Slowing *Concorde*.

Using evolutionary algorithms to perform computation-based analysis of combinatorial optimisation algorithms.

Farhan M. S. Ahammed (3004154)

October 30, 2007

A thesis submitted in fulfillment of the requirements
for the degree of Bachelor of Engineering in Software Engineering at
The University of Newcastle, Australia.

Abstract

This report details the final year project undertaken by Farhan Ahammed as required to complete the Bachelor of Engineering (Software) degree at The University Of Newcastle, Australia.

The aim of this project is to assess the technique of computationally analysing a program by searching for instances which causes the program to run in its worst-case time. *Concorde* is the program used to test this technique in this project. *Concorde* is a Traveling Salesperson Problem solver and currently holds some of the world records of the time taken to solve certain instances. This project takes a further step and attempts to find *classes* of instances which are relatively difficult for *Concorde* to solve.

Each class of instances are described by fractals and these fractals are found by evolving a set of fractals using the evolutionary algorithm approach. The fractals themselves are represented by iterated function systems and L-Systems. Determining the fitness of a 'truncated' fractal involves creating instances of various sizes defined by the fractal, then computing a function $t(n) = n^{\alpha}$ where n is the size of an instance and $t(n)$ is the most time required by *Concorde* to solve the instance of size n. Thus, $t(n)$ is the least upper bound of the time required to solve the instances. The value of α is used in the calculation of the fitness.

Two evolutionary algorithm programs have been created, one to evolve instances of iterated function systems, the other to evolve instances of L-Systems. The results of the testing, and all decisions made during testing, are presented here. The results are compared to the average case and this report shows that some fractals do genenerate instances that take longer to solve than on average.

In addition, it is also shown that when certain instances generated by the fractals are modified slightly, by moving each city to another place close

to its original position, the instance becomes harder to solve. These changes improved more instances generated by iterated function systems than those generated by L-Systems.

Contributions

The author has made the following contributions toward the completion of this project.

1. Designed and implemented the evolutionary algorithms described in this report to find classes of instances that are 'difficult' for *Concorde* to solve.

2. Found the classes of instances described in this report, which are described by iterated function systems and L-Systems (fractals).

3. Designed and implemented the algorithm for computing an upper bound estimate on an arbitrary set of points in $\mathbb{R}^2$.

4. Computed an upper bound on the time required for *Concorde* to solve instances of various sizes from the classes of difficult instances (described by fractals) that were found.

5. Implemented the algorithms to plot the attractor (fractal) of an iterated function system described in this report.

6. Performed all the tests and experiments mentioned in this report. This includes running the evolutionary algorithms, further improvement of the evolutionary algorithms to improve the results and modifying the instances (to see if they become more difficult). Interpretation of the results and conclusions are the author's own.

The Author	The Supervisor
(Farhan Ahammed)	(Assoc. Prof. Pablo Moscato)

Acknowledgements

The author would firstly like to thank his supervisor Associate Professor Pablo Moscato whose guidance and advice has been invaluable. Also, he would like to thank the other staff members at the University of Newcastle that always provided help and general advice over the years.

The author would also like to thank his friends that helped him through school and university to survive what appeared to be a countless number of assignments, tests and hours of study.

Most of the thanks must go towards his family for their exceptional, endless support during the school and undergraduate years and for helping him whenever he needed it (even when he did not realise it).

Preface

This thesis describes the author's research on the potential of employing an evolutionary approach to empirically analyse an arbitrary program. This is a relatively new approach made possible as the speed at which computers can process its instructions continue to increase.

Concorde, a *Traveling Salesperson Problem* solver is the chosen program to test this new technique on. This report will explain how the evolutionary algorithms were designed, what results were obtained and a discussion on the findings and possible advances in the future.

This report is separated into the following sections:

Chapter 1: Introduction. A brief explanation of the situation that is being addressed, what problems this project is attempting to resolve and a literature review is presented here.

Chapter 2: Background Technical Information. Before this project could proceed, knowledge of some background technical information was necessary. An understanding of what exactly are evolutionary algorithms and fractals (and how to model them using Iterated Function Systems and L-Systems) was required. Chapter 2 summarises this required information.

Chapter 3: Implementation. Once a graph of the time taken to solve instances of various sizes is created, a least upper bound must be computed. This upper bound is needed to help determine the fitness of a fractal and estimate the time-complexity of *Concorde*. The exact fitness function used is described in this chapter.

Two evolutionary algorithms were created. One to develop iterated function systems and the other to develop L-Systems. The design of these algorithms are explained in this chapter. This includes the

fitness function and the rules for selecting parents, creating children and mutating children.

Another program was also created to generate random instances for *Concorde* to solve for estimating the average running time *Concorde* requires to solve Traveling Salesperson Problem instances of various sizes.

Chapter 4: Results. The results of some the tests performed is summarised in this chapter. During the project not all test results were favourable and thus resulting in changes being made to the programs and test conditions. This chapter will explain the different test conditions used and the reasoning behind the decisions made to modify the programs.

Chapter 5: Analysis of Results While the previous chapter lists the results (the fractals that were found) of the tests, this chapter will look at how the fractals behave in an extended situation different from what they were subjected to during the evolution stages of the evolution algorithm. Another hypothesis is also tested on those fractals and the results of those tests are presented here. It is hypothesized that the instances created by fractals can be modified slightly to make them more difficult to solve.

Chapter 6: Conclusions. An interpretation of the results is discussed here including all conclusions that have been made.

Chapter 7: Reflections and Further Work. This chapter includes a reflection on the validity of the results and the project in general and a discussion of possible work/changes that can be conducted in the future by anyone wishing to continue this work.

Appendix A: Design of the *Concorde* program. *Concorde* is the program/algorithm being analysed and it attempts to solve Traveling Salesperson Problem instances. This appendix gives a brief explanation of the algorithm this program implements and what the Traveling Salesperson Problem is.

Appendix B: Plotting the attractor of an IFS. An algorithm was implemented for plotting the attractor of an iterated function system.

This chapter explains how the actual plots described by an iterated function system (fractal) are created and an alternative method of plotting iterated function systems is also provided.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Suppose that an air traffic control system is totally computer controlled. It is quite possible that the algorithm is so complex that a rigorous theoretical derivation of the time-complexity of the control system is extremely difficult to perform. It would be of great benefit to the air-traffic controllers if there was some way to describe the situations (e.g. the number of planes and their position at any point in time) in which the computer system will struggle to perform at satisfactory standards (i.e. when the system takes longer to make the appropriate decisions). In these situations, the air-traffic controllers can take control and manually guide the planes to the runways instead of waiting (for a long time) for the program to make its decision.

For this to work, there needs to be some way to find instances (input configurations) for the computer system which makes the system take longer than usual to come to the necessary conclusions. This project is interested in finding these 'difficult' instances.

1.1 Previous Work

In 2003, Cotta and Moscato proposed an evolutionary computation-based attack of algorithms [2]. The evolutionary algorithm tries to evolve instances of a given problem, such that the algorithm (or its software implementation) requires a lot of steps (time) to produce the correct answer. The idea was tested on the sorting problem, where the evolutionary computation method had evolved difficult instances for certain sorting algorithms (e.g. Bubble Sort and Shell Sort) which have already been studied in depth theoretically

and experimentally.

They found that for any problem of finite-size, their analysis was able to provide a useful lower-bound on the worst-case complexity of the algorithms they analysed and that possibly this mixed evolutionary-statistical analysis can provide a positive contribution, when other approaches of analysing algorithms constitute a hard task for the researcher.

Jano I. van Hemert [3] has also used evolutionary algorithms as an aid to finding weaknesses in combinatorial optimisation algorithms. Hemert tested his technique on the binary constraint satisfaction, boolean satisfiability, and the traveling salesperson combinatorial problems.

When analysing his results of the analysis of the traveling salesperson problem, Hemert looked at the distribution of path lengths in order to explain the difficulty of the instances.

The problem solvers used by Hemert were two variants of the Lin-Kernighan algorithm [4]: the *Chained Lin-Kernighan* which combines multiple results of the Lin-Kernighan algorithm and the *Lin-Kernighan with Cluster Compensation* which uses the cluster distance between nodes in the decision making process to avoid less useful search spaces [3].

Hemert found that problems which were difficult for the Chained Lin-Kernighan algorithm to solve contained clusters and have suggested that the position of clusters and the distribution of cities over clusters are important properties which separate the difficult from the easier instances.

Moscato and Norman have shown it possible to generate arbitrary large instances of the traveling salesperson problem and that optimal solutions of these instances described by fractals can be found a priori [5].

1.2 Proposed Work

Evolutionary algorithms will be used to generate traveling salesperson problem instances. Each instance will be described using *fractals* and *Concorde* is the program used to test the technique presented in this report.

Concorde, written by David Applegate, Robert Bixby, V. Chvatal and William Cook [6] is an algorithm which attempts to solve the *Traveling Salesperson Problem* (TSP) and has provided some of the world records in the exact solution of TSP instances [7]. However, its running times are only partially correlated with the instance size. This suggests that it may be

possible to find small, or very small instances for which *Concorde* requires a lot of CPU time to find the optimal solution.

Fractals will be used to define the positions of each city in a TSP instance. This way, it is possible to describe an instance's *structure*. This structure is used to explain the difficulty of the instances.

This project will use the techniques suggested by Cotta and Moscato to attempt to produce fractals which generates relatively 'difficult-to-solve' instances for *Concorde* (i.e. instances that take longer than on average to solve).

Chapter 2

Background Technical Information

2.1 Introduction

In order to successfully complete this project, an *Evolutionary Algorithm* had to be implemented which would evolve instances for the *Concorde* algorithm that would be difficult for it to solve. The instances are constructed as *fractals* and the fractals themselves are described using *iterated function systems* and *L-Systems*. This section will explain the meaning of each term presented here.

A detailed description on the TSP and *Concorde* will not be given since this project is more focused on using evolutionary algorithms (Section 2.3) to create instances for the *Concorde* program. Appendix B has brief explanation on how *Concorde* works.

2.2 Fractals

There are many ways to represent a set of (x, y) points, including just having a set of all points. Finding a set of points is not enough for this project. What is required is a method of describing the pattern of instances that are difficult for *Concorde* to solve. Fractals provide a way to do just this. Theoretically, fractals can describe any set of points and by definition, they represent each set of points by a recursive definition.

2.2.1 Fractal Basics

Fractals were first discovered by Benoit Mandelbrot in 1975 for describing shapes that look the same even when zoomed-in at different levels of zoom. The word 'fractal' was coined by Mandelbrot in his fundamental essay from the Latin word *fractus*, meaning broken, to describe objects that were too irregular to fit into a traditional geometrical setting [8]. Mandelbrot describes fractals as

> "A rough or fragmented geometric shape that can be subdivided in parts, each of which is (at least approximately) a reduced copy of the whole" [9].

In the past, mathematicians worked with the area of mathematics where the theory of calculus can be applied [8] and avoided work on mathematical objects that were not smooth or regular. This attitude has been changing in recent years as people now recognise that nature is not always modelled like this. There have been discoveries recently of fractals appearing in nature. For example, some flowers and trees can be modelled using fractals [10].

These shapes are commonly described as 'self-similar.' Figure 2.1 shows an example of what it means for an object to be self-similar. It is a well known fractal called *Sierpinski's Triangle* (or *Sierpinski's Gasket*). When magnified at different levels you still have the same diagram as the original. This shows how images can be stored as fractals, which will then require much less space than the image itself. Large amounts of research have been devoted to using fractals for encoding arbitrary images.

A curious result about fractals is regarding their dimension. Most people are familiar with the idea that a line or smooth curve is 1-dimensional and that a surface is 2-dimensional. Fractals however usually have dimension between 1 and 2. For example, the dimension of Sierpinski's triangle is $\frac{\log 3}{\log 2} \approx 1.585$ [8].

There are some different ways of describing and creating fractals. Two methods, described in the following sections, are iterated function systems and L-Systems.

2.2.2 Iterated Function Systems

Iterated Function Systems is the name given to one of the methods of generating fractals. An iterated function system is a set of functions in which

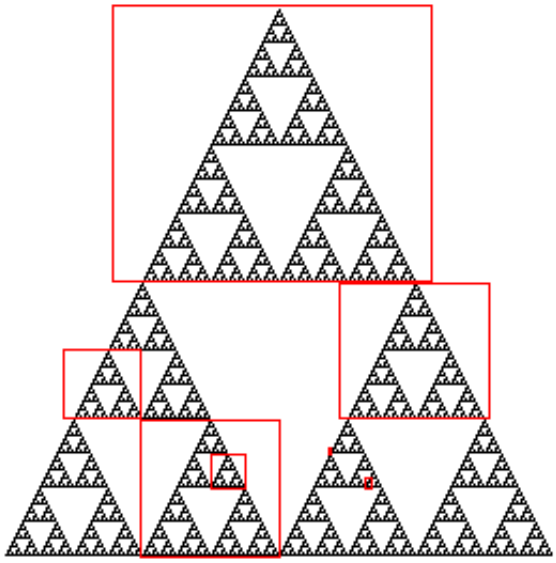


Figure 2.1: Example of self similarity (Sierpinski's Triangle) [1]

all are applied to a set of points or polygons to create a new set of points or polygons. In this report, it will be a set of points that is created and mapped to another set of points.

A two-dimensional iterated function system is described by a set of contractive linear transformations (or a system of functions) [11]. Suppose the set $\{F_1, F_2, \ldots, F_n\}$ is an iterated function system. Let $S \subset \mathbb{R}^2$ and define T to be the function

$$T(S) = \{u \in \mathbb{R}^2 \mid \exists i \in \{1, 2, \ldots, n\}, v \in S \;\; s.t. \;\; u = F_i(v)\}$$

that is, T maps a set of (x, y) points to another set by applying every F_i to each point in S. Now, a set $A \subset \mathbb{R}$ is an *attractor* of the iterated function system if

$$A = T(A)$$

That is, applying all the functions on all the set of points in A will result in the same set A. The fundamental property of an iterated function systems is that it determines a unique attractor, which is usually a fractal [8]. In

addition, given any random set W the following property holds

$$A = \lim_{i \to \infty} T^i(W)$$

Where $T^i(W) = T \circ T^{i-1}(W)$ and $T^1 = T(W)$ [12]. This property guarantees that when approximating the fractal described by an iterated function system, any random set can be the starting point.

There are two methods of plotting a fractal described by an iterated function system: the deterministic method, or by playing the chaos game. The deterministic method starts with a set of points and applies all the functions on all the points to create a new set of points. The old set of points is discarded, then all the functions are again applied to all the points in the new set, and so on. The result is the final generated set. The chaos game starts with one point, then randomly selects a function and applies it to the point to create a new point. Another function is selected at random and is applied to this new point to create another, and so on. The result is the set of all the generated points. See Appendix C for a detailed description of both algorithms.

Example 1 (Sierpinski's Triangle)**.** For the following set of functions

$$F_1 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$
$$F_2 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{bmatrix} \frac{1}{2} \\ 0 \end{bmatrix}$$
$$F_3 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{bmatrix} 0 \\ \frac{1}{2} \end{bmatrix}$$

their attractor is Sierpinski's Triangle.

Figures 2.2, 2.3, 2.4 and 2.5 shows an example of what the plots look like when using the chaos algorithm (Algorithm 3) after 100, 1000, 10 000 and 100 000 iterations respectively.

