Evolving L-Systems as an intelligent design approach to find classes of difficult-to-solve Traveling Salesman Problem instances

Farhan Ahammed[1,2] and Pablo Moscato[3]

[1] NICTA, Australia, Australian Technology Park, Level 5, 13 Garden Street, Eveleigh NSW 2015, Australia
[2] School of Information Technologies, The University of Sydney, NSW 2006, Australia
faha3615@it.usyd.edu.au
[3] School of Electrical Engineering and Computer Science, and Centre for Bioinformatics, Biomarker Discovery and Information-based Medicine, The University of Newcastle, NSW 2308, Australia
Pablo.Moscato@newcastle.edu.au

Abstract. The technique of computationally analysing a program by searching for instances which causes the program to run in its worst-case time is examined. *Concorde* [2], the state-of-the-art Traveling Salesperson Problem (TSP) solver, is the program used to test our approach. We seed our evolutionary approach with a fractal instance of the TSP, defined by a Lindenmayer system at a fixed order. The evolutionary algorithm produced modifications to the L-System rules such that the instances of the modified L-System become increasingly much harder for Concorde to solve to optimality. In some cases, while still having the same size, the evolved instances required a computation time which was 30,000 times greater than what was needed to solve the original instance that seeded the search. The success of this case study shows the potential of Evolutionary Search to provide new test-case scenarios for algorithms and their software implementations.

Keywords: Evolutionary Analysis of Algorithms, Memetic Design, fractals, L-Systems

1 Introduction

Suppose a theoretical analysis for a given algorithm is difficult due to its complex nature. It would be beneficial to describe which instances causes the algorithm to perform at unsatisfactory standards, that is, when the program takes more time than usual to reach the required conclusions. In these situations manually overriding the program may be a better option. For this to work, there needs to be some way to find these 'difficult-to-solve' instances.

The algorithm used in the *Concorde* TSP solver is used for the analysis. *Fractals* are used to define the TSP instances (ie. the positions of each city). In particular, *L-Systems* are used to describe the fractals. Fractals were used to describe the TSP instances as they allow for multiple instances, which share the

same *structure*, to be constructed. It is these common structures that are used to explain the difficulty of the instances.

In this paper, a couple of L-Systems were chosen and modified to make the TSP instance they generate more difficult to solve. Only minor changes were made to the L-Systems so that a completely new L-System is not created, thus avoiding the effect of randomly creating a new L-System. It is desirable that the overall structure, as described by the L-System, is maintained. To achieve this, certain cities are selected which are *perturbed* by a small distance from their original position.

An overview of previous work found in literature is presented in Section 2. Section 3 introduces the notation and defines the problem addressed in this paper. The method used to solve the problem is described in Section 4. Section 5 describes the results of running the optimization algorithm. A discussion of the simulation results and concluding remarks are presented in Section 6 and 7, respectively.

2 Related work

In 2003, Cotta and Moscato proposed an evolutionary computation-based attack on algorithms [3]. 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 [14] has also used evolutionary algorithms as an aid to finding weaknesses in combinatorial optimisation algorithms. Van Hemert tested his technique on the binary constraint satisfaction, boolean satisfiability, and traveling salesperson problems. When analysing his results, Van Hermert looked at the distribution of path lengths in order to explain the difficulty of the instances. The problem solvers used by Van Hermert were two variants of the Lin-Kernighan algorithm [7]. 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 [14]. Van Hermert 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.

Langdon and Poli used genetic programming to find fitness landscapes which highlight the strengths and weaknesses of differet Particle Swarm Optimisation (PSO) algorithms and to contrast population-based swarm approaches with non

stochastic gradient followers [8]. They were able to show that gradient ascent performs better then basic PSO on simple landscapes while slight increases in complexity reverses this observation.

More than a decade before, Moscato and Norman had shown that it is possible to generate arbitrary large fractal instances of the traveling salesperson problem with known optimal solutions. They have proposed that L-systems can be used to define them, and perhaps even more difficult instances could be generated from them [10].

