Genetic Programming for the Evolution of Functions with a Discrete Unbounded Domain

by

Shawn Eastwood

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The idea of automatic programming using the genetic programming paradigm is a concept that has been explored in the work of Koza and several works since. Most problems attempted using genetic programming are finite in size, meaning that the problem involved evolving a function that operates over a finite domain, or evolving a routine that will only run for a finite amount of time. For problems with a finite domain, the internal representation of each individual is typically a finite automaton that is unable to store an unbounded amount of data. This thesis will address the problem of applying genetic programming to problems that have a "discrete unbounded domain", meaning the problem involves evolving a function that operates over an unbounded domain with discrete quantities. For problems with an discrete unbounded domain, the range of possible behaviors achievable by the evolved functions increases with more versatile internal memory schemes for each of the individuals. The specific problem that I will address in this thesis is the problem of evolving a real-time deciding program for a fixed language of strings. I will discuss two paradigms that I will use to attempt this problem. Each of the paradigms will allow each individual to store an unbounded amount of data, using an internal memory scheme with at least the capabilities of a Turing tape. As each character of an input string is being processed in real time, the individual will be able to imitate a single step of a Turing machine. While the real-time restriction will certainly limit the languages for which a decider may be evolved, the fact that the evolved deciding programs run in real-time yields possible applications for these paradigms in the discovery of new algorithms. The first paradigm that I will explore will take a naive approach that will ultimately prove to be unsuccessful. The second paradigm that I will explore will take a more careful approach that will have a much greater success, and will provide insight into the design of genetic programming paradigms for problems over a discrete unbounded domain.

Acknowledgements

I would like to thank my supervisor Dr. Jonathan Buss, my readers Dr. Jeff Orchard and Dr. Jesse Hoey, and my family for their invaluable assistance.

Dedication

This thesis is dedicated to my dad.

Table of Contents

List of Figures

Chapter 1

Introduction

Genetic programming is the creation of programs through the process of natural selection. Viewing the evolved program as a function that accepts input and produces output, the problem has a "finite domain" if the possible input set for the evolved program is finite. Genetic programming has been successfully applied to many problems that have a finite domain [11, 12]. The problem has a "discrete unbounded domain" if the possible input for the evolved programs set is infinite and discrete. I will not consider problems like "symbolic regression", which is the problem of determining an algebraic expression to fit data points over a continuous bounded interval [11, pg. 11]. In symbolic regression the set of inputs to the expression is infinite, however the inputs form a bounded continuum which is outside the scope of this thesis. A problem which does have a discrete unbounded domain is the problem of "sequence induction", which is the problem of determining an algebraic expression to predict an infinite sequence of integers [11, pg. 10]. In sequence induction the set of inputs to the expression is the set of natural numbers which is both discrete and unbounded.

In this thesis, the problem that I will address in particular is the problem of evolving a real-time decider for an arbitrary language. The problem of evolving a real-time deciding program has a discrete unbounded domain because the domain of possible words is both discrete and infinite. This problem will be analyzed as a member of the class of problems with a discrete unbounded domain, and the observations made have the potential to generalize to other problems. In this thesis, while I will examine regular languages, the "unbounded" aspect is only truly meaningful when I examine non-regular languages. Deciding a non-regular language in real time requires explicit internal memory, which is a trait shared by other problems over a discrete unbounded domain. The amount or structure of the internal memory affects what languages are decidable. In this thesis, each agent will have internal memory that is at least equivalent to a single Turing machine tape. While there has been much work done on the problem of evolving a Turing machine to act as a decider for a given language [28, 18], the paradigms described in this thesis will evolve deciding programs that run in real-time.

The deciding programs evolved by the paradigms presented in this thesis may have practical applications due to their real-time aspect. The process for generating the test cases will imply that an algorithm that can decide the sought after language already exists. As the evolved deciding programs run in real-time, the evolved decider will possibly have an advantage over the existing decider.

The first paradigm that will be described in this paper I will refer to as RTGP_1. The programs in RTGP_1 will be naively implemented using expression trees and heavy use of "if" statements. The programs will have access to internal memory in the form of an infinite linked list. RTGP_1 will be shown to be ineffective at evolving deciding programs for all but the simplest regular languages. Analyzing possible reasons to the failure of RTGP_1 will motivate the design of the second paradigm that I will refer to as RTGP_2. The programs in RTGP_2 will be finite automatons each with access to a single Turing tape. RTGP_2 will be shown to very effective at evolving deciding automatons for regular languages and succeeds at evolving deciding programs for a number of non-regular and non-context free languages as well. The contrast between RTGP_1 and RTGP_2 will yield an important observation with regards to the design of genetic programming paradigms.

Chapter 2 will detail the genetic programming process that I will use for RTGP_1 and RTGP_2. I will also

describe some of the alternative choices for the sake of completeness and describe why I believe the choices that I have made for my paradigms are justified. Details that are specific to $RTGP_1$ or $RTGP_2$ are left for chapters 5 and 7.

Chapter 3 will further contrast problems with a finite domain with those with a discrete unbounded domain using the artificial ant problem and a variation of the artificial ant problem. In this section I will also describe internal memory paradigms used in previous works with increasing complexity up to the internal memory capacity that the deciders evolved in this thesis will have.

Chapter 4 will list a number of languages taken from previous publications that I will use as tests for $RTGP_1$ and $RTGP_2$.

Chapter 5 will describe the $RTGP_1$ paradigm. In chapter 6, $RTGP_1$ will be tested on some of the benchmark languages and a discussion as to the causes of $RTGP_1$'s failure will be provided.

Chapter 7 will describe the $RTGP_2$ paradigm, aiming to improve upon the shortcomings of $RTGP_1$. In chapter 8, $RTGP_2$ will be tested on all of the benchmark languages, including a language for which the existence of a real-time decider is not known.

Chapter 9 will conclude the thesis with a discussion on the importance of proper design of genetic programming paradigms using the failure of $RTGP_1$ as an example. There will also be a discussion as to the possible limitations of evolving real time deciders, and a description of possible extensions for $RTGP_2$ to enable $RTGP_2$ to better find real time deciders for some of the more complex languages.

Chapter 2

Background

Genetic programming is a subclass of a more general set of algorithms known a "genetic algorithms". Genetic algorithms solve optimization problems by denoting a set of possible solutions as a "population", where each possible solution is an individual in the population. In this thesis, the individuals/possible solutions will be referred to as "agents". A genetic algorithm starts with a randomly generated population of agents. At each generation, a "fitness value" is ascribed to each agent. A fitness value is a measure of how preferable an agent is, or how close an agent is to solving the "target problem". The next generation is formed by randomly selecting agents from the current population according to their fitness. Agents with better fitness should be preferred to agents with worse fitness. The selected agents may then be subjected to genetic operators such as mutation or crossover, or may be directly placed in the next generation with no alterations. Mutation alters the "genome" of an agent by a small amount, while crossover takes two agents and produces a child whose genome is a mix of the genomes from both parents. The genetic algorithm continues to advance the population from generation to generation until a "termination condition" is met. The termination condition may be met by the creation of an optimal agent, or when a preset number of generations have elapsed. If the termination condition is satisfied by the number of generations running over the generation limit, the genetic algorithm is typically restarted. If the termination condition is met by the creation of an optimal agent, this agent is returned as the output for the genetic algorithm.

Genetic algorithms most commonly denote agents using their "genotype" (also referred to as the "genome"), which is a data structure that stores all information required to specify the agent. This data structure used for the genome is often engineered to be easily used by the genetic operators. The genome for each agent is decoded into a "phenotype", which is a more natural representation of the solution/function denoted by the agent. The phenotype is what is used to measure the fitness of an agent.

Genetic programming is the use of genetic algorithms to produce expressions or programs that solve a specific problem. The agents in the case of genetic programming denote expressions or entire programs. As the search space of programs is enormously complex, genetic programming is likely a more effective search method than other search methods. In the context of genetic programming, the "genotype" refers to an agent's representation within the genetic algorithm, while the "phenotype" refers to the program/function described by the genotype without regards to its inner workings.

When setting up a genetic program to solve a specific problem, important aspects to be considered are the genotype representation of an agent; the phenotype representation of an agent; the genetic operators (initialization/mutation/crossover); the mechanism for determining the fitness of an agent; and the mechanism for selecting agents for the next generation. Some of these aspects will be given consideration in the sections below:

In the following sections, I will detail some of the aspects of genetic programming listed above. Section 2.1 will detail multiple ways an agent's program may be stored as a genome, and indicate with motivation the representation paradigm for the genome that I will be using. Section 2.2 will examine previous works that discuss the effectiveness of mutation vs. crossover. Section 2.2 will also provide reasons as to why I will only use the mutation genetic operator in my paradigms, as opposed to the crossover operator. Section 2.3

will detail the mutation operator with a brief foreword on the initialization process. Section 2.4 will detail how the fitness of each agent is calculated, and how the next generation is created. The aspects described here apply to both paradigms I will be introducing in this thesis.

2.1 The structure of agents

The program used by each agent can be expressed using several styles of representation that determine how the genetic operators of mutation and crossover act on the program. In the next sections, I will list and describe some of the representation schemes that are used to denote the agents. I will start with the basic "expression tree" style of representation used in the work of Koza [11, 12]. I will then describe "Grammatical evolution" which uses a more traditional genome where the program is denoted by a string of integers that act as alleles. As a precursor to the representation style that I will use, next I will describe the "automatically defined functions" extension to the expression tree paradigm described in [12]. Lastly I will describe "Cartesian genetic programming" which is the paradigm that I will be making use of throughout this paper.

2.1.1 Expression trees

In the work of Koza, programs/agents are denoted using expression trees [11, 12]. Each node in the expression tree stores a symbol, and the children of that node are the inputs to the given symbol. Non-terminal (non-leaf) symbols are often connectives that link multiple commands, while terminal (leaf) symbols are single commands.

While there are many ways mutation and crossover can be implemented for expression trees, the straight forward implementations of mutation and crossover are as follows. Mutation performed on agent A randomly selects a subtree of A and replaces it with another randomly generated subtree [11]. The operation of cross-over performed on agents A_1 and A_2 randomly selects subtrees from A_1 and A_2 and swaps them between A_1 and A_2 [11].

2.1.2 Grammatical Evolution

Another style of representation used in what is called "Grammatical Evolution" uses a grammar to denote the program's syntax [26, 20, 19]. In Grammatical Evolution, the genome is a string of integers. The program is generated by starting with the grammar's initial symbol. Each time a transition rule needs to be chosen, an integer is read from the genome and reduced modulo n where n is the number of possible valid transition rules. The transition rule is chosen based on this remainder.

The operation of mutation performed on agent A mutates integers in the genome at random. [26] also describes two other mutation operators: duplication and pruning. Duplication replicates part of the genome making it longer. Pruning removes the parts of the genome that contribute nothing to the phenotype (not all integers may be used in generating the program). The operation of cross-over performed on agents A_1 and A_2 simply breaks both genomes at random locations and swaps the two latter pieces between the two agents.

2.1.3 Expression trees with automatically defined functions

In [12], Koza expands the expression tree concept to include "automatically defined functions". With automatically defined functions, the expression tree contains several subtrees that define functions which can then be reused as much as needed in the main expression tree. These defined subroutines are referred to as "automatically defined functions". Using this paradigm allows complex subexpressions that may appear multiple times to only need to be evolved once. A limitation with automatically defined functions is that for a specific agent, the number of automatically defined functions along with their parameter lists must be fixed

upon creation and is generally not subject to mutation. In [13] however, Koza provides some mutation operators that can change the number of automatically defined functions or their parameter lists. Crossover is difficult to implement with automatically defined functions, as the replacement subtree has to be compatible with the position it is being introduced to.

2.1.4 Cartesian genetic programming

To denote the genome of an agent, I will make use of a relatively recent genetic programming paradigm known as "Cartesian genetic programming" (CGP). Cartesian genetic programming is explored in [22, 16, 17, 15], and [7]. It has been shown that in certain cases, CGP is faster than expression trees with automatically defined functions at generating an optimal solution [22]. In CGP, I use the same expression trees as in conventional genetic programming, only that the nodes are members of a grid with $R > 0$ rows and $C > 0$ columns. Each node is assigned a symbol, and can only have children from the columns to the right of the current node. Nodes in the rightmost column can only be assigned terminal symbols (have no children). To the left of all the columns are extra nodes that denotes the root(s) of the expression tree(s). The expression trees are effectively acyclic directed graphs embedded in the grid of nodes. CGP allows for expression trees to be reused multiple times without being evolved from scratch, and it also enables the storage of useful, but inactive expression trees, as not all nodes in the grid will be part of the expression trees. Neutrality refers to the presence of several genotypes that have identical fitness values, which is very common in CGP due to the presence of inactive nodes. Neutral mutations are mutations that change the genotype with no corresponding change to the fitness [16]. The advantages of neutrality and neutral mutations are discussed in [16, 15]. In this paper, the root nodes will not contain a symbol, but simply a pointer to the actual root of the expression tree. It is noted in [7] that arranging the nodes in a single row makes for better performance, so I will let $R = 1$. CGP also artificially puts a cap on expression tree size as the number of nodes in the array is fixed.

CGP is a generalized version of the automatically defined functions paradigm, as any subexpression can be reused as much as needed without needing to be independently evolved in each context [16]. However automatically defined functions do not need to use the same set of inputs each time they appear, unlike CGP [16]. Automatically defined functions incorporate parameter symbols in their definition. The subexpressions that can be reused in CGP cannot be invoked for different inputs. While this may put CGP at a disadvantage to automatically defined functions, the reusability feature is intrinsic to CGP, and is not intrinsic to ordinary expression trees. There is also no issue in CGP with adjusting the number of automatically defined functions or their parameter lists.

Mutation in Cartesian genetic programming has an agent select a fixed number of nodes at random and changes their symbol/children. In another style of mutation an agent examines each of its nodes, selecting them for mutation with a fixed probability. A naive crossover can also be implemented that breaks the parent's array of nodes at a random point and swaps the two sections [7], and another naive crossover can swap corresponding nodes between the arrays of each parent [16]. When a node is moved from one parent to another, pointers to other nodes can be treated as an index of the child node or as a displacement from the parent node. As I will see later however, crossover in Cartesian genetic programming is not as useful as previously thought.

I will now consider the idea of restricting mutation to "active nodes" in the array, which are nodes that are part of an expression tree. This however results in agents having a limited selection of new expression trees to choose from, and restricts the directions that the agent can explore via mutation. By allowing mutations to inactive nodes, the set of inactive expression trees that an agent (or its descendants) has to choose from becomes more diverse, making the probability of finding a useful mutation more likely. [16, 15] also discuss the advantages of neutral mutations (i.e. "neutral drift"[15]) which are often mutations to inactive nodes so the phenotype will not be affected. [16, 15] have shown that convergence is greatly improved when neutral mutations are allowed. As a result, I will allow mutations to be able to affect all nodes in the array, and not just those that are "active".

2.2 Mutation vs Crossover

I have selected the Cartesian genetic programming paradigm to denote the genome of each agent. Now this section will attempt to justify my decision to use only the mutation operator. This section will summarize previous research and observations about the mutation and crossover operators in order to justify my decision to not make use of the crossover operator.

In [11] and [12] where the expression trees nodes are not embedded in an array, it is argued that mutation for genetic programming is unnecessary for the following reasons. If the problem has a small set of symbols, any given symbol is unlikely to be lost from the population and does not need to be reintroduced via mutation. As long as every symbol is retained by the population as a whole, any expression tree can be generated via a sequence of crossover operators. Another reason states that since subtrees are the fundamental building blocks of the internal structure of each agent, keeping "useful" subtrees intact via crossing over will allow for greater improvements to the fitness than simply altering subtrees via mutation. This is referred to as the "Building Block Hypothesis" [3]. Despite these reasons however results from [14, 3] seems to suggest that using crossover to evolve a population of agents has no real advantage over using mutation. [3] describes the crossover operator as being mutation where the new subtree is taken from elsewhere in the population. In Cartesian genetic programming however, the primary genetic operator is mutation. A naive crossover operator that involves breaking the array of nodes at a single point and swapping the latter pieces can be applied to Cartesian genetic programming and is in fact known to slow the progress of the algorithm in finding a solution [17, 7]. In [7], a new crossover operator is proposed for Cartesian genetic programming which augments the speed of the algorithm in finding a solution. This new crossover operator requires the integer fields in the genome to be replaced with real numbers. Replacing the integer field with real numbers as well as the crossover operator itself may complicate certain aspects of my paradigms, and to avoid over complicating my paradigms, I will do without crossover altogether. However the implementation of the crossover operator described in [7] for my paradigms can be the object of future research.

2.3 Mutation

I have selected the Cartesian genetic programming paradigm to denote the genome of each agent. Now this section will describe in detail the mutation operator.

Initialization of each agent is done randomly. Self adaptive parameters, which I will describe later, do not affect the initialization process and are all initialized to 0.

2.3.1 The basic mutation step

The "basic mutation" operation on an agent A acts as follows: A list of possible "mutation sites" is to be compiled. Each "mutation site" is a value within A that will be subjected to mutation if chosen. Such mutation sites are listed below:

- Each of the root node pointers.
- Each node n from the array has the following mutation sites:
  - Each symbol field of n.
  - Each child pointer of n.

Once the list of mutation sites is compiled, one of these sites is chosen at random. The parameter at this site is assigned a new randomly chosen value.

The "basic mutation" operator can only influence a single parameter within an agent. One possible problem this can have is that helpful changes generally require more than one "step" to pass through a zone of poor fitness. That is why during mutation, the number of "basic mutation" steps will be randomly chosen

from the range $\{1, 2, \ldots, \text{max\_mutations}\}$. Once the number of "basic mutation" steps has been chosen, these steps are carried out one after another in sequence.

Another possible style for mutation is to sweep through each of the mutation sites, and choosing to mutate each of them with a probability of $p_{\text{mutation\_rate}}$. $p_{\text{mutation\_rate}}$ can be chosen to be very small to limit the expected number of mutations that occur. In practice however, I have found that this impedes the progress of the genetic algorithm and so this style of mutation is not used.

2.3.2 Self adaptive parameters

In this paper, the mutation operator will be influenced by component level "self adaptive parameters" [1]. Component level self adaptive parameters are parameters that are assigned to each node in each agent that influence the genetic operators, in this case mutation [1]. The parameters will have "empirical update rules", meaning that the parameters themselves are subject to mutation, and will evolve alongside the agents themselves [1].

In [2], two different component level self adaptive parameter schemes are applied to the crossover genetic operator, and both are shown to improve the standard crossover operator.

Using these parameters, each agent has a degree of control over how it is affected by the mutation operator. To further understand why I will use self adaptive parameters, consider the following example. An agent A has a better fitness than agent A'. Imagine that A's structure is "fragile", meaning that a random mutation to the structure of A is likely to produce an individual of considerably worse fitness than that of A itself. Imagine that A''s structure is "robust", meaning that a random mutation to the structure of A' is not likely to produce an individual of considerably lower fitness than A'. At the current generation, A will have better reproductive success than A', but in the next generation, the children of A' will overall have better success than the children of A effectively implying that A' has better reproductive success than A. Since agents with a more "robust" structure are implicitly preferred to agents with a more "fragile" structure, this will lead to the proliferation of agents that are better able to protect their children from harmful mutations, rather than a proliferation of agents with high fitness.

It could be argued that agents such as A' that produce high fitness children are to be desired over fragile agents like A. However it should also be noted that simply because a larger fraction of mutations harm A doesn't mean that A' should have greater reproductive success than A. If there exists even a small fraction of mutations that enhance A it is worth giving A a greater reproductive success by suppressing the harmful mutations. Self adaptive parameters can help restrict mutations on A to helpful mutations.

As an agent approaches optimality, it is likely that a larger fraction of mutations will harm the agent's fitness due to a larger fraction of the agent's structure being "correct". With self adaptive parameters, agents may protect vital or "completed" parts of their structure so mutations instead focus on sections that are not "correct" or "complete". Without self adaptive parameters, agents may possibly protect themselves from harmful mutations in unintended ways. For example, in the case where expression trees denote the genotype for each agent, an agent may generate "dead code", meaning that there will be branches of the expression trees that will never see use and exist solely to soak harmful mutations. It is possible that if the fraction of harmful mutations for a near optimal agent is too great, the population may be unable advance due to near optimal agents being ruined by mutation faster than they are produced.

The self-adaptive parameter scheme that I will use in this thesis is similar to the scheme used in [10]. Every mutation site s in an agent is assigned an integer value that I will call $\theta(s)$. $\theta(s)$ controls the weight of s being chosen for mutation. During initialization, $\theta = 0$ for all sites. The weight given to a site by θ is:

$$\text{weight}(\theta) = \begin{cases} \frac{1}{1-\theta} & (\theta \leq 0) \\ 2 - \frac{1}{1+\theta} & (\theta \geq 0) \end{cases}$$

When a mutation site is to be chosen, the probability that a specific mutation site s is chosen is:

$$\Pr(s \text{ is chosen}) = \frac{\text{weight}(\theta(s))}{\sum_{s' \text{ is a mutation site}} \text{weight}(\theta(s'))}$$

The θ's are themselves subject to mutation after all "basic mutation" steps have been completed, or the parameters will not be able to evolve alongside the agent. After mutation, the probability that a given θ will mutate is μ_θ. If a self adaptive parameter mutates it either increases or decreases by 1.

The primary purpose of including the θ parameters is to let the agents protect themselves from harmful mutations as larger sections of their structure become "correct" and need to be protected. The larger an important subexpression becomes, the larger the probability that part of this subexpression will be affected by mutation. An agent can "protect" this subexpression by decreasing the subexpression's θ parameters until the probability of mutation affecting this subexpression is bounded from above by a fixed quantity.

The optimal value of a self adaptive parameter is often tied to the quantity (symbol/pointer) currently at the mutation site. If a mutation site is being shielded by its θ, $\theta << 0$, the quantity at that site is what is being protected. If the mutation site is chosen, the quantity will most likely be changed and the protection will no longer be relevant. If a mutation cite is chosen for mutation, its θ will be set to 0 no matter its former sign. This is done because when the quantity (symbol/pointer) at a mutation site changes, the former θ value is no longer relevant.

One possible problem with these self adaptive parameters is the potential for quantities (symbols/pointers) to become "stuck" at a suboptimal state due to the influence of self adaptive parameters. The intention of the parameters as indicated above were to allow the agent to decrease the probability of mutation affecting an arbitrarily large subexpression to below a fixed bound. I do not however want this protection to be applied to incorrect parts of the genome either. I will introduce two new factors that will allow an agent to mutate without the parameters influence with a fixed probability.

With probability α, after mutation the agent will reset all of its self adaptive parameters to 0 instead of mutating them after all "basic mutation" steps have been completed. With probability β, an agent will ignore its self adaptive parameters during all of the "basic mutation" steps. These events are independent, and the probability of either of these events occurring is:

$$1 - (1 - \alpha)(1 - \beta) = \alpha + \beta - \alpha\beta$$

For any $\epsilon > 0$, the agent can reduce the probability of harmful mutations to $\alpha + \beta - \alpha\beta + \epsilon$. What is important is that this probability is independent of the size of the to be protected subexpression, so any large subexpression can be sufficiently protected.

