gives the algorithm the ability to jump out of any local minima it finds itself in at the early stages of execution.(Bertsimas and Tsitsiklis)

Algorithm 6 Simulated Annealing Algorithm

$s \leftarrow s0$; $e \leftarrow E(s)$
$sbest \leftarrow s$; $ebest \leftarrow e$
$k \leftarrow 0$
while $k < kmax$ and $e > emax$ **do**
  $T \leftarrow temperature(k/kmax)$
  $snew \leftarrow neighbour(s)$
  $enew \leftarrow E(snew)$
  if P(e,$enew$,T) > $random()$ **then**
    $s \leftarrow snew$; $e \leftarrow enew$
  end if
  if $enew < ebest$ **then**
    $sbest \leftarrow snew$; $ebest \leftarrow enew$
  end if
  $k \leftarrow k + 1$
end while
return $sbest$

Basically this code is what was mentioned earlier, the state s at the beginning is the start state $s0$ and the energy of the system e is the energy of the start state. At the start logically we set the start state as the best state $sbest$ and its energy is the best known energy $ebest$. Before the annealing process we would like to define the maximum number of annealing iterations $kmax$ to limit the annealing process, this is also a way that we calculate the temperature which is a stop criteria. At the beginning of the iteration the temperature is calculated and we obtain the new state by the function $neighbour(s)$ which returns a neighbor solution. The central function to the algorithm is the acceptance probability function P which decides will the algorithm move to the neighbor solution or not. The implementations of the acceptance probability function, energy function and the temperature function are critical to the efficiency of the simulated annealing algorithm.

Now after this short introduction the simulated annealing algorithm we can define the optimization process for the genetic annealing algorithm, the simulated annealing algorithm can be combined in many ways with the genetic algorithm, the way explained here is implemented in the Evolutionary Computational Framework used for optimization.

Acceptance criteria

In the genetic annealing approach, you assign an energy threshold to each individual. Initially each threshold equals the energy of the individual to which it is assigned. A individuals threshold, not its energy, determines which trial mutations constitute acceptable improvements. If the energy of a mutant exceeds the threshold if the individual that spawned it, the mutant is rejected and the algorithm moves on to the next individual. However, if its energy is less than or equal to the threshold, the algorithm accepts the mutant as a replacement for its progenitor.(Price, 1994)

The genetic annealing algorithm uses the fitness of the individual to drive the annealing process. It uses an energy bank, represented by the real variable DE, to keep track of the energy liberated by successful mutants. Whenever a mutant passes the threshold test, the difference between the threshold and the mutant's energy is added to DE for temporary storage. Once this quantum of heat is accounted for, the threshold is reset so that it equals the energy of the accepted mutant and then the algorithm moves on to the next bit string.(Price, 1994)

Reheating

After each individual has been subjected to a random mutation the population is reheated so to speak by raising each threshold a little. The size of the increase depends both on the amount of energy accumulated in the energy bank and on the rate at which one desires to cool the population. If N equals the number of individuals in the population, then the average contribution to the energy bank is just DE/N. To fully reheat the population DE/N needs to be added to each threshold. Annealing results from the repeated cycles of collecting energy from successful mutants (spontaneous cooling) and then redistributing nearly all of it by raising the threshold energy of each population member equally (uniform reheating).

After they have been reheated, thresholds are higher than the energies of the individuals to which they have been assigned. This means that sometimes the algorithm is forced to accept a mutant even though its energy is not as low as the energy of the individual it replaces. Replacing an individual with a worse one may seem counterproductive, but these occasional reversals of fortune provide floundering individuals with helpful energy boost. In essence, the entire population acts like a giant heat reservoir that exchanges energy among its members. Less successful individuals can escape

suboptimal configurations by borrowing energy they need from the more successful versions.(Price, 1994)

The cooling constant

To relax the individuals into their optimal condition, they must be cooled very slowly. In a genetic annealing program the rate of cooling is controlled with the cooling constant, C, a real number from 0 to 1. For example $C = 1$ holds the population at a constant temperature by using all the energy stored in DE to reheat thresholds. By contrast, $C = 0$ releases all the stored energy in DE from the system and leaves the thresholds unaltered.(Price, 1994)

Algorithm 7 Genetic Annealing

```
Input: Generation g
energyBank = 0
for each individual do
    mutant = mutate(individual)
    if fitness(mutant) < fitness(individual) + threshold(individual) then
        diff = fitness(individual) + threshold(individual) - fitness(mutant)
        energyBank += diff
        replace individual with mutant
    end if
    energyDiff = energyBank * C / N
    for each individual do
        threshold(individual) += energyDiff
    end for
end for
```

4.3.2. Algorithm Application

In this section we will analyze the implementation of the genetic annealing algorithm in this case. If it isn't clear it has to be stated that the fitness function we are using in the genetic annealing algorithm is the same as in the genetic algorithm, nothing has changed. One can look at the genetic annealing algorithm the same as the standard genetic algorithm augmented with simulated annealing to conquer the local minima problem we've talked about. Also we are sticking to the bit string individual representation of the solution.