3 Problem statement

In traditional computational complexity theory, the aim is to find an optimal solution given a particular instance. In this paper, it is desired to find multiple instances for which finding their solution is relatively difficult for a given solver.

Suppose P is a computational problem. Note that P is not necessarily an **NP** problem. Now let I_P be the set of all possible instances for the problem P and define $G_P = \{g \,|\, g : \mathbb{N} \to I_P\}$ to be a set of functions called *instance generators* (Definition 1).

Definition 1 (Instance Generator) *Given a computational problem P and its corresponding set of all possible instances I_P, a function $g : \mathbb{N} \to I_P$ is called an* Instance Generator*. That is, g can produce (or generate) instances $\{g(1), g(2), \dots\}$ for problem P.*

Definition 2 (Solver) *A* solver *is defined to be an algorithm or program which can solve instances of a given problem. Define Ψ_P to be the set of all solvers which solve instances of problem P.*

Define $t : \Psi_P \times I_P$ to be the amount of resources used (e.g. time taken) by a given solver to solve a particular instance. For any solver $A \in \Psi_P$ and instance $x \in I_P$ the value of $t(A, x)$ is the amount of resources used.

Definition 3 (Difficult-to-solve) *Suppose P is a combinatorial problem and $A \in \Psi_P$ is a corresponding solver. An instance $x \in I_P$ is "difficult-to-solve" if for any $y \in I_P$ (where x and y are of the* same size*) it holds that $t(A, x) \geqslant t(A, y)$.*

In this paper, the aim is not to find individual instances that are difficult-to-solve, rather to find a generator $g \in G$ which generates instances for which each are difficult to solve.

4 Method and algorithm

There are many ways to represent a set of (x, y) points (instances to the TSP), including just listing a set of all points. Finding a set of points is not enough for this paper. A method of describing the *pattern* of instances that are difficult for *Concorde* to solve is desired. Fractals provide a method to accomplish this.

Symbol	Meaning	Symbol	Meaning
F	Move Forward and place a city	[	Push. Store current angle and position on stack
G	Move Forward but *don't* place a city	]	Pop. Return to location of last push
+	Increase angle by (a pre-defined) δ degrees	$\backslash x$	Increase angle by x degrees
-	Decrease angle by δ degrees	$/x$	Decrease angle by x degrees
!	Switch meanings of +,-	$@x$	Multiply the current line segment size by $x \in \mathbb{R}$
A,B,...	The rewrite rules		

Table 1: The symbols used to describe an L-System (in the *Fractint* program).

4.1 Instance generator (L-Systems)

An *L-System* (or *Lindenmayer system*) is a grammar based method of describing fractals, first discovered by biologist Aristid Lindenmayer in 1968, as a technique of modelling various plants [13]. Any string generated by an L-System is graphically interpreted using a programming language similar to LOGO® [6,12].

L-Systems are described using *rules*. Each rule is a string of characters which specify how to construct the fractal. An additional rule is designated the *axiom*. It is the starting rule that defines the initial string and is used only once. During every iteration, some or all of the rules are substituted into the current string to create a new one. The string that results after n iterations will produce the L-System of order $n + 1$. L-Systems, and fractals in general, can be used to represent points in the x-y plane.

The L-Systems explained in this paper are described using the syntax defined for the *Fractint* program [4]. The symbols that can be used in each rule are described in Table 1. In order to find 'simple' L-Systems which can generate difficult instances, the number of rules in L-Systems was kept small. To achieve this, the number of rules in each L-System was kept constant.

4.2 Fitness function for an L-System

The fitness function involves comparing each L-System to a base case. The fitter L-Systems should consistently generate instances that are more difficult-to-solve than those generated by the base case.

From experience, creating difficult instances for Concorde of size less than 100 is hard, if not impossible. For this reason, calculation of the fitness depends only on the running time of instances containing greater than 100 cities.