Although concrete results are not presented here, when working with the RTGP_2 paradigm I have observed that self adaptive parameters are very beneficial when the minimum size of a deciding program for a language becomes large.

2.4 Creating the next generation

To create the next generation, a genetic program must first compute the fitness of each agent. After the fitness of each agent has been computed, agents must be selected according to a selection scheme which chooses agents based on their fitness. In this thesis, selection is always done with replacement so an agent may be selected twice. Once an agent has been selected, it will be copied, subjected to mutation and then added to the next generation. I will also make use of what is known as the elitist strategy in this thesis.

2.4.1 Measuring fitness

In this section, I will specify how the fitness of each agent is measured. The specific set of problems being addressed by this thesis is the creation of a real time decider for an arbitrary language of words. The deciders will process an input word character by character. From here on, I will let $\Sigma = \{0, 1, \ldots, |\Sigma| - 1\}$ be the alphabet of letters that each word can use.

For problems with a finite domain, measuring fitness is simply a matter of counting up the number of correct responses for every possible input from the domain. For problems with a very large or infinite domain, it is intractable or impossible to test the agent for every possible input from the domain. The compromise that I can make for large or unbounded domains is that for a specific generation, I will generate $T >> 0$ test

cases using a randomized process for each generation. The T test cases will be constant throughout each generation, and will be randomly generated fresh for each generation, similar to what is done in [33]. This "moving target" [33] approach stands in contrast to the approach taken in most other works where the test cases are fixed.

If test cases from a large domain are fixed from generation to generation, then there is the possibility that the agents will evolve to satisfy *only* those test cases. By randomizing the test cases from generation to generation, a larger portion of the domain can be covered and there is a greater chance that the agents will evolve to form general solutions.

The approach to measuring fitness will differ based on the language that I am attempting to find a decider for. All agents will receive the same tests within the same generation. Let U denote a set of T test cases. For most problems where $|\Sigma| = 1$, my set of test cases is $U = \{0^i : 0 \leq i < T\}$. In this case, the fitness is being measured deterministically. For most problems where $|\Sigma| = 2$, I will randomly generate a set U of T test scenarios at each generation. To generate these scenarios I will follow the following algorithm:

repeat
  Chose a random length L from the range $0, 1, \ldots, L_{\max}$ with a uniform distribution.
  Let w be a new world with length L.
  Choose "temp" at uniform from $\{0, 1\}$.
  if temp $= 0$ **then**
    Create w randomly so it is likely that w belongs to the language.
  else
    Create w randomly so it is likely that w does not belong to the language.
  end if
  for $i = 0$ to L **do**
    Add a word consisting of the first i characters of w to U.
    Break the loop if U contains T test scenarios.
  end for
until U contains T test scenarios

This particular process ensures that I not only test word w, but also check each output of the deciding program as w is being processed. In essence the sequence of outputs leading up to word w are tested, similar to the tests when $|\Sigma| = 1$. For most of the languages I will address, fitness is the number of failed test cases. For some languages however I will perform additional tests that will add to the fitness computed from the test cases. It should be noted that in all problems that the larger fitness values are considered worse than lower fitness values. "Better" fitness refers to having a lower fitness value, while "worse" fitness refers to having a higher fitness value. A fitness value of 0 is considered optimal. This is known as "standardized fitness" [11, pg. 96].

There is no fixed routine for generating the fitness cases. The fitness cases should be generated so that all possible behavior of the language is covered. This is a subjective process that depends on the user's "intuition" regarding the language. As I present my results, I will lay out in detail how the fitness cases are generated.

It may seem that nothing is being gained by generating the tests repeatedly so that a roughly equal number fall into or out of the sought after language. The use of an algorithm for generating tests that are split roughly evenly between the language and its complement implies that the user already has sufficient knowledge about the language to construct a decider without the help of genetic programming. What should be accounted for is that the algorithms that my genetic program produces run in real-time, while the decider that the user is aware of may run with a greater time complexity. By using these genetic programs, it is assumed that a real-time algorithm for a known problem is being sought by the user.

2.4.2 Selecting agents for the next generation

There are many ways an agent can be selected for the next generation. One of these selection methods is "fitness proportionate selection". For fitness proportionate selection, the fitness needs to be positive real number where higher numbers indicate better fitness. Currently for our paradigms, higher fitness values indicate worse fitness. Using the transformation [11, pg. 97]:

$$f'(A) = \frac{1}{1 + f(A)}$$

where $f(A)$ is the original fitness of agent A, the new fitness $f'(A)$ has larger values when A has better fitness. $f'(A)$ is then known as the "adjusted fitness"[11, pg. 97]. In fitness proportionate selection, the probability of randomly selecting A from population P is:

$$\Pr(A) = \frac{f'(A)}{\sum_{A' \in P} f'(A')}$$

The method of selection that I will use in this thesis is "tournament selection". With tournament selection, the numerical value of the agent's fitness does not matter, only its rank relative to other agents. An agent is chosen by first choosing a subset of $k = 2$ agents from the population. The agent that has the better fitness is selected for the next generation with probability $p_{\text{tournament}} > 0.5$, otherwise the agent with worse fitness is selected. The benefit of tournament selection is that the absolute measure of fitness does not have an effect, the only aspect of the fitness that is important is how it compares to the fitness of other agents.

2.4.3 Bloat control

Bloat refers to the unbounded growth in program size as the generations progress. Methods of controlling bloat have included "parsimony pressure" where an individual's fitness is penalized due to its size [23]. Another method is known as "Tarpeian bloat control". An individual whose size is greater than the mean program size may be assigned a very unfit fitness value according to a fixed probability [23]. [24] shows how the weight of the size penalty in the "parsimony pressure" bloat control method can be adjusted to control the average program size. [24] also shows how when using "Tarpeian bloat control", the probability of being assigned a very unfit fitness value for above average size program can be adjusted to control the average program size.

For our paradigm, I may consider using what is known as "lexicographic parsimony pressure" [21]. In lexicographic parsimony pressure, size is only used as a tie-breaker in determining the better fitness when two individuals have equal fitness. The agent selection mechanism must only depend on the comparison of agents rather than an absolute fitness value. Tournament selection fits this restriction.

In this thesis I will not rely on any bloat control mechanisms. There are two reasons for this. Firstly, the finite fixed sized array of nodes used in Cartesian genetic programming puts a natural cap on program size. Lastly, parsimony pressure or lexicographic parsimony pressure makes otherwise neutral mutations non-neutral, reducing the benefits due to neutrality discussed in [16, 15] and in section 2.1.4.

2.4.4 Elitist strategy

The "elitist strategy" refers to a condition where the best individual from a generation is preserved unchanged for the next generation [11, pg. 113]. In addition to the population of $M >> 0$ agents, there is an additional slot where the best individual yet encountered is copied to. This individual I will refer to as the "prime" agent. The prime agent may partake in the creation of the M ordinary agents in the next generation. Once the M agents in the next generation have been created, the prime agent may be replaced by an ordinary agent that performs strictly better than the prime agent on the current test cases. If the prime agent and its contender both have the a fitness of 0 (which is the optimal fitness), then the prime agent may be replaced if

the prime agent did not have a fitness of 0 in some of the previous generations. The test cases are randomly chosen for each generation so a single agent's fitness may vary from generation to generation.

The rationale for using the elitist strategy is the same as the rationale for self adaptive parameters, in that I seek to protect "fragile" agents with good fitness. While the self adaptive parameters attempt to protect agents from harmful mutations, the elitist strategy carries the best agent found so far from generation to generation with no mutation. Near optimal agents are much more likely to be adversely affected by mutation due to the fact that there is less room for further improvement. By retaining the best agent found so far, the best agent can continue to contribute to the search without being "lost".

Chapter 3

Unbounded problems and Internal memory

In this section I will make a distinction between problems that have a "finite domain" vs problems that have a "discrete unbounded domain" using variations of what is known as the "Artificial Ant Problem". The "Artificial ant problem" is *not* what is being studied in this thesis, and is only being used to highlight the distinction between bounded and unbounded problems, and also to highlight the capabilities of increasing levels of internal memory. In this section I will also discuss the importance and capabilities of internal memory for each agent, and also discuss the internal memory schemes of previous works.

One of the traditional problems used as a benchmark for genetic programming is the "Artificial Ant Problem", a description of which can be found in [11]. In the artificial ant problem, an "artificial ant" is confined to a toroidal rectangular grid that contains a trail of food. The ant, and each grain of food can each only occupy one grid square. If the ant passes over a square containing food, the food is consumed. The goal is to program an ant that will consume every grain of food. The ant can be oriented in four different directions, and is able to determine if the square directly in front of it contains food or not. The image below shows the traditional "Santa Fe trail" that is commonly used as the ant's "environment". The green square is the ant's starting position and orientation, and the brown squares denote the food. The environment itself is a 32×32 toroidal grid and a total of 89 pellets are present. The Santa Fe Trail is shown in figure 3.1.

The ant's phenotype is an expression tree consisting of the following symbols:

- VOID(): Denotes an empty program.
- MOVE(): Moves the ant forward in the direction it is facing by exactly one square.
- TURN-LEFT(): Swivels the ant 90° counter clockwise.
- TURN-RIGHT(): Swivels the ant 90° clockwise.
- PROG2(P_1, P_2): Runs programs P_1 and P_2 in sequence.
- PROG3(P_1, P_2, P_3): Runs programs P_1, P_2, and then P_3 in sequence.
- IF-FOOD(P_1, P_2): Runs program P_1 if food is detected ahead, otherwise runs program P_2.

I have referred to the ant's internal expression tree as its "phenotype" as the "genotype" may denote the expression tree in a completely different manner. The "genome" itself may be a string of integers in the case of Grammatical evolution, or an array of nodes in the case of Cartesian genetic programming.

In the traditional artificial ant problem, the ant's environment is fixed. This problem is of course finite, due to the constancy of the environment and moreover due to the fact that the environment is limited in size. For me to consider the problem to be "unbounded", there must be an infinite number of possible

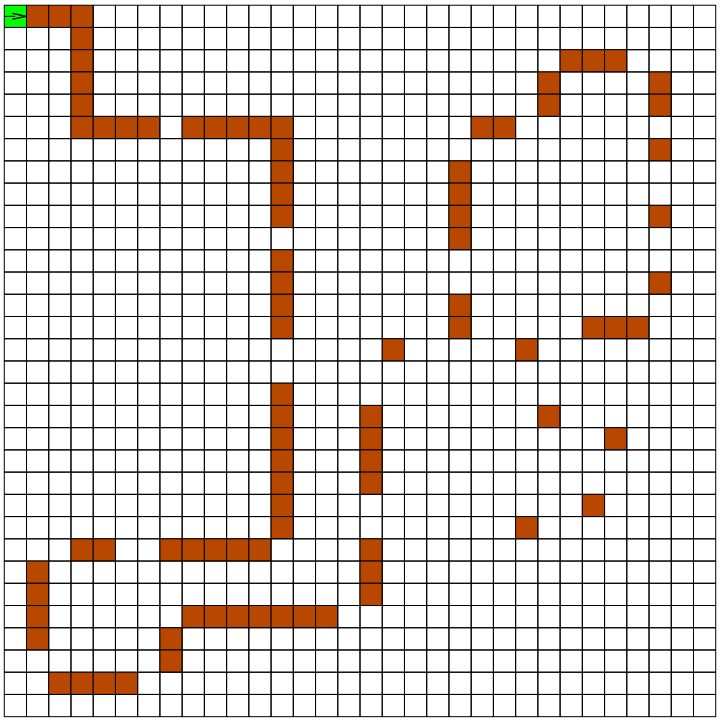


Figure 3.1: The Santa Fe ant trail. Image replicated from [11, pg. 55].

environments, or a single environment with infinite size. To make the artificial ant problem "unbounded", I may assume that the ant is not going to be placed in a fixed environment like that of the Santa Fe trail. The ant can be presented with a randomly generated environment with arbitrarily chosen dimensions, and arbitrarily laid paths of food pellets. The grid will still be toroidal in nature. I will call this problem the "generalized artificial ant problem". It should be noted that an ant that solves the Santa Fe trail is also likely to solve a randomly generated trail like the arbitrarily generated trail shown in figure 3.2. The Santa Fe trail can be seen as a testing scenario for the "generalized artificial ant problem".

An ant that only uses the aforementioned symbols as part of its program is only a finite automaton with no mechanism for storing data for the long term. This memory scheme, where the only stored value is the current position in the program, is known as an "implicit memory scheme" [25]. As a result, the ants will only ever be able navigate straightforward trails like the one shown in figure 3.2. While the "generalized artificial ant" problem is now "unbounded", I have seen that it is trivially different from the original problem in the sense that ants evolved on the Santa Fe trail will likely work for a general environment as well. To make the "unbounded" aspect truly meaningful, the ant's environment will have to consist of more than just a straightforward path. One way the space of environments can be expanded beyond the limited environment of the Santa Fe trail is to instead have a "network" of paths such as the environment shown in figure 3.3.

To navigate the network in figure 3.3, the ant must have what is known as "explicit" internal memory [25]. In particular the ant needs a way of recording its past moves so it can backtrack to other branches.

I will now discuss internal memory schemes that have been used in previous works. [6] continues to explore the artificial ant problem. The set of symbols available to the expression trees is expanded to include symbols that represent loops. The symbols that are introduced are:

- FOR-LOOP1(N, P): Runs program P for N iterations.
- FOR-LOOP2(N_1, N_2, P): Runs program P for $N_2 - N_1 + 1$ iterations.

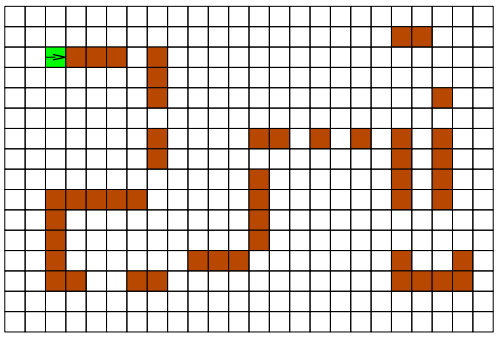


Figure 3.2: An arbitrarily generated ant trail.

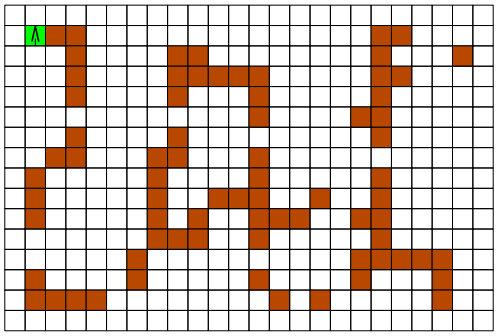


Figure 3.3: An arbitrarily generated network of paths that requires explicit internal memory to traverse completely.

Differing from the paradigm commonly used for the artificial ant problem, in [6] the program is executed only once instead of repeatedly like in the original artificial ant problem. The loops must therefore account for all the necessary steps. In this paradigm, the memory scheme is still implicit, but due to the loop symbols the programs have many more internal states than programs without the loop symbols. The number of internal states is still finite however. The loops themselves can be "unwound" to produce much larger programs of fixed length that have the same effect as a program without loops. In [6], artificial ants that use and do not use the loop symbols are evolved in an environment where the pellets are arranged in large squares. The population that was allowed use of the loop symbols evolved elegant programs that enabled the ant to consume all of the food pellets. When loop symbols were forbidden, only one extremely complicated solution was produced that had the ant consume every food pellet. While having loop symbols does not escape the realm of finite automata, it is a step towards more powerful memory schemes.

In [25], the programming of agents that use explicit internal memory is discussed. Explicit internal memory is an explicit data storage that the program can write and read quantities from. The problem being investigated by [25] is the "Asteroid Avoidance" problem which requires piloting a space craft around moving asteroids. At each time frame, the space craft can detect nearby asteroids, but not their velocities. This means that the space craft must have some internal memory in order to predict the asteroid's trajectories. The space craft's internal memory is a vector of 20 real numbers. I will now examine this memory scheme in the context of the artificial ant problem.

If the internal memory is a finite length vector, and the entry of each vector comes from a finite domain, the ant will still function as a finite automata, but in this case information can still be passed between successive calls to the program which is not possible with implicit memory. If the entry of each vector comes

from an infinite domain such as the set of integers or real numbers, the ant will no longer function as a finite automaton, but this still gives limitations on the type of data that can be stored. It is theoretically possible to store a vector of discrete quantities as an integer, but accessing these quantities will require a complicated mathematical expression.

In [31], an explicit internal memory scheme consisting of a fixed length vector of bounded integers is used to evolve an agent tasked with pushing boxes out of the center of a fixed environment. In both [25] and [31], the use of explicit internal memory outperformed experiments that had no memory or only implicit memory.

In [30], an internal memory paradigm that uses an unbounded array is presented. Entries are referenced via integer addresses. It is shown in [30] that using an unbounded array of integers as internal memory allows an agent to simulate an arbitrary Turing machine. Given an arbitrary Turing machine, there is a program that enables the agent to simulate one step of the Turing machine each time it is called. [30] does not perform any experiments with the proposed paradigm however. The ability to imitate a Turing machine is the expressive power that I will implement for my paradigms.

The problem that I am attacking in this thesis is the problem of generating a decider for an arbitrary language that runs in real time. This problem is of course, unbounded due to the fact that the domain of possible words is infinite. Without an explicit memory scheme only regular languages are decidable. Evolving finite automatons that serve as deciding programs for a regular language has been discussed in [4, 9].

Evolving programs with a single stack that serve as deciding programs for a context free language has been explored in [34]. Evolving production rules for context free grammars has been discussed in [33].

Evolving Turing machines that serve as deciding programs for non-context free languages has been explored in [28, 18, 5]. While [28, 18, 5] evolve Turing machines that decide non-context free languages, the deciding programs evolved by my paradigms are designed to run in real time.

Chapter 4

Test Languages

This section will list the languages that have been studied by previous works, some of which I will use as test languages.

Below are 7 regular languages explored in [4, 9]. These regular languages are known as the "Tomita regular languages". In [4], a deciding finite automaton is evolved for all but one of the "Tomita regular languages", in particular TL_6. In [9], a deciding finite automaton is evolved for all 7 of the "Tomita regular languages".

The "Tomita regular languages" are [4, 9]:

- $TL_1 = b^*$
- $TL_2 = (ba)^*$
- $TL_3 = \{s \in \{a, b\}^* | \text{no odd length block of a's can follow an odd length block of b's}\}$
- $TL_4 = \{s \in \{a, b\}^* | \text{no aaa substrings}\}$
- $TL_5 = \{\text{“”}, a, b\} \cup a\{a, b\}^*a \cup b\{a, b\}^*b$
- $TL_6 = \{s \in \{a, b\}^* | \#\text{ of a's - \# of b's} \equiv 0 \mod 3\}$
- $TL_7 = a^*b^*a^*b^*$

The context free languages studied in [34] are:

$$CFL_1 = \{a^n b^n | n \geq 0\}$$

and the parentheses matching problem:

$$CFL_2 = \{s \in \{\text{“(”}, \text{“)”}\}^* | \text{parentheses are balanced}\}$$

The genetic programming paradigm presented in [34] successfully evolved deciding programs for both languages.

The context free languages studied in [33] are CFL_2 and:

$$CFL_3 = \{s \in \{a, b\}^* | \#\text{ of a's} = \#\text{ of b's}\}$$

The genetic programming paradigm presented in [33] successfully evolved production rules for CFL_2, but not CFL_3.

The non-regular languages studied in [28] are CFL_1 and:

$$NCFL = \{a^n b^n a^n | n \geq 0\}$$

The genetic programming paradigm presented in [28] successfully evolved deciding Turing machines for both CFL_1 and NCFL.

The non-regular languages studied in [18] are CFL_1, NCFL, and

$$\text{CFL}_4 = \{a^n b^{n+1} | n \geq 0\}$$

and,

$$\text{NCFL}_2 = \{a^n b^n c^n | n \geq 0\}$$

The genetic programming paradigm presented in [18] successfully evolved deciding Turing machines for CFL_1, CFL_4, NCFL, and NCFL_2.

When I measure the performance of the two paradigms that I will describe below, I will use all seven Tomita test languages and the languages, CFL_1, CFL_2, CFL_3 and NCFL. I will also use some other languages that will be described later.

Chapter 5

A real-time Turing complete paradigm: RTGP_1

In this section I will describe a paradigm that is capable of generating a decider for a target language that runs in real time. As indicated previously, the decider will be able to emulate one step of a Turing machine for each character processed. I will refer to this paradigm as RTGP_1 (Real Time Genetic Programming). The approach that I am taking is a straightforward approach where each program involves IF statements as its main approach to logic. The agent will be provided with each letter in sequence and will update its internal memory, which is denoted by x. x references a "memory node". A memory node is an object that stores a single bit (b), a pointer (p), and a "next" pointer. The "next" pointer references another unique memory node and is never subject to change. The internal memory x is effectively a linked list. The "pointer" (p) field can point to any other memory node in the linked list. For implementation purposes, a memory node is not created until it has been accessed by a read command, a write command, or has its address assigned to a pointer field (p). Initially all memory nodes have their bit field (b) set 0, and their pointer field (p) points to themselves. As can be seen, this memory scheme can store an unbounded number of bits in a linked list, and the presence of pointer fields allow for remote locations in the linked list to be accessed quickly. Figure 5.1 depicts the initial layout of the internal memory, while figure 5.2 depicts one possible configuration of the internal memory after it has been manipulated by the agent.

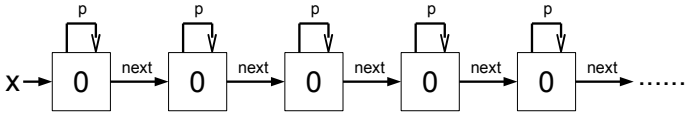


Figure 5.1: Initial internal memory configuration for RTGP_1.

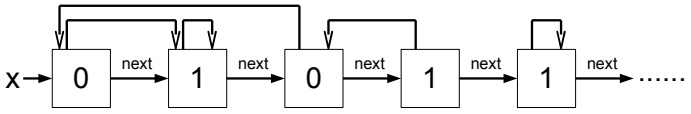


Figure 5.2: Possible internal memory configuration for RTGP_1.

Let Σ denote the set of possible letters that may comprise strings. I will refer to each of these letters

using an index from the range: $\{0, 1, \ldots, |\Sigma| - 1\}$.

I will now describe the programs that comprise each agent. Each of these programs is an expression tree that will be embedded in the array of nodes required for Cartesian genetic programming. All of the programs will share the same array. The programs that comprise an agent are:

- StartUp: This routine prepares the internal memory to receive input.
- Input0: This routine processes the input of letter "0".
- Input1: This routine processes the input of letter "1".
- .....
- Input($|\Sigma| - 1$): This routine processes the input of letter "$|\Sigma| - 1$".
- EndOfInput: This routine prepares the internal memory to return the output bit. (true if the word belongs to the target language, and false if otherwise)
- Output: Not actually a routine, but an address for the final output within the internal memory.

