Improved Measures of Robustness and Evolvability for Evolutionary Systems

by

Rémi BÉDARD-COUTURE

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT OF A MASTER'S DEGREE
WITH THESIS
M.A.Sc.

MONTREAL, DECEMBER 19, 2023

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

M. Patrick Cardinal, Thesis supervisor
Département de génie logiciel et des technologies de l'information,
École de Technologie Supérieure

M. Nawwaf Kharma, Thesis Co-Supervisor
Electrical and Computer Engineering Department, Concordia University

M. Alessandro Lameiras Koerich, Chair, Board of Examiners
Département de génie logiciel et des technologies de l'information,
École de Technologie Supérieure

M. Malcolm Heywood, External Examiner
Faculty of Computer Science, Dalhousie University

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON NOVEMBER 27, 2023

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

FOREWORD

Looking back six years ago, I wanted to expand my technical knowledge of intelligent machines. Reading articles and papers about evolving software on my own got me so far (not very far). This pushed me to want a more structured learning path. Thus began my journey to what ended up with the thesis you are about to read. At the time, I was finishing my bachelor's degree part-time and already had a job in information technology. How hard would it be to take a few more courses then draft a thesis part-time? I did not think it would take me six years. As I imagine most newly graduate students who wish to pursue a master's degree, I was full of hopes and dreams, going about "artificial intelligence" and "self-modifying code". I had so little idea of what I was talking about that every professor I met dismissed me, telling me it was not in their field of research. Until I met Prof. Patrick Cardinal. He told me the same thing but was open learn along with me on this journey if we found someone to guide us. This led me to meet Prof. Nawwaf Kharma, who was able to enlighten me on the strengths (and limitations) of Genetic Programming and understand how it could help me automate certain parts of my software development projects. I spent the first two years completing the required courses during my evenings and learning about Genetic Algorithms, their strengths, and limitations. I ended up traveling to Vienna, Austria, for a week with my wife and nine months old daughter to present an application paper about playing iterated Rock-Paper-Scissors with a Genetic Algorithm. It was a blast.

As the dream of completing my master's degree in three years was getting crushed slowly but surely, life continued and kept making it more difficult. I had a second child and decided to buy a house that would require more than a few weeks of full-time renovations. Meanwhile, Prof. Kharma continued to provide me with insightful ideas on how to build better Genetic Programming algorithms and led me to explore other fields such as biology, with a quick stop at the revolutionary (upcoming) programming language `String`, which unfortunately did not make it into this research. We decided to converge on fundamental concepts of evolvability and robustness, two forces that drive any evolutionary system. This is the main topic covered in this thesis.

ACKNOWLEDGEMENTS

It would not have been possible to complete this work without the continued support of my professors, friends, and family. Prof. Nawwaf Kharma for his contributions and support through all these years, especially with the oscillating circuit study. He always pushed me further and thought me so much about how to conduct good research. I will always remember our brainstorming sessions at the *Café de la gare*. Prof. Patrick Cardinal who probably did not expect me to go at it for all these years, but nonetheless never gave up on me. He must have thought I was a ghost student, reaching out occasionally only to ask for *another* extension, giving yet more justifications as to why it is taking so long. My friends and family who kept listening to my stories of "genetic inspired computing" and "population of programs" for all those years, hearing that I was "almost done" every time, and saw me spent countless hours on this, thank you. A special thanks to my children, who are always there to change my mind and put a smile on my face. Hopefully, they will remember my perseverance. Not to forget my employers, who gave me the flexibility needed to purse a part-time master's degree.

I was lucky enough to exchange with other Genetic Programming experts along the way, with whom I discussed about my aspirations for the future of GP and helped me gain experience with application of GP at the beginning of my journey. Kai Staats, who shared his thesis with me, Staats (2016), and introduced me to other great reads to further my knowledge of GP. Jed Simson, author of Simson & Mayo (2017), who kindly helped me test his java implementation of LGP with new benchmark problems. Jon Groff, from Neocoretechs[1], for providing guidance on the usage of Extreme Genetic Programming (XGP), a promising variant of GP. So many projects did not make it into this thesis, but at the same time it would not have been completed were it not for all these stops along the way.

Many theoretical concepts presented in this research were developed in collaboration with Prof. Peter Grogono, who has regrettably passed away in 2021. He was a close friend of Prof.

[1] http://neocoretechs.com

Nawwaf Kharma and I had the chance to meet him once before his passing. We are deeply grateful for his contributions in bringing our ideas to fruition.

This research was enabled in part by support provided by Calcul Québec (https://www.calculquebec.ca) and the Digital Research Alliance of Canada (https://alliancecan.ca), who provide access to supercomputers for researchers across Canadian universities.

Mesures améliorées de robustesse et d'évolutivité pour les systèmes évolutifs

Rémi BÉDARD-COUTURE

RÉSUMÉ

Cette recherche passe en revue les définitions existantes de l'évolutivité et de la robustesse et introduit de nouvelles mesures de référence qui remédient aux inconvénients des définitions largement utilisées. Ces nouvelles mesures sont appliquées à une variété de systèmes et de problèmes afin de démontrer leur polyvalence et leur facilité d'utilisation, y compris une nouvelle approche de modélisation simple (BNK). Cette nouvelle approche de modélisation est similaire au système NK mais présente l'avantage de fournir à la fois un génotype et un phénotype. Elle est utilisée pour démontrer la facilité d'évolution d'un circuit oscillatoire à l'aide de différentes configurations de système et leur robustesse à la perturbation de l'état, tout en examinant la relation entre la complexité du système et la multimodalité résultante de sa surface de fitness. En outre, les mesures existantes sont comparées aux mesures proposées utilisant le repliement de séquences d'ARN pour évaluer leur capacité respective à capturer l'évolutivité et la robustesse. En outre, une démonstration de l'application des nouvelles mesures aux variantes linéaires de la programmation génétique est également fournie comme preuve concrète de leur facilité d'utilisation.

Mots-clés: Évolutivité, Robustesse, Modélisation des Systèmes, Algorithmes Évolutionnaires, Systèmes Oscillants, Paysages Multimodaux

Improved Measures of Robustness and Evolvability for Evolutionary Systems

Rémi BÉDARD-COUTURE

ABSTRACT

This research reviews existing definitions of evolvability and robustness and introduces new baseline measures that addresses the drawbacks of the widely used definitions. These new measures are applied to a variety of systems and problems to demonstrate their versatility and ease of use, including a new and simple modeling approach (BNK). This new modelling approach is similar to NK System but with the advantage of providing both a genotype and a phenotype. It is used to demonstrate the ease of evolving oscillatory circuit using different system configurations and their robustness to state perturbation, while also reviewing the relationship between the complexity of the system and the resulting multi-modality of its fitness surface. Furthermore, the existing measures are compared with the proposed measures using RNA sequence folding to assess their respective ability to capture evolvability and robustness. In addition, a demonstration of application of the new measures to linear variants of Genetic Programming is also provided as concrete evidence of their ease of use.

Keywords: Evolvability, Robustness, Modelling Systems, Evolutionary Algorithms, Oscillating Systems, Multi-modal Landscapes

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

ADF	Automatically Defined Function
ANN	Artificial Neural Network
CGP	Cartesian Genetic Programming
EA	Evolutionary Algorithm
EC	Evolutionary Computing
ES	Evolutionary Strategy
ETS	École de technologie supérieure
GE	Grammatical Evolution
GEP	Gene Expression Programming
GP	Genetic Programming
LGP	Linear Genetic Programming
LISP	List Processing
MED	Minimum Edit Distance
ML	Machine Learning
RNA	Ribonucleic acid
TED	Tree Edit Distance

INTRODUCTION

Life of any system, rather be it a living organism or a machine, is driven by a goal of survival, often through reproduction mechanisms. This process is also prone to the environment in which life must evolve, and sustained by two major forces: evolvability and robustness. These concepts have been the subject of multiple studies in various fields of science such as biology and computer science.

Evolutionary systems are described with a genotype and a phenotype. A genotype is the complete sequence of genes (or alleles) describing the characteristics of the system. It is a developmental program, which at the conclusion of several phases of development, results in an (adult) phenotype. A phenotype is a description of the characteristics of the resulting system, built from the genotype.

One of the most common definitions of evolvability is the ability of an organism to adapt to a change in its environment, aiming to improve its fitness. This can be translated into the ease with which a system, represented by a genotype and phenotype, can reach a desired phenotype through mutation of its genotype. As for robustness, the widely accepted definition is the ability to maintain a given state despite perturbations. For example, keeping the same time for running a marathon under different temperatures is a form of robustness. When applied to an evolutionary system, this is done by ensuring that phenotype (or fitness) does not change despite changes in the genotype or the environment. The definitions will be further elaborated later in this research.

In the pursuit of building better evolutionary systems came the need to understand what are evolvability and robustness and how they can be measured. Accurate measurement is critical to provide sensible information about the underlying features of a system. It is important that these measures are versatile enough to be applied to a wide range of evolutionary systems, ensuring a collective understanding, and that they are easy to apply, facilitating adoption.

Given the literature review in Chapter 1, it is clear that a unified definition of evolvability and robustness is lacking from the scientific research community. This makes it hard to compare experimental results accurately, and it creates a gap between domains that would otherwise benefit from each other's advances. The goal of the present research is to provide unified definitions of evolvability and robustness, along with a framework that allow their measurement. General applicability can be demonstrated by applying these new measures to both biology and computer science problems. The results of this research should serve as a foundation for future studies of the evolvability and robustness of existing or future programming languages, and hopefully facilitate the development of better evolutionary systems, whether for computing or other domains.

In the following chapters will be presented: a review of the essential work revolving around evolvability and robustness, precise definitions of existing and improved measures, and finally a wide range of applications to understand how evolvability and robustness interact in evolutionary systems. The main contributions of this research are the improved baseline measures of robustness and evolvability and a novel modeling approach allowing to explore fitness landscapes of systems with a phenotype, along with example applications that exhibit the value of both the measures and the modeling approach. These should help close the gap in some of the open issues (like the choice of representation and problem hardness) of evolutionary algorithms by providing an efficient measurement mechanism.

All concepts explained in this research aim to provide comprehensive information, assuming the reader has no or little prior knowledge of the covered areas. However, a basic understanding of Genetic Algorithm is recommended. Some foundational books are mentioned in the review of the literature, which may be of help to readers that wish to perfect their understanding of the related concepts.

CHAPTER 1

LITERATURE REVIEW

This chapter will review the essential literature supporting the main topics explored in this research. First, an overview of the foundational literature of genetic programming (GP), including derived algorithms used in the experimentation, followed by a review of the definitions of evolvability and robustness.

1.1 Genetic Programming

In short, genetic programming is a genetic algorithm where the population is made of executable code. It was pioneered mostly by Koza, starting with his first book Koza (1992), where he covers all the essential components of tree-based GP e.g. representation, initialization, fitness, genetic operators, and demonstrates its application using the LISP language on a wide variety of problems. Banzhaf, Nordin, Keller & Francone (1998) is another book that builds upon Koza (1992) and goes deeper into the influence of biology on the inception of GP and also on the underlying mathematical aspects such as the importance of randomness in evolutionary search. However, interested readers are advised to follow the navigation guide presented in the preface to keep their focus on essential material and avoid getting lost in unnecessary details.

Although the fundamental principles of GP are language agnostic, the examples given by Koza and Banzhaf can seem a bit outdated. For reference, LISP is currently ranked 35 on the TIOBE index (Tiobe.com (2000)), down from its second position in 1987. McPhee, Poli & Langdon (2008) is another book that synthesize all the aspects of both tree-based GP and other types of genetic programming such as linear, graph and probabilistic genetic programming. It also gives a modern perspective on its applications, and includes an implementation of TinyGP in Java. For a broader view of existing evolutionary computing techniques, readers are referred to Eiben, Smith *et al.* (2003).

Not only to mention books, there is also a multitude of papers related to GP. Following the publication of Koza (1992), the research community was quick to explore additional

applications to different representations and systems. The first work on LGP (Linear Genetic Programming) first appeared in Nordin, Banzhaf *et al.* (1995), as the authors demonstrated noticeable performance improvements by manipulating machine code directly. Although the paper never mentions LGP, the manipulation of a (linear) machine register is the premise that gave rise to it. A modern implementation of LGP can be found in Simson & Mayo (2017), where the author makes an important remark about the construct of LGP:

> *A linear approach lends itself to programs which have two unique attributes: a graph-based functional structure and the existence of non-effective instructions*

In the same manner, what came to be known as CGP (Cartesian Genetic Programming) originated from Teller & Veloso (1996); a GP system with a tailored function set that is designed for signal processing. This system is made of directed graphs of nodes with a limited number of edges, and the authors argue that this representation allows for the evolution of more complex programs, given the additional recursion and branching possibilities. Later came the official CGP terminology in Miller *et al.* (1999), further detailed by Miller & Harding (2009), where the representation uses strings of integers as the genotype, and does not only focus on signal processing, but opens the door to a wide variety of applications. This simplicity also makes it easier to apply standard genetic operators rather than needing to implement custom-made unusual operators, such as the "SMART" operators of PADO (Parallel Algorithm Discovery and Orchestration) in Teller & Veloso (1996).

Grammatical Evolution (GE), first published in Ryan, Collins & Neill (1998), is remarkably similar to standard GP using tree representation, but leverages a set of grammar rules (Backus Naur Form) to decode the genotype into a function tree. A GE genotype is a string of bytes that can vary in length, like GP trees can grow and shrink. The grammar follows a sequence of interpretation, and the selection of the grammar rule for each byte is obtained by calculating the modulo of the byte value with the number of rules in the grammar. To increase the likelihood of generating valid expression trees, GE allows wrapping of the genotype, meaning that it can be repeated a certain number of times in the hope to fill an incomplete expression tree. With GE,

there is no validity enforcement of the generated individuals, leading to useless evaluations in the population.

Gene Expression Programming (GEP), introduced in Ferreira (2001), is another kind of GP algorithm which uses fixed length strings to represent expression trees. The genotype is made of multiple chromosomes that all have the same length. Although quite similar to standard GP, GEP pushes its inspiration from biology to take concepts from molecular structures such as proteins. The genotype has noncoding regions (genes that do not affect the behaviour of the system), and each chromosome is divided in two sections (a head and a tail). The head needs at least one function, and the tail is made of only terminals. The genotype is parsed from left to right, constructing the expression tree by filling each function node with the next nodes found in the genome. This often results in part of the genome being unused. The author shows a significant improvement in terms of efficiency compared to standard GP, unfortunately it lacks comparison with more recent GP variants.

There exist other types of evolutionary computation (EC) algorithms, more or less inspired from GP, such as Multi Expression Programming (MEP) from Oltean & Dumitrescu (2002) and Genetic Algorithm for Deriving Software (GADS) from Paterson (2003), but an extensive review of all existing GP techniques is beyond the scope of this research, which focuses on more fundamental features that are measures of evolvability and robustness. The algorithms leveraged in this research are further detailed in Chapter 5.

With these emerging representations, the research community started to develop frameworks that would facilitate the exploration and comparison of various evolutionary systems. Brameier, Kantschik, Dittrich & Banzhaf (1998) proposed a framework, namely SYSGP, that would allow researchers to experiment with GP, by combining various representations and even testing new ones. It is unclear if SYSGP was ever made available to the public, however researchers now have access to multiple Evolutionary Computation frameworks such as DEAP (Distributed Evolutionary Algorithms in Python) from Fortin, De Rainville, Gardner, Parizeau & Gagné (2012) and ECJ (Evolutionary Computing in Java) from Luke (1998). Both became popular

amongst the GP community and provide multiple evolutionary algorithms (EAs) and benchmarks ready to use. ECJ will be leveraged for the experiments presented in this research since it comes with more variants of GP. ECJ also comes with extensive documentation, Luke (2010), which makes it easy to extend the framework and develop new evolutionary algorithms. Beyond comparison between EAs, comparing with other machine learning (ML) algorithms, such as Artificial Neural Networks (ANN), is not trivial due to their differing approach and underlying learning processes, but when done right can provide valuable assessment of the potential of GP. One example of such assessment can be found in Tackett (1994).

There is also interest by the GP community in understanding the importance of behavioural (or semantic) phenotype diversity within a population. Vanneschi, Castelli & Silva (2014) review a few approaches to increasing behavioural diversity, such as new methods of initialization or specialized genetic operators. Although the reported results are encouraging, the authors highlight certain drawbacks, which are mostly related to computational efficiency.

Even though GP research has continued to develop over the years, its adoption has not been as fast as other ML techniques such as ANNs. A comprehensive review of the domain is presented in O'Neill, Vanneschi, Gustafson & Banzhaf (2010), where the authors detail open issues in GP.

All GP systems have a common approach to search space exploration. For computer manipulation of the structure, the solution needs to be encoded in a linear structure no matter how the phenotype looks like. So, the burden of designing a new representation hinges on one question: How should the algorithm interpret the series of numbers (or characters) that is the genotype? Answering this question leads to a visual representation of the solution and also dictates constraints to enforce when generating a new solution.

1.2 Evolvability, robustness and validity

The concepts of evolvability and robustness have been used in different contexts, but with inconsistent definitions. In order to effectively measure those concepts, harmonizing their

definitions is mandatory. In this section, a review of the existing definitions found in the literature will be conducted and refined definitions will be proposed in Chapter 2.

1.2.1 Evolvability

Alberch credited Dawkins (Dawkins (2019)) with the first use of the term Evolvability, which was meant to describe the ability of a particular group to spawn evolutionary radiations (Pigliucci (2008)). This gradually came to mean the number of different phenotypes that are accessible from a given genotype, via mutation.

Evolvability is often used as a synonym to evolution and used as per Darwin's definition of natural selection, such as the definition in Koza (1992), Banzhaf *et al.* (1998) and Wagner & Altenberg (1996). Adaptation or adaptability are also terms that are used to denote evolvability (Tackett (1994)). Some authors do not even feel the need to define evolution, like McPhee *et al.* (2008), and it is fair to assume they referred to the Darwinian definition too. In this context, natural selection applied to EAs is a direct function of fitness evaluation, and thus is entirely dependant on the algorithm's phenotype representation. Wagner & Altenberg (1996) highlight the problem of representation and the implication of evolvability being driven by the choice of genetic operators, by comparing the genotype-phenotype map in evolutionary biology and the same map in EA.

Altenberg *et al.* (1994) take a very mathematical approach to GP evolvability by analyzing the frequency of Automatically Defined Functions (ADFs) and their impact on the fitness of the offspring. It is based on the concept of a transmission function that maps the probabilities of the algorithm's genetic operators to the distribution of solutions in the neighborhood of the parent solution. In this research, evolvability is defined as the ability of a given set of solutions to produce fitter variants. This work is expanded in Smith, Husbands & O'Shea (2002), in which the measure of evolvability is used to assess the fitness landscape of a problem. Although their results present a clear view of the benefits of this measure, their applications were limited to

theoretical problems, and would be quite inefficient if applied to benchmark problems (e.g., the artificial ant) due to the amount of sampling required.

In 2005, an empirical study was reported in Reisinger, Stanley & Miikkulainen (2005) as an attempt to measure evolvability, which in this context was defined as the ability of a representation to learn the underlying factors of a fitness function. A distinction is made between the latent evolvability of the representation and the evolvability acquired through training with a changing fitness function. They experiment with this idea by training a population on an adaptive fitness function, then evaluating it against different targets, measuring how quickly the trained population can reach each test case. An important conclusion from their experiments is that the use of behavioral phenotype encoding increases the evolvability of a system. This approach for assessing evolvability is subject to a few hyper-parameters (e.g., the length of the training phase), which need further investigation to understand how they should be tuned for different problems.

One of the most notable papers on both evolvability and robustness is Wagner (2008). The concept of evolvability is broken into genotype and phenotype evolvability. In his paper, the author argues that evolvability and robustness are negatively correlated in genotype space but positively correlated in phenotype space, for both genotypic and phenotypic spaces. This contrasts with the more common assumption that evolvability and robustness are opposing forces. Wagner defines evolvability as the number of mutations it takes for a fit individual to become fit again in a new environment. More specifically, the measure of genotype evolvability is defined as the number of different phenotypes found in the first neighborhood (individuals linked by a single gene mutation) of the genotype and the measure of phenotype evolvability is defined as the number of different phenotypes found in the first neighborhood of a phenotype. This paper describes clear measures of evolvability and demonstrates their application with ribonucleic acid (RNA) sequence folding. A key limitation of these measures is that they are limited to the first neighborhood, which limits their ability to assess the evolvability of representations whose fitness may well benefit from more than one mutation. These definitions have become well-known and often used as-is or as inspiration for measuring evolvability. They will be further reviewed in Section 2.4.1.

Another paper explored the Darwinian notion of evolvability through the application of Probably Approximately Correct (PAC) Learning Model, first described in Valiant (1984). The approach described in Valiant (2009) models the individuals as hypotheses bounded by polynomial functions, evaluated against a distribution of the target function. The experimentation is done only with multi-argument boolean functions (conjunctions and parity), under the rationale that it is similar to circuits found in living organisms. The conclusions are open-ended, as only one of two experiments showed positive results. More investigation is needed to understand what makes a structure evolvable in PAC.

More recently, evolvability has been associated to the engineering process of manufacturing in Calcott (2014). Using the discipline of software engineering as an analogy, this paper exposes all major studied concepts related to evolvability such as neutrality and modularity, as well as more novel concepts of evolvability such as interfaces, while highlighting how they appear in engineering and how evolvability differs from the notion of *adaptationism*. Although evolvability is the core concept of this paper, the author does not define the concept in detail, but rather make a few references to it as the capacity of a system to facilitate variations.

Following from Wagner (2008), Wagner and Hu continued to explore the relationships between evolvability and robustness, at the genotypic and phenotypic levels, and also at the fitness level, in Hu & Banzhaf (2018). This paper generalizes the definition of evolvability as the ability to generate novel and adaptive phenotypes. Although the textual definition may be a bit generic, the refined measures bring depth to the concept. The measure for genotype evolvability is the same as previously defined in Wagner (2008), but the measure for phenotype evolvability has been adapted to account for their specific experimentation, which leverages a fully connected graph of all the possible genotypes, linked by single gene mutation (they differ by a single gene). Although it provides more information, this is only possible due to the small search space used in their example. Phenotype evolvability is measured as the distribution of single-point mutations that map to another phenotype. The experimentation is done using LGP on a simple boolean circuit, allowing the algorithm to exhaustively explore the whole genotype space. There is a need for a different method to measure evolvability and robustness in EAs using more

complex representations. The results shown in Hu & Banzhaf (2018) support the hypothesis that evolvability and robustness are collaborative forces of evolution. This study suffers from the same issue that the original measures of Wagner (2008) suffered from; it only considers the first neighborhood. The authors make an interesting suggestion for future research: that evolvability and robustness could be measured using a probabilistic method such as Markov Chain analysis. This could have the potential to mitigate the tedious process of empirical analysis for each problem.

More recently, Mayer & Hansen (2017) refuted Wagner's suggestion that phenotype evolvability and robustness are positively correlated. They defined a mathematical approach leveraging boolean matrices to represent the genotype-phenotype mapping, allowing them to conduct advanced analysis of the neighborhoods. First, they explore the first neighborhood (using single point mutation) with controlled boolean genotype and phenotype structures, then explore higher neighborhoods through increasingly complex $G \rightarrow P$ map, pointing out that Wagner's definitions are not sufficient as they depend on single random mutation:

> *An inherent assumption in Wagner's definitions is that evolvability depends on finding a single random phenotype through a single random mutation. This stands in sharp contrast to the traditional neo-Darwinian view of adaptations being built sequentially through many contingent steps.*

Although providing a sound theoretical framework for assessing evolvability and robustness, the authors had to limit their experimentation to small genotype of only six genes due to computational constraints, which may indicate that this technique is not well suited for larger scale applications.

1.2.2 Robustness

De Visser et al. (De Visser *et al.* (2003)) defines robustness as the invariance of phenotypes in the face of perturbation. Obviously, perturbations may be environmental in nature (e.g., change

in temperature), and may be responded to via mechanisms of stabilization (e.g., gene regulation). However, perturbation is often taken to mean genetic mutation.

As evolvability can be tied to the exploration of the search space, robustness is often seen as akin to exploitation, or the ability to keep the same phenotype after a change in the genotype. There are not as many publications on evolutionary robustness. One good paper on this topic is Wagner (2008). Not only does it provide a good perspective on evolvability, as reviewed in Section 1.2.1, but it also details its relationship with robustness. The author defines robustness of a system as its ability to accept mutation without changing its phenotype. As part of his experimentation, the proposed measures of robustness are also defined for both the genotype and the phenotype of a system. The measure of genotype robustness is defined as the number of genotypes that have the same phenotype in the first neighborhood of a given genotype, and the measure of phenotype robustness is defined as the number of genotypes that have the same phenotype in the first neighborhood of all genotypes of a given phenotype. In addition to the remarks made on this paper in Section 1.2.1, it is to be noted that this definition of phenotype robustness implies the ability to map a phenotype back to all its genotypes. In the case of RNA, there exist inverse folders that can find sequences (genotypes) for a given secondary structure (phenotype), but for most evolutionary problems, it is not possible to find all the genotypes that map to a phenotype. One could sample the genotype space and group all the genotypes with the same phenotype together, however this does not guarantee an accurate representation of the phenotype space.

There are also mentions of concepts that are associated to robustness in Calcott (2014), notably the notion of neutrality. In this context, neutrality is a genotypic change that has no impact on the phenotype, but renders the genotype able to attain further phenotypic changes more easily. The author describes neutrality as an inherent feature of the phenotype encoding, meaning that the choice of encoding has a direct impact on the robustness of a system.

Similarly, in Schulte, Fry, Fast, Weimer & Forrest (2014), neutrality is studied in the form of mutational robustness applied to software engineering. In this paper, robustness is defined as

the ability of a system to maintain its functional behaviour across different environments. This study reports inherent mutational robustness of computer programs, both with syntax trees and assembly representations, by inducing random mutations and assessing the functional behaviour of the mutated programs against a test suite. Mutational robustness is measured as the fraction of mutated programs that continue to pass all the tests. Unfortunately, there is no perfect way to assess the effectiveness of a test suite, but the authors have mitigated this issue by carefully selecting problems with established test cases.

Coming back to Hu & Banzhaf (2018), which can be seen as a continuation of Wagner (2008), robustness is textually defined loosely as the resilience to constant perturbations of an evolutionary system. The measures of genotype and phenotype robustness are the same as originally defined in Wagner (2008). The authors experiment with a small scale LGP system using boolean functions, allowing them to build then entire genotype-phenotype map. The results presented for this application suggest that phenotype robustness is strong when exploring the first neighborhood, since most phenotype are overrepresented in the genotype-phenotype map.

Many researchers refer to robustness and evolvability as opposing forces. By definition, opposing forces would mean that they are trying to take over the other. A better qualification would be complementary, as even with or without correlation as they are always present in evolutionary systems.

1.2.3 Validity

There is also the notion of validity when programming is involved. It has been left out of this review since most, if not all, GP algorithms have mechanisms that ensure that every source code is executable. It is important to acknowledge that enforcing validity has the benefit of directly yielding working solutions (although not always fit or useful) by restricting the search space. However, restricting the search space has the effect of potentially discarding evolutionary shortcuts to ideal solutions.

The next chapter will provide detailed definitions of all concepts needed to understand and use the existing measures of robustness and evolvability. Improved measures will be introduced to address the drawbacks of the existing ones, with a direct comparison of both existing and improved measures. The methodology used throughout the experiments of this thesis, based on an EA, will also be described.

CHAPTER 2

MEASURES OF EVOLVABILITY AND ROBUSTNESS

The measures of evolvability and robustness defined by Wagner in Wagner (2008) were innovative in the sense that they proposed a standard definition for any evolutionary system. However, their definitions are limited to a small fraction of the search space and do not capture the performance of applied genetic algorithms due to the nature of the changes induced by their genetic operators. In this chapter, it will be demonstrated how their definitions can be generalized to expand the coverage of the search space and propose new measures to address the weaknesses of the classical definitions.

2.1 Conventions and definitions

Before getting into the detailed measures, some common terms can be defined that will be used throughout the rest of this thesis.

$$
\begin{aligned}
g &\equiv \text{genotype of an individual} \\
p &\equiv \text{phenotype of an individual} \\
f &\equiv \text{fitness of an individual} \\
G &\equiv \text{set of genotypes in the population} \\
P &\equiv \text{set of phenotypes in the population} \\
F &\equiv \text{set of fitness in the population} \\
D &\equiv \text{the distance between two members of a set}
\end{aligned}
$$

Additionally, sizes of sets such as G and P are denoted by preceding it with # (e.g., $\#G$). Members of sets may have a prime ($'$) or subscripts, usually to denote another member of the set (e.g., g'), and the distance function D is often specified with subscript to distinguish between genotype (D_g) and phenotype (D_p) distances. The inequality symbol appended as a subscript to a set denotation ($\{x, y, z\}_{\neq}$) means that the set contains only distinct elements. Inherent to EAs,

the mappings between genotype, phenotype and fitness are also relevant.

$$\Gamma(g) \rightarrow p \equiv \text{genotype to phenotype mapping}$$
$$\Phi(g) \rightarrow f \equiv \text{genotype to fitness mapping}$$
$$\Phi(p) \rightarrow f \equiv \text{phenotype to fitness mapping}$$

2.2 Importance of using the right distance measure

For every GA, the choice of genotype and phenotype representation plays a significant role in its performance. Choosing the right representation is not easy and should consider constraints relevant to the problem to solve. Additionally, the choice of genetic operators will also dictate how the algorithm can explore the search space and exploit the areas of interest (potential solutions). Mutation and crossover are the two most common types of genetic operators, and each can be realized in multiples ways (e.g., single point mutation, uniform mutation, 1-point crossover, 2-points crossover). With that in mind, the representation and genetic operators that act on it should allow for precise measurement of the distance between any two individuals' genotype or phenotype. This brings the question of how to measure the distance between two individuals. The Hamming distance is a good way to measure changes in character strings, especially with GAs where the genotype is represented by its ordered genes as a character string of fixed length. Levenshtein distance (Levenshtein *et al.* (1966)), or commonly known as edit distance, should also be considered if the genetic operators allow for insertion and deletion of genes. The distance between numerical values can be measured using the mean squared error, and the distance between binary strings can be measured using the bitwise exclusive or operator ($\oplus$). Many other distance measures can be devised depending on the genotype-phenotype mapping and the context of the genetic operators, such as real, integer or other types employed in the representation of individuals.

2.2.1 Genotype and Phenotype Encodings

The genotype and phenotype encoding refers to how the genotype and phenotype are written using ASCII characters in sequence, similar to how RNA sequences can be encoded with a few capital letters (*A,U,C,G*). Although sometimes the encoding can be used as the representation of the genotype or phenotype, they are two distinct concepts. The difference of the genotype or phenotype representation is that it can leverage graphical features to illustrate itself, such as tables or diagrams.

For example, a three-dimensional data structure can be flattened by appending the various rows to each other. Similarly, a tree can be encoded linearly by applying a tree traversal method. This is also applicable to other GP methods such as GE (Ryan *et al.* (1998)), GEP (Ferreira (2001)), and CGP (Miller & Harding (2009)) where the genotype or phenotype can be encoded so that the system is represented by a string of characters. The choice of encoding has never been an explicit concern of GP because the algorithm works with the raw (decoded) representation of the individual, often by leveraging a custom data structure, and implicitly enforcing the problem constraints. However, an encoded genotype or phenotype facilitates the evaluation of the distance between two individuals, when standard measures such as Hamming distance are applicable. As the measures of evolvability and robustness bring valuable information to improve a GP problem, it will become essential to dissociate representation from encoding.

To illustrate this concept, consider a simple case of the artificial ant on the Santa Fe trail problem, as originally described in Koza (1992). In this problem, an artificial ant is placed on a toroidal grid, and follows a set of rules in order to collect all the food on the grid. The ant has access to six rules to construct its genotype: turn left (`left`), turn right (`right`), move (`move`), check if there is food ahead (`if-food-ahead`), execute two rules in order (`progn2`) and execute three rules in order (`progn3`). Using a tree representation of the rules, Figure 2.1 shows the genotype of a simple ant with ten nodes. For most people, the natural way to read the tree would be by using the depth-first search; starting from the root node, then descending to each leaf from left to right. This is the same traversal method used in GP, which would give the following encoding

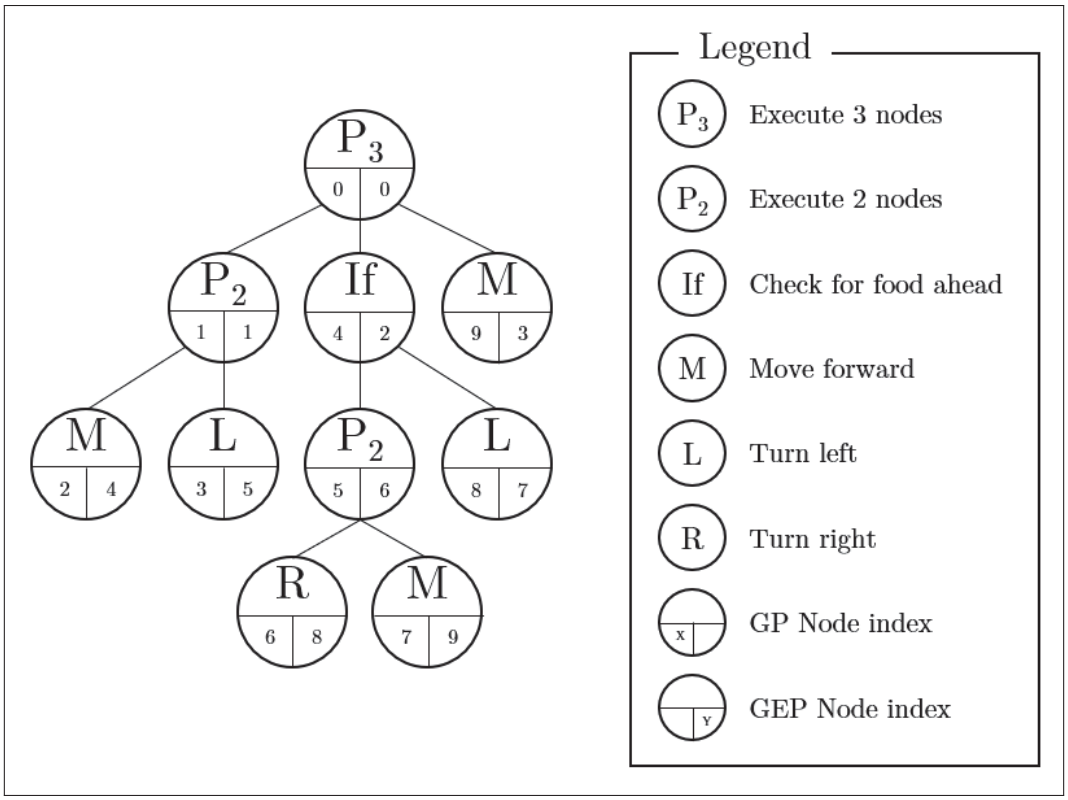


Figure 2.1 Example genotype representation of a program solution to the artificial ant trail problem

```
(progn3 (progn2 move left) (if-food-ahead (progn2 right move) left) move)
```

where the parentheses are used to capture branches of the tree. In other scenarios however, a different traversal method will lead to a different encoding. Take for example the breadth-first search traversal method used in GEP. In this case, the nodes are read from left to right, one level at a time, starting from the root. In GEP, this results in an encoding that looks like:

```
progn3.progn2.if-food-ahead.move.move.left.progn2.left.right.move
```

where genes are connected by periods.

