Internal Report 2010–03 June 2010

Universiteit Leiden
Opleiding Informatica

Compression of DNA Sequences with Genetic Programming

Emiel Witteman

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

In this thesis we will try to evolve a program (tree) which will reproduce an original text or a DNA string or more general data. To this end we use Genetic Programming (GP). The goal is that the program will be shorter than the original, so compression is achieved. Chances are that the program will not exactly reproduce the original, which mean lossy compression. If necessary, the difference between the lossy compressed string and the original string can be stored to make the compression lossless. Is this construction feasible, how does the compression ratio compare to generic data compression algorithms?

The GP we created appears to be more suitable for general pattern recognition than compression, huge run times make real life application unsuitable for compression. Simulating real DNA evolution might be a nice spin-off.

Contents

Chapter 1

Introduction

1.1 Introduction

"Genetic programming addresses the problem of automatic programming, namely, the problem of how to enable a computer to do useful things without instructing it, step by step, on how to do it." [Foreword John R.Koza [1]]

In this thesis the goal is to evolve a program which reproduces an original "text" without us making the actual program itself, tho this end we use Genetic Programming (GP). We simply start with a bunch of random programs, which we will evolve in the same way nature evolves all kinds of animals and plants etc. So our programs will be mutated, mate with other programs, have children and eventually die so the stronger programs will continue the cycle of life. All we have to do is to define how the programs "look" like, which programs are the "fit" ones and are allowed to mate and survive. Start the evolutionary process and see what happens.

Compression in general is a method which aim is to store certain information using less storage space without losing essential information. The Internet is full of compression, photos are often stored in `jpeg` format, source code is stored in an archive and compressed. Compression is all around us. Wouldn't it be great if a genetic program could give new insights in compression?

We start with a study of the available material on Genetic Programming (Chapter 3), Delta Algorithms (Chapter 4) and Compression (Chapter 5). The results of these studies will be summarized in the those chapters. After this study we will apply and combine this knowledge in a software framework (Chapter 6) which will be used to run experiments (Chapter 7). Finally we will discuss the results and theorize on further improvements (Chapter 8).

This Master's Thesis was written at the Leiden Institute of Advanced Computer Science of Leiden University (LIACS). The supervisors are Dr. W.A. Kosters and Dr. M.T.M. Emmerich.

Chapter 2

Problem Description

2.1 Introduction

In this chapter we will describe our problem: We try to achieve compression with the aid of Genetic Programming. At first we try to solve some simple problems. Our first challenge will be compressing "DNA": strings containing only four different characters: A, C, G and T. In nature DNA strings are very long, we will start with some smaller fabricated strings, we don't want to wait days before an evolutionary process is finished. But if the originals are too small compression can't be achieved. Therefore, for some first trial runs we will choose a string length of 100 characters, which seems like a good size. We call the output of a program a candidate string, the more the candidate string will resemble the original string the better. The programs with a better candidate string will have a greater chance of surviving and producing offspring.

2.2 Small benchmark problem sets

In order to see if Genetic Programming will lead to a working compression algorithm we define some small problems. These do not pose a very big challenge for a compression algorithm, and we humans can see very easily how we would compress them. This enables us to monitor if the Genetic Programming does its job in delivering a feasible compression algorithm. The four basic benchmark problems are:

A simple repetition of 100 `A`'s:

`AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`

Table 2.1: Small benchmark problem 1.

A simple `ACGT` repetition 25 times:

```
ACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTAC
GTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGT
```

Table 2.2: Small benchmark problem 2.

A simple `AACCGGTT` repetition max 100 characters:

```
AACCGGTTAACCGGTTAACCGGTTAACCGGTTAACCGGTTAACCGGTTAA
CCGGTTAACCGGTTAACCGGTTAACCGGTTAACCGGTTAACCGGTTAACC
```

Table 2.3: Small benchmark problem 3.

Four times `ACGTAACCGGTT` followed by the DNA mirror image:

```
ACGTAACCGGTTACGTAACCGGTTACGTAACCGGTTACGTAACCGGTT
TGCATTGGCCAATGCATTGGCCAATGCATTGGCCAATGCATTGGCCAA
```

Table 2.4: Small benchmark problem 4.

Chapter 3

Genetic Programming

3.1 Introduction

Genetic Programming (GP) is an approach to problem solving in the area of natural computing; as a general reference for this chapter we mention [1]. With "normal" sequential programming a problem is analyzed, divided into sub-problems, and when all the sub-problems have been defined one starts writing a program to solve the sub-problems one by one and thus solving the problem as a whole. With Genetic Programming solving problems is approached in a different manner. The focus is not on the problem we want to solve but we focus on constructing a virtual environment, which is capable of solving the problem for us. Within this virtual environment computer programs are randomly created, some solving the problem better than others. We only need to direct the computer into the right direction to find the most suitable program to solve the problem.

The inspiration of this approach comes from the theory of evolution as seen in biology. We don't intend to fully reproduce biological evolution but we do take several concepts from it and apply it to our virtual environment. Decisions have to be made concerning the individuals in our virtual environment, in our case programs: the way they should "look" like, how to create new programs from the existing ones, which programs should be considered fit, the total number of individuals in our virtual environment, etc. By making the "right" choices in all these matters, our virtual environment should solve the problem for us and by studying this process and its outcome we hope to get some new insights in the problem we want to solve.

3.2 Natural selection

Darwin [4] argued that: "... if variations useful to any organic being do occur, assuredly individuals thus characterized will have the best chance of being preserved in the struggle for life; and from the strong principle of inheritance they will tend to produce offspring similarly characterized. This principle of preservation, I have called, for the sake of brevity, Natural Selection."
In these words from Darwin we can find some essential conditions needed to facilitate

evolution:

- Variation of individuals should occur in the population.
- Different variants of individuals should differ in the probability of the individual to survive and create offspring.
- Offspring should be able to inherit characteristics from the parent(s).
- Competition between individuals should occur, the "struggle for life".

Our virtual environment will need to address these four conditions. Without them no evolution will take place.

3.3 Representation

Individuals in Genetic Programs are traditionally represented as tree structures. But graphs and linear representations are also possible. A tree representation has the advantage of the relative ease of evaluating the result and implementing variation of individuals by means of mutation and crossover. Humans generally think that a tree representation gives more insight than a linear representation, thus making it easier to keep an eye on the evolution process. In this research we only use tree structures, one of our goals is to get some new insights in compression, so we choose a representation which is easy to understand.

3.4 Tree, the datastructure

In this section we briefly mention some basic aspects of trees, see Figure 3.1 for more information.

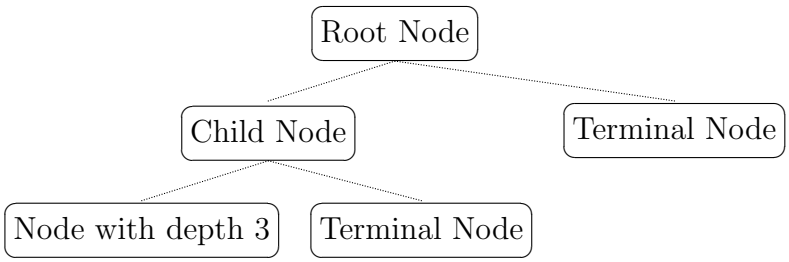


Figure 3.1: Tree, the datastructure.

The following aspects are defined regarding trees:

- Node: Smallest atom of a tree which holds either a terminal or a function, a node can have one parent and several children.

- Root: The top node of the tree which has no parent. All the other nodes have one parent.
- Terminal nodes and set: A terminal node is a node which has no children, and contains an element from the terminal set. The terminal set consists of all the possible terminals which are allowed in the tree. Terminal nodes are also called leaf nodes or leafs.
- Function set: The function set consists of all the possible functions which are allowed in the tree. Functions get their input from the child nodes.
- Arity: The number of child nodes of a function node.
- Depth: The minimal number of nodes that need to be traversed to get from the root node of the tree to the selected node.
- Maximal Tree Depth: The maximal depth of a tree constraints all nodes of the tree to have this maximal depth or less. This is a common way to limit the size of the tree.
- Maximal Tree Size: The maximal number of nodes a tree may contain.
- Binary tree: A tree with at most an arity of 2, child nodes are called *left* and *right*.

3.5 Initialization

The next step is to create the initial population, using some random process. In order to achieve the best results the initial population should be as diverse as possible. Several methods exist to create an initial population:

- Grow: As long as the tree of the individual has not reached the maximum depth randomly choose from the function and terminal set. This way not all nodes from the tree will have the maximum depth.
- Full: Choose randomly from the function set if the current node is not at maximum depth; choose randomly from the terminal set if we are at the maximum depth.
- Ramped half-and-half: Also create trees lower than the maximum depth. For best results half of them are grown and half of them are made with the full method. So if the maximum depth is n, equally spread the number of trees with depth n, $n-1$, ... 1. And create one half with the grow method and the other half with the full method.

We will use the ramped half-and-half method in our GP, because it will give the most variation of the initial population.

3.6 Variation

The diversity in the population is maintained and expanded by two mechanisms:

- Crossover.
- Mutation.

3.6.1 Crossover

The crossover operator creates variation by combining the genetic material of two or more parents. This is done by taking parts of the parents to create the children. The children can "be lucky" and can receive the "good" parts of the parents and thus be improved versions of their parents. See Figure 3.2 for an example. In this example the nodes and terminals which are selected for crossover are marked with a double boxed line. Then the actual crossover takes place and the sub-trees are exchanged. Child 1 has some genetic material from Parent 2; the node labeled "P2 N3" with the leaves "P2 L4" and "P2 L5". Child 2 received the node "P1 N3" with the leaves "P1 L1" and "P1 L2"

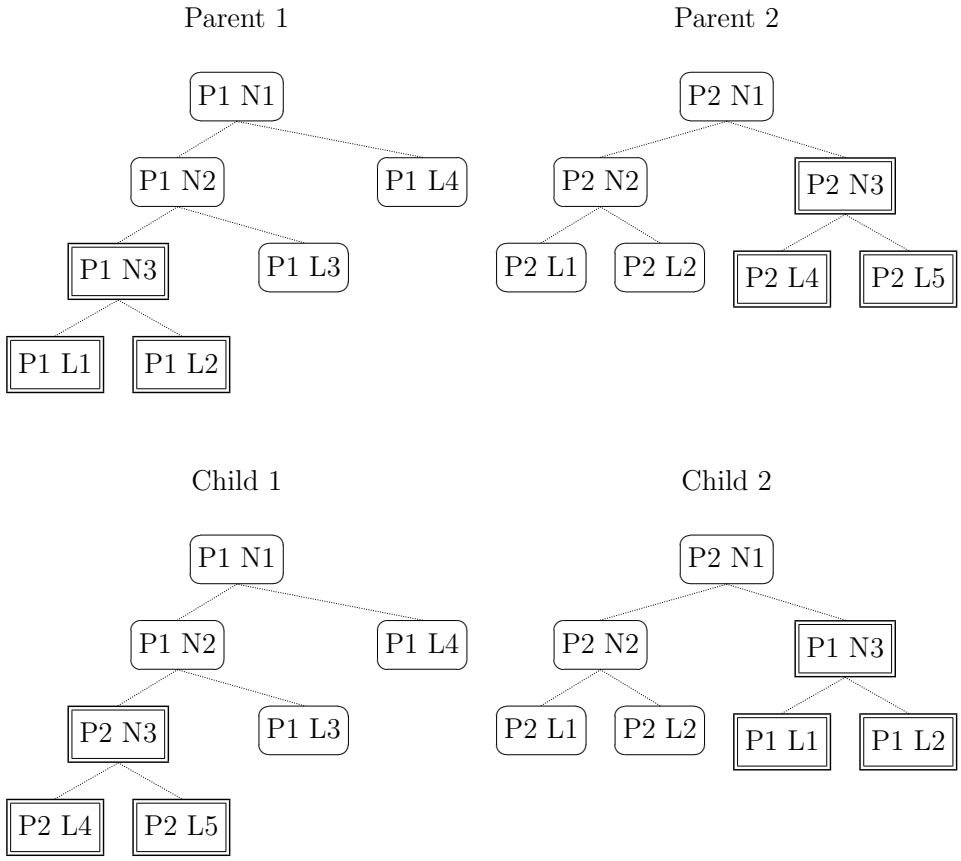


Figure 3.2: Tree-based crossover.

3.6.2 Mutation

Mutation is used to change an existing individual, this opens new possibilities to genetic material which could be very beneficial to solve the problem. Two types of mutation can be distinguished when using trees:

- Node mutation: Select one node and change it. Attention must be payed to either preserve the arity, or "fixing" in case the arity changes.
- Sub-tree mutation: Select one node, delete the corresponding sub-tree and grow a new sub-tree. Attention must be payed to the maximum depth.