When an agent A is given a word w, A begins by calling "StartUp". Each letter of w is processed by A in sequence, and if $l \in \{0, 1, \ldots, |\Sigma| - 1\}$ is the letter currently being processed, A calls "Input(l)". After all characters have been processed, "EndOfInput" is called. The output is returned by the reference "Output".

The symbols that I will use for the expression trees are separated into two categories: "program" (P) symbols, and "reference" (R) symbols. Categories "program" and "reference" are disjoint. Symbols in category P can have symbols from P and R as children, but symbols in category R can only have symbols from R as children. Due to this hierarchy I will modify the row of nodes in Cartesian genetic programming as follows: The row of nodes will consist of two sections. Starting with root node(s) at the left, I will first have a section of $N_p \geq 2$ nodes each whose symbol must come from P. To the right of this section, I have section of $N_r \geq 2$ nodes each whose symbol must come from R. This setup ensures that each agent always has syntactically correct expression trees. It should be noted that in a single agent there are $|\Sigma| + 2$ expression trees that denote programs, so there are in fact $|\Sigma| + 2$ root nodes to the far left of the array that can only point to P nodes. "Output" is the one expression that denotes a reference and will have its root node only able to point to the R nodes. This scenario is depicted in figure 5.3.

Nodes from category P except the rightmost node can have symbols that require up to 2 children from P and up to 2 children from R. Even though not every symbol uses all of these 4 children, all P nodes (excluding the rightmost node) will be equipped with these child pointers in case the symbol is mutated to another symbol that requires these pointers. The rightmost node in category P will just have the 2 children from R. Likewise, nodes from category R except the rightmost node can have symbols that require up to 1 child from R, and even though not all symbols use this child, all nodes except the rightmost node will be equipped with a child from R.

If a node n is assigned a symbol s not all children may see use. I express which children see use using the notation:

$$s : \langle C_1 \times C_2 \times \cdots \times C_k \rangle$$

where $C_1, C_2, \ldots, C_k$ denote the children of n. $\emptyset$ indicates that a child is not being used.

For P nodes; P_1 and P_2 denote the children from P and R_1 and R_2 denotes the children from R.

For R nodes, the child is denoted by R_1.

The symbols in category "program" (P) are:

- Void: $\langle \emptyset \times \emptyset \times \emptyset \times \emptyset \rangle$
- Combine: $\langle P_1 \times P_2 \times \emptyset \times \emptyset \rangle$
- If: $\langle P_1 \times P_2 \times R_1 \times \emptyset \rangle$

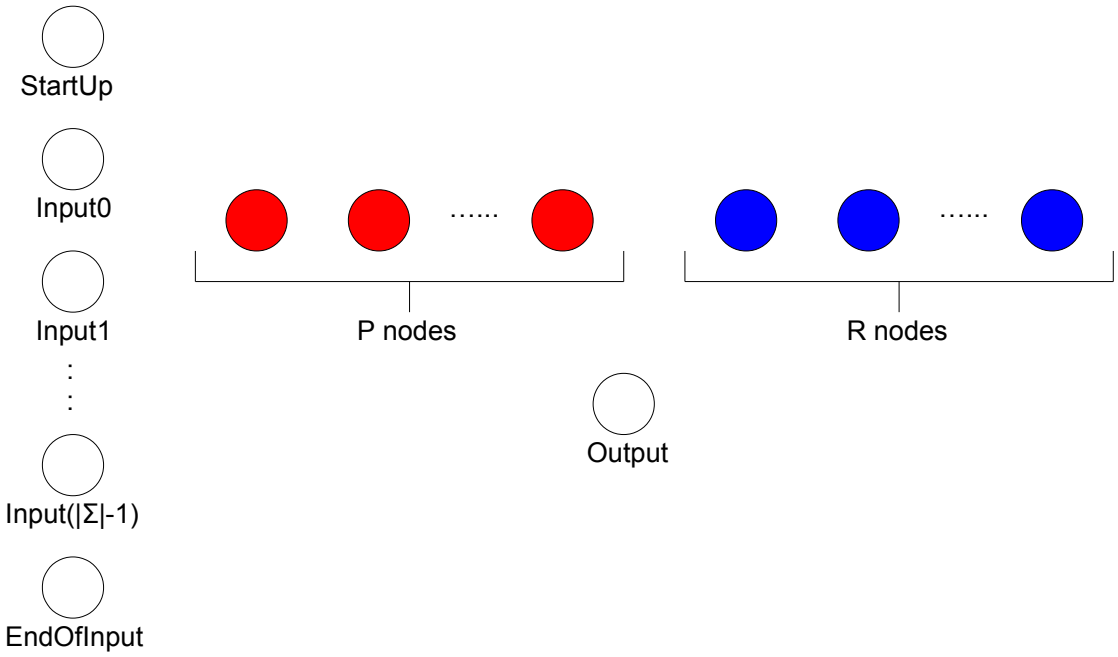


Figure 5.3: The internal structure of an agent in the RTGP_1 paradigm.

- Set0: $\langle \emptyset \times \emptyset \times R_1 \times \emptyset \rangle$
- Set1: $\langle \emptyset \times \emptyset \times R_1 \times \emptyset \rangle$
- SetPointer: $\langle \emptyset \times \emptyset \times R_1 \times R_2 \rangle$

The symbols in category "reference" (R) are:

- X: $\langle \emptyset \rangle$
- Next: $\langle R_1 \rangle$
- Pointer: $\langle R_1 \rangle$

The descriptions of each symbol are shown below.

- Void() denotes an empty program with no statements.
- Combine(P_1, P_2) denotes a program which runs programs P_1 and P_2 is sequence, one after the other. This symbol is used to link commands similar to "PROG2" from the artificial ant problem.
- If(P_1, P_2; R_1) denotes a program that examines the bit field (b) in the node referenced by R_1. If $R_1 = 1$, P_1 is executed, and if $R_1 = 0$, P_2 is executed.
- Set0(R_1) sets the bit field in the node referenced by R_1 to 0.
- Set1(R_1) sets the bit field in the node referenced by R_1 to 1.

- SetPointer(R_1, R_2) sets the pointer field in the node referenced by R_1 to the node referenced by R_2.
- X() is agent's internal memory variable, and references the root memory node.
- Next(R_1) references the node that is pointed to by the "next" pointer of the node referenced by R_1.
- Pointer(R_1) references the node that is pointed to by the pointer field of the node referenced by R_1.

The next several sections will describe the following. I will first give a proof of the real-time Turing completeness of RTGP_1, that is that RTGP_1 can emulate one time step of a Turing machine with each received character. Next I will derive upper bounds for some of the minimal $\langle N_p, N_r \rangle$ pairs that describe the number of P nodes and R nodes required to describe a real time Turing machine given the number of states q; the size of the input alphabet $|\Sigma|$; and the size of the tape alphabet $|\Gamma|$. These upper bounds will be useful when setting up our experiments. I should note that in practice, one likely has no knowledge of the number of nodes required. The ideal number of nodes will have to be determined experimentally. Lastly, I will give example deciding programs for the languages TL_2 and CFL_2 implemented in the RTGP_1 paradigm. These example deciding programs will be engineered to decide the given language (TL_2 and CFL_2) to illustrate to the reader how such programs can be expressed in the RTGP_1 paradigm.

5.1 The real-time Turing completeness of RTGP_1

RTGP_1 is real time Turing complete. I will assume wlog that the tape for the real-time Turing machine I wish to emulate extends infinitely in only one direction, namely to the right. I will also ascribe similar directions to the linked list in memory. "Left" will denote towards the root, while "right" will denote away from the root.

I will let ϵ denote the blank symbol.

I will let Γ denote the set of symbols that the Turing machine will use not counting the blank symbol.

Define $l = \max(1, \lceil \log_2(|\Gamma| + 1) \rceil)$.

l is the number of bits required to denote a symbol from $\Gamma \cup \{\epsilon\}$.

I will let Q denote the set of internal states of the Turing machine.

Define $l_q = \max(1, \lceil \log_2(|Q|) \rceil)$.

l_q is the number of bits required to denote a state from Q.

I will partition the linked list of memory nodes as follows: The first l_q nodes will store bits that denote the current state of the Turing machine. The rest of the nodes will be grouped into blocks of length l. Each of these blocks is a stand in for a cell of the Turing machine, starting with the left most cell and counting to the right.

The head for the Turing machine is denoted by the pointer field of X(). Pointer(X()) references the leftmost memory node of the block that represents the cell under observation.

In one step, a real-time Turing machine reads its current state as well as the letter under the tape head. Depending on the state and letter the Turing machine updates the state and the letter under the tape head and possibly moves the tape head left or right.

The block under the tape head can be reached through the reference Pointer(X()). Using the "If" symbol, it's easy to create a subroutine that reads both the state and the letter under the tape head and performs different actions depending on what is read. Using the "Set0" and "Set1" symbols, it's easy to update both the state and the symbol under the tape head.

When the machine moves the tape head to the right, it first sets the pointer field of the leftmost node in the right side block to point to the leftmost node in the current block. This is so it has a reference to the current block from the right side block so the tape head can return to the current block from the right side later. It then updates Pointer(X()) to point to the first node in the right side block. This can be done using the following sequence of commands:

SetPointer($\underbrace{\text{Next}(\text{Next}(\ldots\text{Next}}_{l}(\text{Pointer}(\text{X}()))\ldots))$, Pointer(X()))

SetPointer(X(), $\underbrace{\text{Next}(\text{Next}(\ldots\text{Next}}_{l}(\text{Pointer}(\text{X}()))\ldots)))$

When the machine moves the tape head to the left, it updates Pointer(X()) to point to the leftmost node in the left side block which is referenced by Pointer(Pointer(X())). This reference was created when the head first moved to the right from the left side block to the current block. If the head is at the leftmost cell, Pointer(Pointer(X())) references the same node as Pointer(X()) so the tape head does not move. The command is:

SetPointer(Pointer(X()), Pointer(Pointer(X())))

Figure 5.4 depicts how the pointer fields in the list x are setup to imitate the tape of a Turing machine as described above. The top part of the diagram depicts the list, while the lower part depicts the Turing tape.

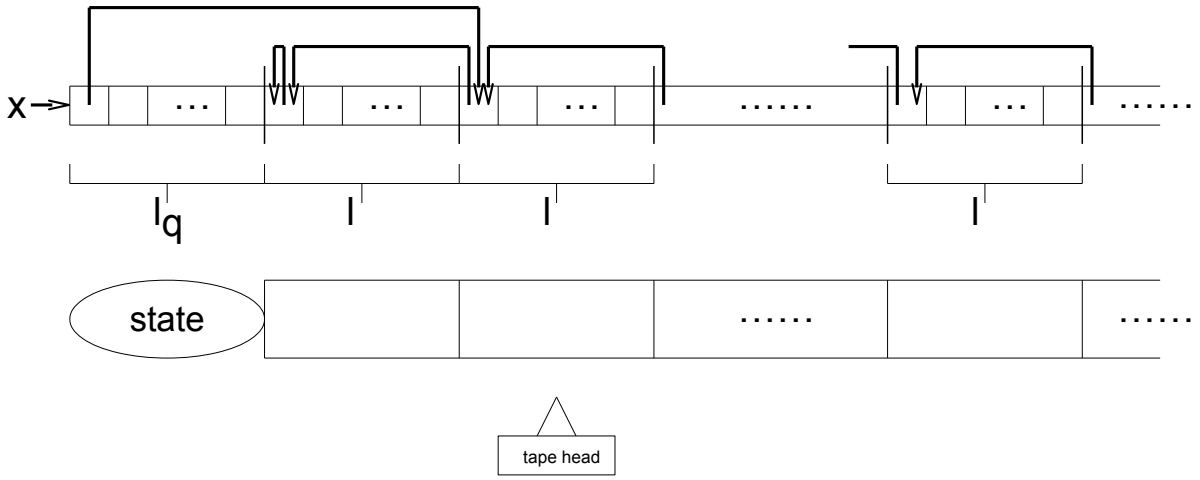


Figure 5.4: A diagram of RTGP_1 imitating a Turing machine. The top part depicts the list, while the lower part depicts the Turing tape.

This finishes the proof that my paradigm can emulate a Turing machine with each step. □

5.2 Size of the representation

In this section, I will derive upper bounds for some of the minimal $\langle N_p, N_r \rangle$ pairs that describe the number of P nodes and R nodes required to describe a real time Turing machine with $q \geq 2$ states, an input alphabet of Σ, and a tape alphabet of Γ, where $|\Gamma| \geq 1$.

I will give upper bounds on the minimal numbers of nodes required to express a real-time Turing machine using the style for emulating a real-time Turing machine described in the proof above of real-time Turing completeness (section 5.1).

The references that the implementation will require are:

- There are $\lceil \log_2(q) \rceil$ references for accessing the current state:
  - X()

  - Next(X())
  - ...
  - $\underbrace{\text{Next}(\text{Next}(\ldots\text{Next}(}_{\lceil\log_2(q)\rceil-1}\text{X}())\ldots))$

- There are $\lceil\log_2(|\Gamma|+1)\rceil$ references for accessing the letter under the tape head.
  - Pointer(X())
  - Next(Pointer(X()))
  - ...
  - $\underbrace{\text{Next}(\text{Next}(\ldots\text{Next}(}_{\lceil\log_2(|\Gamma|+1)\rceil-1}\text{Pointer}(\text{X}()))\ldots))$

- The reference required to set the tape head to its initial position is:
  $\underbrace{\text{Next}(\text{Next}(\ldots\text{Next}(}_{\lceil\log_2(q)\rceil}\text{X}())\ldots))$

- The reference required to move the tape head to the right is:
  $\underbrace{\text{Next}(\text{Next}(\ldots\text{Next}(}_{\lceil\log_2(|\Gamma|+1)\rceil}\text{Pointer}(\text{X}()))\ldots))$

- The reference required to move the tape head to the left is:
  Pointer(Pointer(X()))

This is a total of $\lceil\log_2(q)\rceil + \lceil\log_2(|\Gamma|+1)\rceil + 3$ references. Since the references can be generated from each other using only 1 symbol for each new reference, this requires $\lceil\log_2(q)\rceil + \lceil\log_2(|\Gamma|+1)\rceil + 3$ reference nodes.

I will now indicate how many program nodes are required.

For each of the $\lceil\log_2(q)\rceil + \lceil\log_2(|\Gamma|+1)\rceil$ references that access state and tape letter bits, there must exist "Set0" and "Set1" commands. This yields $2\lceil\log_2(q)\rceil + 2\lceil\log_2(|\Gamma|+1)\rceil$ new program nodes.

There are q possible states that the $\lceil\log_2(q)\rceil$ state bits could be set to. A sub-routine that sets the state to state i consists of up to $\lceil\log_2(q)\rceil$ "Set0" or "Set1" nodes which are already accounted for. These commands are linked by up to $\lceil\log_2(q)\rceil - 1$ "Combine" nodes. The sub-routine for setting the state to state i introduces $\lceil\log_2(q)\rceil - 1$ new "Combine" nodes. The total number of new "Combine" nodes introduced is $q(\lceil\log_2(q)\rceil - 1)$. By similar reasoning, creating sub-routines that set the letter under the tape head introduces $(|\Gamma|+1)(\lceil\log_2(|\Gamma|+1)\rceil - 1)$ new "Combine" nodes.

The following 4 sub-routines allow for initializing the tape head, as well as for moving it left or right. There is also an additional implicit "Combine" node when moving the tape head to the right yielding a total of 5 new program nodes.

Initialize the tape head:
SetPointer(X(), $\underbrace{\text{Next}(\text{Next}(\ldots\text{Next}(}_{\lceil\log_2(|\Gamma|+1)\rceil}\text{X}())\ldots))$

Move the head to the right:
SetPointer($\underbrace{\text{Next}(\text{Next}(\ldots\text{Next}(}_{\lceil\log_2(|\Gamma|+1)\rceil}\text{Pointer}(\text{X}()))\ldots))$, Pointer(X()))
SetPointer(X(), $\underbrace{\text{Next}(\text{Next}(\ldots\text{Next}(}_{\lceil\log_2(|\Gamma|+1)\rceil}\text{Pointer}(\text{X}()))\ldots))$

Move the head to the left:
SetPointer(Pointer(X()), Pointer(Pointer(X())))

The routine "StartUp" simply initializes the tape head. The initial state is denoted by a string of $\lceil \log_2(q) \rceil$ 0's. No new program nodes are introduced.

To read the state and letter under the tape head requires $q(|\Gamma|+1)-1$ "if" statements. $q(|\Gamma|+1)-1$ new "if" nodes are introduced for each "Input$i$" routine.

At the end of each branch in the tree of "if" statements in the routine "Input$i$" the machine can change the state, change the letter under the tape head, or move the tape head. Program nodes have already been accounted for for each of these actions, so up to 2 "Combine" nodes are necessary to link the sub-routines at the end of each branch in the tree of "if" statements. A total of $2q(|\Gamma|+1)$ new "Combine" nodes are introduced for each "Input$i$" routine.

The routine "EndOfInput" only introduces at most $q(|\Gamma|+1)-1$ new "if" statements. Only 1 "Set0" or "Set1" command exists at the end of each branch which generates the output at preferably X().

In summary:

- There is at most $\lceil \log_2(q) \rceil + \lceil \log_2(|\Gamma|+1) \rceil$ "Set0" nodes.
- There is at most $\lceil \log_2(q) \rceil + \lceil \log_2(|\Gamma|+1) \rceil$ "Set1" nodes.
- $q(\lceil \log_2(q) \rceil - 1)$ new "Combine" nodes are introduced by the sub-routines that set the state.
- $(|\Gamma|+1)(\lceil \log_2(|\Gamma|+1) \rceil - 1)$ new "Combine" nodes are introduced by the sub-routines that set the letter under the tape head.
- 5 new program nodes are introduced by the sub-routines that initialize and move the tape-head.
- For each $i = 0, 1, \ldots, |\Sigma|-1$; the routine "Input$i$" introduces at most $q(|\Gamma|+1)-1$ new "if" nodes and $2q(|\Gamma|+1)$ new "Combine" nodes.
- At most $q(|\Gamma|+1)-1$ new "if" nodes are introduced by the routine "EndOfInput"

The total number of program nodes is:

$$\begin{aligned} &2\lceil \log_2(q) \rceil + 2\lceil \log_2(|\Gamma|+1) \rceil \\ &+ q(\lceil \log_2(q) \rceil - 1) + (|\Gamma|+1)(\lceil \log_2(|\Gamma|+1) \rceil - 1) \\ &+ 5 + |\Sigma|(3q(|\Gamma|+1)-1) + q(|\Gamma|+1) - 1 \end{aligned}$$

Therefore:

$$\lceil \log_2(q) \rceil + \lceil \log_2(|\Gamma|+1) \rceil + 3$$

is an upper bound on the minimum number of reference nodes; and

$$\begin{aligned} &2\lceil \log_2(q) \rceil + 2\lceil \log_2(|\Gamma|+1) \rceil \\ &+ q(\lceil \log_2(q) \rceil - 1) + (|\Gamma|+1)(\lceil \log_2(|\Gamma|+1) \rceil - 1) \\ &+ 5 + |\Sigma|(3q(|\Gamma|+1)-1) + q(|\Gamma|+1) - 1 \end{aligned}$$

is an upper bound on the minimum number of program nodes.

In the case where all I am trying to simulate is a finite automaton with $q \geq 2$ states,

$$\lceil \log_2(q) \rceil$$

is an upper bound on the minimum number of reference nodes; and

$$2\lceil \log_2(q) \rceil + q(\lceil \log_2(q) \rceil - 1) + (|\Sigma|+1)(q-1)$$

is an upper bound on the minimum number of program nodes.

5.3 Example deciding programs using RTGP_1

In this section I will give some example deciding programs to better illustrate how the RTGP_1 paradigm works. The languages for which I will provide examples are TL_2 and CFL_2. These programs will be engineered to give the reader an idea as to how programs that decide a specific language can be constructed in the RTGP_1 paradigm.

5.3.1 Example deciding program for TL_2 using RTGP_1

In this section, I will give an example deciding program for the 2^{nd} Tomita regular language. Recall that the 2^{nd} Tomita regular language is $\text{TL}_2 = (ba)^*$. I will index the symbols from the alphabet as follows: $\Sigma = \{a, b\} = \{0, 1\}$.

The subroutines that comprise the program are:

```
StartUp:
Void()

Input0:
if X() then
  Void()
else
  if Next(X()) then
    Set0(Next(X()))
  else
    Set1(X())
  end if
end if

Input1:
if X() then
  Void()
else
  if Next(X()) then
    Set1(X())
    Set0(Next(X()))
  else
    Set1(Next(X()))
  end if
end if

EndOfInput:
if X() then
  Set0(X())
else
  if Next(X()) then
    Void()
  else
    Set1(X())
  end if
end if
```

Output = X()

This program works by representing the 3 states of the memory using the two bit sequences 00, 01, and 10. The first bit is X(), while the second bit is Next(X()). State 00 means that a b is expected as the next character. State 01 means that an a is expected as the next character. State 11 means that the string is not part of the language. Only if the final state is 00 is the string accepted.

Figure 5.5 depicts the above subroutines as expression trees.

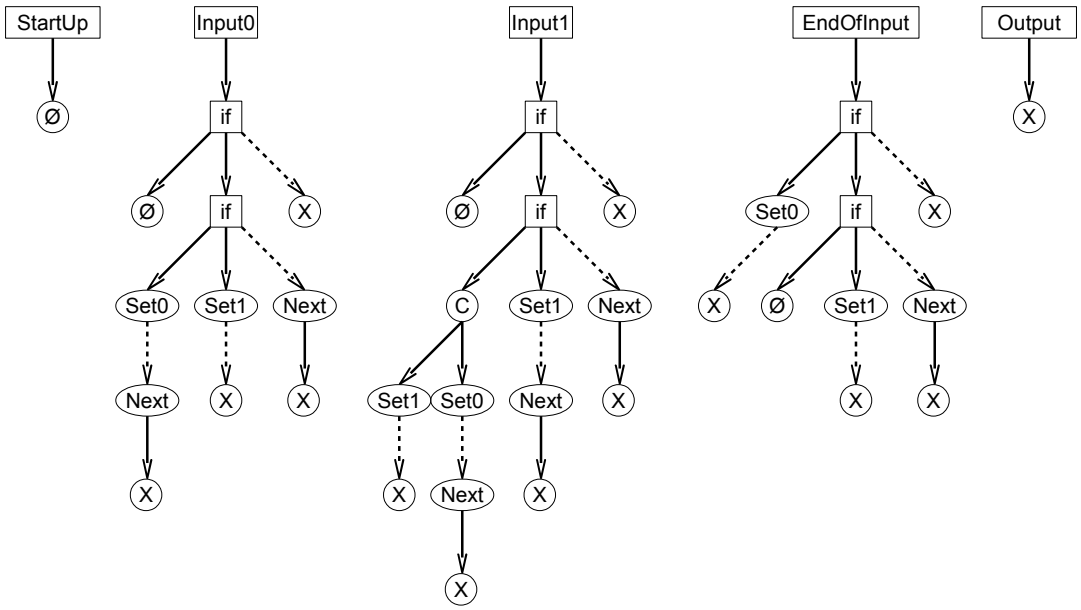


Figure 5.5: Expression trees for TL_2 in the $RTGP_1$ paradigm. ∅ denotes the "Void" symbol, C denotes the "Combine" symbol, and the dashed edges denote pointers from program (P) nodes to reference (R) nodes.