4.3.3. Addition Chain Optimization Results

Table 4.11: Fitness values for the optimization

Exponent	Minimal value	Max value	Average value	Standard deviation
23	6	30	7.33636	5.1361
55	8	64	10.7227	11.7186
130	10	141	20.0423	34.6713
250	12	263	39.5133	78.3195
768	22	792	92.9154	222.706

Table 4.12: The optimal chains obtained through optimization

Exponent	Chain	Length
23	1->2->3->5->10->20->23	7
55	1->2->4->5->9->11->22->33->55	9
130	1->2->4->5->7->12->13->26->52->65->130	11
250	1->2->4->5->8->9->14->22->36->58->67->125->250	13
768	1->2->4->5->6->10->15->17->32->38->43->47->62->86->133->143->145->231->246->290->521->522->768	23

Convergence Graphs

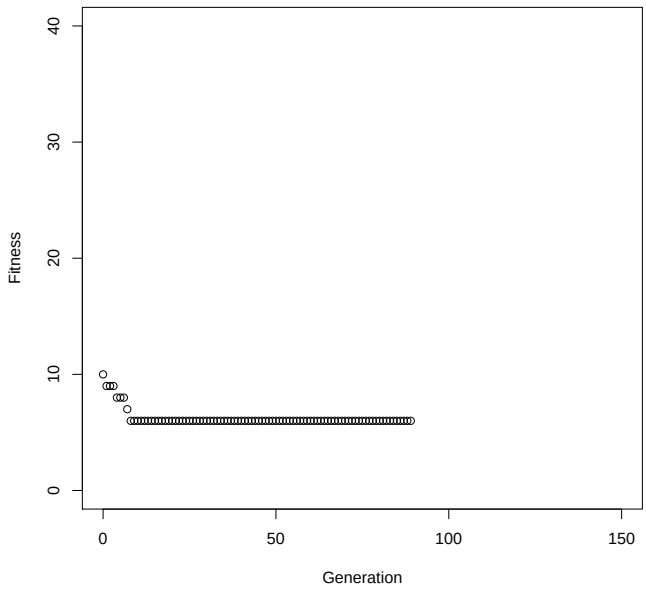

Figure 4.25: Algorithm convergence for exponent 23

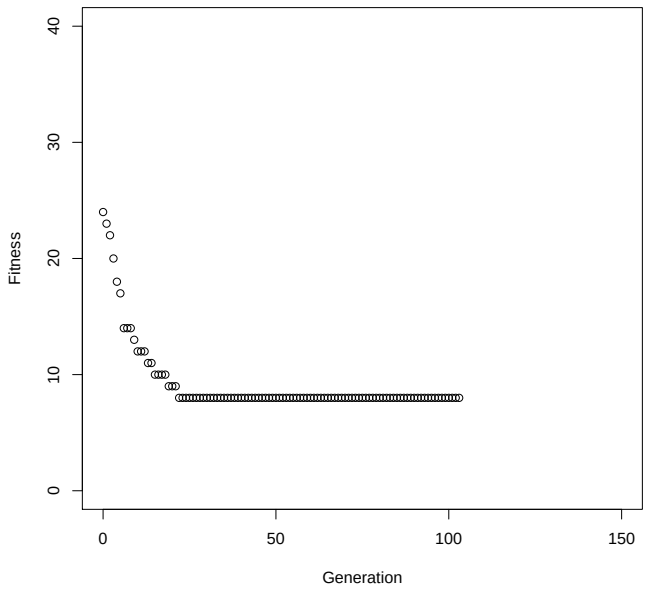

Figure 4.26: Algorithm convergence for exponent 55

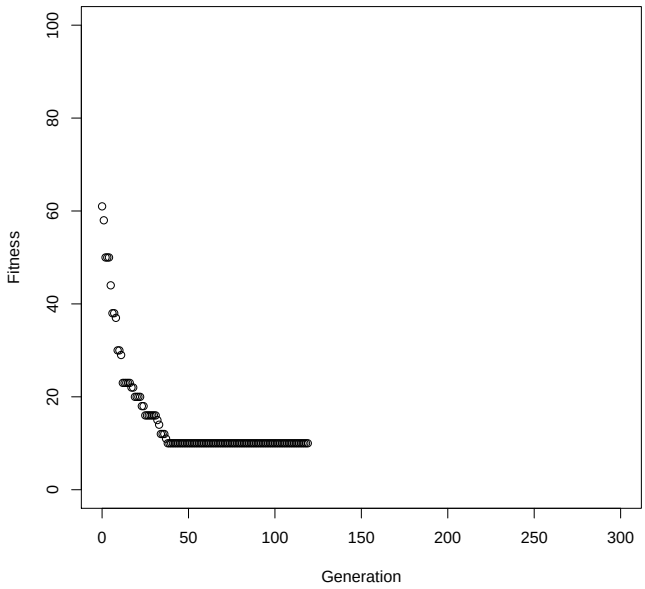

Figure 4.27: Algorithm convergence for exponent 130

