University of Johannesburg

Masters Dissertation

A NEAT Inspired GEP Algorithm

Author:

Louis John Hassett

Supervisor:

Prof. Duncan A. Coulter

Co-supervisor:

Daniel Ogwok

A dissertation submitted in fulfillment of the requirements

for the degree of Master in Computer Science

in the

Faculty of Science

Academy of Computer Science and Software Engineering

"*It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is most adaptable to change. In the struggle for survival, the fittest win out at the expense of their rivals because they succeed in adapting themselves best to their environment.*"

Charles Darwin

Acknowledgements

<TODO> - Add acknowledgements here

Contents

List of Figures

List of Tables

1. Introduction

1.1 Background

The theory of evolution by natural selection, first introduced by Charles Darwin, has profoundly influenced our understanding of the life and adaption in the natural world. Darwin's insight that species evolved over generations through the survival and reproduction of individuals with advantageous traits has not only shaped the biological sciences, but has also inspired computational models that emulate these adaptive processes (C. Li, Han, et al. 2024). Over millions of years, evolution has given rise to complex biological systems, among which the human brain stands as one of the most intricate. Composed of billions of neurons, the brain processes information through electrochemical signaling across complex interconnected networks, enabling perception, reasoning, and decision-making (Engelbrecht 2007). These biological mechanisms have served as a blueprint for the development of artificial intelligent systems, particular in the field of evolutionary computation and neural networks.

In computer science, evolutionary algorithms simulate the process of natural selection to solve complex optimisation problems. These algorithms operate on populations of candidate solutions, applying genetic operators such as mutation, crossover, and selection

to iteratively improve the problem's solution. In conjunction to the expanding field of evolutionary computing, artificial neural networks (ANNs) which are inspired by the structure and function of biological neurons have become foundational in machine learning. These networks consist of interconnected nodes that process information in layers, enabling machines to learn from data and perform tasks such as classification, prediction, and control (Russell and Norvig 2016). The intersection of evolutionary algorithms and neural networks has given rise to the field of neuroevolution, which seeks to evolve both the structure and parameters of neural networks using evolutionary principles.

Among the many algorithms developed within the field of evolutionary computing and neuroevolution, Gene Expression Programming (GEP) and the NeuroEvolution of Augmenting Topologies (NEAT) stand out due to their unique and complementary approaches to evolving computational structures. Gene Expression Programming (GEP) is an evolutionary algorithm that evolves computer programs or symbolic expressions. It represents solutions as linear chromosomes, which are then expressed as expression trees through an effective genotype-to-phenotype mapping scheme. This approach allows the evolution of tree-like structures in a more robust and flexible manner than traditional genetic programming techniques (Ferreira 2006). NEAT, in contrast, focuses on evolving the topology and weights of neural networks. It introduces several key innovations, including historical markings (innovation numbers) to track structural changes, speciation to preserve diversity within the population, and incremental growth of network complexity to efficiently explore the search space. These features enable NEAT to evolve increasingly sophisticated neural architectures over time (Stanley and Miikkulainen 2002).

This dissertation introduces a novel hybrid algorithm, GEP-NEAT, which seeks to combine the structural expressiveness of GEP with the adaptive topology evolution of NEAT. The motivation for developing GEP-NEAT arises from specific limitations observed in both NEAT and GEP-based neural network approaches. While NEAT has demonstrated success

in evolving neural network topologies, it suffers from computational inefficiencies, particularly due to the overhead introduced by topological sorting during network evaluation which becomes increasingly problematic as networks grow in complexity. On the other hand, GEP-NN, an approach that applies GEP to evolve neural networks, offers promising alternative by representing neural structures as expression trees, but it remains relatively underexplored in the literature and lacks the methodological maturity and empirical validation seen in other neuroevolutionary techniques.

GEP-NEAT is proposed a response to these challenges, aiming to combine the structural flexibility of GEP with the evolutionary dynamics of NEAT. At the heart of GEP-NEAT is a new representation scheme in which innovation numbers are encoded as sub-tree configurations. This approach allows for a more expressive and hierarchical encoding of neural structures, facilitating the reuse of functional subcomponents and promoting the emergence of modular architectures. By integrating GEP's symbolic representation with NEAT's evolutionary dynamics, GEP-NEAT aims to provide a more powerful and flexible tool for evolving neural networks.

1.2 Structure

This dissertation begins with this introductory chapter, which outlines the research motivation, and key research questions. It also highlights the academic contributions of the work, including publications that have emerged from the research process. Following the introduction, a dedicated chapter is presented on the research methodology, which adopts a design science approach. This chapter details the methodological framework used to guide the development, implementation, and evaluation of the proposed algorithm.

The core of the dissertation presents a comprehensive literature review, divided into three chapters. The first of these explores the foundations of evolutionary algorithms, providing

context for the broader field in which the work is situated. The second chapter focuses on neuroevolution, examining how evolutionary algorithms have been applied to the development of neural networks. The third chapter delves into gene expression programming, detailing its mechanisms, advantages, and relevance to the proposed approach.

After establishing the theoretical foundation, the dissertation introduces the GEP-NEAT algorithm in detail. This chapter covers the theoretical underpinnings of the algorithm, its practical implementation, and the experimental setup used to evaluate its performance. The results of these experiments are then presented and analysed, with a focus on assessing the algorithm's effectiveness, efficiency, and potential advantages over existing methods. The final chapter concludes the dissertation by summarising the key findings, discussing their implications, and outlining directions for future research.

1.3 Research Questions

The development of algorithms that evolve neural network architectures remains a dynamic and evolving area of research. While various approaches have been proposed to automate the design of neural networks through evolutionary computation, several open questions persist regarding the efficiency, expressiveness, and adaptability of these methods. This dissertation is driven by a set of research questions that aim to explore and address specific limitations in existing neuroevolutionary techniques, particularly GEP and NEAT.

The first research question investigates the structural limitations of current gene expression programming when applied to neural networks. Traditional neural networks typically include architectural features such as bias nodes and non-linear activation functions, which are essential for enhancing representational capacity. However, many implementations of gene expression programming for neural networks do not incorporate these features. This leads to the first question:

How can gene expression programming be extended to evolve neural networks that closely resemble traditional architectures, including the incorporation of bias nodes and activation functions?

The second question addresses a known computational bottleneck in topology-based neuroevolutionary algorithms. Specifically, algorithms that evolve network structures often rely on topological sorting to ensure valid signal flow during evaluation. While effective, this process can become increasingly inefficient as networks grow in size and complexity. This raises the question:

Can the computational inefficiencies associated with topological sorting in neural network evaluation be mitigated through alternative representations or evaluation strategies?

A third area of inquiry concerns the role of innovation numbers in NEAT. Traditionally, innovation numbers are used to track structural changes and align genomes during crossover, however, this usage is largely historical and does not contribute directly to the functional behavior of the algorithm. This lead to the question:

Is it possible to redefine innovation numbers in NEAT to represent meaningful and reusable structural components?

Building on this idea, the fourth question explores the practical implications of such a redefinition. If innovation numbers can be used to encode modular structures, it is important to understand how this can be leverages to improve algorithmic performance. Thus, the next question is:

Provided that innovation numbers are redefined as reusable structural components, how can this representation be exploited to improve the performance, modularity, or evolutionary dynamics of the algorithm?

The fifth question considers the broader hypothesis that combining distinct evolutionary strategies may lead to improved outcomes. Specifically, it examines whether integrating symbolic expression-based representations (GEP) with topological representations (NEAT) can result in a more effecting approach to evolving neural networks. This gives rise to the question:

RQ5 *Does the integration of symbolic-based representations (GEP) with topology-evolving strategies (NEAT) result in improved performance, scalability, or expressiveness compared to using either approach in isolation?*

Finally, the sixth question addresses a practical limitation in many symbolic neuroevolutionary systems, that is, the difficulty of evolving neural networks with multiple outputs. Many real-world tasks require networks to produce more than one output simultaneously, yet existing representations often struggle to accommodate this. This leads to the final question:

RQ6 *How can expression trees be adapted to support the evolution of neural networks with multiple outputs, and what are the implications for multi-output learning tasks?*

Together, these research questions form the foundation of this dissertation. They aim to explore the theoretical and practical challenges of evolving neural networks using symbolic and structural representations, and to investigate whether new approaches can overcome the limitations in existing methods.

1.4 Publications Resulting from this Work

A peer-reviewed conference paper derived from this research was published in the proceedings of the **8th International Conference on Information Science and Systems (ICISS 2025)**. As an established forum in its eight iteration, ICISS maintains rigorous academic standards through its double-blind peer review process, where both author and

reviewer identities are concealed to remove bias and ensure impartial evaluation based solely on scholarly merit. The conference brings together leading researchers across ten interdisciplinary tracks spanning artificial intelligence, data science, and information systems.