When comparing the running times of instances generated by an L-System, to ones generated by another, the difference in running times is considered. Given an L-System L, suppose that each instance generated by L is run ξ times and the sets $\{\overline{t_i}\}$ and $\{\sigma_{t_i}\}$ contain the average running times and standard deviations respectively. Also, suppose the sets $\{\overline{t_i}'\}$ and $\{\sigma'_{t_i}\}$ contain the average running times and corresponding standard deviations respectively of instances generated by the modified L-System L'. The fitness value of L', compared to L is computed using the following weighted sum which gives greater weight to the running times

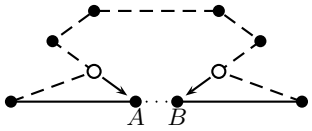


Fig. 1: An example of a situation that may occur, which could make solving the TSP instance more difficult

of the larger instances.

$$f_L(L') = \sum_i i[(\overline{t_i}' - 2\sigma'_{t_i}) - (\overline{t_i} - 2\sigma_{t_i})] \tag{1}$$

4.3 Modifying L-Systems

Given an L-System, which is an instance generator for the TSP, the question addressed in this paper is whether small changes can be made to the L-System such that the instances it generates become more harder to solve. Large changes is undesirable because they may be similar to creating random instances. Instead, by making small changes, the overall structure of the L-System will be preserved however the solver, *Concorde*, will have to make slightly different decisions when solving the instance.

Motivation It is hypothesized that a TSP instance can be made more difficult if some of the cities are 'moved' or *perturbed* to a different position—but not too far so as to preserve the instance's overall structure. It may be possible a situation similar to that shown in Figure 1 can occur

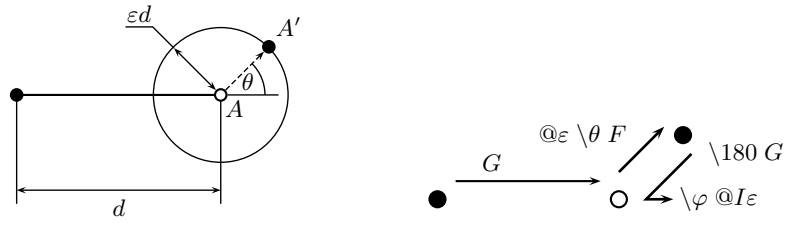
In Figure 1, cities A and B have been moved closer together. In this situation, some algorithms may be 'tempted' to include arc AB instead of following the dashed path. The aim is to slow down the TSP solver by forcing it to make more decisions.

Describing perturbations When a city is perturbed, it needs to be moved a small distance from its original position. The measurement 'small' is defined to be relative to the distance to the last city plotted. In addition, perturbations must not affect the positioning of other cities.

Figure 2a graphically describes the measurements involved when describing perturbations. In this diagram, the city has been moved from position A to A'. Its distance from A is not measured in absolute units but instead relative to its distance from the previously plotted city. An angle θ is required to complete the polar coordinate description of the city's new position.

In order to create a new L-System with the perturbation described in its grammar-based rules, the act of perturbing a city needs to be realized using only the allowable symbols.

Generating a TSP instance from an L-System uses the 'moving turtle' concept [5]. So a method to describe moving the turtle to the city's new position, placing the city, then moving back to the initial position, is required. The turtle needs



(a) The measurements involved when perturbing a city

(b) Describing a perturbation using L-System grammer

Fig. 2: Perturbing a city

to finish at the initial position to not affect the positioning of the other cities, and hence preserving the overall structure of the L-System.