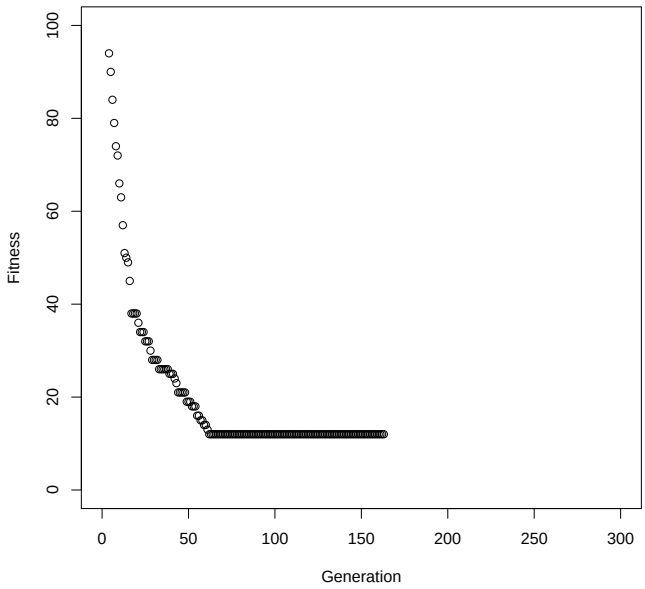

Figure 4.28: Algorithm convergence for exponent 250

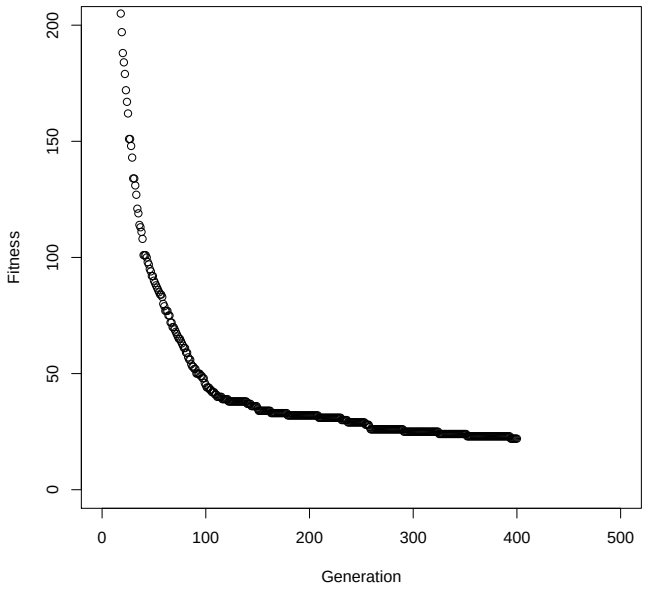

Figure 4.29: Algorithm convergence for exponent 768

4.3.4. Differential Addition Chain Optimization Results

Table 4.13: Fitness values for the optimization

Exponent	Minimal value	Max value	Average value	Standard deviation
23	7	31	8.65455	5.66804
55	8	64	13.4045	16.1899
130	11	142	22.0038	36.2507
250	14	265	48.9167	86.660
768	32	801	144.046	271.828

Table 4.14: The optimal chains obtained through optimization

Exponent	Chain	Length
23	0->1->2->3->5->8->13->18->23	9
55	0->1->2->3->5->10->15->25->30->55	10
130	0->1->2->3->4->7->10->17->24->31->48->65->130	13
250	0->1->2->3->5->7->9->11->18->25->29->32->61->64->125->250	16
768	0->1->2->3->5->8->10->11->12->19->21->22->24->25-> 29->30->38->45->46->50->55->59->75->84->89->120->139-> 150->189->239->259->379->389->768	34