This example uses verbose gene symbols to ease the reading. In practice, the gene symbols would be simplified using shorter character sequence. In this case, it could be as simple as

`3.2.i.m.m.l.2.l.r.m` for the GEP encoding. The encoding can be adapted to facilitate distance measurement, allowing the application of standard distance measurement algorithms directly to the encoded individual. Cases where advanced genetic operators or complex representations are used would warrant the measurement of distance using specialized context-specific algorithms.

2.2.2 Phenotype representation

The phenotype of an individual is the resulting structure of the active genes (genes that contribute to the evaluation of the individual). For example, the phenotype of a human DNA would be its physical body. With GP languages, the phenotype is the function (or software code), often represented as rooted trees. The phenotype is what is used to assess the behaviour in a given environment and assess fitness. Figure 2.2 illustrates the process of decoding a genotype into a structural phenotype, assessing it's behaviour, and evaluating fitness. Firstly, the genotype is decoded to build the structural phenotype. Secondly, a set of inputs are provided to the structural phenotype to assess its behaviour. Thirdly, the resulting behaviour (outputs) is measured against the ideal solution to assess the fitness of the individual.

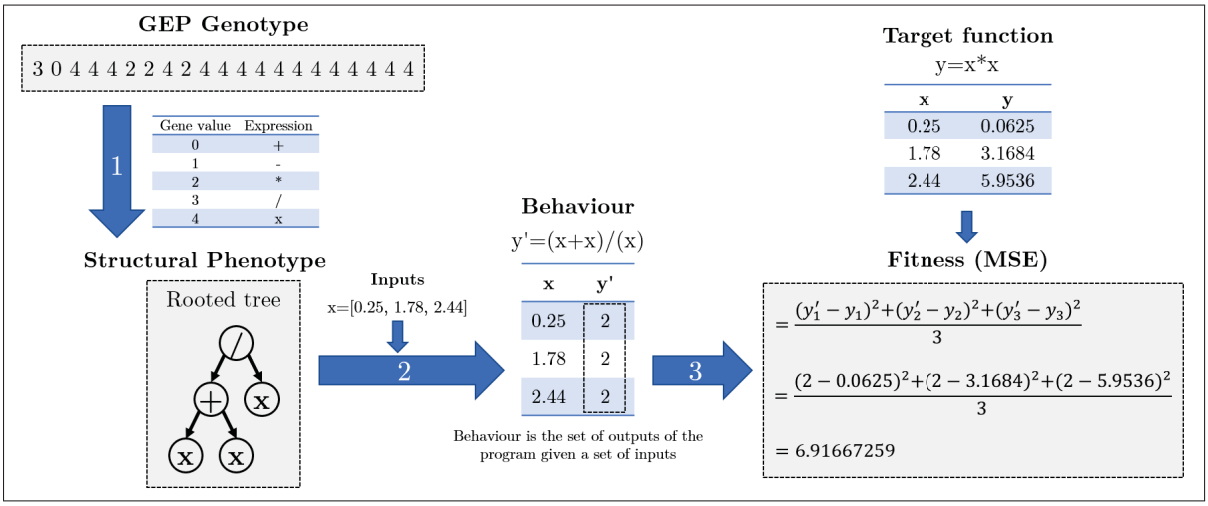


Figure 2.2 An end-to-end example showing how an individual genotype is mapped to a structural phenotype, then evaluated for fitness

2.2.3 Genotype distance

The genotype distance between genotypes g and g' measures the minimum number of mutations n needed to get from g to g'.

$$D_g(g, g') = min_n(g, g', n) \tag{2.1}$$

2.2.4 Phenotype distance

The phenotype distance is more flexible as it highly depends on the choice of encoding and aims at assessing divergence in structure or behavior, given the environment in which the GA performs. For cases when the phenotype encoding is a fixed length string, a syntactic measure such as the Hamming distance can be used, while at other times the behavior can be assessed by fitness as shown in Equation 2.2.

$$D_p(p, p') = |\Phi_p(p) - \Phi_p(p')| \tag{2.2}$$

2.2.5 Hamming distance

The Hamming distance is a good fit for most distance measurements where the encoding of the genotype or phenotype is of fixed length and each character position represents the same feature in the individual. The drawback is that each feature needs to be represented by a single symbol, usually alphanumerical. This can be addressed by maintaining a mapping table of characters to genes, either inside or outside the GA. Equation 2.3 shows how it is computed over sequences x and y, both of length l.

$$D(x, y) = \sum_{i=1}^{l} \delta_i \text{ where } \begin{cases} \delta_i = 0, & \text{if } x_i = y_i \\ \delta_i = 1, & \text{if } x_i \neq y_i \end{cases} \tag{2.3}$$

2.2.6 Mean squared error

Some problems such as symbolic regression need to express their results with real values. In such cases, a numerical distance measure is more appropriate. Mean squared error (MSE) is a good choice for many GAs since mathematical problems need to be evaluated using multiple points in a given range. The standard mean squared error (MSE) equation is represented in Equation 2.4.

$$D(x, y) = \frac{1}{n} \sum_{i=1}^{n} (x_i - y_i)^2 \tag{2.4}$$

2.2.7 Tree Edit Distance

Non-recursive functions can be represented as rooted tree structures. Koza (1992) used that rooted tree representation in his demonstration of Genetic Programming using LISP programs. It will later be shown in Section 5.1 that multiple variants of GP can also represent their structural phenotypes with trees, given some constraints. Rooted trees are special cases of unrooted trees, which in turn are special cases of graphs. Each have their own methods of measuring distance between two structures, such as the Generalized Robinson-Foulds metric presented in Smith (2020) for unrooted trees and Graph Edit Distance (GED) from Kim (2023) for graphs. Although both of these could be leveraged to measure the distance between two rooted trees, the generalization comes at the cost of efficiency. For that reason, the Tree Edit Distance (TED) remains the most appropriate distance measure when comparing two rooted trees.

The basis of TED is to measure the distance by identifying the minimal number of changes required to transform one tree into the other. There are three possible edit operations: deletion of a node, insertion of a node and replacement of a node. By assigning a cost to each operation (usually 1), dynamic programming is used to measure the minimum edit distance between two trees, similarly to the Levenshtein distance presented in Levenshtein *et al.* (1966). An example application of dynamic programming to measure the distance between two sequences is shown in

Figure 2.3, where the cost of each operation (insert, delete, replace) is set to 1, and the distance is the sum of the cost of operations.

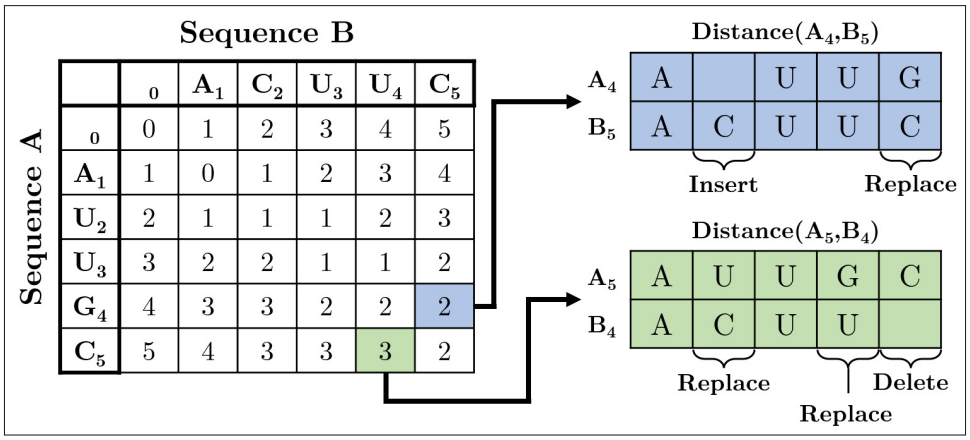


Figure 2.3 Example application of dynamic programming to compute the edit distance between sequences A and B

Although similar in nature, TED is more complex as multiple dynamic programming matrices need to be computed for all subtrees compared during decomposition. The choice of decomposition strategy also plays a key role in the performance of this measure. The decomposition strategy is the method used to decompose a tree into smaller subtrees. The state-of-the-art TED algorithm is AP-TED$^+$ from Pawlik & Augsten (2016b), which uses a decomposition strategy called "All Path" that checks for all root-leaf paths. This contrast with any previous decomposition strategies which focused on either leftmost or rightmost paths. A path decomposes a tree into subtrees by deleting nodes along the path and recursively decomposing the resulting subtrees. A complete description of the TED computation and existing decomposition strategies is beyond the scope of this research. However, interested readers can find a detailed tutorial of the original TED measure proposed by Zhang & Shasha (1989) in Paaßen (2018). There are also multiple implementations freely available for various platforms[1] for anyone that wishes to use TED.

[1] Visit http://tree-edit-distance.dbresearch.uni-salzburg.at for a review of the latest literature and available implementations.

2.3 Mutational neighborhoods

The notion of mutational neighborhood refers to all the individuals or neighbors that are at a given distance k from a given individual, also denoted as *k-neighborhood* (N_k), and can be applied to either the genotype ($N_k(g)$), as shown in Equation 2.5, or phenotype ($N_k(p)$), as shown in Equation 2.6.

$$N_k(g) = \{g' | D_g(g, g') = k\} \tag{2.5}$$

$$N_k(p) = \{p' | D_g(g, g') = k \text{ and } \Gamma(g') = p' \ \forall \ g | \Gamma(g) = p\} \tag{2.6}$$

Neighbors of a genotype that map to the same phenotype are considered to be neutral and denoted by $NN_k(g)$. When $k = 1$, the subscript may be omitted for brevity such that the neighborhood and neutral neighborhood can be written as N and NN.

$$NN_k(g) = \{g' | D_g(g, g') = k \text{ and } \Gamma(g) = \Gamma(g')\} \tag{2.7}$$

We can easily represent this concept using a simple binary string example, using the Hamming distance and a fixed length of 4. Figure 2.4 shows the first and second neighborhoods of the `0110` binary string. It is easy to compute the neighborhood size of a string that uses a fixed alphabet like a binary string or RNA sequence using the equation $(A-1)^N\binom{L}{N}$, where A is the alphabet size, N is the target neighborhood and L is the length of the string. However, it is not always trivial (nor necessary) to know the total size of a neighborhood of more complex genotype or phenotype encodings, where the alphabet can vary depending on the gene or structure of the genotype.

2.3.1 Illustration of the relationships

Figure 2.5 illustrates the various relationships of the first neighborhood ($k = 1$) that can exist in a population. The rounded boxes represent phenotypes ($[p_1..p_3]$) and circles represent genotypes ($[g_1..g_8]$) that map to the phenotype they are in. Dashed lines link genotypes that are one

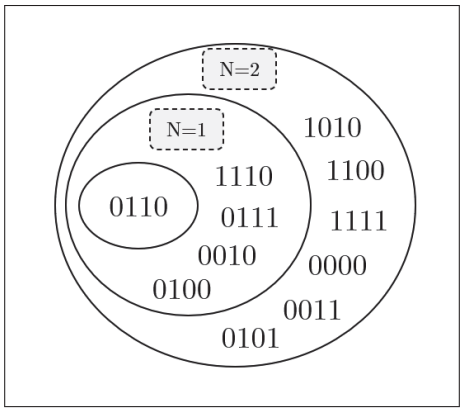


Figure 2.4 Example of the first ($N = 1$) and second ($N = 2$) mutational neighborhood of a binary string

mutational step away from each other and full lines link phenotypes in the first neighborhood of on another. Using the dashed lines to map the genotype neighbors and plain lines to map the phenotype neighbors, the first neighborhood set can be determined. For example, the first neighborhood of genotype g_4 is $[g_2, g_3, g_7, g_8]$ and the first neighborhood of phenotype p_2 is $[p_1]$.

2.4 Generalization of classical measures

From the definitions of evolvability and robustness introduced in Section 1.2, the only notable definitions of how they can be measured came from Andreas Wagner in Wagner (2008). By exploiting the relationship of the first neighborhood, evolvability and robustness can be measured by counting the number of neutral or non-neutral neighbors, using both genotype and phenotype encoding.

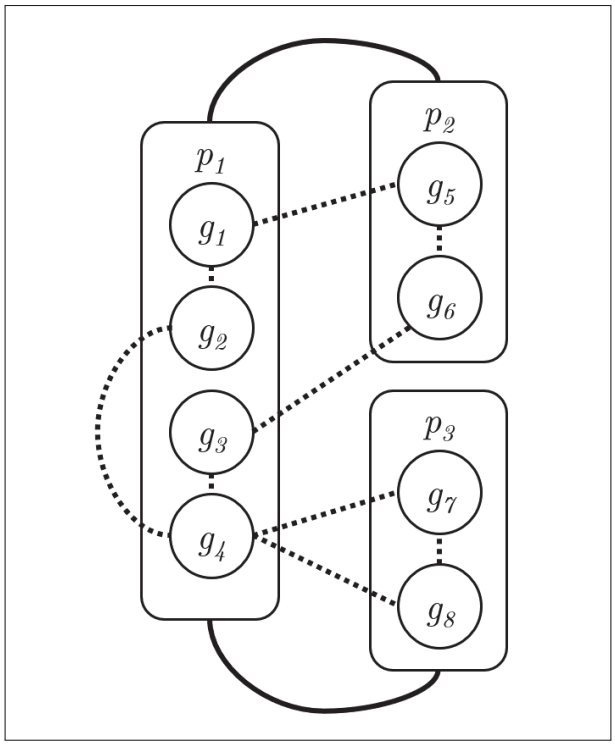


Figure 2.5 Simple illustration of the relationships between genotypes and phenotypes for the first neighborhood ($k = 1$)

2.4.1 Classical definitions of evolvability and robustness

The measures put forward in Wagner (2008) are referred to as the *classical* measures, as they are the first detailed application found in the literature. A total of four measures are devised in the following subsections.

2.4.1.1 Genotype evolvability

Wagner defines genotype evolvability E_g as the number of distinct different phenotypes found in the 1-neighborhood of a genotype g. For a single genotype, g:

$$E_g(g) = \#\{p' | D_g(g, g') = 1 \text{ and } \Gamma(g') = p'\}_{\neq} \tag{2.8}$$

Equation 2.8 can be used to compute the average genotype evolvability of a population:

$$E_g(G) = \frac{1}{\#G} \sum_{g \in G} E_g(g) \tag{2.9}$$

Taking Figure 2.5 as example, the average genotype evolvability is equal to 1.875. The genotype evolvability for each genotype can be found in Table 2.1.

Table 2.1 The genotype evolvability for each genotype presented in Figure 2.5

genotype g	$E_g(g)$
g_1	2
g_2	1
g_3	2
g_4	2
g_5	2
g_6	2
g_7	2
g_8	2

2.4.1.2 Phenotype evolvability

Wagner defines phenotype evolvability E_p as the number of distinct different phenotypes found in the 1-neighborhood of a phenotype p. For a single phenotype p, that is:

$$E_p(p) = \#N(p)_{\neq} \tag{2.10}$$

Equation 2.10 can be used to compute the average phenotype evolvability of a population:

$$E_p(P) = \frac{1}{\#P} \sum_{p \in P} E_p(p) \tag{2.11}$$

Taking Figure 2.5 as example, the average phenotype evolvability is equal to ≈ 2.333. The phenotype evolvability for each phenotype can be found in Table 2.2.

Table 2.2 The phenotype evolvability for each phenotype presented in Figure 2.5

phenotype p	$E_p(p)$
p_1	3
p_2	2
p_3	2

2.4.1.3 Genotype robustness

Wagner defines R_g as the proportion of neutral neighbors of a genotype g:

$$R_g(g) = \frac{\#NN(g)}{\#N(g)} \tag{2.12}$$

Equation 2.12 can be used to compute the average genotype robustness of a population:

$$R_g(G) = \frac{1}{\#G} \sum_{g \in G} R_g(g) \tag{2.13}$$

Taking Figure 2.5 as example, the average genotype robustness is equal to 0.5625. The genotype robustness for each genotype can be found in Table 2.3.

2.4.1.4 Phenotype robustness

Wagner defines R_p as the number of neutral neighbors averaged over all genotypes with a given phenotype p:

$$R_p(p) = \frac{1}{\#G_p} \sum_{G_p} \#NN(g) \text{ where } G_p = \{g | \Gamma(g) = p\} \tag{2.14}$$

Table 2.3 The genotype robustness for each genotype presented in Figure 2.5

genotype g	$R_g(g)$
g_1	$1/2$
g_2	$2/2$
g_3	$1/2$
g_4	$2/4$
g_5	$1/2$
g_6	$1/2$
g_7	$1/2$
g_8	$1/2$

Equation 2.14 can be used to compute the average phenotype robustness of a population:

$$R_p(P) = \frac{1}{\#P} \sum_{p \in P} R_p(p) \tag{2.15}$$

Taking Figure 2.5 as example, the average phenotype robustness is equal to ≈ 1.166. The phenotype robustness for each phenotype can be found in Table 2.4.

Table 2.4 The phenotype robustness for each phenotype presented in Figure 2.5

phenotype p	$R_p(p)$
p_1	$6/4$
p_2	$2/2$
p_3	$2/2$

2.4.1.5 Limitations of the classical definitions

Wagner's definitions suffer from the following drawbacks:

1. They provide **incomplete** information about evolvability and robustness due to their limitation to the first neighborhood. Evolution in most GAs happens through consecutive changes to the genotype of an individual, both from the genetic operators and selection mechanisms used, leading to the exploration of bigger neighborhoods.
2. The phenotype measures require an **inverse mapping** function of the $G \rightarrow P$ map to find all the genotypes for a given phenotype. In many problems, this is impossible to achieve.
3. They lack **normalization**. The number of neighbors may vary with the size of the genotype space, rendering the comparison of the resulting measure useless between two different applications.

2.4.2 Generalized definitions of evolvability and robustness

Generalizing these measures is a straightforward process. The measures simply need to allow for neighborhoods greater then 1. So rather than forcing $k = 1$, the measures can have any value $k \in \mathbb{N}$. Normalization of the measures is also applied in order to provide a metric that can be compared across multiple experiments. The four generalized equations are presented below. As with the classical constrained version, they can be applied to a population of individuals by averaging the individual values (omitted for brevity).

2.4.2.1 Generalized genotype evolvability

Equation 2.8 is generalized by relaxing the constraint on the neighborhood k and normalizing the set of phenotypes over the size of the neighborhood.

$$E_g^+(g) = \frac{\#\{p' | D_g(g, g') = k \text{ and } \Gamma(g') = p'\}_{\neq}}{\#N_k(g)} \tag{2.16}$$

2.4.2.2 Generalized phenotype evolvability

Equation 2.10 is generalized by relaxing the constraint on the neighborhood k and normalizing the set of phenotypes over the biggest phenotype neighborhood found. In the unlikely event that there exists only a single phenotype in the entire space, this will results in a perfect phenotype evolvability ($E_p^+ = 1$).

$$E_p^+(p) = \frac{\#N_k(p)_{\neq}}{\max(\#N_k(p)_{\neq} \in P)} \tag{2.17}$$

2.4.2.3 Generalized genotype robustness

Equation 2.12 is generalized by relaxing the constraint on the neighborhood k.

$$R_g^+(g) = \frac{\#NN_k(g)}{\#N_k(g)} \tag{2.18}$$

2.4.2.4 Generalized phenotype robustness

Equation 2.14 is generalized by relaxing the constraint on the neighborhood k and normalizing the neutral neighborhoods over the entire neighborhood.

$$R_p^+(p) = \frac{1}{\#G_p} \sum_{G_p} \frac{\#NN_k(g)}{\#N_k(g)} \text{ where } G_p = \{g|\Gamma(g) = p\} \tag{2.19}$$

2.4.2.5 Extent of the generalization

The classical measures, even when generalized, only give partial views of the potential evolvability and robustness of a system, as will be demonstrated in Section 3.2. It also has the drawback of needing to validate multiple neighborhoods, ideally all of them, which is often intractable even for moderate neighborhoods ($k > 5$). Inherently, the issue of needing an inverse mapping function for phenotype measures remains.

2.5 Proposed baseline measures

The measures proposed by Wagner and Banzhaf tried to capture the evolvability and robustness of evolutionary system, but failed to do so by limiting the measures to only the first neighborhood. Even when generalized, the first few neighborhoods hold limited information about the surrounding search space and do not reflect the search process of genetic operators. The advantage of the classical measures is that they rely only on the genotype distance, so a single distance measure needs to be defined. However, one of the major downsides is that the phenotype measures (E_p and R_p) need an inverse mapping function to discover the genotypes that map to a phenotype, which is often not possible due to the size of the genotype space and the ability to form a valid phenotype without using a genotype. A potential workaround is to sample the genotype space, map all the genotypes and then sample from the set of phenotypes found. This process is highly inefficient and does not guarantee an accurate representation of phenotype space.

To address this issue, this research proposes two new complimentary measures that rely on the notion of minimum genotypic change through evolutionary exploration. These measures, namely *baseline evolvability* and *baseline robustness*, use a baseline mutation to explore the neighborhoods of an individual and draw conclusions from the ensuing evolutionary walk processed, which are further detailed in the following sections.

2.5.1 Baseline mutation

The baseline mutation is equivalent to a 1-point mutation, where a single gene from the genotype is picked randomly and changed to any other possible value for that gene. Applying a baseline mutation to a parent individual will always result in a mutated individual that has a genotypic distance of 1 from its parent, landing it in the first neighborhood. All other genetic operations can be modeled as functions of the baseline mutation. For example, the result of swapping two genes could be achieved through two coordinate 1-point mutations. The size of the neighborhood determines the number of genes that must be mutated. A single baseline mutation will land the

mutated individual in the first neighborhood, where mutating two genes at random from the genotype will land the mutated individual in the second neighborhood, etc.

Baseline mutation is used to determine the shortest distance between two genotypes. When measuring the distance, the genotypes are compared to assess how many baseline mutations would be needed from the first to reach the second one (and vice-versa). Although the evolutionary process might take detours, when measuring distance only the shortest possible path is of importance.

2.5.2 Baseline evolvability

As implied by their name, evolutionary systems aim at building evolvable models. Interpreting this, evolvability is based on the speed at which any system is able to adapt to a new environment. In other words, *evolvability estimates how fast a genotype that is fit to one environment evolves to a different genotype with a phenotype that is fit to a different environment*.

This means that the measure can be constrained to the same environment or through a changing environment as the starting population of an evolutionary run is always less fit than the target, which implies it would be more adapted to a different environment, like two symbolic regression programs that will behave better or worse than the other one depending on the test cases presented. Translated to genetic algorithms, this means an algorithm with high evolvability will be able to find a fit individual with a minimum number of changes to the initial individual.

The main objections being raised to Wagner's evolvability measures are:

1. There is no evidence that proves *exclusive* dependency of evolvability (of either type) on the diversity of the phenotypes (independently, say, of fitness)
2. The evolution from a starting point to another, in genotype space, usually involves multiple mutations, which may not necessarily occur in *series*. It is possible for several mutations to affect an organism simultaneously. It is also logical that an organism may fail to evolve (to a higher fitness state) even if its mutational neighborhood is carpeted with the most diverse set of phenotypes, all of lower fitness.

3. The phenotype measures proposed by Wagner are often impractical since they require knowledge of the phenotype to genotype map, hence requiring an inverse mapping function (find all the genotypes for a given phenotype).

Therefore, it is proposed here that evolvability (of a $G \rightarrow P$ map) is measured *directly* by assessing the average length of random mutational walks between randomly selected start genotypes and end phenotypes. To be exact, baseline evolvability (E_b) is defined algorithmically as follows:

1. Construct a collection of *m walks* $\{W_1, W_2, .., W_m\}$; a walk is defined by two genotypes (probably, but necessarily different), g_s (start) and g_f (finish) with corresponding phenotypes, p_s and p_f, respectively.
2. A walk W_j goes through a sequence of genotypes $g_s, g_1, .., g_k, .., g_f$, with corresponding phenotypes $p_s, p_1, .., p_k, .., p_f$; the walk could terminate before reaching the final phenotype (p_f).
3. Consider one step of the walk, p_i to p_{i+1}. Single point mutations are applied to the genotype g_i until a new genotype g_{i+1} is found for which $d(p_{i+1}, p_f) \leq d(p_i, p_f)$. That is, the new phenotype is no further from the destination phenotype than the old phenotype; d is an appropriate measure of distance.
4. Continue walk W_j until phenotype p_k. At this point either p_f has been reached or, the distance $d(p_i, p_f)$ has stabilized; stabilization is described below.
5. Let $q = \frac{d(p_k,p_f)}{d(p_s,p_f)}$, and t is the number of steps in the walk. By construction, $d(p_k, p_f) \leq d(p_s, p_f)$ and thus $0 \leq q \leq 1$. If $d(p_s, p_f) = 0$, q is set to 0.
6. The evolvability of the *walk* W_j is E_j, where $E_j = \frac{(1-q)}{(1+t)}$, using $1 + t$ rather than just t in the denominator for the special case where $p_s = p_f$ and $t = 0$. Note that $0 \leq E_j \leq 1$.
7. Continue generating walks in this way until E_b stabilizes. Stabilization is described below. Baseline evolvability E_b is the average evolvability of the walks, E_1 to E_m.

For stabilization, compute the variance V of the last N terms of the series, and assume that stabilization has occurred if $V < e$. Values for N and e are probably best chosen by experiment

(empirically). An in-depth methodology of using the variance as a stopping criterion is described in Bhandari, Murthy & Pal (2012).

When implementing this as an EA, fitness evaluation needs to minimize the distance to the target phenotype. Hence fitness needs to use the same distance measure as D_p.

A simple application of E_b for a single walk W is presented in Table 2.5. Using fictitious RNA sequences and secondary structures to represent genotypes and phenotypes respectively, a random starting genotype, $AUCUAGCG$, with corresponding phenotype (.....), is mutated until a random target phenotype, $p_f = (((..)))$, is reached. The Hamming distance is used to measure the phenotype distances D_p, for both the parent phenotype p_i and offspring phenotype p_{i+1}, at each step t. The gene mutated for each offspring is underlined, and the bolded genotype at each step (between the parent g_i and offspring g_{i+1}) represents the genotype selected for the next generation (shortest distance D_p to the target phenotype).

Table 2.5 Example process of an evolvability walk, reaching the random target phenotype $(((..)))$

Step t	g_i	p_i	$D_p(p_i, p_f)$	g_{i+1}	p_{i+1}	$D_p(p_{i+1}, p_f)$
1	AUCUAGCG	(.....)	4	**A<u>C</u>CUAGCG**	((....))	2
2	**ACCUAGCG**	((....))	2	ACCUAG<u>U</u>G	.....	6
3	ACCUAGCG	((....))	2	**AC<u>U</u>UAGCG**	(((..)))	0

In this example, the walk took only three steps to reach the random target phenotype. Thus, the evolvability of the walk E_w is computed as below:

$$D_p(p_s, p_f) = 4$$

$$D_p(p_k, p_f) = 0$$

$$q = \frac{0}{4}$$

$$E_w = \frac{1 - q}{1 + t} = \frac{1}{4}$$

with a final evolvability of 0.25. When doing multiple walks, average evolvability E_b will tend to stabilize, as illustrated in Figure 2.6.

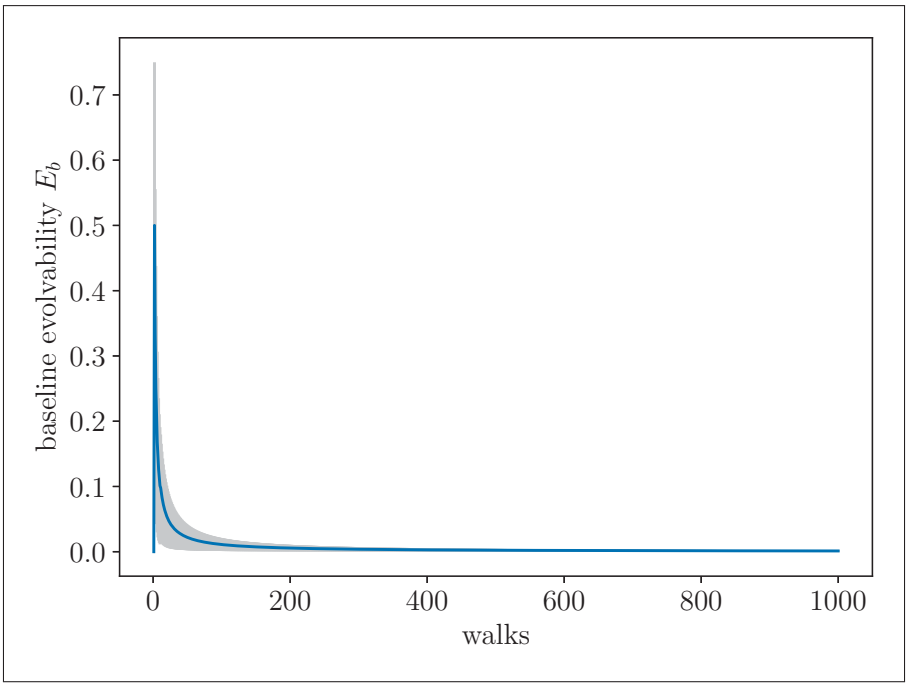


Figure 2.6 Example baseline evolvability plot

2.5.3 Baseline robustness

In the context of evolutionary systems, robustness is defined by the number of changes (in its genotype or environment) a system can sustain before expressing a different behavior. In order to measure that, a system needs to be changed (mutated) incrementally while the genotype and phenotype distances from the starting point are measured.

The main objection being raised to Wagner's robustness measures are that they only look at the immediate mutational neighborhood of a genotype (G), in case of R_g, and the immediate mutational neighborhood of the genotypes mapping to a given phenotype (P), in case of R_p. Both of these measures do not provide the full picture of the relationship between the amount of mutation applied to genotype G (or genotypes mapping to P) and the amount of divergence

from the original phenotype. This is why it is proposed here that one forms a scatter plot that relates the first quantity (amount of mutation) to the other (amount of divergence), and hence, characterize the correlation among the points using confidence ellipses, as shown in Figure 2.7. The orientation Θ of the major ellipse axis is representative of the sensitivity of the robustness, with steeper orientation meaning weaker robustness (lower Θ value means more sensitive), and the aspect ratio α, computed by dividing the major axis with the minor axis, is an indication of the strength of the correlation (higher α value means stronger correlation).

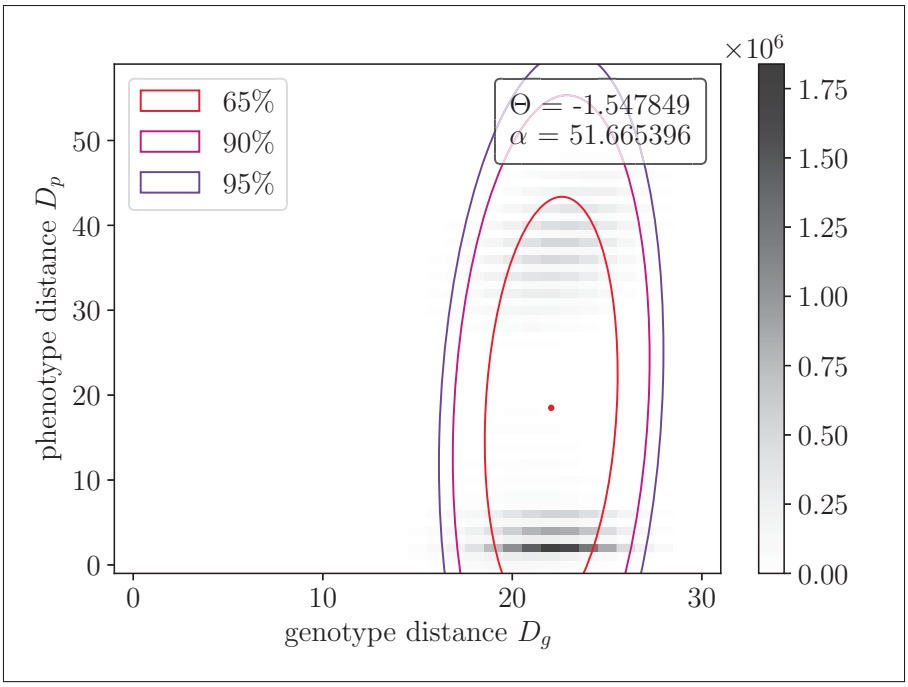


Figure 2.7 Example of baseline robustness plot

Specifically, the set of points between phenotype distance and genotype distance is used as the baseline measure of robustness. It is defined as the set of points $(d_g; d_p)$ in which d_g is a genotype distance (D_g) and d_p is the corresponding phenotype distance (D_p). I.e., Baseline robustness, R_b is:

Scatter plot $\{(D_g(g, g'), D_p(F(g), F(g')))\}$

Where g, g' belong to G and F maps any g to its phenotype p.

In practice, the scatter plot is characterized using a covariance ellipse. The complete set of points would be overwhelming. Fortunately, there are two obvious methods to sample the scatter plot.

1. Choose two random genotypes g_1 and g_2, compute their phenotypes, p_1 and p_2, and plot the pair $(D_g(g_1, g_2), D_p(p_1, p_2))$. The Hamming distance is used for both D_g and D_p.
2. Construct a genotype walk g_1, g_2, .., g_n with $D_g(g_i, g_{i+1}) = 1$ and the corresponding phenotype walk p_1, p_2, .., p_n and plot the pairs $(D_g(g_1, g_i), D_p(p_1, p_i))$ for $i = 1, 2, .., n$.

The distance $D_g(g_1, g_2)$ for random genotypes g_1 and g_2 will typically be large. Since robustness is mainly about small deviations from an initial genotype, the second of these two methods may be preferable in practical cases.

Although baseline robustness is best appreciated as a scatter plot, some metrics can help quantify its behaviour. The main indication of robustness is the orientation of the eigenvector corresponding to the largest eigenvalue, both obtained from the covariance matrix. A steeper angle means lower robustness. The equation for orientation Θ is shown in Equation 2.20, where $\mathbf{v}_1$ is the eigenvector corresponding to the largest eigenvalue.

$$\Theta = \arctan2\left(\frac{\mathbf{v}_1(y)}{\mathbf{v}_1(x)}\right) \tag{2.20}$$

In addition to orientation, a measure of the spread of points, around the major axis (eigenvector of the largest eigenvalue), would complement the measure of orientation of that vector (Θ). For this, the aspect ratio α is used, which also comes from the eigenvalues of the data. To compute α, divide the largest eigenvalue λ_1 by the smallest eigenvalue λ_2, as shown in Equation 2.21.

$$\alpha = \frac{\lambda_1}{\lambda_2} \tag{2.21}$$

To ease the assessment of the baseline robustness scatter plots, three confidence ellipses (65%, 90% and 95%) are laid on top of the data. The ellipses are positioned at the center of mass of the data with angle Θ. Assuming normal distribution, the Chi-Squared likelihood s associated

with the confidence interval of the ellipse can be used to compute the length of the major axis γ_1 and minor axis γ_2, as shown in equations 2.22 and 2.23. For additional details on how to define a confidence ellipse, readers are referred to Spruyt (2014).

$$\gamma_1 = 2\sqrt{s\lambda_1} \tag{2.22}$$

$$\gamma_2 = 2\sqrt{s\lambda_2} \tag{2.23}$$

2.6 High-level setup

The experimentation was done using the ECJ library from Luke (1998) which provides implementation of various Evolutionary Computation algorithms along with common problems used for benchmarking. Since the goal of this research is to assess different measures of evolvability and robustness of an evolutionary system, it is more efficient to leverage a state-of-the-art EC implementation rather then building a new one. This ensures the focus is kept on the measurement and reduces the risk of introducing new bugs in the software. A high-level view of the architecture is presented in Figure 2.8 and detailed applications of ECJ will be outlined for each of the experiments in the following sections. Additionally, the experiments were run on compute clusters provided by the Digital Research Alliance of Canada. The program used to execute the experiments is available through the author's public code repository[2].