2.2.3 L-Systems

An *L-System* (or *Lindenmayer system*) is a grammar based method of describing fractals which was first discovered by biologist Aristid Lindenmayer in 1968 as a technique of modelling various plants [13]. L-Systems are simi-

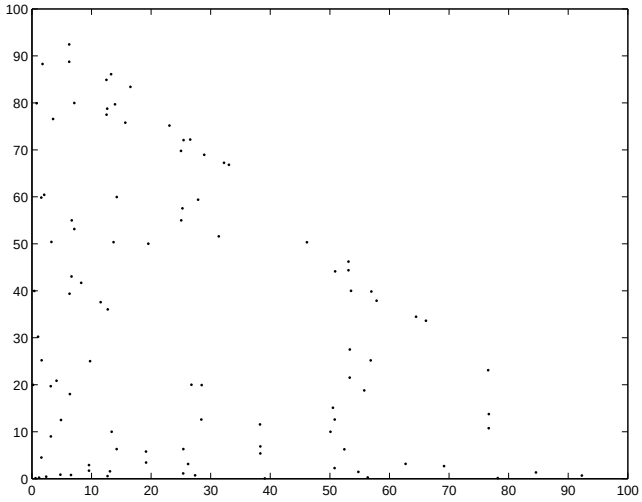


Figure 2.2: Sierpinski's Triangle after 100 iterations

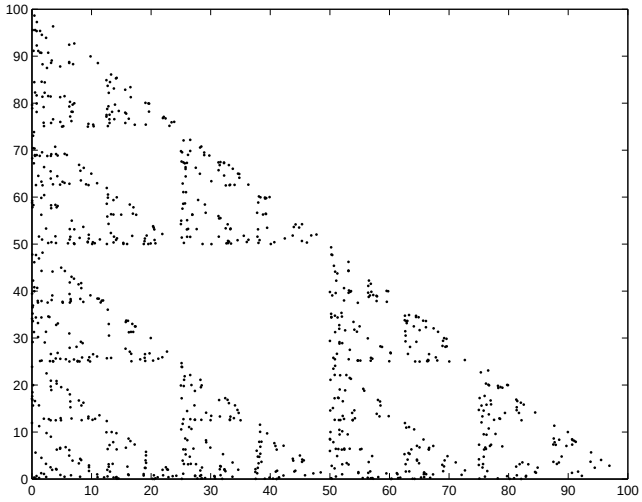


Figure 2.3: Sierpinski's Triangle after 1000 iterations

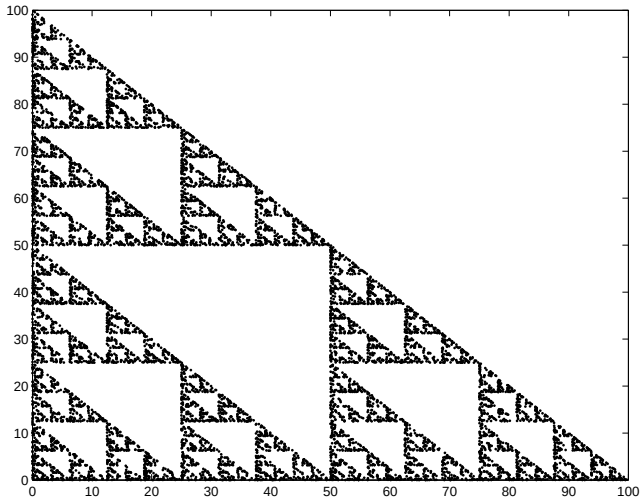


Figure 2.4: Sierpinski's Triangle after 10 000 iterations

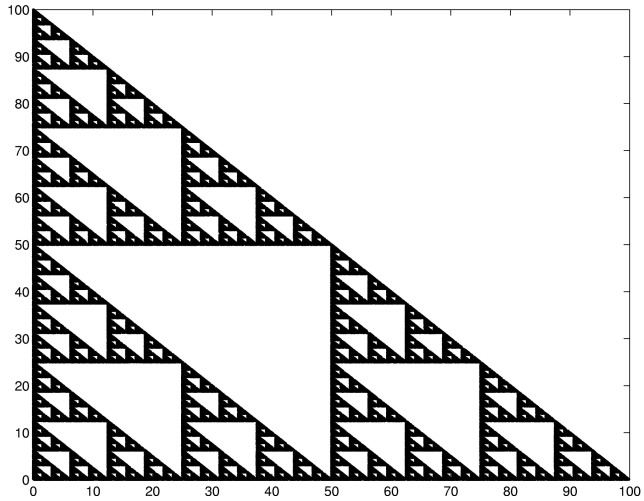


Figure 2.5: Sierpinski's Triangle after 100 000 iterations

lar to Chomsky grammars but differ in certain ways. One difference is that while Chomsky grammars define terminals and non-terminals, this requirement is not needed in L-Systems. Another difference is that in L-Systems, all non-terminal symbols are replaced in parallel at each iteration.

Any string generated by an L-System is graphically interpreted using a programming language similar to LOGO® [14, 15]. In this style of graphics, a "pen-carrying turtle" moves around (a two-dimensional or three-dimensional) plane. If the pen is 'down' then a line is drawn as the turtle moves, otherwise nothing is drawn if the pen is 'up.' The actions controlling the turtle's movement is specified by the sequence of commands generated by the L-System after a certain number of iterations, reading from left to right [16].

L-Systems were initially used to model biological plants and fractals. In recent times, L-Systems have been used to model 3D landscapes [17] and IP traffic [18] among other things. L-Systems are generally used to render lines, however in this report, they will be used to represent points (in the $x - y$ plane)

There are different variations of L-Systems. These include context-free, context-sensitive and stochastic L-Systems. The version used in this report is context-free L-Systems. Formally, a *Context-Free L-System*, $\mathcal{L}$, is specified by the 5-tuple $(\Sigma, A, R, d, \delta)$ where

1. $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ is a finite set of symbols,
2. $A \in \Sigma$ is the start symbol (called the axiom),
3. $R : \Sigma \to \Sigma^*$ is the set of rewrite rules,
   ($*$ is the *Kleene Star*. That is, $\Sigma^* = \{w_1 w_2 \ldots w_n : w_i \in \Sigma, n \geq 0\}$)
4. d is the unit length (to define the distance between two consecutive points) and
5. δ is the unit angle (when a new point to be plotted is not in line with the previous two).

Example 2. Consider the following L-System

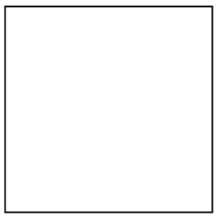
Starting Rule (Axiom):	F+F+F+F
Rewrite rule:	F → F+F–F–FF+F+F–F

and define the following

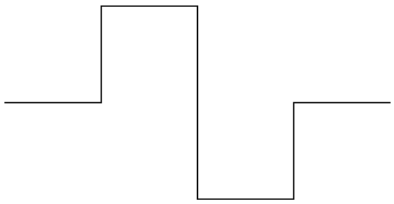
F = Go forward one unit
– = Turn left 90°
+ = Turn right 90°

Then graphically, this is interpreted as:

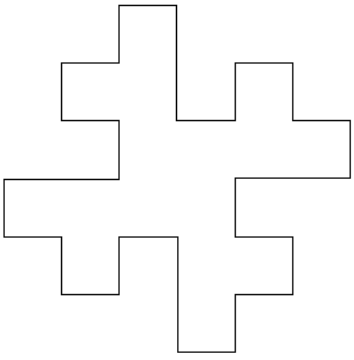
We start with this:



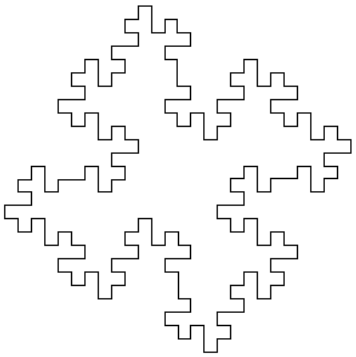
Then replace each straight line with this:



After one iteration we get:



After two iterations we get:



This report will not go into too much detail into plotting the attractor of L-Systems since code provided by Adrian Mariano (coauthor of [19]) was used to achieve this.

2.3 Evolutionary Algorithms

There are actually different variants of Evolutionary Algorithms. The common underlying idea behind all these techniques is the same [20]: "given a population of individuals, the environmental pressure causes natural selection (survival of the fittest) and this causes a rise in the fitness of the population."

The initial set of candidate solutions is created at random and then an *objective function* is applied to each of the candidate solutions to determine which solutions are 'better' than the others. The better solutions are chosen to be parents of the next generation of offspring. Some of the new solutions (offspring) may be mutated (i.e. some aspects are changed). The new solutions are then added to the population. The population itself may be reduced to ensure its size stays below a predefined maximum value. At this point the cycle starts again [21].

This principle has a strong supporting case — it has been used by breeders of plant and livestock for many years to produce species that can provide a higher quality or quantity of the desired product [20].

The concept of evolutionary algorithms was first proposed by Fogel, Owens and Walsh [22]. Initially used to evolve finite state machines, evolutionary algorithms these days are used on many arbitrary structures [22].

2.3.1 Formal Definitions

Evolutionary Algorithms have a number of components, procedures or operators that must be specified in order to be formally defined. The most important components are:

- Representation (definition of individuals or candidate solutions),
- Evaluation function (or fitness function),
- Population (a set of candidate solutions),
- Parent selection mechanism and
- Variation operators: Recombination (or crossover) and mutation

Representation and Population

The first step is to link the 'real world' to the 'programming world,' that is, to discover 'objects' in the original problem context and use them to model the problem space in the program.

Possible solutions within the original problem context in the real world are referred to as *phenotypes* and their encodings are called *genotypes* [20].

In the context of Evolutionary Algorithms, it will be assumed that each individual phenotype can be uniquely identified by a genotype (this is generally not the case in Nature). So the problem is reduced to 'find a good genotype.'

The term *candidate solution* will be used in the Evolutionary Algorithm context and is analogous to a genotype. Define S to be the set of all candidate solutions and the problem is now to find a solution $s \in S$ such that no other solution in S is better than it. The next issue is to define what it means for a solution to be better than another.

Evaluation Function

The evaluation function is a mapping $f : S \to \mathbb{R}$, which assigns a real value to each solution such that if s_1 satisfies the requirements better than s_2 then $f(s_1) > f(s_2)$. The aim is then to find a solution $s \in S$ where $f(s)$ is the maximum value.

The evaluation function defines what it means for a solution to be better than another and Evolutionary Algorithms can be viewed as a technique to solve optimisation problems.

Parent Selection Mechanism

The algorithm starts with a random population $P \subset S$ of candidate solutions. A parent selection mechanism is a function that is used to select a set of solutions from the population P depending on the value $f(s)$ for every $s \in P$. Define a function $g : P \to \{0, 1\}$ to select a set of parents to produce the next generation. This is the parent selection function.

Here, g is used to select a set $P' = \{p \in P \mid g(p) = 1\}$ and it is this set that contains the 'best' candidates in P.

Variation Operators

Every subsequent generation is created by applying recombination and/or mutation. Variation operators form the evolutionary implementation of the elementary steps within the search space [20]. Generating a child is equivalent to moving to a new point in this solution space.

Recombination (or crossover) is a binary operator r that merges information from two parent solutions into one or two solutions. For this project, when evolving iterated function systems, r will be defined to produce one *child* solution. That is,

$$r : S \times S \to S$$

However, when evolving L-Systems, r will be defined to produce two children. That is

$$r : S \times S \to S \times S$$

These decisions are based on how each object is implemented in the software.

It should be noted that the definition of r can be generalised to have 'more than two parents' even though there is no biological equivalent. This is probably the reason why they are not commonly used, although several studies indicate that they have positive effects on the evolution [20].

The principle behind recombination is simple — if two individuals with different but desirable features are used to produce an offspring, it is possible that the offspring will contain a combination of the good features.

Algorithm 1: The general scheme of an Evolutionary Algorithm

```
1  Initialise P with random candidate solutions
2  repeat
3      Construct P' = {p ∈ P | g(p) = 1}
       /* Note: The way parents are paired up is different
          for different implementations.                    */
4      Recombine pairs of parents to generate a set of offspring (O)
5      Select a set of offspring (M ⊂ O) for mutation.
6      Mutate: M' = {m(o) | o ∈ M}
7      Combine solutions: P'' = P ∪ M' ∪ (O − M)
8      Select individuals for the next generation
9      P ← The next generation
10 until Termination condition is satisfied
```

Mutation is a unary operator $m : S \rightarrow S$ which modifies a provided solution to produce a (slightly) modified 'mutant'. In general, mutation is supposed to cause a random, unbiased change [20].

Mutation has a theoretical role. It can guarantee that the solution space the algorithm is searching through is connected [20]. This is important since some theorems which state that an Evolutionary Algorithm will discover the global optimum, rely on this property. However, it should also be noted that many researchers feel these proofs are of little use and thus many implementations of Evolutionary Algorithms do not use a mutation operator [20]. For this project, a mutation operator will be used.

Not every child will be mutated. A random selection of children is chosen during each generation for mutation.

2.3.2 Putting it all together

Algorithm 1 describes the general scheme of an Evolutionary Algorithm and how all the different functions that have been defined above are used together.

Currently, evolutionary algorithms are being used to solve problems such as the timetabling problem, finding new drugs [22] by finding particular shaped protein structures, constructing 2-edge-connected minimal Steiner graphs [23] and even learning to play checkers [24].

An interesting fact about evolutionary algorithms is that the technique mentioned above is inherently restricted by the inadequate distinction be-

tween genotypes and phenotypes. In nature, the fertilized egg cell undergoes a complex process known as *embryogenesis* to become a mature phenotype. It is believed that this process helps in reducing the probability of undesirable mutations. Recent work in the artificial embryogeny and artificial developmental systems are attempting to simulate this technique to provide more efficient algorithms [25].

2.4 Summary

This section has briefly described the most important aspects for which an understanding is required in order to complete this project.

Cotta and Moscato performed an evolutionary computation-based attack on algorithms for which a theoretical bound is already known (e.g. The Bubble Sort algorithm) [2]. This report will attempt to use the same technique to analyse a more complicated algorithm for which a theoretical bound is more difficult to compute. This algorithm is the one used by the *Concorde* program. The reason for the information contained in this section is because

- The *Concorde* program is a combinatorial optimisation algorithm, designed to solve Traveling Salesperson Problem instances.
- One aim is to empirically estimate *Concorde*'s time complexity.
- Another aim is to describe a set of instances that are relatively difficult for *Concorde* to solve.
- Fractals provide a method of defining a complete graph (as a set of (x, y) points in $\mathbb{R}^2$) by describing the *structure* of the graph.
- Fractals allow us to describe what sort of instances are relatively difficult for *Concorde* to solve.
- Iterated function systems and L-Systems will be used to construct fractals to use for creating instances for *Concorde*.
- It will be attempted to find 'simple' fractals which can generate difficult instances, the number of functions in iterated function systems and the number of rules in L-Systems will be kept small. To do this,

the number of functions and rules will be kept constant for iterated function systems and L-Systems respectively.

- The fractals will be generated by creating a population of many fractals and applying the evolutionary algorithm technique on this population to evolve the fractals in order to find one which produces relatively difficult instances for *Concorde* to solve.

Chapter 3

Implementation

This chapter describes what was created and how they were designed in order to complete this project. For this project, an algorithm was designed to compute the fitness of a fractal and two evolutionary algorithms were designed to model and evolve iterated function systems and L-Systems.

3.1 Computing The Upper Bound On A Set Of Data Points.

A fractal can be used to create plots of different sizes. In order to assess the arduousness of a fractal, each plot is used as an instance for the traveling salesperson problem and is solved by *Concorde*. The various times required for solving the instances are plotted on a graph and an upper bound is calculated for the set of data points.

Since the curve that is to be discovered is an *upper bound* of the data points, this curve will be used to estimate the worst-case time complexity of any fractal and can be used to estimate the time complexity of *Concorde*. An algorithm was needed to automatically (and quickly) find such a curve. This idea could not be found in the literature so a program to compute this curve was designed and implemented.

3.1.1 The data used to find the upper bound

Given a fractal F, a function $f : F \mapsto \mathbb{R}$ is required to describe how long *Concorde* takes to solve instances generated by the fractal.

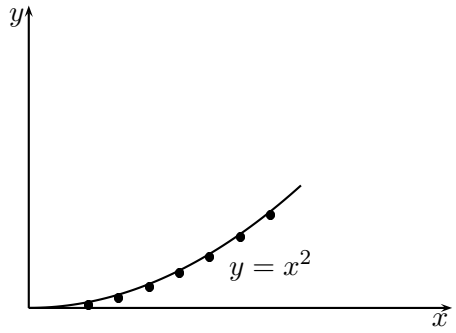


Figure 3.1: An upper bound on a set of (x, y) points.

Define $T : (F, n) \mapsto \mathbb{R}$ to be the time taken to solve a TSP instance with n cities, created by the fractal F.

In order to estimate an upper bound of a fractal, the values of the following set must first be found

$$\tau_F = \{t \mid t = T(F, n), n \in I\}$$

where the set I is used to define the size of the instances that will be used for testing. For example, if $I = \{10, 20\}$ then for any fractal F, the values $T(F, 10), T(F, 20)$ would be computed. If these values are plotted onto a graph then it is possible to find a function which lies above all the points (Figure 3.1).

3.1.2 The function to be found

The assumption will be made that the running time of *Concorde* in the worst case is in polynomial time. So the aim is to find a function $\phi : \mathbb{R} \to \mathbb{R}$ of the form

$$\phi(x) = x^{\alpha}$$

where $\alpha \in \mathbb{R}$ is a constant. It is this α that is used to help determine a fractal's fitness since α is proportional to the rate at which larger instances becomes harder to solve.