Convergence Graphs

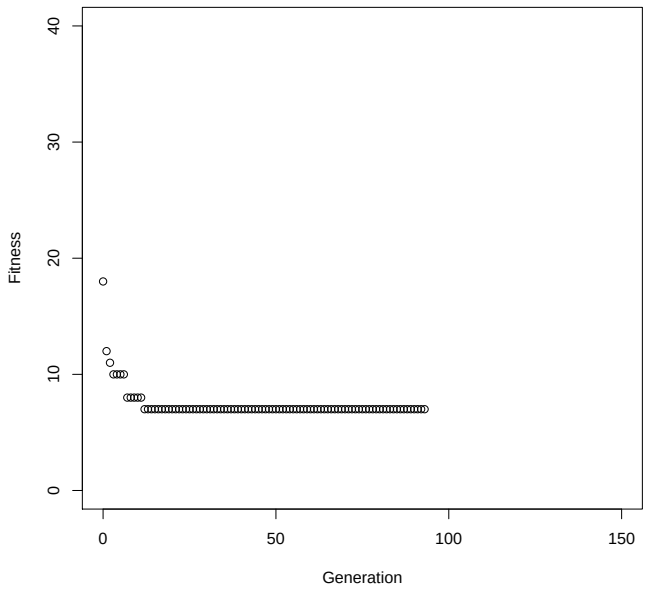

Figure 4.30: Algorithm convergence for exponent 23

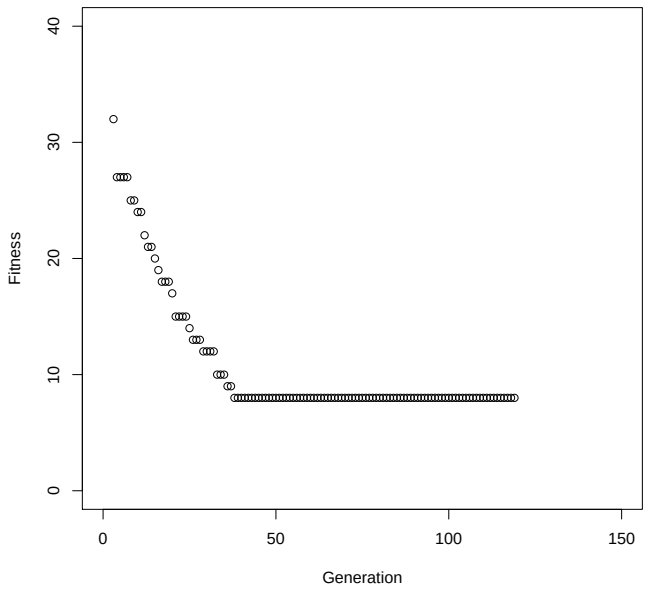

Figure 4.31: Algorithm convergence for exponent 55

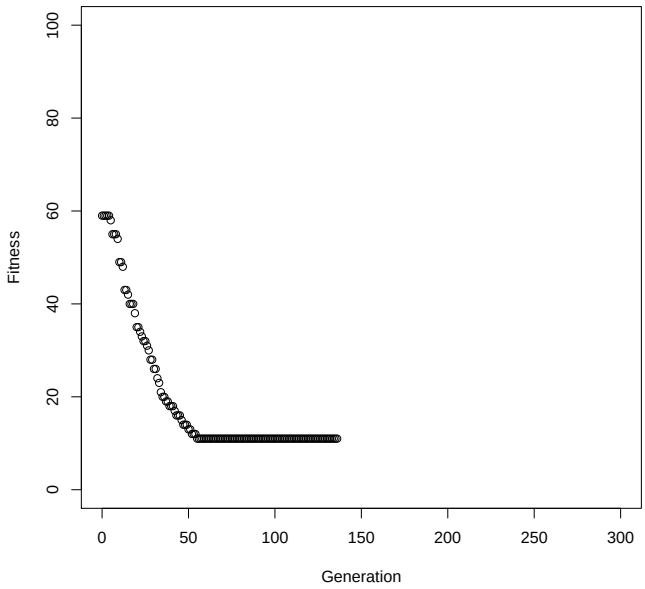

Figure 4.32: Algorithm convergence for exponent 130

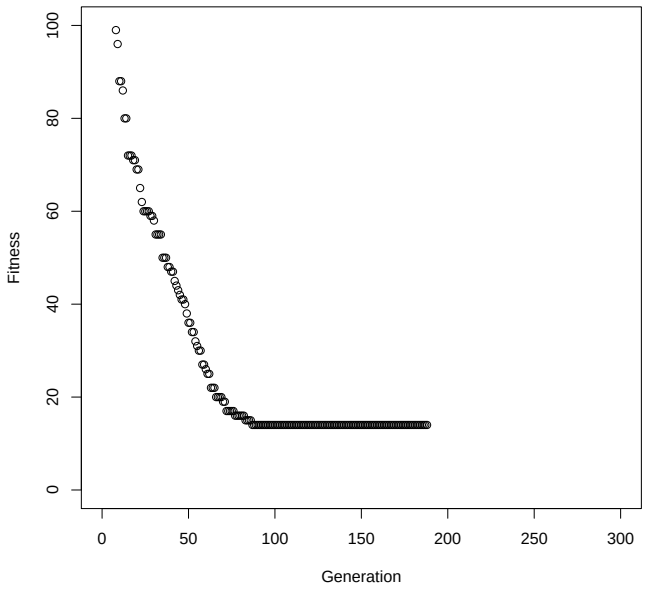

Figure 4.33: Algorithm convergence for exponent 250

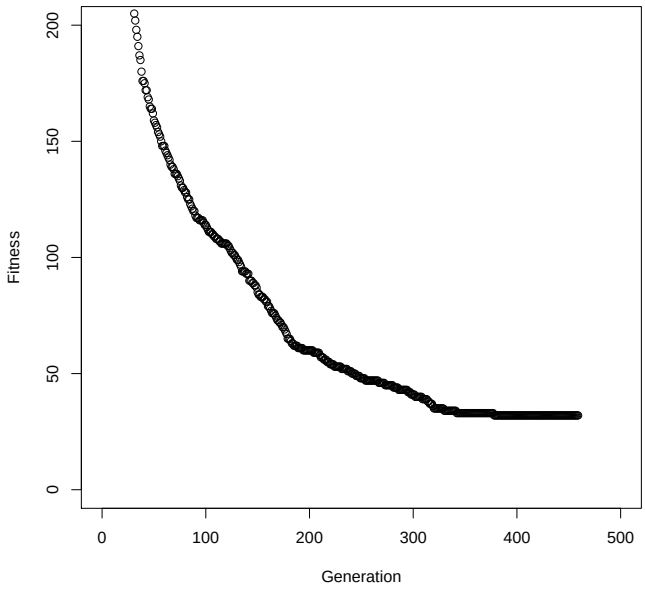

Figure 4.34: Algorithm convergence for exponent 250

4.3.5. Differential Addition-subtraction Chain Optimization Results

Table 4.15: Fitness values for the optimization

Exponent	Minimal value	Max value	Average value	Standard deviation
23	7	32	8.50455	5.53486
55	8	64	13.4091	16.1668
130	12	143	26.8846	41.2729
250	12	263	40.2867	79.2459
768	24	793	100.808	231.29

Table 4.16: The optimal chains obtained through optimization

Exponent	Chain	Length
23	0->1->2->3->4->7->8->15->23	9
55	0->1->2->3->5->10->15->25->30->55	10
130	0->1->2->3->5->6->9->11->17->20->28->37->65->130	14
250	0->1->2->3->5->7->8->9->17->25->50->75->125->250	14
768	1->2->3->5->6->11->12->15->24->35->38->76-> 111->116->123->131->234->365->403->768	26