3.7 Fitness

The fitness of an individual is very important. The more "fit" an individual is, the better its ability to solve the problem. Now comes the best part: We determine which individuals receive a good fitness and which individuals receive a bad fitness. We do this with a *fitness function*, which grades the result of the individual. This fitness is used for selection and reproduction, this way we direct the evolutionary process into the direction which is most likely to produce a solution to our problem. As one can imagine a wrong fitness function could lead to a suboptimal solution or no solution at all. It is not always possible to make the perfect fitness function, fitness functions need to be as "fast" as possible. Every individual in the population needs its fitness to be determined so a slow fitness function results in a slow evolution. In Section 6.3 we describe the fitness function we used.

Several types of fitness have been defined over time:

- Standardized: The fittest individual gets the value zero assigned.
- Normalized fitness: The fitness is always between zero and one.

3.8 Selection and reproduction

We have a way to determine how well a specific individual performs, i.e., fitness, now we need a way to select individuals from the population, decide whether to apply mutation or crossover and whether to keep or replace them in the population. This is the job of the selection operator. A brief summary of some well-known selection methods are given in this section.

- Fitness-Proportional Selection: Each individual has a chance to pass offspring into the next generation. This chance is proportional based on the individuals fitness to the fitness of the whole population. The probability of individual i is:

$$p_i = f_i / \sum_{j=1}^{N} f_j$$

where N is the population size and f_j the fitness of individual j. This method is also called Roulette wheel.

– Tournament: Only a subset of individuals compete for survival. A fixed number, called the tournament size, of individuals is randomly selected. The most fit individuals win, and are allowed to mutate and produce offspring, replacing the least fit individuals of the tournament.

Elitism can be applied when selecting individuals for the next generation. For example one can always put the top 5 of the best unique individuals in the next generation without any alterations by mutations or crossover.

There are also variations of how to manage the population. A brief summary of the most common ones:

– Generational: The creation and selection of individuals is done by distinct generations. The new population is created from the previous generation.

– Steady State: There is a continuous flow of individuals which are selected, mutated and create offspring. The offspring replaces existing individuals in the same population. Commonly the number of generations is counted in the following way: when the number of fitness evaluations is equal to the population size the generation counter is increased by one.

In this thesis we will use the generational algorithm with tournament selection and elitism.

3.9 Building block hypothesis

The building block hypothesis (BBH) was used by Holland [7] to show why genetic algorithms in general are able to solve problems. The BBH has been sharply critized on the grounds that is lacks theoretical justification. We agree with the critism but still think that some basic ideas of the BBH can be helpful when thinking about GP. They give a general idea why GP works and why it can be used to solve very diverse problems. The building block approach also makes one think about mutation, crossover and fitness and how to make the best choices in these matters. So it still can be a useful concept.

The underlying intuition is as follows: Take into account all the generic code of all the individuals in the whole population and assume certain sections are responsible to perform certain tasks. The building blocks are those sections which are needed to solve the problem and by combining building blocks (in the right order) into one individual we get the best possible solution.

So can thinking in building blocks help? The initial population will certainly contain some useful building blocks. But it is not likely that all building blocks are present, so we need a way to create or improve building blocks. This is the job of the mutation operator; it is able to create new building blocks or improve the existing ones by changing the underlying genetic code. The crossover operator's job is to combine the building blocks which are already present in the population as a whole into one individual.

So theoretically mutation and crossover operators are able to improve individuals. But keep in mind that very often they will be destructive instead of constructive. The building blocks also need to be placed in the "right" order to be effective. For example, if our goal is to remove dust in the room and we have the building blocks "turn on vacuum cleaner", "move vacuum cleaner around" and "turn off vacuum cleaner" the building blocks need to be executed in that precise order to reach our goal.

One task like "turn on vacuum cleaner" could be stored by using several consecutive instructions. If the crossover operators are not able to recognize the start and end of these instructions they can break a building block in two, thereby destroying it. Hopefully fitness based selection will be able to detect this and with several copies in the population the building block is not lost. Mutation can also destroy or deteriorate a building block and again we place our hope in fitness based selection to minimize the damage.

Up front it is very difficult to detect the building blocks and award the individuals containing building blocks a high fitness. Building blocks tend to be genetically available but situated at a wrong place in the genetic context.

When building blocks are used at a wrong "time" or "place" it will influence the fitness of an individual in a negative way. The fitness function is likely to only award building blocks which are also placed at the right genetic context. So even if crossover doesn't destroy a building block by breaking it in two, placing it in a wrong genetic context has the same devastating effect on the fitness of the individual.

Realizing this, the chances of a successful crossover seem very slim. This is why large populations and several evolution restarts are needed to achieve good results. Even if we did our very best to design the best possible fitness function, selection, crossover and mutation operators.

Chapter 4

Delta Algorithms

4.1 Introduction

Delta algorithms are algorithms that compute the difference between two strings or files. Delta's are useful in several ways. The obvious one is to see how much has been changed, what was changed and where the changes took place. Delta's can also be used to construct a new version with only the old version and the delta, or the other way around: reconstruct the old version with the new version and the delta. So using delta's can save storage space or transmission time of new versions. The program *diff* [11] is the best-known program to produce delta's for text files. In this chapter we explain and discuss several algorithms and aspects which are related with finding delta's.

4.2 Longest common subsequence

Instead of finding the difference one can do the opposite: find what is common and use that information to find the difference. One method is to find the longest common subsequence, which is a well-known problem. The longest common subsequence is commonly abbreviated as LCS, but since this can be confused with the longest common *sub-string* we will abbreviate it as LCSeq, respectively LCStr, in this thesis. The LCSeq problem is finding a longest sequence which is a subsequence of all sequences in a set of sequences. A subsequence of some sequence is a new sequence which is formed from the original sequence by deleting some of the elements without disturbing the relative positions of the remaining elements (see the example at the end of this section). The LCSeq problem can also be defined to find *all* maximal length common subsequences, which makes it more complex. The general case of the LCSeq problem, with an arbitrary number of sequences, is NP-hard [5; 6]. In this thesis we will mainly use a set size of two: the original and the candidate sequence. If the set size is two the algorithm running time is $O((r\ +n)\ \log\ n)$, where n is the length of the sequences and r is the total number of ordered pairs of positions at which the two sequences match. Thus the worst case running time is $O(n^2\ \log\ n)$, however with relatively few matching positions $O(n\ \log\ n)$ can be expected [12]. Most algorithms use the dynamic programming approach and have a running time of $O(n\ m)$,

where n is the length of the first sequence, and m is the length of the second sequence.

Note that a subsequence is not a sub-string, a sub-string is a consecutive part of a string while a subsequence does not need to be.

Consider the following example:

```
String 1 = ACTG
String 2 = AACTGG
String 3 = AATGG
String 4 = AATGGG
```

String 1 is a subsequence of string 2, and also a sub-string of string 2.
String 3 is a subsequence of string 2, only the `C` is missing. String 3 is obviously not an sub-string of string 2.
The LCSeq of string 2 and 4 would be `AATGG`, so string 3 is the LCSeq of string 2 and 4.

4.3 Levenshtein distance

The Levenshtein distance (also called the edit-distance) between two strings, is the minimum number of operations needed to transform one string into the other. The allowed operations are: insertion, deletion or substitution of a single character. It is named after Vladimir Levenshtein, who developed an efficient algorithm in 1965 [15]. For example consider:

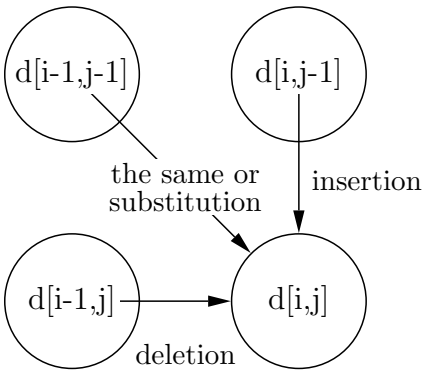
```
String 1 = AACTGG
String 2 = ACTG
```

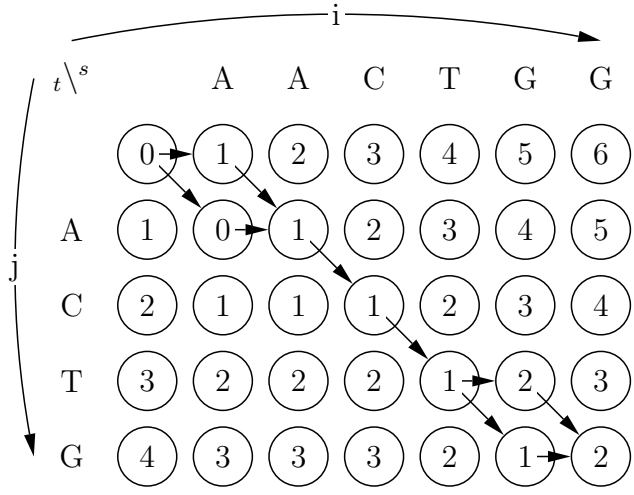
The Levenshtein distance between string 1 and 2 is 2; two deletions will do the trick. Here is a pseudocode to calculate the Levenshtein distance. It is based on the Wagner-Fischer [20] algorithm for edit distance: a bottom-up dynamic programming algorithm which uses an $(n + 1) \times (m + 1)$ matrix, where n and m are the lengths of the two strings. The function has two arguments: string `s` of length m, and string `t` of length n, and it returns the Levenshtein distance between them [23].

```
1  int LevenshteinDistance(char s[1..m], char t[1..n])
2     // d[i,j] = edit distance between s[1..i] and t[1..j]
3     declare int d[0..m, 0..n]
4
5     for i from 0 to m
6         d[i, 0] := i
7     for j from 1 to n
8         d[0, j] := j
9
10    for i from 1 to m
11        for j from 1 to n
12            if s[i] = t[j] then cost := 0
```

```
13                          else cost := 1
14            d[i, j] := minimum(
15                                 d[i-1, j] + 1,     // deletion
16                                 d[i, j-1] + 1,     // insertion
17                                 d[i-1, j-1] + cost // the same or
18                                                    // substitution
19                               )
20
21    return d[m, n]
```

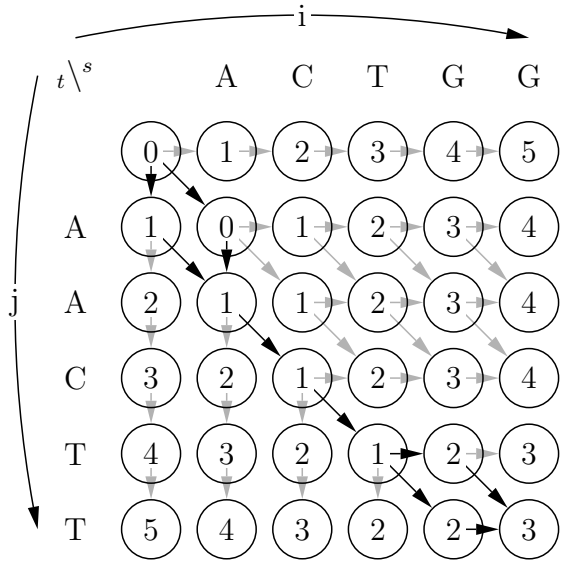
If we write down the matrix `d` in the form of an edit-graph it is much easier to see how the algorithm works. Within the circles the lowest cost is placed.





The four shortest paths represent the four possible minimal edit operations to transform string one into string two:

```
AACTGG   AACTGG   AACTGG   AACTGG
-ACT-G   A-CT-G   -ACTG-   A-CTG-
```



Again the four shortest paths represent the minimal edit operations:

```
-ACTGG  A-CTGG  -ACTGG  A-CTGG
AACT-T  AACT-T  AACTT-  AACTT-
```

4.4 Diff on text files

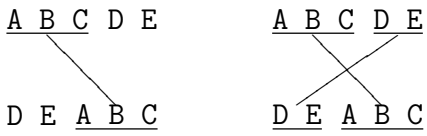
The classic program diff [11] is designed to work with normal text files. To speed up things all the lines of the file are hashed and the hashed values are used to make a LCSeq. It is quite possible that several lines are hashed to the same value so the LCSeq found using the hashed values needs to be checked and repaired. This can be done by comparing the LCSeq found using the hashed values with the original lines and editing out the false equalities. It seems that the LCSeq which is found using this method is quite alright despite repairs, only one case was found by someone and brought to diff's authors attention in two years time, in that case the repaired LCSeq was short by one [11].

The diff algorithm has been improved over time, although the basics are still the same. "An $O(ND)$ Difference Algorithm and it Variations" was written by Eugene W. Myers in 1986 [16]. In this paper some variations are summarized and the paper itself presents an algorithm which is based on an intuitive edit graph formalism. The presented algorithm employs the "greedy" design paradigm unlike others and exposes the relationship of the longest common subsequence to the single-source shortest path problem. This algorithm served as the basis for the new UNIX diff program.

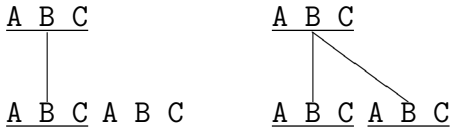
A different approach, which does not use the LCSeq is described in Section 4.5.