The precise structure of an agent in the $RTGP_1$ paradigm that implements the subroutines is shown below. The nodes are listed in order from left-to-right. Recall that all program nodes except the rightmost node has 2 program node and 2 reference node children; and that all reference nodes except the rightmost node has 1 reference node child. Unused children are not part of the description.

StartUp: P0
Input0: P6
Input1: P9
EndOfInput: P11
Output: R0

Program nodes:

P11: If(P1, P10; R0)
P10: If(P0, P2; R1)
P9: If(P0, P8; R0)
P8: If(P7, P4; R1)
P7: Combine(P2, P3)
P6: If(P0, P5; R0)

P5: If(P3, P2; R1)
P4: Set1(R1)
P3: Set0(R1)
P2: Set1(R0)
P1: Set0(R0)
P0: Void()

Reference nodes:

R1: Next(R0)
R0: X()

5.3.2 Example deciding program for $\mathbf{CFL}_2$ using $\mathbf{RTGP}_1$:

In this section I will give a deciding program for the parenthesis matching problem, CFL_2. I will index the symbols from the alphabet as follows: $\Sigma = \{`(', `)'\} = \{0, 1\}$.

The subroutines are:

StartUp:
Set1(X())
Set1(Next(X()))
SetPointer(X(), Next(X()))

Input0:
SetPointer(Next(Pointer(X())), Pointer(X()))
SetPointer(X(), Next(Pointer(X())))

Input1:
if Pointer(X()) **then**
  Set0(X())
else
  SetPointer(X(), Pointer(Pointer(X())))
end if

EndOfInput:
if Pointer(X()) **then**
  Void()
else
  Set0(X())
end if

Output = X()

This program works by storing the output in X(). Next(X()) always stores 1, and all subsequent nodes store 0. Pointer(X()) points to Next(X()) if there are no unpaired '('. Pointer(X()) points to $\underbrace{\mathrm{Next}(\mathrm{Next}(\ldots\mathrm{Next}(}_{n}\mathrm{Next}(\mathrm{X}()))\ldots))$ if there are n unpaired '('. X() stores 0 when there are excess ')'.

Chapter 6

Results for RTGP_1

In this section I will collect and present data related to the convergence and overall speed of RTGP_1 on several test bed problems. Important parameters for the population are:

- $M = 500$ is the size of the population (not including the prime agent).
- $p_{\text{tournament}} = 0.9$ is the probability of selecting the better agent during tournament selection.
- max_mutations $= 2$ is the maximum number of basic mutation steps during mutation.
- $\mu_\theta = 0.1$ is the probability of mutating any given θ during mutation.
- $\alpha = 0.1$ is the probability of reseting the self adaptive parameters during mutation.
- $\beta = 0.1$ is the probability of ignoring the self adaptive parameters during mutation.
- $N_p = 20$ and $N_r = 10$ are the sizes of the arrays of P and R nodes respectively.

When the genetic program succeeds in finding a solution, I will measure $P(i)$, the probability of finding a solution by generation $i = 0, 1, 2, \ldots$. I now perform an analysis described by Koza in [11]. I let $z = 99\%$ be the desired probability of success. If a genetic program were to "give up" and reset the population after i generations, the number of independent runs $R(i)$ required to find a solution by generation i with probability at least z is:

$$R(i; z) = \left\lceil \frac{\log(1 - z)}{\log(1 - P(i))} \right\rceil$$

The total number of individuals that need to be generated and tested to find a solution with $z = 99\%$ probability where the population resets after i generational steps is:

$$I(i; z) = M \times (i + 1) \times R(i; z) = M(i + 1) \left\lceil \frac{\log(1 - z)}{\log(1 - P(i))} \right\rceil$$

$I(i; z)$ is effectively a measure of the computational "effort" required of the genetic program to solve a problem with a probability of $z = 99\%$. Also note that when $i = 0$, no run advances a single generation and the genetic program reduces to random search.

In addition to measuring the probability of success and the effort required for success, I will also keep track of the average minimum fitness for each generation to get a look at how the genetic program converges. I will denote the average minimum fitness with $f(i)$.

6.1 TL_1 using $RTGP_1$

$TL_1 = b^* = 1^*$

For each generation a set U of $T = 1000$ test strings are randomly generated using the algorithm described 2.4.1. The exact algorithm used to generate the test cases can be found in A.1.

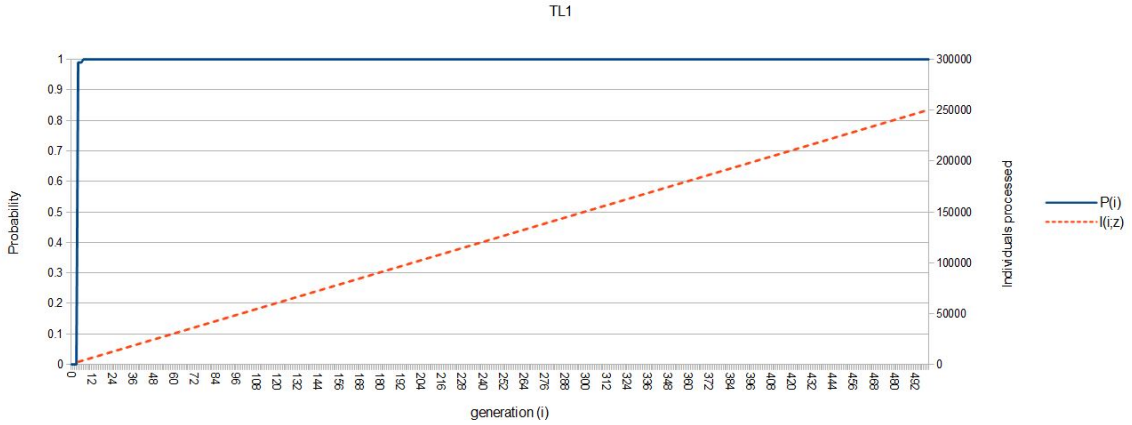
$T = 1000$ is the number of test cases used in [4], although in [4] these test cases were fixed for the entire genetic algorithm.

As was indicated at the end of section 2.4.1, the algorithm for generating the test cases already relies on advance knowledge of the language. I have noted however that the decider that will be produced by the genetic program is guaranteed to run in real time.

The termination condition occurs when the prime agent has never had a fitness other than 0, and has been the prime agent for 5 generations. This condition allows us to further test the prime agent over 5000 test cases while propagating the population at the same time. The 5000 test cases that the prime agent is tested over helps diminish the possibility of the prime agent being chosen as optimal due to a "bad" set of test cases.

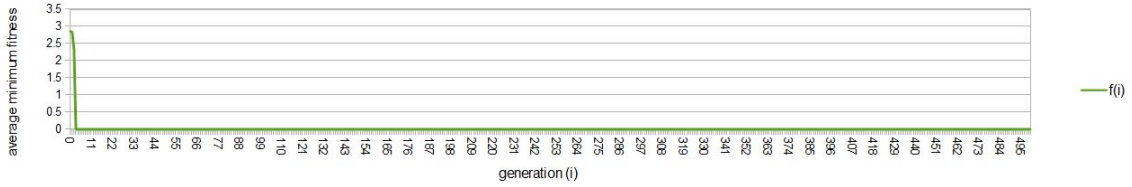
The following graphs use a sample of 100 runs with a generation limit of 500.

Below is a graph displaying $P(i)$ (solid curve) and $I(i; 99\%)$ (dashed curve) vs generation i for TL_1:



The gap before the start of the $I(i; z)$ curve indicates that the required effort is "infinite": a solution was not found before generation i.

The next graph shows the average minimum fitness $f(i)$ vs generation i:



As can be seen, $RTGP_1$ evolves a decider for TL_1 almost always in the initial population. The delay of 5 generations seen in the graph is due to the necessity of testing the prime agent for 5 generations so the prime agent is shown to be correct for 5000 test cases.

6.2 TL_2 using $RTGP_1$

$TL_2 = (ba)^* = (10)^*$

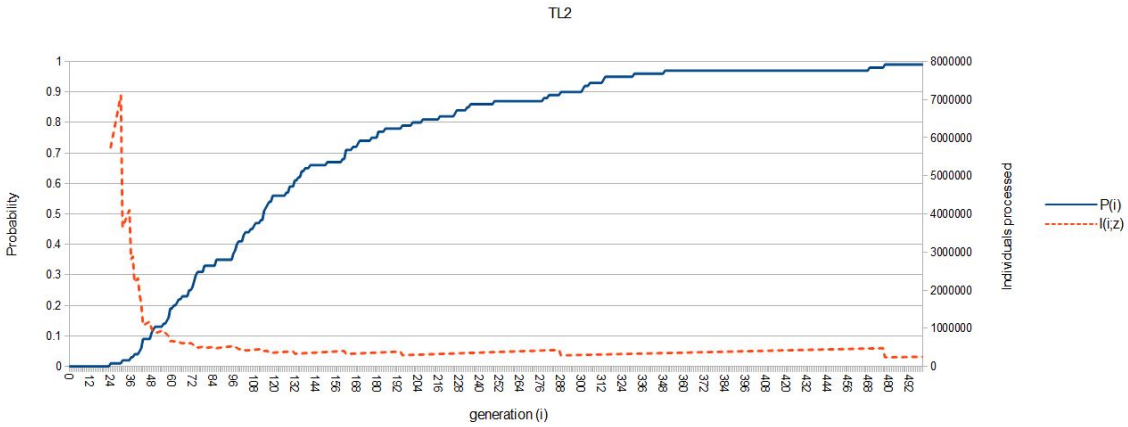
This language can be decided by a 3 state finite automaton. Using the results from 5.1, $q = 3$ and $|\Sigma| = 2$ indicates that $\lceil \log_2(q) \rceil = 2$ is an upper bound on the minimum number of reference nodes, and that $2\lceil \log_2(q) \rceil + q(\lceil \log_2(q) \rceil - 1) + (|\Sigma| + 1)(q - 1) = 2(2) + 3(2 - 1) + (2 + 1)(3 - 1) = 4 + 3 + 6 = 13$ is an upper bound on the minimum number of program nodes.

For each generation a set U of $T = 1000$ test strings are randomly generated using the algorithm described 2.4.1. The exact algorithm used to generate the test cases can be found in A.2.

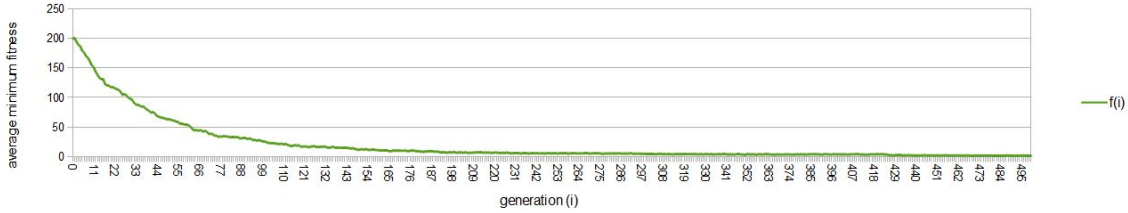
The termination condition occurs when the prime agent has never had a fitness other than 0, and has been the prime agent for 5 generations.

The following graphs use a sample of 100 runs with a generation limit of 500.

Below is a graph displaying $P(i)$ (solid curve) and $I(i; 99\%)$ (dashed curve) vs generation i for TL_2:



The next graph shows the average minimum fitness $f(i)$ vs generation i:



6.3 TL_3 using RTGP_1

$\text{TL}_3 = \{s \in \{0,1\}^* | \text{no odd length block of 0's can follow an odd length block of 1's}\}$

This language can be decided by a 5 state finite automaton. Using the results from 5.1, $q = 5$ and $|\Sigma| = 2$ indicates that $\lceil \log_2(q) \rceil = 3$ is an upper bound on the minimum number of reference nodes, and that $2\lceil \log_2(q) \rceil + q(\lceil \log_2(q) \rceil - 1) + (|\Sigma| + 1)(q - 1) = 2(3) + 5(3 - 1) + (2 + 1)(5 - 1) = 6 + 10 + 12 = 28$ is an upper bound on the minimum number of program nodes. For this problem, I will set $N_p = 40$.

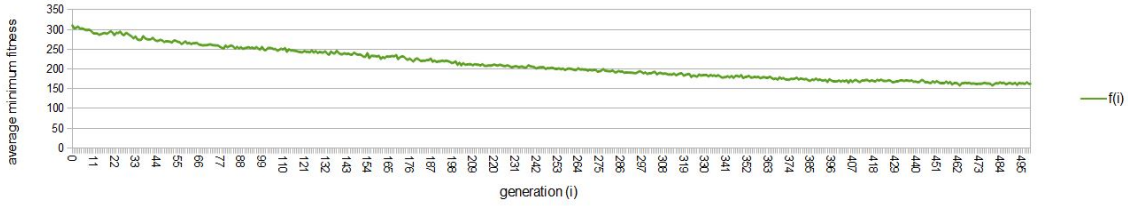
For each generation a set U of $T = 1000$ test strings are randomly generated using the algorithm described 2.4.1. The exact algorithm used to generate the test cases can be found in A.3.

The termination condition occurs when the prime agent has never had a fitness other than 0, and has been the prime agent for 5 generations.

My data uses a sample of 100 runs with a generation limit of 500.

Only 2 of the 100 runs found a decider for TL_3.

The following graph shows the average minimum fitness $f(i)$ vs generation i for TL_3:



6.4 Simple cyclic output using $RTGP_1$

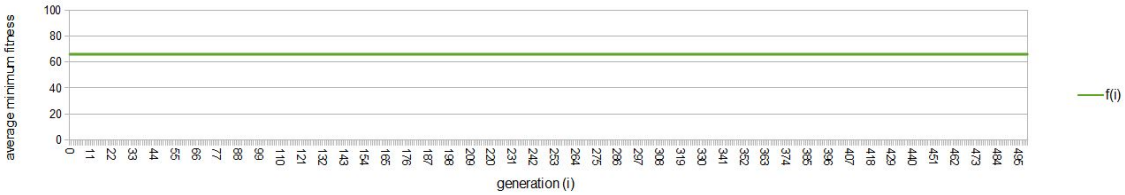
As indicated by the previous tests, $RTGP_1$ can find a decider for TL_2 but has considerable difficulty finding a decider for TL_3. To give an indication of the limits of $RTGP_1$, I will attempt to evolve a decider where the output follows a cycle with a period of 3. The alphabet of symbols will simply be: $\Sigma = \{0\}$, and the specific language will be Cyclic_Output $= \{0^n | n \text{ is divisible by } 3\}$. The set of test strings is $U = \{0^i | 0 \leq i \leq 200\}$. Due to the fitness being computed in a deterministic manner (no cases are generated randomly), the termination condition is when the prime agent has a fitness of 0.

This language can be decided by a 3 state finite automaton. Using the results from 5.1, $q = 3$ and $|\Sigma| = 1$ indicates that $\lceil \log_2(q) \rceil = 2$ is an upper bound on the minimum number of reference nodes, and that $2\lceil \log_2(q) \rceil + (|\Sigma| + 1)(q - 1) + |\Sigma| q(\lceil \log_2(q) \rceil - 1) = 2(2) + (1 + 1)(3 - 1) + 1(3)(2 - 1) = 4 + 4 + 3 = 11$ is an upper bound on the minimum number of program nodes.

My data uses a sample of 100 runs with a generation limit of 500.

No run found a decider for "Cyclic_Output".

The next graph shows the average minimum fitness $f(i)$ vs generation i for "Cyclic_Output":



As can be seen, the average minimum fitness shows no improvement as the generations progress.

6.5 Failure of $RTGP_1$

Having failed to evolve a decider whose output follows a 3-cycle as the word length increases, I will not bother testing the rest of the languages. In this section I will investigate several possible causes of the failure of $RTGP_1$ to produce deciding programs for mildly complex regular languages.

I will examine possible reasons as to why $RTGP_1$ failed to evolve a decider for the language "Cyclic_Output". This explanation will focus on the general complexity of a program/expression in $RTGP_1$ that cycles between 3 distinct states. To denote 3 distinct states in $RTGP_1$, I need to use at least 2 memory nodes from the internal memory. The pair of bits in these memory nodes can have up to 4 states 00, 01, 10 and 11. A program/expression built from the library of symbols that can cycle between these 3 of the 4 states requires at least 2 nested "if" statements. Agents that are a short distance "mutation wise" from a correct program likely express very different behavior than the correct program.

In [29] Teller notes that when using genetic programming to evolve an expression or function, small changes to the genotype may result in small changes to the phenotype. In the case of evolving an expression that is iterated over a number of steps however, small changes to the genotype "usually cause[s] complete chaos" [29] as differences in the internal state build with each iteration. The mapping of genotype to phenotype is so discontinuous, that small changes to the genotype very rarely yield small changes to the phenotype [32]. The near absence of minor mutations to the genotype that result in minor changes to the phenotype means that the fitness of the parent(s) is not at all indicative of the fitness of the children. This nullifies the reason for giving fitter agents priority in reproduction. Genetic programming is effectively reduced to random search. In the next section, I will try to "smooth" the fitness landscape by allowing for plenty of mutations that result in small/gradual changes to the phenotype.

Chapter 7

A new real time Turing complete paradigm: RTGP_2

The new paradigm will be similar to the direct state transition table/graph representation that can be found in [9, 34, 28, 18] ([4] instead uses a program that "grows" a finite automa from scratch). In these papers, the state transition graph of the program is directly embedded in either an expression tree with some semi-complicated schemes to denote back-pointers (pointers that can connect to ancestor nodes to complete cycles) [9, 34], or a graph itself [28, 18]. Since I am using Cartesian genetic programming I can allow for back pointers in the array of nodes by lifting the restriction that nodes in the array can only have nodes strictly to the right as children. I will refer to this paradigm as RTGP_2.

The array of nodes in RTGP_2 will not be broken into 2 pieces, and all nodes in the array will be created equal with the possibility of back pointers. Instead of $|\Sigma| + 3$ root nodes, there will be only 1 root node. Like the transition graph paradigms found in found in [4, 9, 34, 28, 18], I will use an internal state graph rather than a complex program to advance from state to state. I will also include a Turing tape as part of the program's internal memory. Unlike what can be simulated in RTGP_1, I will have to fix the number of symbols the embedded Turing tape can use upfront, as this quantity cannot be set during runtime. I will let Γ denote the set of letters that our embedded Turing tape can use, excluding the blank symbol. The blank symbol will be denoted by ϵ. Each node in the array is a possible internal state. Each node in the array will have 1 "label", and 2 "tags" that I will refer to as tag_0 and tag_1. "label" can have the values 0 or 1. tag_0 can have the values 0 or 1. tag_1 can have the values 0, 1, 2, or 3. Each node in the array will have $|\Sigma|$ "primary" children indexed from $0, 1, \ldots, |\Sigma| - 1$; and $|\Gamma|$ "secondary" children indexed from $0, 1, \ldots, |\Gamma| - 1$. The primary child edges will also be be assigned 1 letter from $\Gamma \cup \{\epsilon\}$ (not the child index), and 1 "direction": -1, 0, 1 2. Figure 7.1 shows the new structure of an agent and of one of the nodes.

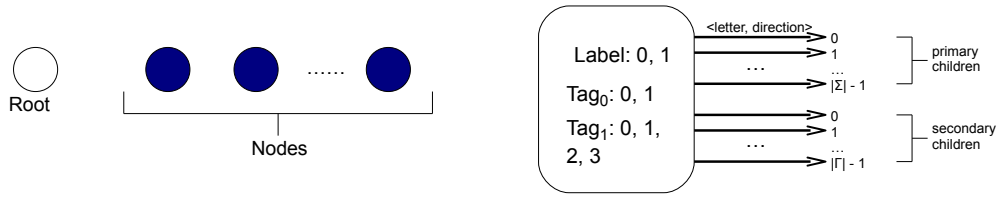


Figure 7.1: The left image depicts the structure of an agent in the RTGP_2 paradigm, and the image on the right depicts the structure of a single node.

7.1 Processing words in RTGP_2

This section will describe how an agent A processes input.

A high level description of how A processes words is as follows. The node array of A functions as a transition graph of a finite automata, where the primary edges are the transition edges for the possible input characters. The letter and direction assigned to the primary edges determines how the Turing tape is updated when the transition is used. tag_1 affects what changes can be made to the Turing tape. If $\text{tag}_0 = 1$, the node's "proxy node" is used instead of the node itself for the label, tag_1 and the primary edges. The "proxy node" is the node referenced by the current node's secondary pointer determined by the letter under tape head.

A word $w \in \Sigma^*$ is processed by an agent A as follows:

Each node in A's node array is an internal state. The current node/state will be denoted by s.

The initial state s_0 is the array node referenced by the root node. $s = s_0$.

A's Turing tape is initialized with all blank symbols, with the tape head focused on what will be referred to as the "origin cell".

A will process the letters of w in sequence.

At each step, including the final step after w has been emptied, A will first find the node s' that acts as a "proxy" for node s. A will examine "tag_0" of node s. If $\text{tag}_0 = 0$, then $s' = s$. If $\text{tag}_0 = 1$, then A will read the current cell of the Turing tape to get letter $b \in \Gamma \cup \{\epsilon\}$. If $b = \epsilon$, then $s' = s$. If $b \neq \epsilon$, then s' is the secondary child of s indexed by b.

If w has been emptied, the output is the "label" of s'. The label can be 0 or 1.

If w still has letters, let $a \in \Sigma$ be the current letter that is being processed. Let s_{new} be the primary child of s' (note the "$'$") indexed by a. A first updates s to s_{new}, without updating s' yet. Let l and d be respectively the letter and "direction" on the primary child edge indexed by a from s' to s_{new}. A now examines tag_1 of node s'. If $\text{tag}_1 = 1, 3$, then the letter in the current cell of the Turing tape changes to l. If $\text{tag}_1 = 2, 3$, then the tape head moves in direction d. $d = -1$ is "move to the left"; $d = 0$ is "no movement"; $d = 1$ is "move to the right"; and $d = 2$ returns the head to the "origin cell". In the case where the tape cell letter is changed *and* the tape head moves (i.e. $\text{tag}_1 = 3$), the tape head moves *after* the cell is updated.

Since RTGP_2 is a more direct imitation of a Turing machine, I do not need a proof that RTGP_2 can emulate the step of a Turing machine with each character consumed.

7.2 Changes in Mutation from RTGP_1 to RTGP_2

The initialization of the population of $M >> 0$ agents in the population is the same as in RTGP_1. With mutation however, there are many more mutation sites for each array node. The listing of mutation sites are:

- The root node pointer.
- Each array node n as the following mutation sites:
  - label
  - tag_0
  - tag_1
  - Each primary child pointer.
  - Each secondary child pointer.
  - Each primary child edge c of n has the following mutation sites:
    - The letter assigned to c.
    - The direction assigned to c.