The accepted paper, which contributes to the Machine Learning and Artificial Intelligence track, presents the algorithm GEP-NEAT with its innovation number novelty, showcasing the ability to solve the XOR and Cart Pole problem effectively. ICISS 2025 facilitated valuable scholarly exchange through keynote presentations by field leaders, technical workshops, and interdisciplinary discussion bridging academic and real-world application. The conference proceedings are to be published into **Communications in Computer and Information Sicence (Electronic ISSN: 1865-0937 & Print ISSN: 1865-0929)** as a proceedings book volume and indexed by EI Compendex, Scopus, INSPEC, SCImago and other databases.

2. Research Methodology

This chapter outlines the methodological framework used to guide the development, implementation, and evaluation of the GEP-NEAT algorithm, a novel hybrid approach that combines Gene Expression Programming (GEP) and NeuroEvolution of Augmenting Topologies (NEAT). To ensure methodological relevance, the research is structured around the Design Science research (DSR) paradigm, which supports the iterative development of innovative artifacts to be used in real-world application. This is complemented by Olivier's insight on DSR, which provides a more refined approach in implementing DSR pragmatically. This chapter is organized in two sections. The first presents the design science methodology and application. The second discusses DSR from Olivier's perspective.

2.1 Design Science Research

Design Science Research (DSR) is a research paradigm centered on the creation of innovative artifacts that contribute meaningfully to the existing body of scientific knowledge within a specific domain. According to Hevner et al. 2004, DSR integrates the principles of relevance, rigor, and iterative design to produce solutions that are both practically useful and theoretically grounded. This methodology is particularly well-suited to algorithmic research, where the objective is to construct novel solutions and evaluate their effectiveness

through cycles of design, implementation, and refinement. In the context of this research, the artifact developed is the GEP-NEAT hybrid algorithm, which aims to address the specific limitations in existing neuroevolutionary approaches by combining the strengths of GEP and NEAT.

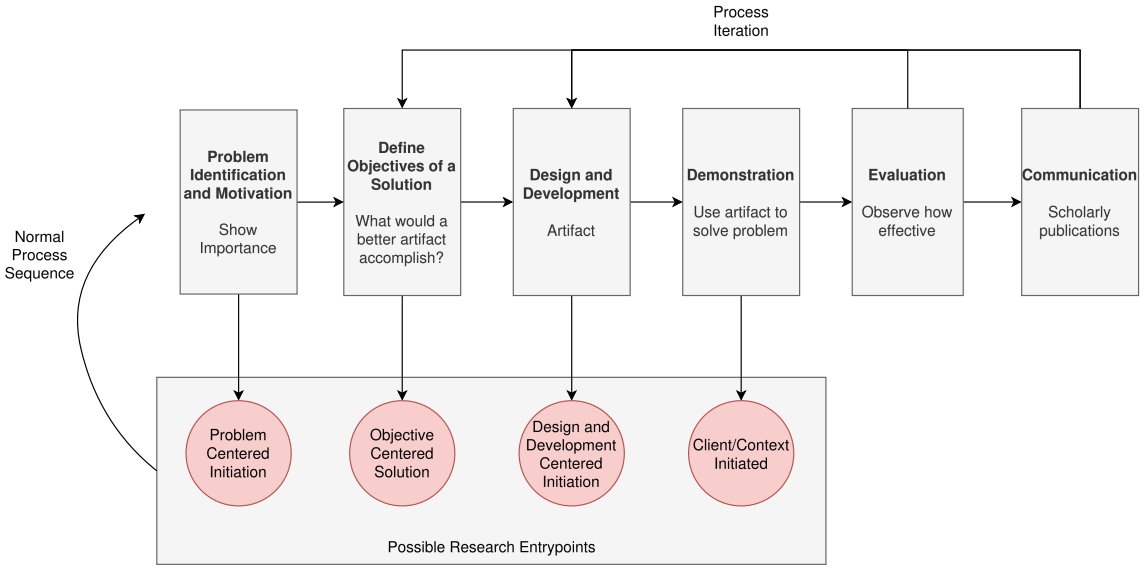


Figure 2.1: Design Science Research Process Diagram (adapted from Hevner et al. 2004)

The use of design science as the chosen research methodology can be examined as 6 process elements as follows with reference to Figure 2.1:

1. **Problem identification and motivation**: The first stage's aim is to formulate the research problem and justify the necessity of a solution. Importantly, this can be broken down into smaller problems in order for the solution to better capture the problem's complexity. Providing the problem and motivation to the reader accomplishes two things, that is, the solution to the problem is motivated to be pursued and secondly, the reader has a much better understanding of what the intention behind the conducted design, development of the prototype and its respective results are.
2. **Objectives of a solution**: The second stage of this methodology is to create a set of objectives based on the problem definition defined above. These objectives can be

either, quantitative, qualitative, or both. Quantitative objectives deal with measurable outcomes that can be expressed numerically whereas qualitative objectives are difficult to quantify and focus on the quality or nature of the solution.

3. **Design and development**: This stage deals with the creation of an artifactual solution along with detailing the artifact's functionality and architecture which will be used to create the actual artifact.
4. **Demonstration**: This stage aims to show the efficacy of the artifact to solve the problem at hand by means of ideologies such as simulation, case studies or experimentation.
5. **Evaluation**: This stage's aim is to measure essentially how well the created artifact supports a solution to the problem which involves the comparison of tried and tested real world results to that of the artifacts. As mentioned in the objective phase, a quantitative and qualitative approach can be taken; the quantitative comparison being based on quantifiable metrics, such as convergence speed, solution quality, etc., and the qualitative comparison being based on solution innovations, adaptability, ease of use, etc.
6. **Communication**: The final stage is to effectively communicate the following:

    - *Problem and it's importance*: This will essentially be the problem statement detailed along with its justification.
    - *Artifact*: An overview of the artifact.
    - *Artifact's utility and novelty*: A background and technical literature that make up the artifact will be provided to the reader.
    - *Rigor of its design*: The way in which the newly formed algorithm/design will be detailed to the reader with explicit explanation in the intricate design choices with mentions to previous results of algorithms the prototype is based on.
    - *Effectiveness*: An analysis will be done on the constructed artifact with comparison to other existing designs using quantitative and qualitative metrics in

order to showcase its use and efficacy to the research field at large.

2.2 Olivier's Insight

While Design Science Research (DSR) provides a high-level framework for the creation and evaluation of innovative artifact, it lacks a fixed set of operational steps or tools. To address this, Olivier 2009, have proposed complementary activities that can be integrated into the DSR process to support knowledge construction, problem exploration, and artifact validation. Each activity contributes to a different phase of the research process, from identifying the problem space to validating the proposed solution.

2.2.1 Literature Review

The research process begins with a comprehensive review of existing literature, which serves as the foundation for identifying gaps in current knowledge and framing the research problem. In line with Olivier's view, the literature view is not a one-time task, but an ongoing process of gathering, filtering, and synthesizing information. It enables the researcher to build a solid theoretical foundation, avoid redundant approaches, and identify opportunities for innovation.

The literature review focuses on three core areas, that is, evolutionary computation, neuroevolution, and gene expression programming. By critically analysing existing work in these domains, the review highlights the limitations of current approaches and motivates the development of a hybrid solution. In selecting sources, priority is given to peer-reviewed journal articles, followed by conference proceedings, textbook, and reputable online resources. While blogs and traditional *'google'* searches are generally treated with caution and used only as a means to support academic findings.

The literature review also serves several strategic purposes, that is, it helps to define the scope of the research problem, identify methodological approaches, avoid unproductive

directions, and uncover new lines of inquiry. In the context of DSR, it provides the initial input for the design cycle by informing the development of the conceptual model and guiding the evaluation criteria for the artifact designed.

2.2.2 Conceptual Modeling

Once the research problem is clearly defined, the next step is to develop a conceptual model that captures the essential components of the proposed solution. In this context, a model serves as a structured representation of the system or process under investigation. It helps to clarify boundaries of the solution space, and provide a blueprint for implementation.

Olivier 2009, emphasises that models can take various forms depending on the research context. They may be descriptive, metaphorical, or formal, and can be developed using principles, scientific notation, or visual languages. This research employs Unified Modeling Language (UML) diagrams to achieve this as it provides a standardised and widely accepted visual language for representing architecture and behaviour (Koc et al. 2021). The diagrams used include:

- **Class and component diagrams**: These diagrams represent the structural composition of the algorithm.
- **Activity diagrams**: These diagrams illustrate the flow of control and decision-making processes.
- **Sequence diagrams**: These diagrams show interactions between components during execution.