Convergence Graphs

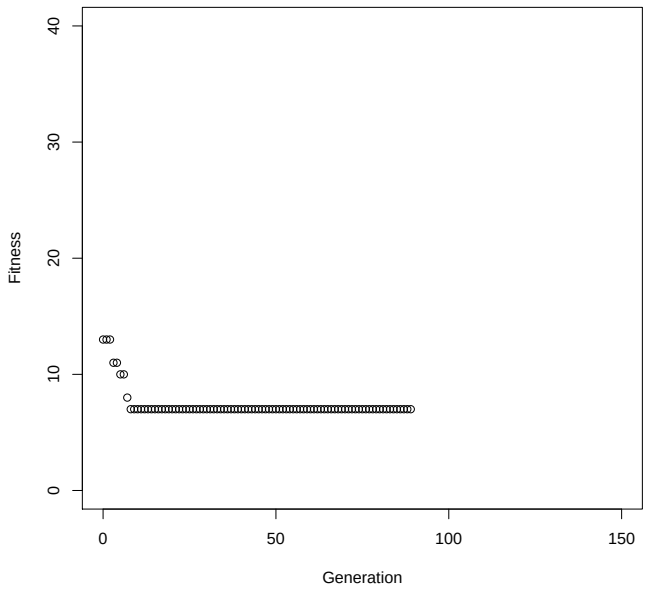

Figure 4.35: Algorithm convergence for exponent 23

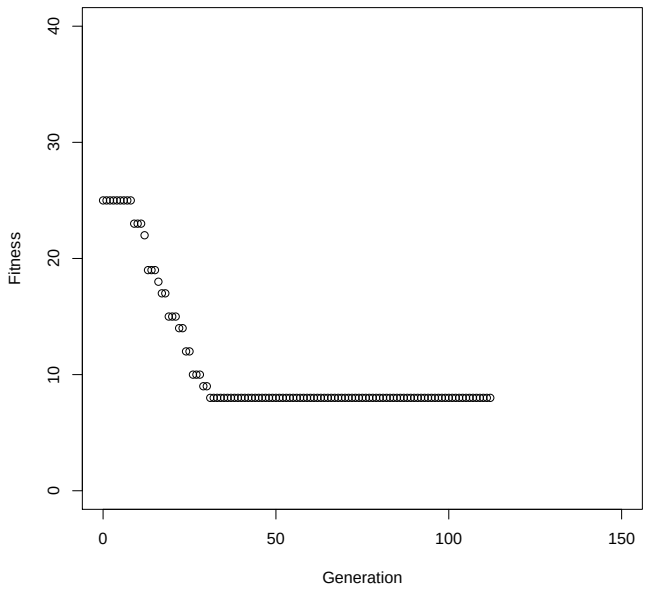

Figure 4.36: Algorithm convergence for exponent 55

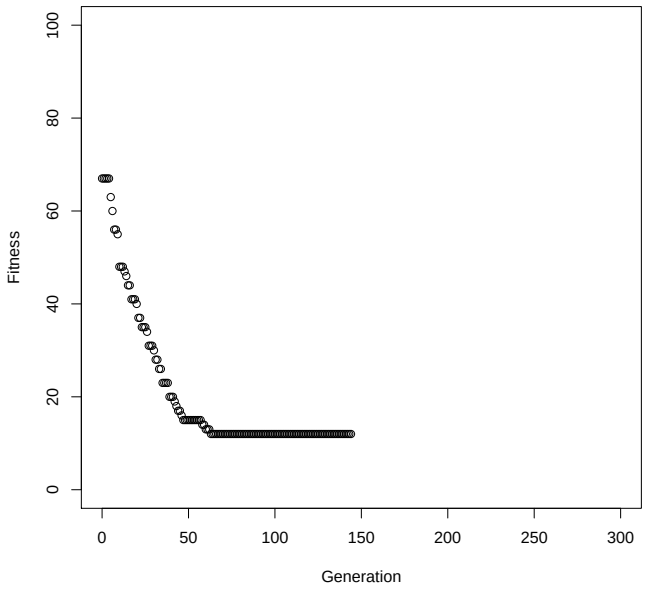

Figure 4.37: Algorithm convergence for exponent 130

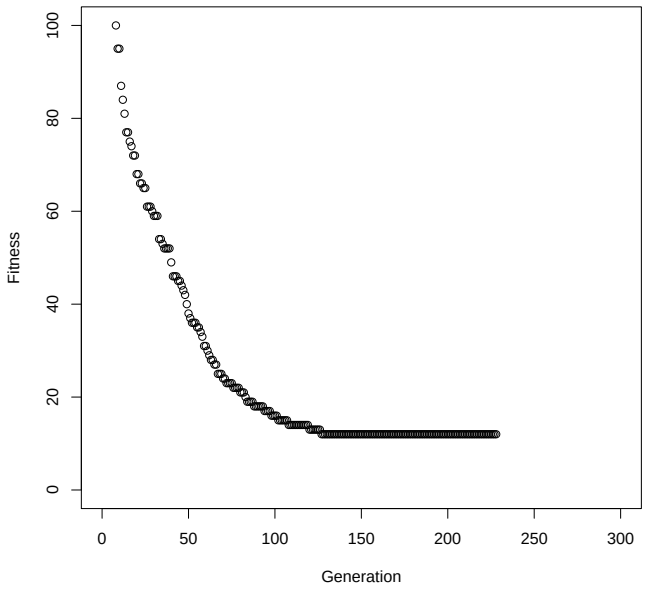

Figure 4.38: Algorithm convergence for exponent 250

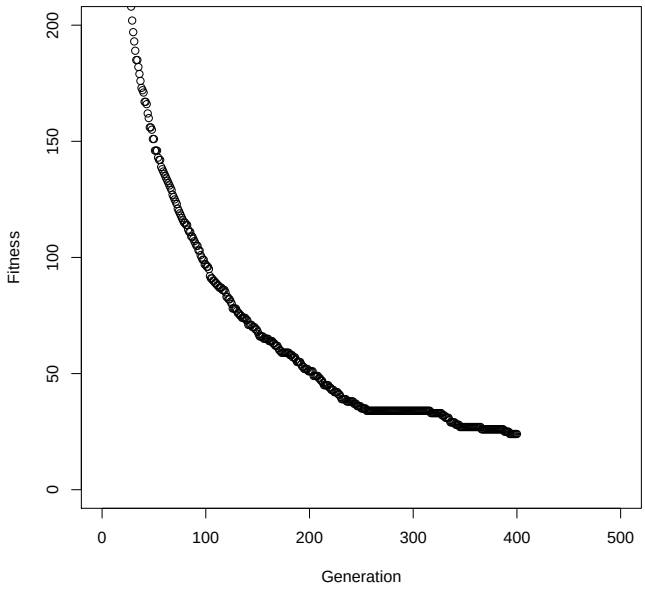

Figure 4.39: Algorithm convergence for exponent 768

4.3.6. Addition-subtraction Chain Optimization Results

Table 4.17: Fitness values for the optimization

Exponent	Minimal value	Max value	Average value	Standard deviation
23	6	30	7.05909	4.53612
55	8	64	12.7636	15.3581
130	9	140	17.9692	32.8383
250	13	264	36.1933	72.2702
768	19	788	60.3077	173.69

Table 4.18: The optimal chains obtained through optimization

Exponent	Chain	Length
23	1->2->4->5->9->14->23	7
55	1->2->4->5->9->14->23->46->55	9
130	1->2->3->5->10->15->30->60->65->130	10
250	1->2->4->5->10->12->17->24->36->48->84->101->125->250	14
768	1->2->3->5->6->11->12->15->24->35->38->76-> 111->116->123->131->234->365->403->768	20