3.1.3 Standardising the data

Before the α can be found, the data needs to be 'standardised' because there are other constants which can affect the shape of a polynomial. In order for

the ideas in this section to be applicable, there must be at least two data points. In fact, there must be at least *three* data points before a meaningful result is produced.

So if A represents the set of data points for which an upper bound is to be computed, it will be assumed that

$$(x_0, y_0), (x_1, y_1) \in A$$

such that

$$x_0 < x_i, \quad \forall x_i \in \big\{x \,|\, \exists y, \big((x, y) \in A\big) \wedge (x \neq x_0)\big\}, \text{ and}$$
$$x_j < x_1 \implies x_j = x_0$$

That is, of all the x's in the set, x_0 is the smallest one and x_1 is the second smallest one.

Note that a polynomial need not have only one term. For example the following is a valid polynomial

$$p(x) = x^{3.2} + x^3 + x^{2.8}$$

But the *order* of the polynomial is $\mathcal{O}\big(x^{3.2}\big)$. So we look to find a polynomial that has only one term.

However it is not enough to assume that for any set of (x, y) points, the polynomial has the form x^α. There may be other constants which distort the shape of the polynomial in some way. In fact the most general form of a polynomial with one term is

$$y(x) = a\left(\frac{x - c}{b}\right)^\alpha + d, \quad a, b, c, d, \alpha \in \mathbb{R}$$

It will now be explained how the multiplicative and additive constants affect the polynomial.

The first issue that must be considered is the fact that the data points may be *shifted* away from the origin. Consider Figure 3.2. Here there are two sets of data, both of the same order:

$$\mathcal{O}\big((x - 3)^2 + 20\big) = \mathcal{O}\big(x^2 - 6x + 29\big) = \mathcal{O}\big(x^2\big)$$

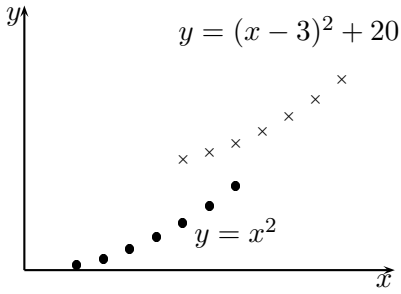


Figure 3.2: Two sets of data of the same order. One set has been shifted away from the origin.

however, one set starts at the origin while the other has been translated to a different origin.

So to simplify the algorithm, the data points will first be shifted towards the origin. So the following change is made to all the points:

$$(x_i', y_i') = (x_i - x_0, y_i - y_0), \quad i > 0$$

Thus, in the new set of data points, we have that $x_0' = y_0' = 0$.

The other issue that must be considered is the fact that (even after shifting) the points may still be *scaled*. Consider Figure 3.3. Here there are two sets of data, both of the same order:

$$\mathcal{O}(x^2) = \mathcal{O}\left(6\left(\frac{x}{2}\right)^2\right)$$

Here one set has the property that $x_1 = y_1 = 1$ whereas in the other set, we have that $x_1 = 2$ and $y_1 = 6$. The x's are scaled by 2 and the y's are scaled by 6.

So to simplify the algorithm, the data points will also be scaled so that the new set of data points has the property that $x_1'' = y_1'' = 1$. To achieve this, the following change is made to all the points:

$$(x_i'', y_i'') = \left(\frac{x_i'}{x_1'}, \frac{y_i'}{y_1'}\right), \quad i > 0$$

Notice that we cannot have $x_1' = 0$ (i.e. $x_1 = x_0$) and similarly we cannot have $y_1' = 0$ (i.e. $y_1 = y_0$). That is, the first two points must have distinct x and y coordinates.

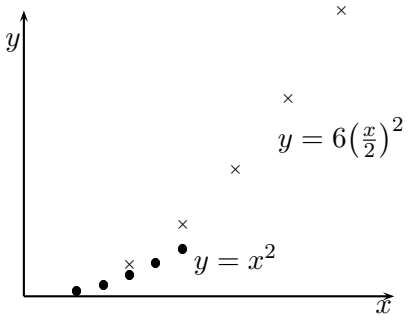


Figure 3.3: Two sets of data of the same order. One set has been scaled up in both the x and y direction.

3.1.4 Finding the upper bound

A *divide and conquer* approach is used to find the required constant α once the data has been 'standardised'. The algorithm starts with the initial range $[0, \alpha_0]$, where α_0 is large enough so that x^{α_0} lies above all the points (this value is not guaranteed to be a 'good' value, it is simply a starting point).

Then the midpoint $\alpha_1 = \frac{\alpha_0+0}{2}$ is calculated. If x^{α_1} lies above all the points, then the range becomes $[0, \alpha_1]$ otherwise the new range is $[\alpha_1, \alpha_0]$. This ensures that the required constant α is within the selected range.

The divide and conquer strategy is applied repeatedly, ensuring that at all times, α is contained in the range. This continues on until the range $[\alpha_i, \alpha_j]$ is sufficiently small enough. When this happens, the function is estimated to be x^{α_j}.

The procedure explained above will find an α such that $y = x^\alpha$ is an upper bound of the data points allowing us to conclude that the order of the data points is $\mathcal{O}(x^\alpha)$. In fact, the upper bound would actually be

$$y(x) = y_1\left(\frac{x - x_0}{x_1}\right)^\alpha + y_0$$

but the *order* is still the same.

3.1.5 The Algorithm

Function `find_upper_bound()` (page 24) describes how the upper bound on a set of data points is calculated.

Function `find_upper_bound` Computing an upper bound on a set of (x, y) points.

output: A number $\alpha \in \mathbb{R}$ such that x^α is an upper bound of a set of predefined data points.

```
/* 'Standardise' the data points first.                    */
```
1 **foreach** $i \in \{1, 2, \ldots, n\}$ **do**
2 | $x_i \leftarrow (x_i - x_0)$
3 | $y_i \leftarrow (y_i - y_0)$
4 **end**
5 **foreach** $i \in \{2, 3, \ldots, n\}$ **do**
6 | $x_i \leftarrow x_i / x_1$
7 | $y_i \leftarrow y_i / y_1$
8 **end**

```
/* We are trying to find a polynomial of the form x^α.
   initial_polynomial() is a function which returns a
   value c such that x^c lies above all the points.        */
```
9 $\alpha_{high} \leftarrow$ `initial_polynomial()`
10 $\alpha_{low} \leftarrow 0.0$

```
/* Now try to reduce the interval [α_high, α_low] until we have
   a good approximation of an upper bound.                 */
```
11 $\delta \leftarrow 0$
12 **repeat**
| `/* Use the 'divide & conquer' approach to find a good upper bound estimate. */`
13 | $\alpha_{temp} \leftarrow (\alpha_{high} + \alpha_{low})/2$
14 | **if** *No point lies above the function* $x^{\alpha_{temp}}$ **then**
15 | | $\alpha_{high} \leftarrow \alpha_{temp}$
16 | **else**
17 | | $\alpha_{low} \leftarrow \alpha_{temp}$
18 | **end**
19 | $\delta \leftarrow (\alpha_{high} - \alpha_{low})$
20 **until** $\delta < 0.0005$
21 **return** α_{high}

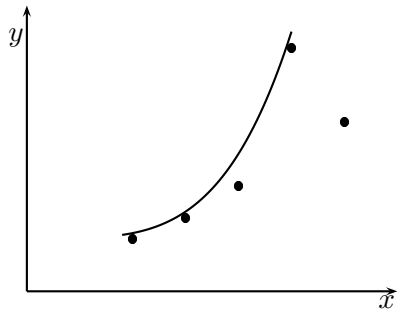


Figure 3.4: The fourth data point has affected the upper bound so much that, as a result, the upper bound is no longer a good estimate for the data set.

3.2 Determining the 'fitness' of a fractal

The previous section explained how to find an upper bound on a set of data points. This upper bound is used to help determine the fitness of a fractal. The reasoning is that the larger the upper bound (i.e. the greater the rate of increase of time required to solve the instance), the more 'fit' the fractal.

However the fitness function cannot solely rely on the upper bound. This is because one data point can greatly affect the upper bound forcing the other points to have little effect on the upper bound. Consider Figure 3.4. The fourth data point has affected the upper bound so much that the upper bound is no longer a good estimate for the data set.

Hence the fitness function must also penalise data sets for which the upper bound is not a 'good estimate' for all the points. The method used to determine this is similar to the *residual sum of squares* metric used for linear regression. Instead of computing the squares, the difference between the estimated value and the actual value is used (since every point will lie either *on* or *below the line*, we will not have negative numbers). This will be denoted as the *sum of errors (SE)*.

So given the upper bound $u(x)$ and a set of data points $\{(x_n, y_n)\}_{n=0}^{N}$ the sum of errors is computed as

$$SE = \sum_{n=0}^{N} (u(x_n) - y_n)$$

Given a fractal F, a function $f : F \mapsto \mathbb{R}$ is required to assign a value to F indicating the fitness of the fractal. This value is used to compare fractals

to one another for determining which ones should be selected to be parents.

For this project, the fitness value is defined to be

$$f(F) = \frac{\alpha}{SE}$$

where α is the order of the upper bound and SE is the sum of errors of the upper bound compared to the running times $\{(n, T(F, n))\}_{n \in I}$.

This definition allows the fitness function to be proportional to the order of the upper bound *and* inversely proportional to the sum of errors. A higher order (of the upper bound) will increase the fitness, whereas a higher sum of errors will decrease the fitness.

3.3 The Evolutionary Algorithms to Find Difficult Instances for *Concorde*

This section describes the parameters of the evolutionary algorithms that are used to find relatively difficult instances for *Concorde* to solve.

With a definition of what it means for a fractal to be better than another the remaining details of the evolutionary algorithm are needed to be defined.

3.3.1 Evolving Iterated Function Systems

The algorithm that will be used to find iterated function systems representing fractals to create TSP instances will be an evolutionary algorithm. Thus there are a few attributes of this algorithm that must be defined.

Representation

Representation of an individual in the population is simple. Since every function is affine, it can be represented as a matrix. In the actual program, it will be represented in an array of size 6. An affine transformation ϕ has the form

$$\phi \begin{pmatrix} x \\ y \end{pmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

So the array is constructed like so

$$\phi = [a, b, e, c, d, f]$$

Evaluation Function

The evaluation function explained in Section 3.2 is used with

$$I = \{50, 100, 150, 200, 250\}$$

(see Section 3.1.1). This set will be kept constant throughout the testing.

Population

The size of the population is of the form $p + p/2$, where p is the number of parents and $p/2$ is the number of children (i.e. each pair of parents will produce one child). This implies that the best p individuals in the population will be used as the parents to produce the next generation.

The value of p will *not* be kept constant. Different values will be tested in an attempt to find an ideal situation which produces good solutions. The range chosen is $20 \leqslant p \leqslant 100$ since most notes found in the literature use these sizes.

Recombination

There are a many ways of combining two parents to produce a child. The chosen method was suggested by the author's supervisor. Suppose we have two parents (iterated function systems)

$$p_1 = \{F_{11}, F_{12}, \ldots, F_{1n}\}$$
$$p_2 = \{F_{21}, F_{22}, \ldots, F_{2n}\}$$

Then the child created will have the form

$$c = \{F_{c1}, F_{c2}, \ldots, F_{cn}\}$$

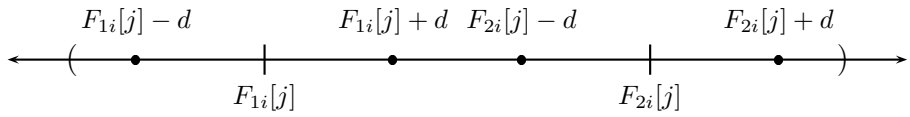
Recall that an iterated function system is represented as an array of size 6 and so each F_{ci} is computed as:

$$F_{ci}[j] \leftarrow \text{a random number from } \{F_{1i}[j] - d, F_{1i}[j] + d, F_{2i}[j] - d, F_{2i}[j] + d\}$$

where

$$d = \frac{1}{3}\Big|F_{1i}[j] - F_{2i}[j]\Big|$$

The following diagram shows the four possible values that $F_{ci}[j]$ may be assigned (assuming $F_{2i}[j] > F_{1i}[j]$),



This ensures that no value will be bounded. To understand this reasoning, assume that taking averages is the method used. Suppose that A is the largest value any parent has and that B is the smallest value that any parent has. Then no child will have a value greater than A or less than B. This is undesirable since the largest possible search space is beneficial.

Parent Selection Mechanism

The population is sorted in decreasing order of $f(F)$ and the first p individuals (iterated function systems) are used as parents. If the sorted set of individuals is $\{\xi_0, \xi_1, \ldots, \xi_{p-1}, \ldots, \xi_{p+\frac{p}{2}}\}$ then each pair of parents are selected at random according to the following probability distribution:

Individual	$P(\xi_i)$
ξ_0	0.5
ξ_1	0.25
ξ_2	0.125
$\vdots$	$\vdots$
ξ_{p-1}	$\frac{P(\xi_{p-2})}{2}$

The reasoning is that since the most fit individual (ξ_0) has a 50% chance of being selected, there is a greater chance for it contribute to the production of the next generation. The next best individual (ξ_1) has a 25% chance of being selected and will not contribute to the next generation as much as ξ_0, but will affect the next generation more than the rest ($\xi_2, \ldots, \xi_{p-1}$).

For iterated function systems, it is possible to select the same parent twice. Selecting the same pair more than once will most likely result in different offspring being produced each time.

Property	Implementation Method
Representation	A set of arrays (of size 6).
Population size	$p + p/2$ (p parents, $p/2$ children). Variant.
Parent Selection	Best p individuals in the population.
Recombination	$F_{ci}[j]$ is a random number from the set: $\{F_{1i}[j] - d, F_{1i}[j] + d, F_{2i}[j] - d, F_{2i}[j] + d\}$ where $\frac{1}{3}(F_{1i}[j] + F_{2i}[j])$.
Mutation	$F_{ci}[j]$ is assigned a random number from the range: $[F_{ci}[j] - 0.5, F_{ci}[j] + 0.5]$.
Mutation probability	$33\frac{1}{3}\%$.
Initialisation	Random values for $F_i[j]$ between -5 and 5.
Termination Condition	Reached predefined number of generations.

Table 3.1: Description of the EA used for finding iterated function systems

Mutation

Of all the children generated, each child is selected for mutation with probability $\frac{1}{3}$. That is, each child has a 1 in 3 chance of being mutated. If a child (iterated function system) is selected for mutation, then one of its functions is selected at random. Every value ζ in the array representing this function will be assigned a new value according to the following rule:

$$\zeta \leftarrow \text{A random number from within the range } (\zeta - 0.5, \zeta + 0.5)$$

Number of Generations

The number of generations of children produced will not be kept constant. Different values will be tested in an attempt to find an ideal situation which produces good solutions.

Summary

The description of this evolutionary algorithm to find fractals, described by Iterated Function Systems, which generates instances that are difficult for *Concorde* to solve, is summarised in Table 3.1

3.3.2 Evolving L-Systems

The algorithm that will be used to find L-Systems representing fractals to create TSP instances will be an evolutionary algorithm. Thus there are a few attributes of this algorithm that must be defined.

Representation

The axiom and every rewrite rule in an L-System can each be represented as strings. So each L-System is represented as an array of strings. The allowable characters will be "F,+,-,!,A,B,..." the meaning of each is

F	Move Forward
+	Increase angle
-	Decrease angle
!	Reverse direction (switch meanings of +,-)
A,B,...	The rewrite rules

The code which generates fractals from L-Systems requires a value called `angle`. This is a natural number and is used to compute δ like so (see Section 2.2.3)

$$\delta = \left(\frac{360}{\texttt{angle}} \right)^{\circ}$$

Also, every L-System will have the same number of rules.

Initialisation

When creating an L-System at random or when mutating a rule of an L-System, the characters defined above will need to be randomly selected. However, not every character will have an equal probability of being selected.

There will be a greater probability that the symbols '+,-' are selected over the other symbols except one. The letter 'F' will have the greatest probability of being selected.

Evaluation Function

Same as for Iterated Function Systems. However, the set I (Section 3.1.1) cannot be kept constant. The code obtained for generating fractals described by L-Systems creates instances based on the *order* not size. An

order refers to the number of iterations used to create the fractal (see Example 2, page 11).

Population

The size of the population is of the form $2p$, where p is the number of parents. Thus each pair of parents will produce *two* children. Again, the best p individuals in the population will be used as the parents to produce the next generation.

The value of p will *not* be kept constant. Different values will be tested in an attempt to find an ideal situation which produces good solutions. The range chosen is again $20 \leqslant p \leqslant 100$ since most notes found in the literature use these sizes.