When randomly choosing the value for tag_0, choosing 0 has a weight of 4, while choosing 1 has a weight of 1. When randomly choosing the value for tag_1, choosing 0 has a weight of 4; while choosing 1, 2, or 3 each have a weight of 1.

When randomly choosing a letter from $\Gamma \cup \{\epsilon\}$, the letter is chosen with a uniform distribution. When choosing a direction, choosing 0 has a weight of 5; choosing -1 has a weight of 2; choosing 1 has a weight of 2; and choosing 2 has a weight of 1.

The weights described above are designed to provide preference to "inert" tags ($\text{tag}_0 = 0$, $\text{tag}_1 = 0$) or directions (the head remains stationary).

7.3 Advantages of RTGP_2 over RTGP_1

I will now detail advantages RTGP_2 possesses over RTGP_1. I have already noted that any mutation in RTGP_1 is likely to dramatically alter the phenotype of an agent to the point that selection in favor of fitter agents is useless. To establish that RTGP_2 does not have this same pitfall, I will show that the genotypes in RTGP_2 are a more explicit representation of the corresponding phenotype. A real-time deciding program for a language with character set Σ can be pictured as an infinite decision tree where each node has a 0 or 1 label and $|\Sigma|$ children (not counting the parent node). An input string is processed by starting with a reference to the root node. As each character c is being processed, the reference is moved to the child of the current node indexed by c. After the input string has been processed, the output is the label of the node currently being referenced. This "infinite decision tree" style of representation is one of the most explicit ways the phenotype can be represented. Moreover a change to one of the labels only changes the output for one string, so the phenotype is minimally changed. As an infinite decision tree cannot be stored as a genotype, it must be "coiled up" into a more compact form such as the genotypes used in the RTGP_2 paradigm. This is the main reason why mutations in the RTGP_2 paradigm do not completely scramble the phenotype.

More specifically, changing the label of one of the nodes in RTGP_2 will at most change the output for some inputs but not completely scramble the phenotype. Changing one of the edges; edge labels; or tags of a node s will partially scramble the output for words w for which node s is used while w is being processed. The output for words w that do not use state s during processing are unchanged. If a word w accesses state s during processing, patterns in the output that occur as characters are appended to the end of w may be partially conserved by a mutation to s.

Referring to the state transition graph of the program, mutations in the RTGP_1 paradigm dramatically alter the state transition graph, while mutations in RTGP_2 only mutate part of the state transition graph. Hence the phenotype is better preserved by mutations in RTGP_2 than by mutations in RTGP_1.

7.4 Example deciding programs using RTGP_2

In this section, just as with RTGP_1, I will give some example deciding programs to better illustrate how the RTGP_2 paradigm works. The languages for which I will provide examples are TL_2 and CFL_2. These programs will be engineered to give the reader an idea as to how programs that decide a specific language can be constructed in the RTGP_2 paradigm.

The following sections will give depictions of the programs as directed graphs. The label (L), the tag_0 (T_0) and, the tag_1 (T_1) of each node is shown. Irrelevant edges are not shown. Primary child edges are denoted by $a, < l, d >$ where $a \in \Sigma$ is the index of the edge, $l \in \Gamma \cup \{\epsilon\}$ is the letter, and $d \in \{-1, 0, 1, 2\}$ is the direction. Secondary child edges are denoted by $[b]$ where $b \in \Gamma \cup \{\epsilon\}$ is the edge index. For simplicity, directed edges between the same pair of nodes are depicted as a single edge with multiple labels.

7.4.1 Example deciding program for TL_2 using RTGP_2

Recall that $\text{TL}_2 = (ba)^*$, and $\Sigma = \{a, b\} = \{0, 1\}$. A program that decides TL_2 is shown in figure 7.2.

All nodes have both tags set to 0 as the Turing tape has no relevance for a finite state automaton. The $< l, d >$ pairs also have no effect and are defaulted to $< \epsilon, 0 >$.

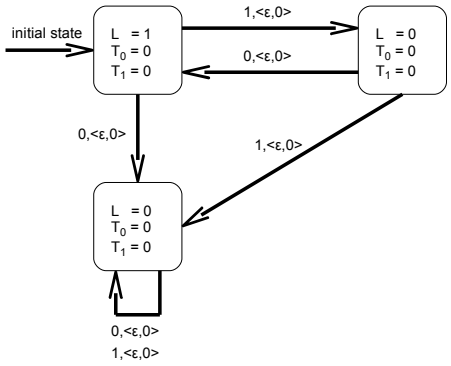


Figure 7.2: Example decider for TL_2 in the $RTGP_2$ paradigm.

7.4.2 Example deciding program for CFL_2 using $RTGP_2$

Recall that $CFL_2 = \{s \in \{\text{“(”, “)”}\}^* | \text{parentheses are balanced}\}$, and $\Sigma = \{\text{“(”, “)”}\} = \{0, 1\}$. I will use the alphabet: $\Gamma = \{0\}$. A program that decides CFL_2 is shown in figure 7.3.

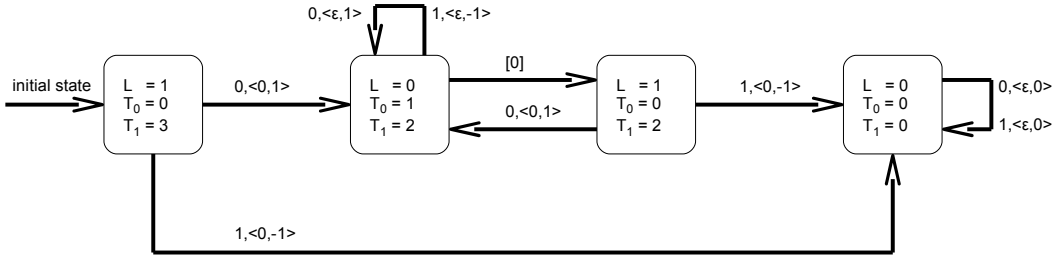


Figure 7.3: Example decider for CFL_2 in the $RTGP_2$ paradigm.

This program in figure 7.3 marks the initial square of the Turing tape with the letter “0” (ϵ is the blank symbol). The number of squares to the right that the tape head is located relative the initial square denotes the parenthesis level. If the tape head ever moves to the left of the initial square, the string is invalid.

Chapter 8

Results for RTGP_2

For my tests, I will use the parameters:

- $|\Gamma| = 1$ is the size of the alphabet of symbols that the Turing tape can use.
- $M = 500$ is the size of the population.
- $p_{\text{tournament}} = 0.9$ is the probability of selecting the better agent during tournament selection.
- max_mutations $= 2$ is the maximum number of basic mutation steps during mutation.
- $\mu_\theta = 0.1$ is the probability of mutating any given θ during mutation.
- $\alpha = 0.1$ is the probability of reseting the self adaptive parameters during mutation.
- $\beta = 0.1$ is the probability of ignoring the self adaptive parameters during mutation.
- $N = 10$ is the size of the array of nodes.

8.1 TL_1 using RTGP_2

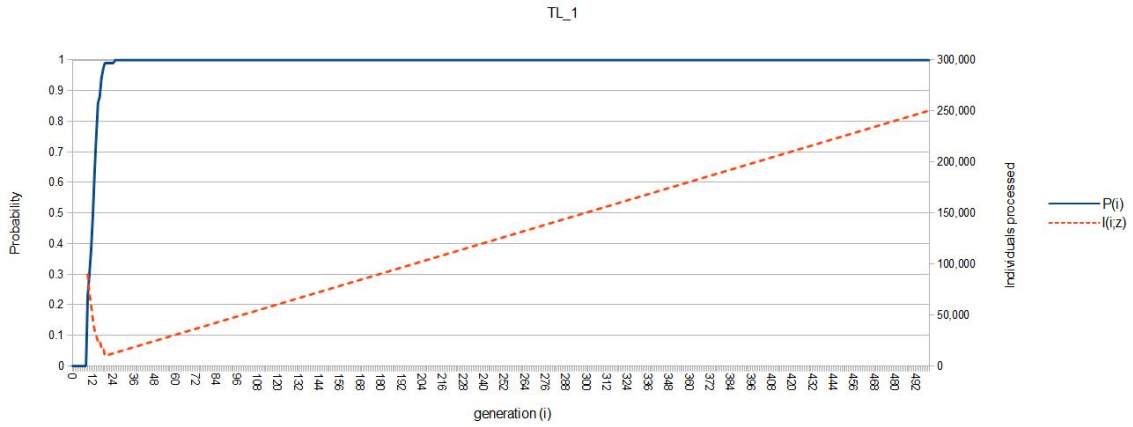
$\text{TL}_1 = b^* = 1^*$

For each generation a set U of $T = 1000$ test strings are randomly generated using the algorithm described 2.4.1. The exact algorithm used to generate the test cases can be found in A.1. However the maximum number of "0"s or "1"s is increased to 50 from 20.

The termination condition still occurs when the prime agent has never had a fitness other than 0, and has been the prime agent for 10 generations.

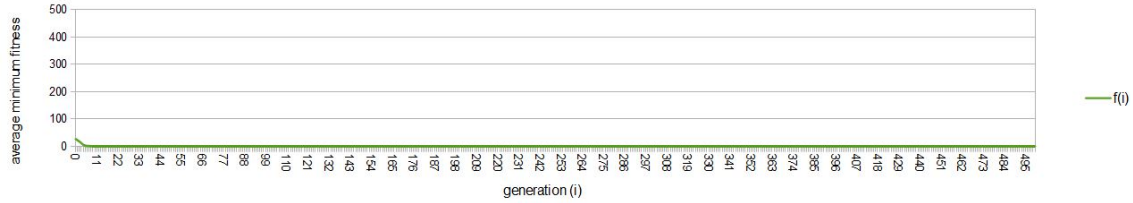
The following graphs use a sample of 100 runs with a generation limit of 500.

Below is a graph displaying $P(i)$ (solid curve) and $I(i; 99\%)$ (dashed curve) vs generation i for TL_1:



The gap before the start of the $I(i;z)$ curve indicates that the required effort is "infinite": a solution was not found before generation i.

The next graph shows the average minimum fitness $f(i)$ vs generation i:



8.2 TL_2 using $RTGP_2$

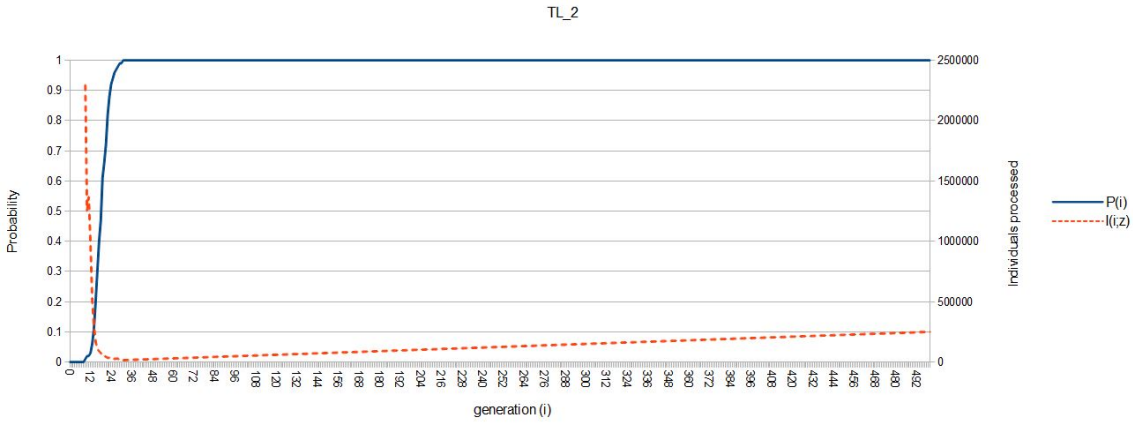
$TL_2 = (ba)^* = (10)^*$

For each generation a set U of $T = 1000$ test strings are randomly generated using the algorithm described 2.4.1. The exact algorithm used to generate the test cases can be found in A.2. However, the maximum word length is increased to 50 from 20.

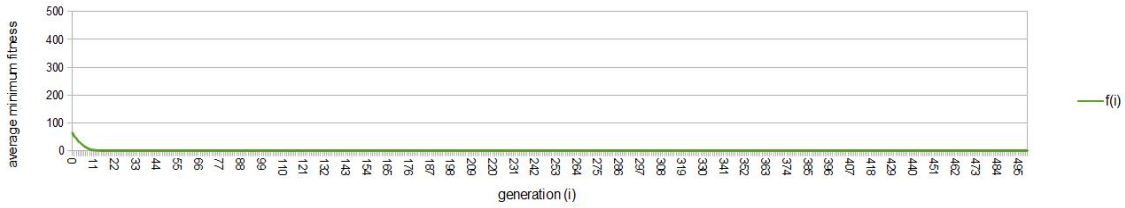
The termination condition still occurs when the prime agent has never had a fitness other than 0, and has been the prime agent for 10 generations.

The following graphs use a sample of 100 runs with a generation limit of 500.

Below is a graph displaying $P(i)$ (solid curve) and $I(i; 99\%)$ (dashed curve) vs generation i for TL_2:



The next graph shows the average minimum fitness $f(i)$ vs generation i:



8.3 TL_3 using $RTGP_2$

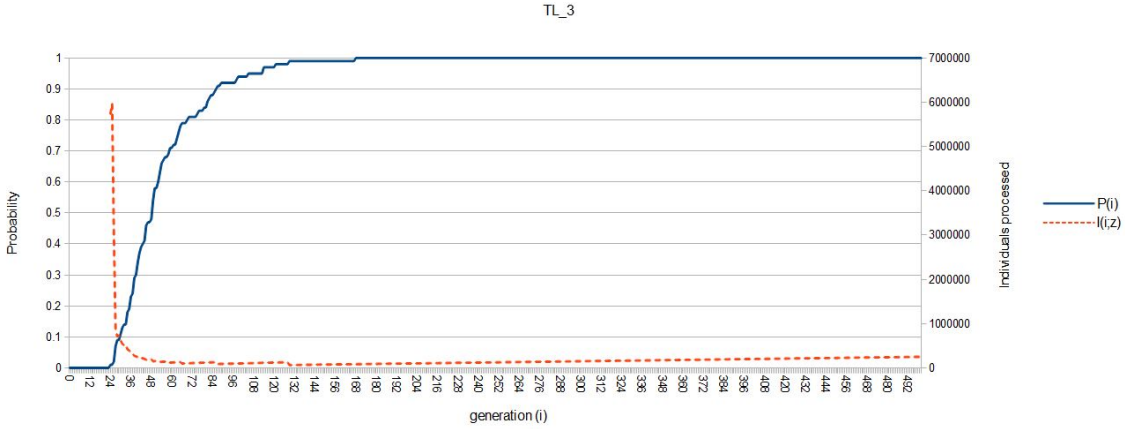
$TL_3 = \{s \in \{0,1\}^* | \text{no odd length block of 0's can follow an odd length block of 1's}\}$

For each generation a set U of $T = 1000$ test strings are randomly generated using the algorithm described 2.4.1. The exact algorithm used to generate the test cases can be found in A.3.

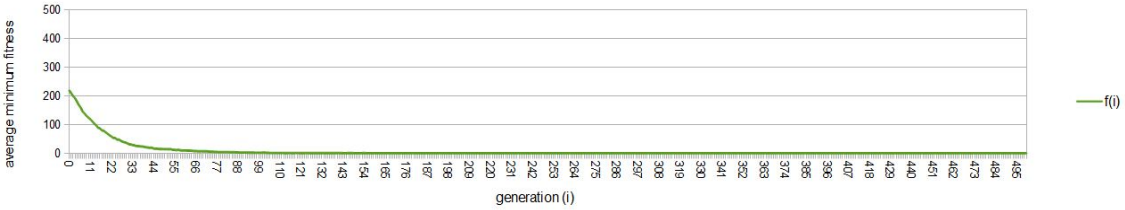
The termination condition still occurs when the prime agent has never had a fitness other than 0, and has been the prime agent for 10 generations.

The following graphs use a sample of 100 runs with a generation limit of 500.

Below is a graph displaying $P(i)$ (solid curve) and $I(i; 99\%)$ (dashed curve) vs generation i for TL_3:



The next graph shows the average minimum fitness $f(i)$ vs generation i:



As can be seen, $RTGP_2$ has a much greater success with TL_3 than $RTGP_1$. I will continue with the rest of the languages.

8.4 TL_4 using $RTGP_2$

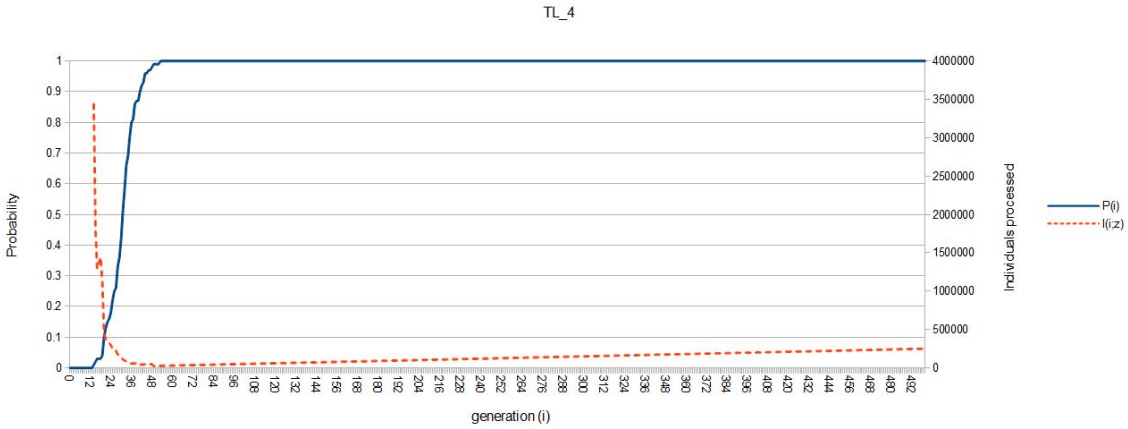
$TL_4 = \{s \in \{0,1\}^* | \text{no 000 substrings}\}$

For each generation a set U of $T = 1000$ test strings are randomly generated using the algorithm described 2.4.1. The exact algorithm used to generate the test cases can be found in A.4.

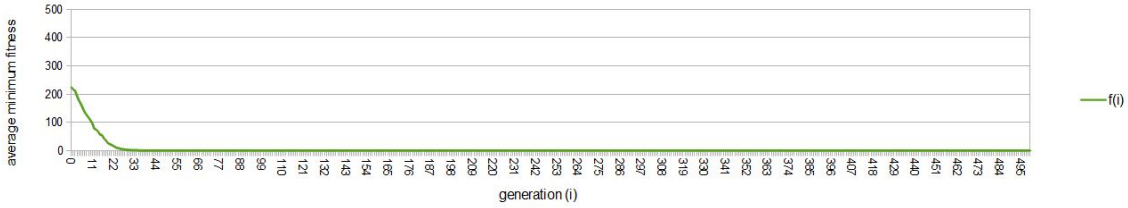
The termination condition occurs when the prime agent has never had a fitness other than 0, and has been the prime agent for 10 generations.

The following graphs use a sample of 100 runs with a generation limit of 500.

Below is a graph displaying $P(i)$ (solid curve) and $I(i; 99\%)$ (dashed curve) vs generation i for TL_4:



The next graph shows the average minimum fitness $f(i)$ vs generation i:



8.5 TL_5 using RTGP_2

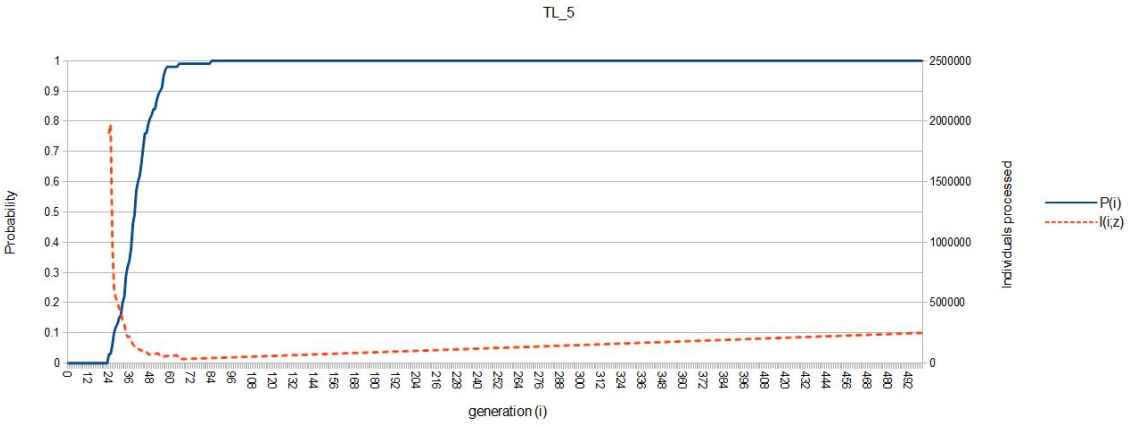
$\mathrm{TL}_5 = \{\text{“”}, 0, 1\} \cup 0\{0,1\}^*0 \cup 1\{0,1\}^*1$

For each generation a set U of $T = 1000$ test strings are randomly generated using the algorithm described 2.4.1. The exact algorithm used to generate the test cases can be found in A.5.

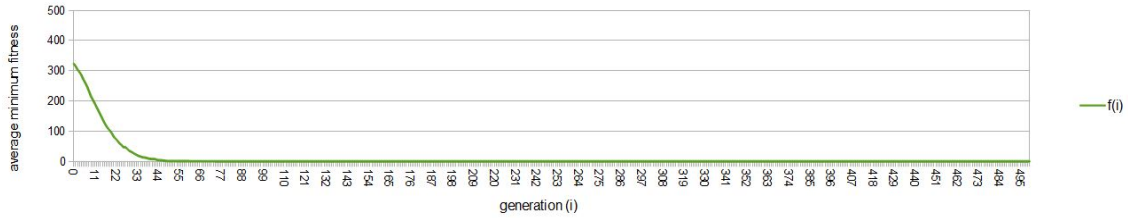
The termination condition occurs when the prime agent has never had a fitness other than 0, and has been the prime agent for 10 generations.

The following graphs use a sample of 100 runs with a generation limit of 500.