Convergence Graphs

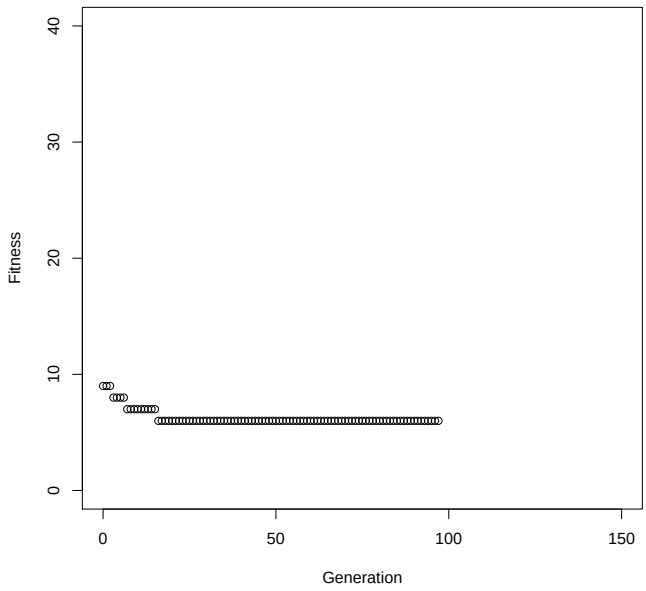

Figure 4.40: Algorithm convergence for exponent 23

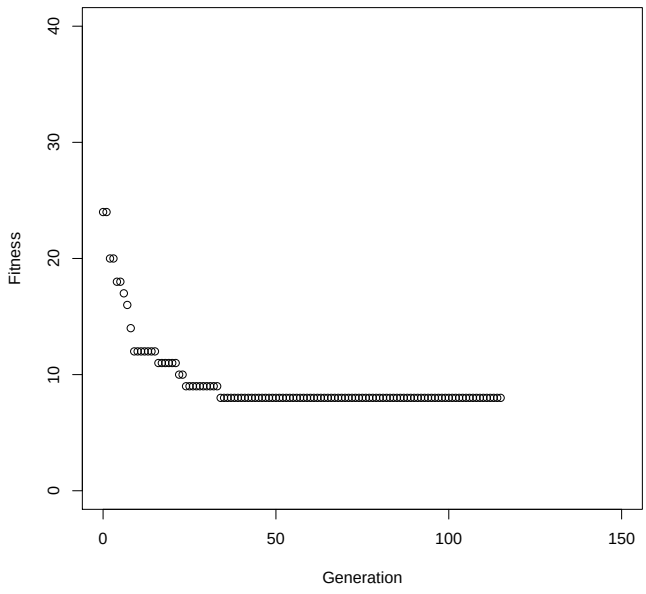

Figure 4.41: Algorithm convergence for exponent 55

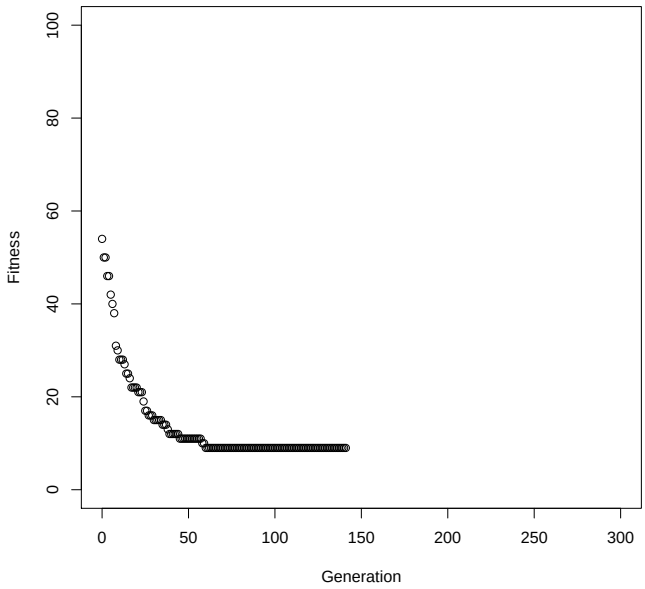

Figure 4.42: Algorithm convergence for exponent 130

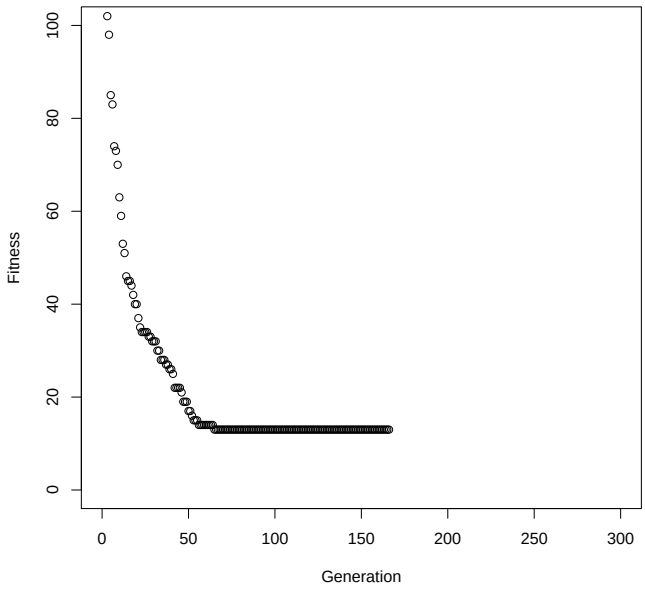

Figure 4.43: Algorithm convergence for exponent 250

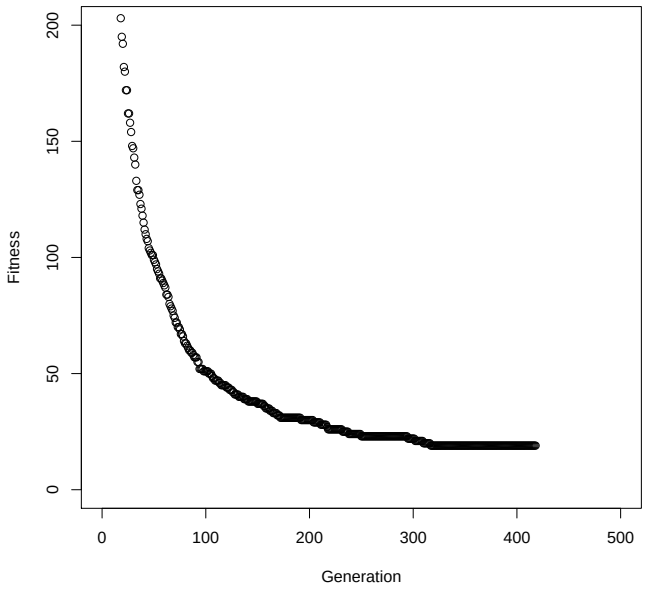

Figure 4.44: Algorithm convergence for exponent 768

4.4. Genetic Optimization Results Analysis

As it can be seen from the results obtained in the optimization, the genetic algorithm obtains better results in the optimization in most cases, but not in all cases. From the convergence graphs it is safe to deduct that the convergence of both algorithms is relatively equal. In the case of the addition-subtraction chains and differential addition-subtraction chains the results yield an interesting fact, the algorithms converge toward a standard addition chain solution or differential addition chain solution rather than using the benefits of subtraction to ease the exponentiation, therefore the algorithm proposed for addition-subtraction chain optimization in (Nedjah and de Macedo Mourelle) is not satisfactory for the problem solution and a better algorithm is needed.

The time consumption of the experiments also needs to be mentioned. The experiments were conducted on a laptop with an Intel i5 CPU, for the exponents lower than 250 this didn't present a problem, but for larger exponents the time consumption of the experiments becomes drastically worse, concretely for the exponent 768 the average run time is about 1531 seconds.

4.5. Particle Swarm Optimization

The particle swarm optimization algorithm was discovered by C.W.Reynolds. The algorithm is a population-type algorithm, the population consists of particles that fly through a multidimensional search space and change their position based on their experience and their close neighbors experience. Throughout the optimization process an individual has access to the info of its best found solution so far and the globally best found solution so far, that way the algorithm combines the global search with the local search i.e. local fine tuning.(Čupić, 2013)