ECJ's easy extensibility and extensive documentation facilitated the experimental setup for all the following experiments. The neighborhood exploration and random walks described in the definition of the measures can be achieved with a simple EA implementation, akin to a 1+1 EA. The following section describes the generic components of the EA used to conduct the experiments.

[2] The code repository can be accessed at https://github.com/remz1337/VRE_Experiment

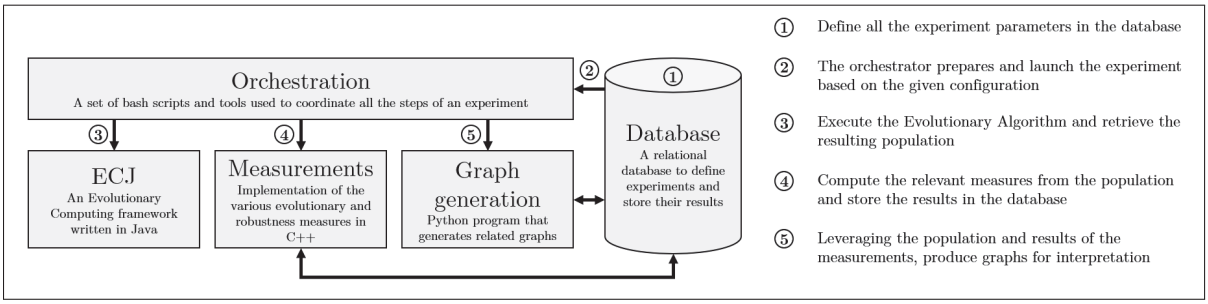


Figure 2.8 High-level architecture of the experimentation software

2.6.1 Evolutionary Algorithm

A high-level **flow chart** of the EA is followed by separate sub-sections that describe every component of the EA; this includes: representation and initialization, fitness evaluation, parent selection, offspring generation (*via* mutation which requires fitness evaluation of new individuals), survivor selection and termination criteria.

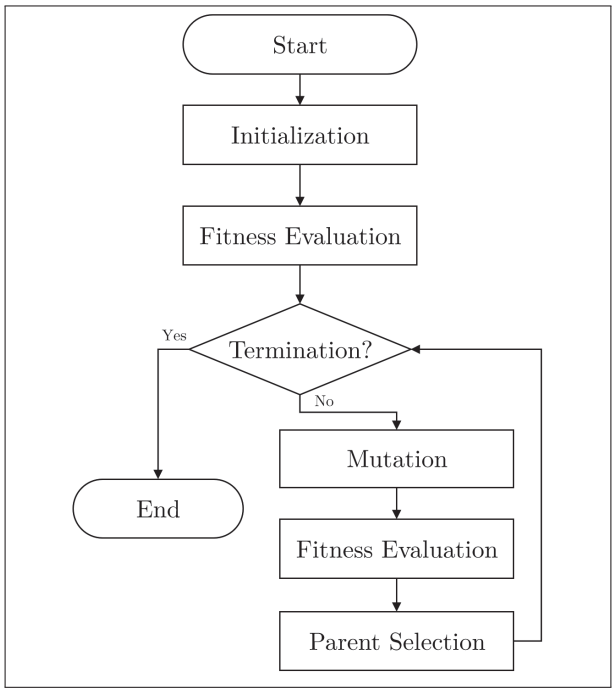


Figure 2.9 Overall structure of the EA

Baseline measures build random walk by initializing a random individual and applying baseline mutation for a number of generations, or until the target is reached in the case of E_b. The only difference is that E_b compares the offspring to the parent, while R_b always keeps the offspring.

This EA can also be adapted to do the neighborhood exploration needed for the classical measures. The population starts with μ random individual from which λ offspring are generated through mutation for each individual in the starting population. Running this for a single generation and keeping the offspring yields a resulting population that represents the neighborhood. Then the classical measures can be evaluated by post-processing the resulting population, as defined in Section 2.4.2. Using this configuration, λ needs to be set to the number of samples to be taken from the neighborhood N (N also dictates the number of genes mutated).

2.6.1.1 Initialization

Random initialization, respecting the range of possible values for a given gene.

2.6.1.2 Termination

Termination occurs when a pre-set maximum number of generations is exceeded or, in the case of E_b, the EA finds the target phenotype (with ideal fitness).

2.6.1.3 Parent selection

There is no parent selection, as the parents comprise the whole current population.

2.6.1.4 Mutation

Pick n (where n is the size of the neighborhood, $n = 1$ for baseline measures) genes randomly from the genotype and change its/their value(s) randomly to another possible value for that gene.

2.6.1.5 Survivor selection

Survivor selection here means the compilation of the next generation. In the case of E_b, it is done by comparing the offspring to its parent and keeping the fittest of them, or the offspring in case of equal fitness. For R_b or the neighborhood exploration of the classical measure, the offspring are always kept.

2.6.1.6 Fitness Evaluation

For R_b and the neighborhood exploration, there is no need for fitness evaluation since the individuals are not compared (fitness can be set to an arbitrary value). For E_b, the fitness is the phenotype distance D_p to the random target phenotype, which needs to be minimized. As will be demonstrated later, E_b can also be adapted for specific problems, in which cases the fitness evaluation is tailored for the problem to be solved. These occurrences will be defined along their problem definition.

CHAPTER 3

VALIDATION OF THE MEASURES TO SIMPLE RNA SEQUENCE FOLDING

The goal of this first experiment is to compare the applications of all reviewed and proposed measures and assess how they individually can provide insight with respect to evolvability and robustness. The application to a different domain than traditional computer science is also a good demonstration of the potential extensibility of the measures.

RNA is similar to DNA, where protein bases are represented by various characters of the alphabet. To understand how an RNA sequence behaves, it needs to be "folded" to obtain its secondary structure. The process of folding tries to identify the most likely way the bases will pair together in nature. Protein folding has been the subject of many studies, but any review goes outside the scope of this work. Evolvability and robustness will be measured on RNA sequence folding, using the RNA sequences as genotypes and secondary structures as phenotypes. RNA sequences are generated randomly using the four bases (or nucleotides), `AUCG`, and then evaluated against a folding algorithm to obtain their secondary structures, represented using the Dot-Bracket Notation format. The `ViennaRNA` suite of RNA folding tools was selected for its popularity and ease of use. A detailed description of the tools and their recent evolution can be found in Lorenz *et al.* (2011). Readers interested in comparing various RNA sequence folding tools are referred to Churkin *et al.* (2018).

For example, `AUUUGCAAAUGGCAACCAUUGGGUGUGAGU` is an RNA sequence of 30 bases, and `(((((((((((...)))))...)))))))` is its secondary structure in Dot-Bracket Notation, which would be represented using base pair probabilities like shown in Figure 3.1[1].

3.1 Experimental setup

This section will detail how the experimentation for the RNA sequence folding problem was conducted for both the classical and new baseline measures.

[1] Image generated using the `RNAFold` web service, found at http://rna.tbi.univie.ac.at/cgi-bin/RNAWebSuite/RNAfold.cgi

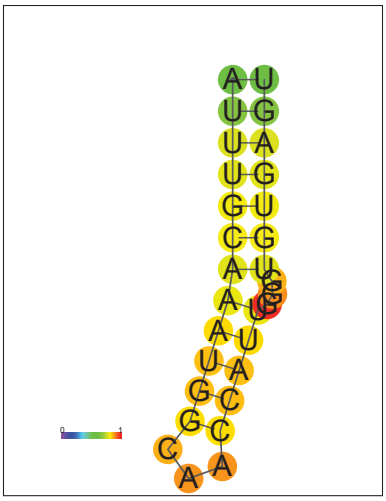


Figure 3.1 An example representation of RNA sequence folding

3.1.1 Distance measurement between two individuals

Hamming distance for both genotype and phenotype as they are both well-defined and fixed length strings of characters.

3.1.2 Folding and inverse folding RNA sequences and secondary structures

As the classical genotype measures and baseline measures require a mapping of genotype to phenotype ($\Gamma(g) \rightarrow p$), `RNAsubopt` utility from the `ViennaRNA` package will do the work of folding RNA sequences into secondary structures. Any random RNA sequence can be folded in the sense that the folder will return a secondary structure. However, secondary structures exclusively comprised of dots are discarded as they mean the folding did not find a suitable solution. `RNAsubopt` is used with the default configuration[2] and sorted by free energy to provide the best solution first.

[2] The default configuration of `RNAsubopt` includes using linear MFE structures without structural constraints at 37 degrees centigrade. Complete configuration can be found at https://www.tbi.univie.ac.at/RNA/RNAsubopt.1.html.

The classical phenotype measures require an inverse mapping function to identify the entire set of genotypes that can map to a given phenotype. There is currently no known method to achieve this, but the results can be approximated by sampling $\Gamma(g) \rightarrow p$. Secondary structures in dot notation are sequences of the same length as their source RNA sequence(s) that uses three characters: periods, opening parentheses and closing parentheses. However, any arrangement of periods and parentheses does not mean it is a valid secondary structure. A set of rules also need to be applied, such as matching the opening parentheses with closing parentheses in the right order. Yet, a secondary structure may appear to be valid but does not have any genotype that can fold into it. For this reason, the set of random phenotypes is obtained by folding random RNA sequences and keeping every valid and unique phenotype found in the process. This can be processed ahead of the experiment.

`ViennaRNA` offers a tool to explore the phenotype to genotype map, namely `RNAInverse`, which can search for multiple RNA sequences mapping to the provided secondary structure. Additional constraints can be passed to this inverse folding search algorithm, but are not used in this experiment. With the set of valid secondary structure in hand, `RNAInverse` will be used to search for up to 50 mapping genotypes for each provided phenotype. Sometimes, the inverse folding will provide RNA structures that can fold into the given phenotype, but would naturally (using less energy) fold into a different secondary structure. For this reason, the set of genotype obtained from `RNAInverse` is folded back again and any genotype that did not fold back into the original phenotype is discarded. This provides a sampling of the phenotype to genotype map, allowing to compute the classical phenotype measures. This process is shown in Figure 3.2, using fictitious RNA sequences and secondary structures.

The choice of setting a maximum of 50 genotypes that can be returned by the `RNAInverse` program was defined empirically through trial and error. Figure 3.3 shows the distribution of 1,000 random RNA secondary structure (100 bases) processed with `RNAInverse`, returning an average of ≈ 17 genotypes found per phenotype. This does not mean that more cannot be found, but it becomes computationally more expensive to continue searching. It is assumed that the

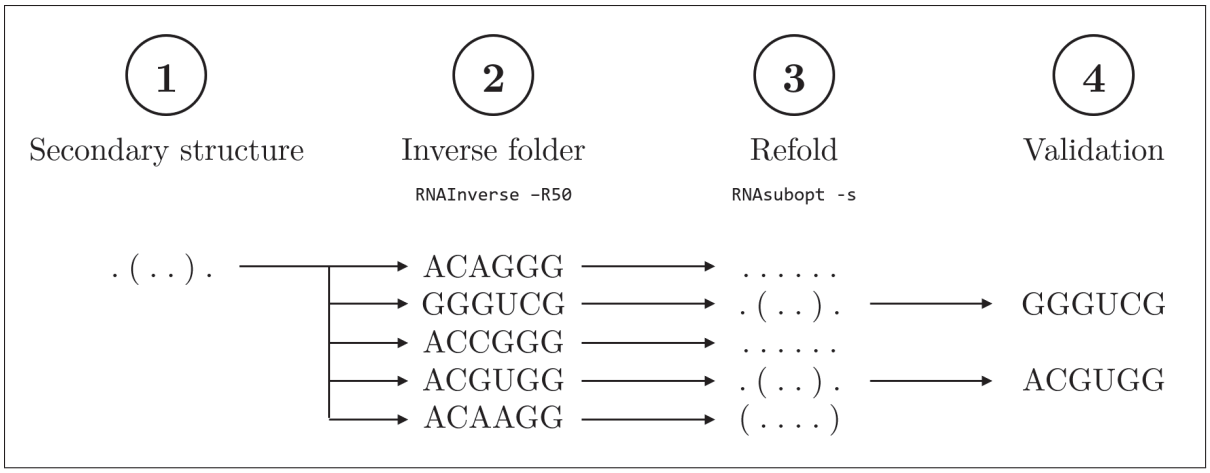


Figure 3.2 Process used to build the $P \rightarrow G$ map through inverse folding validation

entire phenotype to genotype map for a given phenotype is proportional to the results of the inverse folder.

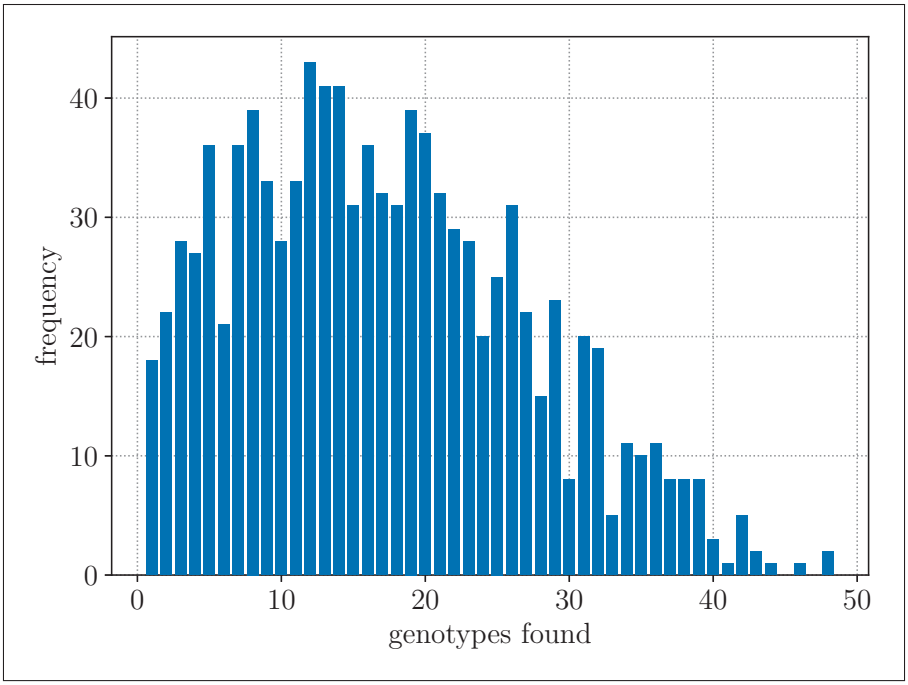


Figure 3.3 The distribution of valid genotypes found through inverse folding

3.2 Application of generalized classical measures

The goal of this section is to demonstrate the generalization of Wagner's measures. For this, only the first five neighborhoods ($N = 1, 2, 3, 4, 5$) will be measured. Since the search space grows exponentially with N, sampling becomes insignificant with higher neighborhoods. Also, the objective is to understand what information was missed in Wagner's original measure, which can be assessed with the first five neighborhoods.

Since the neighborhood distance N controls the number of mutated genes, the search space can be easily computed with $M^N \binom{L}{N}$, where M is the number of possible mutations (in this case, each RNA base has only three possible mutations), N is the neighborhood (number of mutated genes) and L is the length of the genotype (number of RNA bases). Figure 3.4 presents the size of the search spaces on a logarithmic scale for the first ten neighborhoods of RNA sequences of lengths 30, 50 and 100. Although the length of the RNA sequence ($L = [30, 50, 100]$) has an impact on the search space, it is negligible compared to the choice of neighborhood ($N = [1 \ldots 10]$).

3.2.1 Neighborhood size

As presented in Figure 3.4, the neighborhood sizes increase exponentially as the neighborhood distance N is increased. Following the original definition of the measures defined in Wagner (2008), the ideal measurement would look at the entire search space. This is possible when $N = 1$, but becomes computationally expensive when $N > 2$. To address this, sampling of the search space will be leveraged. It would be fair to assume that the bigger the sample the more accurate the results will be, however the increase in accuracy often diminishes as the sample size grows. This was demonstrated in Taherdoost (2017), which will be applied to calculate the sample size for each neighborhood distances, using a population variance of 50%, confidence level of 99% and error margin of 1%. The sample sizes are presented in Table 3.1.

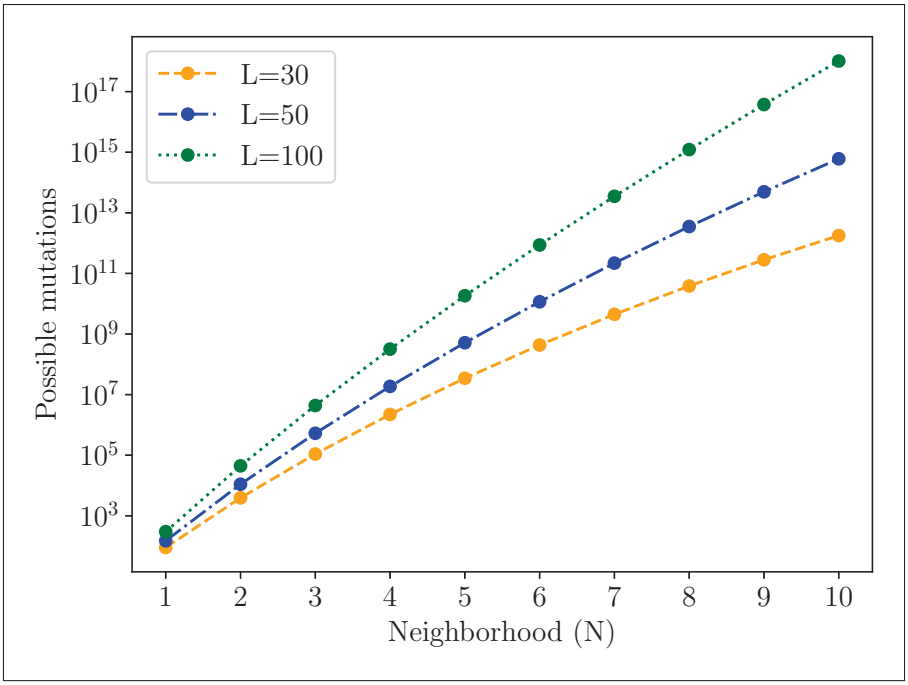


Figure 3.4 Figure representing the expansion of the search space for the RNA problems of varying lengths across the first ten neighborhoods

Table 3.1 Sample size for each neighborhood distance N, along with the provided coverage of the search space

N	Sample size	Search space	Coverage
1	295	300	≈ 0.9833
2	12,048	44,550	≈ 0.2704
3	16,451	4,365,900	≈ 0.0037
4	16,512	317,619,225	≈ 5.19e−5
5	16,513	18,294,867,360	≈ 9.02e−7

3.2.2 Population size

With the neighborhood sampling set for each single random starting individual, it is left to define the number of individuals needed to get representative results from the classical measures. This will be achieved by assessing the number needed for the measures to stabilize. Stabilization can

be computed as described in the definition of E_b in Section 2.5.2. This provides a repeatable way to address stabilization, however it requires defining additional hyper-parameters for the threshold and number of terms to include in the variance calculation. It can also lead to bias when comparing the performance of two algorithms, because the better one is likely to stabilize sooner. The bias happens when walks are not able to reach the target phenotype within the given stabilization period, because the walk that goes on for longer will end with a worse E_b. To address this, the stabilization point can be approximated by visually inspecting the change in variance of the results and determining a stabilization period applicable to all reviewed algorithms. For this assessment, the number of walks has been set to 1,000. The results are shown in Figure 3.5 where stabilization of the variance can be seen happening for all four measures within approximately 500 walks, so 1,000 walks is sufficient to get representative results.

3.2.3 Results

The generalized measures can now be evaluated with the parameters defined above. The results of the measures across the first five neighborhoods with RNA sequences of 100 bases are presented in Figure 3.6. As expected, the generalization to higher order neighborhoods yields different results than the first neighborhood. Interesting is the fact that each walk behaves the same (relative to the other walks) across the neighborhoods, either performing better or worse throughout the experiment. This hints that a random genotype that performs well in a single neighborhood will also perform well in all neighborhoods for a given measure. This can be seen when there is a sharp change happening at a specific walk for all four measures.

Comparing the evolvability measures E_g and E_p against the robustness measures R_g and R_p, higher neighborhoods provide better evolvability but worse robustness. This goes against Wagner's original assessment that robustness and evolvability are positively correlated in phenotype space. For the RNA problem, the neighborhood does not impact the phenotype evolvability, where each neighborhood was able to find, proportionally to its size, the same amount of unique phenotypes.

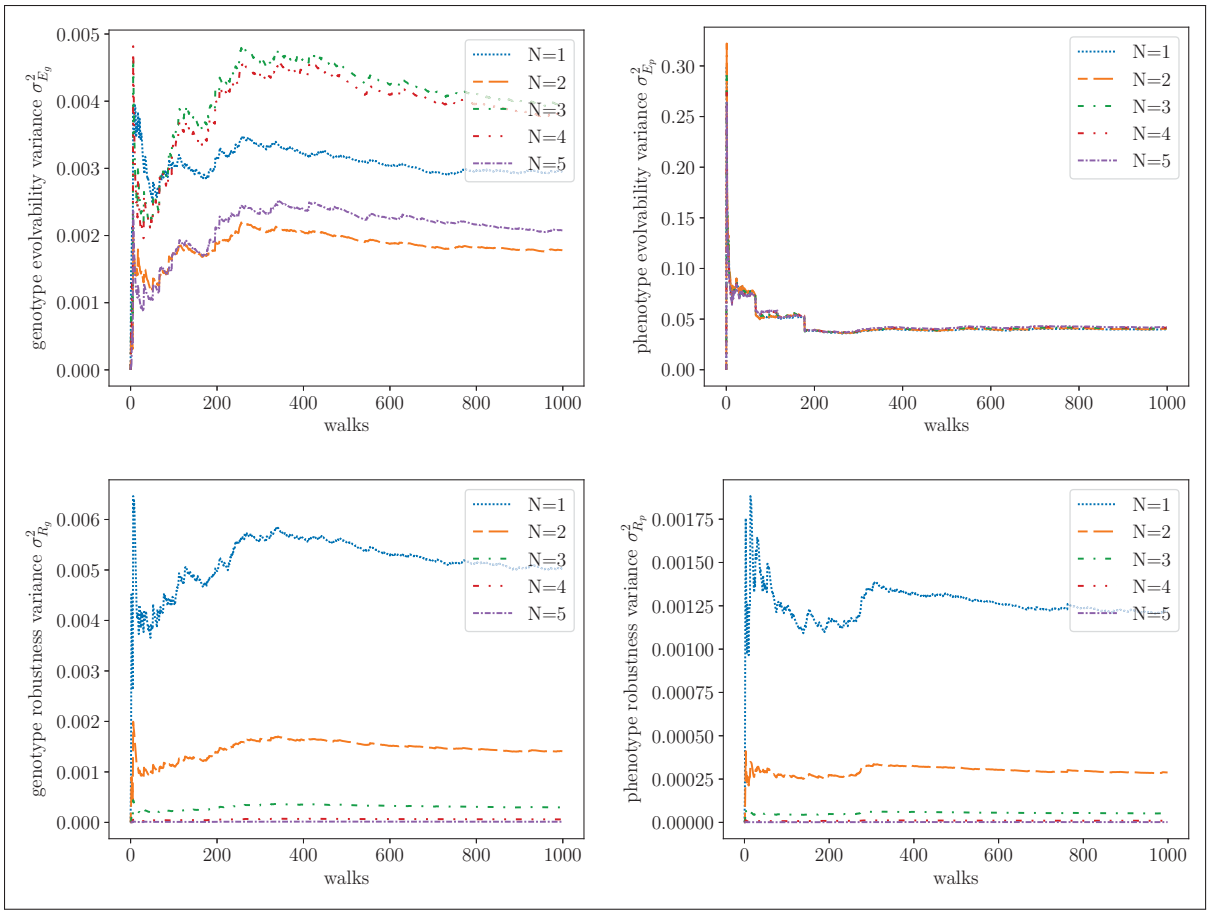


Figure 3.5 Stabilization of the average variance for the four classical measures, across the first five neighborhoods N for RNA sequences of length 100

These results show that the original definitions of robustness and evolvability proposed by Wagner are missing crucial features of a problem representation by limiting its exploration to the first neighborhood. However, generalizing the measures to include further neighborhoods brings additional challenges, mainly the computational effort required to measure each neighborhood and most notably the difficulty of building an inverse mapping function of the $P \rightarrow G$ map.

3.3 Application of baseline measures

Following the demonstration of the generalized classical measure, this section will show how the new baseline measures can be applied. For baseline measures, the number of steps (generations

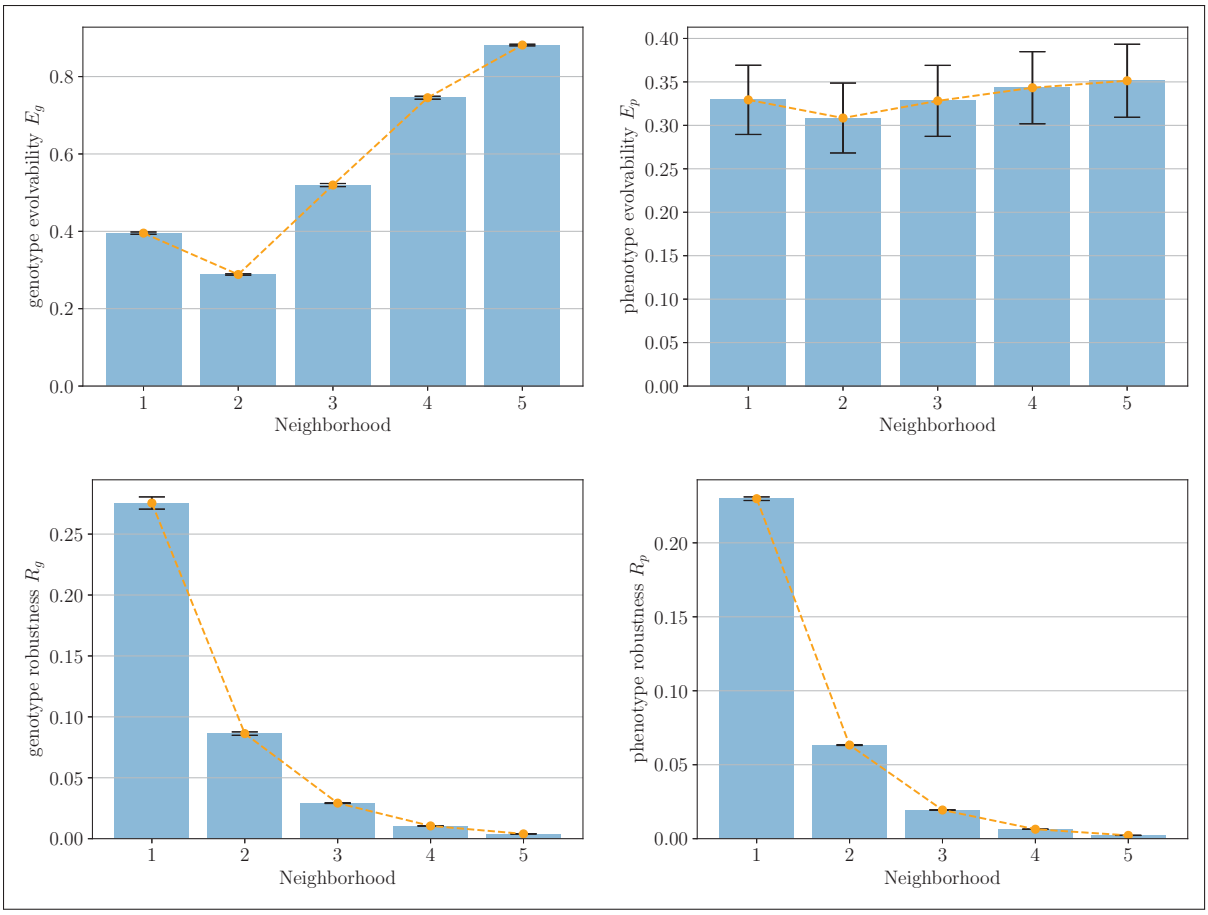


Figure 3.6 Results for the four measures across the first five neighborhoods N for RNA sequences of length 100

in terms of EA) per walk needs to be defined. This parameter offers similar control on the experiment as the sample size of the neighborhood in classical measures in the sense that they both control the total number of offspring that will be generated in the walk from a single starting individual. Like for the classical measures, the number of walks (samples) also needs to be set, which controls the same thing (the amount of random starting individuals).

Ideally, both the number of steps and the number of walks would go until stabilization, as defined earlier. However, for practical reasons, these parameters will be empirically defined by running the experiment with large enough number to assess through visual inspection of the results when the variance stabilizes. If the visual inspection is inconclusive, the experiment is run again with

a larger number. This has not been an issue with the following experiments as they tend to stabilize quickly. In all cases, the values fixed for these parameters have been very conservatives.

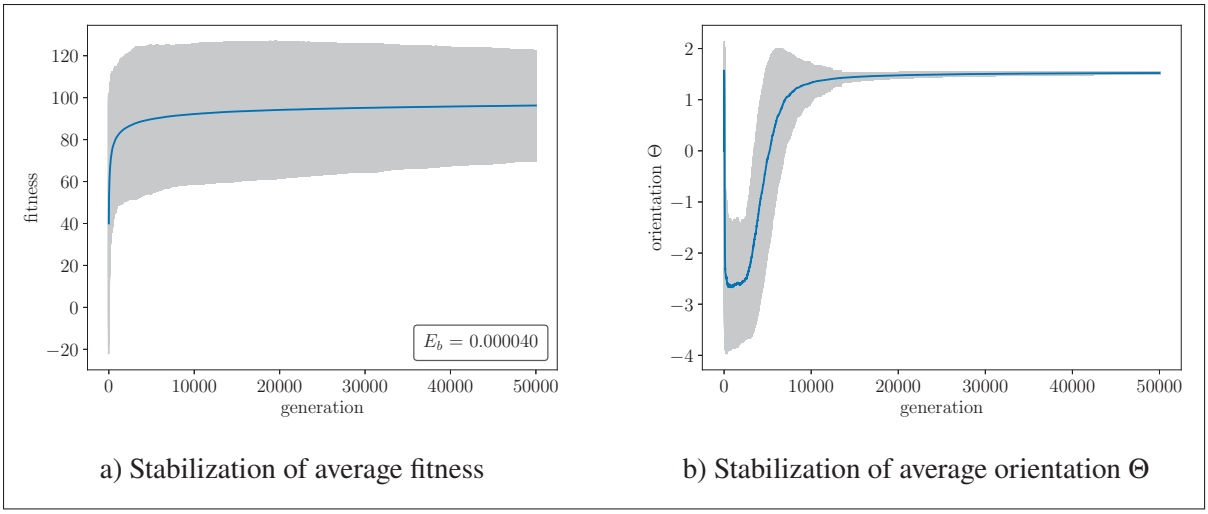
3.3.1 Walk length

To empirically assess the average number of steps needed before stabilization, a first experiment of baseline evolvability E_b will be ran, consisting of 1,000 walks of 50,000 steps. Fitness is the number of characters in the secondary structure that match the random target phenotype. Ideal fitness will always be equal to the length of the RNA sequence, hence 100 in this experiment. Figure 3.7a shows that for RNA sequences of 100 bases, the fitness of the population stabilizes within approximately 30,000 steps. Defining a maximum number of steps has a direct impact on the results of baseline evolvability only in the case where the target is not reached in time. Hypothetically, if a walk needs an infinite number of steps to reach its random target, E_b would tend to 0.

The same parameters (1,000 walks of 50,000 steps) are used to determine stabilization of baseline robustness R_b by looking at the orientation Θ of the ellipse generated from the data. Figure 3.7b shows that Θ also stabilizes within 30,000 steps. The orientation undergoes some drastic changes during the first few generations because the distances are changing rapidly with each mutation, and the ellipse is much more biased with only a few points.

3.3.2 Population size

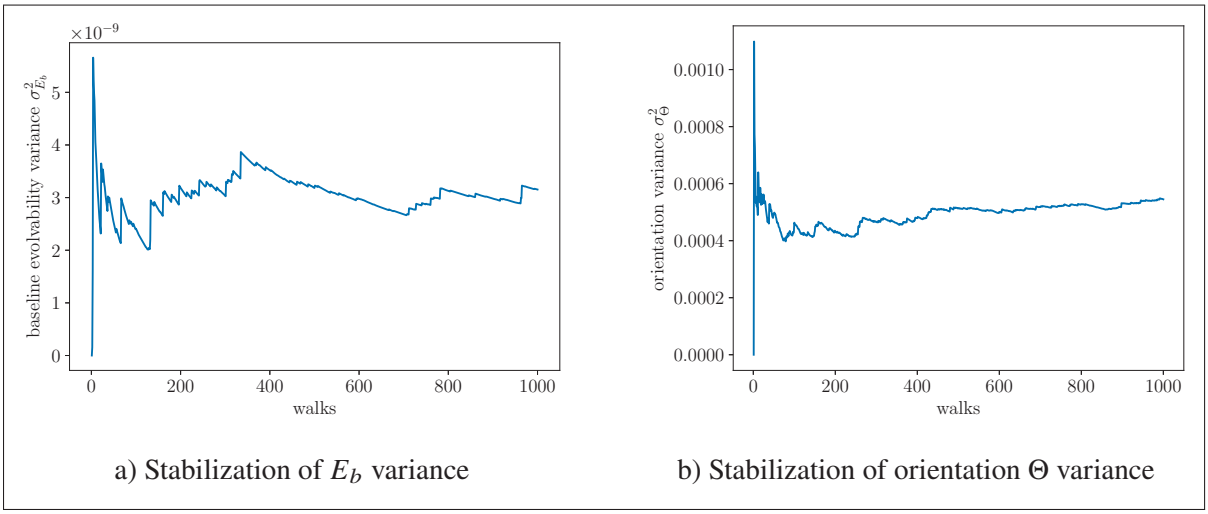
Like for the classical measures, 1,000 walks of 50,000 steps is enough to assess the stabilization of the population size (each walk is an individual of the population). Figure 3.8a shows the average baseline evolvability variance. The variance is so low ($\approx$ 3e$-$9) that it will be prone to any slight change of E_b. Nonetheless, it can be considered stable within about 700 walks. Stabilization for baseline robustness is presented in Figure 3.8b by tracking the variance of Θ. Stabilization happens within 500 walks in the case of R_b.



a) Stabilization of average fitness

b) Stabilization of average orientation Θ

Figure 3.7 Stabilization point of the number of generations for E_b and R_b



a) Stabilization of E_b variance

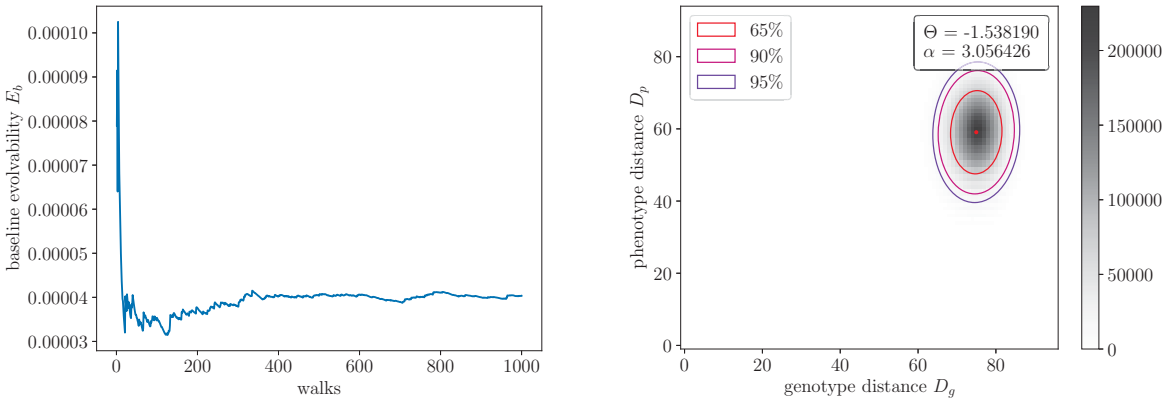
b) Stabilization of orientation Θ variance

Figure 3.8 Stabilization point of population size for E_b and R_b

3.3.3 Results

Evaluating the RNA folding problem ($L = 100$) with the baseline measures using 1,000 walks of 50,000 steps, it obtains a baseline evolvability E_b of $\approx$ 4e−5. Baselines robustness is measured with the orientation Θ and aspect ratio α, which have respective values of ≈ -1.5381 and ≈ 3.056. Both results are presented in Figure 3.9. Interpreting the scatter plot of R_b (Figure 3.9b), the orientation is steep and the center of mass is almost at its maximum distance (100

for both axis), indicating overall poor robustness. Although E_b may seem low, the best way to assess evolvability is by comparing with another algorithm using the same parameters (number of walks and steps per walk). This will be demonstrated later in Chapter 5.



a) Progression of average baseline evolvability for the RNA folding experiment

b) Scatter plot representing the baseline robustness of the RNA folding experiment

Figure 3.9 Results of the baseline measures for the RNA folding experiment

As opposed to the classical measures, the baseline measures can capture all neighborhoods, and mimic the natural process of incremental change, as is often used in GAs. This section presented a successful application of the baseline measures, showing that it integrates easily with common genetic algorithms. The next section will be a direct comparison of both classical and baseline measures.