Recombination

There are a many ways of combining two parents to produce a child. The method chosen is sometimes referred to as the *single point crossover* technique. If r_{p_1} is a rule of parent p_1 and r_{p_2} is a corresponding rule of parent p_2, then they have the form

$$r_{p_1} = a_1 a_2 \cdots a_m$$
$$r_{p_2} = b_1 b_2 \cdots b_n$$

where a_i, b_i is an allowable character (described earlier in the section *Representation*). Two random numbers ξ, ζ will be chosen such that $1 \leqslant \xi \leqslant m$ and $1 \leqslant \zeta \leqslant n$. The corresponding rule for the two new children c_1, c_2 are constructed like so

$$r_{c_1} = a_1 a_2 \cdots a_\xi \, b_{\zeta+1} b_{\zeta+2} \cdots b_m$$
$$r_{c_1} = b_1 b_2 \cdots b_\zeta \, a_{\xi+1} a_{\xi+2} \cdots a_n$$

where r_{c_1}, r_{c_2} are the new rules for child c_1 and c_2 respectively.

Parent Selection Mechanism

Similar to the one for Iterated Function Systems except for one feature. The only difference for L-Systems is that it will *not* be possible to select the same

Property	Implementation Method
Representation	A set of L-System structures. Each L-System structure is represented by a set of strings.
Population size	$2p$ (p parents, p children). Variant.
Parent Selection	Best p individuals in the population.
Recombination	Single point crossover for each rule.
Mutation	One rule (selected at random) completely replaced from the selected child.
Mutation probability	$1/p$.
Initialisation	Random characters according to a specified weighted probability distribution.
Termination Condition	Reached predefined number of generations.

Table 3.2: Description of the EA used for finding L-Systems

parent twice. Although selecting the same pair more than once can result in different offspring being produced each time, the probability of the offspring being different can be significantly smaller than in the situation of iterated function systems—especially when the rewrite rules are small in length.

Mutation

Once the p children have been created, one is selected at random for mutation. Thus each child has a 1 in p chance of being selected. Then one of the child's rule is selected at random and is replaced with a new random rule. Selecting a new symbol, while constructing the new rule, is similar to when initialising an L-System.

Number of Generations

Same as for Iterated Function Systems.

Summary

The description of this evolutionary algorithm to find fractals, described by L-Systems, which generates instances that are difficult for *Concorde* to solve, is summarised in Table 3.2

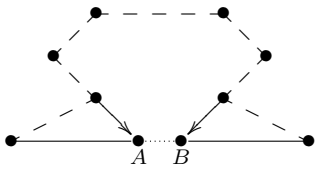
3.4 Average Running Times

In order to justify that the fractals found produce instances that are in fact harder for *Concorde* to solve they need to be compared to a set of *average* running times. These times are used as a *base case* to compare with other results.

The code simply generates random x and y coordinates within the range $[0, 100]$. Random instances of a particular size are created 50 times and the average of the running times is calculated.

3.5 Modifying An Instance To Increase Difficulty

It is hypothesized that the instances can sometimes be made more difficult if the points are 'moved' to a different place—but not too far. It may be possible a situation similar to the diagram below may occur



In the diagram, cities A and B have been moved closer together. In this situation, some algorithms may be 'tempted' to include arc AB instead of following the dashed path. This idea was tested and the results for fractal IFS_2.92_041007 are shown in Table 5.5. The results for fractal LS_1.36_011007 are shown in Table 5.6.

Each point (x, y) will be replaced by a new point (x', y') chosen at random such that $x' \in (x-\varepsilon, x+\varepsilon)$ and $y' \in (y-\varepsilon, y+\varepsilon)$ where ε is a small number. The values of $0.05, 0.1$ and 0.2 will be used for ε.

3.6 Testing Conditions

Concorde requires the use of an initial random seed. The value zero was used for the seed. Also *Concorde*'s default options were used. This was to ensure that the test conditions can be recreated whenever the experiments are needed to be repeated. See the Colophon chapter for details about the hardware and other software used.

Chapter 4

Results

This section starts of with an estimate of *Concorde*'s average running time and presents the fractals of greatest fitness found during this project. The instances generated by the fractals are presented including the equation of the upper bound (of the $(n, T(F, n))$ data points—see Section 3.1.1).

4.1 Average/Base Case

As explained in Section 3.4 a program was created to find the average running time it takes *Concorde* to solve instances of different sizes.

After running the tests, it was found that on average, *Concorde* runs in $\mathcal{O}(n^{1.6})$ time. The running times are in Table A.1 and the graph of the upper bound is shown in Figure 4.1.

Every result will have an upper bound and each upper bound will be compared to $\mathcal{O}(n^{1.6})$. For example, in Figure 4.3 the average upper bound is scaled so that it can be more easily compared to the fractal's upper bound. The upper bound is scaled similar to the technique explained in Section 3.1.3.

4.2 Iterated Function Systems

The Fitness function described in Section 3.2 was not the first fitness function used. Throughout the project, the fitness function was continuously modified in an attempt to produce better results. The results found using the different fitness functions are presented here.

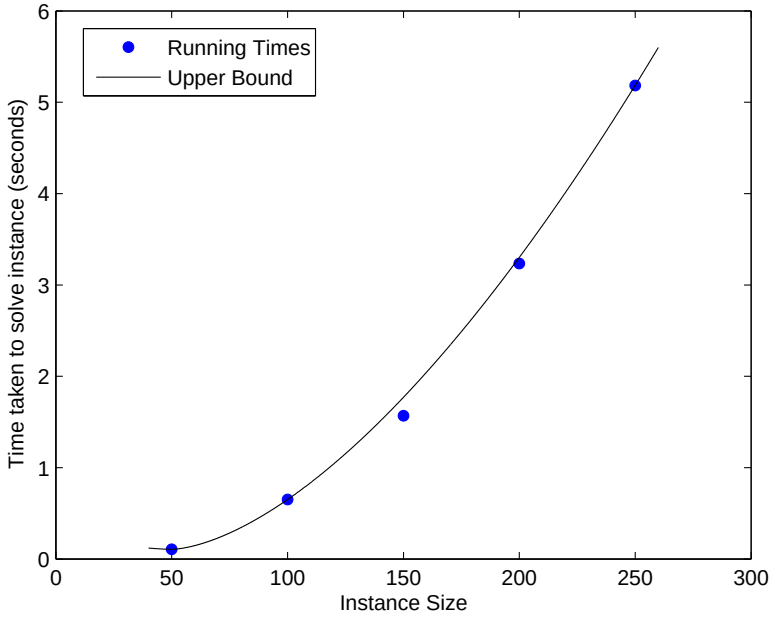


Figure 4.1: Upper bound of the Average Running Times

4.2.1 IFS_1.99_200507

Initially, the fitness function was simply the order of the upper bound. Using this, one of the first iterated function systems found was IFS_1.99_200507[1]. The population had 10 parents and 1000 generations were created. The affine transformations for this IFS is shown in Table 4.1, the running times are in Table A.3 and the graph of the upper bound is shown in Figure 4.3. The order of the upper bound is $\mathcal{O}(1.99)$

Like many of the initial iterated function systems found, the upper bound is not a good estimation of the running times of the larger instance sizes. The fitness of the population (Figure 4.7) did not improve after the first 20% of the generations were created.

4.2.2 IFS_3.96_100707

Another run of the evolutionary algorithm with the fitness function unchanged produced fractal IFS_3.96_100707. The population again had 10

[1]The naming convention used is [*fractal type*]_[*order of upper bound*]_*ddmmyy* where the date is when the fractal was found.

parents and 1000 generations were created. The affine transformations for this IFS is shown in Table 4.2, the running times are in Table A.4 and the graph of the upper bound is shown in Figure 4.6.

Once again, the upper bound is not a good estimation of the running times of the larger instance sizes and the fitness of the population (Figure 4.7) did not improve after the first 20% of the generations were created.

It was decided that the parent selection mechanism should be modified in an attempt to have the fitness of the population improve at a better rate. If the sorted set of individuals in the population is $\{\xi_0, \xi_1, \ldots, \xi_{p-1}, \ldots, \xi_{p+\frac{p}{2}}\}$ then previously, the parents were paired up like so

$$\begin{aligned} &\text{Couple } 1 && \xi_0, \xi_1 \\ &\text{Couple } 2 && \xi_2, \xi_3 \\ & && \vdots \\ &\text{Couple } i && \xi_{2i-2}, \xi_{2i-1} \\ & && \vdots \\ &\text{Couple } p/2 && \xi_{p-2}, \xi_{p-1} \end{aligned}$$

The reasoning was that the two most fit individuals were paired together to hopefully produce a very fit child. The algorithm was changed to select parents at random according to a non-uniform probability distribution as explained in Section 3.3.1. This gave the more fit individuals more chances to produce offspring (and the less fit individuals would have less chances).

4.2.3 IFS_5.89_170707

In order to test the new parent selection mechanism, the evolutionary algorithm was run with a population having only 2 parents and 1000 generations. The result was a population which improved four times (Figure 4.10) — a large improvement over previous results. This test run produced IFS_5.89_170707. The affine transformations for this IFS is shown in Table 4.3, the running times are in Table A.5 and the graph of the upper bound is shown in Figure 4.9.

4.2.4 IFS_12.65_250707

The evolutionary algorithm was run again with 40 parents and produced 10 000 generations. This test run produced fractal IFS_12.65_250707. Only

the affine transformations (Table 4.4) and the upper bound (Figure 4.11) is given. It is clear from the upper bound that the outlier has affected the equation of the upper bound so much that it no longer is a good estimate of the other running times.

It was for this reason the final modification was made to the fitness function which was changed to the one explained in Section 3.2.

4.2.5 IFS_2.29_250907

Another run of the evolutionary algorithm with the new fitness function produced fractal IFS_2.29_250907. The population had 10 parents but this time only 300 generations, because previous runs of the algorithm did not improve the population after 300 generations. The affine transformations for this IFS is shown in Table 4.5, the running times are in Table A.7 and the graph of the upper bound is shown in Figure 4.13.

Note that this time the IFS is generated by 6 affine transformations. Based on past results, it is assumed that a more complicated IFS is required. Initially, it was desired to have a 'simple' IFS (described by only 3 functions), however the results suggest that this situation cannot occur.

4.2.6 IFS_2.92_041007

The most fit IFS found during testing is IFS_2.92_041007. The population had 20 parents and 300 generations. The affine transformations for this IFS is shown in Table 4.6, the running times are in Table A.8 and the graph of the upper bound is shown in Figure 4.16.

Table 4.1: IFS_1.99_200507

a	b	e	c	d	f
-0.070273	-0.363166	0.596501	-0.737695	0.941127	0.550180
-0.162051	0.171974	0.569920	0.058224	0.167818	0.049012
0.347313	0.120308	0.247653	0.449957	0.186290	0.170650

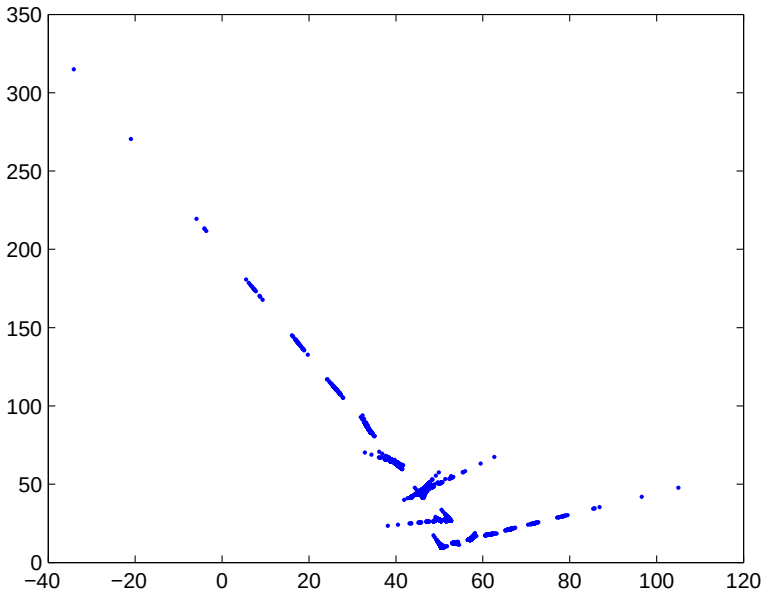


Figure 4.2: A plot of IFS_1.99_200507 with 10 000 points

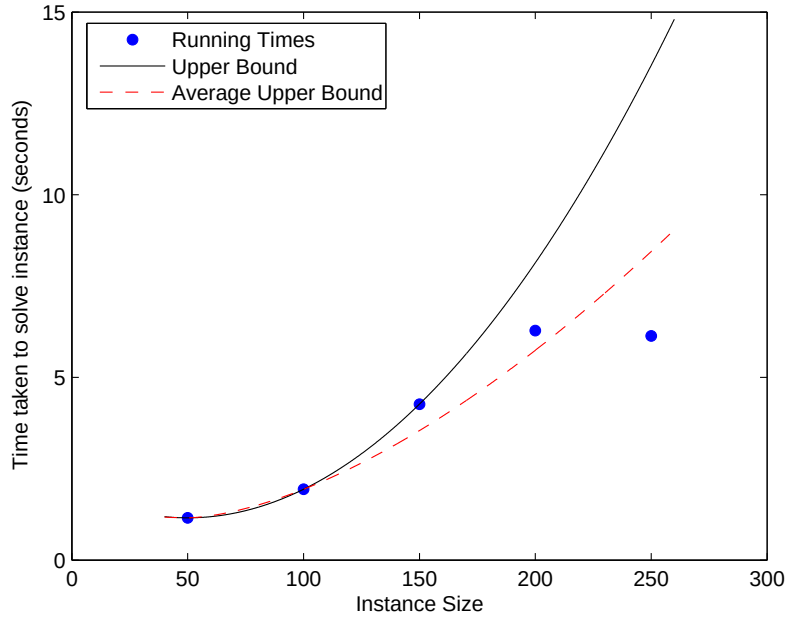


Figure 4.3: Upper bound of the average running times for IFS_1.99_200507

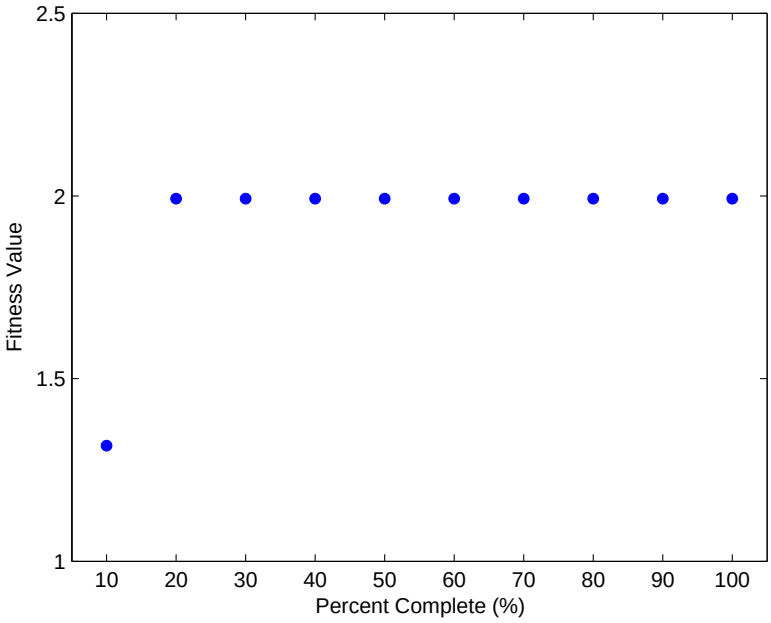


Figure 4.4: The rate of improvement of the population when the evolutionary algorithm found IFS_1.99_200507

Table 4.2: IFS_3.96_100707

a	b	e	c	d	f
-0.049536	-0.357767	0.835150	-0.338633	-0.852805	0.905463
-0.984300	-0.654117	0.810901	-0.165061	0.747261	0.453812
0.272724	-0.104708	0.550801	-0.303610	-0.489208	0.987499