4.5.1. Particle Representation

We will implement the algorithm that will only handle legal addition chains so we need not concern ourselves with their validity in the fitness function.

Each particle is directly represented by the addition chain, that is, the elements of the particle are the elements of the addition chain. For example, for the exponent $e = 79$, a possible addition chain is the following: $1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 12 \rightarrow 24 \rightarrow 30 \rightarrow 60 \rightarrow 72 \rightarrow 78 \rightarrow 79$.(Leon-Javier et al., 2009)

4.5.2. Algorithm Initialization

First off we need to initialize the algorithm because we will be building valid addition chains from the start. The code to generate particles for a given exponent e is:

Algorithm 8 Swarm Initialization

for $n = 1$ to $n = PopulationSize$ **do**
  Set $p_{n1} = 1$
  Set $p_{n2} = 2$
  for $i = 3$ to $p_{ni} = e$ **do**
    repeat
      Randomly select k with $1 \leq k < i$
      $p_i = p_k + p_{i-1}$
    until $p_i > e$
  end for
end for

4.5.3. Fitness Function

The fitness function in trivial, as we mentioned we are dealing with valid addition chains, thus in the example of the exponent $e = 79$ the addition chain is $1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 12 \rightarrow 24 \rightarrow 30 \rightarrow 60 \rightarrow 72 \rightarrow 78 \rightarrow 79$ and its fitness value is 10 (the real value of 11 was subtracted by 1 to maintain congruency with previously known works).(Leon-Javier et al., 2009)

4.5.4. Velocity Values

The velocity chain is associated with the particle, the particles and the velocity chains are of the same length. A velocity chain contains at each elements position the index of the element that was added to obtain it. As we can see in figure 4.46

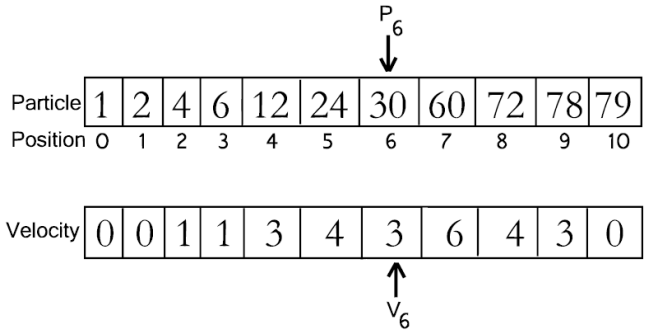

Figure 4.45: A particle and its velocity chain(Leon-Javier et al., 2009)

the number 30 in the particle chain is at position 6, in the velocity chain we have the number 3 which is the index of the number 6 in the particle chain. Therefore we can conclude that the number 30 in the chain was obtained by adding 6 to 24.