3.4 Comparison of classical and baseline measures

The goal of this section is to compare the classical measures with the baseline measure and assess if there is any correlation. For measuring E_g, 100 random RNA sequences of 100 bases ($L = 100$) were mutated 100 times to explore their first neighborhood, then E_g was measured for each one of them, by counting the number of unique phenotypes found, as described in Equation 2.16. The distribution of the results, grouped in ten uniform bins based on the minimum and maximum values obtained, are shown in Figure 3.11, with a mean value of ≈ 0.5431.

This indicates that approximately 54 unique phenotypes were found in the first neighborhoods (among the 100 mutations) of each RNA sequence. Figure 3.10[3] is an illustration of the unique phenotypes found in the first neighborhood of a random RNA sequence, where the initial random RNA sequence is circled and P1, P2 and P3 are the three phenotypes found across six different mutations of the initial genotype.

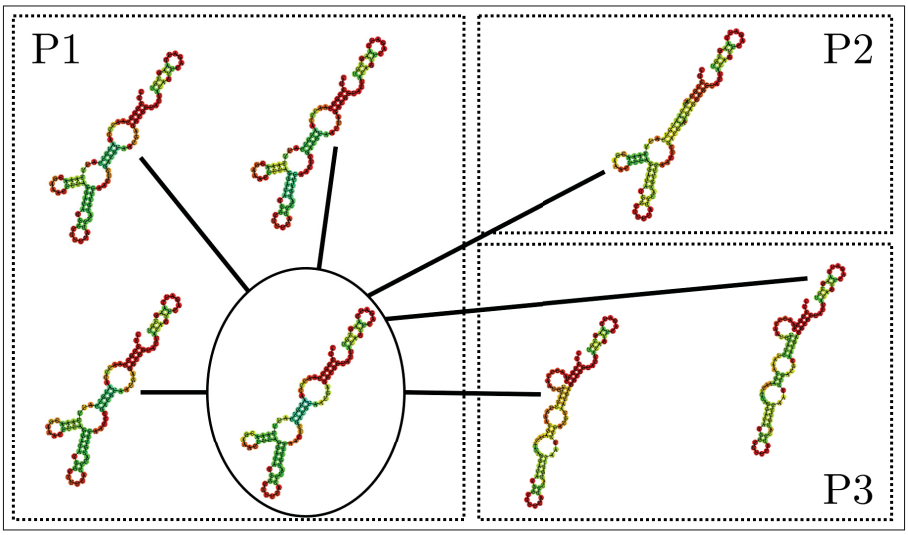


Figure 3.10 Illustration of the unique phenotypes found in the first neighborhood of a random RNA sequence

Table 3.2 EA Parameters of the measure comparison experiment

Parameter	Setup
RNA length	100 bases
RNA bases	AUCG
Initialization	Random
Genetic operators	1-point mutation
Population	100
Generations	100
Fitness	Hamming

[3] The images have been generated from the online `RNAFold` web service, available at http://rna.tbi.univie.ac.at//cgi-bin/RNAWebSuite/RNAfold.cgi.

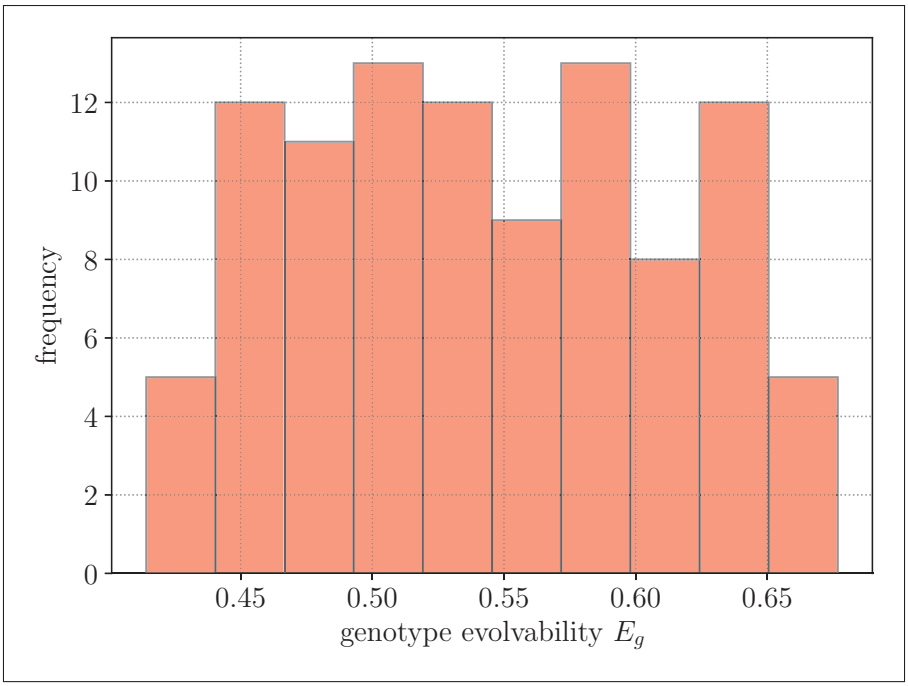


Figure 3.11 Distribution of genotype evolvability, as per Wagner's definition

Comparing this with E_b, the same starting RNA sequences used to measure E_g were mutated for 100 generations and each mutated RNA secondary structure was compared against its parent to see which one is closer to the target phenotype. The target phenotypes were obtained by folding different random RNA sequences, ensuring validity and uniqueness. The one with the best fitness would be selected for the next generation, as described in the E_b algorithm of Section 2.5.2. Fitness evaluation is the Hamming distance between the individual's secondary structure and the random target secondary structure. Figure 3.12 is a scatter plot of the 100 starting RNA sequences, used in both E_g and E_b, positioned according to their resulting E_g and E_b value.

The lack of correlation is an indication that the first neighborhood is not sufficient to assess the evolvability of a population.

Lastly, the robustness measures were evaluated using R_b with the same starting RNA sequences. The starting population is mutated for 100 generations, without fitness evaluation (always

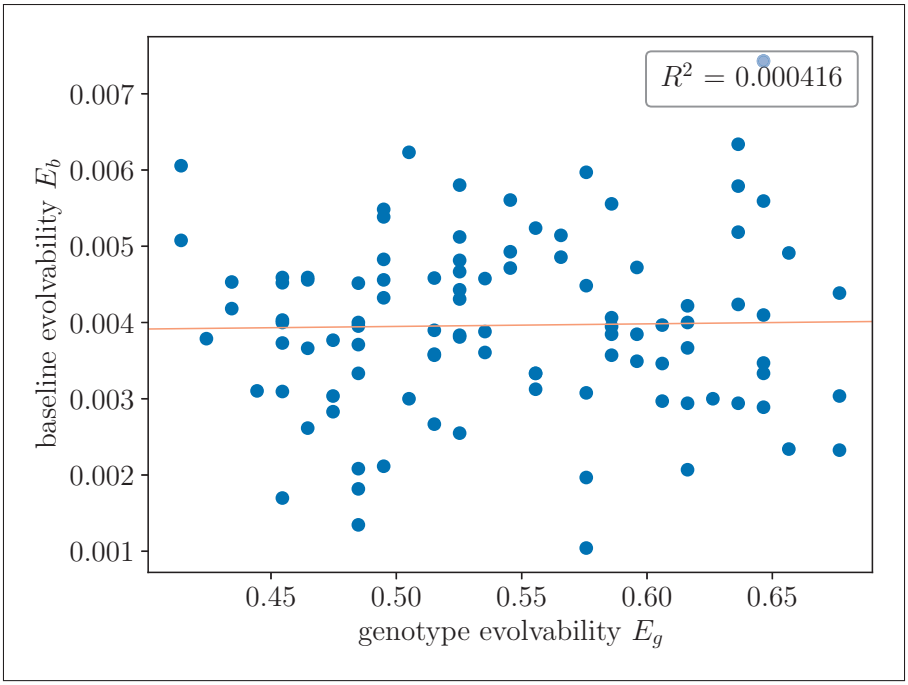


Figure 3.12 Comparison between genotype evolvability E_g of the first neighborhood and baseline evolvability E_b

keeping the offspring). The resulting distribution is presented in Figure 3.13 in which can be observed that lower genotype distances from the starting RNA sequence produce secondary structures that are closer to the original one. Darker areas mean that more points occur at that location, following the scale on the right.

The region outlined in green in Figure 3.13 highlights the distribution of phenotype distances found in the first neighborhood ($D_g = 1$), which is equivalent to the classical measure R_g. It is obvious that limiting the measure to the first neighborhood would present only a very limited slice of the complete picture of robustness, otherwise presented by the whole plot.

As demonstrated in the previous experiments, the proposed baseline measures offer more depth to the evolvability and robustness assessment while also being more practical, as opposed to the classical measures. For this reason, the focus will be kept on the new baseline measures for the following chapters, aiming to further validate the baseline measures with various problems.

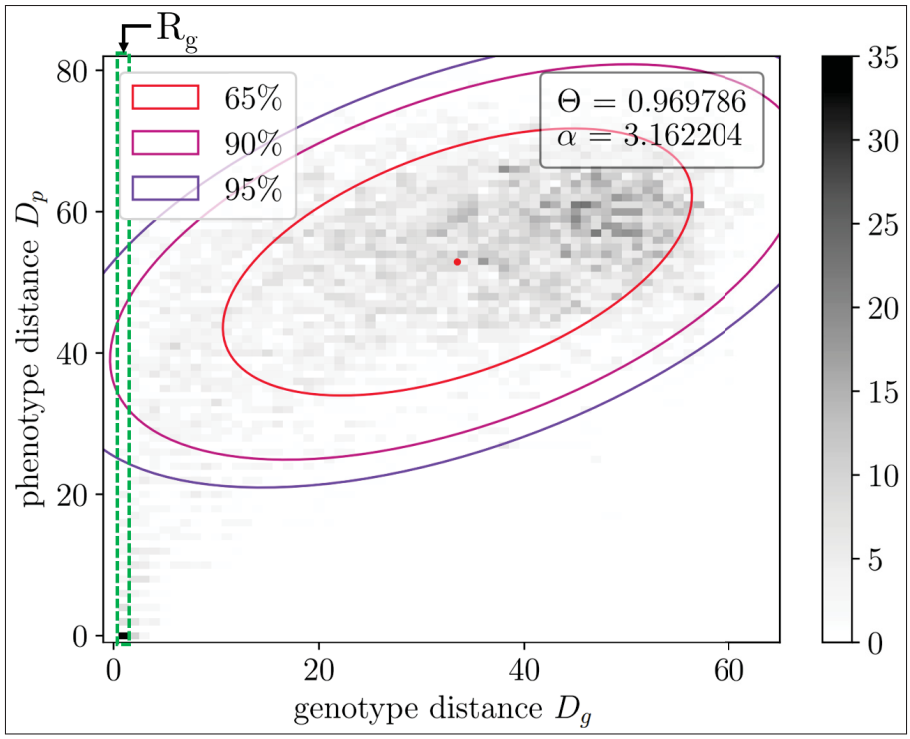


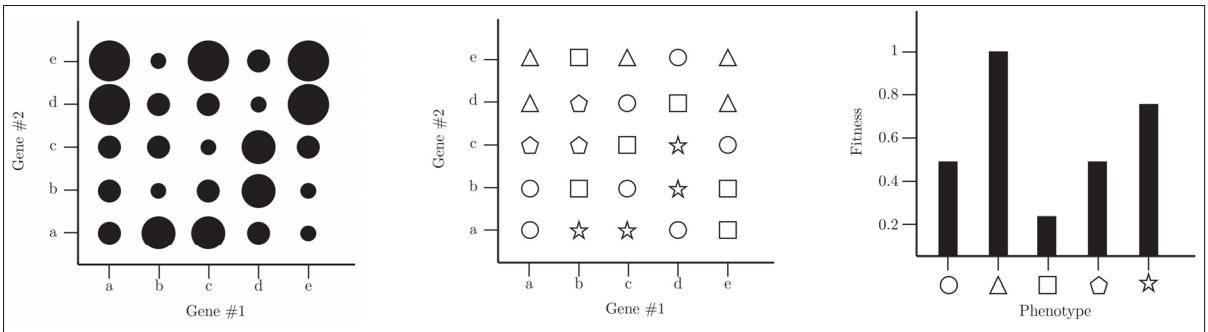
Figure 3.13 Baseline robustness for 100 random RNA sequences

In this chapter, new and modified measures of evolvability and robustness were applied on RNA folding. Focusing on the new baseline measures, the next chapter will continue applying the improved measures to different types of systems in order to further validate the claim that these measures are applicable to a wide variety of systems. Using a modeling method as a target, it is possible to apply the measures to models of systems from different domains of application (though with different levels of fidelity).

CHAPTER 4

EVOLVING OSCILLATING CIRCUIT

Looking for a generic modeling method that allows for the modeling of a variety of real-world systems, but one that also allows the modeler to increase the complexity (interpreted as multi-modality) of the system's fitness surface(s) easily, NK System (Kauffman (1992)) appeared as an attractive approach, but for one critical deficiency: it maps the genotype of a system directly to a fitness surface, without first generating any phenotype. In response, BNK System (short for Binary NetworK) is proposed as a new modeling method which has a genotype mapped to a phenotype, which in turn can be evaluated using any number of different fitness functions, depending on the quality (or qualities) of interest to the modeler. Further investigation will show that, at least in one case, the multi-modality of a BNK system increases with increasing k, where k in a BNK system is the number of inputs to each node (this is detailed below). See Figure 4.1 for illustrative examples of mappings used by NK and BNK. Subfigure 4.1a shows the genotype to fitness map used by NK, where the axes represent the design variables (genes in the genotype), mapped to fitness points (bigger points represent better fitness). Subfigures 4.1b and 4.1c present the genotype to phenotype and phenotype to fitness mappings respectively, where the five shapes represent different phenotypes.



a) Genotype to fitness map b) Genotype to phenotype map c) Phenotype to fitness map

Figure 4.1 Comparison of genotype to fitness mapping between NK and BNK

4.1 BNK: Genotype and Phenotype

As an example, **BNK(n,k)** systems with, say, $n = 3$ and $k = 2$, has exactly three nodes, with every node accepting *between* one and two distinct inputs. An input to a node comes from the output (= current state) of a node, including itself. This is the general form of a BNK model, and it is the one adopted for the two examples of this section.

4.1.1 BNK Genotype

A BNK system consists of a system diagram (nodes with connections) along with the truth tables of each node. A graphical representation is great for human readability but poorly suited for computer manipulation. To facilitate computer manipulations of a BNK system, its features are encoded in a genotype. The gates are encoded in ascending order. And, each gate comprises the decimal indexes of its input gates (e.g., G_0, G_1), followed by the binary output of its truth table (e.g., '0001'). The additions of delimiters and conversion of binary values to their decimal equivalents are done to produce a linear textual representation of the genotype, more compact and appropriate for computer processing. For the textual representation, the input gates (connectivity) are separated by commas (,). A colon (:) separates the connectivity from the functionality (function of the gate) of each gate. Functionality is represented using the decimal value of the output of the truth table. The encoding of every gate concludes with a semi-colon (;). An example BNK(3,2) genotype is presented in Figure 4.2. In Figure 4.2d, the first part ("1,2:1;") represents the encoding of the first gate (G_0), where $1, 2$ come from the gate number of the connected gates and the 1 after the colon comes from the decimal representation of the output column of G_0's truth table (shown in Figure 4.2b).

4.1.2 BNK Phenotype

The phenotype of a BNK system represents the **structure** of the system, comprehensively captured by a state transition diagram (STD). This STD is computed from the genotype. All states in a BNK system lead to other states or themselves, thus leading to *loops* and *fixed points* in

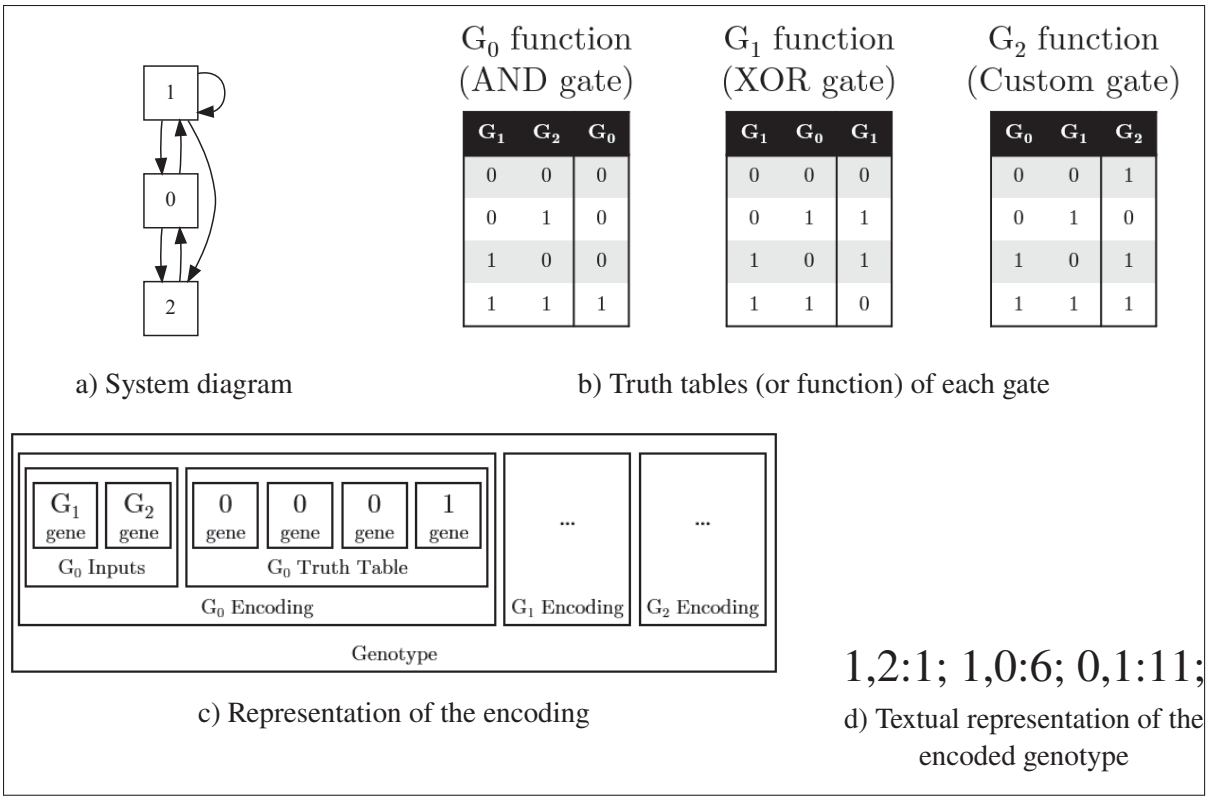
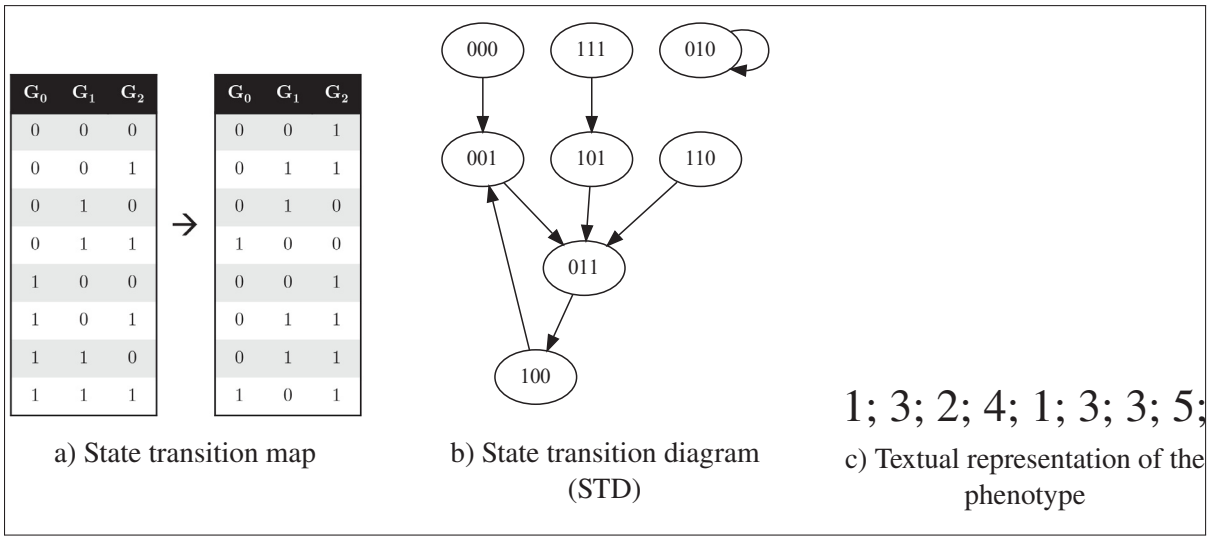


Figure 4.2 Example of how a BNK(3,2) system is encoded into a genotype string

the STD. The behaviour is captured by the oscillations found in loops, which are evaluated using different fitness functions, depending on the problem. Like the genotype, a textual encoding of the STD allows for easier manipulation by a computer program. Continuing from the BNK(3,2) example presented in the previous Section 4.1.1, Figure 4.3 shows the process of encoding the structural phenotype of a BNK system. The caveat of this encoding is that the order of the gates is fixed and cannot be interchanged. This has the effect to produce multiple phenotypes with the exact same behaviour (oscillations).

The STD in Figure 4.3b contains two distinct loops: The first loop where all the states except one lead to a 3-states loop, and another loop for the state '010'. The textual encoding in Figure 4.3c is realized by listing the future states of the state transition map, in ascending order of the initial state, using decimal values. In this case, based on the state transition map presented in

Figure 4.3a, state 0 (‘000’) maps to 1 (‘001’), state 1 (‘001’) maps to 3 (‘011’), state 2 (‘010’) maps to 2, etc.



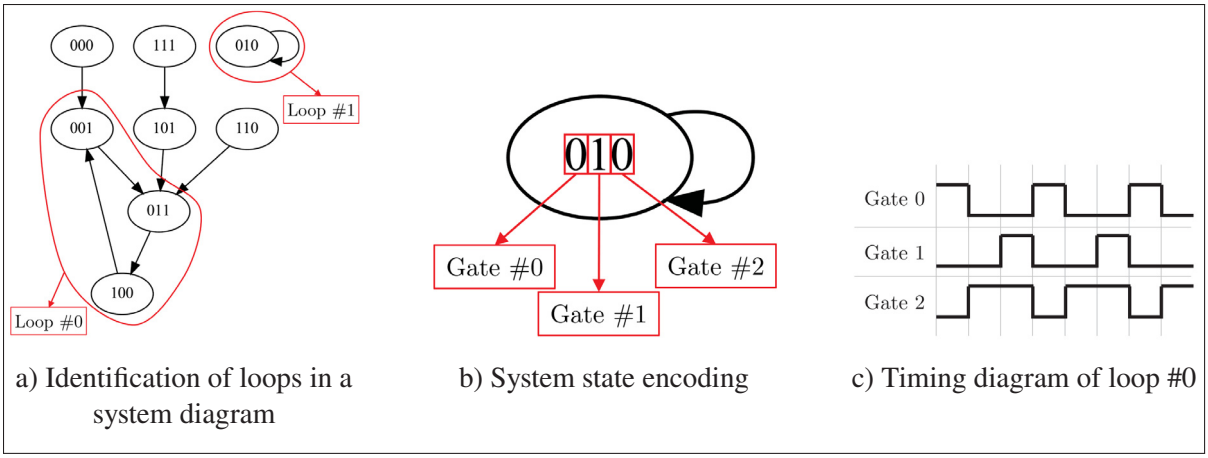
a) State transition map

b) State transition diagram (STD)

c) Textual representation of the phenotype

Figure 4.3 Example phenotype encoding of the same BNK(3,2) system presented in Figure 4.2

Figure 4.4 shows how to analyze the BNK(3,2) phenotype (STD) of Figure 4.3b. Subfigure 4.4a demonstrate how to identify loops in the STD. Loops can be identified by traversing each state until they come back to a previously encountered state (in the same traversal). This example has two loops: loop #0 which shows an asymmetric oscillation through three states and loop #1 which is a single state loop hence does not oscillate. Subfigure 4.4b shows how each state is encoded using binary values. The binary values of each state encode the value of a corresponding gate, depending on the position. An arbitrary choice was made to list the gates from left to right, starting at index 0 and moving in ascending order of gates $(0, \ldots, N-1)$. Subfigure 4.4c is the timing diagram of loop #0 that goes through states ‘001’ $\rightarrow$ ‘011’ $\rightarrow$ ‘100’, starting at state ‘100’. With all loops identified, the timing diagrams for each loop allows to extract the repeating sequence for each gate. For example, the repeating sequence for gate #1 in Figure 4.4c is ‘001’.



a) Identification of loops in a system diagram

b) System state encoding

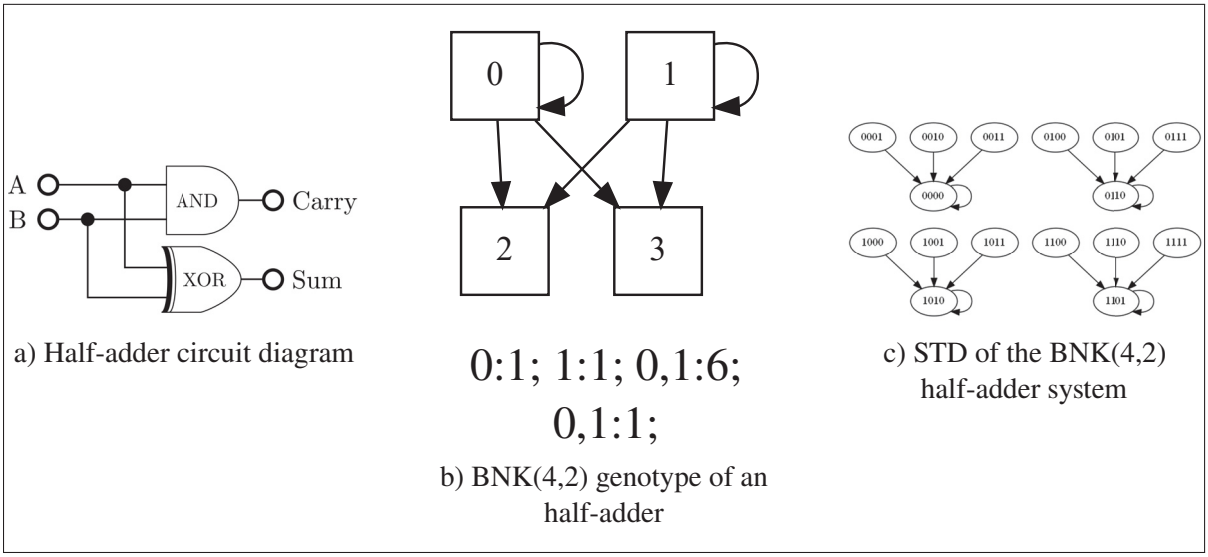
c) Timing diagram of loop #0

Figure 4.4 Interpretation of the BNK(3,2) phenotype presented in Figure 4.3b

4.2 Examples: Digital Circuits and Genetic Networks (using BNK)

4.2.1 The Half-Adder: a simple combinational digital circuit

As shown in Figure 4.5, a half-adder circuit comprises four nodes: two input *Yes* gates (A is #0 and B is #1), an XOR gate (#2) that generates the sum, and an AND gate (#3) that computes the carry-out. Typically this is presented as a circuit diagram, like Figure 4.5a.



a) Half-adder circuit diagram

0:1; 1:1; 0,1:6;
0,1:1;

b) BNK(4,2) genotype of an half-adder

c) STD of the BNK(4,2) half-adder system

Figure 4.5 Representation of a half-adder system

When represented with a BNK system, like in Figures 4.5b and 4.5c, the STD presents four loops, one for each possible outcome of the system (gates 2 and 3 are the outputs, which can theoretically both have binary values of 0 or 1). The genetic representation for the circuit, in BNK standard comprises a list of nodes, where each node is described by the sources of its inputs followed by its logical functionality (in the same order as the bits in the output column of a standard truth table, but converted to its decimal equivalent). As such, the **genotype** of the half-adder in Figure 4.5 comes to: (0:1; 1:1; 0,1:6; 0,1:1). On the other hand, the **structural phenotype** of the circuit comprises the next state of all current states, in ascending order. These are (0; 0; 0; 0; 6; 6; 6; 6; 10; 10; 10; 10; 13; 13; 13; 13), where the next states are listed in ascending order of current states, starting with 0 (decimal) and concluding with 15 (decimal).

4.2.2 The Repressilator: an oscillating genetic network

Oversimplifying, a gene may be viewed as a <condition><action> construct, where the condition is satisfied (or not) by a combination of proteins, and the action (i.e., protein expression) occurs upon satisfaction of the condition. Furthermore, one can model a genetic circuit, comprising multiple genes, as a network of interacting nodes, where each node responds to certain inputs (proteins) by out outing (or not) its own protein. The repressilator (shown in Figure 4.6) is a famous example (Elowitz & Leibler (2000)) of a genetic circuit, one that exhibits an oscillatory behaviour.

A repressilator is made of three nodes, every node accepts one input, which comes from the preceding node. Every node inverts its input to produce its output. As such, the **genetic** representation of the repressilator is (2d:10; 0d:10; 1d:10), showing that every node is an inverter. The **phenotype** is (7d; 3d; 6d; 2d; 5d; 1d; 4d; 0d). Looking at the behaviour in Figure 4.6c, it can be seen that if all genes are initialized with the same values (either all ones or zeros), then they will all oscillate at every step (top loop), otherwise they will oscillate every three steps (bottom loop).

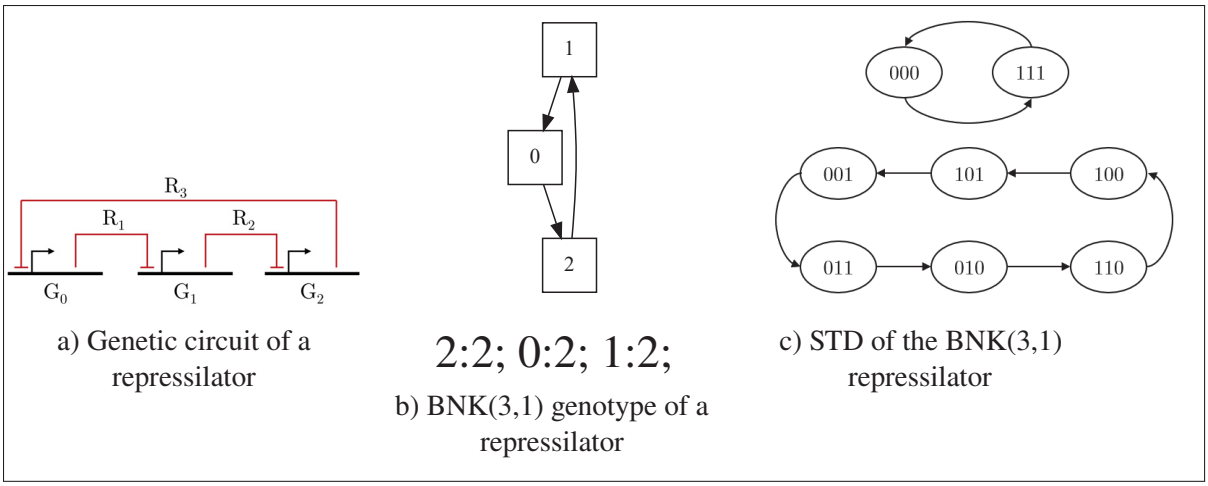


Figure 4.6 Repressilator system, represented in both the domain specific circuit diagram and BNK system diagram

This second example makes it enticing to investigate other genetic networks that exhibit *oscillating* behaviours. This requires the definition of a fitness function that measures the closeness of the behaviour of the genetic network to perfect oscillation. Defining perfect oscillatory behavior, regardless of initial conditions, proved to be a non-trivial exercise. This (fitness evaluation of oscillators) and the rest of an evolutionary algorithm used to evolve genetic circuits with perfect and imperfect oscillators, of various sizes (i.e., number of genes) is the subject of the following Section 4.3.

4.3 Case Study: the Application of E_b and R_b to Oscillating Circuits

The EA defined in Section 2.6.1 was used to find genetic circuits exhibiting oscillatory behaviour. In the EA, every individual has a genotype and a phenotype. BNKe, a slightly constrained version of BNK, is used, where all the nodes have the same number of inputs, exactly equal to rather than up to k. BNKe allows to define more elegant and efficient data structures and algorithms, while still allowing to evolve oscillators, which are studied for ease of evolvability and robustness under perturbation. The genotype and phenotype of a BNKe system are identical to those presented at the beginning of this chapter, except for the constraint just stated.

4.3.1 Random initialization and mutation

With regards to random initialization and mutation, the possible values are $[0, \ldots, N-1]$ if the gene represents an input, or $[0, 1]$ if the gene represents a truth table function.