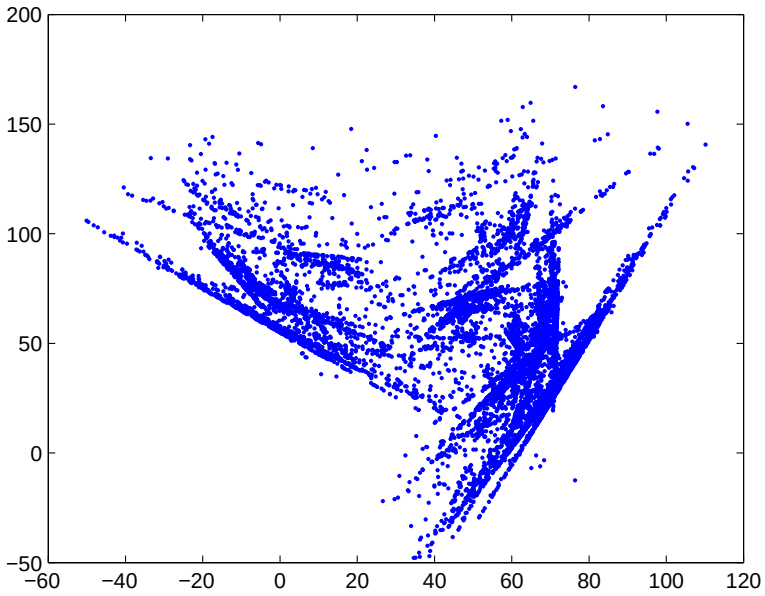


Figure 4.5: A plot of IFS_3.96_100707 with 10 000 points

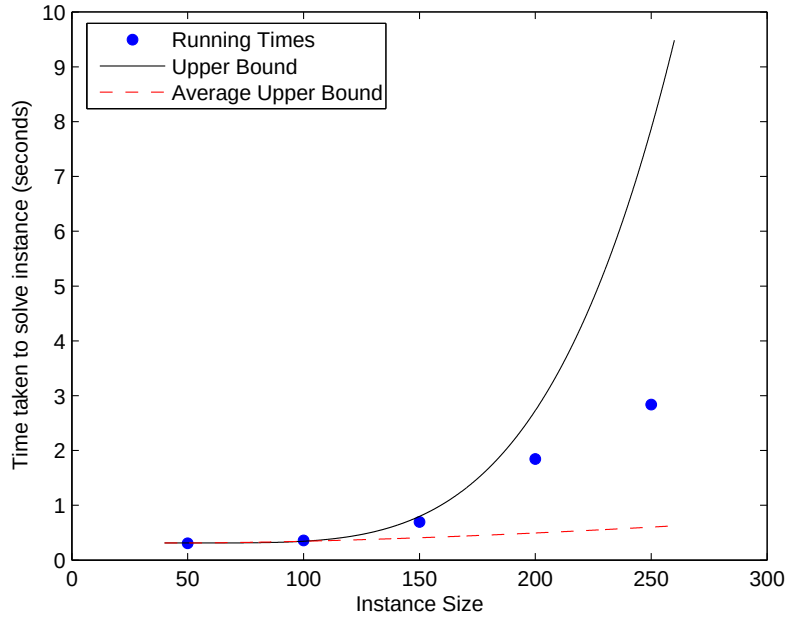


Figure 4.6: Upper bound of the average running times for IFS_3.96_100707

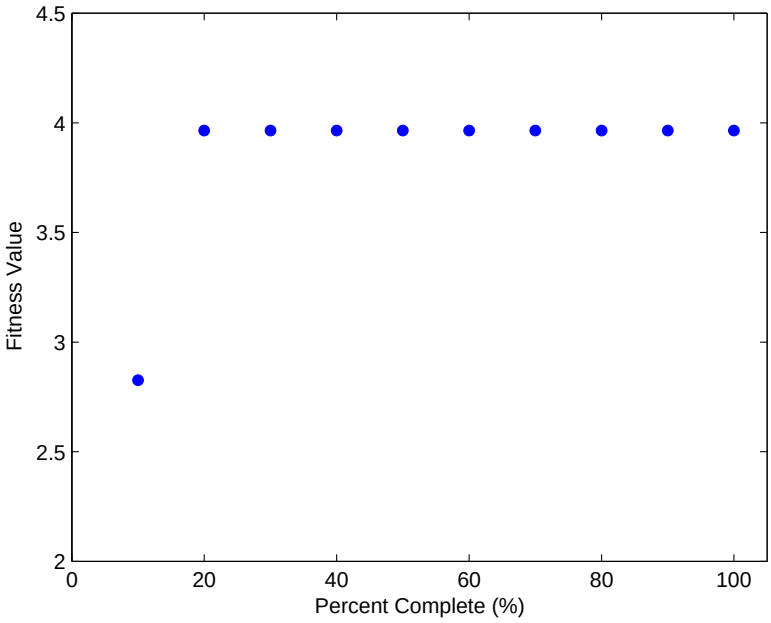


Figure 4.7: The rate of improvement of the population when the evolutionary algorithm found IFS_3.96_100707

Table 4.3: IFS_5.89_170707

a	b	e	c	d	f
-0.471221	0.155521	0.597505	0.234312	0.296413	0.180397
-0.153060	0.285571	0.284315	0.534090	-0.180104	0.292743
-0.542475	0.003537	0.574609	-0.505343	-0.056521	0.915391

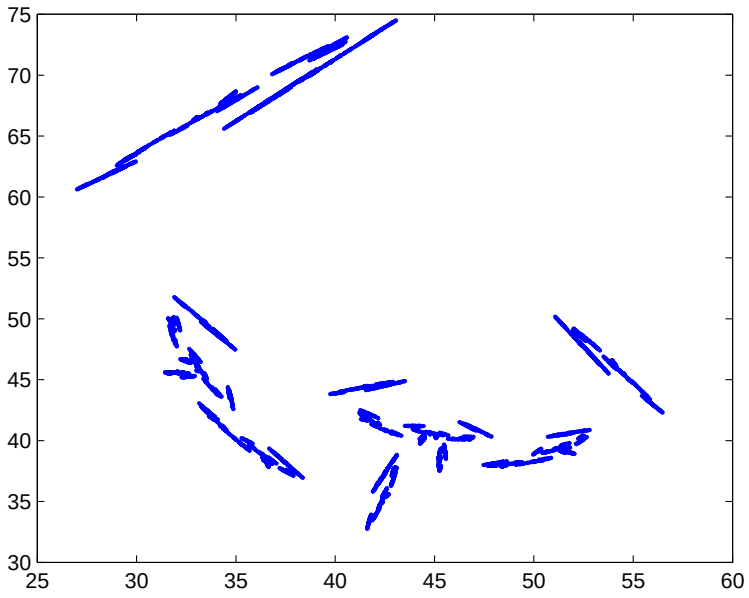


Figure 4.8: A plot of IFS_5.89_170707 with 10 000 points

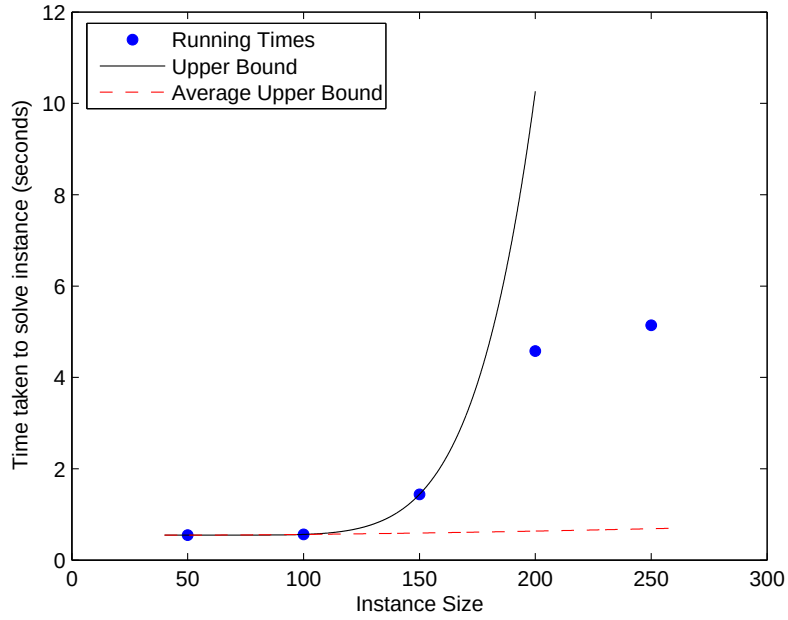


Figure 4.9: Upper bound of the average running times for IFS_5.89_170707

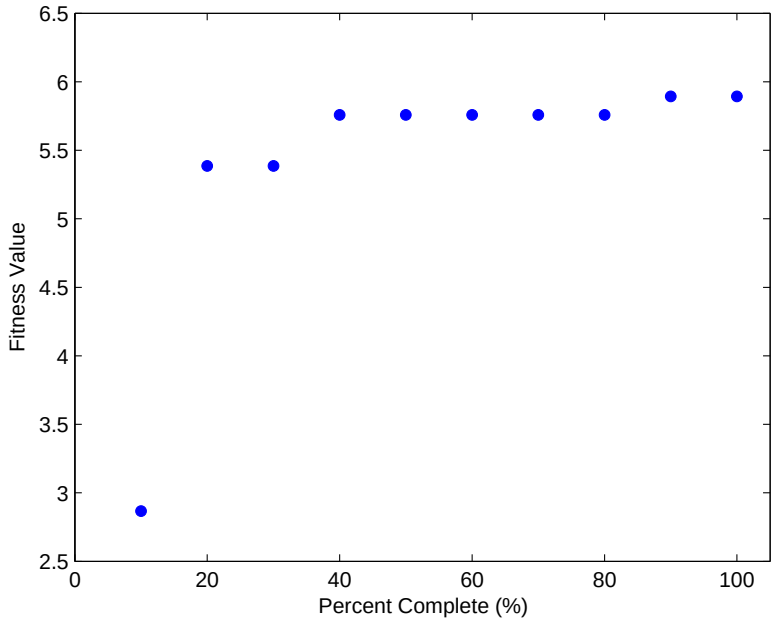


Figure 4.10: The rate of improvement of the population when the evolutionary algorithm found IFS_5.89_170707

Table 4.4: IFS_12.65_250707

a	b	e	c	d	f
-0.109107	-0.675755	0.834425	-0.408667	0.082710	0.791837
0.121939	-0.712331	0.763521	-0.114059	0.407411	0.291033
0.439438	0.156083	0.258423	0.149936	-0.155689	0.430117

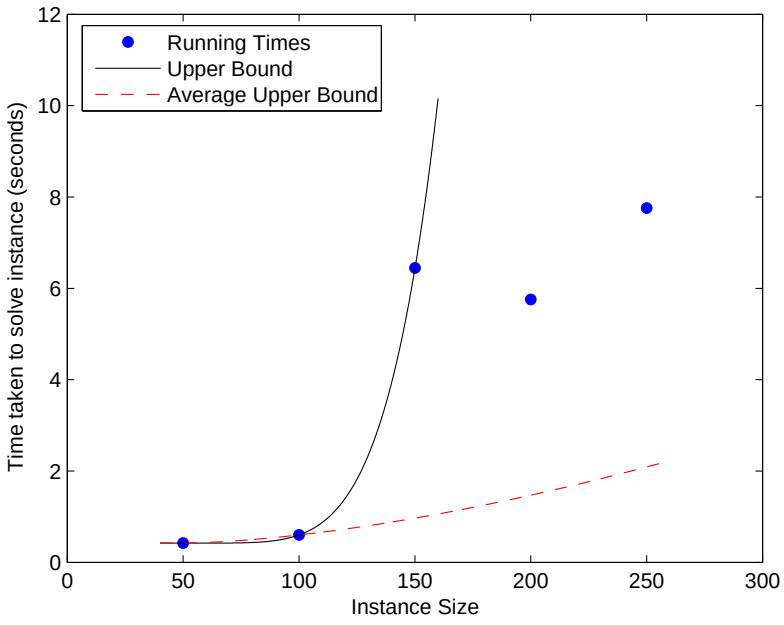


Figure 4.11: Upper bound of the average running times for IFS_12.65_250707

Table 4.5: IFS_2.29_250907

a	b	e	c	d	f
0.082702	0.075030	0.672350	0.426448	-0.350838	0.533666
0.214194	-0.403997	0.497661	0.045338	0.646384	0.299027
0.477439	-0.635604	0.293430	0.081912	-0.143925	0.781368
-0.035623	0.555388	0.385679	0.131212	0.069854	0.470052
-0.621264	0.634354	0.807574	0.242920	-0.366756	0.615292
-0.028696	0.243309	0.076081	0.038292	-0.733554	0.722176

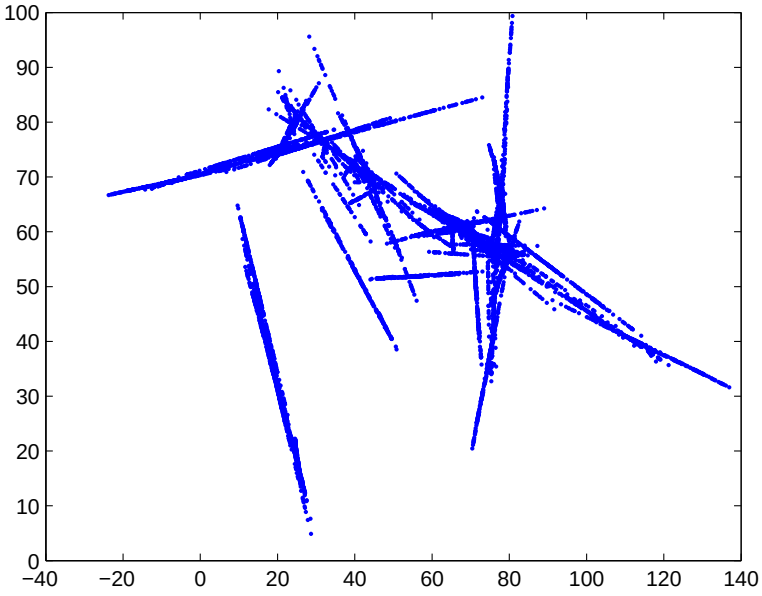


Figure 4.12: A plot of IFS_2.29_250907 with 10 000 points

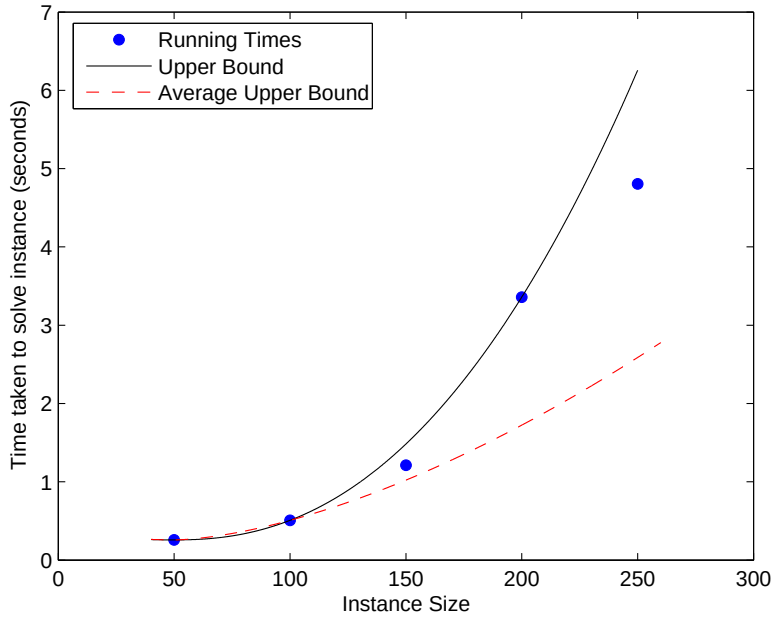


Figure 4.13: Upper bound of the average running times for IFS_2.29_250907

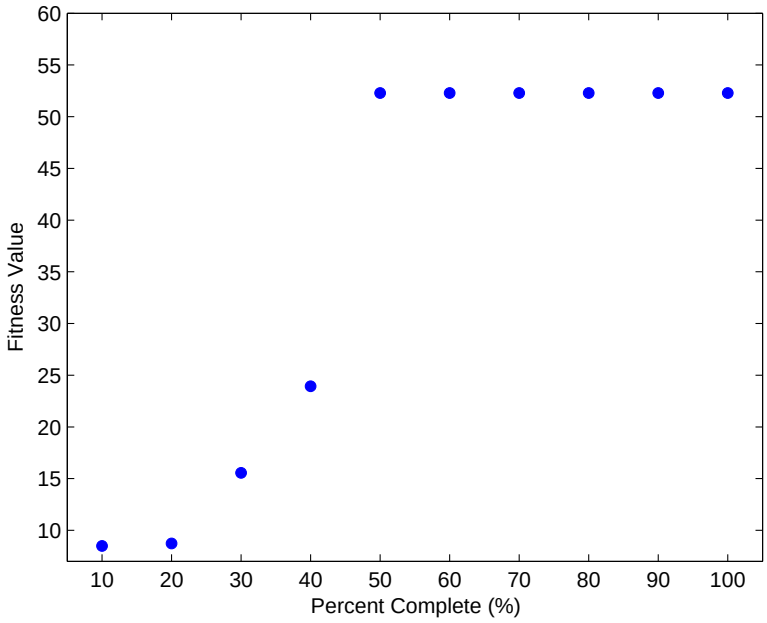


Figure 4.14: The rate of improvement of the population when the evolutionary algorithm found IFS_2.29_250907