4.5.5. Particles and Velocities Updating

The particles position is affected by the particle's best experience ($pBest$) and its neighborhoods best experience (gBest). For this case the neighborhood is composed of all the elements in the swarm.(Leon-Javier et al., 2009)

We scan the values in each particle. Then, at each step, we randomly select if the particle is affected by the rule in the velocity chain associated with: (1) the particle's best experience ($pBest$) or, (2) the particle's best neighbor ($gBest$) or, (3) its own velocity. This procedure is applied for each particle. (Leon-Javier et al., 2009)

Algorithm 9 Particles and Velocities Updating

for $i = 2$ to l **do**
  if $2p_{i-1} < e$ **then**
    if $flip(Pb)$ **then**
      if $pBest$'s length < i **then**
        $p_i = p_{i-1} + p_{v_pBest_i}$
        $v_i = v_pBest_i$
      else
        $v_i = random(0, i-1)$
        $p_i = p_{i-1} + p_{v_i}$
      end if
    else if $flip(Pg)$ **then**
      if $gBest's\ length < i$ **then**
        $p_i = p_{i-1} + p_{v_gBest_i}$
        $v_i = v_gBest_i$
      else
        $v_i = random(0, i-1)$
        $p_i = p_{i-1} + p_{v_i}$
      end if
    else
      if $v_i < i$ **then**
        $p_i = p_i + p_{v_i}$
      else
        $v_i = random(0, i-1)$
        $p_i = p_{i-1} + p_{v_i}$
      end if
    end if
  else
    if $p_{i-1}! = e$ **then**
      for $m = i-2$ to 0 **do**
        if $p_{i-1}+ \leq e$ **then**
          $p_i = p_{i-1} + p_m$
          $v_i = m$
          break
        end if
      end for
    end if
  end if
end for

4.5.6. Optimization Results

Because this is an interesting fresh method it's worth mentioning the results of the optimization process, the results stated here are obtained from (Leon-Javier et al., 2009).

e	Addition chain	length
3	1-2-3	2
5	1-2-4-5	3
7	1-2-4-6-7	4
11	1-2-3-6-9-11	5
19	1-2-4-6-12-18-19	6
29	1-2-3-6-12-13-26-29	7
47	1-2-4-8-9-13-26-39-47	8
71	1-2-4-5-10-20-30-60-70-71	9
127	1-2-3-6-12-24-48-72-120-126-127	10
191	1-2-3-5-10-11-21-31-62-124-186-191	11
379	1-2-3-6-9-18-27-54-108-216-324-378-379	12
607	1-2-4-8-9-18-36-72-144-288-576-594-603-607	13
1087	1-2-4-8-9-18-36-72-144-145-290-580-870-1015-1087	14
1903	1-2-4-5-9-13-26-52-78-156-312-624-637-1261-1898-1903	15
3583	1-2-3-5-10-20-40-80-160-163-326-652-1304-1956-3260-3423-3583	16
6271	1-2-3-6-12-18-30-60-120-240-480-960-1920-3840-5760-6240-6270-6271	17
11231	1-2-3-6-12-24-25-50-100-200-400-800-1600-3200-4800-8000-11200-11225-11231	18

Figure 4.46: Results for particle swarm optimization (Leon-Javier et al., 2009)

Clearly when comparing the results obtained with the particle swarm optimization algorithm with the genetic algorithm and the genetic annealing algorithm the particle swarm optimization algorithm has a certain edge over the two, the greatest difference can be seen for exponents greater than 700, where the particle swarm algorithm outperforms its rivals substantially. On the other hand, the problem of addition-subtraction chains, differential addition chains and differential addition subtraction chains for this algorithm is not defined mainly because of the difficulty in defining the fitness function.

5. Conclusion

From the undertaken experiments it can be concluded that the particle swarm optimization algorithm has a clear advantage over the genetic and genetic annealing algorithm in the larger exponents which is achieved with its continuous construction of valid addition chains that make the validity checking redundant. When comparing the genetic annealing algorithm and the genetic algorithm a small difference in convergence can be seen thanks to the annealing that conquers the local minima although the genetic annealing algorithm (the ECF implementation) didn't provide better results in every case.

Concretely when observing the algorithm provided for addition-subtraction chains and differential addition chains it can be concluded that the methods suggested so far rather converge to a clear addition chain or differential addition chain than use the subtraction benefit, therefore, there is a need for exploring better methods in optimizing addition-subtraction chains in future research. The processing power needed to compute small addition chains and their variants although using methods of evolutionary computation remains an expensive and a decisive factor in the problem. All in all, the area of addition chains remains an interesting subject for future research and enhancement.

BIBLIOGRAPHY

Dimitris Bertsimas and John Tsitsiklis. *Simulated Annealing*.

Marin Golub. *Genetski algoritam*. 2004.

Daniel J.Bernstein. *Differential addition chains*. 2006.

Donald Knuth. *The Art of Computer Programming - Seminumerical Algorithms*. 2002.

Alejandro Leon-Javier, Nareli Cruz-Cortes, Marco A. Moreno-Armendariz, and Sandra Orantes-Jimenez. *Finding Minimal Addition Chains with a Particle Swarm Optimization Algorithm*. 2009.

Nadia Nedjah and Luiza de Macedo Mourelle. *Minimal Addition-Subtraction Chains Using Genetic Algorithms*.

Kenneth V. Price. *Genetic Annealing*. 1994.

Kruno Tomola-Fabro. *Optimizacija parametara stohastičkih algoritama uporabom metode podijeli i usporedi*. 2013.

Marko Čupić. *Prirodom inspirirani optimizacijski algoritmi*. 2013.

Abstract

This paper is about optimizing exponentiation and analyzes some past exponentiation techniques before focusing on addition chains as the means of optimal exponentiation. It also provides some mathematical basis for understanding addition chains and their variations such as differential addition chains, addition-subtraction chains and differential addition-subtraction chains. The papers core theme is finding optimal addition chains through methods of evolutionary computation with algorithms such as the genetic algorithm, genetic annealing and particle swarm optimization for which results of optimization are given and compared.

Keywords: addition chains, strong addition chains, differential addition chains, addition-subtraction chains, exponentiation, evolutionary computation, particle swarm optimization, genetic annealing, genetic algorithm