UML diagrams are chosen for their clarity and ability to convey complex system interactions in a digestible format, ensuring that the model is comprehensible to both technical and academic audiences. In line with Olivier 2009 perspective, models in computer research can be developed through various means, including formal specification, metaphorical representation, or practical design. In this research, the model is primarily constructed

through design, using system architecture and algorithmic logic to represent the proposed solution. Where appropriate, descriptive metaphors are used to explain abstract concepts, and formal notation is employed to define algorithmic behaviour.

2.2.3 Prototype Development

With the conceptual model in place, the next step is to construct a working prototype that embodies the proposed solution. In DSR, the prototype serves as a tangible instantiation of the model and provides a means of validating the design through various test and benchmarking mechanisms. As noted by Olivier 2009, a prototype is not merely a demonstration tool but a vehicle for inquiry. This allows the researcher to explore the behaviour of the system, identify limitation, and refine the design based on feedback.

Prototypes are also recognized as essential tools for reducing uncertainty and improving design outcomes. Camburn et al. 2017, highlight that prototyping enables real-time feedback, supports iterative development, and facilitates the early identification of design flaws. This allows researchers to test algorithmic behaviour under realistic conditions and to make data-driven decisions about further development. Other research indicates that prototypes provide proof by construction, offering concrete evidence that a theoretical model can be realized in practice. They simply serve as a foundation for further experimentation and analysis, particularly in exploratory research where the goal is to uncover new insights or validate emerging paradigms (Nunamaker Jr, Chen, and Purdin 1990).

2.2.4 Experimental Evaluation

The final activity in the research process is the empirical evaluation of the prototype. Olivier 2009, emphasises that experiments in computing research can serve multiple purposes, that is, they can be used to test hypotheses, explore parameter spaces, or validate theoretical models.

The evaluation of the prototype is approached through both quantitative and qualitative methods, in line with DSR principles of the artifact being rigorously tested to validate the effectiveness in solving the identified problem. Quantitative evaluation in this research involves measuring the algorithm's performance using standard metrics such as accuracy, precision, convergence speed, and computational efficiency (Gregar 2023). Complementing this, qualitative evaluation focuses on the artifacts structural and functional qualities such as modularity, innovation, and scalability. Olivier 2009 notes that qualitative insights are crucial for understanding how well a prototype aligns with its conceptual model and whether it contributes meaningfully to the body of knowledge.

2.3 Conclusion

By integrating literature review, conceptual modeling, prototype development, and experimental into the DSR framework along with Olivier's insight, this research adopts a comprehensive and methodological rigour approach to artifact construction. Each activity contributes to a different phase of the design cycle and supports overarching goal of developing a novel, effective, and theoretically grounded solution to the problem of evolving neural network architectures. The result is a research process that is capable of producing meaningful contributions to both theory and practice.

3. Evolutionary Algorithms

This chapter provides the reader with a comprehensive overview of what evolutionary algorithms entail along with a representative selection of existing work. An introduction to evolutionary algorithms is first given in Section 3.1. Section 3.2 paints a picture of the historical background of evolutionary algorithms. The reader is then provided with fundamentals behind evolutionary algorithms in Section 3.3. Section 3.4 reviews the different evolutionary algorithms that exist and their differences, and finally Section 3.5 provides an overview of current advancements in the field of evolutionary computation.

3.1 Introduction

Evolutionary Algorithms (EAs) belong to a large group of optimisation methods that take inspiration from Darwinian biological evolution (Eiben and Smith 2015). These methods replicate processes like natural selection, mutation, and reproduction. They simulate the adaptive and competitive dynamics observed in nature, where a set of candidate solutions, called the population, is iteratively refined over multiple generations (C. Li, Han, et al. 2024).

In order to mimic the lifecycle of organisms, selecting a representation of a candidate

solution that is amenable to algorithmic manipulation is a prerequisite. With the proper representation of a candidate solution in place, EAs follow the general framework of mimicking the lifecycle of organisms. To explain simply, this framework works firstly by generating an arbitrary number of handful candidate solutions that make up the initial population. Each candidate solution will have an attempt at solving the problem at hand after which this attempt will be scored and recorded relative to a particular fitness. This scoring scheme is achieved by a *fitness function* and generally has the primary role of determining how strong an individual is at solving the problem at hand. Each candidate solution will then go through a selection process whereby the best solution will be chosen, and the others disregarded, ensuring that only the best-performing candidates pass their desirable traits onto offspring and consequently forthcoming generations. Additionally, a mechanism is established to combine candidate solutions, facilitating the exchange of traits to enhance the diversity and quality of the population over successive iterations. Surviving candidates will produce offspring by applying genetic operators such as mutation (which introduces variability by altering the gene of offspring) and crossover (which combines the traits from two or more parents). After population initialization, this repetitive process of selection, mutation, and reproduction enhances the likelihood that the successive generations migrate towards a global optima while mitigating the chances of falling into a local optima (Mirjalili 2018).

EAs have been applied to a wide range of different fields ranging from engineering, to artificial intelligence. EAs are typically used in optimisation problems where there is a single or multiple objectives due to their ability to explore the solution space in a flexible manner. EAs have application in a vast number of fields where for example EA algorithms were used to optimise solar panel layouts in real-time by aircrafts in the engineering realm. Another example was EAs involvement to predict flood routing in natural channels using gene expression programming (GEP) in the environmental field (Slowik and Kwasnicka

2020).

3.2 Historical Background

EAs go back to the early computational era of the 1950s and 60s where researchers began to draw inspiration from Darwinian principles of natural selection (Petrowski and Ben Hamida 2016). These researchers managed to mathematically model the competitive and adaptive processes found in nature in order to solve and optimise complex and multi-dimensional problems. The idea was simple - just as organisms in nature evolve to adapt according their environment, organisms in the mathematical realm (candidate solutions) can *"evolve"* by competing and improving over time, akin to natural generations. This approach in solving optimisation problems became compelling to be used in problems that were ill-defined or had far too many variables that lead to exhaustive searching and suboptimal solutions.

EAs began with contributions from many important people. British mathematical and geneticist Alex Fraser took early steps to model genetic ideas using computers in the 1950s (Links and BookDust 2002). Hans-Paul Schwefel and Ingo Rechcenberg were inspired by his work and soon developed evolution strategies in Germany during the 1960s primarily for application in engineering (Petrowski and Ben Hamida 2016). Around that time, an American Scientist, John Holland, formulated the concept of genetic algorithms (GAs) which established a formal framework to simulate biological evolution in a mathematical sense. Holland's work in the 1970s revolutionized EAs extending their application in many areas (Petrowski and Ben Hamida 2016). After the publication of the book, *Genetic Algorithms in Search, Optimisation and Machine Learning* by E. Goldberg in 1989, interest in the field became much more widespread (Petrowski and Ben Hamida 2016).

As the field of computers evolved in terms of technological sophistication and breadth

of application, so did the capabilities and appeal of evolutionary algorithms. Due to advancements in computer science in the late 20th century, researchers began to explore more complex and sophisticated EA models. Also PhD student of John Holland, John Koza, a computer science researcher, formulated genetic programming which opened new avenues for using evolution to optimise nonlinear structures such as symbolic regression functions and treelike structures (Koza 1994). Differential Evolution (DE) was introduced in the 1990s which provided a mechanism in solving problems suited for continuous parameter optimisation (Das and Suganthan 2010).

3.3 Core Concepts in Evolutionary Algorithms

EAs are built from several core concepts that mimic Darwinian principles, that is, the process of selection, variation, and inheritance. These core elements form the backbone of the way in which EAs operate, creating a system whereby candidate solutions are evolved over iterative generations to meet or exceed a given objective. Understanding these core concepts are crucial in understanding the mechanics and flexibility of EAs, as each component dictates how the algorithm explores and optimises the search space.