Table 4.6: IFS_2.92_041007

a	b	e	c	d	f
-0.242148	0.124551	0.734454	0.690120	-0.643649	0.174443
-0.921490	0.611179	0.987295	0.141245	-0.071454	0.406653
-0.338204	0.063917	0.614937	0.167724	-0.203444	0.101971
-0.239089	0.636728	0.285769	-0.189496	0.898275	0.223096
0.636261	-0.290140	0.314671	-0.295176	0.491945	0.750215
0.000632	-0.128834	0.796600	-0.117658	-0.279169	0.547944

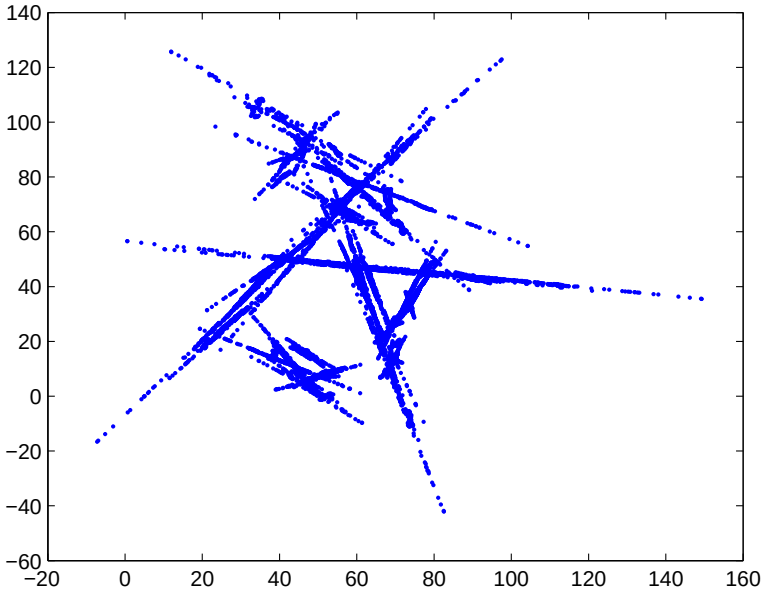


Figure 4.15: A plot of IFS_2.92_041007 with 10 000 points

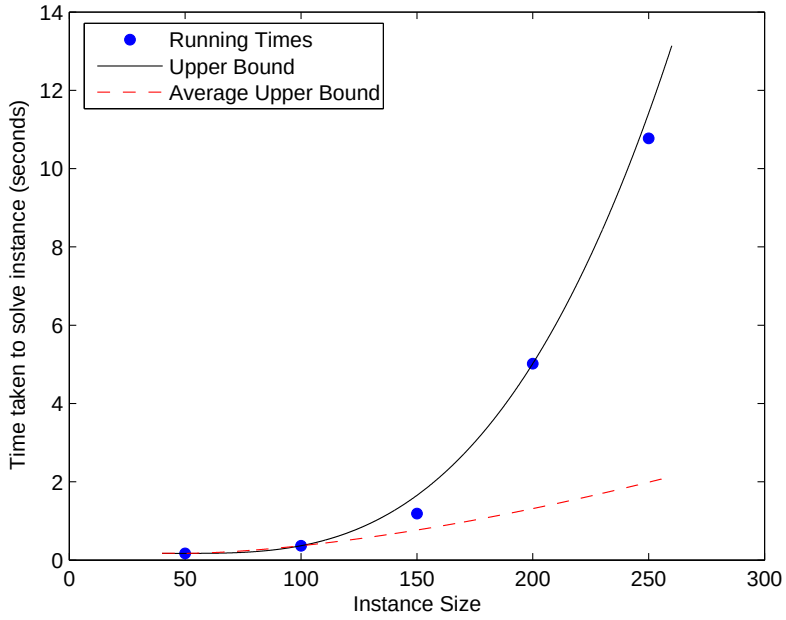


Figure 4.16: Upper bound of the average running times for IFS_2.92_041007

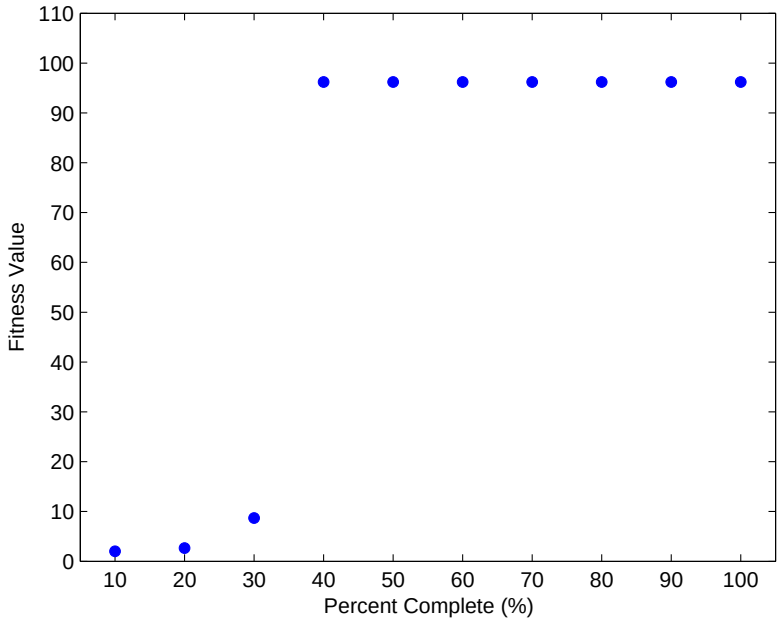


Figure 4.17: The rate of improvement of the population when the evolutionary algorithm found IFS_2.92_041007

Axiom: CBAF+--FF!FFF
Angle: 8
A → ABCFFFF!
B → +BC-
C → BAC!FFFF-+

Table 4.7: LS_1.36_011007

Axiom: [B+]CABBAC
Angle: 8
A → -CGCC!CFBF
B → A[AF+FC]B!
C → CC++G[GC]B

Table 4.8: LS_2.66_081007

4.3 L-Systems

While the IFS evolutionary algorithm was being run and modified, a similar algorithm which evolves L-Systems was being written. The L-System evolutionary algorithm was not run until the final version of the fitness function was decided. This section describes two of the most fit L-Systems that were found.

4.3.1 LS_1.36_011007

Fractal LS_1.36_011007 is a fractal whose upper bound of its running times is less than the upper bound of the average case, for the first five instances (Figure 4.19) yet, as will be shown in Section 5.2, one of its larger instances can be modified to become more difficult to solve than in the average case.

The test run which produced this fractal had 20 parents and 300 generations. The axiom and rules are shown in Table 4.7 with the running times in Table A.9 and the graph of the upper bound is shown in Figure 4.19.

4.3.2 LS_2.66_081007

Another test run with 20 parents and 300 generations produced fractal LS_2.66_081007. The axiom and rules are shown in Table 4.8 with the running times in Table A.10 and the graph of the upper bound is shown in Figure 4.21.

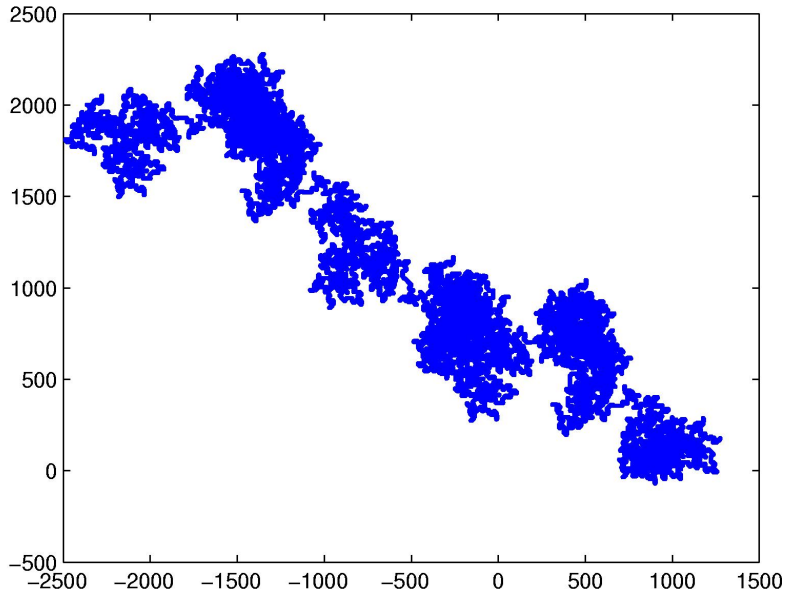


Figure 4.18: A plot of LS_1.36_011007 of order 11 (185 474 points).

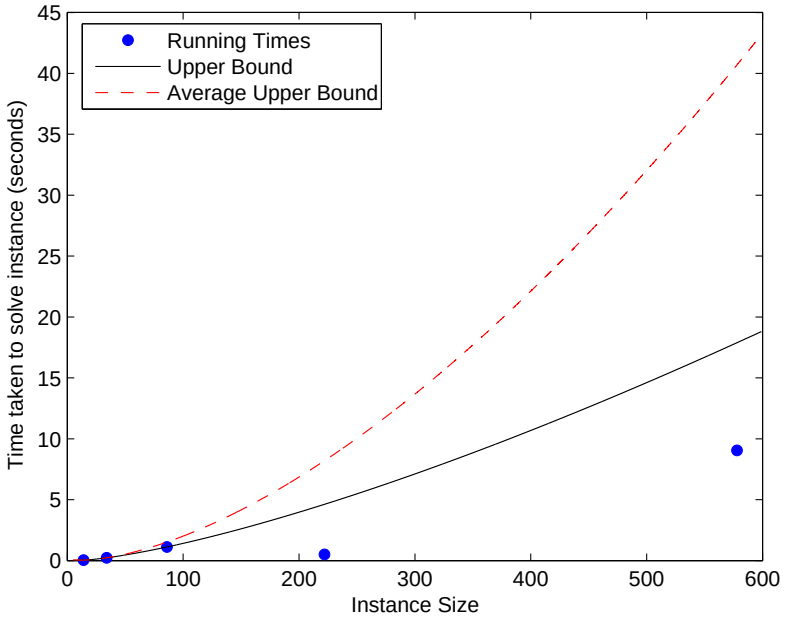


Figure 4.19: Upper bound of the average running times for LS_1.36_011007

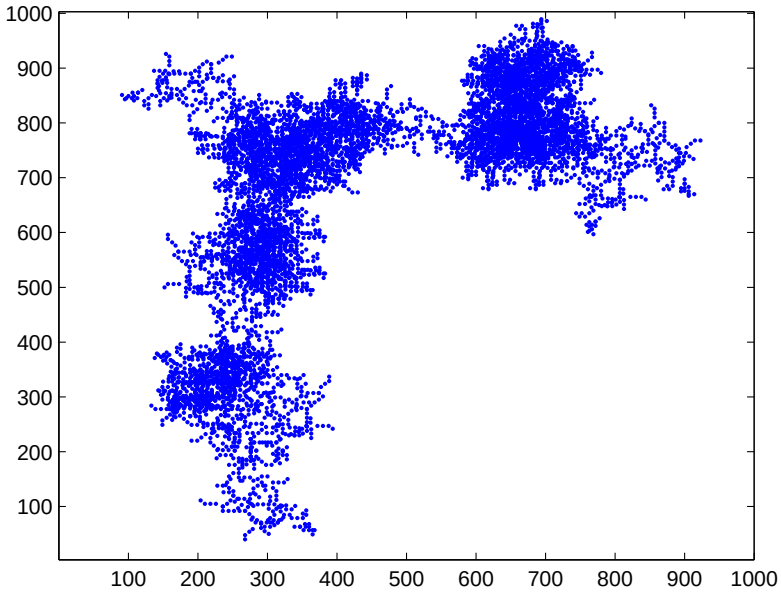


Figure 4.20: A plot of LS_2.66_081007 of order 6 (8 372 points).

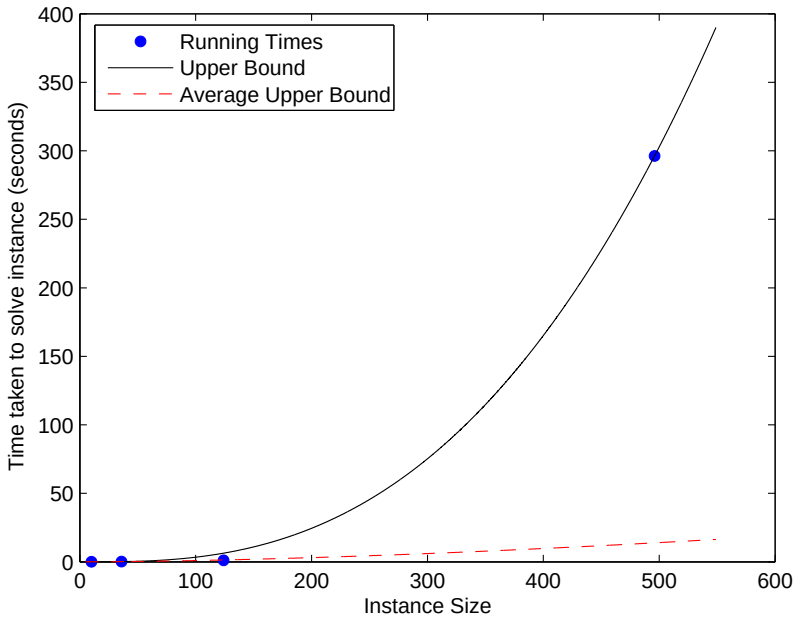


Figure 4.21: Upper bound of the average running times for LS_2.66_081007

Chapter 5

Analysis of Results

Now that the evolutionary algorithm has evolved some instances, this section will take a closer look at some of the most fit individuals (fractals) and see how *Concorde* handles the instances generated by these fractals.

Firstly, the fractals are used to create instances that are larger than what was used when running the evolutionary algorithm. It will be shown that one of the fractals does not continue to generate difficult instances, but another does.

Secondly, another idea is tested. Once the fractals are used to create instances, it may be possible to modify the instance slightly to make it more difficult solve. One possible method of modifying the instance is explored and the results of using the new instances are given.

5.1 Prediction of Running Times

When determining the fitness of an iterated function system, the evolutionary algorithm only looked at the time required to solve instances the IFS generated of sizes 50 to 250. The issues to resolve now are, whether the larger instances are also difficult to solve and how good the upper bound estimate is.

When creating an instance, a prediction will be made as to how long it will take *Concorde* to solve it. These predictions were computed using the equation of the upper bound (see Chapter A) used by the evolutionary algorithm when determining the fitness of the fractals.

Table 5.1 shows the predicted and actual running times required by

Concorde to solve the larger instances which were generated by fractal IFS_2.29_250907. Table 5.2 shows similar results for fractal IFS_2.92_041007.

It seems that sometimes the prediction is more than the actual, while other times it is less. Table 5.3 shows a comparison between these new times and the average case. It shows that fractal IFS_2.29_250907 does not produce difficult instances that often, however fractal IFS_2.92_041007 does seem to produce difficult instances more regularly.

Table 5.4 shows the predicted and actual running times required by *Concorde* to solve the larger instances which where generated by fractal LS_1.36_011007. Only two instance sizes were tested as the fractals of larger order have a much larger number of nodes and there was not enough time to run *Concorde* on them[1].

Table 5.4 shows that for an instance of size 1510, it took *Concorde* 1670.50 seconds to solve it. Table 5.3 shows that on average it takes *Concorde* 1655.62 seconds to solve an instance of size 1500. So the fractal-generated instance was not too difficult. However, it will be shown in the next section that this instance can be modified to become more difficult to solve.

It should be noted that results for fractal LS_2.66_081007 are not given here since fractals of order 5 or more took a longer amount of time than was available towards the end of this project. The fractal of order 5 has 2032 nodes and a result was not obtained after a couple of days.

5.2 Modifying the Generated Instances

As mentioned in Section 3.5, values of $0.05, 0.1$ and 0.2 were used for an ε which specifies the distance any point will be moved from its original position at most.

Making these changes to fractals produced by iterated function systems sometimes makes the instance harder to solve. For the L-System, the instance became harder to solve only when there were 1510 cities. It appears this modification has more of an affect on iterated function system instances than it does on L-System instances.