4.5 Block move algorithm

In "The string-to-string correction problem with block moves" W.F. Tichy [18] describes why the LCSeq will not always give the shortest edit sequence. We start by giving two examples to illustrate the shortcomings of the LCSeq. Consider the strings $S =$ `ABCDE` and $T =$ `DEABC`. These strings have LCSeq `ABC`. The sub-string `DE` is not part of the LCSeq, a block move would be able to detect `DE` in both strings and thus give a shorter edit script:



Repetition of sub-strings is also something that makes LCSeq based edit scripts inefficient, there is no other solution than a lot of single inserts. A block move based edit script could simply repeat the sub-string. Consider the strings $S =$ `ABC` and $T =$ `ABCABC`:



Block moves could be a lot better than using LCSeq based edit scripts. Let us give the formal definition: Consider two strings $S = S[0, \ldots, n], n \geq 0$ and $T = T[0, \ldots, m], m \geq 0$ over an alphabet A with α symbols. A block move is a triple (p, q, l) such that $S[p, \ldots, p + l - 1] = T[q, \ldots, q + l - 1] (0 \leq p \leq n - l, 0 \leq q \leq m - l + 1, l > 0)$. Thus a block move represents a nonempty common sub-string of S and T with length l, starting at position p in S and position q in T.

4.6 Diff on binary files

Normal diff, which is used on text files, would not suffice as a good Delta algorithm for our problem. Diff uses end of line markers in the text files to make a fast approximation of the LCSeq. We have no end of line markers in our original string, so this poses a problem.

In "Delta algorithms: An empirical analysis" [9] three Delta algorithms are compared: diff, Bdiff and VDelta. Diff is made binary compatible by uuencoding. As expected combining uuencode and diff gives very poor results in comparison to Bdiff and VDelta algorithms. The paper [9] also states that Bdiff and VDelta exploit reordering of blocks to produce short differences, we notice a striking resemblance. In Genetic Programming "building blocks" are used and combined to work toward the solution, so an individual with the right sub-strings but in the wrong order should not be punished, the order of the blocks could be corrected with mutation or crossover. Bdiff and VDelta are both derived from W.F. Tichy's block-move algorithm [18].

4.7 Further reading on delta algorithms

Two more interesting papers which we will not discuss here are: “An Empirical Study of Delta Algorithms” by James J. Hunt and Kiem-Phong Vo and Walter F. Tichy [10] which describes the VDelta algorithm.

“Engineering a Differencing and Compression Data Format” by David G. Korn and Kiem-Phong Vo [14] which describes the VCdiff algorithm.

Chapter 5

Compression

5.1 Introduction

With the aid of data compression algorithms one can represent certain sets of data in such a way that it takes less storage space than the plain straightforward representation. This is particularly useful when storing or transmitting large amounts of data. In this chapter we shall describe some well-known data compression algorithms.

5.2 Lossless vs. lossy compression

Data compression algorithms can be divided into two categories: the ones that exactly reproduce the original and the ones that approximate the original. In most cases lossless compression is needed: without being able to get the exact original back, the compression would be useless. Good examples are computer code and a written essay. But there are several other cases in which small loss is acceptable, for example in case of images and sound. As long as the loss is not noticeable or disturbing there is no problem. With Genetic Programming we will probably create a lossy compression algorithm (see Chapter 6), but we can make it lossless by also storing "the diff", this way the exact original can be reconstructed.

5.3 Run length encoding

Quite a few compression algorithms are based on run length encoding [21]. These algorithms are the most simple form of compression and most logical first step in compression. Consider this example, where the original consists of the following string:

```
AAAAAACCCCCCTTTTTTGGGGGG
```

Run length encoding simply detects repetition and states how many times the repetition occurs. So the string could be represented as:

```
A6C6T6G6
```

This should be interpreted as `A` 6 times, `C` 6 times etc. So the original 24 characters are represented in only 8 characters, this is a 67% compression ratio. The problem with this form of run length encoding is that if, for example, the original consists of alternating characters the compressed string could end up to be much bigger than the original indeed. Consider this example:

```
ACTGACTGACTG
```

If one attempts to compress with run length encoding, one will get:

```
A1C1T1G1A1C1T1G1A1C1T1G1
```

The result is a "compressed" string which is twice as long as the original! This problem can be fixed by altering the encoding algorithm just a little [3]: only use run length encoding if two or more identical characters are repeated, and leave the others untouched. For example, consider:

```
ACTGAACCCTTTTGGGGGAAAAAAAAAAACTG
```

Compressed with the first run length encoder:

```
A1C1T1G1A2C3T4G5A11C1T1G1
```

Compressed with the improved run length encoder:

```
ACTGAA0CC1TT2GG3AA9CTG
```

The original 32 characters get compressed to 24 by our first algorithm, this is a 25% compression ratio. In this example the improved algorithm performs slightly better, using 22 characters which is a 31% compression ratio.

Decompressing the improved version is very easy. If you encounter two of the same characters the third character states the number of repetitions. When we assume all characters are stored as bytes, the maximum number of repetitions we can represent in one character is 255. We need to define what happens when we encounter a sequence which has more repeated characters. `AA255` will decode to 257 `A`'s, but what if we have 500 `A`'s? The most simple solution is `AA255AA241` ($2+255+2+241 = 500$), the run length decoder algorithms stays as it is, while the run length encoder only needs to check for the maximum number of repetitions.

If we look back to the first example, the improved algorithm would perform worse compared to the first algorithm:

```
AA4CC4TT4GG4
```

using 12 characters, or 50%. This is why several variants of run length encoding algorithms emerged, they were adapted to the type of data they were expected to compress.

Run length encoding algorithms perform lossless data compression, but in general these algorithms only achieve good compression ratios if much repetition is involved. Because the encoding and decoding algorithms are very simple and do not require much computing

power or memory, they can easily be built in non sophisticated machines. A lot of patents can be found in which some form of run length encoding is used.

Common examples which use some form of run length encoding are fax machines, tape streamers and audio storage and transmission. Faxed documents are black and white and contain mostly white space, resulting in a lot of repetition of white bits and good run length encoding compression ratio [13]. In the case of tape streamers, audio storage and transmissions, run length encoding combined with delta encoding performs quite nice.

5.4 Minimum redundancy codes

Huffman describes a method in which the most commonly used messages get the shortest code [8]. This way the entire message is represented in the smallest manner. Let us see how this works with the example we already used in the run length encoding section:

```
ACTGAACCCTTTTGGGGGAAAAAAAAAAACTG
```

We start by using the optimum binary code coding procedure, described in [8]. The first step is to determine the character count and calculate the probabilities, the results can be seen in Table 5.1:

Character	Times	Probability
`A`	14	43.750%
`G`	7	21.875%
`T`	6	18.750%
`C`	5	16.625%

Table 5.1: Huffman character count and probabilities.

We want to assign a binary code to each of the characters, the least used ones should get the longest code, the most frequent used ones the smallest code. So we start at the bottom, these are the least common characters, in our case `C` and `T`. `T+C` have a combined percentage of 34.375%, which puts it between `A` and `G`, the result can be seen in Table 5.2:

	Chars	prob.	prob.
5	`A`	43.750%	43.750%
4	`T+C`		34.375%
3	`G`	21.875%	21.875%
2	`T`	18.750%	
1	`C`	15.625%	

Table 5.2: Huffman optimum binary coding procedure, step 1.

Now `G` and `T+C` are the least common, combined they have a percentage of 56.250%, which puts them above `A`, this results in Table 5.3:

	Chars	prob.	prob.
4	`T+C+G`		56.250%
3	`A`	43.750%	43.750%
2	`T+C`	34.375%	
1	`G`	21.875%	

Table 5.3: Huffman optimum binary coding procedure, step 2.

Which only leaves `A` and `T+C+G`, when we put these and the previous steps in one big table, we get Table 5.4:

	Chars	prob.	prob.	prob.	prob.
7	`T+C+G+A`				100.000%
6	`T+C+G`			**56.250%** `0`	
5	`A`	43.750%	43.750%	**43.750%** `1`	
4	`T+C`		**34.375%** `0`		
3	`G`	21.875%	**21.875%** `1`		
2	`T`	**18.750%** `0`			
1	`C`	**15.625%** `1`			

Table 5.4: Huffman optimum binary coding procedure, final result.

From the Table 5.4 we can read the optimum binary code for our characters. `A` gets a `1`, `G` get a `0` from line 6 and a `1` from line 3, which are combined to `01` etc. This leads to Table 5.5 with the optimum binary codes:

Character	Times	Probability	Code
`A`	14	43.750%	`1`
`G`	7	21.875%	`01`
`T`	6	18.750%	`000`
`C`	5	16.625%	`001`

Table 5.5: Huffman optimum binary code.

Our example of 32 characters could be written down in bits. When we use normal binary code we only need 2 bits to enumerate our 4 possible characters, so the example would take $32 \times 2 = 64$ bits of storage space. While with Huffman encoding you get the result which can be seen in Table 5.6:

`A`	`C`	`T`	`G`	`A`	`A`	`C`	`C`	`C`	`T`	`T`	`T`	`T`	`G`	`G`	`G`
1	001	000	01	1	1	001	001	001	000	000	000	000	01	01	01
`G`	`G`	`A`	`A`	`A`	`A`	`A`	`A`	`A`	`A`	`A`	`A`	`A`	`C`	`T`	`G`
01	01	1	1	1	1	1	1	1	1	1	1	1	001	000	01

Table 5.6: Example encoded with Huffman.

Our example encoded with Huffman would be $14 \times 1 + 7 \times 2 + 11 \times 3 = 61$ bits long. This is only 5% compression ratio, not so good! Run length encoding performed much better. But consider this example:

AAGAAGAAGAAGAAGAAGAAGTCTCTCTCTCT

This example has the same number of A's C's T's and G's so Huffman would compress this to 61 bits as well, whereas run length encoding would fail horribly. Normally Huffman encoding is not used on such a small scale, and other factors are far from optimum. With real live situations Huffman performs quite well. Huffman also is an example of a lossless compression algorithm.

5.5 LZ77/LZ78 and LZW

More advanced well-known data compression algorithms are LZ77 en LZ78 [24]. Abraham Lempel and Javob Ziv describe them in papers written in 1977 en 1978, hence the algorithms got known by the names LZ77 and LZ78. Both are lossless data compression algorithms. We have already seen run length encoding and minimum redundancy coding; LZ77 and LZ78 algorithms have a different approach: LZ77 is a "sliding window" compression algorithm, LZ78 uses an explicit dictionary technique.

The LZ77 "sliding window" method is fairly straightforward, it tries to replace a recurring pattern with a "copy command". LZ77 maintains 2 buffers: A history buffer which contains the characters that have been sent to output and a lookahead buffer which contains the characters that must be sent to output. If a match can be made from the lookahead buffer to the history buffer, a copy command is made. The command consists of the offset which indicates where to start to copy and the number of characters to copy. Consider the following example:

ACTGAACCCTTTTGAACCCT

In Table 5.7 we show the steps the LZ77 compression algorithm performs:

Step	Output	History	Lookahead
			ACTGAACCCTTTTGAACCCT
1	A	A	CTGAACCCTTTTGAACCCT
2	C	AC	TGAACCCTTTTGAACCCT
3	T	ACT	GAACCCTTTTGAACCCT
4	G	ACTG	AACCCTTTTGAACCCT
5	A	ACTGA	ACCCTTTTGAACCCT
6	5,2	ACTGAAC	CCTTTTGGAACCCTG
7	C	ACTGAACC	CTTTTGAACCCT
8	7,2	ACTGAACCCT	TTTGGAACCCTG
9	T	ACTGAACCCTT	TTGAACCCT
10	2,2	ACTGAACCCTTTT	GAACCCT
11	10,7	ACTGAACCCTTTTGG$GAACCCT$	

Table 5.7: LZ77 compression.

The beginning is straightforward, in steps 1 through 5 we output an `A`, a `C`, a `T`, a `G` and an `A`. The first repetition is step 6 `AC`, it can be found in the history buffer 5 places back (begin to count at the right and count toward the left) and has length 2. In step 7 a normal `C` is sent to output. Step 8 brings us to the second match; `CT`, 7 places back in the history buffer, also length 2. Step 9; a normal `T`, step 10; the third match `TT` and finally step 11; the fourth match is `GAACCCT` 10 places back in the history buffer with length 7, this gives us following compressed string:

```
A C T G A 2,5 C 2,7 T 2,2 7,10
```

Decompressing is straightforward, we only need the history buffer there is need to set up special data structures. Also notice there is no need to transfer a dictionary: it is made as we go. If the decompress algorithm encounters a normal character it is sent to output, in case of a copy command the sequence is searched for in the history buffer and sent to output. We illustrate the decompression process with Table 5.8:

Step	Input	History	Output
1	A		A
2	C	A	C
3	T	AC	T
4	G	ACT	G
5	A	ACTG	A
6	5,2	ACTGA	AC
7	C	ACTGAAC	C
8	7,2	ACTGAACC	CT
9	T	ACTGAACCCT	T
10	2,2	ACTGAACCCTT	TT
11	10,7	ACTGAACCCTTTT	GAACCT

Table 5.8: LZ77 Decompression.