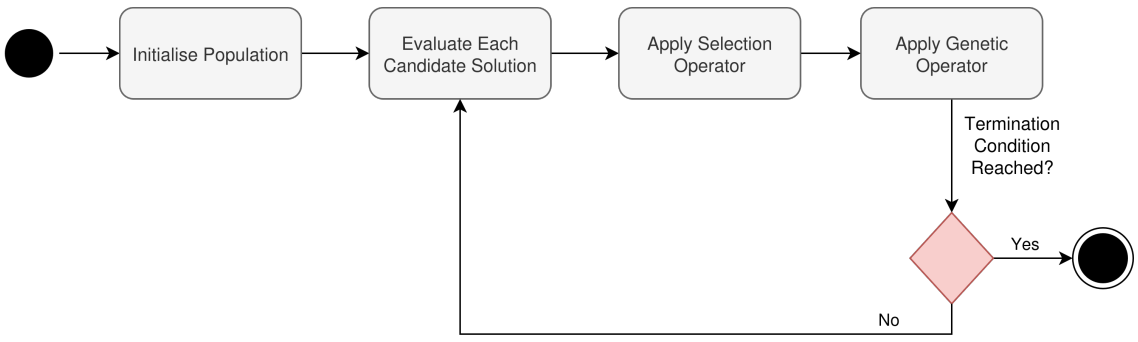


Figure 3.1: Evolutionary Algorithm Framework (adapted from Wirsansky 2024)

3.3.1 Representation

The term chromosome is often used to refer to representations of candidate solutions within the population which are a critical aspect of the evolutionary algorithms, an attempt to model genes and chromosomes seen in nature. The representation dictates how individuals are encoded, manipulated, and evaluated within the evolutionary process. A representation that is well-defined within the context of the optimisation problem enhances the algorithm's ability to explore the search space efficiently while maintaining diversity and feasibility among solutions.

The genotype-phenotype mapping translates encoded information into a meaningful solution. There are two forms in an evolutionary algorithm, namely, the **genotype** which is the encoded representation of a solution and is typically manipulated directly by genetic operators such as crossover and mutation, and the **phenotype** which is the decoded, functional representation of the solution which is what gets evaluated against the fitness function or objective (Kramer 2017). For example, consider an optimisation problem where a 4-bit binary number that decodes the highest integer value needs to be found. The genotype would be a chromosome representing a 4-bit string (e.g., 1010). The phenotype would be the decoded integer value (e.g., $1010 \rightarrow 10$ in decimal). It is not necessarily for every optimisation problem to have an explicit genotype-phenotype mapping, in particular, when the genotype is the solution itself (Kramer 2017).

There are many ways in which chromosomes can be represented as solutions to the problem at hand. Representations are divided into two categories, namely, direct representations and indirect representations. Direct representations are those in which the genotype-phenotype mapping is one-to-one, allowing genetic operators to be applied directly without transformation. In contrast, indirect representations require an additional decoding step to map genotypes to phenotypes before genetic operators can be applied (Rothlauf 2009). Genetic

algorithms have originally been proposed to be binary coded to mimic the genetic material of natural organisms, however, different representations are used based contextually on the optimisation problem at hand (Yu and M. Gen 2010). Common representations include those seen in Figure 3.2. Binary representations are commonly used for problems with discrete or combinatorial solutions, such as feature selection or Boolean optimisation, where each bit can encode the presence/absence of a feature or a binary decision. Integer or real-valued representations are suited for numerical optimisation tasks, such as parameter tuning in machine learning or engineering design, where variables require fine-grained precision. Tree-based representations model hierarchical structures, making them ideal for symbolic regression, program synthesis, or parsing tasks, where solutions naturally map to decision trees or recursive expressions.

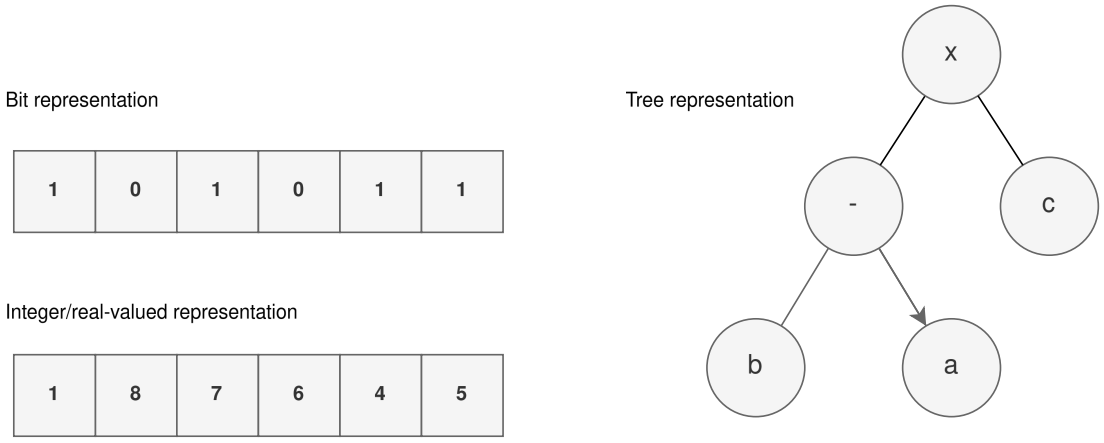


Figure 3.2: Common Gene Representation (adapted from C. Li, (Shoufe), and Zeng 2024)

3.3.2 Population and Fitness

The starting point in any evolutionary algorithm is the initialization of a population as shown in Figure 3.1 which represents a group of candidate solutions that could potentially solve the problem at hand. Unlike traditional optimisation techniques which mainly focus on a single-solution pathway, EAs work within the entire solution space by means of parallelism (Kramer 2017). To put it simply, each generation evaluates candidate solutions

as potential answers, with the algorithm progressively identifying better ones through selection and variation. This parallel search ensures that local optima are avoided, an issue which high-dimensional optimisation typically face. By ensuring that the initial population generated is diversified, there is an increased chance of finding a global or near global optimal solution. There are many methods for initializing a population, with randomized approaches being the most common, specifically those drawn from a uniform probability distribution, to ensure a broad coverage of the search space (Maaranen, Miettinen, and Penttinen 2007). Other mechanisms for initializing the population include heuristic and intelligent based approaches when existing solutions are known which could potentially ensure a better coverage spread of the search space (Maaranen, Miettinen, and Penttinen 2007). When taking advantage of using existing solutions to explore the search space, repetitive computations are mitigated yielding in an increased efficiency. Likewise, non-random initial populations can directly gear the algorithm to search a particular region of the solution space from the beginning which again, with the aid of known solutions, can allow the algorithm to converge quicker towards an optimal solution. Other mechanisms include deterministic based approaches in which candidate solutions are generated in a more controlled and evenly distributed manner leading to a better coverage of the solution space (Tharwat and Schenck 2021).

In genetic algorithms, diversity refers to the variety of genetic material (e.g., different solution features or values) within a population. The population's diversity closely relates to how well the candidate solutions explore the search space. If individuals are too similar, they lack diversity leading to premature convergence causing stagnation around a suboptimal solution. Conversely, if the individuals are too diverse from one another, convergence can be delayed meaning that more generations are needed in order to hone in on an optimal solution (Kramer 2017). A fine balance is therefore needed between exploration and exploitation. A highly diverse initial population is crucial to ensure thorough exploration of

the search space early in the optimisation process. As generations progress, the algorithm selectively converges toward higher-quality solutions, naturally reducing diversity while refining the best candidates. There are well studied mechanisms which can control this balance such as mutation rate adjustment, niche-reserving mechanisms, or even hybrid approaches. Using these mechanisms ensures that both diversity and convergence can be capitalized.

With reference to Figure 3.1, after the population is initialized, each candidate solution will be vetted by means of a fitness function which is a mathematical measure that scores how well the candidate solution meets the objective (Kramer 2017). This score reflects the quality of the candidate solution in regard to the optimisation goal and is used to guide the selection process in subsequent generations. For example, given an optimisation problem such that x is an integer to maximize the parabolic equation $f(x) = -x^2 + 25$, the fitness function would be taking a candidate solution, and feeding it as input into this equation. Evidently, the maximum fitness to this function would be 5. Fitness functions are defined in different ways depending on the application at hand, for example, in engineering design problems, fitness may represent efficiency, structural stability, or cost-effectiveness, whereas in artificial intelligence, fitness could measure a predictive accuracy. The way in which a fitness function is defined shapes how effectively the EA converges towards an optimal solution, especially in multi-objective problems (MOPs) where the fitness needs to account for multiple goals (Eiben and Smith 2015).

MOPs are mathematical problems that involve optimizing multiple conflicting objectives at the same time. The fitness function in this regard is defined by the algorithm's performance in relation to these conflicting objectives as opposed to one (Kramer 2017). A common approach in formulating the fitness function is by aggregating all the objectives in a singular fashion by means of a weighted sum (Kramer 2017). Another approach in defining the fitness function appropriately is by changing the structure of algorithm by means of

co-evolution - instead of evolving a single population, the problem is broken down into smaller problems which in turn have their own population which is evolved (A. Zhou et al. 2011). In multi-objective optimisation problems which are common in economics and engineering, the fitness function may use methods like Pareto front optimisation which balances competing objectives without forcing single trade-off solutions. Pareto optimality describes a solution that cannot be improved in any objective without degrading another. In these kind of scenarios, EAs evolve a population of solutions that collectively represent a spectrum of optimal trade-offs, giving the decision-maker flexibility in choosing the most suitable solution based on contextual priorities (Rachmawati and Srinivasan 2009). EAs are therefore robust and versatile in solving both single and multi-objective problems which can often be riddled with narrowly defined and complex optimisation challenges.