Below is a graph displaying $P(i)$ (solid curve) and $I(i; 99\%)$ (dashed curve) vs generation i for TL_5:



The next graph shows the average minimum fitness $f(i)$ vs generation i:



8.6 TL_6 using $RTGP_2$

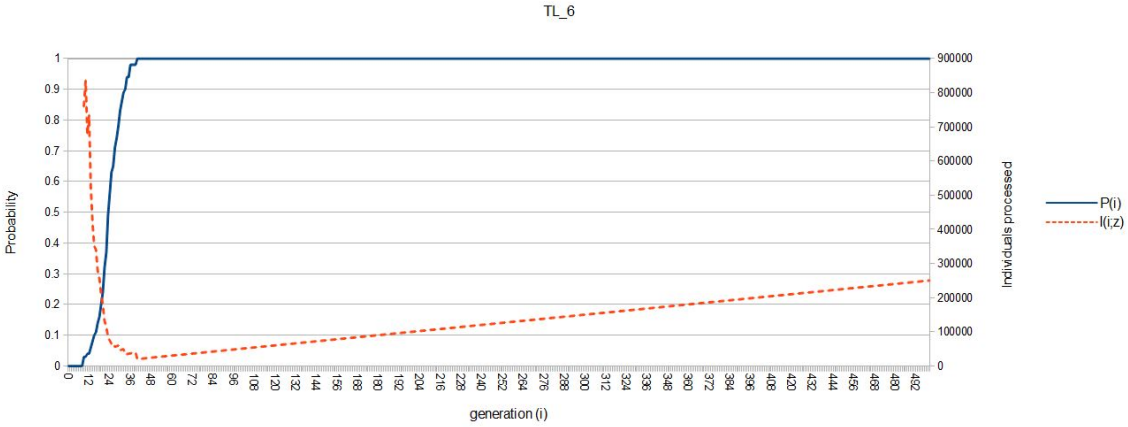
$TL_6 = \{s \in \{0,1\}^* | \#\text{ of 0's - \# of 1's} \equiv 0 \mod 3\}$

For each generation a set U of $T = 1000$ test strings are randomly generated using the algorithm described 2.4.1. The exact algorithm used to generate the test cases can be found in A.6.

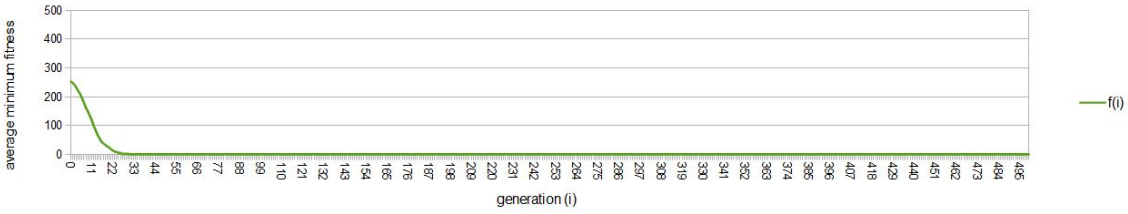
The termination condition occurs when the prime agent has never had a fitness other than 0, and has been the prime agent for 10 generations.

The following graphs use a sample of 100 runs with a generation limit of 500.

Below is a graph displaying $P(i)$ (solid curve) and $I(i; 99\%)$ (dashed curve) vs generation i for TL_6:



The next graph shows the average minimum fitness $f(i)$ vs generation i:



8.6.1 An evolved decider for TL_6

Below is an example of an evolved decider for TL_6:

Each node is denoted by:

$$\text{index} : (\text{label}, \text{tag}_0, \text{tag}_1, [\text{primary\_child}_0, \text{letter}_0, \text{direction}_0], [\text{primary\_child}_1, \text{letter}_1, \text{direction}_1], \text{secondary\_child}_1)$$

With regards to the tape alphabet, the blank character is "0" while $\Gamma = \{1\}$.

The boldface indexes indicate which nodes are active in the program.

Root Index: 0

Nodes:

9: (0, 0, 0, [7,1,0], [5,0,0], 2)
8: (0, 0, 0, [3,1,0], [0,0,0], 0)
7: (0, 0, 0, [9,0,0], [2,1,0], 2)
6: (0, 0, 3, [0,0,0], [8,0,1], 5)
5: (0, 1, 2, [8,0,0], [9,0,1], 6)
4: (0, 0, 3, [8,0,0], [4,0,0], 4)
3: (0, 1, 0, [0,1,0], [8,1,1], 8)
2: (1, 1, 0, [8,1,1], [9,0,-1], 5)
1: (1, 0, 0, [6,0,0], [9,0,1], 8)
0: (1, 0, 0, [8,1,0], [6,1,0], 7)

8.7 TL_7 using $RTGP_2$

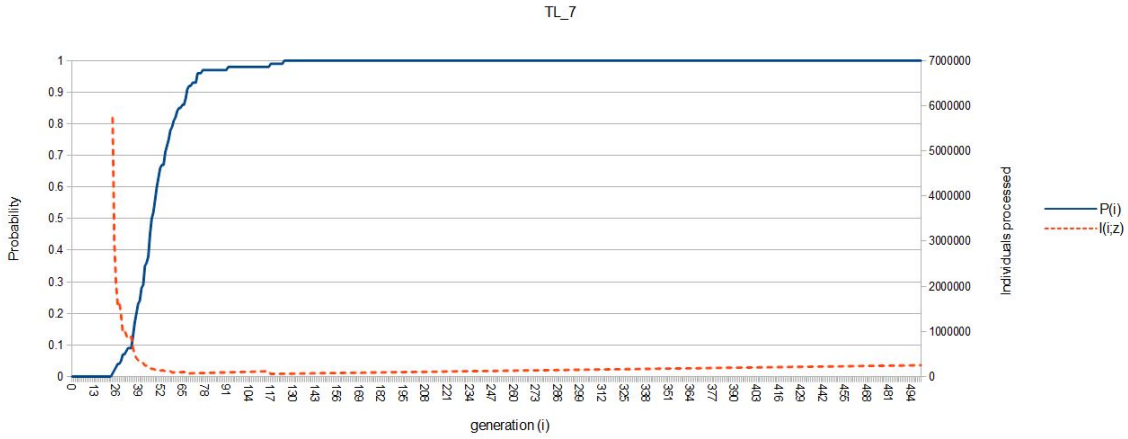
$TL_7 = 0^*1^*0^*1^*$

For each generation a set U of $T = 1000$ test strings are randomly generated using the algorithm described 2.4.1. The exact algorithm used to generate the test cases can be found in A.7.

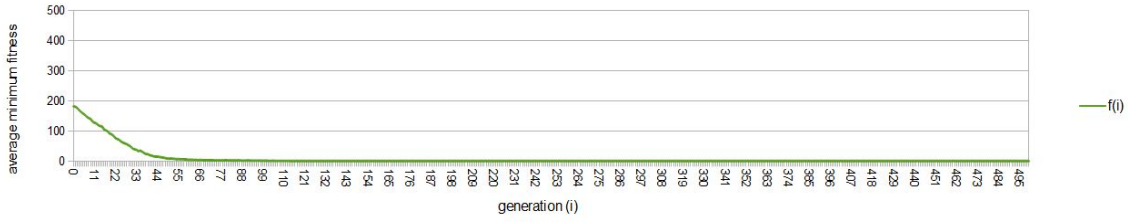
The termination condition occurs when the prime agent has never had a fitness other than 0, and has been the prime agent for 10 generations.

The following graphs use a sample of 100 runs with a generation limit of 500.

Below is a graph displaying $P(i)$ (solid curve) and $I(i; 99\%)$ (dashed curve) vs generation i for TL_7:



The next graph shows the average minimum fitness $f(i)$ vs generation i:



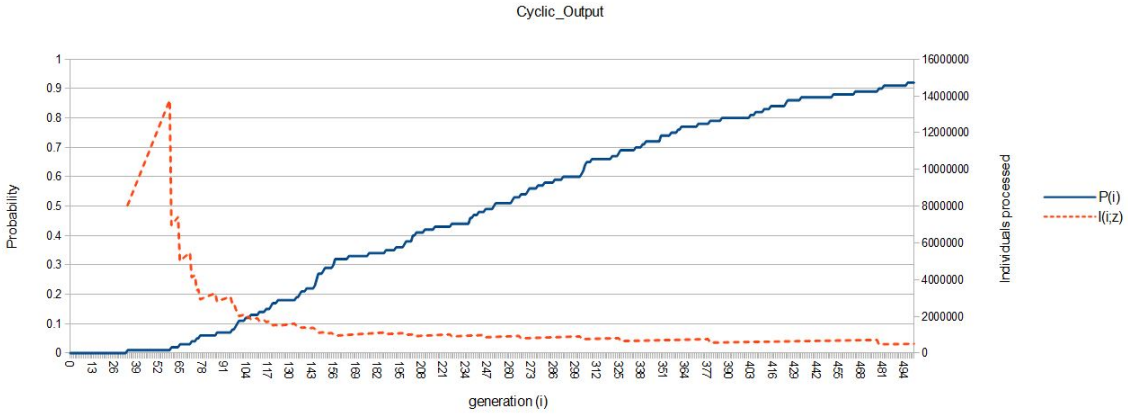
8.8 Cyclic output using $RTGP_2$

I will now repeat the cyclic output test that I gave to $RTGP_1$. To show how successful $RTGP_2$ is compared to $RTGP_1$, I will attempt to evolve a decider where the output follows a cycle with a period of 20 as opposed to 3. The alphabet of symbols will simply be: $\Sigma = \{0\}$, and the specific language will be Complex_Cyclic_Output $= \{0^n | n \equiv 10, 11, \ldots, 19 \mod 20\}$. The set of test strings is $U = \{0^i | 0 \leq i \leq 1000\}$. Due to the fitness being computed in a deterministic manner (no cases are generated randomly), the termination condition is when the prime agent has a fitness of 0.

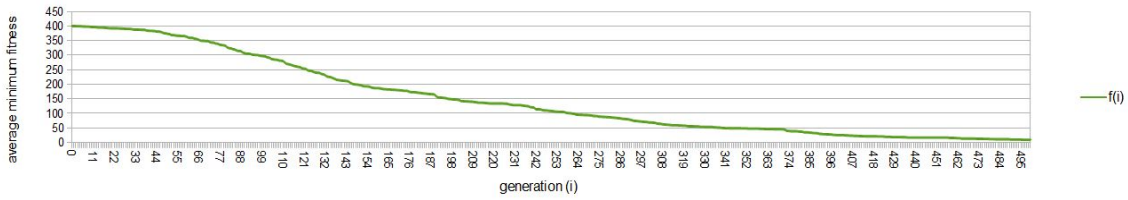
An optimal decider requires at least 20 distinct states so each agent has $N = 30$ nodes as opposed to 10.

The following graph uses a sample of 100 runs with a generation limit of 500.

Below is a graph displaying $P(i)$ (solid curve) and $I(i; 99\%)$ (dashed curve) vs generation i for "Complex_Cyclic_Output":



The next graph shows the average minimum fitness $f(i)$ vs generation i for "Complex_Cyclic_Output":



8.8.1 An evolved decider for "Complex_Cyclic_Output"

Below is an example of an evolved decider for "Complex_Cyclic_Output":

Each node is denoted by:

$$\text{index} : (\text{label}, \text{tag}_0, \text{tag}_1, [\text{primary\_child}_0, \text{letter}_0, \text{direction}_0], \text{secondary\_child}_1)$$

With regards to the tape alphabet, the blank character is "0" while $\Gamma = \{1\}$.

The boldface indexes indicate which nodes are active in the program.

Root Index: 24

Nodes:

29: (1, 0, 2, [2,0,-1], 3)
28: (1, 0, 1, [27,0,-1], 17)
27: (1, 1, 2, [21,0,-1], 4)

26: (1, 0, 0, [14,1,-1], 24)
25: (1, 1, 3, [24,0,2], 14)
24: (0, 0, 0, [15,1,0], 21)
23: (0, 1, 0, [8,1,0], 5)
22: (0, 0, 0, [19,1,1], 22)
21: (1, 0, 3, [8,1,2], 4)
20: (1, 1, 3, [6,1,1], 15)
19: (0, 0, 2, [23,1,1], 1)
18: (1, 1, 1, [20,0,0], 19)
17: (0, 0, 3, [18,1,0], 6)
16: (0, 0, 0, [1,0,0], 8)
15: (0, 0, 3, [17,1,-1], 19)
14: (1, 0, 0, [29,0,0], 6)
13: (0, 1, 3, [19,1,0], 23)
12: (1, 0, 0, [14,1,0], 12)
11: (0, 0, 0, [20,1,0], 29)
10: (1, 0, 0, [9,1,0], 1)
09: (0, 1, 0, [27,1,-1], 2)
08: (0, 1, 0, [16,0,1], 25)
7: (1, 0, 0, [8,0,1], 4)
06: (1, 1, 0, [26,0,2], 5)
05: (0, 0, 0, [19,0,0], 21)
04: (0, 1, 0, [13,1,0], 25)
03: (1, 0, 0, [6,1,0], 12)
02: (0, 1, 0, [29,1,0], 28)
01: (0, 0, 0, [18,1,0], 17)
00: (0, 0, 2, [9,0,0], 25)

To understand how this solution works, the following table shows the progression of the states as successive 0's are processed:

For the Turing tape, the square brackets surround the character under the tape head, while the underlined character is the initial position of the tape head.

Node	Turing tape	proxy node (if $\text{tag}_0 = 1$)	label (proxy node label if $\text{tag}_0 = 1$)
24	[$\underline{0}$]		0
15	[$\underline{0}$]		0
17	[0]$\underline{1}$		0
18	[1]$\underline{1}$	19	0
23	1[$\underline{1}$]	05	0
19	1[$\underline{1}$]		0
23	1$\underline{1}$[0]	23	0
08	1$\underline{1}$[0]	08	0
16	1$\underline{1}$[0]		0
01	1$\underline{1}$[0]		0
18	1$\underline{1}$[0]	18	1
20	1$\underline{1}$[0]	20	1
06	1$\underline{1}$1[0]	06	1
26	1$\underline{1}$1[0]		1
14	1$\underline{1}$1[0]		1
29	1$\underline{1}$1[0]		1
02	1$\underline{1}$[1]0	28	1
27	1$\underline{1}$[0]0	27	1

21	1[1]00		1
08	1[1]00	25	1
24	1[0]00		0
15	1[0]00		0
17	[1]100		0
18	[1]100	19	0
....	...	...	...

As can be seen, RTGP_2 can evolve a decider for language with a period of 20, while RTGP_1 can't even evolve a decider for a language with a period of 3.

In the following sections I now investigate the non-regular languages.

8.9 CFL_1 using RTGP_2

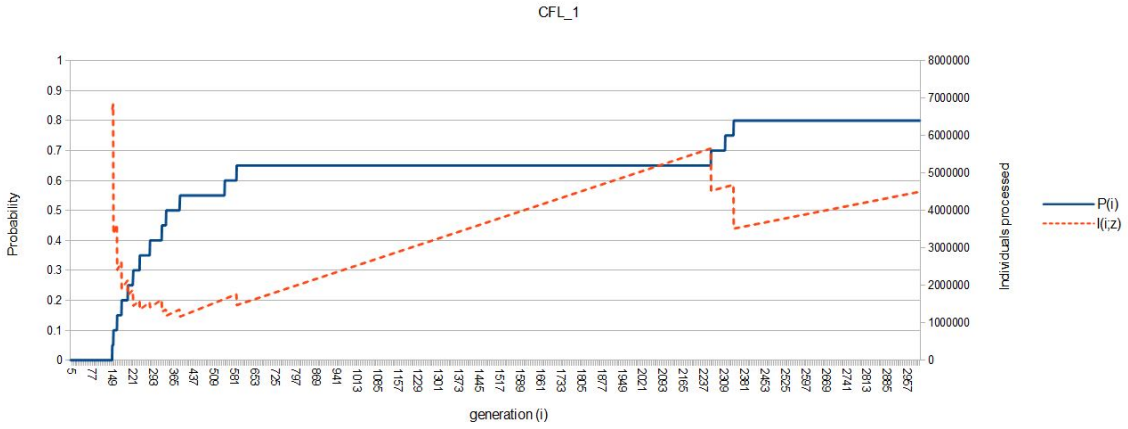
$\text{CFL}_1 = \{0^n 1^n | n \geq 0\}$

The method of computing the fitness of each agent A is given in B.1.

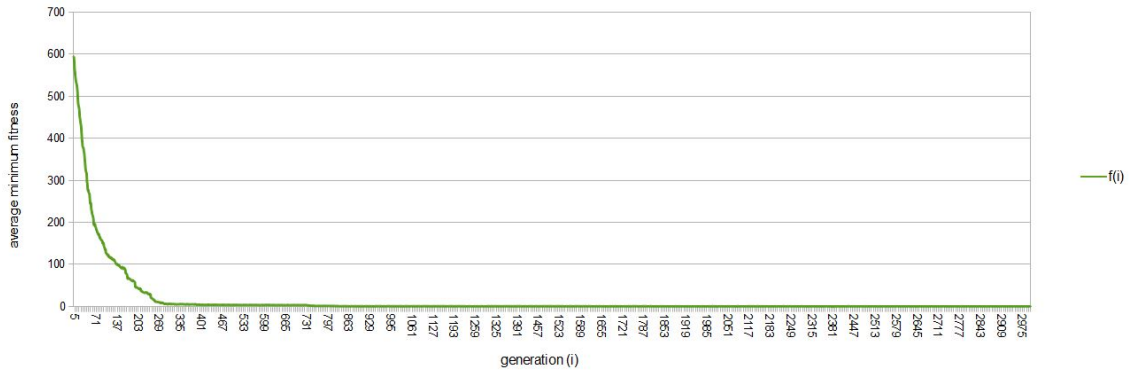
The termination condition occurs when the prime agent has never had a fitness other than 0, and has been the prime agent for 50 generations. I required the prime agent to perfect for a larger number (50) of generations due to the fact that there are many agents that are "near perfect", meaning that they have perfect fitness but are still incorrect for some inputs.

The following graphs use a sample of 20 runs with a generation limit of 3000. Since the number of data points/runs is much smaller (20 vs 100) than what was used for the regular languages, the graph below looks much more blocky than its regular language counter parts.

Below is a graph displaying $P(i)$ (solid curve) and $I(i; 99\%)$ (dashed curve) vs generation i for CFL_1:



The next graph shows the average minimum fitness $f(i)$ vs generation i:



8.10 CFL_2 using RTGP_2

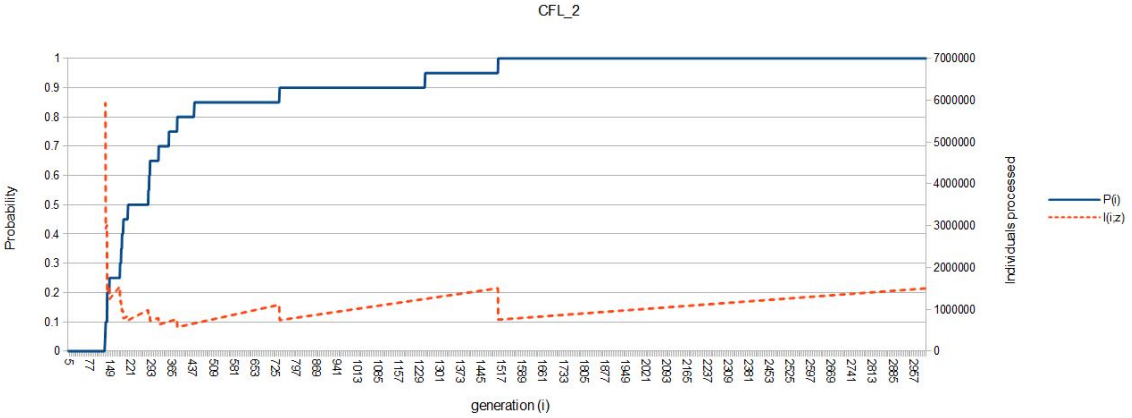
$\mathrm{CFL}_2 = \{s \in \{0,1\}^* | \text{parentheses are balanced where } 0 = \text{“(” and } 1 = \text{“)”}\}$

The method of computing the fitness of each agent A is given in B.2.

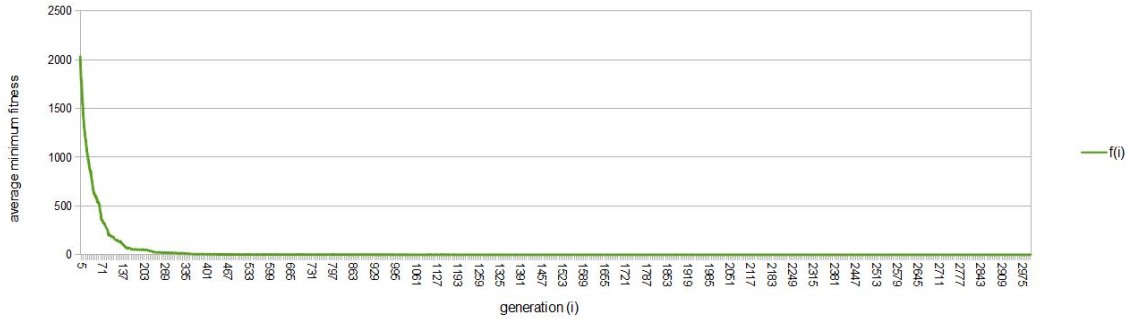
The termination condition occurs when the prime agent has never had a fitness other than 0, and has been the prime agent for 50 generations.

The following graphs use a sample of 20 runs with a generation limit of 3000. Since the number of data points/runs is much smaller (20 vs 100) than what was used for the regular languages, the graph below looks much more blocky than its regular language counter parts.

Below is a graph displaying $P(i)$ (solid curve) and $I(i; 99\%)$ (dashed curve) vs generation i for CFL_2:



The next graph shows the average minimum fitness $f(i)$ vs generation i:



8.10.1 Evolved decider for CFL_2

Below is an example of an evolved decider for CFL_2:

Each node is denoted by:

$$\text{index} : (\text{label}, \text{tag}_0, \text{tag}_1, [\text{primary\_child}_0, \text{letter}_0, \text{direction}_0], \\ [\text{primary\_child}_1, \text{letter}_1, \text{direction}_1], \text{secondary\_child}_1)$$

With regards to the tape alphabet, the blank character is "0" while $\Gamma = \{1\}$.

The boldface indexes indicate which nodes are active in the program.

Root Index: 5

Nodes:

9: (1, 1, 3, [6,1,2], [9,0,0], 1)
8: (0, 0, 3, [2,0,0], [0,0,0], 4)
7: (0, 0, 0, [7,1,0], [7,1,0], 9)
6: (0, 1, 1, [0,1,0], [5,1,-1], 1)
5: (1, 0, 0, [6,1,1], [7,0,-1], 5)
4: (0, 0, 3, [4,1,1], [2,0,-1], 2)
3: (1, 0, 0, [6,1,-1], [4,1,0], 5)
2: (0, 1, 1, [4,1,0], [8,0,1], 4)
1: (0, 0, 0, [0,1,1], [5,1,-1], 5)
0: (0, 0, 1, [8,0,0], [1,1,1], 4)

7 is the sink state.

Imagine that the string 0^n1^n is being processed.