Steps 1 through 4 are straightforward, and we build up the history buffer. In step 5 the first lookup takes place.

LZ77 also has some shortcomings: the size of the history and lookahead buffer determines the compression efficiency. If for example the history buffer is too small, a repetition will not be seen, if the lookahead buffer is too small, the repetition could have been larger. The bigger the buffers get, the harder it gets to find a match.

LZ77 is used quite often, popular archivers like `arj`, `lha` and `zip` use variations of the LZ77 algorithm.

LZ78 builds a dictionary of all previously seen phrases. The phrases can occur long before the first repetition, in which case LZ77 would have missed it. Another benefit is that the size of the phrase is encoded in the dictionary, so there is no need to store it in the compressed string. A very popular variant of LZ78 is LZW [22]. The algorithm of

LZ78 and LZW differs slightly. The main difference is the manner in which the dictionary is used. LZW starts the standard ASCII set as initial dictionary, while LZ78 starts with nothing. So with LZW we have all the single characters with indexes from 0 to 255 in the initial dictionary. LZW has a fixed length dictionary (mostly 4096), LZ78 has no fixed length and uses (index, next symbol) notation. LZW starts to expand the dictionary as it processes the text, using the entries 256 to 4095 to refer to sub-strings. As a new string is parsed, a new code is generated. Strings for parsing are formed by appending the current character `K` to an existing string `w`. The algorithm for LZW compression is shown below:

```
w = Nil
while not empty input
    read a character K
    if wK exists in the dictionary
        w = wK
    else
        output the code for w
        add wK to dictionary
        w = K
endwhile
output the code for w
```

As the dictionary grows, redundant strings will be coded as a single 2-byte number, resulting in a compressed file.

We will illustrate LZW compression in Table 5.9 with an example which uses the string from previous examples:

ACTGAACCCTTTTGAACCCT

Decompression has the following pseudo code:

```
read a code C
output lookup(C)
w=C
while not empty input
  read a code C
    if C is in dictionary
      output lookup(C)
      add to dictionary (lookup(w)+firstchar(lookup(C)))
      w = C
   else
    os = lookup(w) + firstchar(lookup(w))
    output os
    add to dictionary (os)
    w = C
  }
endwhile
```

In Table 5.10 we show the steps the LZW decompression algorithm performs. Iteration 9 is somewhat interesting, the code is not yet present in the dictionary, the `else` comes into action to create the output string and add the code to the dictionary.

LZW is patented and well know for its use in `GIF` images.

Iteration	w	K	wk exists	w	Output	Dictionary
1	Nil	A	yes	A		
2	A	C	no	C	A	256=AC
3	C	T	no	T	C	257=CT
4	T	G	no	G	T	258=TG
5	G	A	no	A	G	259=GA
6	A	A	no	A	A	260=AA
7	A	C	yes	AC		
8	AC	C	no	C	256	261=ACC
9	C	C	no	C	C	262=CC
10	C	T	yes	CT		
11	CT	T	no	T	257	263=CTT
12	T	T	no	T	T	264=TT
13	T	T	yes	TT		
14	TT	G	no	G	264	265=TTG
15	G	A	yes	GA		
16	GA	A	no	A	259	266=GAA
17	A	C	yes	AC		
18	AC	C	yes	ACC		
19	ACC	C	no	C	261	267=ACCC
20	C	T	yes	CT		
21	CT				257	

Table 5.9: LZW Compression.

Iteration	lookup(w)	C	Output	Dictionary
0		A	A	
1	A	C	C	256=AC
2	C	T	T	257=CT
3	T	G	G	258=TG
4	G	A	A	259=GA
5	A	256	AC	260=AA
6	AC	C	C	261=ACC
7	C	257	CT	262=CC
8	CT	T	T	263=CTT
9	T	264	TT	264=TT
10	TT	259	GA	265=TTG
11	GA	261	ACC	266=GAA
12	ACC	257	CT	267=ACCC

Table 5.10: LZW Decompression.

5.6 The counting theorem

Although compression seems great, not all strings will get smaller when run through a compression algorithm. Solomonoff (1964), Kolmogorov (1965) and Chaitin (1966) were the pioneers in descriptive complexity, in a sense they all were aware that given a language certain strings already had "high information density" and could not be written down shorter.

Simply by counting one can see there is no universal lossless compression algorithm which will compress **all** possible strings. [19]

- Theorem: No program can compress (i.e., store with less bits than the original) without loss *all* strings of size $\geq N$ bits, for any given integer $N \geq 1$.
- Proof: Assume that the program can compress without loss all strings of size $\geq N$ bits. Compress with this program *all* the 2^N strings which have exactly N bits. All compressed strings have at most $N-1$ bits, so there are at most $2^N - 1$ different compressed strings ($2^{(N-1)}$ strings of size $N-1$, $2^{(N-2)}$ of size $N-2$, and so on, down to 1 string of size 0). So at least two different input strings must compress to the same output string. Hence the compression program cannot be lossless.

– For example: $N = 8$. So we have $2^8 = 256$ original strings of 8 bits ranging from 00000000 to 11111111. Our compression program must make compressed strings which are smaller than 8 bits. We can make at most $2^7 = 128$ different strings with a size of 7 bits, $2^6 = 64$ different strings with a size of 6 bits, ..., ..., $2^1 = 2$ different strings with a size of 1 bits, $2^0 = 1$ string with a size of 0 bits. This gives $128+64+32+16+8+4+2+1 = 255$ different compressed strings. This means that at least two original strings must compress to the same compressed string, hence the compression algorithm is not lossless! Decompressing this one compressed string can never give two different decompressed strings.

5.7 Further reading on compression algorithms

The number of articles on compression are vast, we only discussed a few notable ones. One more notable article to read is: "A Block-soring Lossless Data Compression Algorithm" by M. Burrows and D.J. Wheeler [2] which describes a block-sorting, lossless data compression algorithm. The popular name for this algorithm is the Burrows-Wheeler transform and introduces a few novell ideas which we have not discussed in this chapter.

Chapter 6

Implementation

6.1 Implementation of software

In this chapter we get into the details on how the experiments were done. The software we used to run our experiments is ECJ 18, a Java-based Evolutionary Computation Research System [17]. ECJ is feature rich, well documented and easy to expand. Java has the advantage of being machine independent, class based and object oriented. So our work can be extended by others without a total rewrite of code and experiments can be run on any machine without difficult porting problems. The basic required features were available: ECJ had good build-in support for GP, several tree creation algorithms (even multiple tree forest support) and several GP breeding operators. More advanced topics made the decision to use ECJ very easy: Master/slave evaluation support over multiple processors, if needed we can use this to scale up and to speed up run times. Multiple subpopulations with optional exchange and island models are also supported, should we encounter problems with global convergence then these facilities provide ways to counter such problems. Experimenting with the examples provided with ECJ made it easy to get used to parameter files, the Java classes involved and how they were related. We started to write our own GPcompression class files and run some initial tests, this chapter will describe several choices we made. In particular we define our fitness function.

6.2 Representation of individuals

The *binary tree* is the first and most logical representation which comes to mind. Using this idea seemed like a good place to start. One benefit of complete binary trees is that they can be stored in a linear datastructure such as an array, so simple and short storage is possible. A binary tree is complete when every level, except possibly the last, is completely filled, and all nodes are as far left as possible. Let us look at a example of a binary tree with 7 internal nodes N1, N2, ..., N7 and 8 leaves L1, L2, ..., L8:

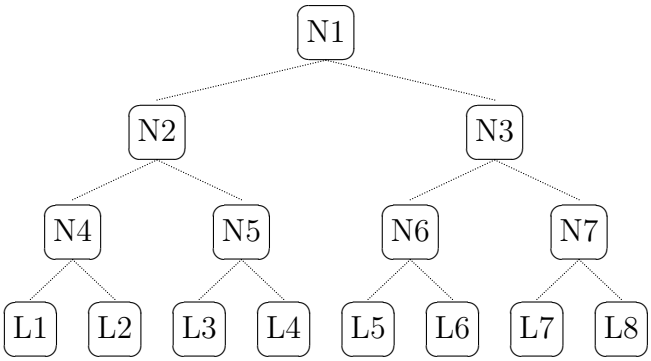


Figure 6.1: Binary tree.

Our tree had to be able to generate a string and we needed a representation which would make the tree capable of generating output with some form of repetition. We decided to store a number in the nodes, which represents the number of times a subtree should be *executed*. The leaves are used store symbols, which are written to the output. We also define a "null"-symbol (denoted as x), meaning "no output". To create the string, the tree is traversed in pre-order (root-left-right) order, where the number in the internal nodes determine how often the left subtree is executed, i.e., its output is concatenated to the final output this number of times.

Optionally we could make the tree always output a string of the desired length, e.g., if the output is too short, we restart the tree again and stop as soon as we have reached the desired length. In the case the output is too long we simply discard the excess output. Soon we realized restarting the tree was a good idea but it made the benchmark problems to easy to solve, so we didn't implement this feature.

To summarize everything mentioned earlier, consider the following example, a tree which will output `AACAACAACTC`:

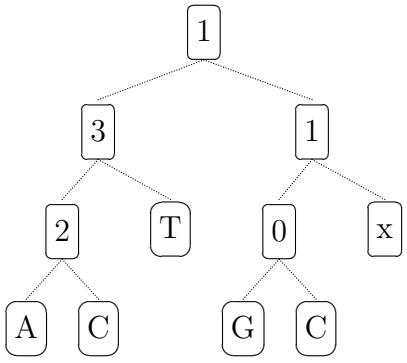


Figure 6.2: Example tree.

6.3 Fitness function

In Section 3.7 we described the important role of a fitness function in the evolutionary process. Even before a single test run was started several ideas on how to define the fitness function came to mind. Several elements we could use in the fitness function, in no particular order, are:

- Count the symbols in the candidate and the desired string and use the number of differences as a penalty.
- The length of the desired string minus the length of the LCSeq between the candidate and the desired string.
- The length of the desired string minus the length of the LCStr between the candidate and the desired string.
- Use diff on the candidate en desired string and count the number of insertions and deletions in the edit script.
- Levenshtein edit distance.
- Hamming distance.

During the first test runs (see Section 6.4) it became clear that using diff in the fitness function worked very well. If we would set our goals on achieving lossless compression we could use the edit script produced by the diff algorithm to transform the candidate string to the desired string. Using diff as main component meant that all candidate strings which were longer than the desired string automatically got a large penalty for the extra length. The extra length was not necessarily a bad thing and we thought it would be advantageous if the evolution process had some place to store useful information which would not be given a fitness penalty. Hence we allowed the candidate string to be longer than the desired string (excess length can be discarded easily). On the other hand the individual trees should not become too large, observed behavior of uncontrolled growth is not wanted as we will see in Section 6.4. We found that an elegant and simple solution was using the number of nodes in the tree as penalty in the fitness function and only give a penalty if the length of the candidate string became twice the length of the desired string.

After several test runs we found the following fitness founction worked best: The fitness function $\mathit{fitness}(x)$ used in this thesis for an individual x with a fixed desired string d is defined as:

$$\begin{aligned} \mathit{fitness}(x) &= 50 \times \mathit{Diff}\,(\mathit{Gen}(x), d) + \mathit{NodesPenalty}(x) \\ &\quad + \begin{cases} |\mathit{Gen}(x) - d| & \text{if } |\mathit{Gen}(x)| > 2 \times |d| \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{6.1}$$

where $\mathit{Gen}(x)$ is the string generated by the tree x, according to the mechanism described in Section 6.2; $|s|$ is the length of string s. Furthermore $\mathit{NodesPenalty}(x) =$

$Max(0, NumberOfNodes(x) - AllowedNumberOfNodes)$ where $NumberOfNodes(x)$ is the number of Nodes in tree x, *AllowedNumberOfNodes* is the number of nodes we allow the tree to contain in order to be able to produce the desired string. This is a number we choose. In simple cases we are able to determine the ideal number of nodes as can be seen in Figure 6.6 and described in Section 6.5.

6.4 First test runs

In this section we show some examples of individuals which appeared while running the first test run and which were used to make some kind of change to the GP.

After running some experiments on the small benchmark problems it became clear the trees grew very large, which is a common problem with GP. Using the number of nodes in the tree as a small penalty in the fitness function solved this problem very well. For example this tree was generated while running experiments with the small benchmark problem 2 from Section 2.2. Keep in mind the desired string is `ACGT` concatenated 25 times. The following tree indeed produces the desired string.