4.3.2 Fitness Evaluation

Fitness evaluation of an oscillating system requires new definitions, including those of **System Symmetry** and **System Robustness**. Each gate of a BNK system have symmetry and robustness values, which contribute to overall system symmetry and robustness. The fitness of a gate is obtained by multiplying its symmetry by its robustness (with values lying between 0 and 1). The gate with the best fitness (where higher values are better), provides system fitness, since in the case of a perfect oscillator only a single gate is required to exhibit ideal behaviour. Fitness evaluation is presented in Equation 4.1, where $S(n)$ is the symmetry of gate n and $R(n)$ is the robustness of gate n.

$$F = \max\{S(n) \times R(n) : n = 0, \ldots, N-1\} \tag{4.1}$$

4.3.2.1 System Symmetry

To compute the symmetry of a BNK system, the state transition diagram (STD) is formed first, and used to identify every loop (including fixed points). For every gate in a cycle, identify its repeating sequence (including constant values) and, if applicable, its sub-sequences. Compute the symmetry of any and all sub-sequences, and use these values to calculate gate symmetry. Gate symmetry is then multiplied by a penalty, which reflects the ratio of the length of the shortest sub-sequence to the length of the longest sub-sequence (if sub-sequences exist). Loop symmetry is the sum of penalized gate symmetries divided by the number of gates. System symmetry is the weighted average of all loop symmetries, where the weight associated with a loop symmetry equals the number of states in the associated cycle.

In intuitive terms, system symmetry reflects how close the behaviour of every gate in a system is to a perfectly symmetric oscillation, but without requiring all the gates to have the same length of oscillation or to oscillate in synchronization with each other.

A worked example of how system symmetry is calculated is shown in Algorithm 4.1, and the pseudo-code of the general algorithm is presented in additional materials.

Algorithm 4.1 A worked example of symmetry for BNK(3,2) system presented in Section 4.1

```
First loop = '001' → '011' → '100';
    G0 = '001';
        First sub-sequence = '001';
        sub-sequence symmetry = 1/2 * 3/3 = 0.5;          /* Weighted ratio */
    G0 symmetry = 0.5 * 3/3 = 0.5;  /* Sum of all sub-sequences symmetry
     times penalty */
    G1 = '001';     /* Rotate the sequence so that it starts with a
     complete low period */
        First sub-sequence = '001';
        sub-sequence symmetry = 1/2 * 3/3 = 0.5;
    G1 symmetry = 0.5 * 3/3 = 0.5;
    G2 = '011';
        First sub-sequence = '011';
        sub-sequence symmetry = 1/2 * 3/3 = 0.5;
    G2 symmetry = 0.5 * 3/3 = 0.5;
First loop symmetry = (0.5+0.5+0.5)/3 = 0.5;
Second loop = '010';
    G0 = '0';              /* No oscillation results in no symmetry */
Second loop symmetry = 0;
System symmetry = 0.5 * 3/4 + 0 * 1/4;    /* Weighted sum based on the size of
 the loop */
```

4.3.2.2 System Robustness

Robustness of an oscillating circuit is different from the measures of robustness defined earlier. System robustness is equal to average gate robustness. Gate robustness measures the closeness of gate behaviour to ideal robustness. A gate is ideally robust if all possible transient perturbations to its state do not alter its steady-state behaviour. In BNK-modeled systems, perfect robustness

occurs if all states are part of one loop or the loop's basin of attraction or, alternatively, if there are multiple loops and they all exhibit the same gate-level behaviour (and therefore, are phenotypically equivalent to a single loop).

Intuitively, this means that regardless of the initial state of the system, the system exhibits one steady-state behaviour.

A worked example of how system robustness is calculated is shown in Algorithm 4.2, and the standard pseudo-code is can be found in the additional materials.

4.4 Results I: A Tableau of Perfect & Imperfect Oscillators

To find different kinds of oscillators, with varying degrees of symmetry and robustness, a simple Evolutionary Algorithm (EA) was used to generate random BNKe(4,3) systems and evolve them towards perfect oscillation. A BNKe system with $n = 4$ nodes and $k = 3$ inputs was empirically found to provide a good variety of individuals while maintaining a reasonable size for analysis purposes, but similar results were achieved with different settings of n and k.

The parameter values of the EA are shown in Table 4.1; these summarize the descriptions of Section 2.6.1. The resulting individuals (from all generations) where assessed for their respective system symmetry and robustness, and four examples were chosen to exhibit systems with different values of symmetry and robustness. It is impossible to obtain systems with system robustness of 0.

4.4.1 Perfect Oscillator

A **perfect oscillator** is defined as a genotype with a phenotype that exhibits **perfect symmetry *with* perfect robustness**, by at least **one** of its nodes. Intuitively, this means that at least one specific node will generate a perfectly symmetric wave, and return to doing so, after any transient disturbance of state.

Algorithm 4.2 A worked example of robustness for BNK(3,2) system presented in Section 4.1

```
1  G_0;                                                          /* First gate */
2      First loop (L_1) = '001' → '011' → '100';
3          G_0(L_1) = '001';
4          Second loop (L_2) = '010';      /* Check for same behaviour in other
               loops */
5              G_0(L_2) = '0' *3 = '000';
6              For k = 0 to 1;          /* Rotate the shortest sequence by its
                   initial length and compare behaviour */
7                  if G_0(L_1) == G_0(L_2);
8                      Equivalent loops ← (G_0(L_1), G_0(L_2));      /* Keep track of
                           equivalent loops */
9                  G_0(L_2) = '000';  /* Rotate G_0(L_2) by 1 position forward */
10     S_{L_1} = 7, S_{L_2} = 1;      /* Create distribution of states going in each
           loop */
11     S = {7, 1, 0, 0, 0, 0, 0, 0};           /* Merge loops expressing the same
           behaviour and fill with zeros until the size of S is equal to
           the total number of states in the system */
12     σ_S = 2.29;                        /* Compute standard deviation of S */
13     S_max = {8, 0, 0, 0, 0, 0, 0, 0};     /* Create ideal distribution where all
           the loops would express the same gate-level behaviour */
14     σ_max = 2.64;                  /* Compute standard deviation of S_max */
15     Gate 0 robustness = σ_S / σ_max = 0.866;
16 G_1;                                                          /* Second gate */
17     First loop (L_1) = '001' → '011' → '100';
18         G_1(L_1) = '001'; /* Rotate the sequence so that it starts with a
               complete low period */
19         Second loop (L_2) = '010';
20             G_1(L_2) = '1' *3 = '111';
21             For k = 0 to 1;
22                 if G_1(L_1) == G_1(L_2);
23                     Equivalent loops ← (G_1(L_1), G_1(L_2));
24                 G_1(L_2) = '111';
25     S_{L_1} = 7, S_{L_2} = 1;
26     S = {7, 1, 0, 0, 0, 0, 0, 0};
27     σ_S = 2.29;
28     S_max = {8, 0, 0, 0, 0, 0, 0, 0};
29     σ_max = 2.64;
30     Gate 1 robustness = σ_S / σ_max = 0.866;
31 G_2;                                        /* Third gate, skipped for brevity */
32     Gate 2 robustness = σ_S / σ_max = 0.866;
33 System robustness = (0.866+0.866+0.866) / 3 = 0.866;
```

Table 4.1 EA parameters used to find various BNKe(4,3) systems

Parameter	Setup
Initialization	Random
Genetic operators	1-point mutation
Population	10,000
Generations	50
Parent selection	None
Survivor selection	Best of parent & offspring

Examples of BNKe-modeled systems with different levels of systems symmetry and system robustness are presented in figures 4.7, 4.8, 4.9 and 4.10, all done with four gates ($n = 4$) and three inputs ($k = 3$).

Figure 4.7 shows an oscillating circuit of perfect symmetry (i.e., 1) and robustness (i.e., 1). The behaviour shown in the state transition diagram (Figure 4.7b) illustrates perfect oscillation as the only oscillation present is perfectly symmetrical with sequences of lows (0) and highs (1) alternating at every step for all the gates, and perfect robustness as all states lead to the same behaviour (oscillation).

Figure 4.8 shows an oscillating circuit with no symmetry (i.e., 0) and perfect robustness (i.e., 1). This system has no oscillation as the only loop comes from a state leading to itself, meaning the values of the gates will not change thus producing no oscillation. The system still achieves perfect robustness as all states lead to the same behaviour.

Figure 4.9 shows an oscillating circuit with high symmetry (≈ 0.6818) and low robustness (≈ 0.5083). In this example, a good system symmetry is achieved due to all four loops presenting oscillating behaviour, out of which only one is asymmetric (the loop with three states). All the other loops express perfect symmetry through all gates, with the 4-states loop oscillating every two steps while the two small loops (two states) oscillate every step. The robustness is quite low (in practice all systems have a minimum of level of robustness) and due to the various behaviours found in the system. With four states oscillating every two steps, seven states

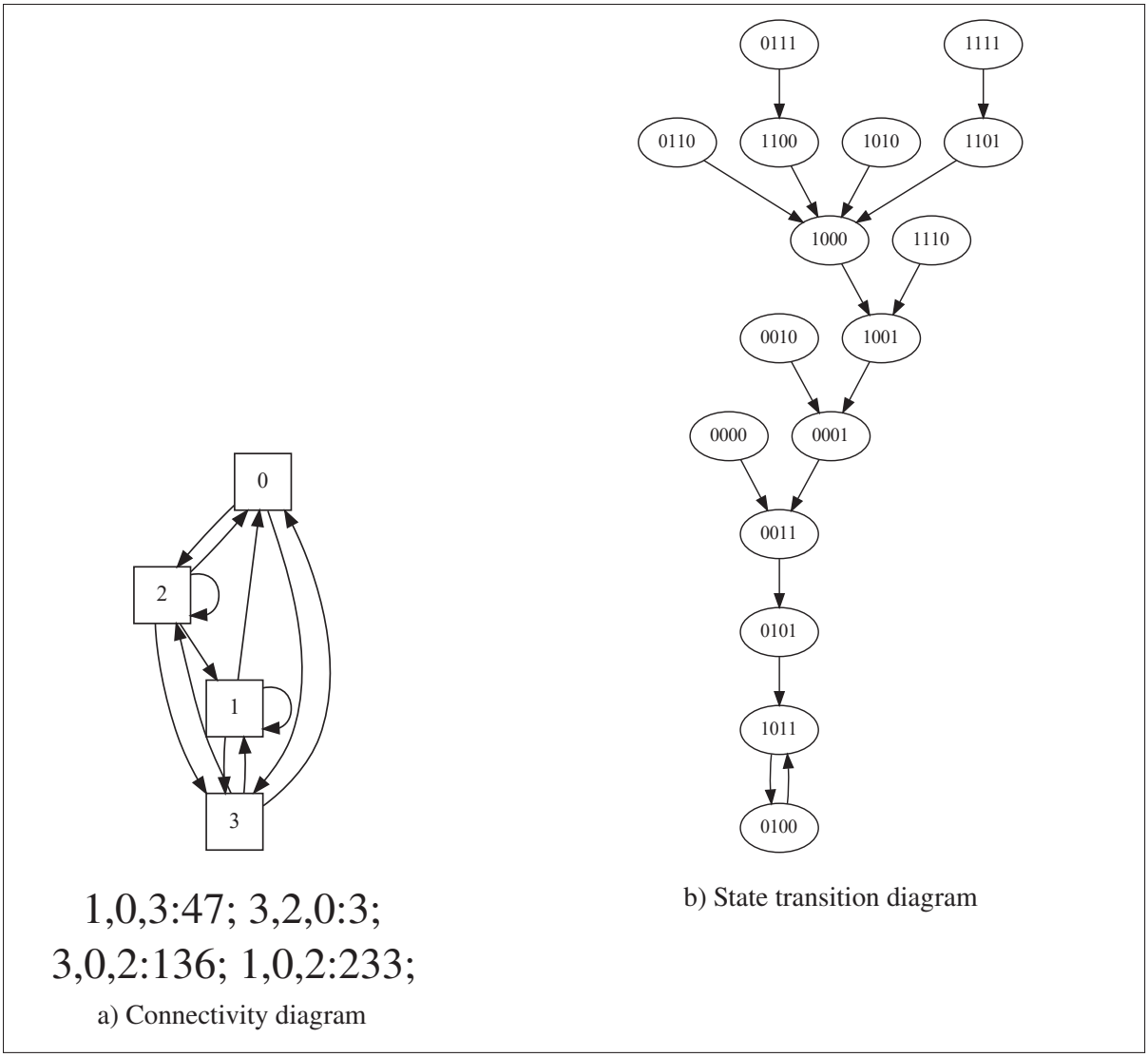


Figure 4.7 Oscillating circuit of perfect symmetry and robustness

oscillating asymmetrically and five states oscillating every step, there is a fair chance that a random state change would drive the system to a different behaviour.

Figure 4.10 shows an oscillating circuit with very low symmetry (≈ 0.2777) and low robustness (≈ 0.5274). This system expresses poor symmetry as across the five loops, only three oscillate. From the three loops that express oscillation, one is asymmetrical (penalizing the symmetry) and the other two have perfect oscillation only for a single gate (the other three gates have no oscillation). This example has low robustness for the same reasons as the previous example

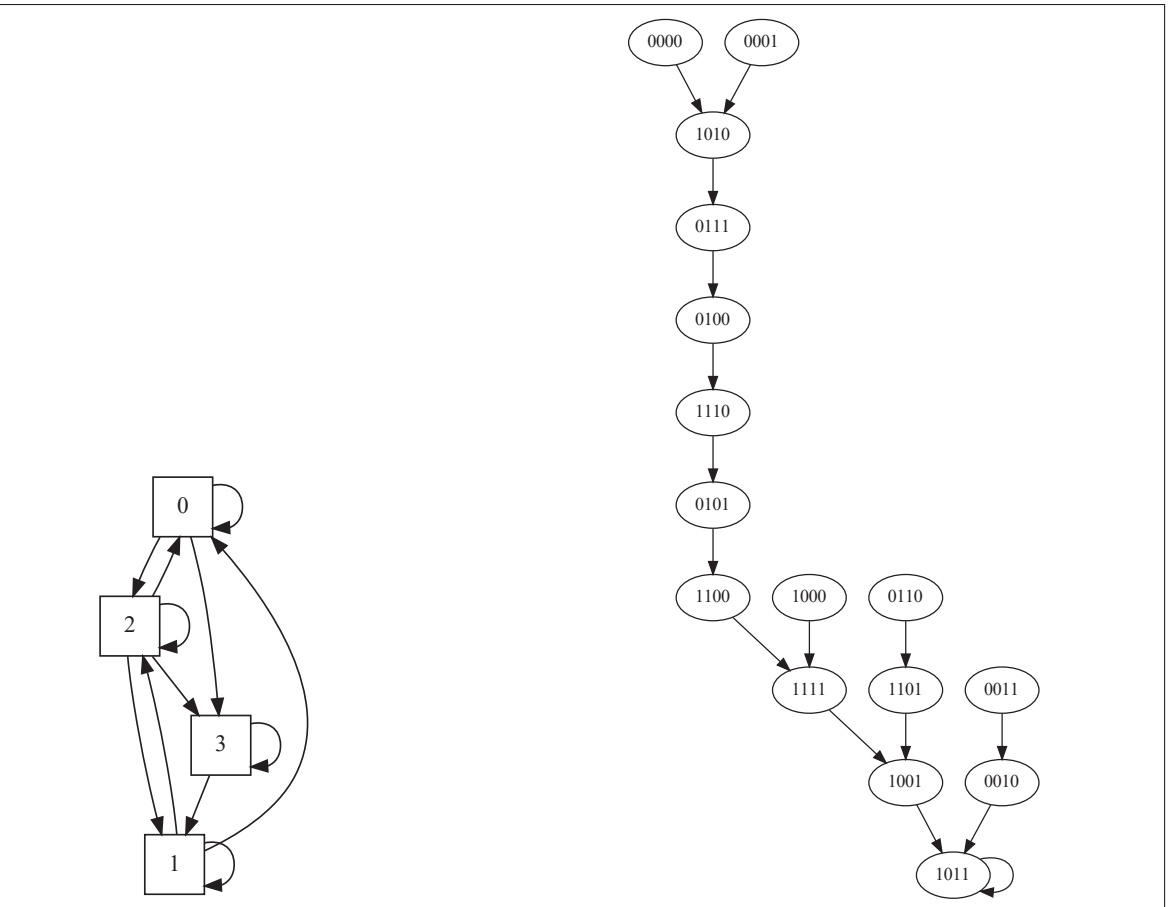


Figure 4.8 Circuit with no symmetry and perfect robustness

(presenting high symmetry with low robustness), although in this case it is a bit more complex since not all gates express the same behaviour within the same loop. Please refer to worked example in Algorithm 4.2 to see how robustness is calculated in detail.

In summary, four different systems were evolved: a system (Figure 4.7) which exhibits perfect symmetry and perfect robustness, with all its states leading to a single ("1011" → "0100") loop; another system (Figure 4.8) with all its states leading to a fixed point ("1011"), a perfectly robust but non-oscillating system; a third system (Figure 4.9) with four different steady-state

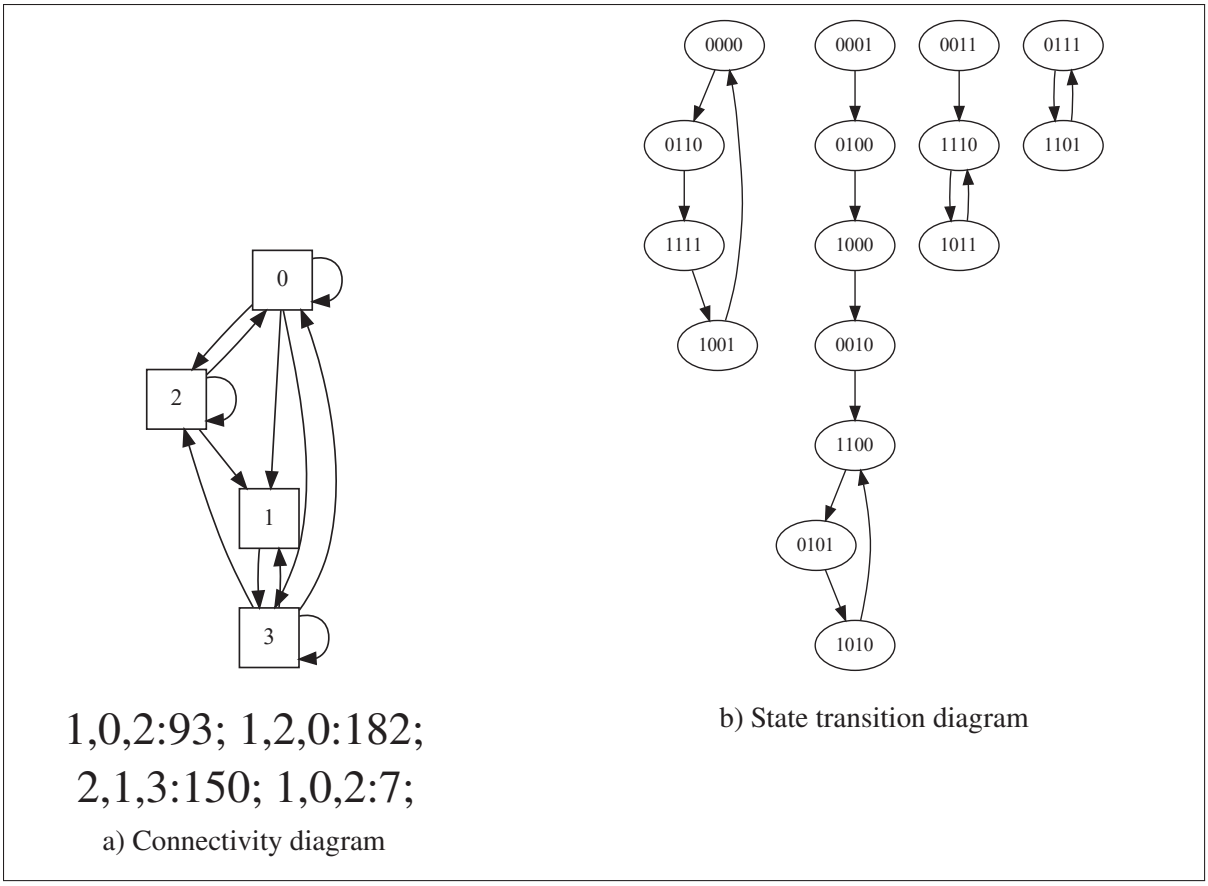
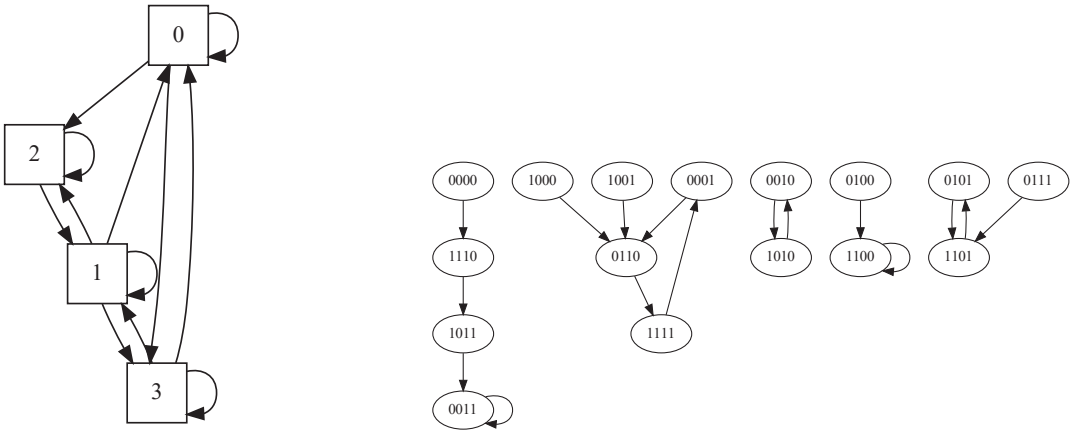


Figure 4.9 Circuit with high symmetry and low robustness

loops, depending on its initial state; a final system (Figure 4.10) that scores quite badly on both robustness and symmetry.

4.5 Results II: The Evolvability & Robustness of Perfect Oscillators

The purpose of this section is to answer three questions, which deal with the evolvability and robustness of oscillating circuits (modeled as BNKe systems) and the relationship between the value of connectivity k and the multi-modality of a fitness surface, where fitness here reflects the closeness of the phenotype to perfect oscillation.



a) Connectivity diagram

b) State transition diagram

Figure 4.10 Circuit with high symmetry and low robustness

4.5.1 How easy is it to evolve a perfect oscillator (which is a phenotype) starting from arbitrary starting points (genotypes)?

To answer this question, the space of genotypes of a BNKe(n,k) system (with specific n and k values, such as $n = 4$ and $k = 3$) are sampled, on a uniformly random basis. This sample represents the starting group of genotypes. Hence, following the general definition of E_b, each genotype is evolved towards perfect oscillation, until either a perfect oscillator is found or a pre-set maximum number of generations is exceeded. Hence, E_b, which is a function of the closeness of the final phenotype to a perfect oscillator and the time it took to reach the final phenotype, is computed over the whole sample.

This experiment leverages the same EA described in Section 2.6.1, along with the parameters listed in Table 4.2. Random BNKe systems are generated and mutated every generation until a perfect oscillator (as defined in Section 4.4.1) is found or the maximum number of generation is reached.

Table 4.2 EA parameters used to assess evolvability of BNKe(1,1) to BNKe(4,4)

Parameter	Setup
Initialization	Random
Genetic operators	1-point mutation
Population	100
Generations	200
Selection	Best fitness

Five different BNKe systems were explored for this experiment, BNKe(1,1), BNKe(2,1), BNKe(2,2), BNKe(3,1) and BNKe(3,2). Figure 4.11 presents the evolution of fitness, along with the walk of one individual across the symmetry × robustness landscape. The fitness evolution plots demonstrate how the number of gates (n) or inputs (k) impact the ability to find perfect oscillators, while the evolutionary walks of single individuals facilitate the understanding of how (in one case) evolution navigates the symmetry × robustness landscape in its journey to perfect oscillation. The details (genetic circuit, BNK genotype and BNK phenotype) of the resulting individuals of each sample walk are presented in Figure 4.12.

In Figure 4.11, the plots on the left show the evolution of average fitness of the population (with the variance σ^2 in grey), together with the resulting baseline evolvability E_b value. Each figure on the right comprise a sample walk of an individual towards perfect oscillation. The evolutionary path can be traced by following the numbered dots or the color gradient (from dark blue to dark red).

In Figure 4.12, the left column shows the equivalent genetic circuits, where each thick black line is a gene (G). The genes in these examples are regulated by repressors and activators. A right-angle arrow marks the start of transcription (of the mRNA), while the thick gray line marks the coding sequence of the protein of the gene (except in the case of `gRNA`, which is a guide RNA). `dCas9_rep` represents a protein that when bound to `gRNA` acts as a repressor. The red lines ending with pointed and flat arrows stand for activators and repressors of transcription of the genes they point to. For all genetic networks presented in Figure 4.12, it is assumed that

repression overrides activation and activated genes have a very low non-activated output. Also, a protein can function as a repressor for one gene and an activator of another (different) gene.

- **BNKe(1,1)**: With a single gate connected to itself, there are only two possible scenarios: the gate has no oscillation or the gate oscillates perfectly. Hence, it is very easy to find all perfect oscillators across the population within only a few generations. The sample walk illustrates the scenario, where the starting individual had no oscillation, until a mutation brought it to perfect oscillation. The genotype and phenotype of the sampled walk confirm this behaviour.
- **BNKe(2,1)**: In this case, only part of the population evolved into perfect oscillators. Upon investigation, a genotype (1:1; 0:1;) was found with low fitness, which cannot be improved via any 1-point mutation. This individual remains sub-optimal. In principle, this situation can be addressed by genetic operators that apply two or more simultaneous mutations to the genotype. On the other hand, the perfect oscillator of the sample walk, has a gate that acts as a pass-through of its value to the other gate, which in turn inverts it and passes it back. This set-up results in a perfect oscillation with a wavelength of two.
- **BNKe(2,2)**: Perfect oscillators are found quickly across the population even with the slightly increased complexity. However, baseline evolvability is lower than the previous case due to the slightly lower density of perfect oscillators in the search space. The evolutionary walk shows that it needed a mutation that traded away all its robustness for enhanced symmetry, before landing on a perfect oscillator. The resulting STD of the sampled walk shows perfect oscillation (with a wavelength of one) on both gates, after a short transition. The first gate simply represses (inverts) itself continuously, and the second gate activates itself only when at least one gate is turned off. This eventually leads the system to a stable cycle, where both gates are oscillating at every step.
- **BNKe(3,1)**: Having three gates with only one input makes it harder to evolve perfect oscillators, but only by a small margin. The evolutionary walk needed to backtrack to increase symmetry before achieving perfect oscillation. This perfect oscillator has a length of two (as in the previous example), but it realizes it through an unusual system. Only the first gate inverts its input and passes the output to the other two gates. These two gates are simple pass-through devices that make no contribution to system oscillation.

- **BNKe(3,2)**: This is the most complex design. The added degree of freedom, due to the additional input, impacts evolvability positively. This complexity results in more convoluted evolutionary walks. The individual presented in the sampled walk has a first gate (G_0) that always keeps its initial value, a second gate (G_1) that inverts itself, and a third gate (G_2) that turns on only when G_1 is active and G_2 is inactive. This leads to two distinct loops in the STD of the system, depending on the initial (constant) value of the first gate. However, in both loops, the second gate oscillates at every step (i.e., exhibiting equivalent behaviour), which renders the system a perfect oscillator.

4.5.2 How robust is that perfect oscillator under mutation (to its genotype)

To answer this question, a perfect oscillator is picked off each of the five runs (above) and apply R_b to it, to produce the scatter plot (and overlaid ellipses) exhibiting the relationship between D_p and D_g. However, to present a summary statistic that reflects the degree of correlation between genotypes and phenotypes distances from the original perfect oscillator, Θ and α are computed for each plot in Figure 4.13.

As a reminder, a **perfectly robust** *system* is one whose states are part of the basin of attraction of *one* loop or fixed point. This means that the steady-state behaviour of the system is always the same regardless of initial state or temporary perturbation. As a reminder, **A system that exhibits perfect oscillation does *not* require perfect system robustness**; it only requires that (at least) one gate exhibits perfect oscillation, at steady-state, and returns to it despite any transient perturbation of state.

This experiment leverages the same EA described in Section 2.6.1, along with the parameters listed in Table 4.3. The resulting perfect oscillators found in Section 4.4.1 are used to initialize the experiment. No selection is applied after mutation since the goal is to see how random mutations affect the behaviour (phenotype) of perfect oscillators. Higher D_p means that the phenotype of the mutated individual is straying away from perfect oscillation. Hamming distance is used to measure the distance between genotypes, while difference in fitness reflects phenotype

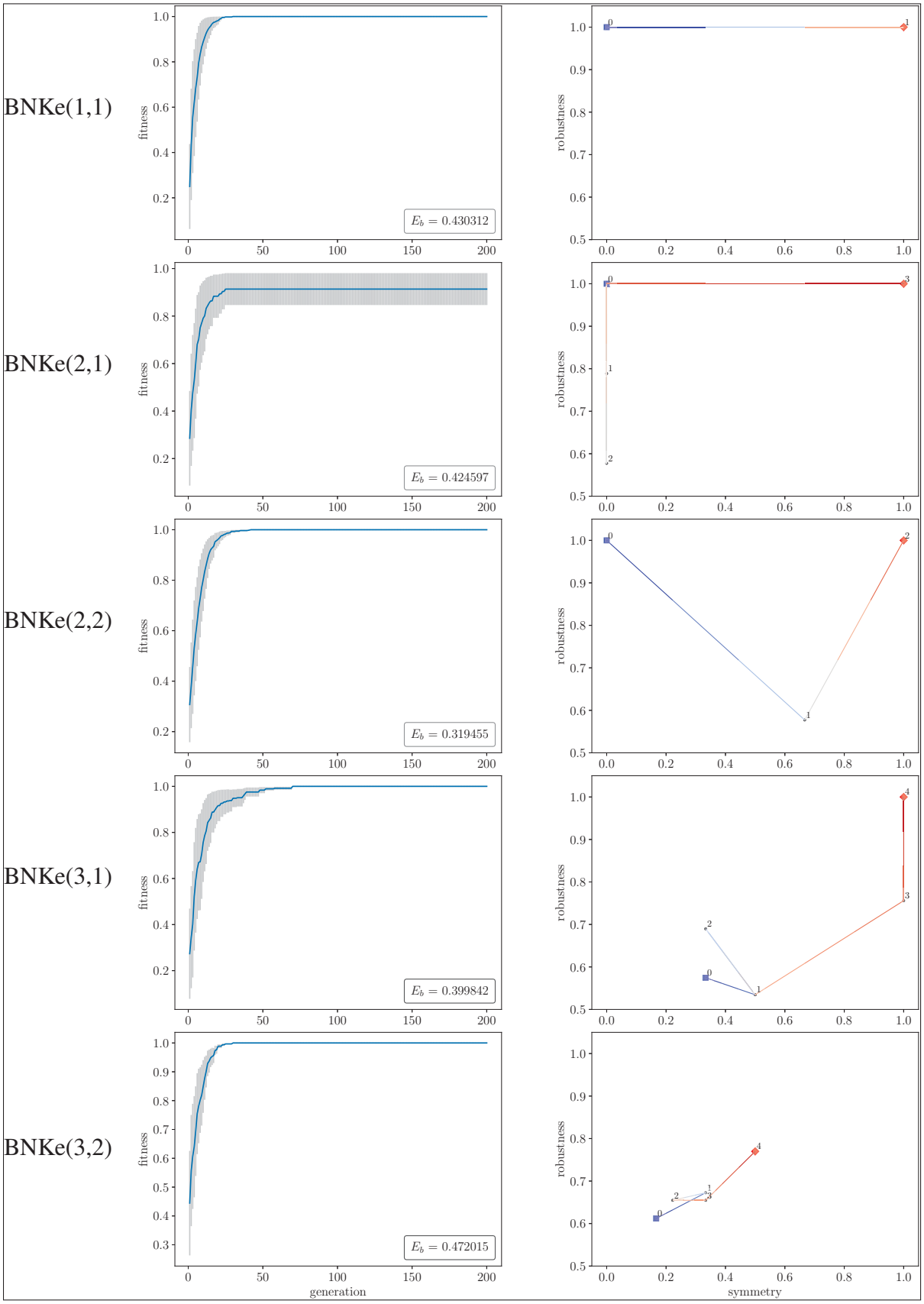


Figure 4.11 Evolution of fitness (left-column), along with the walk of one individual across the symmetry × robustness landscape (right-column)

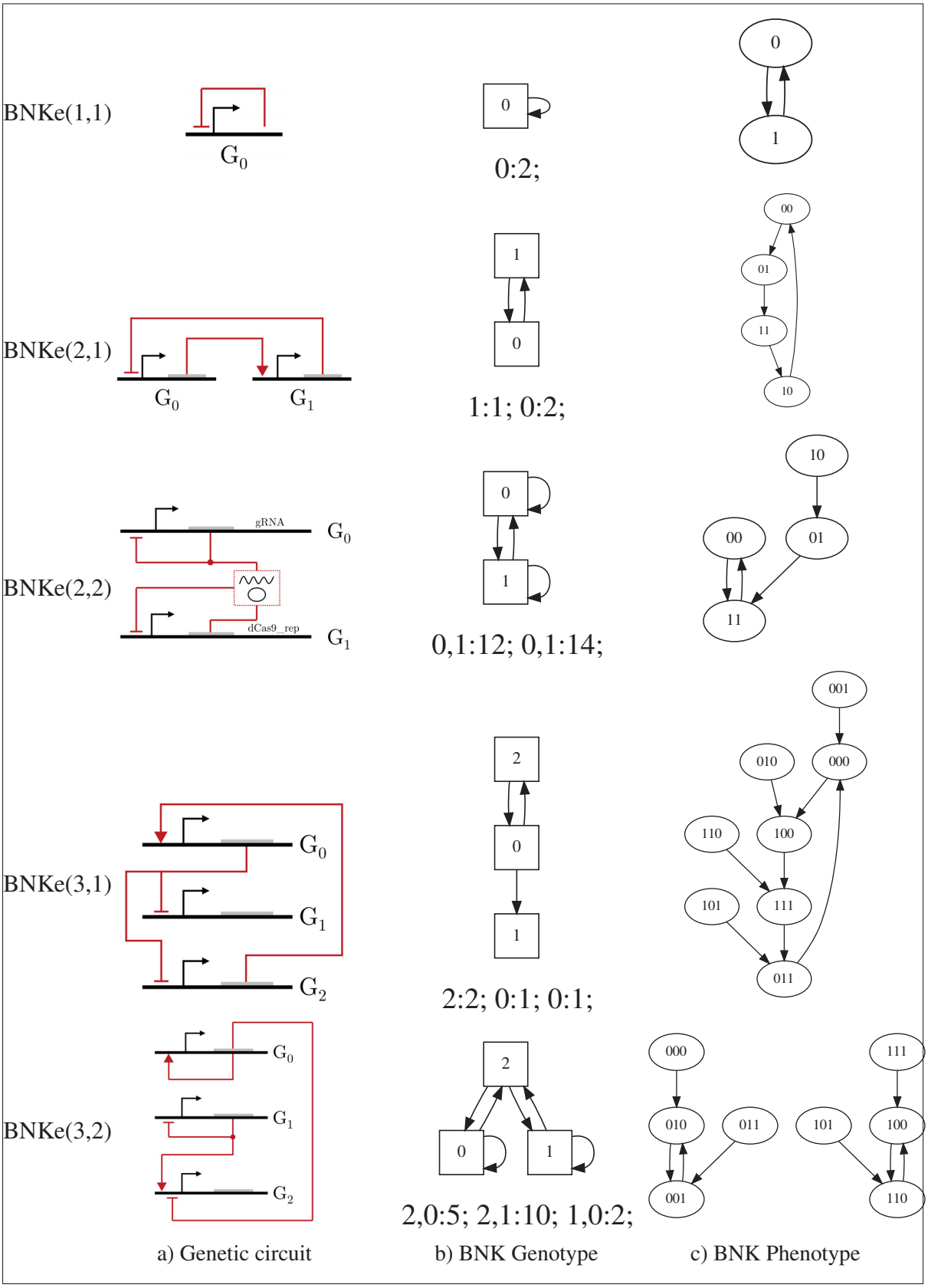


Figure 4.12 Genotypes and phenotypes of the evolved perfect oscillators, found at the conclusion of the evolutionary walks (right column) of Figure 4.11

distance. The hamming distance will work for measuring genotype distance while $n \leq 10$, since the gates can be encoded using a single digit. In the case of $n > 10$, either the gates would need a different basis of encoding (alphanumeric) or use a different method of distance measurement.