3.3.3 Genetic Operators

In EAs, genetic operators - crossover and mutation - serves as crucial mechanisms by means of variation and adaption to guide the exploration of the solution space while refining existing solutions. Variation in evolution can be divided into two types based on their arity, namely, unary (mutation) and n-ary (recombination) both of which produce offspring (Eiben and Smith 2015). These operators mimic the evolutionary processes seen in nature among organisms, where gene recombination and mutation introduce new traits that contribute to a population's overall fitness and adaptability (Wirsansky 2024).

3.3.3.1 Crossover

Crossover is an evolutionary process based on sexual reproduction whereby genetic material from two parents are combined to produce offspring that inherit characteristics from both (Kramer 2017). Crossover plays a crucial part in diversifying the algorithm's search path by combining beneficial traits and thereby ideally producing offspring with improved fitness. There are different ways in which crossover can occur which provide varying levels of control over the inheritance process. For simplicity, consider linear gene representations

for the following crossover mechanisms.

Single-point Crossover selects a single point along the chromosome and exchanges genetic material on either side as shown in Figure 3.3. This simple approach works well when minor structural adjustments results in significant improvement (C. Li, (Shoufe), and Zeng 2024).

Multi-point crossover introduces more than one cutoff point, allowing for more complex combinations of genetic material from each parent thereby creating a more diverse offspring as shown in Figure 3.4 (Zainuddin, Abd Samad, and Tunggal 2020). Multi-point crossover is typically used in scenarios that are non-linear.

Uniform crossover selects each gene randomly from one of the corresponding genes of the parent as shown in Figure 3.5 (Zainuddin, Abd Samad, and Tunggal 2020). This technique produces highly diversified offspring and is suitable when seeking greater diversity in the population thereby leveraging exploration.

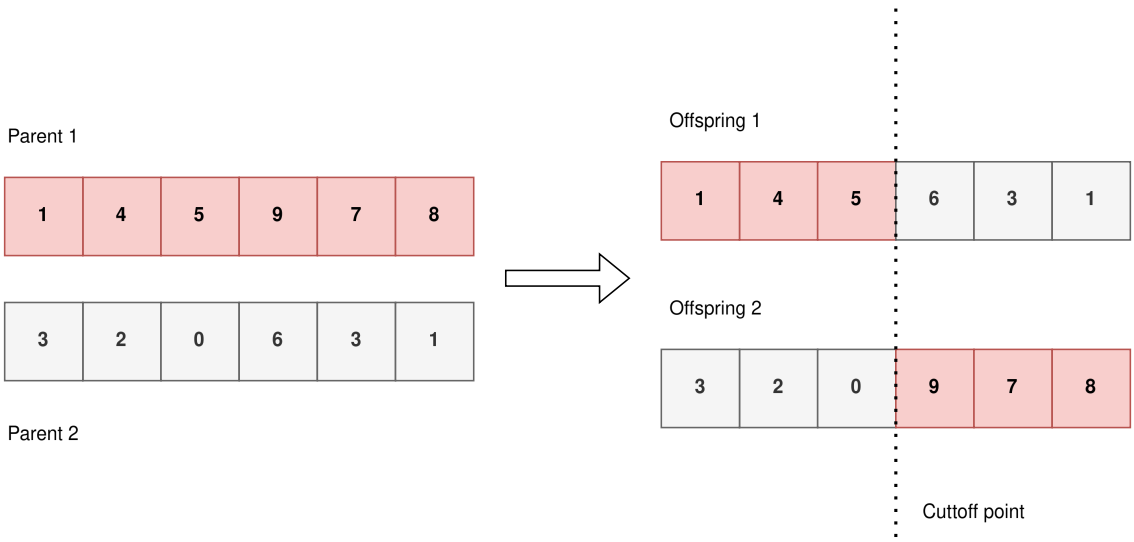


Figure 3.3: Single-point crossover

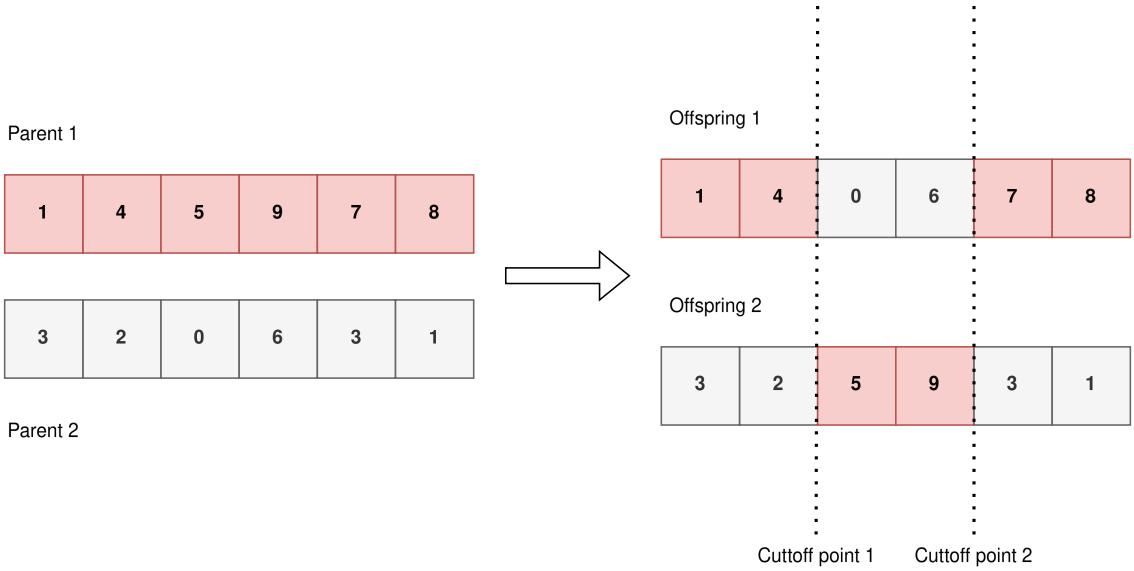


Figure 3.4: Multi-point crossover

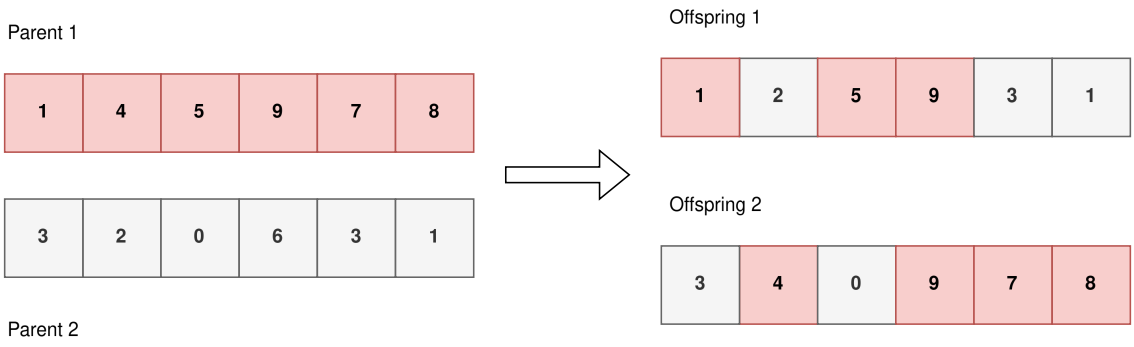


Figure 3.5: Uniform crossover

In scenarios where problem requirements are specific, crossover strategies can be tailored to meet particular constraints. For example, order crossover (OX1) is useful for optimisation problems such as the traveling salesman problem whereby maintaining certain order relationships between genes (cities) is critical (Larranaga et al. 1999). Such customized crossover methods ensure that the offspring retain problem-specific constraints while still inheriting the most desirable traits from parent solutions. Beyond sexual recombination, horizontal gene transfer (HGT), inspired by bacterial gene exchange, can also diversify

populations by transferring genetic material directly between solutions without parent-offspring relationships (Rafajłowicz 2018).

3.3.3.2 Mutation

Mutation in contrast to crossover introduces randomness in the algorithm by altering one or more genes in an individual's chromosome in order to create new traits and variation (Mirjalili 2018). Mutation serves as the primary source of diversity within the population and important when solutions begin to become too familiar, a condition known as *genetic drift* (Ahn 2006). When individuals become too similar, this results in stagnation around suboptimal solutions causing convergence towards a local maxima. Mutation can occur in many ways, such as:

Bit-flip mutation (aka bitwise mutation) is applied to chromosomes represented in binary whereby each gene has a probability that its bit is inversed (changing from 0 to 1 or vica-versa) as shown in Figure 3.6 (C. Li, (Shoufe), and Zeng 2024).