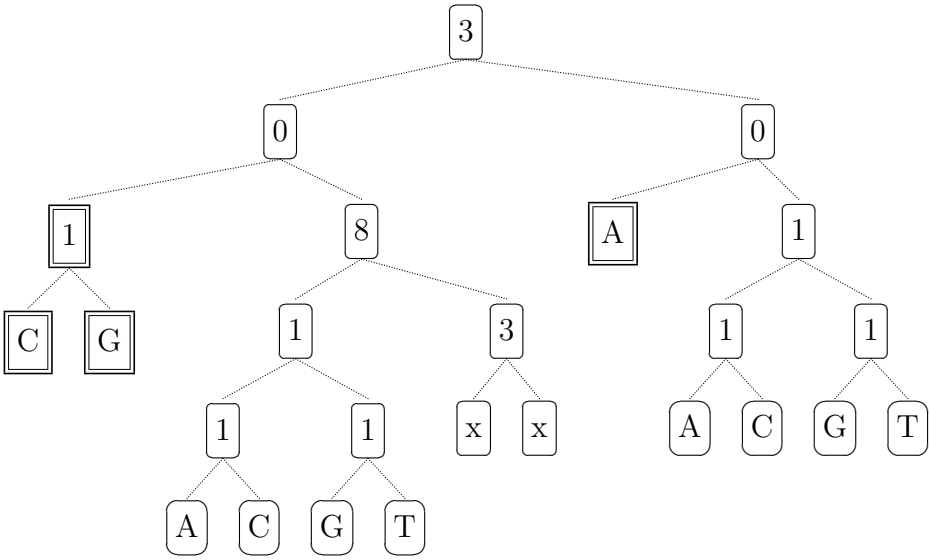


Figure 6.3: Large tree.

When we prune the "not run part" from the tree, i.e., the two marked subtrees with a 0 in their parent node, then one can see that there are still unnecessary doubles in the tree. We also removed unnecessary "null" symbols: four x-es are replaced with one.

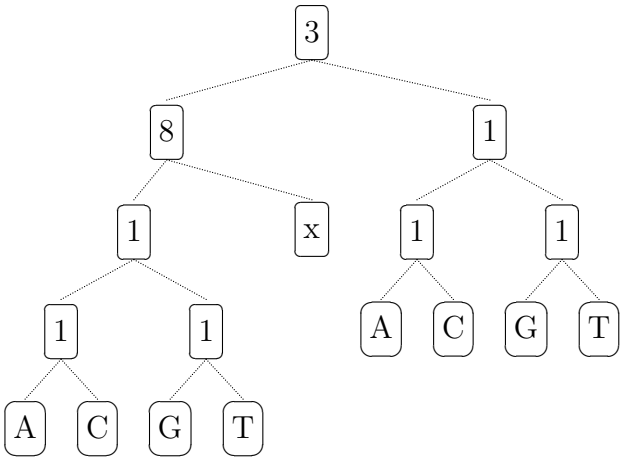


Figure 6.4: Pruned tree.

This pruned tree also raises the question how it could improve. It would for example be much better if the value 8 was replaced with 9, and the right subtree of the root replaced with "null". Mutation until now has been: Select a node and grow a random subtree. This makes it almost impossible to improve the above tree. Point mutation of for example the root and deletion of the right side subtree is needed. This could certainly happen over several generations of evolution provided:

– No (or a small) fitness penalty is imposed for trees that produce excess output.

– Point mutation is possible in nodes.

After these adjustments a better tree was found: It is a perfect match with only 11 nodes, but this tree produces some excess output:

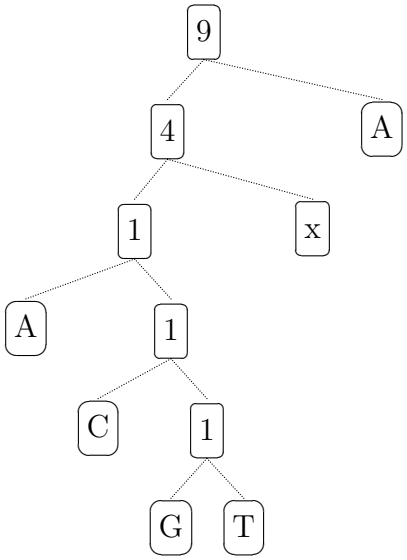


Figure 6.5: Improved tree.

And after some more runs a perfect match with only 11 nodes and exact length was found:

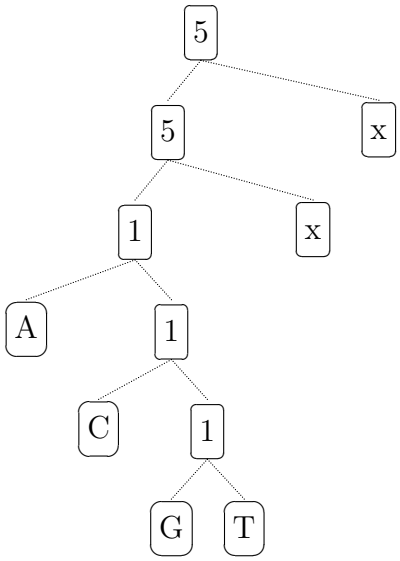


Figure 6.6: Perfect tree.

As another similar example why point mutation in the nodes is needed, consider the following tree: two exact copies of the same subtree are seen here. An increment of one in the root node, and removal of the right subtree, produces the same string.

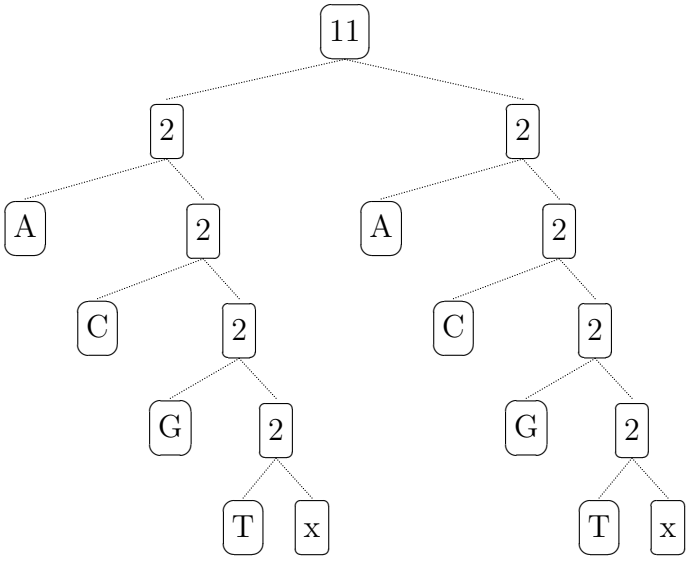


Figure 6.7: Tree with two copied subtrees.

We were curious how the GP would react if the repetition was disturbed on purpose. For a first try we took small benchmark problem 2 and made two modifications. The input

string was changed to 11 times `ACGT` followed by `AAAT` and 13 times `ACGT`. The GP came with a solution which in essence was still the same as the solution found earlier for small benchmark problem 2. The solution is small, but its output is very near to the desired string. This gave us confidence the GP was still able to detect repetition even if there was "fuzziness".

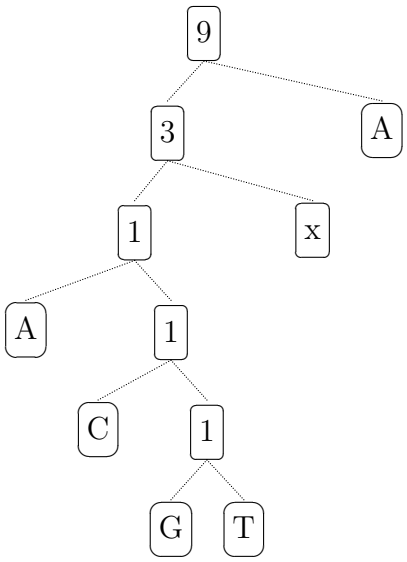


Figure 6.8: Fuzzy tree.

6.5 Stopping criteria

During the initial test runs the GP stopped when it found an individual which had a fitness of zero or reached the maximum number of generations. Introducing the node penalty in the fitness function helped in creating smaller individuals but consequently the fitness function never reached zero. So the GP had lost a stopping criterium, and kept running until the maximum number of generations was reached. One way to repair this is changing the fitness function again and only give a penalty for too many nodes. This meant we had to know the ideal length for the individual, or choose it as best we can, and incorporate this in the fitness function so the fitness could reach zero again.

To determine the absolute minimum number of nodes or length of an individual, we analyze the first benchmark problem which is 100 `A`'s. Consider this tree, which will output 10 `A`'s:

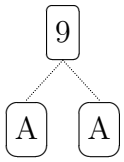


Figure 6.9: 10 `A`'s.

The best way to improve the output is to add another 9 node in the root and a `A`, which will output $10 * 9 + 1 = 91$ `A`'s:

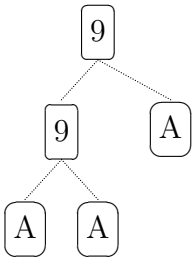


Figure 6.10: 91 `A`'s.

So with 5 nodes we can generate an output of 91 `A`'s, add two nodes and it should be capable to reach the desired length of 100 `A`'s.

The same analysis on the second benchmark problem results in an ideal length of 11 nodes, the third has an ideal length of 15 nodes. With the fourth benchmark problem we observed a minimal length of 37 nodes. An other solution, which we didn't implement, is to monitor improvement and when this is absent for a large number of generations to stop the run.

Chapter 7

Results

In this chapter we describe the results of our experiments. First we present and discuss the solutions our GP found for the benchmark problems we introduced in Section 2.2. Next we present solutions to other bigger problems.

7.1 Solutions for the benchmark problems

For small benchmark problem 1 the GP found following trees:

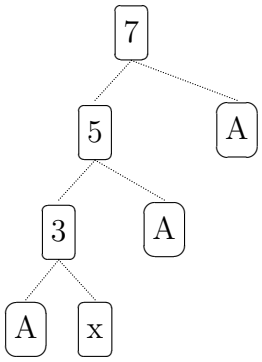


Figure 7.1: Excess output: 113 `A`'s.

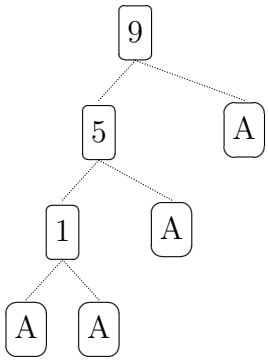


Figure 7.2: Perfect output: 100 `A`'s.

Note that output of the tree seen in Figure 7.1 is too long, but this is not penalized: we tolerate a small amount of excess output, see the fitness function in Section 6.3. The current fitness function scores the two trees with the same value. Also note that these trees are the best possible. The value in the nodes is at most 9, which restricts the possibilities for further improvement. Finally, if we would have allowed repetition, a tree consisting of a single `A` would do the trick.

Benchmark problem 2 was solved very well also, the tree seen in Figure 7.3 creates the string `ACTG` and repeats that string $4 * 7 = 28$ times and appends a `C`. We only needed 25 repetitions, so there is some extra length $(28 - 25) * 4 = 12$ and the single `C` makes 13 characters which are thrown away and will not be given a penalty by the fitness function.

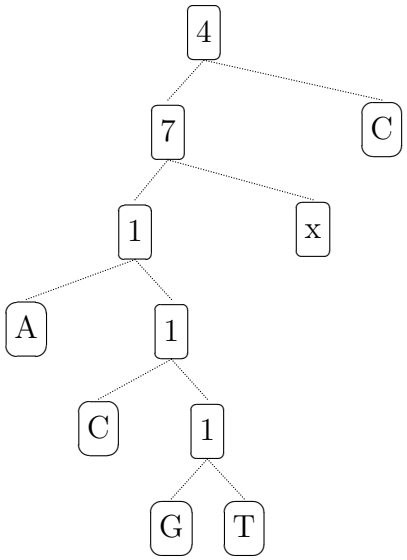


Figure 7.3: Benchmark problem 2 example.

Benchmark problem 3 also posed little of a challenge, the string `AACCGTT` is created quite efficiently and repeated $8 * 2 = 16$ times and followed by a single `T`. Again there is some extra length of $16 * 8 - 100 + 1 = 29$ characters and the single `T` has no ill effect.

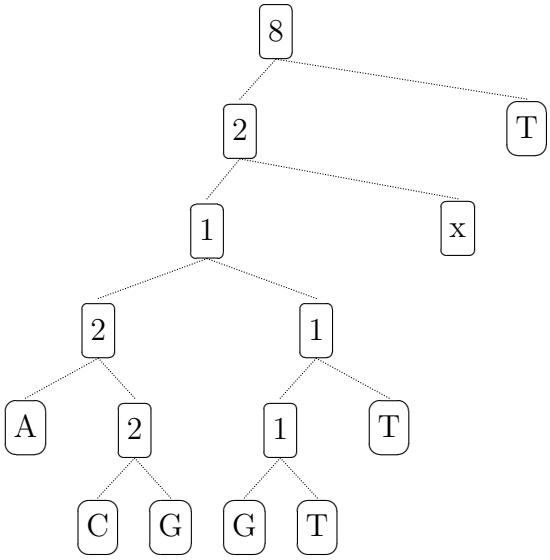


Figure 7.4: Benchmark problem 3 example.

Trying to disturb the GP by replacing the 24'th character with an `A` instead of an `T` had little effect. Still a nice repetition of $3 * 7 = 21$ repetitions of `AACCGGTT` was found in the very straight forward tree which can be seen in Figure 7.5.