Table 4.3 EA parameters used to assess robustness of BNKe(1,1) to BNKe(4,4)

Parameter	Setup
Initialization	Perfect oscillator
Genetic operators	1-point mutation
Population	100
Generations	200
Selection	-

Figure 4.13 illustrates the results of the baseline robustness experiments applied to perfect oscillators. The figures on the left show the cumulative robustness of perfect oscillators while the figures on the right are an animation of the progress of robustness during a single walk (animated figure, best viewed with a professional PDF reader software). A two-dimensional histogram is laid underneath to illustrate the density of individuals found at various coordinates (the gray scale indicates density). The red point is the center of the ellipses, and the three confidence ellipses (65%, 90%, 95%) help visualize the correlation, along with the Θ and α values.

In the first example, BNKe(1,1), robustness is balanced, but this is only due to the extremely limited genotype space. In the other four examples, BNKe(2,1) to BNKe(3,2), robustness is low, as most oscillators lose their perfect symmetry with very few mutations. However, as the complexity of the system increases (by increasing n and k), it is able to retain more of the solutions as perfect oscillators (the center of mass is closer to zero on the vertical axis D_p).

4.5.3 Does increasing k increase the multi-modality of the fitness surface (where fitness here reflects the level of perfection of oscillatory behaviour)

For this experiment, the search space for each problem ($n = 1, 2, 3, 4$) was sampled randomly 10^7 times. Fitness evaluation for a perfect oscillator as described in Section 4.4.1.

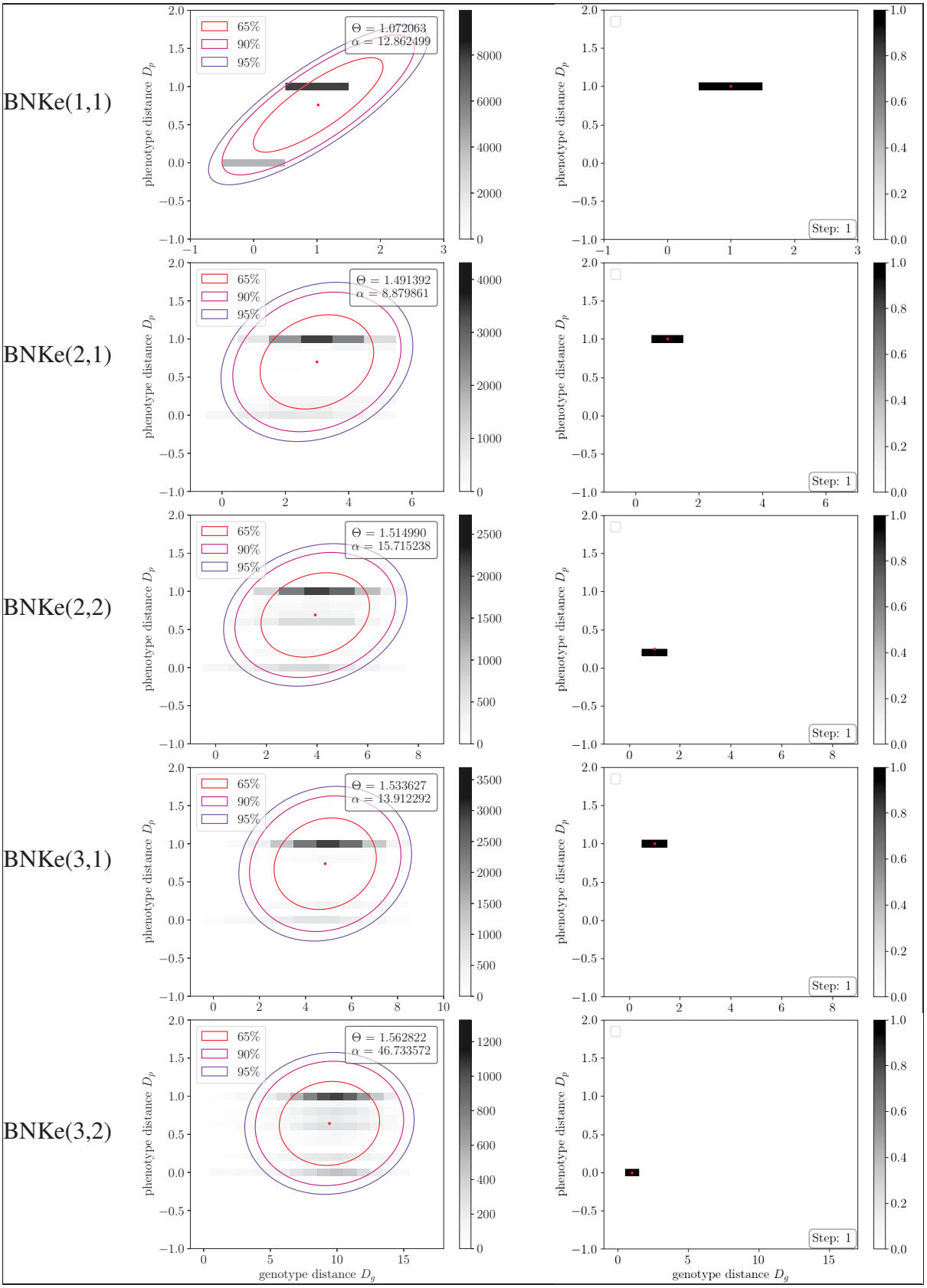


Figure 4.13 Results of the baseline robustness experiments applied to perfect oscillators

Taking example from Figure 4.2a, the genotype can be divided into connectivity and functionality. Figure 4.14 illustrates how each part is extracted from the genotype, with the connectivity elements identified with red squares and the functionality elements with red circles.

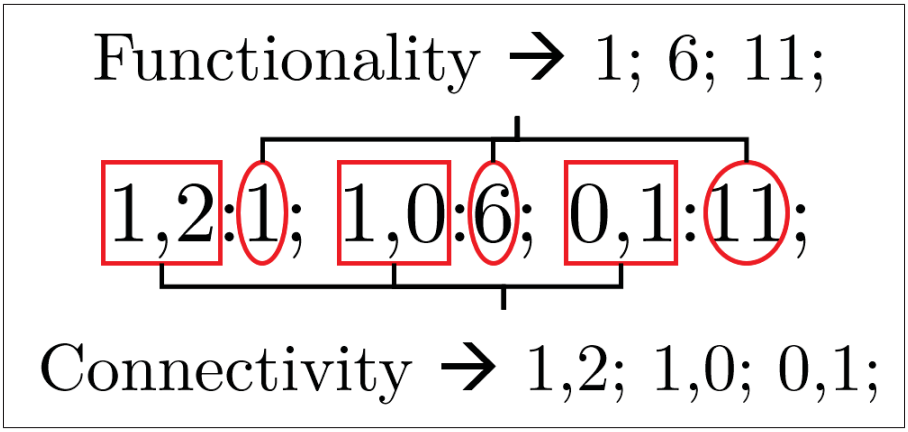


Figure 4.14 Building connectivity and functionality strings from a BNK genotype

4.5.3.1 Functionality

Functionality being the decimal value of the output of each gate's truth table, the possible values always start at 0, and the upper limit will be $2^{2^k} - 1$, which is 15 for each gate in this example. Then, find the index of the functionality string in the ordered set of all possible functionality strings ("0;0;0;", "0;0;1;", . . . , "15;15;15;"). The axis represents the indexes in the ordered set of functionality strings, and is normalized by dividing with the size of that set (16^3 in this example). The example functionality string is positioned above the axis approximately, for illustrative purpose.

4.5.3.2 Connectivity

A BNK system does not allow to be connected to the same gate twice, hence the set of possible values for each gate of this example is $\{(0,1), (0,2), (1,0), (1,2), (2,0), (2,1)\}$ (the order of the connected gates matter). As for the connectivity axis, find the index of the connectivity

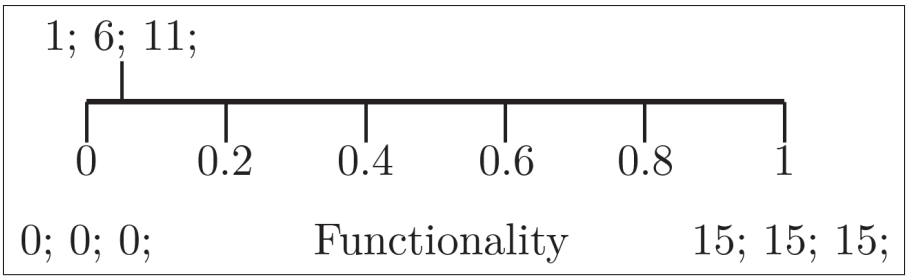


Figure 4.15 The scale of functionality, based on all the possible values in logical ascending order, with the most significant digit starting from the left

string in the ordered set of all possible connectivity strings. The axis represents the indexes in the ordered set of connectivity strings, and is normalized by dividing with the size of that set (6^3 in this example). The example connectivity string is positioned above the axis approximately, for illustrative purpose.

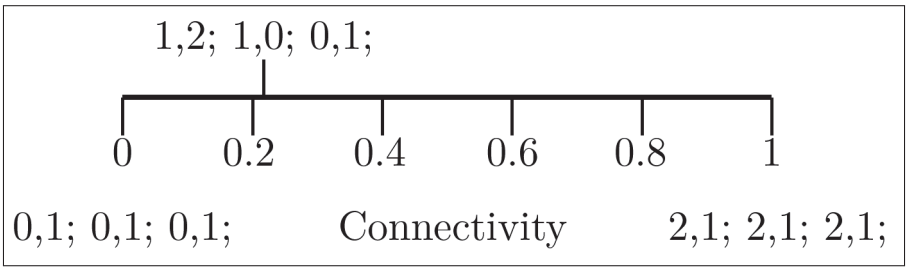


Figure 4.16 The scale of connectivity, based on all the possible values in logical ascending order, with the most significant digit starting from the left

4.5.3.3 Fitness landscape

The two axes of connectivity and functionality form a very dense 2D plane, with every point having a fitness value on the third axis. For practical reasons, a discrete grid is superimposed on what may be a very dense 2D plane. If the total number of coordinates along an axis is less or equal to 10^3 then the coordinates of the grid points (along each axis) equal the coordinates of the points. If they are greater than 10^3 then only 10^3 are used for that axis. Once that is done and

the fitness value is computed, a local 3×3 local maximum filter is applied to the discrete fitness surface, to identify (and hence count) the total number of local optima Σ found. Σ is obtained by applying a maximum filter with a 3×3 binary filter (filter values set to 1). This operation uses the filter to replace the pixel at the center of the filter with the maximum value found in the connected neighborhood, effectively "dilating" the image. Then the original image is compared with the dilated image (using ⊕), giving a map of a local peaks, which are then counted. The number of local optima identified using this filter is used as a measure of multi-modality of the fitness surface.

4.5.3.4 Measure of multi-modality

The 3D and 2D views of the fitness landscape for each BNKe system with $n = 4$ and $k = 1, 2, 3, 4$ in Figure 4.17 demonstrate how the multi-modality of the fitness landscape increases with k. The figures on the left are an interpolation of all the points sampled, generating a fitness landscape in three dimensions. The figures on the right are a top view of the fitness landscape capturing the local optima Σ. An amplification of a 50×50 region better illustrates the density of local optima in the search space. The fitness value is represented using a red gradient (white when fitness is 0 up to dark red when fitness is 1). From the entire search space of BNKe(4,1), a total of 18,053 peaks were identified. For $k = 2$, the estimated number of peaks is 6,857,365, for $k = 3$ the estimated number of peaks is 8,152,073 and for $k = 4$ the estimated number of peaks is 8,294,400.

Starting with BNKe(4,1), clearly defined regions of perfect oscillators can be observed. In this case, the entire search space was sampled. With $k = 2, 3, 4$, the search spaces are now much bigger than the 10^7 samples taken, which means the shown landscape is less precise. However, some concentrated regions of perfect oscillators can still be seen with $k = 2$, which dissipate when $k = 3$ and $k = 4$. The zoomed regions of the 2D landscapes also confirm that the local optima get sparser as the search space increases.

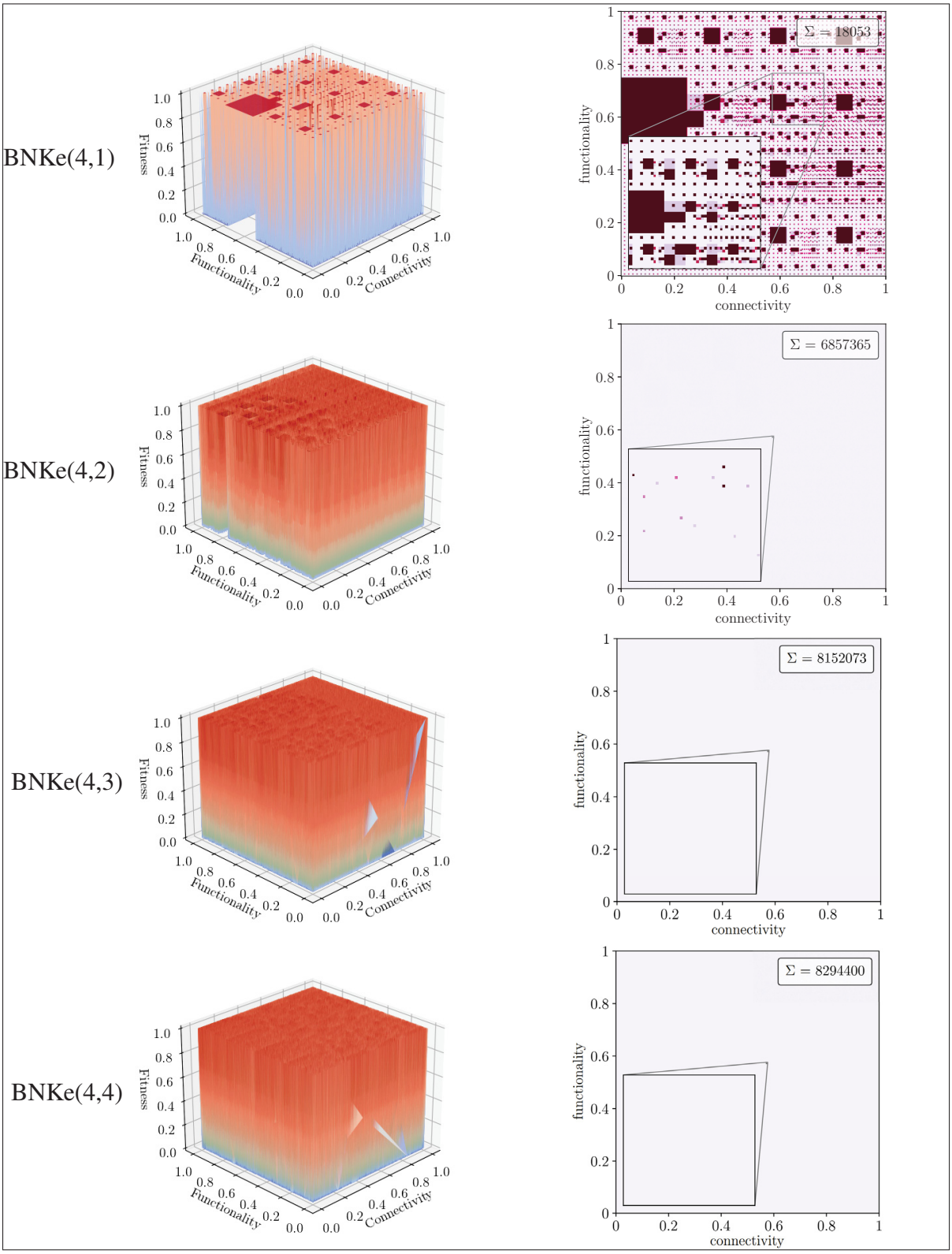


Figure 4.17 The multi-modality of the fitness landscape of BNKe($n = 4$, $k = 1, 2, 3, 4$)

4.5.3.5 Relationship of the number of inputs on the multi-modality

As shown in Table 4.4, the number of local optima Σ increases with k. However, the density of local optima, measured as the fraction of local optima over the entire search space, decreases with the exponential expansion of the search space. Hence, there are a greater number of ways to achieve perfect oscillation in larger spaces, but it is also harder to find them, due to the need for more extensive searches.

Table 4.4 Relationship between k and Σ in a BNKe system

k	Σ	density of Σ
1	18,053	≈ 0.2754
2	6,857,365	≈ 0.0050
3	8,152,073	$\approx 2.46e{-6}$
4	8,294,400	$\approx 2.5e{-6}$

4.6 Conclusion of the application of the baseline measures to evolving oscillating circuits

In order to demonstrate the versatility of the baseline measures, a generic modeling approach was developed to build oscillating circuits. This model, namely BNK, is similar to NK System, but offers the intermediary phenotype representation necessary to assess evolvability and robustness.

The baseline measures E_b and R_b were applied to the specific problem of finding perfect oscillators, and provided insight into how the complexity of the genotype and phenotype space affect the evolvability and robustness of such system by changing the number of gates (n) and inputs (k) in the system.

CHAPTER 5

EVOLVABILITY OF GENETIC PROGRAMMING LANGUAGES

The goal of this chapter is to demonstrate how the baseline measures E_b and R_b can be applied to existing (and future) Evolutionary Algorithms to assess their performance and understand the potential limitations of these measures. A simple experiment will be conducted with three different variants of GP languages on three well known benchmark problems, aiming to give a fair assessment of their performance in terms of baseline evolvability, baseline robustness and fitness.

5.1 Selecting Genetic Algorithms

One of the key aspects of measuring distances between two genotypes is choosing an appropriate method that captures the minimum number of genetic operations needed to go from one genotype to the other. Hence, a GA that leverages advanced genetic operators such as multi-point crossover will increase the complexity of defining a good distance measurement algorithm. As this chapter aims to offer a demonstration of application, complex languages with non-linear representation will be left for further research. Thus, the focus will be kept on GP languages that use integer strings of fixed lengths for their genotype and enforcing baseline mutation as their only genetic operator. Given these constraints, the Hamming distance can be used to compute the minimum number of mutations between two genotypes in all cases.

The three GP languages assessed will be Grammatical Evolution (GE), Gene Expression Programming (GEP) and Cartesian Genetic Programming (CGP), previously introduced in Section 1.1. Variants of GP have been reviewed previously by Oltean & Grosan (2003), which covered these three languages but only in application to symbolic regression problems. In that paper, the authors demonstrated that CGP performs better (in terms of fitness) for the selected regression problems. However, their respective evolvabilities were not assessed.

Since all three GP languages also have invariant structures in their genotype, the Hamming distance can be used to measure the distance between two of their genotypes. In this case,

Hamming distance does not reflect the difference between characters, but rather the number of genes with different values, as illustrated in Figure 5.1. The length is defined by the number of genes. The value of genes $g_0 \ldots g_4$ are compared and the distance is incremented by 1 for each differing gene.

	g_0	g_1	g_2	g_3	g_4	
Genotype A	54	145	28	204	10	
Genotype B	8	145	28	104	10	
	1	0	0	1	0	$\longrightarrow \Sigma = 2$

Figure 5.1 Application of the Hamming distance to genotypes of identical length

5.1.1 Structural phenotype representation

When evaluating evolvability and robustness, a structural representation is most suited as it is independent of the environment. Assessing the ability to move from one random phenotype to another simulates a change of the environment, where the starting structural phenotype had an ideal fitness in the previous environment, and the target structural phenotype has an ideal fitness given a new environment. In the case of GP languages, the structural phenotype is the rooted tree structure of the execution of functions and terminals. The baseline measures of evolvability and robustness are about changing and keeping their structures, respectively.

As the following experiments are about the baseline measures, the algorithms will use a structural phenotype representation. The structural phenotype of a GP language program can be represented as a tree as long as it has a single output, which will be the case with the selected benchmark problems. The Tree Edit Distance will be used to measure the distance between two phenotypes. Rooted tree structures can be encoded using the Bracket Notation format, as represented in Figure 5.2. The choice of bracket does not matter in the Bracket Notation format. TED algorithms leverage that notation to parse and measure tree structures.

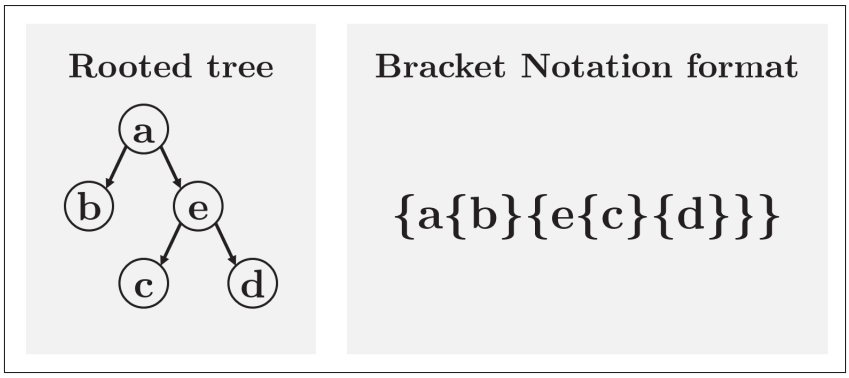


Figure 5.2 A simple example of rooted tree structure encoding using the Bracket Notation format

5.1.2 Grammatical Evolution (GE)

In GE, the nodes of a rooted tree are random integer values, which are translated into program code using a grammar table. The structure of the tree is built by following the sequence of functions and terminals in the genotype, filling them with the following genes until no more are needed to build a valid tree. This means that if the first gene is a terminal, the tree will be a single node, discarding the rest of the genotype.

For the following experiments, the validity of the generated GE individuals will be enforced at initialization. Any invalid individual generated during random initialization is replaced by another random individual. Tree size will be limited to exactly 60 nodes, resulting in a genotype of 60 genes, with each gene represented by an integer between 0 and 255 inclusively.

In GE, fitness is evaluated in the same way as GP, defined by Koza (1992). The fitness evaluation is set as a minimization problem, and values are normalized to $[0, 1]$, using the worst possible fitness as the normalizing factor.

The following example illustrates how GE works. The algorithm uses genotypes of length 10 and has access to a single terminal (x) and four arithmetic operators ($+$, $-$, $*$, $/$). The associated

Backus Naur Form grammar used is

<start> ::=<op>

<op> ::=(+<op><op>)

|(−<op><op>)

|(∗<op><op>)

|(/<op><op>)

|(x)

where the <start> expression translates directly to <op>, and in turn <op> can translate to one of the five expressions linked to the sets of functions and terminals, in the order stated. Taking the genotype 34,106,242,99,158,240,239,214,134,214 as an example, the genes are translated using the grammar rules shown in Table 5.1. The values in the modulo column are computed by dividing the gene value by the number of choices available from the grammar (<start> has only one and <op> has five).

Table 5.1 GE genotype decoding using the Backus Naur Form grammar

Gene index	Gene value	Grammar expression	Modulo	Translation
0	34	<start>	0	<op>
1	106	<op>	1	- <op><op>
2	242	<op>	2	* <op><op>
3	99	<op>	4	x
4	158	<op>	3	\<op><op>
5	240	<op>	0	+ <op><op>
6	239	<op>	4	x
7	214	<op>	4	x
8	134	<op>	4	x
9	214	<op>	4	x

This results in the rooted tree structure shown in Figure 5.3. Given this grammar, the first gene has no impact on the structural phenotype since its sole purpose is to function as the root of the tree.

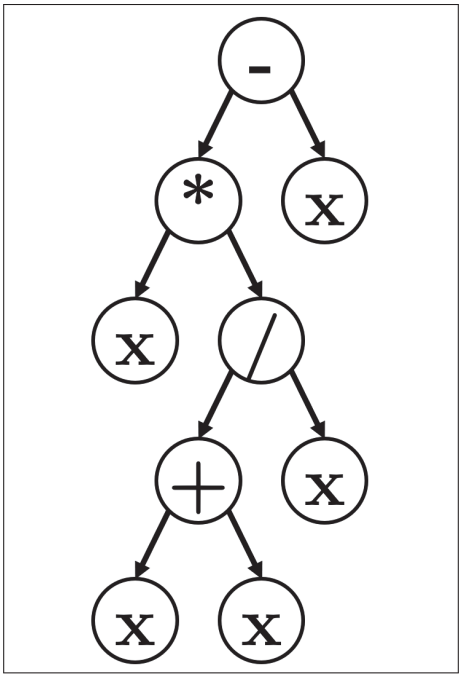


Figure 5.3 Resulting rooted tree from decoding the GE genotype in Table 5.1

5.1.3 Gene Expression Programming (GEP)

GEP also uses a tree representation but differs from GE by splitting the genotype into several sections (or chromosomes). Each chromosome is a sequence of genes. The chromosomes are of fixed length and comprise a head and a tail. The head comprises at least one function and the tail is made of terminals only. This imposes a significant constraint on the complexity of viable solutions when only a single terminal is available, as the entire tail portion will never change. Like GE, the structural phenotype is built by decoding the genotype from left to right, filling the functions until completion. While GE allows a single node tree (terminal only), GEP enforces a function as the starting node, so the GEP tree will never have less than two nodes. In GEP, it is also possible to adjust the weight of each function in the function set, adjusting their respective probabilities to be picked during mutation. All functions will have equal weights in the following experiments.

The genes in GEP are represented by unsigned integers. The possible values for a gene are determined by the sets of available functions and terminals for that gene. Given a set of functions F and a set of terminals T, the maximum value for a gene would be $|F| + |T| - 1$, assuming the range starts at 0. In GEP, fitness is maximized, is often normalized to $[0, 1]$, and is sometimes scaled.

For the following experiments, a single chromosome with a head size of 30 will yield a genotype of 61 genes. This is due to the tail being of equal size to the head and because there is a linking gene at the beginning of the genotype. The linking gene can be any function available in the function set and is defined as part of the hyper-parameters, but is unused when GEP is set up with a single chromosome.

The following example illustrates how GEP interprets its genotypes, using a single chromosome with a head size of ten genes, a single terminal (x) and four arithmetic operators ($+$, $-$, $*$, $/$). Table 5.2 shows how a GEP genotype is decoded, by replacing the gene value within the genotype with the associated expression, until a complete tree is built (all leaf nodes are terminals).

Table 5.2 GEP genotype decoding using the sets of functions and terminals available

Gene value	Expression	Arity
0	+	2
1	-	2
2	*	2
3	/	2
4	x	0

Taking genotype 3,0,4,4,4,2,2,4,2,4,4,4,4,4,4,4,4,4,4,4,4 as an example, a complete tree is built using only the first five genes (3,0,4,4,4), which produces the structural phenotype $\{/\{+\{x\}\{x\}\}\{x\}\}$. The rest of the genotype, including the tail containing only the sole terminal x, are neglected because all prior functions have already been filled with terminals found in the head part of the genotype. The resulting rooted tree structure is presented in Figure 5.4.

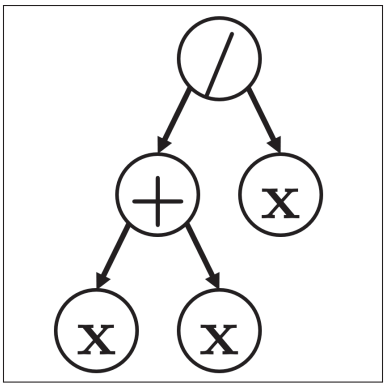


Figure 5.4 Rooted tree from decoding the a GEP genotype

5.1.4 Cartesian Genetic Programming (CGP)

CGP is more unique than GE and GEP because it uses a graph. Instead of a tree, the nodes in CGP are organized in rows and columns, in a cartesian fashion, hence its name. It looks similar to a basic neural network, with input nodes at the left end feeding layers of function nodes from left to right, leading to the output nodes at the right end. Nodes in a given column can only accept inputs from nodes in previous columns. The function with the highest arity in the function set defines the number of inputs for all nodes, which means that some inputs of a node may get discarded if its function does not need them (inputs are used in sequential order). An illustrative example is presented in Figure 5.5, where the functions all take two inputs. The notation of the genotype shows the functions as F_i where i is the index of the function in the table, to ease readability, however only the index i shall be used when implementing this GA, making the genotype a pure string of integers. When there is only a single output node specified, which is the case in the selected benchmark problems, CGP can also be represented as a rooted tree, with the single output representing the root.

For the following experiments, every problem will have a set of functions with a maximum arity of two and a single output. Considering that each node in CGP will have two inputs, using 20 nodes will yield a genotype with 61 genes. The topology will take the form of a single row, since

this imposes the least constraints on connectivity and is the choice recommended when there is no specialist knowledge of the problem. Fitness evaluation in CGP is a matter of function minimization, and normalization is optional.

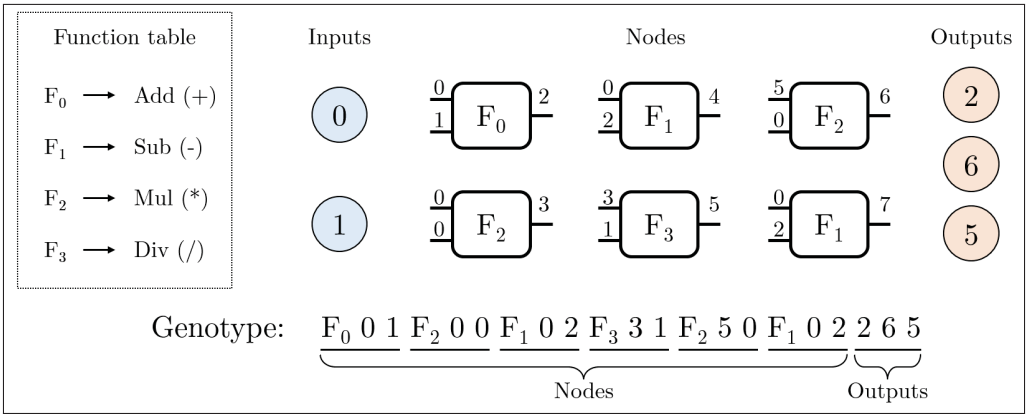


Figure 5.5 Example CGP system with four functions, six nodes, two inputs and three outputs

5.2 Selecting benchmark problems

Certain GP languages can perform better than others on one type of problem or another. For this reason, the three GP languages will be applied to three types of well-known problems, often used for benchmarking performance, namely a **parity** problem, a **regression** problem and a **control** problem. Each problem difficulty can be adjusted, however what is most important is ensuring a fair basis of comparison. Hence, an arbitrary difficulty level will be chosen for each of the three problems, and hence, fixed during comparison. Furthermore, the set of functions and terminals available to the GP languages will be identical for a given problem, regardless of GP language.

Each problem is described below. The test cases and fitness functions defined for each problem are applicable only to the fitness evaluation experiments, which will be presented in Section 5.3.4. In contrast, when applying the baseline measures, only the set of functions and terminals are relevant, as fitness evaluation is not involved in the process.

5.2.1 Parity problem

The *boolean parity* problem consists of counting the number of active bits (value is 1) in a binary string. The parity is said to be odd or even based on the total number of ones in the input. The difficulty can be controlled by the total number of bits to check parity for and by the set of boolean functions available to the GP language for finding a suitable program. The fitness of a program is assessed by testing binary strings and comparing the resulting binary values of the program evaluation with the expected parity values. Usually, all combinations of binary string are assessed.

This experiment will check for even parity of a string of four bits. However, the choice of checking for odd or even parity does not matter. With four bits, there will be 2^4 training samples (0 to 15 inclusively, in binary format), as shown in Table 5.3.

Table 5.3 Truth table of the even and odd parities on four bits

b_0	b_1	b_2	b_3	Even parity	Odd parity
0	0	0	0	0	1
0	0	0	1	1	0
0	0	1	0	1	0
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	1	0
1	0	0	0	1	0
1	0	0	1	0	1
1	0	1	0	0	1
1	0	1	1	1	0
1	1	0	0	0	1
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	0	1

The behaviour is represented with a binary string of the output of the truth table for even parity in ascending order, encoded from left to right. Fitness evaluation is the Hamming distance

between the output and the expected parity values for each input. Fitness is a maximization function normalized in $[0, 1]$.

The algorithms will have access to four terminals (one per bit) and five boolean functions (*and*, *or*, *not*, *nor* and *nand*).

5.2.2 Regression problem

Regression problems are problems where the goal is to find the underlying function, provided a set of sampled points from said function. Some types of regression can be given a priori knowledge of the target function, as opposed to *symbolic regression* which does not know what the function should look like. The difficulty can be controlled with the choice of function and the number of samples provided.

For this experiment the quartic function ($x^4 + x^3 + x^2 + x$) was chosen arbitrarily as it is often used for symbolic regression and offers a manageable complexity. Twenty (20) samples in the range of $[-1, 1]$ will be provided as training data to the algorithms. The training data is listed in Table 5.4.

The algorithms will have access to one terminal (the value of the training point x) and four functions (*add*, *subtract*, *multiply* and *divide*).

The phenotype is a list of resulting values from the evaluation of the 20 training points. Fitness is evaluated by computing the MSE between the output and the expected value of the quartic function. It is standardized as per Equation 5.1, making it a maximization function. Since floating point number calculations in computers are prone to approximation errors, the ideal fitness is relaxed from $F = 1$ to $F >= 0.99999999$.

$$F = \frac{1}{1 + E} \tag{5.1}$$

In the event that the evaluation of a function produces an infinitely high (or low) value, exceeding the maximum capacity of a floating point number of the computing system, that value is capped at the highest (or lowest) value that this computer can handle, using floating point numbers (for example, in *C++*, the maximum value of a `double` variable is 1.797 69e308).

Table 5.4 Training points used for the symbolic regression problem

x	y
-0.97678574	-0.044310153
-0.921672563	-0.133518145
-0.917816363	-0.138970296
-0.708543397	-0.310184465
-0.549206795	-0.322255382
-0.542213906	-0.321193061
-0.44333413	-0.295294156
-0.421426863	-0.287129953
-0.331091595	-0.245747851
-0.133366673	-0.117635787
0.067684277	0.072596498
0.105053924	0.11737146
0.129867682	0.149208044
0.204417033	0.25649129
0.238579242	0.312319133
0.267929057	0.364101814
0.376840385	0.592530128
0.539974999	1.074004943
0.821473933	2.506021174
0.86685775	2.834359357

5.2.3 Control problem

A control problem consists of controlling a system in an environment by leveraging its functions to explore and exploit the environment, maximizing its fitness. One of the most well-known control problem is the *artificial ant* problem, where an artificial ant is positioned (with a direction) on a toroidal grid with food pellets positioned across, and the ant needs to eat as much food as possible in a given number of movements. The ant can check if there is food ahead,

move forward, turn left, or turn right. Moving into a grid cell containing food automatically consumes it, increasing the fitness of the ant. The difficulty of the problem can be adjusted by changing the size of the grid, the number and geographical distribution of food pellets and the maximum number of actions allowed by the artificial ant (checking for food ahead, turning left, turning right, and moving forward all count for one action).