[1] One particular instance had run for over a week before it was decided to terminate the program

Instance Size	Running Times (seconds)		Error (%)
	Prediction	Actual	
300	10.26	8.45	-17.5
500	38.73	40.00	+3.3
800	124.35	68.34	-45.0
1000	213.60	65.26	-120.0
1500	562.62	1178.92	+109.5

Table 5.1: The predicted and actual running times of solving larger instances generated by fractal IFS_2.29_250907.

Instance Size	Running Times (seconds)		Error (%)
	Prediction	Actual	
300	27.83	10.28	-63.1
400	73.95	61.70	-16.6
500	153.86	117.70	+23.5
800	683.67	1043.00	+52.6
1000	1363.84	2585.40	+47.2
1500	4691.88	2688.78	-42.7

Table 5.2: The predicted and actual running times of solving larger instances generated by fractal IFS_2.92_041007.

Instance Size	Average Running Times	Actual Running Times	
		IFS_2.29_250907	IFS_2.92_041007
300	12.69	8.45	10.28
400	24.03	22.32	61.70
500	132.91	40.00	117.70
800	221.56	68.34	1043.00
1000	1812.82	65.26	2585.40
1500	1655.62	1178.92	2688.78

Table 5.3: Comparison between the average and actual running times (IFS) for larger instance sizes.

Order	Instance Size	Running Times (seconds)		Error (%)
		Prediction	Actual	
6	1510	67.53	1670.50	-17.5
7	3950	252.46	$>$ 30780.47	+3.3

Table 5.4: The predicted and actual running times of solving larger instances generated by fractal LS_1.36_011007.

Instance Size	Running Times (seconds)			
	Actual ($\varepsilon = 0$)	$\varepsilon = 0.05$	$\varepsilon = 0.1$	$\varepsilon = 0.2$
50	0.17	0.17	0.16	0.19
100	0.37	0.48	0.41	0.53
150	1.19	1.56	1.02	1.24
200	5.02	4.74	3.69	6.66
250	10.77	5.84	5.86	4.50
300	10.28	14.46	21.13	16.79
400	61.70	108.44	96.33	30.92
500	117.70	101.83	109.05	63.08
800	1043.00	649.53	692.16	4849.75
1000	2585.40	2133.17	$>$ 28800.00	2240.41

Table 5.5: The times required to solve the instances after they were modified so that each point has 'moved' from their original position by a small amount. The original instances were generated by fractal IFS_2.92_041007.

Order	Instance Size	Running Times (seconds)			
		Actual ($\varepsilon = 0$)	$\varepsilon = 0.05$	$\varepsilon = 0.1$	$\varepsilon = 0.2$
1	14	0.04	0.04	0.03	0.03
2	34	0.23	0.21	0.23	0.25
3	86	1.12	0.90	0.87	1.46
4	222	0.51	0.62	0.62	0.60
5	578	9.05	10.09	10.48	12.30
6	1510	1670.50	3185.24	2456.19	2619.50

Table 5.6: The times required to solve the instances after they were modified so that each point has 'moved' from their original position by a small amount. The original instances were generated by fractal LS_1.36_011007.

Chapter 6

Conclusions

In this thesis, it was shown that it is possible to evolve difficult to solve instances, in particular for the program *Concorde*. This project used evolutionary algorithms to evolve instances in the effort to find some difficult to solve instances. The instances themselves were constructed using fractals. In doing so, it is possible to describe classes of instances which take *Concorde* longer than on average to solve.

As this area of research is relatively new, not much information was found in the literature suggesting good fitness functions. Evolutionary algorithms, on the other hand has been extensively researched and still is. This project inevitably required time to experiment different fitness functions before one was found which seemed satisfactory. The evolutionary algorithms themselves also required some modifications to the parent selection mechanism and crossover functions.

The fractal IFS_2.92_041007 was found to produce the more difficult instances than the others. With a minimum upper bound of $\mathcal{O}(2.92)$ it survived the evolutionary competition to become the most fit in a population of 30. Time did not allow much extensive testing for the L-Systems. It seems that for fractal LS_1.36_011007 difficult instances are not generated except for the modified larger instances. For fractal LS_2.66_081007 it may be possible that it does produce difficult instances for the larger sizes, but further testing is needed.

Another interesting result is that when the instances, generated by iterated function systems, are modified so that each point (x, y) is replaced by another point in the range $(x \pm \varepsilon, y \pm \varepsilon)$ for some small ε (i.e. the points are

'moved' by a small distance from their original position), *Concorde* sometimes take a longer time to solve the modified instance than required to solve the original.

The same situation did not occur for instances generated by L-Systems. For almost all instances, the modification did not have much of an effect. Although, more tests should be performed before this idea can be accepted or rejected. The larger instances generated by the L-Systems seem to be much more difficult to solve already without modification.

It also seems that the iterated function systems, which produce the more difficult instances, describe fractals that seem to have a 'star' appearance. As seen in Figure 4.15, fractal IFS_2.92_041007 consists of many straight 'line' segments crossing over each other and none are parallel to any other. A similar situation arises for fractal IFS_2.29_250907 (Figure 4.12). It may be possible that this is one of the characteristics of difficult instances.

For L-Systems, their common characteristics seem that the nodes are located in clusters. These differences between the instances generated by iterated function systems and L-Systems may be related to the reason why modifying the instances, by moving each point, was more effective for the iterated function systems.

Overall, the results do show that it is possible to evolve instances which are difficult for *Concorde*, a combinatorial optimisation program, to solve. Furthermore, in some cases, the instance can be modified a little (not too much so as not to loose the whole structure) to make it more difficult to solve. This situation arises more frequently with iterated function systems than with L-Systems.

Chapter 7

Reflections and Further Work

This project is a Software Engineering project and the aim was to apply a computation based attack on a combinatorial optimisation program to find instances that are difficult for it to solve.

The instances created were described using fractals. The fractals themselves were modelled using iterated function systems and L-Systems. This was the author's supervisor's suggestion and no time was spent researching other possible techniques of describing a set of points on the $x - y$ plane. A possible work for the future is to use other methods of describing instances for *Concorde*.

The results do seem favourable. They suggest that the hypothesis that "it is possible to evolve difficult instances" is true, and further, it is possible to evolve difficult structures of instances. The fractals that were found can be used to create large instances that are difficult to solve than on average. Some instances can be made even more difficult when modified a little by randomly moving each point a small distance from their original position.

Although the program *Concorde* was used in this project, the technique used is not tightly coupled to *Concorde*'s code in any way to prevent it being used for testing other programs[1]. Instances are created and the program is timed to measure how long it takes to solve the instance. So a possible future development is to perform a similar analysis on other programs. In fact, the

[1]While it is possible to use a timer when the program is running, the *Concorde* program already outputs this information. So it was decided to use the *Concorde* programmers' definition of *Concorde*'s running time.

aim of this project was to determine the merit of using the evolutionary algorithm technique of analysing an arbitrary algorithm.

Throughout the duration of this project many decisions, regarding the design of the algorithms and test conditions, had to be made and changed. One decision was to keep the number of functions in each IFS constant. Dasgupta *et al.* [26] have designed an evolutionary algorithm which evolves IFS objects where the number of functions vary. Anyone wishing to continue this research would have to consider the restrictions the chosen recombination operator imposes.

In an attempt to find 'simple' fractals, the number of functions in the iterated function systems and the number of rewrite rules in the L-Systems were kept small. It may be that one requires more 'complex' fractals in order to generate more difficult instances.

Similarly, there are different possible variations of recombination operators to use on L-Systems. Chaiyaratana and Zalzala [27] suggested a technique which utilises both uniform crossover and two-point crossover in the same population. It was decided not to do this, instead the algorithms were designed to test the uniform crossover technique only. A future development could be to use a two-point crossover and later on combine both techniques.

Due to the abstract nature of evolutionary algorithms, there are many different ways to define fitness functions. The problem of determining which fitness function is better than others is another area of research altogether.

The fitness functions used was modified twice during the project. Initially, the upper bound solely determined the fitness, then after some less favourable results, the total running time was used as the fitness function. Still the results were not as good as desired. Finally, the fitness function was modified to the form described in Section 3.2. The fractals found did not always produced difficult instances, so the fitness function may still need more modifications.

There are clearly many other possible fitness functions that can be used. Notice that the fitness functions used in this project are not dependent on the *Concorde* algorithm nor were they dependent on how the fractals were modelled. They all can work on any traveling salesperson problem solver and any representation of fractals.

One of the reasons the results are not as favourable as hoped for may be because the fitness of the fractals were based on the running times of

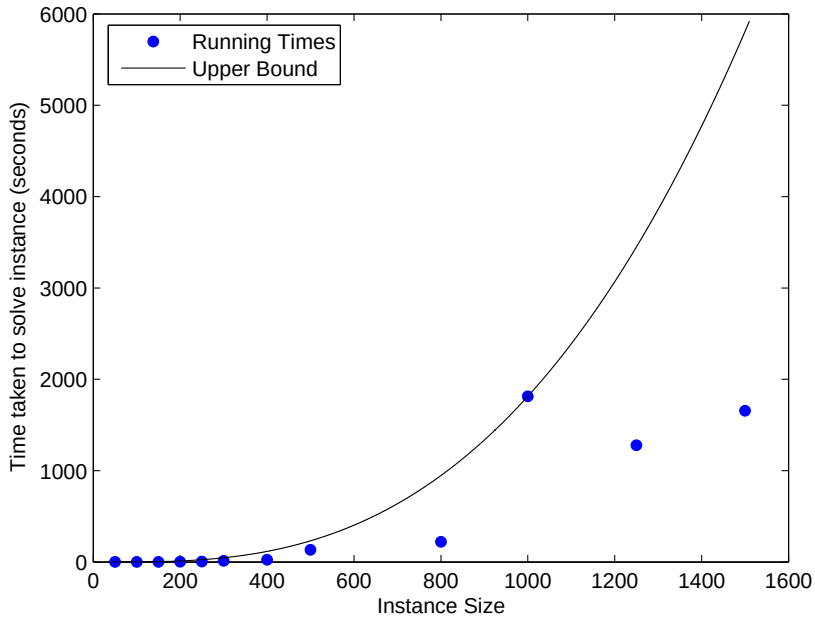


Figure 7.1: Upper bound of the Average Running Times, including the times to solve the larger instances.

small instances. As seen in Figure 4.1 and Figure 7.1 the average running time for smaller instances appear to be modelled by the upper bound more consistently, whereas the running times for the larger instances appear less consistent and does not closely follow a polynomial. The data used to create the graph came from Table A.1 and Table 5.3.

It is not suggested that theoretical analysis is no longer needed. Indeed, the technique presented in this project can be used in conjunction with other theoretical analyses of an algorithm to provide practical evidence or proof to support the theory.

There are some circumstances where theoretically proving the complexity of an algorithm has become too difficult [3]. In these cases, an empirical analysis is useful and the technique presented in this report can always be an option.

Appendix A

Data obtained during testing

The actual data obtained, corresponding to the graphs and analysis presented in this report are provided in this chapter.

The equation of the upper bounds are denoted as $t(n)$ and is interpreted as the running time t, given an instance size n.

A.1 Average/Base Case

The equation of the upper bound of the average time to solve random instances (Table A.1) is

$$t(n) = 0.106 + 0.545\left(\frac{n-50}{50}\right)^{1.610107}$$

A.2 Fractals

Table A.2 shows the equation of the upper bound of each fractal (iterated function system and L-System) mentioned in this report and their corresponding table which contains the running times (which is used to compute the upper bound).

Instance Size (number of points)	Average time to solve (seconds)
50	0.106
100	0.651
150	1.568
200	3.235
250	5.183

Table A.1: Average running times for random instances.

Fractal Name	Upper Bound	Table Reference
Iterated Function Systems:		
IFS_1.99_200507	$t(n) = 1.157 + 0.782\left(\frac{n-50}{50}\right)^{1.992493}$	A.3
IFS_3.96_100707	$t(n) = 0.313 + 0.031\left(\frac{n-50}{50}\right)^{3.964844}$	A.4
IFS_5.89_170707	$t(n) = 0.547 + 0.015\left(\frac{n-50}{50}\right)^{5.892639}$	A.5
IFS_12.65_250707	$t(n) = 0.359 + 0.001\left(\frac{n-50}{50}\right)^{12.648926}$	A.6
IFS_2.29_250907	$t(n) = 0.258 + 0.250\left(\frac{n-50}{50}\right)^{2.292175}$	A.7
IFS_2.92_041007	$t(n) = 0.172 + 0.195\left(\frac{n-50}{50}\right)^{2.924805}$	A.8
L-Systems:		
LS_1.36_011007	$t(n) = 0.039 + 0.188\left(\frac{n-14}{20}\right)^{1.363525}$	A.9
LS_2.66_081007	$t(n) = 0.032 + 0.124\left(\frac{n-10}{26}\right)^{2.656555}$	A.10

Table A.2: Summary of each IFS's upper bound and the table which contains the running times.

Instance Size (number of points)	Average time to solve (seconds)
50	1.157
100	1.938
150	4.266
200	6.282
250	6.133

Table A.3: Average running times for IFS_1.99_200507.

Instance Size (number of points)	Average time to solve (seconds)
50	0.305
100	0.360
150	0.696
200	1.844
250	2.837

Table A.4: Average running times for IFS_3.96_100707.

Instance Size (number of points)	Average time to solve (seconds)
50	0.547
100	0.562
150	1.438
200	4.578
250	5.141

Table A.5: Average running times for IFS_5.89_170707.

Instance Size (number of points)	Average time to solve (seconds)
50	0.359
100	0.360
150	6.781
200	1.813
250	3.281

Table A.6: Average running times for IFS_12.65_250707.

Instance Size (number of points)	Average time to solve (seconds)
50	0.258
100	0.508
150	1.212
200	3.359
250	4.805

Table A.7: Average running times for IFS_2.29_250907.

Instance Size (number of points)	Average time to solve (seconds)
50	0.172
100	0.367
150	1.188
200	5.018
250	10.774

Table A.8: Average running times for IFS_2.92_041007.

Order	Instance Size (number of points)	Average time to solve (seconds)
1	14	0.039
2	34	0.227
3	86	1.117
4	222	0.508
5	578	9.046

Table A.9: Average running times for LS_1.36_011007.

Order	Instance Size (number of points)	Average time to solve (seconds)
1	10	0.032
2	36	0.156
3	124	1.079
4	496	296.281

Table A.10: Average running times for LS_2.66_081007.

Appendix B

Design of the *Concorde* program

B.1 The Traveling Salesperson Problem

The Traveling Salesperson Problem (TSP) is a *combinatorial optimisation problem* because it has the form "minimise $f(x)$ where x is a feasible solution" and each feasible solution has combinatorial features [28]. The TSP belongs to the class of problems known as *NP-Complete* problems [29].

The TSP derives its name from the following scenario: suppose that the vertices of a graph G represent a collection of cities and the weight on each edge represents the cost of traveling between the two cities connected to the edge. Suppose now there is a salesperson who would like to travel to all the cities, visiting each city once. The salesperson will most likely want to follow a route of minimum cost [30].

In general, there is no algorithm that can solve this problem in polynomial time [30] and so we use algorithms that provide approximate solutions (which include *heuristic* and *metaheuristic* algorithms).

B.2 The *Concorde* Algorithm

Concorde is the program being studied in this project. It attempts to solve the Traveling Salesperson Problem. The program uses the branch-and-cut approach to create a linear programming representation of the TSP instance,

which is then solved using a linear programming solver (*Qsopt*[1] is the solver used in this project).

Concorde approaches the TSP from the point of view that it can be described as a *Linear Programming* (LP) problem. These problems are optimization problems in which the objective function and constraints are all linear [31]. Generally, the aim is to maximise a linear objective function subject to linear constraints. This section briefly explains how an association between TSP and LP is possible.

Denote $dist(i, j)$ to represent the distance between city i and city j and let $x(i, j) = 1$ if the road between i and j is included in the solution, otherwise $x(i, j) = 0$. Hence the length of any tour (solution) is calculated as [6]

$$dist(1, 2)x(1, 2) + dist(1, 3)x(1, 3) + \cdots + dist(n - 1, n)x(n - 1, n)$$

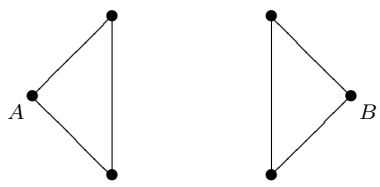
The designers of *Concorde* noticed that this formula is a weighted sum of the variables and so can be used as the objective function for a linear programming problem. However this definition is not complete without some constraints being placed on the values of $x(i, j)$.