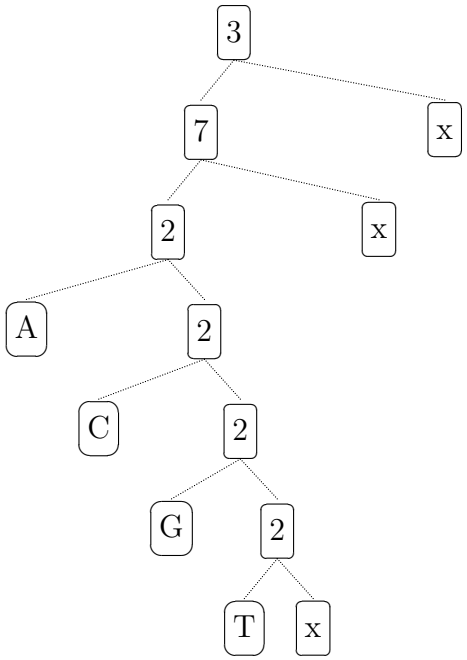


Figure 7.5: Benchmark 3 problem with 1 edit example.

Benchmark problem 4 offered the real first challenge for the GP. A solution was not always found and runtimes increased as can be seen in Section 7.2.

The first solution which looked promising can be seen in Figure 7.6. At that moment the fitness function was still experimented with and looked like this:

```
1 fitness=50*gp_ind.diff;
2 fitness+=gp_ind.numNodespenalty;
3 fitness+=5*(expected.length()-gp_ind.lcstr);
4 fitness+=2*(expected.length()-gp_ind.lcseq);
5 if (expected.length()*2<gp_ind.length_generated_string_tree)
6     fitness+=gp_ind.length_generated_string_tree-expected.length();
```

The candidate string had a diff of 2 a LCSeq of 95 with 103 nodes in the tree.

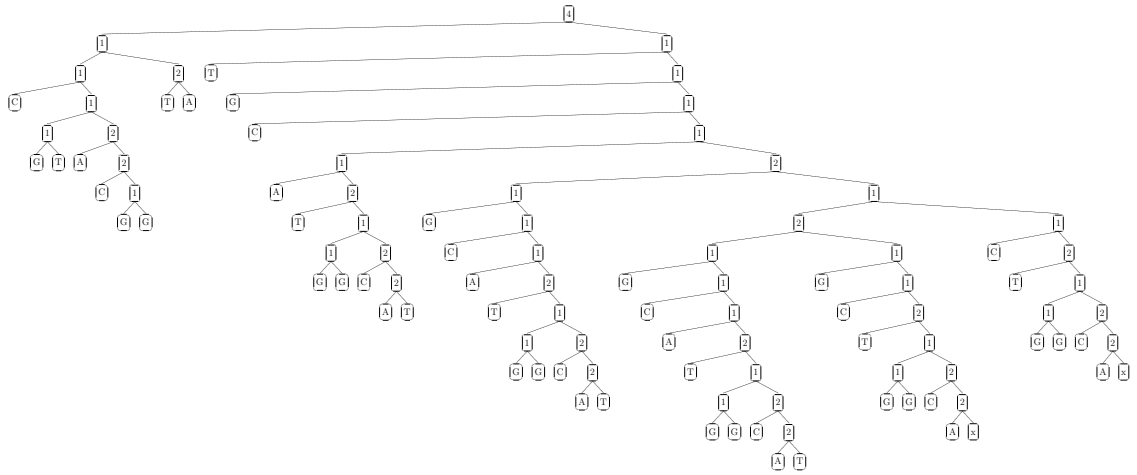


Figure 7.6: Benchmark problem 4 with alternative fitness function.

After several generations the tree improved a lot, the same diff and LCSeq with fewer nodes, as can be seen in Figure 7.7:

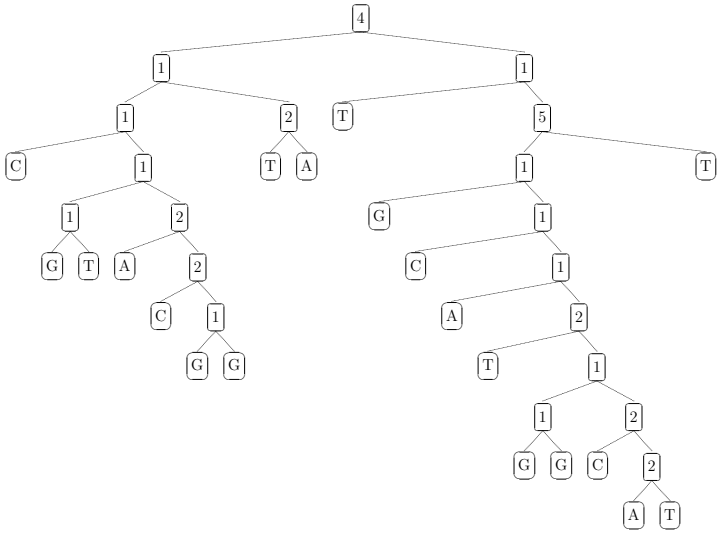


Figure 7.7: After several generations.

After 28,084 generations with a population size of 100 and after several runs, a very good solution was found. The diff is 0, the generated length is 96, which is exact the length we desire. The number of nodes is 37. The question we ask: “Could this solution improve?” After a careful study of the tree and trying several alternatives, we could not create a better solution. Also numerious reruns of the GP yielded no better result. So this indeed seems to be a perfect solution.

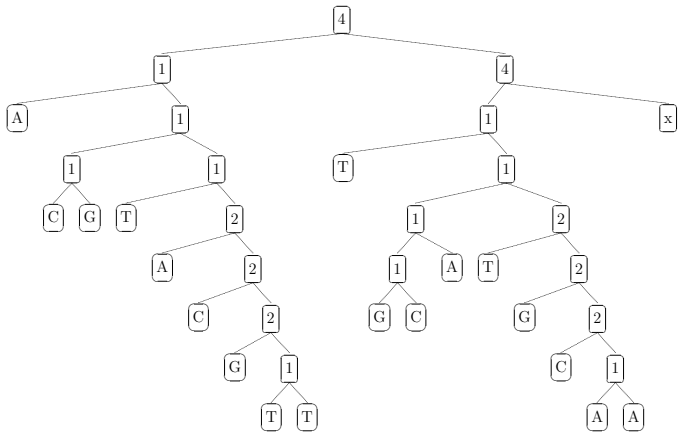


Figure 7.8: Perfect solution for benchmark problem 4.

7.2 Benchmark problem results

The evolutionary framework was run 100 times for each problem using the fitness function defined in Section 6.3. The tables in this section describe the results. In the first column we see the percentage of runs which yielded a diff edit script of length zero, the compression was not lossy. The second column is the percentage of runs which yielded ideal solutions: a diff of zero and the number of nodes in the tree was less than the number we had choosen. Also listed are the lowest, average and highest number of generations to find a ideal solution. Runtimes are based on a Intel(R) Core(TM)2 Duo CPU E6750 @ 2.66GHz with 2048Mb of memory. The runs were single threaded.

Problem	Diff=0 Percentage	Ideal solution Percentage	Generations Minimum	Maximum	Average	Time
1	100%	100%	1	42	7.74	00:01:19
2	92%	45%	24	49,526	5,127.31	11:04:09
3	94%	49%	40	44,896	10,503.61	11:45:24
4	3%	1%	19,888	19,888	19,888.00	22:09:38

Table 7.1: 100 population size, maximum 50,000 generations, elite 5.

Problem	Diff=0 Percentage	Ideal solution Percentage	Generations Minimum	Maximum	Average	Time
1	100%	100%	1	56	8.24	00:01:17
2	98%	55%	12	93,199	14,602.69	19:32:45
3	99%	42%	34	80,056	11,364.05	24:03:39
4	1%	0%				43:40:02

Table 7.2: 100 population size, maximum 100,000 generations, elite 5.

As expected, when an ideal solution is not found the number of generation scales almost linear with the time, which can be seen in Table 7.1 and Table 7.2 by looking at benchmark problem 4. It seems though we had a "lucky" high percentage on benchmark problem 4, Table 7.1 scores much better than Table 7.2. With 1,000 we expect such luck is unlikely to occur, results can be seen in Table 7.3.

Problem	Diff=0 Percentage	Ideal solution Percentage	Generations Minimum	Maximum	Average	Time
1	100%	100%	1	154	9.55	00:03:14
2	93.1%	48.5%	12	49,735	4,763.36	100:04:55
3	93.8%	46.6%	19	49,296	9,771.55	113:48:29
4	0.9%	0.2%	8,366	16,660	12,513.00	214:38:21

Table 7.3: 100 population size, maximum 50,000 generations, elite 5, 1,000 runs.

Choosing a larger population and fewer generations can be interesting, the runtime is much better than those in Table 7.1. The percentages to solve the problem are still quite good and ideal solutions can be found in very few generations as can be seen in Table 7.5. With such a big population one would expect no elitism also reduces runtime considerably (compare Table 7.4 and Table 7.5), but this doesn't seem to be the case. Elitism improves the results and doesn't hurt the time to run much, so we will keep this parameter setting at 5.

Problem	Diff=0 Percentage	Ideal solution Percentage	Generations Minimum	Maximum	Average	Time
1	100%	100%	2	10	4.74	0:02:06
2	72%	24%	13	1,936	299.38	2:58:39
3	66%	29%	19	1,970	773.00	3:19:56
4	0%	0%				4:46:36

Table 7.4: 500 population size, maximum 2,000 generations, elite 0.

Problem	Diff=0 Percentage	Ideal solution Percentage	Generations Minimum	Maximum	Average	Time
1	100%	100%	1	19	4.77	0:02:27
2	78%	32%	13	1,596	297.53	2:55:33
3	65%	32%	19	1,937	460.81	3:04:44
4	0%	0%				5:08:08

Table 7.5: 500 population size, maximum 2,000 generations, elite 5.

If we compare Table 7.2 and Table 7.6 runtimes and results are in the same order of magnitude, so it seems population size and maximum number of generations scale linear with the amount of time and quality of results.

Problem	Diff=0 Percentage	Ideal solution Percentage	Generations Minimum	Maximum	Average	Time
1	100%	100%	1	10	4.48	00:02:07
2	100%	55%	19	19,189	2,922.38	21:02:24
3	96%	44%	18	17,907	1,999.05	21:38:28
4	1%	1%	17,486	17,486	17,486.00	47:15:17

Table 7.6: 500 population size, maximum 20,000 generations, elite 5.

Since benchmark problem 4 is not so easy to solve, we look a bit closer and try to determine which parameter settings are the most effective. The first number is the population size, the second the maximum number of generations. A Q denotes the runs were done an a different processor: an Intel Q9450@3.2GHz with 4096Mb of memory, which is a little faster per core than the E6750@2.66GHz. Diffc means a change is counted as 1 for the diff penalty, not as a deletion and insertion which would be 2.

Problem	Diff min	max	avg	LCSeq min	max	avg	LCStr min	max	avg
100.50000-Q	6	49	22.22	55	93	80.56	5	60	23.95
100.50000-Q-Diffc	0	42	21.24	58	96	79.75	3	96	18.81
500.2000	0	48	23.12	52	96	80.63	5	96	20.75
500.2000-Q-Diffc	7	46	26.36	56	92	75.49	3	48	12.43
Problem	Nodes min	max	avg	Length min	max	avg	perfect	Runs	Time
100.50000-Q	15	57	35.88	61	181	98.57	2	100	16:18:21
100.50000-Q-Diffc	15	55	33.74	71	181	108.18	5	100	18:42:14
500.2000	21	55	35.02	56	189	97.81	4	100	04:16:55
500.2000-Q-Diffc	13	55	30.96	70	182	105.58	5	100	03:50:57

Table 7.7: Comparison of several benchmark problem 4 runs.

The two methods to calculate the penalty do not impact the results of the GP significantly, so we conclude it doesn't matter if you give a change a penalty of two or one in the fitness evaluation.

7.3 A bigger challenge

We have seen the GP can solve our benchmark problems, but these were small and simple problems. Could the GP cope with a bigger problem? In order to test the GP capabilities we concatenated the four benchmark problems to one big string, thus creating a bigger challenge. We will call this desired string benchmark poblem 5. We started the GP with a population size of 500 and allowed 10,000 generations as a maximum per run. Results can be seen in Table 7.8.

Problem	Diff			LCSeq			LCStr		
	min	max	avg	min	max	avg	min	max	avg
Bench5	8	96	63.50	310	392	343.08	99	300	130.96
Problem	Nodes			Length				Runs	Time
	min	max	avg	min	max	avg	perfect		
Bench5	25	109	44.52	314	449	357.31	3	101	245:31:18

Table 7.8: Benchmark problem 5 run overview.

In run 24, after about 2 hours and 25 minutes, generation 8,274 presented a solution with a diff of 8 and a LCSeq of 392. The solution has 107 nodes and generates only 1 extra character; it can be seen in Figure 7.9.