As the n "0"s are processed, the sequence of states (not showing the proxy nodes) is:

$$5 \to 6 \to 0 \to 8 \to 2\underbrace{\to 4 \to 4 \cdots \to 4}_{n-4}$$

As the n "1"s are processed, the sequence of states (not showing the proxy nodes) is:

$$4\underbrace{\to 2 \to 2 \cdots \to 2}_{n-4} \to 8 \to 0 \to 1 \to 5$$

8.11 CFL_3 using RTGP_2

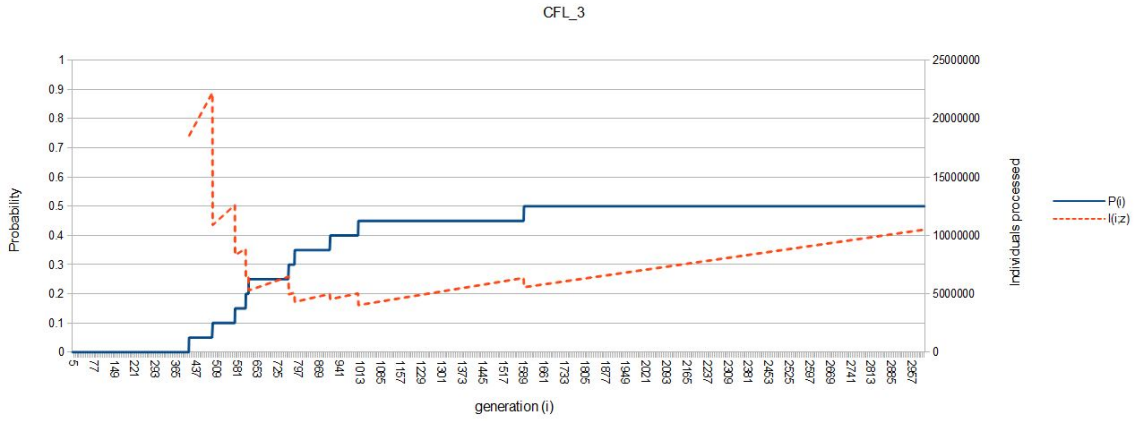
$\text{CFL}_3 = \{s \in \{0,1\}^* | \#\text{ of 0's} = \#\text{ of 1's}\}$

The method of computing the fitness of each agent A is given in B.3.

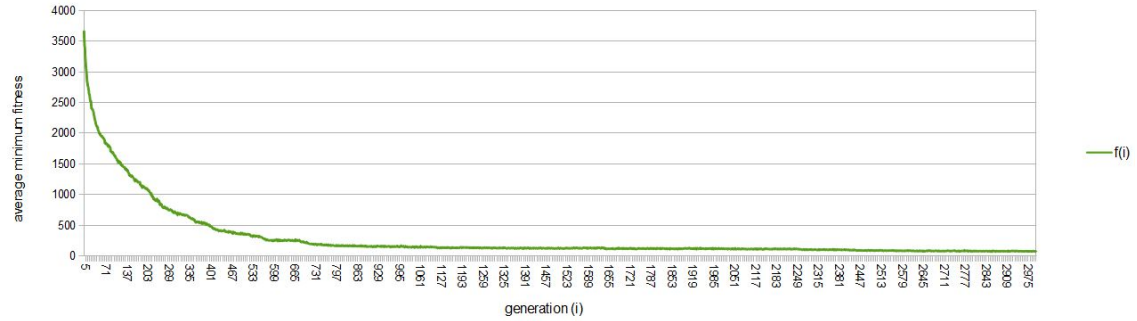
The termination condition occurs when the prime agent has never had a fitness other than 0, and has been the prime agent for 50 generations.

The following graphs use a sample of 20 runs with a generation limit of 3000. Since the number of data points/runs is much smaller (20 vs 100) than what was used for the regular languages, the graph below looks much more blocky than its regular language counter parts.

Below is a graph displaying $P(i)$ (solid curve) and $I(i; 99\%)$ (dashed curve) vs generation i for CFL_3:



The next graph shows the average minimum fitness $f(i)$ vs generation i:



8.12 The final tests

The previous tests focused on regular languages and context free languages. In the following tests I will focus on three languages that are neither regular nor context free.

8.12.1 NCFL using RTGP_2

$\text{NCFL} = \{a^n b^n a^n | n \geq 0\}$

For this test, each agent has $N = 15$ nodes as opposed to 10 nodes due to the fact that an optimal decider for NCFL in RTGP_2 paradigm requires approximately 7 nodes.

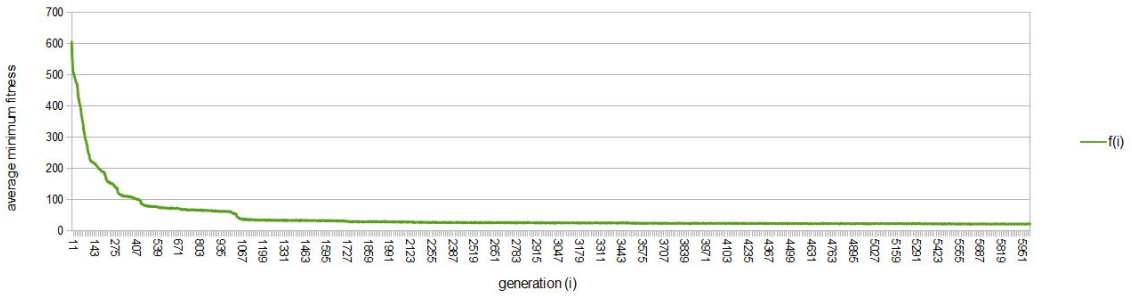
The method of computing the fitness of each agent A is given in B.4.

The termination condition occurs when the prime agent has never had a fitness other than 0, and has been the prime agent for 100 generations.

My data uses a sample of 20 runs with a generation limit of 6000.

Only 3 of the 20 runs found a decider for NCFL.

The following graph shows the average minimum fitness $f(i)$ vs generation i:



8.12.2 Non cyclic output using RTGP$_2$

The second non context free language I will consider uses only a single character: $\Sigma = \{0\}$. The only possible inputs are strings of various lengths. The sequence of outputs as the string length increases from the empty string is:

$$1101001000100001000001000000100000001\ldots$$

This language I will refer to as NCO (non-cyclic output). More precisely,

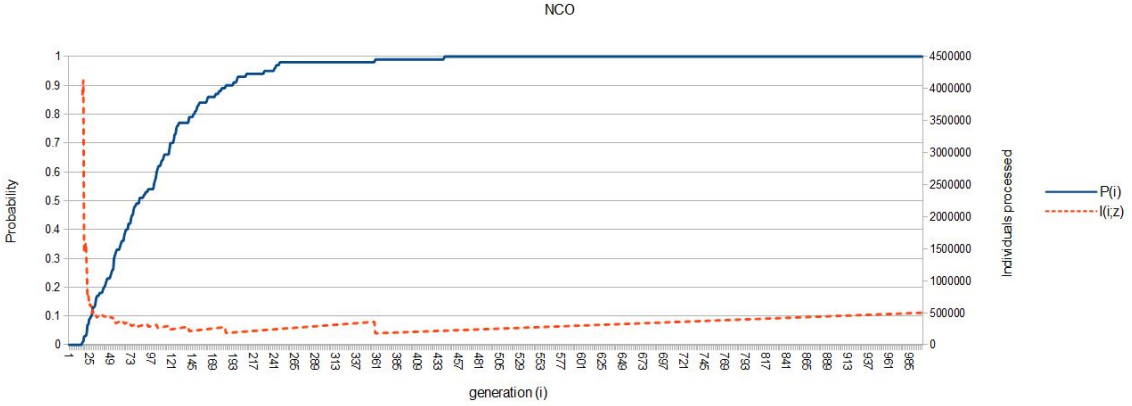
$$\text{NCO} = \{0^i : i = \sum_{j=0}^{n} j \quad \text{for some } n \geq 0\}$$

The method of computing the fitness of each agent A is given in B.5.

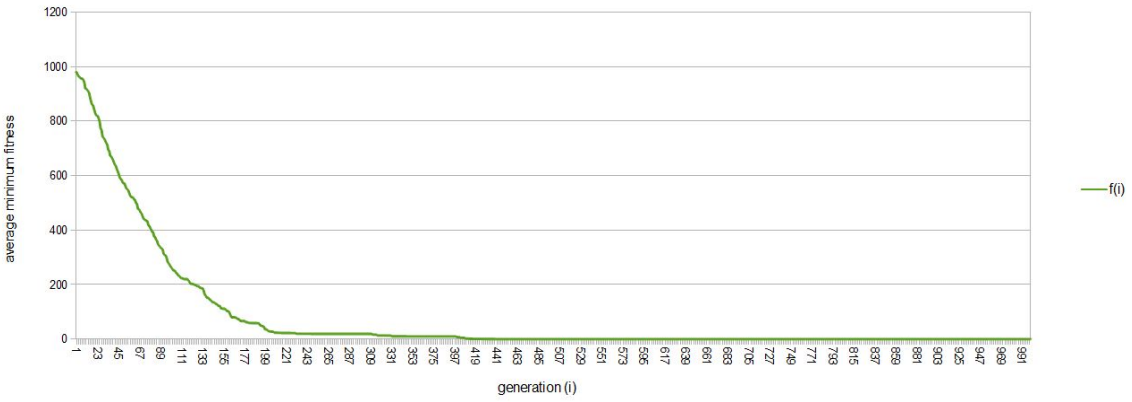
The termination condition occurs when the prime agent has a fitness of 0. No successive generations with the prime agent having a fitness of 0 is required as the algorithm for measuring fitness does not depend on any randomly generated test cases.

The following graphs use a sample of 100 runs with a generation limit of 1000.

Below is a graph displaying $P(i)$ (solid curve) and $I(i; 99\%)$ (dashed curve) vs generation i for NCO:



The next graph shows the average minimum fitness $f(i)$ vs generation i:



Evolved decider for NCO

Below is an example of an evolved decider for NCO:

Each node is denoted by:

$$\text{index} : (\text{label}, \text{tag}_0, \text{tag}_1, [\text{primary\_child}_0, \text{letter}_0, \text{direction}_0], \text{secondary\_child}_1)$$

With regards to the tape alphabet, the blank character is "0" while $\Gamma = \{1\}$.

The boldface indexes indicate which nodes are active in the program.

Root Index: 9

Nodes:

9: (1, 0, 0, [6,0,-1], 0)
8: (0, 0, 2, [5,1,-1], 4)
7: (0, 0, 0, [8,1,1], 8)
6: (1, 0, 0, [4,0,0], 3)
5: (1, 1, 0, [3,1,0], 8)
4: (0, 1, 1, [5,0,-1], 8)
3: (0, 0, 3, [5,1,2], 2)
2: (0, 0, 0, [4,0,1], 0)
1: (1, 1, 0, [5,1,0], 1)
0: (0, 0, 0, [9,1,1], 0)

To understand how this solution works, the following table shows the progression of the states as successive 0's are processed:

For the Turing tape, the square brackets surround the character under the tape head, while the underlined character is the initial position of the tape head.

Node	Turing tape	proxy node (if $\text{tag}_0 = 1$)	label (proxy node label if $\text{tag}_0 = 1$)
9	[0]		1
6	[0]		1
4	[0]	4	0
5	[0]	5	1
3	[0]		0
5	[1]	8	0

5	[0]$\underline{1}$	5	1
3	[0]$\underline{1}$		0
5	1[$\underline{1}$]	8	0
5	[1]$\underline{1}$	8	0
5	[0]1$\underline{1}$	5	1
3	[0]1$\underline{1}$		0
5	11[$\underline{1}$]	8	0
5	1[1]$\underline{1}$	8	0
5	[1]1$\underline{1}$	8	0
5	[0]11$\underline{1}$	5	1
...	...	...	...

8.12.3 Finding triangles in a graph using RTGP$_2$

In this final test of RTGP$_2$, I will attempt to find a real time algorithm that can determine if an arbitrary simple graph has a 3-cycle/triangle sub-graph. A brute force algorithm for determining if a triangle is present runs in $O(n^3)$ steps where n is the number of verticies in the graph.

Let E denote the "edge matrix", an $n \times n$ matrix where $E_{i,j} = 1$ if $\{i, j\}$ is an edge and $E_{i,j} = 0$ if $\{i, j\}$ is not an edge. The (i, j) entry of E^2 is nonzero if a walk of length 2 exists between verticies i and j and zero if otherwise. A triangle exists iff there exists an (i, j) pair such that the (i, j) entry of both E and E^2 is nonzero.

There exists multiple "fast matrix multiplication" algorithms for computing E^2. Ordinary matrix multiplication requires $O(n^3)$ operations. "Strassen's algorithm" for matrix multiplication requires $O(n^{\log_2 7}) \approx O(n^{2.807})$ steps. A description of Strassen's algorithm can be found in [27]. The "Coppersmith-Winograd algorithm" for matrix multiplication requires $O(n^{2.376})$ steps [8].

Both Strassen and the Coppersmith-Winograd algorithm respectively yield an $O(n^{2.807})$ step and a $O(n^{2.376})$ step algorithm for determining if an n-vertex graph has a triangle.

For this last test, an arbitrary graph will be denoted by the following paradigm. A graph of n verticies will be denoted by a binary string with length at most: $1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$. In the following binary string, $e_{i,j} = 1$ iff an edge exists between verticies i and j:

$$e_{2,1}e_{3,1}e_{3,2}e_{4,1}e_{4,2}e_{4,3}e_{5,1}e_{5,2}e_{5,3}e_{5,4}\ldots e_{n,1}e_{n,2}\ldots e_{n,n-1}$$

If the string is too short, $e_{i,j} = 0$ by default.

If RTGP$_2$ does find a real time algorithm that can detect triangle subgraphs in the input graph, then the algorithm will run in $O(n^2)$ steps for a graph with n verticies.

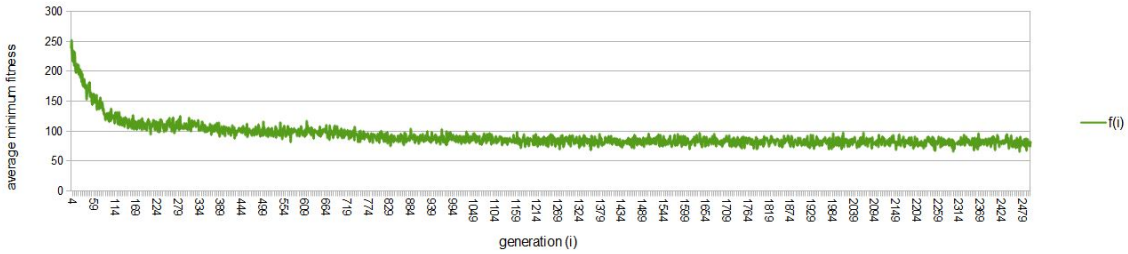
For this test, each agent has $N = 30$ nodes as opposed to 10 nodes to allow for greater complexity in the structure of the agents.

To improve the expressive power of RTGP$_2$ the number of non-blank symbols available to the Turing tape was increased to $|\Gamma| = 3$.

Unlike all of the previous non-regular languages, the fitness will simply be the number test cases failed. For each generation a set U of $T = 1000$ test strings are randomly generated using the algorithm described 2.4.1. The exact algorithm used to generate the test cases can be found in A.8.

The termination condition occurs when the prime agent has never had a fitness other than 0, and has been the prime agent for 10 generations.

With a sample of 20 runs over 2500 generations, no solution was found. Below is a graph depicting the average minimum fitness as a function of generation over the 20 run sample:


average minimum fitness
300
250
200
150
100
50
0
4
59
114
169
224
279
334
389
444
499
554
609
664
719
774
829
884
939
994
1049
1104
1159
1214
1269
1324
1379
1434
1489
1544
1599
1654
1709
1764
1819
1874
1929
1984
2039
2094
2149
2204
2259
2314
2369
2424
2479
generation (i)
f(i)

Chapter 9

Concluding remarks and possible future work

In this thesis I have presented two distinct genetic programming paradigms RTGP_1 and RTGP_2 with the aim of evolving real-time deciding programs for an arbitrary language of strings. The first conclusion that I will present here relates to the importance of the proper design of genetic programming paradigms. Concerns about the feasibility of using genetic algorithms to evolve programs using a Turing complete syntax have been raised in [29, 32]. Objections raised in [29, 32] point out the "roughness" of the fitness landscape, which ultimately reduces the genetic algorithm to random search. RTGP_1 and RTGP_2 have real-time Turing complete syntaxes so there exist deciding programs for the non-context free languages. RTGP_1 was created without any heed to the roughness of the fitness landscape and is subject to the criticisms raised in [29, 32]. The syntax of RTGP_1 bears a resemblance to the syntax described in [30]. [30] introduces a possible Turing complete syntax but does not state any results. RTGP_1 was shown to not perform well on slightly complicated regular languages. RTGP_2 aimed to improve upon RTGP_1 addressing concerns raised in [29, 32] about the roughness of the fitness landscape. It has been found that RTGP_2 greatly outperforms RTGP_1 in its ability to find a decider for a complex language. This difference in performance appears to confirm that the roughness of the fitness landscape has a large effect on how well programs can be evolved, as was suggested in [29, 32]. The difference in performance also indicates more generally that when designing a genetic programming paradigm to solve any problem, careful attention must be paid to how the phenotype is represented by the genotype. Several points to consider in the design of a genetic programming paradigm are listed below.

- Firstly the genotype must be designed in such a way that the phenotype is explicit as possible. Explicit representation of the phenotype within the genotype is likely to give rise to genetic operators that alter the phenotype in logical ways. As an example, the genotypes in the RTGP_2 paradigm are much more explicit representations of the functions they denote than are genotypes in the RTGP_1 paradigm. This was discussed in section 7.3.
- Secondly, mutation must preserve part of the phenotype. If mutation fails to preserve part of the phenotype, then it is pointless to discriminate in favor of parents that have better fitness as there is no guarantee that the children will have good fitness. If the genotype was engineered to be an explicit representation of the phenotype, then it is relatively easy to design a mutation operator that will preserve much of the phenotype. While this thesis has only looked at the mutation operator, since crossover is also a significant genetic operator, future work may also examine the crossover operator as well. A desired feature of such a crossover operator will be to combine the phenotypes of the parents in a logical way so that the resultant phenotype resembles a "blend" between the two parents.
- Thirdly, while balancing the previous point, the mutation operator must also cover a large range of possible changes to the phenotype to prevent the population from being trapped in a local optima.

This trade-off has not been addressed in this thesis and may be the subject of future research. Likewise, any crossover operators must also produce a variety of mixtures between the two parents.

The second conclusion that I will present here relates to the performance of $RTGP_2$. $RTGP_2$ has been successful at evolving a decider for every tested language except the decider for detecting 3-cycles in a graph, for which no real time decider is known to exist. Additionally, $RTGP_2$ also had some difficulty evolving a deciding program for the language NCFL. The works [28, 18, 5] have also investigated evolving deciding Turing machines using a paradigm similar to $RTGP_2$. In [28, 18, 5] the evolved Turing machines produced by their genetic algorithms are not guaranteed to run in real-time. One possible reason for the difficulty $RTGP_2$ has in evolving deciding programs for NCFL relative to other works is the real-time constraint of $RTGP_2$. Programs from $RTGP_2$ have to process each character and update their internal memory (Turing tape) in a single transition, and moreover do not have the ability to go back and reexamine the input. Turing machines described in [28, 18, 5] may take may several steps to process each character, and are able to reexamine input entries as much as needed.

From the data for $RTGP_2$, it seems that evolving deciding programs is easy for regular languages, and gets more difficult for context free languages, and more difficult for non-context free languages culminating with NCFL being the most difficult language to evolve a decider for within the scope of this thesis (not counting the language of simple graphs with a 3-cycle). This contrasts with what is observed in [5]. [5] studies the evolution of Turing machines that are not necessarily real-time. In [5] it is concluded that a language's position within the "Chomsky Hierarchy" does not affect how difficult it is to evolve a Turing machine decider for that language.

With the non-regular languages studied in this thesis and in other works [28, 18, 5] none of these languages required any nested loops to be present in a deciding Turing machine [32]. The tested languages were not complex enough to require more than one level of loop nesting as indicated by [32]. The claim made in [32] genetic programming is infeasible for evolving Turing machines with more than one level of loop nesting is not contradicted by this thesis. The performance of $RTGP_2$ and other paradigms on non-regular languages that require at least two levels of nested loops can be the subject of future research.

$RTGP_2$ may be improved in evolving deciding programs for the non-regular languages that $RTGP_2$ performed poorly on (i.e. NCFL). Such improvements can be to include more mutation operators that create a larger variety of changes to help eliminate local optimums from which the population cannot escape, as mentioned above. One possible new mutation operation can duplicate a number of states which will not change the phenotype, but open the path for more useful mutations. Another improvement can be to implement the crossover operator described in [7], and experiment with other crossover operators that mix the phenotypes of the parents in logical ways. A possible new crossover operator may be the operator described in [7], or it could be an operator that forms a new program by multiplying the sets of internal states from each of the parents together to form a new set of internal states.

Further extensions to $RTGP_2$ that may increase the range of languages for which $RTGP_2$ can evolve a real-time decider could include the addition of multiple Turing tapes, or a multi-dimensional tape to the internal memory scheme of $RTGP_2$. While these extensions will not increase the expressive power of an ordinary Turing machine, for a real-time Turing machine such extensions to the internal memory will enable more parallel computations. These extensions may make it easier for $RTGP_2$ to evolve a decider for NCFL or other non-regular languages.

Finally, noting the difficulty $RTGP_2$ has evolving deciding programs for the more complex languages, a practical application of $RTGP_2$ may not involve $RTGP_2$ creating an algorithm from scratch. Instead $RTGP_2$ can search for new algorithms by performing a "local search" around an approximate or near optimal deciding program devised by the user. The initial population will be filled with these approximations, and $RTGP_2$ can be used to search for an exact solution in the local neighborhood of these approximate solutions.

References

[1] Peter J Angeline. Adaptive and self-adaptive evolutionary computations. In *Computational intelligence: a dynamic systems perspective*. Citeseer, 1995.

[2] Peter J. Angeline. Two self-adaptive crossover operations for genetic programming, 1995.

[3] P.J. Angeline. Subtree crossover: Building block engine or macromutation. *Genetic Programming*, 97:9–17, 1997.

[4] Scott Brave. Evolving deterministic finite automata using cellular encoding. In *Proceedings of the First Annual Conference on Genetic Programming*, GECCO '96, pages 39–44, Cambridge, MA, USA, 1996. MIT Press.

[5] Clayton Burger and Mathys C Du Plessis. Does chomsky complexity affect genetic programming computational requirements? In *Proceedings of the South African Institute of Computer Scientists and Information Technologists Conference on Knowledge, Innovation and Leadership in a Diverse, Multidisciplinary Environment*, pages 31–39. ACM, 2011.

[6] V. Ciesielski and Xiang Li. Experiments with explicit for-loops in genetic programming. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 1, pages 494 – 501 Vol.1, june 2004.

[7] Janet Clegg, James Alfred Walker, and Julian Frances Miller. A new crossover technique for cartesian genetic programming. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1580–1587, New York, NY, USA, 2007. ACM.

[8] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9(3):251–280, 1990.

[9] B.D. Dunay, F.E. Petry, and B.P. Buckles. Regular language induction with genetic programming. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 396 –400 vol.1, jun 1994.

[10] Lawrence J Fogel, Peter J Angeline, and David B Fogel. An evolutionary programming approach to self-adaptation on finite state machines. *Evolutionary Programming*, pages 355–365, 1995.