Notice that for each city only *two* roads connected to it can be selected (one to enter the city and one to leave it). Hence the following sum must hold

$$x(i, 1) + x(i, 2) + \cdots + x(i, i - 1) + x(i, i + 1) + \cdots + x(i, n) = 2$$

for every city i.

Finally, there is one more constraint that needs to be defined. Consider the attempted TSP solution below



Here, there is no path from city A to city B. In this example, there are two *sub-tours* and so it must be ensured that sub-tours are never created. Let

[1] Qsopt can be found on the Concorde website.

S be any collection of cities (having at least two cities and not containing every city). Then to forbid a sub-tour through the cities in S it must be ensured that the number of roads entering or leaving S is at least two. Thus,

$$\sum_{i \in S, j \notin S} x(i,j) \geqslant 2$$

Concorde defines the Traveling Salesperson Problem as the following version of linear programming known as the *fractional 2-matching LP* [32]

Minimize

$dist(1,2)x(1,2) + dist(1,3)x(1,3) + \cdots + dist(n-1,n)x(n-1,n)$

Subject To

$$x(1,2) + x(1,3) + \cdots + x(1,n) = 2$$
$$x(1,2) + x(2,3) + \cdots + x(2,n) = 2$$
$$\vdots$$
$$x(i,1) + x(i,2) + \cdots + x(i,i-1) + x(i,i+1) + \cdots + x(i,n) = 2$$
$$\textstyle\sum_{i \in S, j \notin S} x(i,j) \geqslant 2$$
$$x(i,j) \geqslant 0, \quad \forall i \neq j \; x(i,j) \leqslant 1, \quad \forall i \neq j$$

So *Concorde* utilises a linear programming algorithm as one of its methods when solving the traveling salesperson problem. There are additional techniques used by *Concorde* to assist with finding tours and checking if a generated solution is optimal or not, but they will not be discussed here because the programs written for this project interacts with *Concorde* via its interface and are not dependent on the exact algorithm implemented by *Concorde*. Visit the *Concorde* website [32] for more detailed information.

Appendix C

Plotting the attractor of an IFS

There are two methods to plot the fractal described by iterated function systems. One is a probabilistic method (usually called the chaos game) and the other is a deterministic method. When creating instances for *Concorde* to solve, the deterministic method was used with $\{(0, 0)\}$ as the initial set. This was to ensure that the same plot was always created. This allows the same TSP instance to be created every time.

C.1 The Algorithms

The chaos game starts with a random point (usually in the bi-unit square) then selects a function at random and applies it to the point. Another function is selected at random and is applied to the new point. This continues on until there are sufficient points. Algorithm 3 describes how the attractor of an iterated function system is approximated by 'playing' the chaos game. The algorithm was obtained from [11].

In contrast, the deterministic algorithm starts with a random set of points (may contain only one point, or many). The algorithm then applies each function to all the points in the set. This generates a new (larger) set of points. Again each function is applied to all the points in the new set. This continues on until there are sufficient points. Algorithm 4 describes how the attractor of an iterated function system is approximated by applying the deterministic method. The algorithm was obtained from [33].

Algorithm 3: Plotting the attractor of an IFS by playing the chaos game.

```
1 {F_0(x,y), F_1(x,y), ..., F_n(x,y)} is the iterated function system
2 (x,y) ← a random point in the bi-unit square [0,1]
3 repeat
4 |   i ← a random integer from 1 to n inclusive
5 |   (x,y) ← F_i(x,y)
6 |   plot (x,y) except during the first 20 iterations
7 until Plotted enough points
```

Algorithm 4: Plotting the attractor of an IFS by the deterministic method.

```
1  {F_0(x,y), F_1(x,y), ..., F_n(x,y)} is the iterated function system
2  S ← {(x_0,y_0)} where (x_0,y_0) is a random point in the bi-unit square
3  repeat
4  |   i ← 0
5  |   S' ← ∅
   |   /* Iterate over the points in S.                          */
6  |   repeat
7  |   |   j ← 0
   |   |   /* Iterate over the functions in the IFS.             */
8  |   |   repeat
9  |   |   |   S' ← S' ∪ {F_j(x_i,y_i)} where (x_i,y_i) ∈ S
10 |   |   |   j ← (j+1)
11 |   |   until (j < n) ∨ (S' has enough points)
12 |   |   i ← (i+1)
13 |   until (i < |S|) ∨ (S' has enough points)
14 |   S ← S'
15 until S has enough points
16 plot each point (x,y) ∈ S
```

Glossary

A

Affine Map A linear map that performs scaling, rotation, translation and shear. An affine map on $\mathbb{R}^2$ can be represented as [34]:

$$F\begin{pmatrix} x \\ y \end{pmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

where $a, b, c, d, e, f \in \mathbb{R}$.

Attractor A set of points A such that for a given iterated function system $\{F_i \,|\, i = 1, \ldots, n\}$ the following property holds

$$A = \bigcup_{i=1}^{n} F_i(A)$$

This set is usually contains an infinite number of points (but does *not* necessarily contain all points in $\mathbb{R}^2$) and is usually a fractal [8]., p. 7.

C

Chaos Game A probabilistic method of plotting the attractor of an iterated function system by applying a random function (from the IFS) to a point, then applying another random function to the current point, then applying another random function to the current point, and so on. The result is the set of all generated points., p. 71.

Combinatorial Optimisation Problem A problem which has the form "minimise $f(x)$ where x is a feasible solution" and each feasible solution has combinatorial features [28]. Here, f is called the *objective function*., p. 67.

Concorde A Traveling Salesperson Problem solver that has provided some of the world records in the exact solution of TSP instances. For more information, visit its website [6]., p. 67.

Contraction Mapping An affine transformation T in which the scale is reduced. That is,

$$T \text{ is a contraction} \iff \forall x, y \in S,\ d(x, y) > d(T(x), T(y))$$

where $d : S \times S \to \mathbb{R}$ is a *distance* function [35]., p. 7.

D

Deterministic Method (of Creating Fractals) A method of plotting the attractor of an iterated function system by applying all functions (from the IFS) to a set of points creating a new set of points, then applying all functions to the new set, then applying all functions to the newly created set, and so on. The result is the final generated set., p. 71.

E

Evolutionary Algorithm (EA) A technique to solve optimisation problems by starting with a random group of potential solutions then producing new solutions by merging two (or more) selected solutions and making small changes to some of the newly created solutions, akin to the biological process of reproduction and evolution., p. 13.

H

Heuristic Algorithm A type of algorithm that provides approximate ('almost optimal') solutions. Some examples include the *greedy method* and *hill climbing local search*. There are other methods called *Meta-heuristic* algorithms which combines heuristic tools [28]., p. 67.

I

Iterated Function System (IFS) A method of describing fractals. An IFS is a set of maps that act on a metric space S for which a metric d is defined. The set of maps must be finite and contain only affine, linear contractive maps [26]., p. 6.

L

Linear Map/Transformation A function $T : \mathbb{R}^n \to R^m$ is a linear map (or linear transformation) from $\mathbb{R}^n$ to $\mathbb{R}^m$ if [36]:

$$\begin{array}{lll} (i) & T(u) + T(v) = T(u+v) & \text{for all } u, v \in \mathbb{R}^n \\ (ii) & \lambda T(u) = T(\lambda u) & \text{for all } u \in \mathbb{R}^n, \lambda \in \mathbb{R} \end{array}$$

The Linear transformation T can be represented by an $m \times n$ matrix.

Linear Programming (LP) Optimisation problems in which the objective function and constraints are linear [31]. Generally, you want to maximize the function:

$$z = a_{01}x_1 + a_{02}x_2 + \cdots + a_{0n}x_n$$

Subject to some or all of the primary constraints:

$$x_1 \geqslant 0, \quad x_2 \geqslant 0, \quad \cdots \quad , x_n \geqslant 0$$

and any additional constraints of the form:

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leqslant b_i \quad \text{for } i = 1, 2, \ldots, m.$$

The name arises from the fact that each constraint is represented by a linear function., p. 68.

M

Metaheuristic Algorithm A type of algorithm which combines *heuristic* tools in 'more sophisticated frameworks' [28]. Examples include *simulated annealing*, *evolutionary algorithms* and *tabu search*. Metaheuristic algorithms generally follow two steps; (1) Search for new solutions with the aid of previous data. (2) Evaluate the solutions found in Step 1 and use this information to guide the next search., p. 67.

T

Traveling Salesperson Problem (TSP) A combinatorial optimisation problem where a cycle is to be found from an arbitrary graph such

that every vertex is visited exactly once and the sum of the weights of the selected edges is a minimum. In this project, only *complete asymmetric* graphs (where there is an edge between every pair of vertices and $dist(u, v) = dist(v, u)$ for all vertices u, v) are considered., p. 67.

Bibliography

[1] E. Green, "Introduction to fractal theory," 1998. Last accessed: May 18, 2007.
URL=http://www.cs.wisc.edu/%7Eergreen/honors_thesis/contents.html.

[2] C. Cotta and P. Moscato, "A mixed evolutionary-statistical analysis of an algorithm's complexity," *Applied Mathematics Letters*, vol. 16, pp. 41–47, 2003.

[3] J. I. van Hemert, "Evolving combinatorial problem instances that are difficult to solve," *Evolutionary Computation*, vol. 14, no. 4, pp. 433–462, 2006.

[4] B. Kernighan and S. Lin, "An effective heuristic algorithm for the traveling salesman problem," *Operations Research*, vol. 21, pp. 498–516, 1973.

[5] P. Moscato and M. Norman, "An analysis of the performance of traveling salesman heuristics on infinite-size fractal instances in the euclidean plane," tech. rep., CeTAD - Universidad Nacional de La Plata, 1994.

[6] W. Cook, "The *Concorde* web site." Last accessed: March 1, 2007.
URL=http://www.tsp.gatech.edu/concorde/index.html.

[7] W. Cook, "*Concorde* benchmarks." Last accessed: March 1, 2007.
URL=http://www.tsp.gatech.edu/concorde/benchmarks/bench99.html.

[8] K. Falconer, *Fractal Geometry*. John Wiley & Sons, Ltd., second ed., 2003.

[9] P. Bourke, *An Introduction to Fractals*. The University of Western Australia, May 1991. Last accessed May 21, 2007.
URL=http://local.wasp.uwa.edu.au/~pbourke/fractals/fracintro/.

[10] Webopedia, *What is a fractal?* Internet.com, September 01, 1996. Last accessed May 21, 2007.
URL=http://www.webopedia.com/TERM/f/fractal.html.

[11] S. Draves, "The fractal frames," 1992. Last accessed April 30, 2007.
URL=http://flam3.com/flame.pdf.

[12] J. C. Hart, "Fractal image compression and recurrent iterated function systems," *IEEE Computer Graphics and Applications*, pp. 25–33, July 1996.

[13] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[14] D. J. Holliday, B. Peterson, and A. Samal, "Recognizing plants using stochastic L-Systems," *Proceedings. IEEE International Conference Image Processing, 1994*, vol. 1, pp. 183–187, 1994.

[15] P. Prusinkiewicz, "Graphical applications of l-systems," in *Proceedings on Graphics Interface '86/Vision Interface '86*, (Toronto, Ont., Canada, Canada), pp. 247–253, Canadian Information Processing Society, 1986.

[16] J. S. Hanan, *Parametric L-Systems and their application to the modelling and visualization of plants*. PhD thesis, Faculty of Graduate Studies and Research, University of Regina, Saskatchewan, 1992.

[17] D. A. Ashlock, K. M. Bryden, and S. P. Gent, "Evolution of L-Systems for compact virtual landscape generation," *The 2005 IEEE Congress on Evolutionary Computation*, vol. 3, pp. 2760–2767, 2005.

[18] P. Salvador, A. Nogueira, and R. Valadas, "Framework based on stochastic L-Systems for modeling IP traffic with multifractal behavior," tech. rep., Institute of Telecommunications / University of Aveiro, Portugal, 2003.

[19] A. Mariano, P. Moscato, and M. G. Norman, "Using L-Systems to generate arbitrarily large instances of the euclidian travelling salesman problem with known optimal tours," in *Anales del XXVII Simposio Brasileiro de Pesquisa Operacional*, (Vitoria, Brazil), Nov 1995.

[20] A. Eiben and J. Smith, *Introduction to Evolutionary Computing*, ch. What is an Evolutionary Algorithm? Natural Computing Series, 2003.

[21] D. Fogel, "What is evolutionary computation?," *Spectrum, IEEE*, vol. 37, pp. 26,28–32, Feb 2000.

[22] X. Song, J. Xu, and J. Zhang, "Evolutionary programming: The-state-of-the-art," *Intelligent Control and Automation, 2006. WCICA 2006. The Sixth World Congress on*, vol. 1, pp. 3296–3300, 2006.

[23] A. Konak, S. Kulturel-Konak, and A. Smith, "Minimum cost 2-edge-connected steiner graphs in rectilinear space: an evolutionary approach," *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, vol. 1, pp. 97–103, 2000.

[24] D. B. Fogel, "Evolving a checkers player without relying on human experience," *Intelligence*, vol. 11, no. 2, pp. 20–27, 2000.

[25] Wikipedia, *Evolutionary algorithm*. Wikipedia, The Free Encyclopedia, 2006. Last accessed March 6, 2007.
URL=http://en.wikipedia.org/wiki/Evolutionary_algorithm.

[26] D. Dasgupta, G. Hernandez, and F. Niño, "An evolutionary algorithm for fractal coding of binary images," *IEEE Transactions on Evolutionary Computing*, vol. 4, July 2000.

[27] N. Chaiyaratana and A. Zalzala, "Recent developments in evolutionary and genetic algorithms: theory and applications," *Second International Conference On Genetic Algorithms In Engineering Systems:Innovations And Applications*, pp. 270–277, 1997.

[28] T. Ibaraki and M. Yagiura, "On metaheuristic algorithms for combinatorial optimization problems," *Systems and Computers in Japan*, vol. 32, no. 3, pp. 33–55, 2001.

[29] C. H. Papadimitriou and K. Steiglitz, "Some complexity results for the traveling salesman problem," in *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, pp. 1–9, 1976.

[30] M. O. Albertson and J. P. Hutchinson, *Discrete Mathematics with Algorithms*. John Wiley & Sons, Inc., 1988.

[31] Wikipedia, *Linear programming*. Wikipedia, The Free Encyclopedia, 2006. Last accessed August 21, 2006.
URL=http://en.wikipedia.org/wiki/Linear_programming.

[32] W. Cook, "Subtour elimination (The *Concorde* web site)." Last accessed: March 10, 2007.
URL=http://www.tsp.gatech.edu/methods/opt/subtour.htm.

[33] R. T. Stevens, *Fractal Programming in C*. M&T Publishing, 1989.

[34] E. W. Weisstein, *Affine Transformation*. MathWorld–A Wolfram Web Resource, March 10, 2004. Last accessed April 27, 2007.
URL=http://mathworld.wolfram.com/AffineTransformation.html.

[35] E. W. Weisstein, *Geometric Contraction*. MathWorld–A Wolfram Web Resource, November 16, 2002. Last accessed April 27, 2007.
URL=http://mathworld.wolfram.com/GeometricContraction.html.

[36] H. Anton, *Elementary Linear Algebra*. Wiley, eighth ed., 2000.

[37] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator," *ACM Trans. on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.

Colophon

All programs created by the author for this project was written in the C programming language for the advantage in speed and also because the *Concorde* program was written in C. It was initially attempted to merge *Concorde*'s code with the evolutionary algorithm, but unforeseen problems meant this could not happen.

The code was modularized as much as possible with the modules partitioned into separate files containing related functions and `structures`. This was done to reduce the effort needed to convert the code into a C++ (an Object Oriented language) equivalent, if this will ever be done in the future. This change may occur since it would allow for easier modification to the code for the testing of other programs.

Random numbers were generated using the *Mersenne Twister*, coded by Makoto Matsumoto and Takuji Nishimura [37].

The code for plotting the attractor of an L-System was provided by Adrian Mariano (coauthor of [19]).

Matlab® Version 7.2.0.232 (R2006a) was used to plot the fractals and graphs.

All programs were run on an Intel® Core Duo (2×2.13GHz) processor running Windows XP Professional Version 2002, Service Pack 2 with 2 gigabytes of RAM. The programs (including *Concorde*) were compiled and run inside a Cygwin[1] environment (version 2.510.2.2).

Finally, this report was created using the LaTeX 2_ε typesetting system with BibTeX.

[1] Website: http://www.cygwin.com/ Last Accessed: September 7, 2007