The most popular trail (predefined grid) is called the Santa Fe trail (Figure 5.6), and will be used for this experiment. It is a 32×32 grid with 89 food pellets at fixed locations, with a fixed starting position. The ant is allowed a maximum of 400 actions to consume as much of the food pellets as possible.

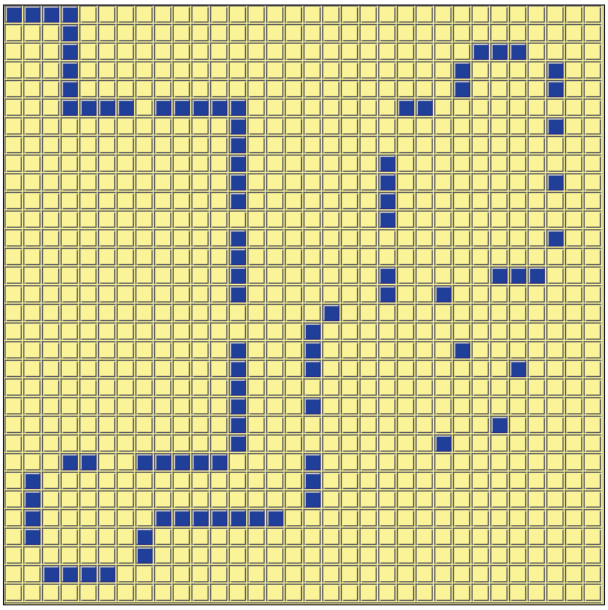


Figure 5.6 Figure depicting the Santa Fe Trail for artificial ant problem. Obtained from Wikipedia, original submission by Rdlaw (2011)

The algorithms will have access to three terminals (*move*, *turn left* and *turn right*) and two functions (*check for food ahead* and *execute two actions sequentially*).

The behaviour is represented by a binary string of the state of the 89 food pellets on the trail after program execution. After an individual has been evaluated, food pellets that have been

eaten are set to 1, otherwise left at 0. Fitness improves as more food pellets are eaten. Fitness is a maximization function normalized in $[0, 1]$. It is calculated by dividing the number food pellets eaten with the starting number of food pellets on the map.

5.3 Results

This section will review the results of the application of the baseline measures, E_b and R_b, to three benchmark problems across three GP variants, as listed previously. It is meant to demonstrate how a GA can be adapted to assess its evolvability and robustness while preserving its problem specific representation. Finally, the fitness of each algorithm will be evaluated against the benchmark problems, and the results compared with E_b in order to validate a possible correlation. The results presented are representative of the specific configurations used for each algorithm. It is possible to achieve different results with different configurations of the algorithms, such as changing the number of genes or the sets of functions and terminals. However, the proposed configurations are meant to provide a fair basis of comparison for the GP variants.

5.3.1 Generating random phenotypes

To conduct a baseline evolvability walk, a random structural phenotype is needed. It is random in the sense that it is chosen randomly from the set of all possible phenotypes, given the structural constraints of the genotype (such as its representation, the number of genes and the sets of functions and terminals). So, if two algorithms use the same sets of functions and terminals but different representations (e.g., GEP and CGP), they may not have access to the same structural phenotypes. For this reason, random phenotype selection will be sampled uniformly from the available phenotypes for each experiment, made of the nine combinations of three GP variants and three test problems.

For each experiment, one million random phenotypes were generated by conducting 10,000 baseline robustness walks, for 100 steps (mutations) each, and the entire population is kept.

This was done in order to find more similar structures that can differ by only a few genes, but with large enough sample it should not matter; one could simply generate a million random individuals and get a similar distribution. The resulting samples, grouped by problem, are shown in figures 5.7, 5.8 and 5.9. These figures show short length programs are much more common in phenotype space. The ability to have a more diverse set of phenotypes, based on length, might contribute to evolvability.

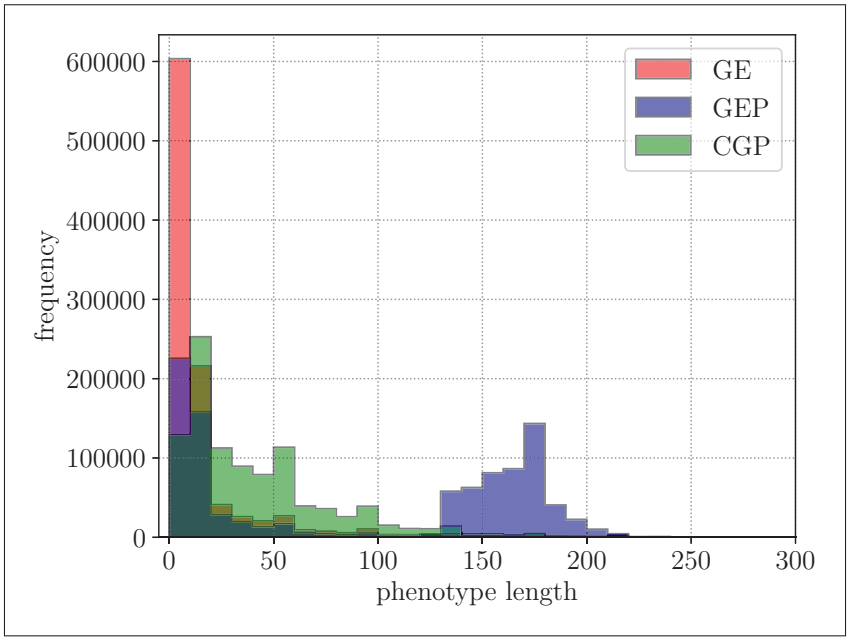


Figure 5.7 Distribution of the size of random phenotypes for GE, GEP and CGP, using the function set of the artificial ant problem

The distributions might give a hint about evolvability, but since E_b seeks to assess the ability to move to a different phenotype, the random target phenotypes will be drawn uniformly from the set of unique phenotypes $\{P\}_{\neq}$. The diversity of phenotypes found in the space of each algorithm provides further information on the robustness of the GP languages. The numbers of unique phenotypes for each experiment are presented in Figure 5.10

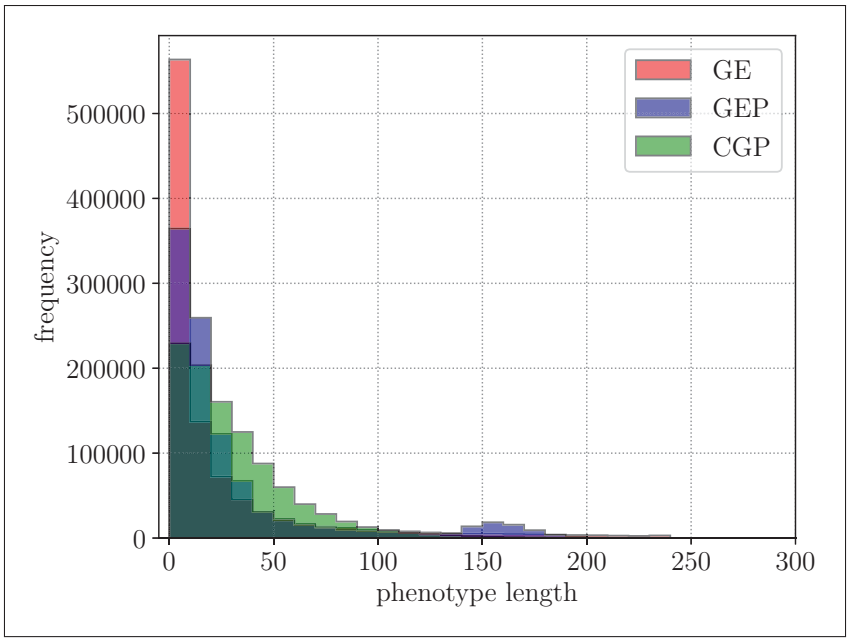


Figure 5.8 Distribution of the size of random phenotypes for GE, GEP and CGP, using the function set of the boolean parity problem

5.3.2 Stabilization

Stabilization of both E_b and R_b are achieved within 1,000 walks of 100,000 steps each, following the empirical process described in Section 3.2.2

5.3.3 Baseline measures

The results for baseline evolvability are presented in Figure 5.11. For the artificial ant problem, GE had significantly higher evolvability then the other GP languages, whereas CGP showed the highest evolvability in the boolean parity problem. All three algorithms performed similarly for the symbolic regression problem. This shows that the size of the phenotype space does not have an impact on its evolvability and most importantly how much the environment (set of functions and terminals available) can influence evolvability of an algorithm.

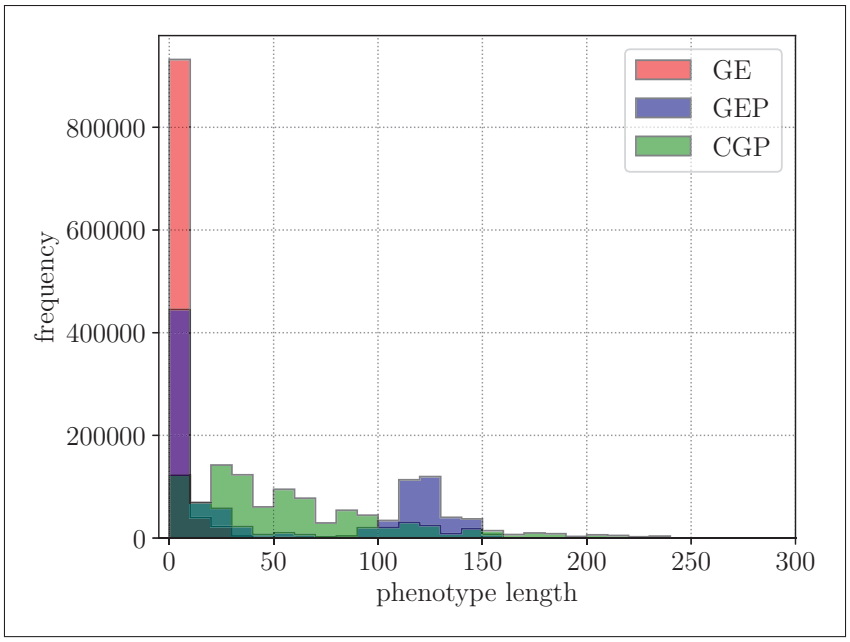


Figure 5.9 Distribution of the size of random phenotypes for GE, GEP and CGP, using the function set of the symbolic regression problem

The results for baseline robustness are presented in Figure 5.12. The scatter plots and ellipses are similar across all problems for a given algorithm. GE is somewhat robust, with an almost vertical ellipse orientations, as the sampled neighbors stay far from the starting genotype and show minimal changes in the phenotype. This is likely due to the low number of unique phenotypes found and the concentration of short length random phenotypes for GE. GEP shows weaker robustness than GE. It boasts larger ellipses, indicating more spread in the sampled neighbors. As shown in the scatter plots of GEP, there are always two clusters appearing both with low and high phenotype distances (the second cluster for the parity problem is barely visible, but sits right outside the upper bound of the largest ellipse). This also correlates with the distributions of phenotypes lengths, where GEP had a peak at short length and one at high length. The higher number of unique phenotypes found with GEP can also be influencing its baseline robustness. For the regression problem, the maximum genotype distance D_g for GEP is 30, as there is only one terminal, preventing any mutation to the tail of the genotype. CGP demonstrates the strongest robustness of the three algorithms. For all problems, the phenotype distances D_p of

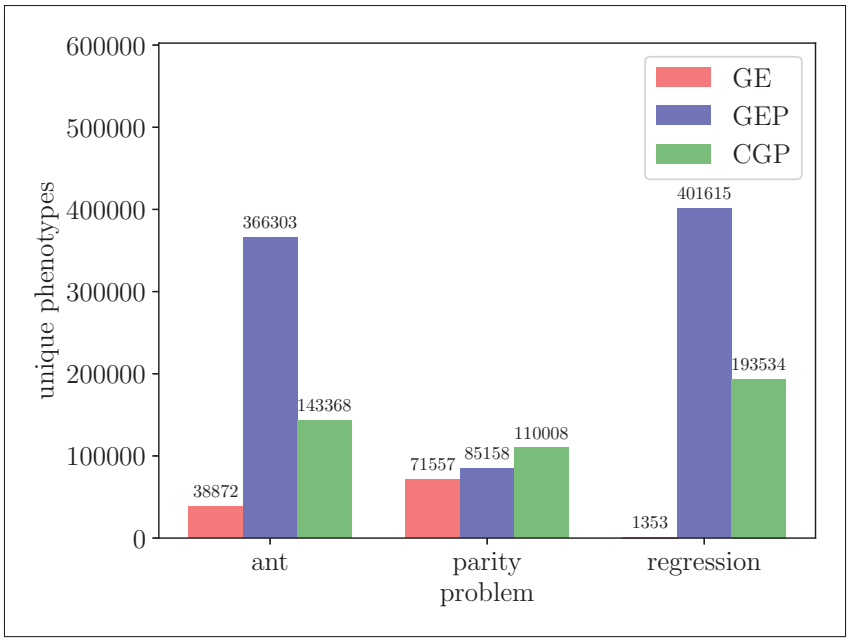


Figure 5.10 Number of unique phenotypes found for each of the three sets of problem functions, across GE, GEP and CGP

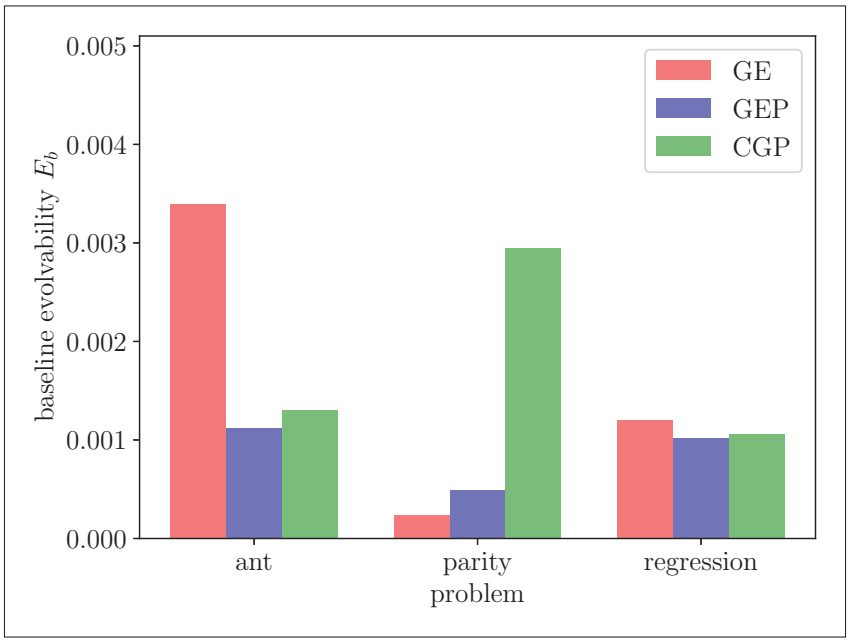


Figure 5.11 Baseline evolvability results for each of the three sets of problem functions, across GE, GEP and CGP

CGP are close to zero, and the ellipse orientations are almost perfectly horizontal, meaning that changes in genotype have negligible impact on the phenotype.

Based on this assessment, there is no indication that E_b and R_b are correlated. Evolvability for a given GP language will depend on the set of functions and terminals available to build programs, whereas robustness is driven by the choice of representation of the algorithm (how the genome is decoded to build a solution).

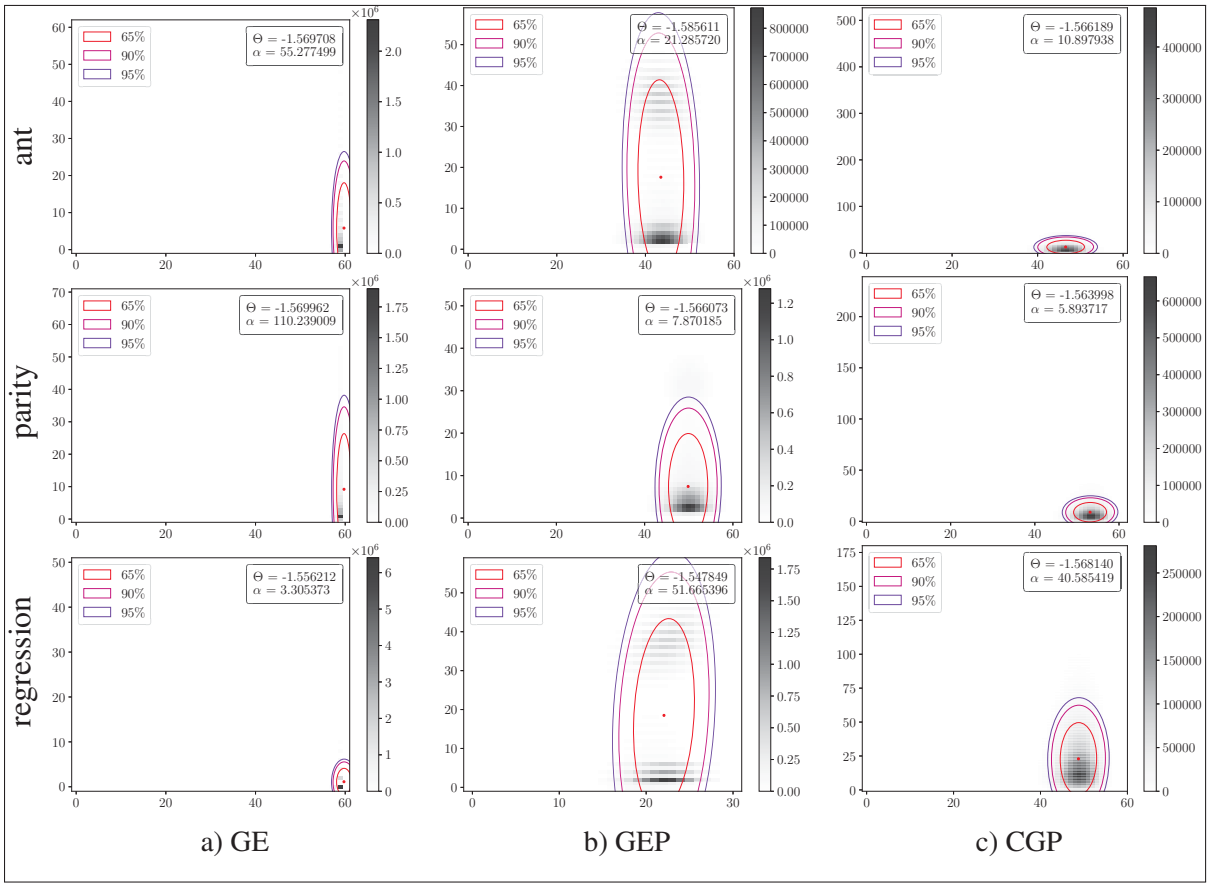


Figure 5.12 Baseline robustness results for the three problems across GE, GEP and CGP, where the horizontal axis is the genotype distance D_g and the vertical axis is the phenotype distance D_p

5.3.4 Exploring correlation with fitness

Considering that evolvability is the ability to move from one phenotype to another quickly, then there it is possible that a correlation exists between evolvability and fitness. In order to find if there is correlation between the ability to find a fit individual given fixed environment and evolvability, the fitness will be evaluated for each problem with the three GP variants, using the most common EA parameters for each GP language.

The performance of the algorithms will be measured using average fitness at termination (AFT). To measure AFT, an EA is run multiple times and the resulting best fitness for each run are averaged. There exists other performance metrics for EAs, such as the success rate and average number of evaluations to solution, but AFT was chosen since it does not require the EA to find an ideal solution.

The number of runs, size of populations and number of generations are set to allow the same number of total evaluations (10^8) for each GP language as in the E_b experiments. This configuration, presented in Table 5.5, will be the same for all the fitness experiments. The choices of parent selection mechanisms and genetic operations are based on the common usage found in the literature, and presented in the following sections.

Table 5.5 Common EA parameters used

Parameter	Value
Runs	100
Population	100
Generations	10,000

5.3.4.1 GE configuration

In GE, one of the most common configurations is a combination of one point crossover and probabilistic mutation, and parent selection is done with tournament selection. The tournament size is set to seven, mutation probability to 1% and probability to apply the crossover operation

is set to 90%. Tournament selection is done by picking a number of individuals at random from the population and keeping the one with the best fitness, as illustrated in Figure 5.13. One point crossover is achieved by selecting a random position in the genotype and exchanging the sections between two parents, as illustrated in Figure 5.14, and probabilistic mutation uses the probability to assess each gene for mutation, as illustrated in Figure 5.15, where the blue arrows demonstrate which gene did not pass the mutation probability threshold, so it is kept as is, while red crosses indicate the mutation probability threshold was achieved, leading to a mutation of the gene (bit flip in the case of a binary string). Figure 5.16 presents the evolutionary process that produces a new generation of the population.

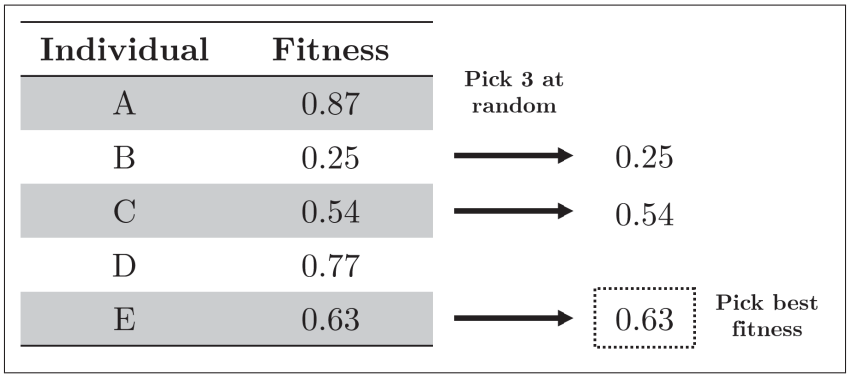


Figure 5.13 Example of selecting a parent from a population using tournament selection, with a tournament size of three

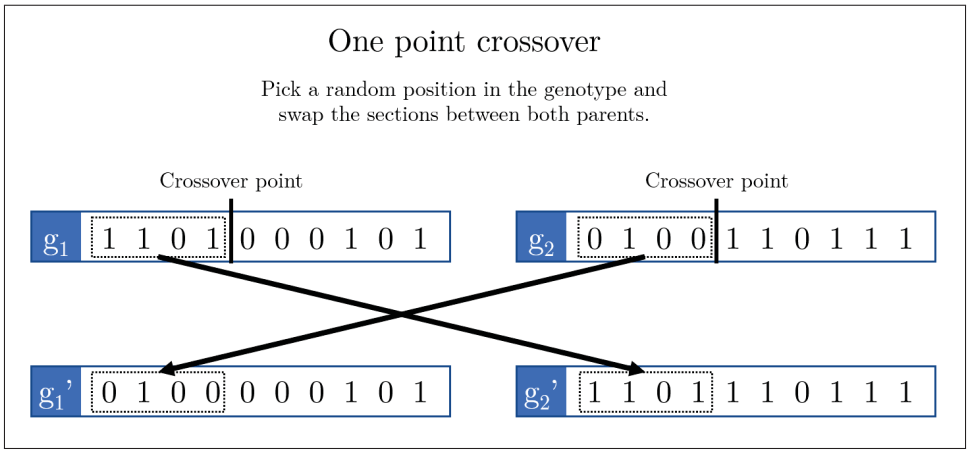


Figure 5.14 One point crossover genetic operation

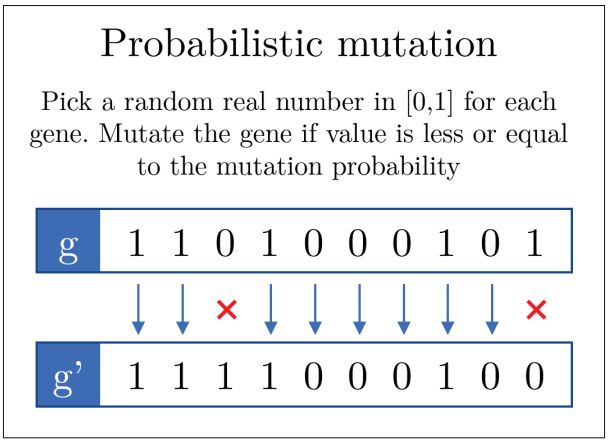


Figure 5.15 Probabilistic mutation genetic operation

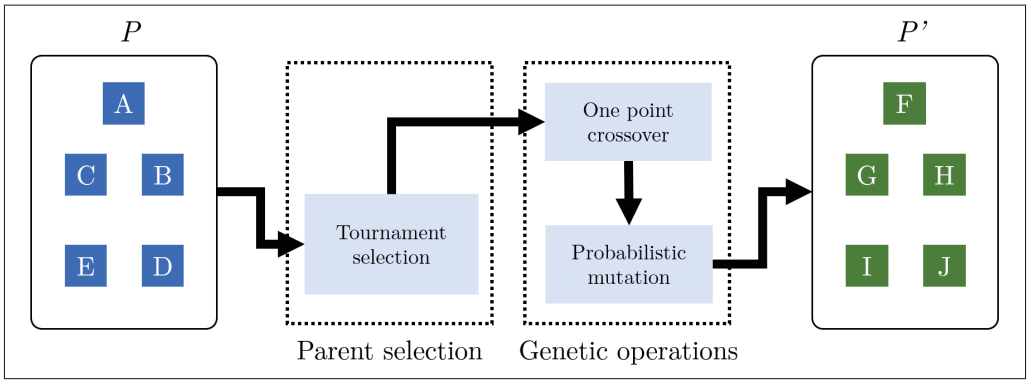


Figure 5.16 Complete evolutionary process used in GE to produce a new generation

5.3.4.2 GEP configuration

The evolutionary process suggested by the author of GEP is more complex than GE and CGP. First, elitism is used to always keep the current best individual into the next generation. Elite individuals are always excluded from parent selection. Parent selection is done by using the fitness proportionate selection method, as illustrated in Figure 5.17. In fitness proportionate selection, often called roulette wheel selection, each individual gets a probability of being picked proportional to its fitness, then the wheel is *spun* and the individual where the arrow lands gets selected as the parent. Then, multiple genetic operations are applied in sequence, each with their

own probability of being applied. First, probabilistic mutation, as defined for GE, is applied with a 4.4% probability. Then inversion, the process of inverting (or reversing) a section of the genotype, as illustrated in Figure 5.18a, is applied with a probability of 10%. After that, transposition, the process of moving a random subsection of the genotype at a different position, as illustrated in Figure 5.18b, is applied with a probability of 10%. It is also possible to do crossover in GEP, but only when there is more than one chromosome (which is not the case in this experiment).

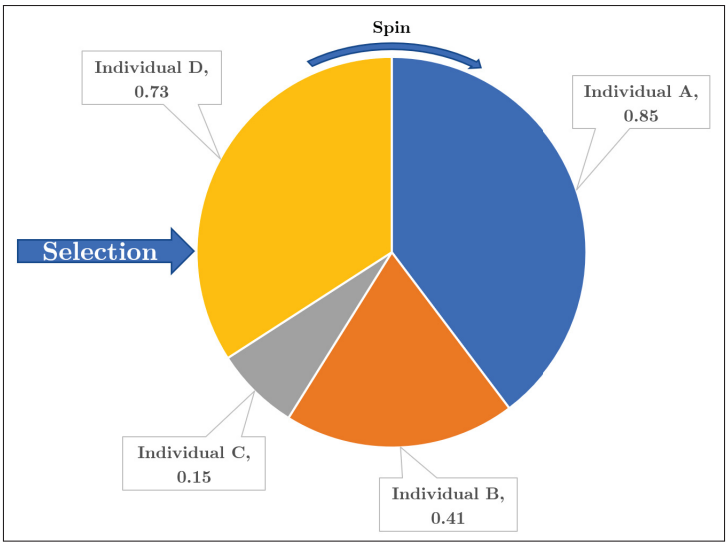


Figure 5.17 Fitness proportionate selection

Figure 5.19 presents the evolutionary process that produces a new generation of the population.

5.3.4.3 CGP configuration

CGP recommends a simple $\mu + \lambda$ Evolutionary Strategy (ES) to evolve a population. In this evolutionary process, μ parents are selected from the population and inserted into the new generation. Then, λ offspring are generated by mutating the parents in the new generation. This process is illustrated in Figure 5.20. Parent selection is done uniformly at random among all the individuals of equally best fitness, as presented in Figure 5.21. Probabilistic mutation, as

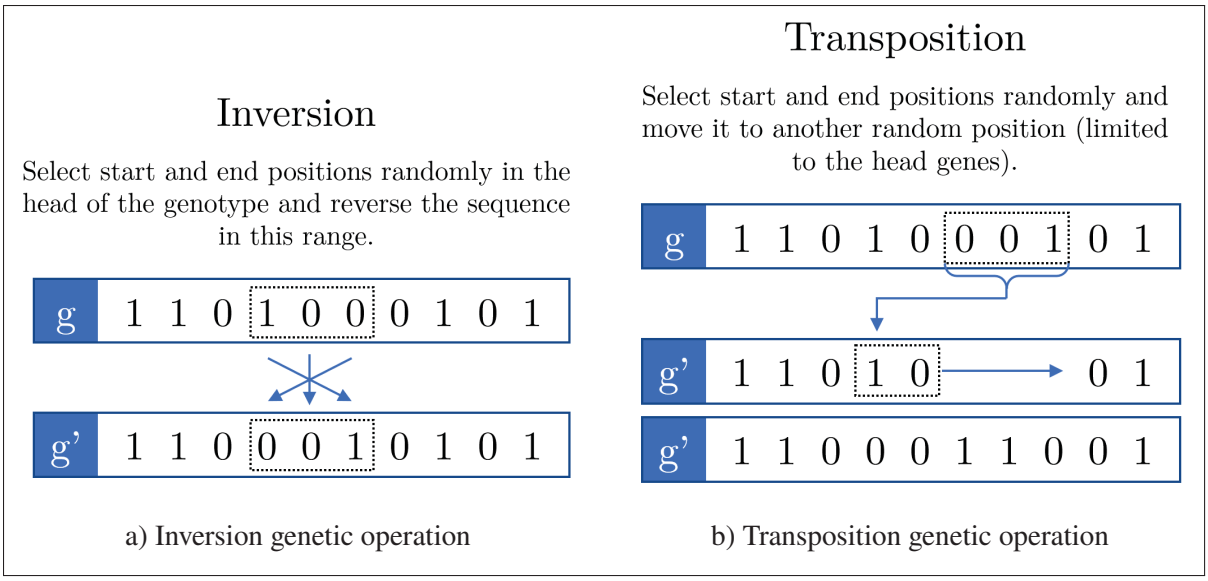


Figure 5.18 Genetic operations used in GEP

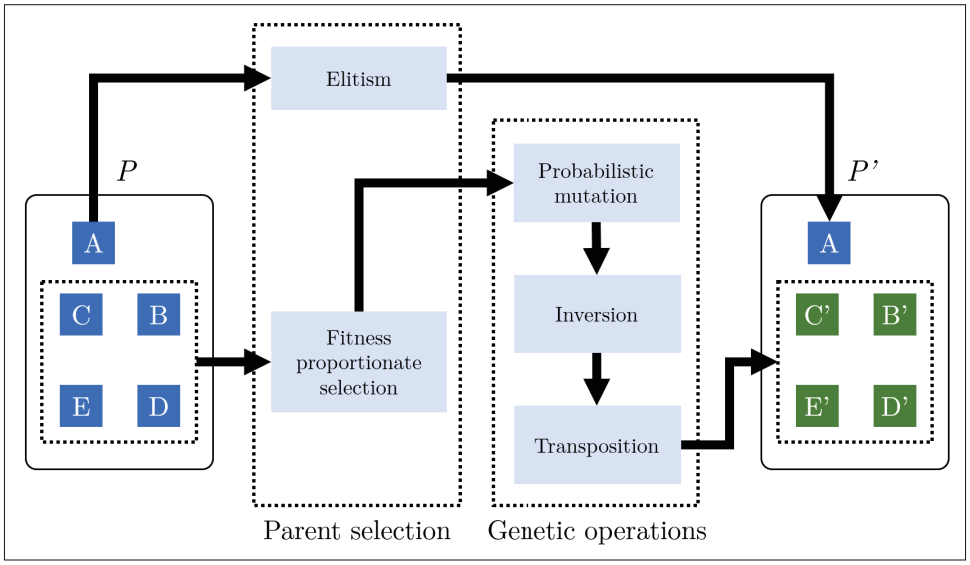


Figure 5.19 Complete evolutionary process used in GEP to produce a new generation

presented earlier in Figure 5.15, is used with a probability of 4%. For this experiment, $\mu = 1$ and $\lambda = 99$ in order to have a total population of 100 at each generation.

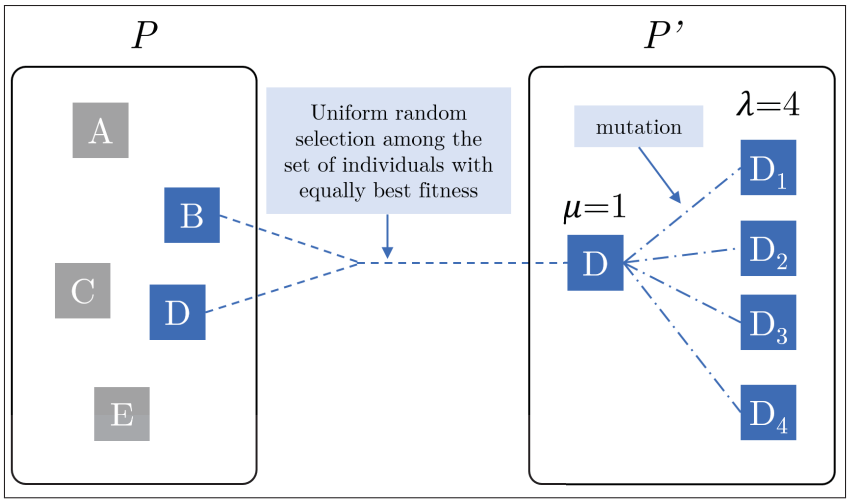


Figure 5.20 Illustration of the $\mu + \lambda$ breeding pipeline used in CGP

5.3.4.4 Average fitness at termination

Results are presented in figures 5.22, 5.23 and 5.24 for the artificial ant, boolean parity and symbolic regression problems respectively. These figures present the average across all runs of the best fitness in the population at each generation, and the shaded areas represent the variance. For the artificial ant problem, GEP shows the best performance, followed closely by CGP, while GE has more difficulty getting close to ideal performance. For the boolean parity problem, CGP is the only GP language to get close to ideal fitness (1), while GE and GEP stagnate at an approximate fitness of 0.75. For the symbolic regression problem, both GEP and CGP are able to achieve ideal fitness quickly, while GE barely passes an average fitness of 0.6 within the allowed 10,000 generations. These findings are aligned with Oltean & Grosan (2003), except for the fact that GEP performed closer to CGP in two of the three problems.