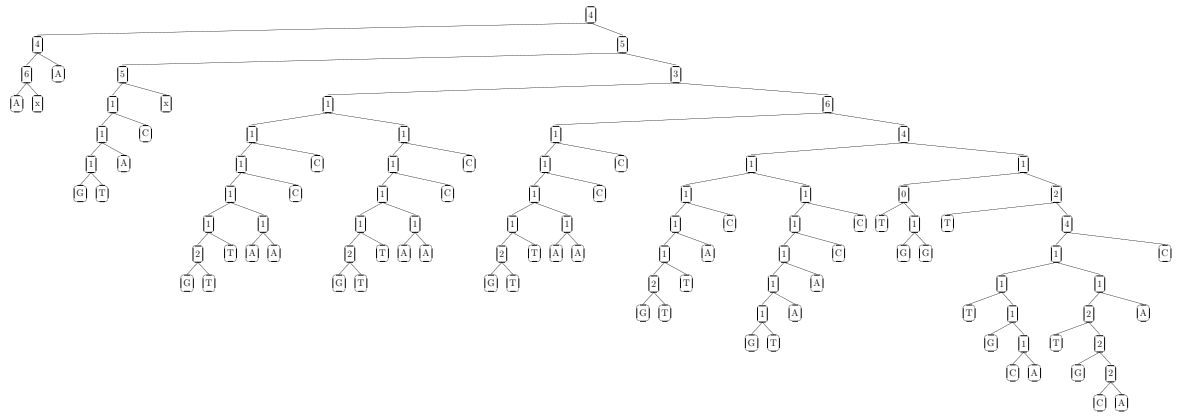


Figure 7.9: Solution for benchmark problem 5, diff=8.

The change script looks like this:

```
101,102d
202,203d
301a ACCA
```

Thus only 4 characters are deleted at position 101 and 102, and again at position 202 and 203. Finally `ACCA` is added at position 301. It is remarkable to see that the diff change script entries operate at the concatenation points of the smaller benchmark problems. These are the points where a new form of repetition starts, the solution the GP found was not perfect but very acceptable. The desired string can be seen in Table 7.9, the output from the tree can be seen in Table 7.10. Characters which are deleted or inserted by diff are underlined, the `C` which is not used in the diff is also made lower case.

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTAC
GTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGT
AACCGGTTAACCGGTTAACCGGTTAACCGGTTAACCGGTTAACCGGTTAA
CCGGTTAACCGGTTAACCGGTTAACCGGTTAACCGGTTAACCGGTTAACC
ACGTAACCGGTTACGTAACCGGTTACGTAACCGGTTACGTAACCGGTTTG
CATTGGCCAATGCATTGGCCAATGCATTGGCCAATGCATTGGCCAA
```

Table 7.9: Desired string for benchmark problem 5.

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAGT
ACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTAC
GTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGGTT
AACCGGTTAACCGGTTAACCGGTTAACCGGTTAACCGGTTAACCGGTTAA
CCGGTTAACCGGTTAACCGGTTAACCGGTTAACCGGTTAACCGGTTAAAC
ACGTAACCGGTTACGTAACCGGTTACGTAACCGGTTACGTAACCGGTTTG
CATTGGCCAATGCATTGGCCAATGCATTGGCCAATGCATTGGCCAAc
```

Table 7.10: String generated by the tree.

Our analysis of run 24 continues, by asking ourselves how many successful steps the evolution took to reach the result. This can be seen in Table 7.11. The first 302 generations the fitness increases fast, we see a steady grow in the number of nodes and a steady decline in the diff. Then there is no improvement for 446 generations, we call this the first stall of the run. These 446 generations were needed to reduce the number of nodes by 2. The second big stall of 1,361 generations, which are needed to lower the diff by 3. The GP enters a fruitful period at generation 3,027, a lot of quick improvements are made, followed by a big stall to lower the diff from 12 to 10. Finally the diff 8 solution is found at generation 8,172. The GP manages to reduce the number of nodes from 111 to 103 over 695 generations. The last 1,141 generations offer no useful changes.

Generation	Fitness	Length	Diff	LCSeq	LCStr	Nodes
1	12028.0	222	240	189	9	27
2	11916.0	246	238	202	6	15
3	11616.0	170	232	167	25	15
4	11314.0	170	226	170	101	13
6	11164.0	173	223	173	101	13
7	10914.0	178	218	178	101	13
9	10216.0	224	204	208	101	15
11	10166.0	225	203	209	17	15
12	8168.0	273	163	253	101	17
14	8068.0	273	161	254	16	17
15	7768.0	245	155	243	91	17
16	7318.0	254	146	252	100	17
17	7172.0	257	143	255	100	21
23	7124.0	260	142	257	100	23
24	7122.0	258	142	256	100	21
27	7074.0	261	141	258	100	23
40	7072.0	261	141	258	100	21
41	7024.0	262	140	259	100	23
48	6980.0	269	139	263	100	29
50	6930.0	270	138	264	100	29
55	6836.0	278	136	269	100	35
58	6786.0	277	135	269	100	35
61	6690.0	305	133	284	100	39
62	6640.0	304	132	284	100	39
63	6436.0	302	128	285	100	35
66	6236.0	294	124	283	100	35
68	6234.0	298	124	285	100	33
69	6136.0	294	122	284	100	35
71	6086.0	295	121	285	100	35
73	6044.0	318	120	297	100	43
74	6038.0	298	120	287	100	37
76	6036.0	294	120	285	100	35
78	5994.0	317	119	297	100	43
79	5990.0	309	119	293	100	39
81	5840.0	306	116	293	100	39
85	5790.0	305	115	293	100	39
90	5640.0	302	112	293	100	39
93	5590.0	301	111	293	100	39
100	5588.0	301	111	293	100	37
105	5392.0	305	107	297	100	41
111	5390.0	305	107	297	100	39
131	3742.0	354	74	338	100	41
137	3692.0	327	73	325	100	41
143	3642.0	326	72	325	100	41
168	3598.0	333	71	329	100	47
177	3550.0	334	70	330	100	49
180	3548.0	326	70	326	100	47
215	3500.0	333	69	330	100	49

Table 7.11: Progress report Part 1.

Generation	Fitness	Length	Diff	LCSeq	LCStr	Nodes
253	3452.0	336	68	332	100	51
258	3402.0	335	67	332	100	51
291	3356.0	338	66	334	100	55
302	3354.0	338	66	334	100	53
748	3352.0	338	66	334	100	51
2109	3212.0	353	63	343	100	61
2114	3062.0	356	60	346	100	61
3027	3038.0	357	59	347	100	87
3029	2888.0	354	56	347	100	87
3034	2886.0	354	56	347	100	85
3037	2788.0	352	54	347	100	87
3038	2746.0	355	53	349	100	95
3040	2742.0	355	53	349	100	91
3041	2734.0	355	53	349	100	83
3042	2652.0	391	51	368	100	101
3044	2646.0	391	51	368	100	95
3045	2514.0	394	48	371	100	113
3046	2504.0	394	48	371	100	103
3047	2496.0	394	48	371	100	95
3048	2378.0	358	46	354	100	77
3050	1904.0	370	36	365	100	103
3054	1902.0	370	36	365	100	101
3055	1754.0	373	33	368	100	103
3057	1748.0	373	33	368	100	97
3059	1728.0	394	32	379	100	127
3061	1576.0	389	29	378	100	125
3063	1574.0	389	29	378	100	123
3064	1570.0	389	29	378	100	119
3067	1566.0	389	29	378	100	115
3069	1424.0	392	26	381	100	123
3073	1422.0	392	26	381	100	121
3075	1418.0	392	26	381	100	117
3076	1414.0	392	26	381	100	113
3079	1412.0	392	26	381	100	111
3081	1408.0	392	26	381	100	107
3082	1310.0	372	24	372	100	109
3086	910.0	380	16	380	100	109
3100	908.0	380	16	380	100	107
3101	906.0	380	16	380	100	105
3108	902.0	380	16	380	100	101
3151	900.0	396	16	388	100	99
3165	700.0	384	12	384	100	99
8164	612.0	557	10	391	100	111
8172	512.0	557	8	392	100	111
8235	510.0	557	8	392	100	109
8255	508.0	397	8	392	100	107
8547	506.0	433	8	392	100	105
8859	504.0	433	8	392	100	103

Table 7.12: Progress report Part 2.

7.4 Real life example: E.Coli

We used the sequence cutter from `http://www.shigen.nig.ac.jp/ecoli/pec/tools.jsp` to get a piece of a DNA sequence from the E.Coli bacteria. First we did a run on the first 500 characters, results can be seen in Figure 7.10 and Table 7.13.

Surprisingly there are a lot of high numbers in the tree, which suggests a lot of repetition in the DNA sequence. But when we look closer at the fitness we see that the diff is very big, 260. And the LCSeq is no where near the 500. It seems the GP is more tuned to deliver a small sized tree than a tree which delivers an accurate output.

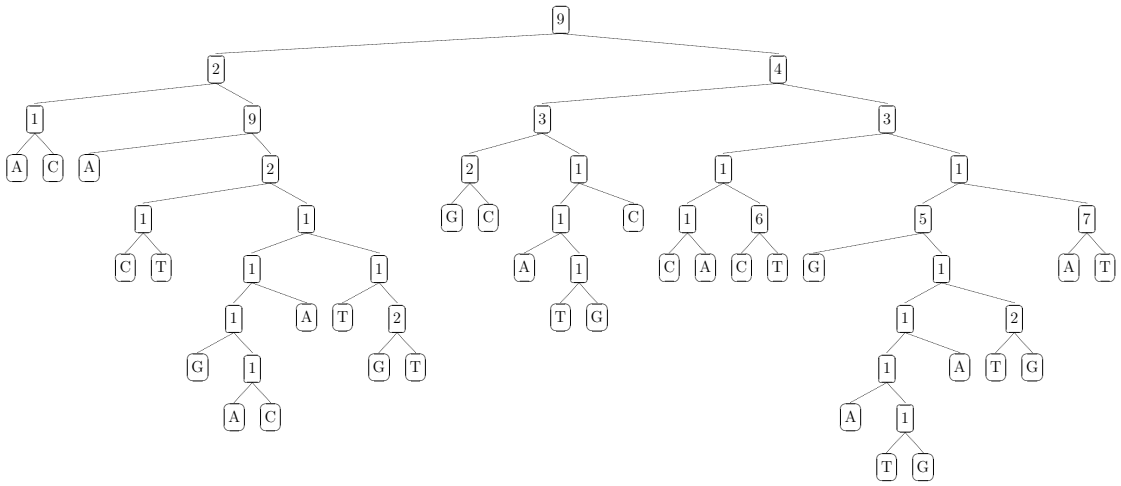


Figure 7.10: 1st 500 characters E.Coli.

Length	Generated Length	Diff	LCSeq	LCStr	Nodes
500	324	260	282	14	61

Table 7.13: First 500 characters E.Coli.

We were curious what would happen when the length was increased, we ran another test on first 5,000 characters, results can be seen in Figure 7.11 and Table 7.14.

Even a smaller tree, and again a big diff. This is more a general pattern recognition than it is compression.

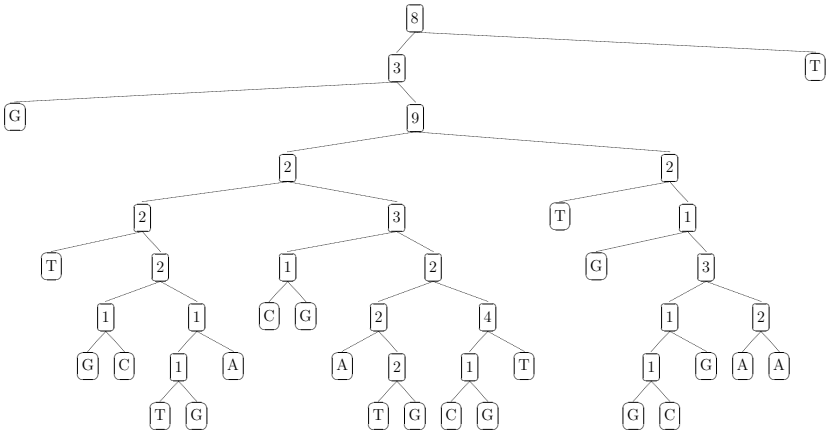


Figure 7.11: First 5,000 characters E.Coli.

Length	Generated Length	Diff	LCSeq	LCStr	Nodes
5,000	3,241	2,935	2,653	10	45

Table 7.14: First 5,000 characters E.Coli.

In order to compare results with the benchmark problems, we analyze the results of 100 runs on the first 100 characters of the sequence of the E.Coli. A population size of 100, and maximal 50,000 generations with elitism of 5 was used. The best individual had a diff of 30 and can been seen in Figure 7.12.