Figure 2b shows how a perturbation can be described using L-System grammar. In order to place a city, the symbol 'F' is used. This makes the turtle move forward (a calculated distance) and place the city where it stops. The first change needed is to replace the 'F' with the letter 'G.' This will make the turtle move forward but not place a city. Next, the angle angle and distance is changed using the command '$@\varepsilon\ \backslash\theta$'. The 'F' is added after these commands to place a city at the new location. The turtle is turned around and moved back to the initial position using the command '$\backslash 180\ G$' and finally, it is faced in the direction it would have faced had there been no perturbation, that is, '$\backslash\varphi\ @I\varepsilon$' where $\varphi = 180 - \theta$. Hence given $0 < \varepsilon < 1,\ 0 \leqslant \theta < 360$, a perturbation is described by replacing the chosen F with:

$$G\ @\varepsilon\ \backslash\theta\ F\ \backslash 180\ G\ \backslash\varphi\ @I\varepsilon \tag{2}$$

where $\varphi = 180 - \theta$. Thus perturbations are uniquely defined by ε and θ.

4.4 Algorithm Overview

A simple local search is used to find a perturbation which makes the L-System produce instances that are more difficult to solve. For a given fractal, to be used as a base case, a number of new fractals are created and compared to it using Equation (1). The algorithm used follows the structure explained in Algorithm 1. A fractal, after one iteration, places a certain number of cities on the x-y plane. For every combination of μ cities placed (i.e. every occurance of the symbol 'F'), ξ random instances are created. Note that if the fractal plots n cities at every iteration, there are $\binom{n}{\mu}$ different combinations to consider. In each new L-System, every chosen 'F' is replaced with the sequence in Equation (2).

5 Simulation results

The local search optimization algorithm was run on a computer running a 32bit Linux OS with two 3GHz processors and 2GB of ram. The values chosen for the simulation parameters were $\mu = 3$ and $\xi = 10$ (Algorithm 1). The *Concorde* program, with the QSopt LP solver [1], was used as the TSP solver.

Algorithm 1: Basic structure of the algorithm used to modify a fractal to generate more difficult instances.

```
L ← Original L-System (base case)
for every μ-combination of 'F' in L do
    for i = 1 to ξ do
        Replace each chosen 'F' with the sequence in Equation (2)
        L_i ← Modified L-System
        Use Equation (1) to compute fitness of L_i relative to L
        if f(F_i) > f(F) then
            Save F_i to memory/file
        end
        if Not enough samples produced difficult-to-solve instances then
            Move on to next combination
        end
    end
end
```

```
Axiom: A
Angle: 8
A → F[+X]FB
B → F[-Y]FA
F → @1.36F@I1.36
X → A
Y → B
```

(a) *Leaf2*

```
Axiom: A
Angle: 8
A → G@0.25\016F\180G\164@I0.25
  [+X] G@0.15\299F\180G\241@I0.15 B
B → G@0.23\102F\180G\078@I0.23 [-Y]FA
F → @1.36F@I1.36
X → A
Y → B
```

(b) *Leaf2_1*

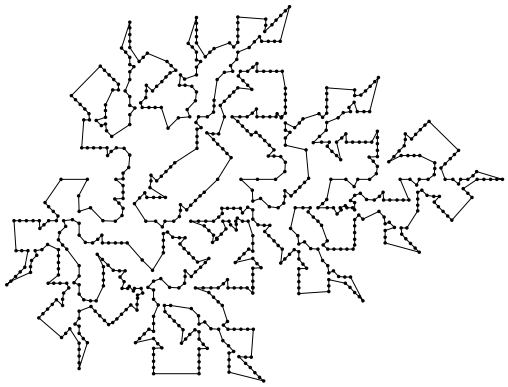
Fig. 3: Modifying the L-System *Leaf2*.

Figure 3a shows the rules of L-System *Leaf2*. After applying the local search Algorithm 1, the L-System shown in Figure 3b was found. Figure 4 shows a graphical representation of TSP plots, and the corresponding solution found by *Concorde*, of instances generated by *Leaf2* and *Leaf2_1*. The fractals are of order 12 and contain 754 cities.