[11] John R. Koza. *Genetic programming : On the programming of computers by means of natural selection.* MIT Press : Cambridge, Mass., 1992.

[12] John R. Koza. *Genetic Programming II : Automatic Discovery of Reusable Programs.* MIT Press : Cambridge, Mass., 1994.

[13] J.R. Koza. Evolving the architecture of a multi-part program in genetic programming using architecture-altering operations. *McDonnell, John R., Reynolds, Robert G., and Fogel, David B.(editors)*, pages 695–717, 1995.

[14] Sean Luke and Lee Spector. A comparison of crossover and mutation in genetic programming. In *Proceedings of the Second Annual Conference on Genetic Programming*, 1997.

[15] Julian F. Miller. Cartesian genetic programming. In Julian F. Miller, editor, *Cartesian Genetic Programming*, Natural Computing Series, pages 17–34. Springer Berlin Heidelberg, 2011. 10.1007/978-3-642-17310-3 2.

[16] Julian F. Miller and Peter Thomson. Cartesian genetic programming. In *EuroGP 2000*, pages 121 – 132, 2000.

[17] Julian Francis Miller and Simon L. Harding. Cartesian genetic programming. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, GECCO '09, pages 3489–3512, New York, NY, USA, 2009. ACM.

[18] Amashini Naidoo and Nelishia Pillay. Using genetic programming for turing machine induction. *Genetic Programming*, pages 350–361, 2008.

[19] M. ONeill and C. Ryan. Under the hood of grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1143–1148. Citeseer, 1999.

[20] M. O'Neill and C. Ryan. Grammatical evolution. *Evolutionary Computation, IEEE Transactions on*, 5(4):349 –358, aug 2001.

[21] L. Panait and S. Luke. Alternative bloat control methods. In *Genetic and Evolutionary Computation–GECCO 2004*, pages 630–641. Springer, 2004.

[22] R. Poli. Evolution of graph-like programs with parallel distributed genetic programming. *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 346–353, 1997.

[23] Riccardo Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming*, volume 2610 of *Lecture Notes in Computer Science*, pages 43–76. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-36599-0-19.

[24] Riccardo Poli. Covariant tarpeian method for bloat control in genetic programming. In *Genetic Programming Theory and Practice VIII*. Springer, 2010.

[25] Simon Raik and David Browne. Evolving state and memory in genetic programming. In Xin Yao, Jong-Hwan Kim, and Takeshi Furuhashi, editors, *Simulated Evolution and Learning*, volume 1285 of *Lecture Notes in Computer Science*, pages 73–80. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0028523.

[26] Conor Ryan, JJ Collins, and Michael Neill. Grammatical evolution: Evolving programs for an arbitrary language. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence Fogarty, editors, *Genetic Programming*, volume 1391 of *Lecture Notes in Computer Science*, pages 83–96. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0055930.

[27] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[28] Julio Tanomaru. Evolving turing machines from examples. In *Artificial Evolution*, pages 167–180. Springer, 1998.

[29] A. Teller. Genetic programming, indexed memory, the halting problem, and other curiosities. In *Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium*, pages 270–274, 1994.

[30] A. Teller. Turing completeness in the language of genetic programming with indexed memory. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 136 –141 vol.1, jun 1994.

[31] Astro Teller. Learning mental models. In *PROCEEDINGS-SPIE THE INTERNATIONAL SOCIETY FOR OPTICAL ENGINEERING*, pages 147–147. SPIE INTERNATIONAL SOCIETY FOR OPTICAL, 1993.

[32] John R Woodward and Ruibin Bai. Why evolution is not a good paradigm for program induction: a critique of genetic programming. In *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 593–600. ACM, 2009.

[33] P. Wyard. Context free grammar induction using genetic algorithms. In *Grammatical Inference: Theory, Applications and Alternatives, IEE Colloquium on*, pages P11/1 –P11/5, apr 1993.

[34] A. Zomorodian. Context-free language induction by evolution of deterministic push-down automata using genetic programming. In *Working Notes for the AAAI Symposium on Genetic Programming*, pages 127–133, 1995.

Appendix A

Test case generation algorithms

A.1 TL_1

$TL_1 = b^* = 1^*$

The $T = 1000$ test cases for the language TL_1 are generated by the following algorithm:

repeat
  Start with an empty word w.
  Choose "temp" at uniform from $\{0, 1\}$.
  if temp $= 0$ **then**
    Choose L_1 at random from $\{0, 1, \ldots, 20\}$.
    Add L_1 "1"s to w.
  else
    Choose L_0 at random uniformly from $\{0, 1, \ldots, 20\}$.
    Choose L_1 at random uniformly from $\{0, 1, \ldots, 20\}$.
    Create w randomly using L_0 "0"s and L_1 "1"s.
  end if
  for $i = 0$ to length(w) **do**
    Add a word consisting of the first i letters of w to U.
    Break loop if U is full; $|U| = 1000$.
  end for
until there is $T = 1000$ test strings in U

As can be seen, word w is chosen so that the probability of $w \in TL_1$ is approximately 50%. When temp $= 0$, w is guaranteed to come from TL_1. When temp $= 1$, w is very likely to not come from TL_1.

A.2 TL_2

$TL_2 = (ba)^* = (10)^*$

The $T = 1000$ test cases for the language TL_2 are generated by the following algorithm:

repeat
  Start with an empty word w.
  Choose L at random uniformly from $\{0, 1, \ldots, 20\}$.
  Choose "temp" at uniform from $\{0, 1\}$.
  if temp $= 0$ **then**

Alternately add 1 followed by 0 to w until length$(w) = L$.
else
Create w by randomly selecting 0 or 1 until length$(w) = L$.
end if
for $i = 0$ to length(w) **do**
Add a word consisting of the first i letters of w to U.
Break loop if U is full; $|U| = 1000$.
end for
until there is $T = 1000$ test strings in U

When temp $= 0$, w is guaranteed to come from TL_2 when L is even, otherwise w cannot come from TL_2. When temp $= 1$, w is very likely to not come from TL_2. Although the chances of w coming from TL_2 is in fact around 25%, the important detail is that this fraction is still significant and does not shrink to 0 as L increases.

A.3 TL_3

$\mathrm{TL}_3 = \{s \in \{0,1\}^* | \text{no odd length block of 0's can follow an odd length block of 1's}\}$
The $T = 1000$ test cases for the language TL_3 are generated by the following algorithm:

repeat
Start with an empty word w.
Choose L at random uniformly from $\{0, 1, \ldots, 20\}$.
Choose "temp" at uniform from $\{0, 1\}$.
if temp $= 0$ **then**
Assign to w a randomly generated string of length L that belongs to TL_3.
else
Create w by randomly selecting 0 or 1 until length$(w) = L$.
end if
for $i = 0$ to length(w) **do**
Add a word consisting of the first i letters of w to U.
Break loop if U is full; $|U| = 1000$.
end for
until there is $T = 1000$ test strings in U

When temp $= 0$, w is guaranteed to come from TL_3. Generating such a w can easily be done by using a finite automata to avoid the scenario where an odd block of "0"s follows an odd bock of "1"s. When temp $=$ 1, given a arbitrary block of "1"s followed by a block of "0"s, there is a $1/4$ probability that both blocks will be of odd length, which will violate the condition for membership in TL_3. As L increases, the probability of the odd block of "0"s following an odd bock of "1"s configuration occurring in a randomly generated string of length L rises to 1 as $L \to +\infty$. Hence a significant fraction of the w's generated when temp $= 1$ will not belong to TL_3.

A.4 TL_4

$\mathrm{TL}_4 = \{s \in \{0,1\}^* | \text{no 000 substrings}\}$
The $T = 1000$ test cases for the language TL_4 are generated by the following algorithm:

repeat
Start with an empty word w.

Choose L at random uniformly from $\{0, 1, \ldots, 20\}$.
Choose "temp" at uniform from $\{0, 1\}$.
if temp $= 0$ **then**
Assign to w a randomly generated string of length L that belongs to TL_4.
else
Create w by randomly selecting 0 or 1 until length$(w) = L$. For each letter however, there is a 1/5 probability that a 000 block will be initiated.
end if
for $i = 0$ to length(w) **do**
Add a word consisting of the first i letters of w to U.
Break loop if U is full; $|U| = 1000$.
end for
until there is $T = 1000$ test strings in U

When temp $= 0$, w is guaranteed to come from TL_4. Generating such a w can easily be done by using a finite automata to avoid the scenario where a "000" substring occurs. When temp $= 1$, given a block of 3 characters, there is a 1/8 probability that block is "000", which will violate the condition for membership in TL_4. As L increases, the probability of "000" block occurring in a randomly generated string of length L rises to 1 as $L \to +\infty$. Hence a significant fraction of the w's generated when temp $= 1$ will not belong to TL_4. L has a maximum length of 20 and so the probability of a "000" block occurring is at least $1 - (7/8)^{\lfloor 20/3 \rfloor} \approx 0.551$ when $L = 20$. To increase the probability of a "000" block occurring I added a 1/5 probability of a "000" block being inserted with each step.

A.5 TL_5

$\text{TL}_5 = \{\text{“”}, 0, 1\} \cup 0\{0, 1\}^*0 \cup 1\{0, 1\}^*1$
The $T = 1000$ test cases for the language TL_5 are generated by the following algorithm:

repeat
Start with an empty word w.
Choose L at random uniformly from $\{0, 1, \ldots, 20\}$.
Create w by randomly selecting 0 or 1 until length$(w) = L$.
for $i = 0$ to length(w) **do**
Add a word consisting of the first i letters of w to U.
Break loop if U is full; $|U| = 1000$.
end for
until there is $T = 1000$ test strings in U

TL_5 is roughly 1/2 of all strings from $\{0, 1\}^*$, so I don't need to actively choose w from TL_5 or its complement.

A.6 TL_6

$\text{TL}_6 = \{s \in \{0, 1\}^* | \#\text{ of 0's - \# of 1's} \equiv 0 \mod 3\}$
The $T = 1000$ test cases for the language TL_6 are generated by the same algorithm that generated test cases for TL_5 (see above).
TL_6 is roughly 1/3 of all strings from $\{0, 1\}^*$, so I don't need to actively choose w from TL_6 or its complement.

A.7 TL_7

$\mathrm{TL}_7 = 0^*1^*0^*1^*$

The $T = 1000$ test cases for the language TL_7 are generated by the following algorithm:

repeat
  Start with an empty word w.
  Choose L at random uniformly from $\{0, 1, \ldots, 20\}$.
  Choose "temp" at uniform from $\{0, 1\}$.
  if temp $= 0$ **then**
    Assign to w a randomly generated string of length L that belongs to TL_7.
  else
    Create w by randomly selecting 0 or 1 until length$(w) = L$.
  end if
  for $i = 0$ to length(w) **do**
    Add a word consisting of the first i letters of w to U.
    Break loop if U is full; $|U| = 1000$.
  end for
until there is $T = 1000$ test strings in U

When temp $= 0$, w is guaranteed to come from TL_7. Generating such a w can easily be done by using a finite automata to constrain w to $0^*1^*0^*1^*$. When temp $= 1$, the probability of w not belonging to TL_7 when $L = 20$ is at least $1 - 0.5^{19}(\binom{19}{0} + \binom{19}{1} + \binom{19}{2} + \binom{19}{3}) \approx 0.998$, and increases to 1 as $L \to +\infty$.

A.8 Finding triangles in a graph

The $T = 1000$ test cases for the language of graphs that contain a 3-cycle are generated by the following algorithm:

repeat
  Start with an empty word w.
  Choose L at random uniformly from $\{0, 1, \ldots, 28\}$.
  Choose "temp" at uniform from $\{0, 1\}$.
  if temp $= 0$ **then**
    Assign to w a randomly generated string of length L that denotes a triangle-free graph.
  else
    Create w by randomly selecting 0 or 1 until length$(w) = L$.
  end if
  for $i = 0$ to length(w) **do**
    Add a word consisting of the first i letters of w to U.
    Break loop if U is full; $|U| = 1000$.
  end for
until there is $T = 1000$ test strings in U

As $L \to +\infty$, the probability that a triangle is present for a randomly generated graph approaches 1. For $L = 28$ specifically, the probability of a triangle being present is at least $0.5(1 - 0.5^{\binom{\lfloor 7/2 \rfloor}{2}}) = 0.4375$, likely much greater.

Appendix B

Measuring fitness for the non-regular languages

B.1 CFL_1

$\text{CFL}_1 = \{0^n 1^n | n \geq 0\}$

For each generation, a set U of $T = 2000$ test strings are randomly generated using the following algorithm:

repeat
  Start with an empty word w.
  Choose L at random uniformly from $\{0, 1, \ldots, 20\}$.
  Choose n_0 and n_1 at random uniformly from $\{0, 1, \ldots, \lfloor L/2 \rfloor\}$.
  for $i = 1$ to n_0 **do**
    Append 0 to the end of w.
  end for
  for $i = 1$ to n_1 **do**
    Append 1 to the end of w.
  end for
  for $i = 1$ to $L - n_0 - n_1$ **do**
    Append 0 or 1 chosen randomly to the end of w.
  end for
  for $i = 0$ to length(w) **do**
    Add a word consisting of the first i letters of w to U.
    Break loop if U is full; $|U| = 2000$.
  end for
until there is $T = 2000$ test strings in U

Once the test strings have been generated, the fitness of an individual agent A is measured according to the following algorithm:

The notation $A(w)$ will denote the return value when A is called on word w.

Let $f = 0$

if $A() \neq 1$ (the empty string) **then**
  $f = f + 1$
end if

for $n = 1$ to 30 **do**
  Reset agent A.
  for $i = 1$ to n **do**
    Feed character 0 to A.
  end for
  for $i = 0$ to $2n$ **do**
    Let $v = A(0^n 1^i)$.
    if the current iteration is the first iteration where $v = 1$ **then**
      $f = f + |n - i|$
    else if $v = 1$ for a previous iteration **then**
      if v is the incorrect return value for the string $0^n 1^i$ **then**
        $f = f + 1$
      end if
    end if
  end for
  if v never had the value of 1 in the previous loop **then**
    $f = f + (n + 1)$
  end if
end for

for $i = 1$ to T **do**
  if A does not return the correct output on the i^{th} test case **then**
    $f = f + 1$
  end if
end for

return f

The above algorithm starts with the fitness f being set to 0. I first test the agent A on the empty string which should return 1. If this test fails, 1 is added to f.

For each $n = 1, 2, \ldots, 30$, A will be subjected to the following tests. For each $i = 0, 1, 2, \ldots, 2n$, the string $0^n 1^i$ is tested. Let i' be the smallest i for which A returned 1. $|n - i'|$ is then added to f. For each $i > i'$, if A returned the incorrect output, 1 is added to f. If A does not return 1 for any i, then $n + 1$ is added to f. These tests penalize A depending on how too soon or how too late 1 is ultimately returned as i increases.

Lastly A is tested on the $T = 2000$ test cases, most of which are negatives (i.e. A should return 0 for most of these cases). For each failed test 1 is added to f.

As can be clearly seen, this algorithm for calculating the fitness of an individual relies heavily on knowledge about the language CFL_1. What is to be gained by generating a decider in the TCGP_2 style of representation using genetic programming is that the decider in the TCGP_2 style of representation is guaranteed to run in real-time.

B.2 CFL_2

$\text{CFL}_2 = \{s \in \{0,1\}^* | \text{parentheses are balanced where } 0 = \text{“(” and } 1 = \text{“)”}\}$

For each generation, a set U of $T = 1000$ test strings are randomly generated using the following algorithm:

repeat
  Start with an empty word w.
  Choose L at random uniformly from $\{0, 1, \ldots, 20\}$.
  Choose “temp” at uniform from $\{0, 1\}$.

if temp = 0 **then**
Generate w by randomly selecting 0 or 1 until length(w) = L. While w is being generated, the number of "1"s should never exceed the number of "0"s.
else
Generate w by randomly selecting 0 or 1 until length(w) = L.
end if
until there is $T = 1000$ test strings in U

Once the test strings have been generated, the fitness of an individual agent A is measured according to the following algorithm:

The notation $A(w)$ will denote the return value when A is called on word w.

Let $f = 0$

for $i = 1$ to T **do**
Let w_i be the i^{th} test case from U.
Let p denote the parenthesis level, which will start at 0.
Let $g = false$ be a flag that denotes if the parenthesis level has ever become negative as A scans w_i.
for $j = 1$ to length(w_i) **do**
Let c be the j^{th} character from w_i.
if $c = 0$ **then**
$p = p + 1$
else
$p = p - 1$
if $p < 0$ **then**
$g = true$
end if
end if
end for

if $g = true$ **then**
if $A(w_i) \neq 0$ **then**
$f = f + 1$
end if
else
for $j = 0$ to $2p + 5$ **do**
Let $v = A(w_i 1^j)$.
if the current iteration is the first iteration where $v = 1$ **then**
$f = f + |p - j|$
else if $v = 1$ for a previous iteration **then**
if v is the incorrect return value for the string $w_i 1^j$ **then**
$f = f + 1$
end if
end if
Feed character 1 to A.
end for
if v never had the value of 1 in the previous loop **then**
$f = f + (p + 6)$
end if
end if
end for

return f

The above algorithm starts with the fitness f being set to 0.

For each test string w_i, A is subjected to the following tests. If the parenthesis level of w_i dips below 0, 1 is added to f if $A(w_i) \neq 0$. If the parenthesis level of w_i remains ≥ 0 and is p at the end of w_i, for each $j = 0, 1, 2, \ldots, 2p + 5$ the string $w_i 1^j$ is tested. Let j' be the smallest j for which A returned 1. $|p - j|$ is then added to f. For each $j > j'$, if A returned the incorrect output, 1 is added to f. If A does not return 1 for any j, $2p + 6$ is added to f. These tests penalize A depending on how too soon or how too late 1 is ultimately returned as j increases.

B.3 CFL_3

$\text{CFL}_3 = \{s \in \{0,1\}^* | \#\text{ of 0's} = \#\text{ of 1's}\}$

For each generation, a set U of $T = 1000$ test strings are randomly generated using the following algorithm:

repeat
  Start with an empty word w.
  Choose L at random uniformly from $\{0, 1, \ldots, 20\}$.
  Create w by randomly selecting 0 or 1 until length$(w) = L$.
until there is $T = 1000$ test strings in U

Once the test strings have been generated, the fitness of an individual agent A is measured according to the following algorithm:

Let $f = 0$

for $i = 1$ to T **do**
  Let w_i be the i^{th} test case from U.
  Let p denote the number of 0's minus the number of 1's.
  for $j = 1$ to length(w_i) **do**
    Let c be the j^{th} character from w_i.
    if $c = 0$ **then**
      $p = p + 1$
    else
      $p = p - 1$
    end if
  end for

  for $j = 0$ to $2|p| + 5$ **do**
    if $p \geq 0$ **then**
      Let string $s = w_i 1^j$
    else
      Let string $s = w_i 0^j$
    end if
    Let $v = A(s)$.
    if the current iteration is the first iteration where $v = 1$ **then**
      $f = f + ||p| - j|$
    else if $v = 1$ for a previous iteration **then**
      if v is the incorrect return value for string s **then**

$f = f + 1$
end if
end if
end for

if v never had the value of 1 in the previous loop **then**
$f = f + (|p| + 6)$
end if
end for

return f

For each test string w_i, A is subjected to the following tests. Let p be the number of 0's minus the number of 1's in w_i. For each $j = 0, 1, 2, \ldots, 2|p| + 5$ the string $w_i 1^j$ is tested if $p \geq 0$ or the string $w_i 0^j$ is tested if $p < 0$. Let j' be the smallest j for which A returned 1. $||p| - j|$ is added to f. For each $j > j'$, if A returned the incorrect output, 1 is added to f. If A does not return 1 for any j, $2|p| + 6$ is added to f. These tests penalize A depending on how too soon or how too late 1 is ultimately returned as j increases.

B.4 NCFL

NCFL = $\{a^n b^n a^n | n \geq 0\}$

For each generation, a set U of $T = 4000$ test strings is generated by the following algorithm:

repeat
Start with an empty word w.
Choose "temp" at uniform from {0,1}.
if temp = 0 **then**
Choose L at random uniformly from $\{0, 1, \ldots, 30\}$.
Choose n_0, n_1, and n_2 at random uniformly from $\{0, 1, \ldots, \lfloor L/3 \rfloor\}$.
else
Choose n_0 at random uniformly from $\{0, 1, \ldots, 15\}$.
Choose n_1 at random uniformly from $\{0, 1, \ldots, 30 - 2n_0\}$.
Let $n_2 = n_0$ and $L = 2n_0 + n_1$.
end if
for $i = 1$ to n_0 **do**
Append 0 to the end of w.
end for
for $i = 1$ to n_1 **do**
Append 1 to the end of w.
end for
for $i = 1$ to n_2 **do**
Append 0 to the end of w.
end for
for $i = 1$ to $L - n_0 - n_1 - n_2$ **do**
Append 0 or 1 chosen randomly to the end of w.
end for
for $i = 0$ to length(w) **do**
Add a word consisting of the first i letters of w to U.
Break loop if U is full; $|U| = 4000$.
end for
until there is $T = 4000$ test strings in U

Once the test strings have been generated, the fitness of an individual agent A is measured according to the following algorithm:

Let $f = 0$

Reset agent A.
if $A() \neq 1$ (the empty string) **then**
  $f = f + 1$
end if

for $n = 1$ to 30 **do**
  for $i = 0$ to $2n$ **do**
    Let $v = A(0^n 1^n 0^i)$.
    if the current iteration is the first iteration where $v = 1$ **then**
      $f = f + |n - i|$
    else if $v = 1$ for a previous iteration **then**
      if v is the incorrect return value for the string $0^n 1^n 0^i$ **then**
        $f = f + 1$
      end if
    end if
  end for
  if v never had the value of 1 in the previous loop **then**
    $f = f + (n + 1)$
  end if
end for

for $i = 1$ to T **do**
  if A does not return the correct output on the i^{th} test case **then**
    $f = f + 1$
  end if
end for

return f

This algorithm for computing the fitness works similar to the algorithm for computing the fitness for CFL_1, described in B.1.

B.5 Non cyclic output

$\text{NCO} = \{0^i : i = \sum_{j=0}^{n} j \quad \text{for some } n \geq 0\}$
The fitness of an agent A is measured by the following algorithm:

Let $f = 0$
if $A() \neq 1$ (the empty string) **then**
  $f = f + 1$
end if

Let w be an empty string.
desired_gap_length = 0

```
gap_length = 0
for i = 1 to 1000 do
  if A(w) = 1 then
    f = f + |desired_gap_length − gap_length|
    desired_gap_length = desired_gap_length + 1
    gap_length = 0
  else
    gap_length = gap_length + 1
  end if
  w = w0, 0 is appended to the end of w.
end for

if gap_length > desired_gap_length then
  f = f + (gap_length − desired_gap_length)
end if

return f
```

This algorithm penalizes the agent depending on how many "0"s appear between successive "1"s as the sequence of outputs progresses. The number of "0"s expected between each pair of "1"s increments by 1 with each "1" encountered.