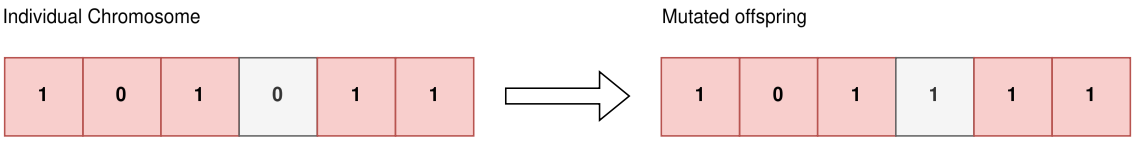


Figure 3.6: Bit-flip mutation

Gaussian mutation is applied to real valued chromosomes whereby a random gene is selected and a Gaussian distribution applied by means of the function below where x_i^i is the offspring, σ is a fixed parameter for all variables, $[b_i, a_i]$ represents the range of random gene, erf^{-1} represents the inverse of erf which is the Gauss error function, defined by $erf(y) = \frac{2}{\sqrt{\pi}} \int_0^y e^{-t^2} dt$ (Bell 2022, Hinterding 1995). Gaussian mutation allows for controlled but continuous alterations.

$$x_i^{'} = \sqrt{2}\sigma(b_i - a_i) erf^{-1}(u^{'} i) \tag{3.1}$$

Inversion mutation selects a subset of genes and inverts the position as shown in Figure 3.7. This mutation scheme is useful in combinatorial optimisation problems where the specific arrangement of elements matters such as the travelling salesman problem (C. Li, (Shoufe), and Zeng 2024).

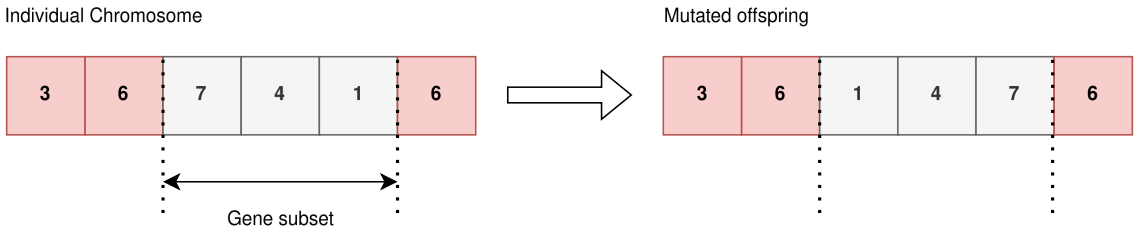


Figure 3.7: Inversion mutation

The parameters that govern genetic operators do not have to be static. There is more complex EAs, these parameters can be adjusted dynamically as the algorithm progresses. Adaptive mutation rates for example can be useful in maintaining diversity during later stages of the search when converging towards an optima is prioritized. Likewise, crossover rates could increase when diversity is low in order to ensure that favourable genetic material continues to recombine as the algorithm refines its search (Meyer-Nieberg and Beyer 2007). In hybrid approaches, self-adaptive genetic operators are used whereby the parameters of mutation and crossover evolve and form part of the chromosome, allowing the EA to essentially learn optimal rates over time based on the fitness at hand (Meyer-Nieberg and Beyer 2007).

Hybrid approaches in genetic operators have become popularized due to its powerful nature. For example, memetic algorithms combine the typical genetic operators described above with local search techniques which allow candidate solutions to further refine the search space by means of exploiting problem-specific knowledge or heuristics (Neri and Cotta 2012). This combination is useful in complex landscapes where small local improvements

can lead to an increased fitness score after major variations have been introduced through crossover and mutation in individuals. Hybrid genetic operators can therefore enhance the EAs ability to explore local optima thoroughly while still retaining a broad search capacity.

There is a fine balance between crossover and mutation in the efficacy of EAs. Mutation ensures that the EA explores new possibilities and avoids premature convergence while crossover allows the search space's best regions to be explored by recombining favourable traits of candidate solutions. These operators form a dynamic balance between exploration and exploitation, equipping EAs with the necessary flexibility to solve complex and non-linear optimisation landscapes.

3.3.4 Selection Mechanisms

Selection occurs in an evolutionary process in two instances - parent selection and survivor selection. Parent selection has its primary purpose in distinguishing individuals based on their quality in order to allow better individuals to become parents of the next generation (Wirsansky 2024). Survivor selection, similar to parent selection, aims to distinguish individuals apart based on quality. Parent selection occurs before variation operators are applied while survivor selection occurs after offspring are produced and evaluated against the fitness function (Eiben and Smith 2015). This selection mechanism mimics the natural selection seen in nature where animals with desirable traits have increased likelihood of producing better offspring - akin to this, candidate solutions with better fitness scores are more likely to produce better quality offspring to solve the objective at hand. An effective selection mechanism will ensure that there is a balance between promoting high-quality solutions whilst maintaining population diversity ideally driving the EA towards a global optimum. Below are common selection mechanisms that are typically used within evolutionary algorithms:

Fitness-proportionate selection which is one of the simplest yet most renowned selection methods. In this selection approach, each candidate solution's probability of being selected is proportional to its fitness score relative to the total fitness of the population (Shukla, Pandey, and Mehrotra 2015). Mathematically, the probability that each individual is selected is given by the formula below where f_x is the fitness of individual x (Shukla, Pandey, and Mehrotra 2015).

$$p_x = \frac{f_x}{\sum_{i=1}^{n} f_i} \tag{3.2}$$

This selection mechanism is computationally efficient ($O(n)$ complexity) and ensures that fitter individuals are more likely to survive and reproduce while still giving less fit individuals the chance to contribute and preserve genetic diversity thereby preventing stagnation. The most common implementation of this approach is *roulette-wheel selection*, which assigns selection probabilities analogously to slots on a roulette wheel such that higher fitness solutions occupy larger wheel segments and thus have greater likelihood of being selected. Although this process is simple, this mechanism may suffer from genetic drift if highly fit individuals dominate which results in a loss of diversity in subsequent generations. This approach raises considerations about the *takeover time*, that is, the number of generations required for the best candidate solution's genes to dominate the population under selection pressure alone. Short takeover times indicate strong selection pressure that may lead to premature convergence, while longer takeover times encourage diversity within the population at the cost of slower optimisation.

Tournament selection offers a more controlled selection mechanism which is done by firstly by selecting a subset of individuals based on random sampling. Each subset known as tournaments will have individuals compete and the strongest survive. The size of the subset or tournament impacts the selection pressure - larger tournaments increases the chances that the fittest individuals will dominate while smaller tournaments result in a more diversified selection pool of individuals (Shukla, Pandey, and Mehrotra 2015).

Tournament selection is computationally efficient ($O(n)$ complexity) and provides a simple way to adjust selection pressure making it a popular choice in EAs (Shukla, Pandey, and Mehrotra 2015).

Rank-based selection addresses the limitations of the fitness-proportionate selection scheme by ranking individuals based on their fitness rather than their raw fitness score (Mitchell 1998). This would mean that individuals with exceptionally low fitness scores would have an increased chance to be selected, lowering the chances of genetic drift. This selection method has complexity $O(nlogn)$ and maintains selection pressure without skewing the algorithm aggressively towards highly fit individuals which will allow a broad exploration of the search space (Mitchell 1998).

Elitism is a technique that is generally used with other selection mechanisms to ensure that the most fit solutions are preserved across generations. In this mechanism, a hall of fame is produced by taking the top n individuals which are then carried to the next generation without modification guaranteeing that the best solutions are always kept (Mitchell 1998). This method essentially prevents the best solutions from being lost due to changes made by genetic operators thereby accelerating convergence. Although elitism speeds up the rate at which the algorithm will converge, it can lower diversity leading into premature convergence on a suboptimal solution (S. Malik and Wadhwa 2014).

Niche and speciation techniques are employed to encourage diversity in specific traits by creating subpopulations, or niches within the larger population, however, may be computationally more taxing (complexity $O(n^2)$) as opposed to the aforementioned selection techniques (Bäck, Kok, and Rozenberg 2012). Speciation can be categorised into four types based on the geographical result that manifested the speciation (Cilliers 2022):

- **Allopatric speciation** occurs when the population is divided and no interactions occur between the resulting subpopulations. Over many generations, the groups start

to diverge until they can be distinguished apart to form their own new species.