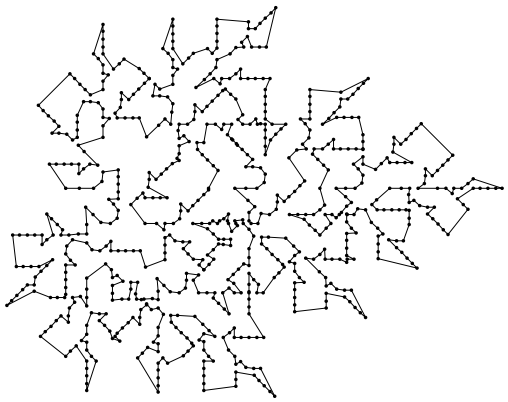
The local search was also performed on the L-System known as *MNPeano* [9]. The optimal tour of TSP instances generated by this fractal can be found using nearest neighbour heuristics [11]. Figure 5a shows the rules of *MNPeano*. After applying the local search, the L-System *MNPeano_1* (Figure 5b) was found.

Two forms of metric are used to measure the amount of resources *Concorde* used to solve the given TSP instances. The average running times is one measurement, and was used to assess the fitness of each L-System. Another metric is the number of *branch-and-bound* search nodes used by the *Concorde* program, which uses the branch-and-cut approach to solve a linear programing relaxation of a TSP instance.

Table 2 shows the running times taken and the number of branch-and bound nodes created by *Concorde* to solve instances generated by the two L-Systems *Leaf2* and *Leaf2_1* and Table 3 is the corresponding table showing the difference in running times and number of branch-and-bound nodes created for instances generated by the L-Systems *MNPeano* and *MNPeano_1*.



(a) *Leaf2*. Average running time: 7.901s. Average number of branch and bound nodes created: 1



(b) *Leaf2_1*. Average running time: 103.243s. Average number of branch and bound nodes created: 26.467

Fig. 4: Graphical representations of two L-Systems analysed (*Leaf2* and the evolutionarily generated *Leaf2_1*). Both fractals of order 12 (754 cities)

6 Discussion and analysis

Table 2, which contains the running times taken for *Concorde* to solve instances generated by the two L-Systems *Leaf2* and *Leaf2_1*, shows that the modified L-System does generate instances that are more difficult-to-solve. While Table 3 shows that every modified instance of *MNpeano* took longer to solve. In addition to running times, the number of branch-and-bound nodes needed by *Concorde* to solve the modified TSP instances either equalled or increased by a large factor.

The largest increase in time, for the *Leaf2_1* instances occured for order 13 (1218 nodes) where the time required to solve the modified instance increased by a factor of 412. For the *MNPeano_1* instances, for order 8 (1452 nodes) the amount of time required to solve the modified instance was almost 30 000 times greater than what was needed to solve the original instance.

```
Axiom: XFF--AFF--XFF--AFF
Angle: 8
A → B@Q2F@IQ2
B → AFF
F →
X → +!X!@2F@.5-B@Q2F@IQ2-!X!FF+
Y → FFY
```

(a) *MNPeano*

```
Axiom: XFF--AFF--XFF--AFF
Angle: 8
A → B@Q2F@IQ2
B → AFF
X → +!X!@2
      G@0.04\151F\180G\029@I0.04
      @.5-B@Q2F@IQ2-!X!
      G@0.86\324F\180G\216@I0.86
      G@0.75\051F\180G\129@I0.75 +
Y → FFY
F →
```

(b) *MNPeano_1*

Fig. 5: Modifying the L-System *MNPeano*.

Order	No. Cities	Time taken (s)		No. BB Nodes	
		Leaf2	*Leaf2_1*	*Leaf2*	*Leaf2_1*
8	108	0.366	0.158	1.000	1.000
9	176	0.525	6.379	1.133	5.533
10	286	1.152	18.594	1.000	21.800
11	464	2.564	65.812	1.000	22.467
12	754	7.901	103.243	1.000	26.467
13	1218	12.608	5196.229	1.000	497.333
14	1972	41.028	8442.554	1.667	303.245

Table 2: A comparison of the average running times taken (seconds) and the average number of branch-and-bound (BB) nodes created by *Concorde* to solve the TSP instances generated by *Leaf2* and *Leaf2_1*.