Comparing AFT against the baseline evolvability obtained previously in Section 5.3.3, there seems to be no correlation between the two measures. Figure 5.25 presents the results of E_b and AFT for GE. While both measures are at their highest in the artificial ant, symbolic regression shows a lower AFT but higher E_b than boolean parity. Figure 5.26 presents the results for GEP. For GEP, there seems to be some degree of correlation between E_b and AFT. Figure 5.27 presents

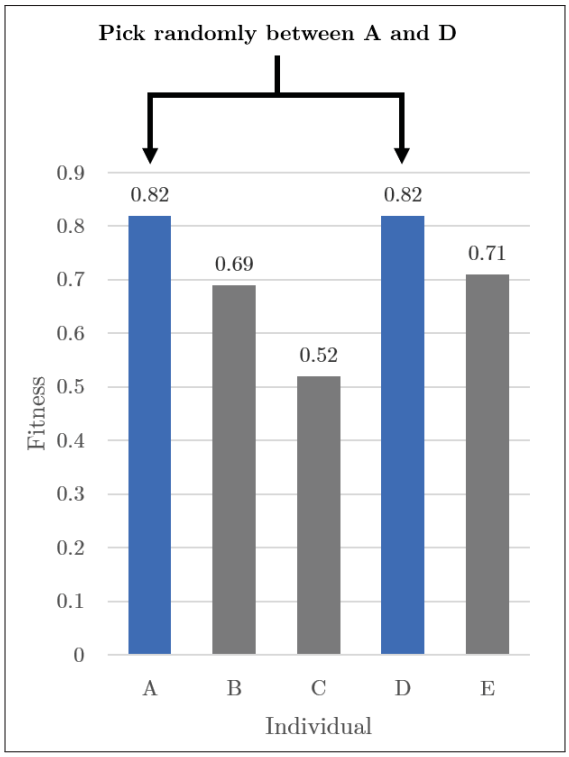


Figure 5.21 Example of uniform random selection among the individuals with equally best fitness

the results for CGP. In that case, CGP demonstrated high AFT across all problems, while E_b in the artificial ant and symbolic regression problems are approximately half of the boolean parity problem. This supports the hypothesis that evolvability and fitness are not correlated.

5.4 Conclusion of the application of baseline measures to GP variants

This section demonstrated the application of the baseline measures to three different benchmark problems for three different variants of GP. The results showed that evolvability is dependent on both the choice of representation of the genotype and the set of functions and terminals, while robustness is reflective of the phenotype landscape. Both baseline measures are thus complementary and provide valuable insight to understand how different genotype representations can perform. Furthermore, it was demonstrated that evolvability is not correlated with fitness.

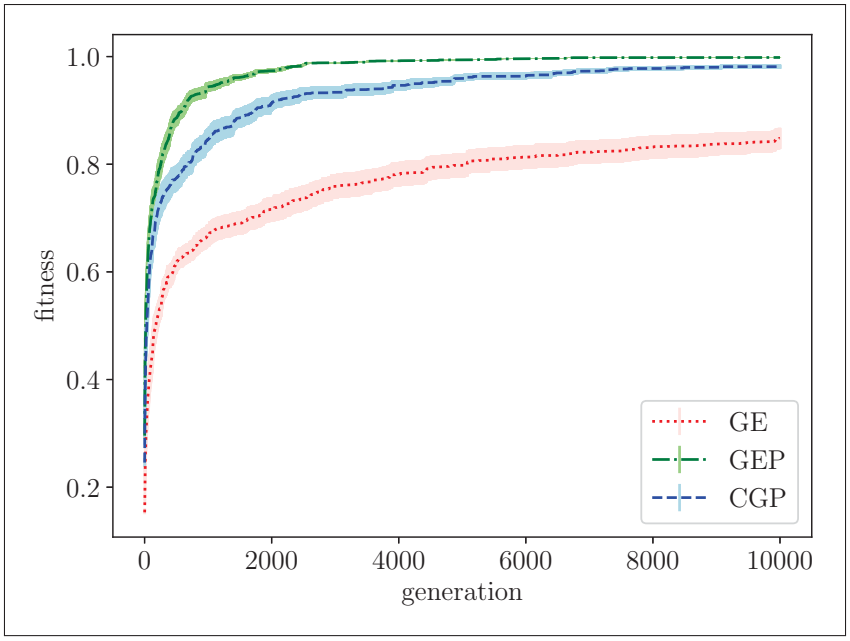


Figure 5.22 Average fitness on the artificial ant problem

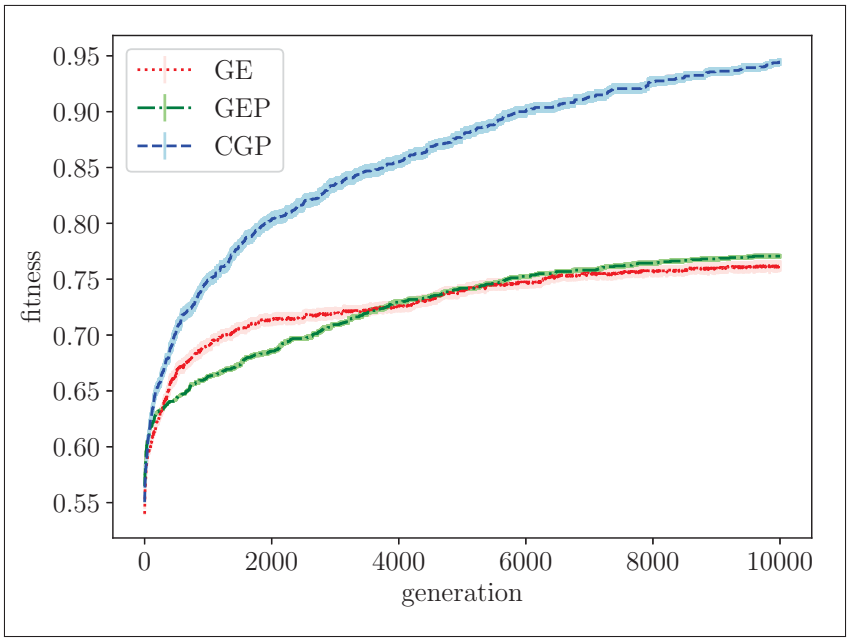


Figure 5.23 Average fitness on the boolean parity problem

E_b and fitness are two different measures that serve different purposes. It is important to note

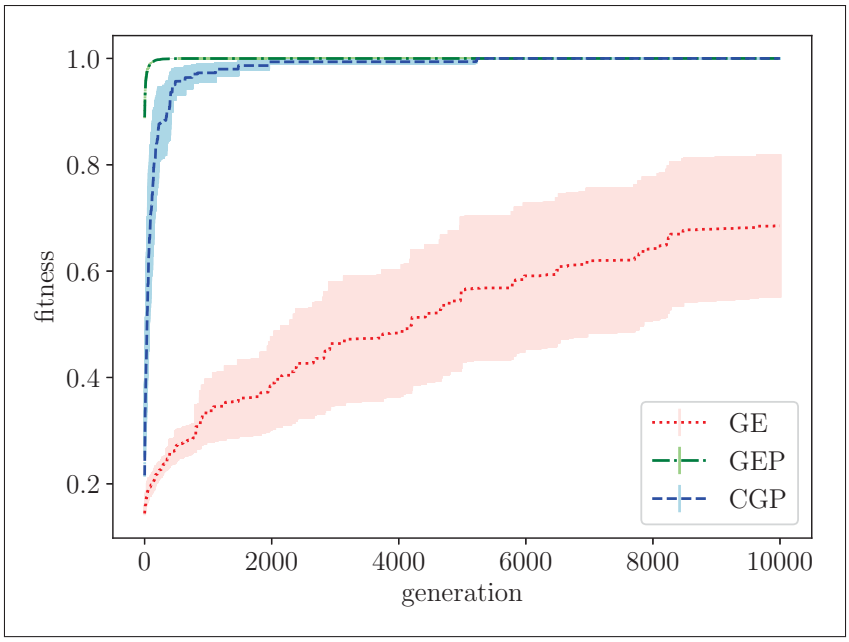


Figure 5.24 Average fitness on the symbolic regression problem

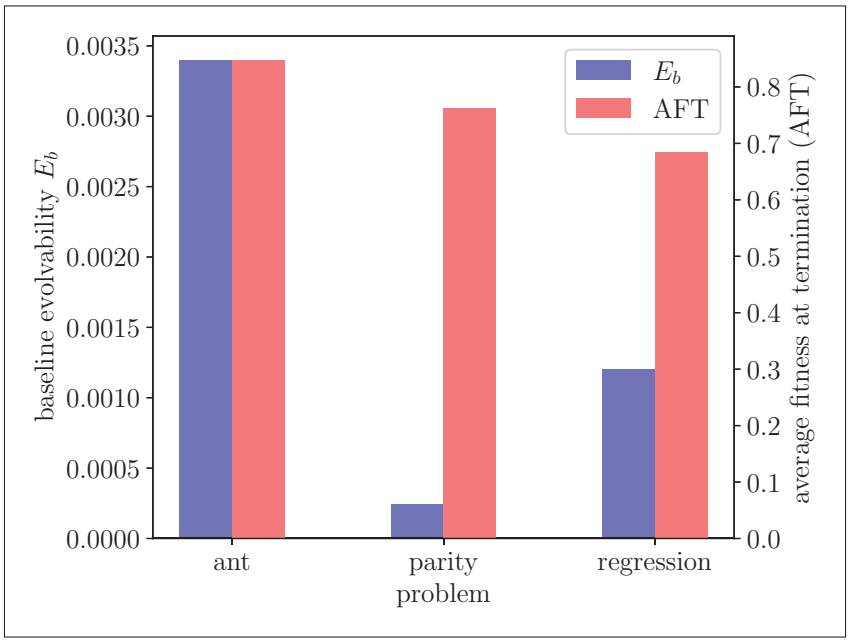


Figure 5.25 Comparison of E_b and AFT for GE across the three benchmark problems

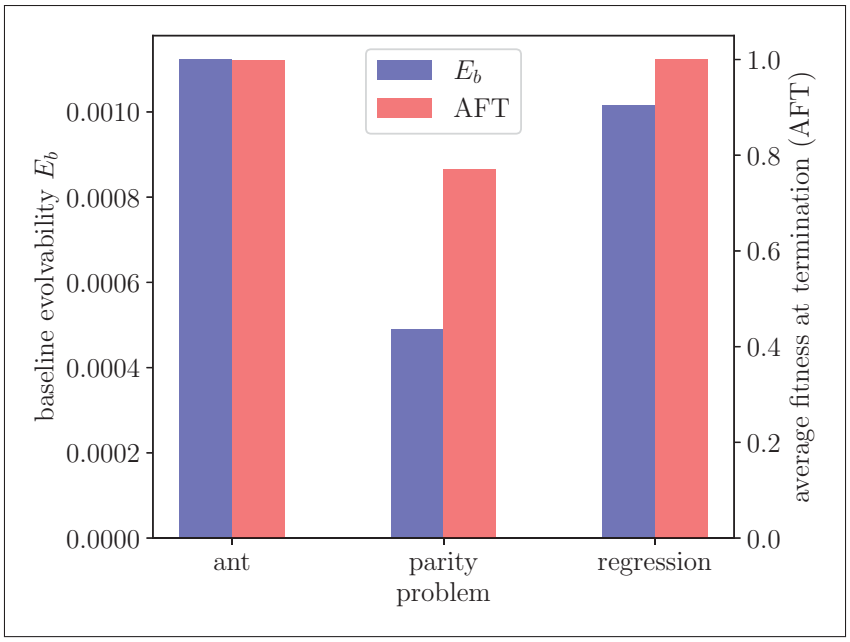


Figure 5.26 Comparison of E_b and AFT for GEP across the three benchmark problems

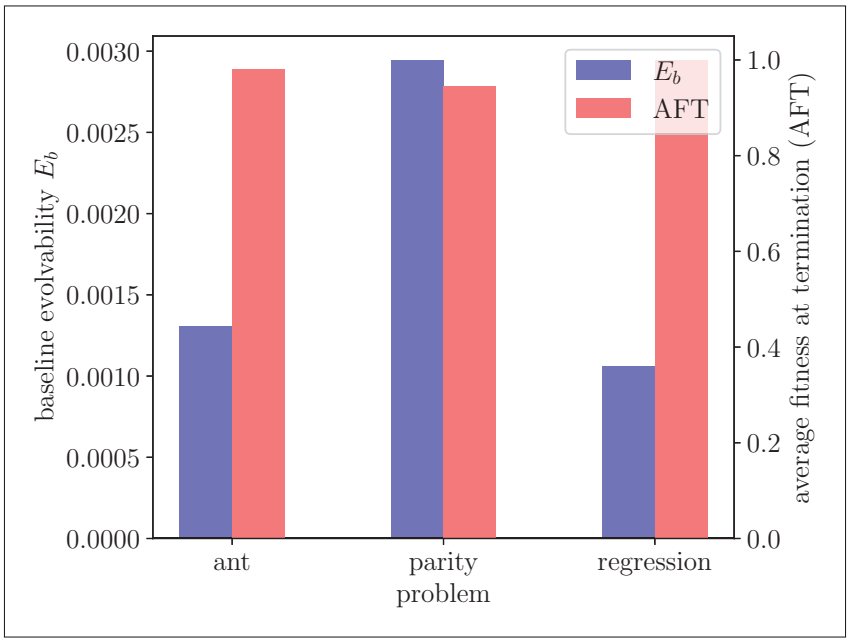


Figure 5.27 Comparison of E_b and AFT for CGP across the three benchmark problems

that the evolutionary process used by each GP language varied from the proposed approach of baseline evolvability.

CONCLUSION AND RECOMMENDATIONS

The popularity of any tool or method is driven by its ease of use and the magnitude of its results. In the case of performance metrics evaluating evolvability and robustness, frameworks proposed so far required significant additional manipulations to an existing algorithm in order to get the results. The baseline measures proposed in this research can be applied with almost no changes required to an existing GA, while providing meaningful information on the evolvability and robustness of an algorithm, allowing researchers to experiment and compare different genotype and phenotype encodings. The baseline measures are still susceptible to hyper-parameters configuration (e.g., stabilization point), however once discovered and aligned for each algorithm, they provide a simple and fair comparison without inducing any bias.

First, the limitations of the classical measures proposed by Wagner in Wagner (2008), even when generalized, were demonstrated using RNA sequence folding problem. The classical measures suffer from incomplete information due to their restricted neighborhood view (only one neighborhood assessed at a time) and do not reflect the natural evolution pattern that accumulates genotypic changes sequentially over a period of time. Another major drawback is the need for an inverse mapping function to identify the genotypes that map to a given phenotype in order to evaluate phenotype evolvability.

To address this, new measures of evolvability and robustness (namely the baseline measures E_b and R_b) were defined. These new measures assess the amount of change in genotype and phenotype across an evolutionary walk by measuring the genotype and phenotype distances. This approach makes it suitable for applications that leverage Genetic Algorithms as it can be applied with seemingly no changes to the GA. Applications of the baseline measures were demonstrated across various domains such as RNA sequence folding, system modeling and Genetic Programming. E_b provides a single value that is easy to understand, but is mostly relevant when comparing the evolvability of multiple systems. R_b is more nuanced, as a figure

is needed to fully capture how phenotypes change along genotype mutations. This is partly addressed by using the orientation Θ and aspect ratio α of the resulting ellipse as proxies for R_b.

It has been demonstrated that baseline measures can provide more information on evolvability and robustness than Wagner measurements, but still have certain drawbacks. Baseline evolvability depends on its ability to reach the target phenotype. As with other evolutionary algorithms, the mutational walk may stall in a local optimum, in which case the walk may continue indefinitely without ever reaching its target. This has a negative impact on the resulting evolvability, since E_b considers the number of steps taken. Baseline robustness has the disadvantage of over-representing high genotypic distances. As the mutational walk continues, the scatterplots of the distances show stabilization after many changes in genotype. This provides useful information about the overall phenotype landscape, but robustness should focus at least uniformly on all neighborhoods, as it is more interesting to understand how many mutations can be sustained before a change in phenotype occurs.

In the process of applying the baseline measures, the need for a generic modeling approach arose to ensure versatility, which led to the definition of the BNK System. This system can be used to model a wide range of problems. It was able to find a diverse set of oscillating genetic circuits by defining measures of symmetry and robustness of an oscillating circuit. Applying the baseline measures to the specific problem of evolving perfect oscillators, it was possible to measure the ease of evolving perfect oscillators using BNK and the robustness of these perfect oscillators under mutation. As an extension to the NK System, the multi-modality of the fitness surface of BNK Systems was also demonstrated by increasing the number of inputs for each gate in the system.

The main requirement when evaluating the baseline measures is that measuring the distance between two genotypes (D_g) should consider the genetic operators available and create the shortest sequence of genetic operations needed to change one genotype into the other. This is

simple when only a single gene is mutated randomly per generation, however most evolutionary systems leverage more advanced operators such as crossover. Finding the shortest distance between two graphs is a problem that is yet to be solved, and the reason why the proposed measures are called "baseline": they assess evolvability and robustness using the smallest genotypic change possible.

It may be possible to further expand the baseline measures. For now, the applications have been limited to genotypes of fixed lengths and distance between two genotypes measured using the Hamming distance. In the case where it is possible to insert or delete a gene, edit distances, such as the Levenshtein distance, could be used to evaluate the distance and capture an insertion or deletion the same way a single gene mutation is measured. This would also be required in the cases where a mutation could change a function arity, since replacing a function using two arguments with a function using three arguments would require an additional insertion to fill the new argument. If in order to effectively measure these types of changes, there needs to be a way to know how many genes were removed or added through each operation.

Although it was demonstrated that fitness and evolvability are not correlated, it would be worthy to investigate if embedding measures of robustness or evolvability within a genotype representation can improve the performance of the algorithm to solve a wide range of problems. This could be done through a measure of effort, such as the number of mutations accumulated so far, in the fitness evaluation to increase pressure to find evolvable solutions. Another avenue would be measuring the impact of each gene on evolvability and robustness to get a better understanding of the genotype representation in a given algorithm. That way, it might be possible to assign different weights to certain genes in order to control evolvability and robustness. Or another possibility would be to embed a measure of population diversity that can be used by an EA to dynamically adjust the probabilities of genetic operators (e.g., focus on crossover for exploration and mutation for exploitation).

Ultimately, the goal is to provide standardized measures of evolvability and robustness across multiple evolutionary domains, and incidentally a framework that allows for easy assessment of the representation used by an evolutionary system to solve a problem. This is possible using E_b and R_b by leveraging a simple genetic operator (baseline mutation), as was demonstrated in this research.

APPENDIX I

ADDITIONAL MATERIAL

1. Programs and data

The software and scripts used to generate the results are available at https://github.com/remz1337/VRE_Experiment. The data is available upon request only due to its large size.

2. Symmetry and robustness pseudo-code

Compilation of functions in pseudo-code required to compute symmetry and robustness of BNK oscillators.

2.1 Loop identification

From the phenotype of a BNK system (State Transition Diagram), identify each loops by traversing the STD with each state. Details in I-1.

2.2 Gate oscillation

This function allows to build the oscillation strings (binary strings such as “01011”) for each gate in a given loop. Details in I-2.

2.3 System symmetry

Leveraging loop identification and the derived oscillation for each gate in each loop, compute the symmetry by counting the lengths of passive (0) and active (1) periods. Details in I-3.

2.4 Identify equivalent oscillations

Algorithm to compare two oscillation strings and assess if they have the same behaviour. First, the lengths are aligned by repeating the shortest oscillation string, then one of the oscillation

string is rotated at every possible character in the string and compared against the other oscillation. For example, “01” and “0101” have the same behaviour. Details in I-4.

2.5 System robustness

Robustness is obtained by counting for each gate the number of states in the STD that produce the same behaviour. Details in I-5.

Algorithm-A I-1 Loop identification

```
1  Initialize the empty map of loops M_L ;      /* Map of loops used to identify
     every unique loop and count the number of states that transition
     into each loop.  */
2  Total number of states X = 2^N − 1, where N is the number of gates ;  /* States are
     represented in binary to show the individual gate value, but
     encoded in decimal for easier manipulation.  */
3  for S ← 0 to X do
4      Current state S_c = S;
5      Initialize empty list of visited states V;
6      while S_c ∉ V do
7          V ← S_c;
8          S_c = Θ(S_c) ;        /* Θ is the mapping function from the state
             transition map.  */
9      end while
10     I_s ← index of S_c in V ;  /* Identify the start index of the loop from
         the list of visited states.  */
11     L_s = V[I_s, ...] ;  /* Extract the looping states from all the visited
         states */
12     I_min = index of min(L_s) in L_s ; /* Identify the smallest state (decimal
         value) in the loop.  */
13     L_s = rotate(L_s, I_min) ;  /* Rotate the array of looping states so that
         it always start at the same point, no matter where the initial
         entry point was.  */
       ;     /* Increase the count of number of states transitioning to
         this specific loop.  */
14     if L_s ∈ M_L then
15         M_L(L_s) + +
16     else
17         M_L(L_s) = 1
18     end if
19 end for
```

Algorithm-A I-2 Gate oscillation in a loop

Data: L: Ordered list of states in the loop, g:gate index
Result: O: Binary string representing the gate oscillation

```
O = [];      /* Initialize empty string for the gate oscillation.  */
foreach state S ∈ L do
    B_s =binary(S);      /* Convert the state decimal value to a binary
     array.  */
    B_g = B_s[g];      /* Retrieve the binary value of the gate from the
     array.  */
    O+ = B_g;           /* Append the binary value of the gate to the
     oscillation string.  */
end foreach
return O
```

Algorithm-A I-3 Symmetry

```
1  Identify all loops using algorithm I-1
2  foreach loop L in the state transition diagram do
3      foreach gate G do
4          O ← Gate oscillation using algorithm I-2;
5          L_l ← size of O;
6          if no oscillation then
               ;          /* O is either all passive (0) or active(1).  */
7              Local symmetry S_L = 0;
8              Period length L_p = 1;
9          else
10             Rotate O if needed so that it starts with a passive (low) period and end with
                an active (high) period;
11             Identify pairs of consecutive passive and active periods;
12             foreach pair p do
13                 T_p ← length of the passive periods of the pair;
14                 T_a ← length of the active periods of the pair;
15                 Period length of the pair L_p = T_p + T_a;
16                 Local weight W_L ← L_p / L_l;
17                 Local symmetry S_L ← min T_p,T_a / max T_p,T_a;
18             end foreach
19         end if
20         Gate symmetry S_g ← ∑(S_L · W_L) · min L_p / max L_p
21     end foreach
22     Loop symmetry S_l ← ∑ S_g / # of gates;
23     Loop weight W_l ← loop length / ∑ all loop sizes;
24 end foreach
25 Overall symmetry S_o ← ∑(S_l · W_l);
```

Algorithm-A I-4 Identify equivalent oscillations

Data: g: gate index
Result: E: list of pairs of loops that have the same oscillation
$E = \{\}$; /* Initialize empty list of pairs of loops that express the same oscillation. */
foreach *pair of loops* L_i, L_j *in the state transition diagram* **do**
  $O_i \leftarrow$ Gate oscillation using algorithm I-2 of loop L_i;
  $O_j \leftarrow$ Gate oscillation using algorithm I-2 of loop L_j;
  $\Delta_i =$**length**(O_i);
  $\Delta_j =$**length**(O_j);
  if $\Delta_i > \Delta_j$ **then**
    longest $O_l = O_i$;
    shortest $O_s = O_j$;
  else
    longest $O_l = O_j$;
    shortest $O_s = O_i$;
  end if
  $\Delta_l =$**length**(O_l);
  $\Delta_s =$**length**(O_s);
  if $\Delta_s \mod \Delta_s == 0$ **then**
    ; /* If they cannot align then it's impossible for both oscillations to behave the same. */
    $O_s = O_s * (\frac{\Delta_l}{\Delta_s})$; /* Repeat the short string in order to check if the oscillation is the same but repeated over multiple periods. */
    for $r \leftarrow 0$ *to* Δ_s **do**
      ; /* Check if the oscillations can match by rotating the shortest one. */
      if $O_s == O_l$ **then**
        $E \leftarrow (L_i, L_j)$;
      end if
      $O_s =$**rotate**$(O_s, 1)$; /* Rotate oscillation string by one character. */
    end for
  end if
end foreach
return E

Algorithm-A I-5 Robustness

```
1  Identify all loops using algorithm I-1;
2  foreach gate g do
3      E ←List of pairs of loops that have the same oscillation, using algorithm I-4;
4      S = {};          /* Initialize empty list of number of states that
        transition to a loop.  */
5      foreach loop L_i in the state transition diagram do
6          S_i ←number of states that lead to loop L_i;
7      end foreach
8      foreach pair (L_i, L_j) of equivalent loops in E do
           ;   /* Merge the number of states that lead to a loop if they
            have the same oscillation.  */
9          S_i = S_i + S_j;
10         S_j = 0;
11     end foreach
12     Fill S with zeros so that the size of S equals X(the number of states in the state
        transition diagram);
13     σ_g =std. deviation(S);
14     S_max = {X};
15     Fill S_max with zeros so that the size of S_max equals X(the number of states in the
        state transition diagram);
16     σ_max =std. deviation(S_max);
17     Gate robustness R_g = σ_g / σ_max;
18 end foreach
19 Overall robustness R_o = ∑ R_g / # of gates;
```

BIBLIOGRAPHY

Altenberg, L. et al. (1994). The evolution of evolvability in genetic programming. *Advances in genetic programming*, 3, 47–74.

Banzhaf, W., Nordin, P., Keller, R. E. & Francone, F. D. (1998). *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*. Morgan Kaufmann Publishers Inc.

Bédard-Couture, R. & Kharma, N. N. (2019). Playing Iterated Rock-Paper-Scissors with an Evolutionary Algorithm. *IJCCI*, pp. 205–212.

Bhandari, D., Murthy, C. & Pal, S. K. (2012). Variance as a stopping criterion for genetic algorithms with elitist model. *Fundamenta Informaticae*, 120(2), 145–164.

Brameier, M., Kantschik, W., Dittrich, P. & Banzhaf, W. (1998). SYSGP-A C++ library of different GP variants. *Technical Rep. No. CI-98/48, Collaborative Research Center*, 531.

Calcott, B. (2014). Engineering and evolvability. *Biology & Philosophy*, 29(3), 293–313.

Churkin, A., Retwitzer, M. D., Reinharz, V., Ponty, Y., Waldispühl, J. & Barash, D. (2018). Design of RNAs: comparing programs for inverse RNA folding. *Briefings in bioinformatics*, 19(2), 350–358.

Csaszar, F. A. (2018). A note on how NK landscapes work. *Journal of Organization Design*, 7(1), 1–6.

Daida, J. M., Ward, D. J., Hilss, A. M., Long, S. L., Hodges, M. R. & Kriesel, J. T. (2004). Visualizing the loss of diversity in genetic programming. *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, 2, 1225–1232.

Dawkins, R. (2019). The evolution of evolvability. In *Artificial life* (pp. 201–220). Routledge.

De Visser, J. A. G., Hermisson, J., Wagner, G. P., Meyers, L. A., Bagheri-Chaichian, H., Blanchard, J. L., Chao, L., Cheverud, J. M., Elena, S. F., Fontana, W. et al. (2003). Perspective: evolution and detection of genetic robustness. *Evolution*, 57(9), 1959–1972.

Eiben, A. E., Smith, J. E. et al. (2003). *Introduction to evolutionary computing*. Springer.

Elowitz, M. B. & Leibler, S. (2000). A synthetic oscillatory network of transcriptional regulators. *Nature*, 403(6767), 335–338.

Feldkamp, U., Saghafi, S., Banzhaf, W. & Rauhe, H. (2001). DNASequenceGenerator: A Program for the construction of DNA sequences. *International Workshop on DNA-Based Computers*, pp. 23–32.

Ferreira, C. (2001). Gene expression programming: a new adaptive algorithm for solving problems. *arXiv preprint cs/0102027*.

Ferreira, C. (2002). Gene expression programming in problem solving. In *Soft computing and industry* (pp. 635–653). Springer.

Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A. G., Parizeau, M. & Gagné, C. (2012). DEAP: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, 13(1), 2171–2175.

Garcia-Martin, J. A., Clote, P. & Dotu, I. (2013). RNAiFold: a web server for RNA inverse folding and molecular design. *Nucleic acids research*, 41(W1), W465–W470.

Hu, T. & Banzhaf, W. (2018). Neutrality, robustness, and evolvability in genetic programming. In *Genetic Programming Theory and Practice XIV* (pp. 101–117). Springer.

Kauffman, S. A. (1992). Origins of order in evolution: self-organization and selection. *Understanding origins: Contemporary views on the origin of life, mind and society*, 153–181.

Kim, J. (2023). Efficient graph edit distance computation using isomorphic vertices. *Pattern Recognition Letters*, 168, 71–78.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press.

Kuchaiev, O., Milenković, T., Memišević, V., Hayes, W. & Pržulj, N. (2010). Topological network alignment uncovers biological function and phylogeny. *Journal of the Royal Society Interface*, 7(50), 1341–1354.

Langton, C. (1989). Proceedings of the Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems, ALIFE'87.

Levenshtein, V. I. et al. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8), 707–710.

Lorenz, R., Bernhart, S. H., Höner zu Siederdissen, C., Tafer, H., Flamm, C., Stadler, P. F. & Hofacker, I. L. (2011). ViennaRNA Package 2.0. *Algorithms for molecular biology*, 6(1), 1–14.

Luis, S. & dos Santos, M. V. (2013). On the evolvability of a hybrid ant colony-cartesian genetic programming methodology. *European Conference on Genetic Programming*, pp. 109–120.

Luke, S. [Available for free at http://cs.gmu.edu/~eclab/projects/ecj/]. (1998). ECJ Evolutionary Computation Library.

Luke, S. (2010). The ECJ owner's manual. *San Francisco, California, A user manual for the ECJ Evolutionary Computation Library*, 1–206.

Mayer, C. & Hansen, T. F. (2017). Evolvability and robustness: a paradox restored. *Journal of theoretical biology*, 430, 78–85.

McPhee, N. F., Hopper, N. J. et al. (1999). Analysis of genetic diversity through population history. *Proceedings of the genetic and evolutionary computation conference*, 2, 1112–1120.

McPhee, N. F., Poli, R. & Langdon, W. B. (2008). *Field guide to genetic programming*. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk.

Miller, J. F. et al. (1999). An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. *Proceedings of the genetic and evolutionary computation conference*, 2, 1135–1142.

Miller, J. F. (2020). Cartesian genetic programming: its status and future. *Genetic Programming and Evolvable Machines*, 21(1), 129–168.

Miller, J. F. & Harding, S. L. (2009). Cartesian genetic programming. *Proceedings of the 11th annual conference companion on genetic and evolutionary computation conference: late breaking papers*, pp. 3489–3512.

Nordin, P., Banzhaf, W. et al. (1995). Evolving Turing-Complete Programs for a Register Machine with Self-modifying Code. *ICGA*, 95, 318–325.

Oltean, M. & Dumitrescu, D. (2002). Multi expression programming. *Journal of Genetic Programming and Evolvable Machines*.

Oltean, M. & Grosan, C. (2003). A comparison of several linear genetic programming techniques. *Complex Systems*, 14(4), 285–314.

O'Neill, M., Vanneschi, L., Gustafson, S. & Banzhaf, W. (2010). Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11(3), 339–363.

Paaßen, B. (2018). Revisiting the tree edit distance and its backtracing: A tutorial. *arXiv preprint arXiv:1805.06869*.

Paterson, N. R. (2003). *Genetic programming with context-sensitive grammars*. University of St. Andrews (United Kingdom).

Pawlik, M. & Augsten, N. (2015). Efficient computation of the tree edit distance. *ACM Transactions on Database Systems (TODS)*, 40(1), 1–40.

Pawlik, M. & Augsten, N. (2016a). Tree Edit Distance. Retrieved on 2023-04-27 from: http://tree-edit-distance.dbresearch.uni-salzburg.at.

Pawlik, M. & Augsten, N. (2016b). Tree edit distance: Robust and memory-efficient. *Information Systems*, 56, 157–173.

Pigliucci, M. (2008). Is evolvability evolvable? *Nature Reviews Genetics*, 9(1), 75–82.

Pike, G. (2012). CityHash: fast hash functions for strings. *Stand ford university class slides, October 2012*.

Rdlaw. [File: `SantaFeTrail.gif`]. (2011). Santa Fe Trail problem for artificial ant programming. Retrieved on 2022-03-23 from: https://commons.wikimedia.org/w/index.php?curid=15160912.

Reisinger, J., Stanley, K. O. & Miikkulainen, R. (2005). Towards an empirical measure of evolvability. *Proceedings of the 7th annual workshop on Genetic and evolutionary computation*, pp. 257–264.

Ryan, C., Collins, J. J. & Neill, M. O. (1998). Grammatical evolution: Evolving programs for an arbitrary language. *Genetic Programming: First European Workshop, EuroGP'98 Paris, France, April 14–15, 1998 Proceedings 1*, pp. 83–96.

Sanderson, C. & Curtin, R. (2016). Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software*, 1(2), 26.

Schäling, B. (2011). *The boost C++ libraries*. Boris Schäling.

Schulte, E., Fry, Z. P., Fast, E., Weimer, W. & Forrest, S. (2014). Software mutational robustness. *Genetic Programming and Evolvable Machines*, 15(3), 281–312.

Simson, J. & Mayo, M. (2017). *Open-Source Linear Genetic Programming*. (Ph.D. thesis, Ph. D. thesis, Waikato).

Smith, M. R. (2020). Information theoretic generalized Robinson–Foulds metrics for comparing phylogenetic trees. *Bioinformatics*, 36(20), 5007–5013.

Smith, T., Husbands, P. & O'Shea, M. (2002). Fitness landscapes and evolvability. *Evolutionary computation*, 10(1), 1–34.

Šošić, M. & Šikić, M. (2017). Edlib: a C/C++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9), 1394–1395.

Spruyt, V. (2014). How to draw an error ellipse representing the covariance matrix? Retrieved on 2023-02-14 from: https://www.visiondummy.com/2014/04/draw-error-ellipse-representing-covariance-matrix/.

Staats, K. (2016). *Genetic programming applied to RFI mitigation in radio astronomy*. (Master's thesis, University of Cape Town).

Tackett, W. A. (1994). *Recombination, selection, and the genetic construction of computer programs*. (Ph.D. thesis, University of Southern California Los Angeles).

Taherdoost, H. (2017). Determining sample size; how to calculate survey sample size. *International Journal of Economics and Management Systems*, 2.

Teller, A. & Veloso, M. (1996). PADO: A new learning architecture for object recognition. *Symbolic visual learning*, 81–116.

Tiobe.com. (2000). index | TIOBE - The Software Quality Company. Retrieved on 2022-02-06 from: https://www.tiobe.com/tiobe-index/.

Turney, P. D. (2002). Increasing evolvability considered as a large-scale trend in evolution. *arXiv preprint cs/0212042*.

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11), 1134–1142.

Valiant, L. G. (2009). Evolvability. *Journal of the ACM (JACM)*, 56(1), 1–21.

Vanneschi, L., Clergue, M., Collard, P., Tomassini, M. & Vérel, S. (2004). Fitness clouds and problem hardness in genetic programming. *Genetic and Evolutionary Computation Conference*, pp. 690–701.

Vanneschi, L., Castelli, M. & Silva, S. (2014). A survey of semantic methods in genetic programming. *Genetic Programming and Evolvable Machines*, 15(2), 195–214.

Wagner, A. (2008). Robustness and evolvability: a paradox resolved. *Proceedings of the Royal Society B: Biological Sciences*, 275(1630), 91–100.

Wagner, G. P. & Altenberg, L. (1996). Perspective: complex adaptations and the evolution of evolvability. *Evolution*, 50(3), 967–976.

Zhang, K. & Shasha, D. (1989). Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6), 1245–1262.