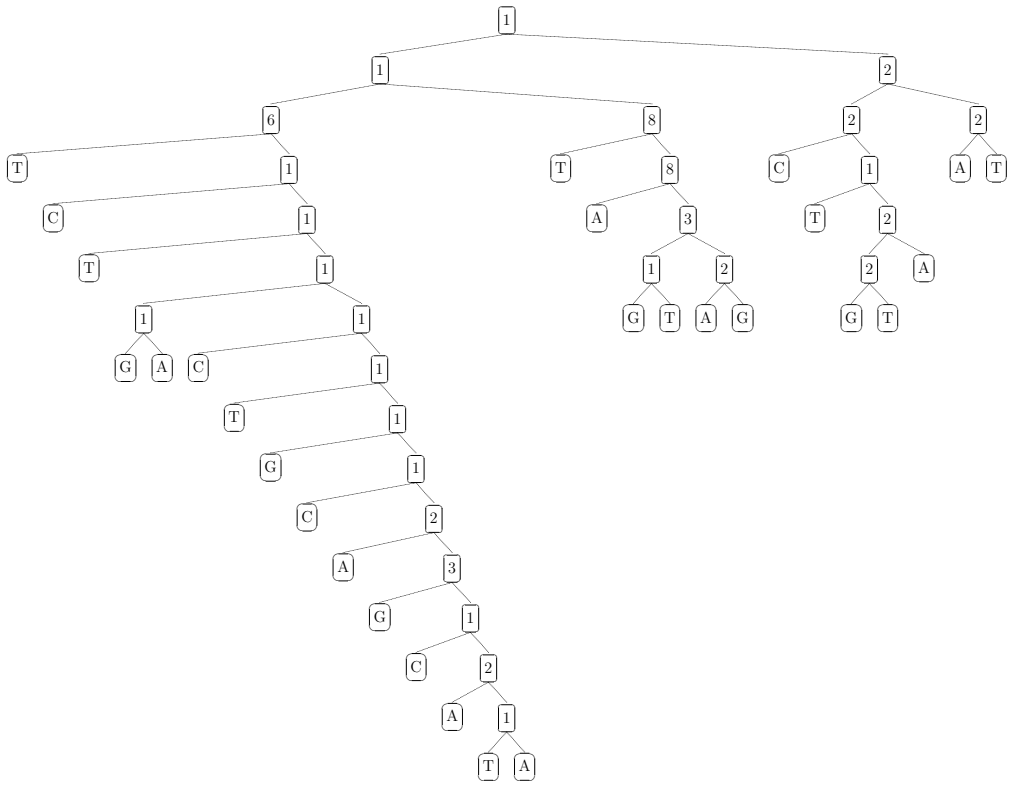


Figure 7.12: First 100 characters E.Coli, best individual of 100 runs.

Problem	Diff			LCSeq			LCStr		
	min	max	avg	min	max	avg	min	max	avg
sequence100	30	56	44.21	49	76	60.98	4	20	8.66
Problem	Nodes			Length				Runs	Time
	min	max	avg	min	max	avg	perfect		
sequence100	13	55	29.74	50	83	66.17	0	100	20:30:31

Table 7.15: First 100 characters E.Coli, results 100 runs.

Looking at Figure 7.12 we suspect that the end of the candidate string matches the desired string very well. This should also be seen in the diff edit script. We analyze the diff edit script further, which has 29 insertions and 1 deletion:

```
The diff edit scipt:
66a A     <- append an A at position 66
63a CC
58a T
54c AAC   <- c means change, so the G is deleted at position 54
50a TT
47a GC
45a A
44a CT
39a G
30a GGA
29a G
28a G
27a C
26a C
25a G
16a C
4a CA
0a AGC
```

We apply this diff edit script to the candidate string in order to visualize the transformation of the candidate string to desired string. Line 1 (`CS`) show the unaltered candidate string, line 2 (`DS`) the desired string. First we position the characters of the candidate string to match those of the desired string by adding spaces, resulting in line 3 (`PS`). The character postitions have also been numbered, which can be seen in the lines 5 and 6. As the last step the diff edit script is executed, which results in the line 4 (`Ed`)

```
CS :TTTTTTCTGACTGCAAGGGCAATATTTTTTTTAAAAAAAAGTGTGTAAGCCTGGTGGTACCTGGTGGTAAAT
DS :AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTAAAT
PS :   TTTT  TTCTGACTGCAA GGGCAATAT T T T T T   TTAAAAAAA AGTGT  G TA  AGC  CTG  GTGGT ACCTG  GTG GTAAAT
Ed :AGC    CA            C         G C C G G GGA         G     CT A  GC   TT   AAC    T     CC   A
10 :   0000  000001111111 111222222 2 2 2 2 3   333333333 44444  4 44  445  555  55555 56666  666 666777
01 :   1234  567890123456 789012345 6 7 8 9 0   123456789 01234  5 67  890  123  45678 90123  456 789012
```

Figure 7.12 shows onlytwo subtrees which are repeated, all the other repetitions are single characters. `GT` is repeated 3 times at position 43. (after the 8 `A`'s.) and `CCTGGTGGTA` is repeated 2 times at position 50. The diff edit script makes changes throughout the string, even these two found repetions are in need of some repairs. The GP simply can't detect enough repetition.

7.5 Speed optimization

The time needed to get usable results is very long. In this section we try some simple speed optimizations in order to see how we could improve the runtime. First we do a run without speed optimization; results can be seen in Table 7.16.

Problem	Diff			LCSeq			LCStr			Nodes			Length				Runs	Time
	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg	perfect		
1	0	0	0	100	100	100	100	100	100	7	7	7	100	199	139.82	5	100	0:01:19
2	0	25	2.73	75	100	97.52	3	100	89.3	9	27	12.82	88	195	129.43	4	100	0:35:22
3	0	36	6.16	64	100	94.41	3	100	66.59	11	29	18.24	82	196	122.43	6	100	0:47:39
4	15	46	30.12	52	86	71.44	2	46	8.97	11	47	31.02	59	183	99.13	4	100	0:57:53

Table 7.16: Run with pop.size 100, 2,000 generations, no speed optimization.

The subroutines which calculate the diff, LCSeq and LCStr are executed for every individual at every generation in order to provide statistics and provide information to calculate the fitness. Currently the diff subroutine is not multithread safe, so we are unable to utilize the multithread evaluation option of ECJ. If the fitness function will be based only on the LCSeq or LCStr, multithreading of ECJ could be enabled. Furthermore if we choose not to calculate the diff and LCSeq and LCStr but only the one we need to determine the fitness a lot of unnecessary calculations are not performed.

We change the fitness function and substitute the diff penalty for a LCSeq penalty. Now only the LCSeq needs to be calculated and the ECJ multithread evaluation parameter can be turned on; results can be seen in Table 7.17.

Problem	Diff			LCSeq			LCStr			Nodes			Length				Runs	Time
	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg	perfect		
1Q				100	100	100				7	7	7	100	199	140.49	5	100	0:00:31
2Q				98	100	99.29				11	29	12.46	101	196	134.86	0	100	0:11:21
3Q				92	100	99.04				13	31	17.8	99	197	136.88	3	100	0:12:39
4Q				64	93	78.65				13	47	32.48	95	192	126.05	5	100	0:17:54

Table 7.17: Run with pop.size 100, 2,000 generations, with speed optimization, LCSeq.

The fitness function is changed again and we use the LCStr as main penalty; results can be seen in Table 7.18.

Problem	Diff			LCSeq			LCStr			Nodes			Length				Runs	Time
	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg	perfect		
1Q							100	100	100	7	7	7	100	200	131.14	11	100	0:00:28
2Q							97	100	99.3	11	31	16.28	100	197	131.08	6	100	0:06:40
3Q							96	100	99.09	13	35	21.44	99	195	128.16	1	100	0:07:43
4Q							13	96	46.72	19	119	41.44	39	184	104.06	2	100	0:11:58

Table 7.18: Run with pop.size 100, 2,000 generations, with speed optimization, LCStr.

If we compare Table 7.16 with Table 7.17 and Table 7.18 we see improved runtimes and better results.

Chapter 8

Conclusions

8.1 Conclusions

In this thesis we examined how Genetic Programming (GP) can be used for the compression of relatively small DNA strings. GP will work as long as a representation of the individuals allows to recreate the original. Choosing a good representation is hard. The current one is loosely inspired by run length encoding, a more sophisticated representation will likely perform better.

When choosing a fitness function one must consider the computational cost very carefully. A sophisticated fitness function which seems much better but takes far more computations than a simple fitness function, could increase the total runtime to find good results.

Although it is possible to achieve compression on simple strings with GP the current implementation is not applicable for full length DNA strings found in real organisms. The amount of time it takes to get results is bad compared to the traditional compression algorithms. As expected the GP will find repetition of patterns even if we disturb them on purpose, but current representation of the individuals can not handle large strings and one can say, based on our results, the GP is better at fuzzy repetition pattern recognition than compression.

8.2 Future research

There are many ideas for further research:

- Split the input string into chunks and use a forest of trees, one tree per chunk, perhaps with crossover between different trees.
- Implement some kind of goto operator. Perhaps combine with forests.
- Use the block moves instead of the classic diff edit script length in the fitness function.

- Create genetic operators which are based on block moves like a swap subtree mutation operator.
- Add a new stop criterium: if no improvement for x generations, then stop this run.
- Speed up the runs by using further multithreading or island models over TCP/IP.
- Optimize source code and calculate less statistics.

One other possible use for GP on DNA strings would be to try to simulate how the DNA strings were formed. If all the genetic operators would perform like in real nature likely ancestors of certain DNA strings could be identified.

Chapter 9

Acknowledgements

I really appreciate the assistance of Dr. W.A. Kosters. I wrote this thesis part time and his immeasurable patience was really without end. He kept on motivating me, when I suffered from writer's block, without him I very likely would not have succeeded. He has in-depth knowledge on several subjects and I could always talk with him or brainstorm about possible solutions and subject to research. Walter thanks a lot!.

Also I would like to thank Dr. M.T.M. Emmerich, his fresh view and questions gave some new perspectives. Thanks to Willemijn de Boer, I practiced my presentation with her as an audiance, which helped a lot and I got some useful tips. Finally I would like to thank my family and all my friends for their support and proof reading, in particular Ben Brouwer, my brother Jeroen and my parents Gerard en Lenie.

References

[1] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming, An Introduction*. Morgan Kaufmann Publishers, Inc., 1998.

[2] Michael Burrows and David Wheeler. A Block-sorting Lossless Data Compression Algorithm. Technical report, Digital System Research Center, Pola Alto, California, 1994.

[3] Arturo S.E. Campos. Run Length Encoding. Webpage/Whitepaper, 1999. `http://www.arturocampos.com/ac_rle.html`.

[4] Charles Robert Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. John Murray, 1859.

[5] David Maier. The Complexity of Some Problems on Subsequences and Supersequences. *Journal of the ACM*, 25(2):322–336, 1978.

[6] Michael R. Gary and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[7] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[8] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the I.R.E.*, pages 1098–1101, 1952.

[9] James J. Hunt and Walter F. Tichy. Delta Algorithms: An Empirical Analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2):192–214, 1998.

[10] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. An Empirical Study of Delta Algorithms. *Proceedings of the SCM-6 Workshop on System Configuration Management*, pages 49–66, 1996.

[11] J.W. Hunt and M.D. McIllroy. An Algorithm for Differential File Comparison. Technical Report 41, Bell Laboratories, Murray Hill, New Jersey, 1976.

[12] J.W. Hunt and T.G Szymanski. A Fast Algorithm for Computing Longest Common Subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[13] Alan R. Katz. Decoding Facsimile Data from the Rapicom 450. *RFC798*, 1981. `http://www.apps.ietf.org/rfc/rfc798.html`.

[14] David G. Korn and Kiem-Phong Vo. Engineering a differencing and compression data format. *Proceedings of the General Track: 2002 USENIX Annual Techical Conference Paper*, pages 219–228, 2002.

[15] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[16] Eugene W. Myers. An $O(ND)$ Difference Algorithm and Its Variations. *Algorithmica*, 1(2):251–266, 1986.

[17] Sean Luke, Liviu Panait, Gabriel Balan, Sean Paus, Zbigniew Skolicki, Elena Popovici, Joseph Harrison, Jeff Bassett, Robert Hubley and Alexander Chircop. ECJ, A Java-based Evolutionary Computation Research System. `http://cs.gmu.edu/~eclab/projects/ecj/`.

[18] Walter F. Tichy. The String-to-String Correction Problem with Block Moves. *ACM Transactions on Computer Systems*, 7(2):309–321, 1984.

[19] Usenet comp.compression Frequently Asked Questions. Compression of Random Data. `http://www.faqs.org/faqs/compression-faq/part1/index.html`.

[20] Robert A. Wagner and Michael J. Fischer. The String-to-String Correction Problem. *Journal of the ACM(JACM)*, 21(1):168–173, 1974.

[21] D. R. Weber. An Adaptive Run Length Encoding Algorithm. In *International Conference on Communications, ICC-75, IEEE*. San Francisco, California, 1975.

[22] Terry Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, pages 8–19, 1984.

[23] Wikipedia. Levenshtein Distance. Webpage, [retrieved May 31, 2010]. `http://en.wikipedia.org/ wiki/Levenshtein_distance`.

[24] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, 1977.