- **Peripatric speciation** occurs when a portion of the population migrates away, forming their own subpopulation. Unlike allopatric speciation, individuals from different subpopulations can still interact with each other. Over successive generations, these groups diverge to become their own separate species.
- **Parapetric speciation** is the similar to peripatric speciation with the difference that some interaction can still occur between separated subpopulations. Although the interaction between groups are limited, over time the groups can still diverge to create separate species.
- **Sympathetic speciation** occurs when a group forms within the population still occupying the same geographical space. Over many generations, the group is still likely to diverge to form part of a separate species, however, with the added benefit of interacting with the same pool of individuals during the selection and reproduction stages.

To ensure effective speciation and to thoroughly explore the search space, maintaining a diverse population is essential. The diversity can b categorised at three distinct levels, namely, the gene level, chromosome level, and the population level. At the gene level, variation is evaluated at each genetic locus across the entire population. At the chromosome level, the focus shifts to the diversity within individual chromosomes throughout the population. Finally, at the population level, diversity is considered by examining the distribution of each bit across all chromosomes in the population (Diaz-Gomez and Hougen 2007). To support this diversity and prevent premature convergence, various diversity management techniques are applied. These techniques are designed to encourage exploration and maintain a healthy spread of genetic material throughout the population, and they typically fall into several categories. These can be categorised as follows with an example mechanism (Segura et al. 2016):

- **Selection-based**: These methods regulate the selection pressure during the parent

selection phase. Diversity is typically preserved by adjusting parameters such as takeover time and selection intensity. While these techniques can effectively maintain diversity in the short term they often struggle to sustain it over successive generations due to the homogenising effects of crossover operations (Blickle and Thiele 1996).

- **Population-based**: These approaches modify the structure of the population itself, often by introducing subpopulations or altering mating restrictions. A common example is the use of island models which is discussed below.
- **Crossover/mutation-based**: In this category, diversity is maintained by imposing constraints on genetic operators such as crossover and mutation. For instance, the Cataclysmic mutation scheme (CHC) avoids mating individuals that are too genetically similar and periodically reintroduces variation through reinitialisation mechanisms (Segura et al. 2016).
- **Fitness-based**: These methods modify the way fitness is calculated to encourage the survival of diverse individuals. A well-known example is fitness sharing which is discussed below.
- **Replacement-based**: These strategies focus on how individuals are replaced in the population. A common method is elitism, which as previously discussed, ensures that the best-performing individuals are retained across generations. While generally used to preserve high-quality solutions, elitism can also contribute to diversity by preventing the premature loss of valuable genetic material.

Research has yielded many techniques in order to stimulate niching within evolutionary algorithms:

- **Fitness Sharing** which operates on the principle that fitness is a shared resource among individuals in an attempt to reduce redundancy within a population or search space (Bäck, Kok, and Rozenberg 2012).
- **Clearing** which is a mechanism whereby the top individuals per niche can take primary advantage of the resource of that particular niche (Bäck, Kok, and Rozenberg

2012).

- **The Islands Model** which is a mechanism inspired directly from organic evolution whereby the population is divided into multiple subpopulations and evolved independently. During evolution, gene information is exchanged between subpopulations in an attempt to encourage trait migration in order to exploit a subpopulations search space (Bäck, Kok, and Rozenberg 2012). A recent development in island model approaches is the CLISDE model, which applies a nested island genetic algorithm to evolve decision trees for classification tasks. By evolving separate populations of decision trees in parallel and periodically exchanging information, the model enhances diversity and improves classification performance (Cullinan, Coulter, and Appel 2023).

3.4 Types of Evolutionary Algorithms

EAs cover a large set of a large set of optimisation approaches that share the fundamental principles of population-based search, genetic variation, and selection, mimicking evolutionary processes. These algorithms have morphed into distinct types each aimed to be used in a particular class of problems. Well known evolutionary algorithms include genetic algorithms, evolution strategies, genetic programming and differential evolution. While all follow the core principles of evolutionary processes, each type is uniquely structured to address specific challenges in optimisation, making EAs a versatile tool for solving for a many complex problems.

3.4.1 Genetic Algorithms

Genetic Algorithms (GAs) are one of the most well known types of EAs, initially formalized by John Holland in the 1970s (Mitchell 1998). In traditional GAs, candidate solutions are represented as chromosomes with binary gene values (bit strings), however, these have now been extended to include real values representations as well. These chromosomes

undergo genetic operations like crossover and mutation to produce new offspring for subsequent generations. The basic genetic algorithm follows the basic framework shown in Algorithm 1.

Algorithm 1 Basic Evolutionary Algorithm that Genetic Algorithms follow (Bäck, Kok, and Rozenberg 2012)

1: Initialise population of individuals
2: Evaluate all individuals by fitness function
3: **repeat**
4: Select individuals (parents) for reproduction
5: Vary selected individuals in randomized manner to obtain new individuals (offspring)
6: Evaluate offspring by fitness function
7: Select individuals for survival from offspring and possibly parents according to fitness
8: **until** stopping criterion fulfilled

Traditional optimisation algorithms typically struggle with high-dimensional or noisy problem spaces, however, EAs find their solution for a given problem by dynamically adapting their search pathways. Solving for the solution is similarly tautological making EAs favorable when confronted with complex, or poorly-defined problems where the normal traditional algorithms struggle to explore the solution space as intricately (C. Li, (Shoufe), and Zeng 2024).

GAs have been proven to be effective in both discrete and combinatorial optimisation problems where the solution space is vast and discontinuous (Eiben and Smith 2015). Due to the flexibility by which the chromosomes can be represented, GAs are widely used in engineering design, machine learning, and scheduling problems where they can efficiently

explore and exploit the search space at hand which can be of varying complexity (Bäck, Kok, and Rozenberg 2012).

One of the distinguishing features that GAs have to offer are their ability to handle discrete variables and constraints naturally. This is simply done by defining the individual chromosomes as a sequence of discrete numbers as shown in Figure 3.8. This makes GAs highly suitable in applications such as resource allocation, routing, and other combinatorial problems that require search across many permutations (Bäck, Kok, and Rozenberg 2012). The use of crossover and mutation gives GAs the flexibility to explore the search space thoroughly while selection mechanisms help guide the algorithm towards good quality solutions. As mentioned, GAs are characterised as a kind of local search which are also hybridized with other techniques (such as memetic algorithms), to refine solutions and enhance the performance of the search. While genetic algorithms are used as a general purpose approach, they still run a risk of failure. When genetic algorithms are used in optimisation problems that lack detail and are represented poorly, the algorithms' performance is directly affected (Bäck, Kok, and Rozenberg 2012).

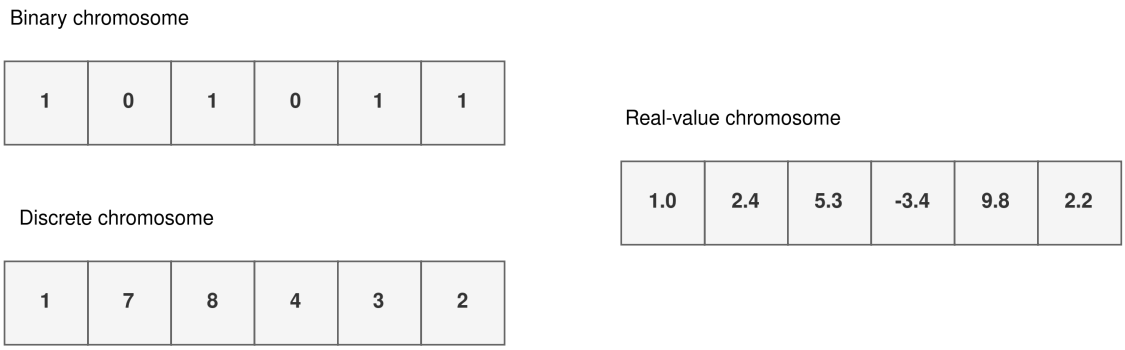


Figure 3.8: Different encoding schemes

In situations where real-valued parameters need to be encoded into binary chromosomes, a common approach is to use a representation similar to the *IEEE 754* floating-point format.

(Satman and Akadal 2020). This involves encoding each real umber by separating it into three components, namely, the **sign bit** which indicates whether the number is positive or negative, the **mantissa** which represents the precision bits of the number, and the **exponent** which scales the value. This method allows real values to be efficiently represented and manipulated within a binary representation of the chromosome.