7 Conclusions

Fractals (defined by L-Systems in this particular case) were used to generate TSP instances which share the same structure, in an attempt to find similar difficult-to-solve TSP instances. Small changes can be made to the fractals such that the amount of resources required to solve the new modifie instances (e.g. time) increases by large amounts. A local search optimization algorithm is used to find how such fractals can be modified. The results found showed large increases in time and decisions needed (branch-and-bound nodes) occured when solving the modified instances. Thus classes of difficult-to-solve instances of the TSP problem were found using evolutionary algorithms that employ local search techniques.

References

1. David Applegate, William Cook, Sanjeeb Dash, and Monika Mevenkamp. The *qsopt* linear programming solver web site. Last accessed: November 15, 2010. URL=http://www2.isye.gatech.edu/~wcook/qsopt/.

Order	No. Cities	Time taken s		No. BB Nodes	
		MNPeano	*MNPeano_1*	*MNPeano*	*MNPeano_1*
5	180	0.045	0.696	1.000	1.000
6	364	0.094	21.311	1.000	4.000
7	724	0.242	35.380	1.000	1.000
8	1452	0.518	15278.544	1.000	177.000
9	2900	1.264	10916.553	1.000	18.000

Table 3: A comparison of the average running times taken (seconds) and the average number of branch-and-bound (BB) nodes created by *Concorde* to solve the TSP instances generated by *MNPeano* and *MNPeano_1*.

2. William Cook. The *Concorde* web site. Last accessed: March 1, 2007. `URL=http://www.tsp.gatech.edu/concorde/index.html`.
3. Carlos Cotta and Pablo Moscato. A mixed evolutionary-statistical analysis of an algorithm's complexity. *Applied Mathematics Letters*, 16:41–47, 2003.
4. Noel Giffin. The *fractint* web site. Last accessed: November 9, 2010. `URL=http://spanky.triumf.ca/www/fractint/fractint.html`.
5. James Scott 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.
6. David J. Holliday, Brian Peterson, and Ashok Samal. Recognizing plants using stochastic L-Systems. *Proceedings. IEEE International Conference Image Processing, 1994*, 1:183–187, 1994.
7. B. Kernighan and S. Lin. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.
8. William B. Langdon, Riccardo Poli, Owen Holland, and Thiemo Krink. Understanding particle swarm optimisation by evolving problem landscapes. In *Swarm Intelligence Symposium, 2005. SIS 2005. Proceedings 2005 IEEE*, pages 30–37, Jun 2005.
9. Adrian Mariano, Pablo Moscato, and Michael G. Norman. Using L-Systems to generate arbitrarily large instances of the euclidean traveling salesman problem with known optimal tours. In *In Anales del XXVII Simposio Brasileiro de Pesquisa Operacional*, pages 6–8, 1995.
10. Pablo Moscato and Michael G. Norman. On the performance of heuristics on finite and infinite fractal instances of the euclidean traveling salesman problem. *INFORMS J. on Computing*, 10:121–132, February 1998.
11. Michael G. Norman and Pablo Moscato. The euclidean traveling salesman problem and a space-filling curve. *Chaos, Solitons & Fractals*, 6:389–397, 1995. Complex Systems in Computational Physics.
12. P Prusinkiewicz. Graphical applications of l-systems. In *Proceedings on Graphics Interface '86/Vision Interface '86*, pages 247–253, Toronto, Ont., Canada, Canada, 1986. Canadian Information Processing Society.
13. Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.
14. Jano I. van Hemert. Evolving combinatorial problem instances that are difficult to solve. *Evolutionary Computation*, 14(4):433–462, 2006.