3.4.2 Evolution Strategies

Evolution Strategies (ES) were developed by Hans-Paul Schwefel and Ingo Rechenberg in the 1960s, as a means for solving continuous optimisation problems in the engineering field (C. Li, (Shoufe), and Zeng 2024). Unlike traditional GAs which typically use binary or discrete representations, ES uses real-valued vectors to represent solutions making them well-suited for optimizing real-valued parameters. Evolution strategies emphasize mutation as the primary genetic operator, applying random changes to solution vectors to generate offspring while crossover plays a less pivotal role (Bäck, Kok, and Rozenberg 2012). Evolutionary Strategies operates as shown in Algorithm 2.

Algorithm 2 Evolutionary Strategy Algorithm (Bäck, Kok, and Rozenberg 2012)

1: Initialise population of individuals
2: Evaluate all individuals by fitness function
3: **repeat**
4: Select parents for reproduction uniformly at random
5: Vary selected individuals in randomized manner to obtain offspring
6: Evaluate offspring by fitness function
7: Select individuals from best ranks for survival
8: **until** stopping criterion fulfilled

ES differs from GAs in another fundamental way which is its self-adaptive approach to mutation. This self-adaption is achieved by evolving the strategy parameters alongside

the chromosome, allowing ES to tune its search almost autonomously (Bäck, Kok, and Rozenberg 2012). ES has been proven effective in solving engineering design problems where precise adjustments to real-valued parameters are essential. Its ability to control balance between exploration and exploitation, especially in continuous and high dimensional search spaces makes ES a powerful tool for optimisation problems requiring fine-grain control (Eiben and Smith 2015).

3.4.3 Genetic Programming

Genetic Programming (GP) introduced by John Koza in the early 1990s, represents a unique branch of EAs that evolve tree-like structures as opposed to linear chromosomes of fixed string (Koza 1994). These trees represent executable programs, typically made up of mathematical functions, logical operations, or other computational expressions. GPs are suitable for optimisation problems such as symbolic regression, machine learning, or automated problem-solving. The main idea behind GPs are simple: evolve these program structures iteratively over generations through core evolutionary algorithm operations such as crossover, mutation, reproduction. A key distinction of GP is how fitness is evaluated which is by executing each candidate program on representative test cases with its performance directly determining its fitness value.

GP's primary strengths are its flexibility and interoperability as it allows users to analyse and modify the resulting programs directly. GPs can be applied to generate models from regression problems, develop classification systems, or even design control systems in robotics (O'Neill 2009). For example, in symbolic regression, GPs evolve mathematical expressions that fit observed data, often discovering intricate relationships that traditional regression techniques struggle to formulate (Koza 1994). Additionally, GP has been applied in fields like finance, image analysis, and automated design, where evolving decision rules or functional expressions can produce valuable insights and practical solutions (O'Neill 2009).

GP follows the same framework that was shown in Figure 3.1. A population is firstly initialized which in GP sense is by means of producing randomized trees. In order to prevent trees that lack too much diversity and are far too complex, the *ramped half-and-half* method is used whereby a fraction of the population is initialized with trees having a maximum depth of 1, another fraction having a depth of 2, and so on (Bäck, Kok, and Rozenberg 2012). Once a population is generated, the algorithm will go through the trivial evolutionary process of selection, mutation, crossover and reproduction. Genetic operators however vary vastly in the way they are applied compared to traditional GAs. For example, when crossover occurs, this is implemented by means of subtree replacement as shown in Figure 3.9 where an entire subtree is exchanged with another to produce offspring.

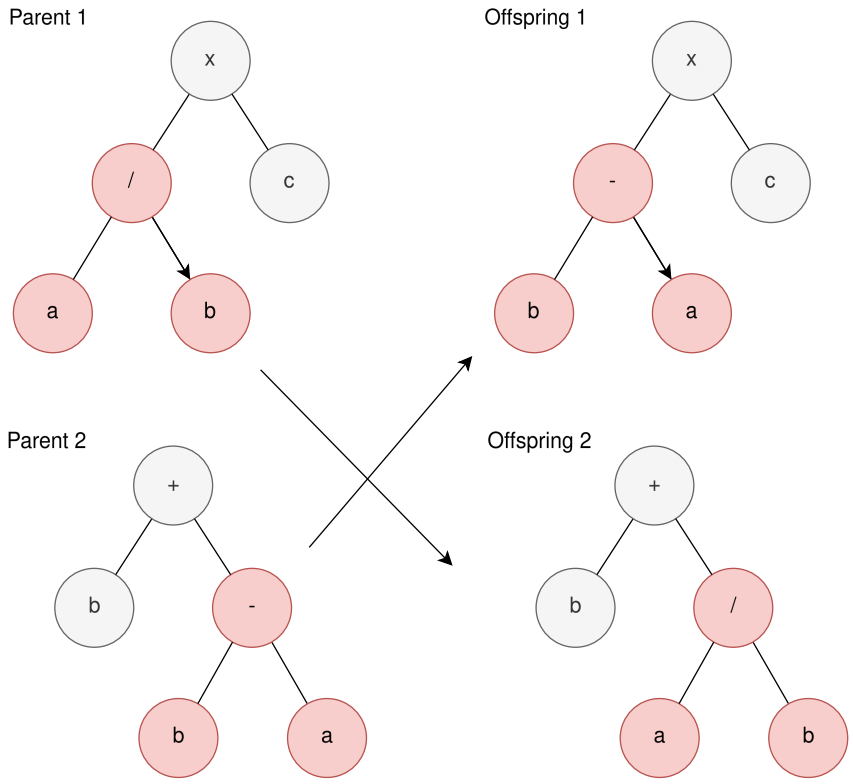


Figure 3.9: Genetic Programming Crossover (adapted from Eiben and Smith 2015)

Subtree mutation, on the other hand, replaces a randomly selected subtree with a newly generated one, introducing new expressions and broadening the search space as shown in

Figure 3.10.

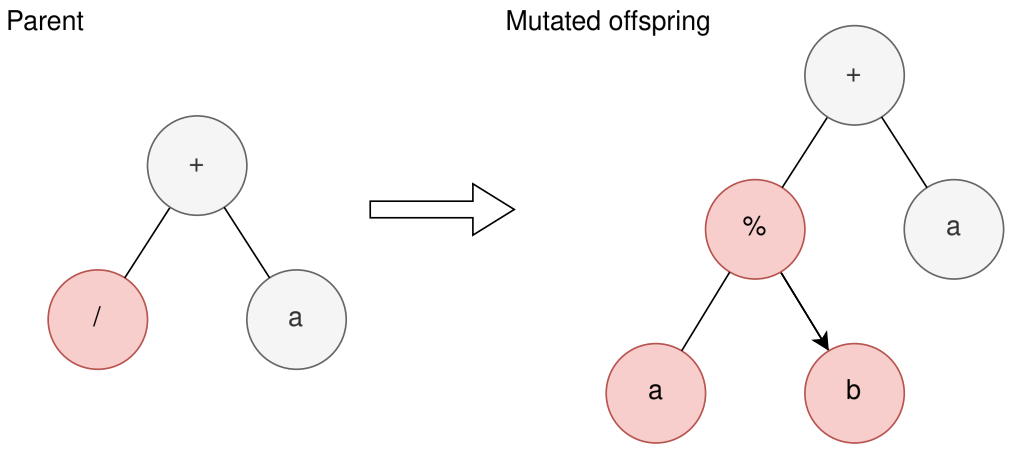


Figure 3.10: Genetic Programming Mutation (adapted from Eiben and Smith 2015)

Crossover and mutation allow GP to navigate a wide variety of functional forms and solution pathways making it well suited for problems that benefit from structural flexibility such as symbolic regression or predictive models (Bäck, Kok, and Rozenberg 2012).

One of the challenges that GP faces is in managing computational resources as evolving complex trees can be computationally intensive (Koza 1994). This means that large populations and lengthy expressions may require significant processing power making GPs slower than better suited evolutionary methods. There are many ways to counteract this, one of which is to limit the tree depth, however, come at the disadvantage of less diverse individuals (Koza 1994). More advance approaches include techniques such lexicographic parsimony pressure where selection is modified to prefer smaller trees only in situations where fitness remains the same (Luke and Panait 2002). Another challenge is the additional effort required in ensuring that the trees resulting from genetic operation are valid programs (Koza 1994).

3.4.4 Differential Evolution

Differential Evolution (DE) offers a distinctive approach to continuous optimisation problems, particularly suited for high-dimensional search spaces with real-valued parameters (Storn and Price 1995). Unlike GAs which rely heavily on crossover and mutation of binary-coded chromosomes, DEs strength lies in its vector-based mutation scheme. The way in which it works is as follows:

1. For each parent in the population, a trial vector $u_i(t)$ is generated. This trial vector is generated using the following formula:

$$u_i(t) = x_{i_1} + \beta(x_{i_2}(t) - x_{i_3}(t)) \tag{3.3}$$

   where $x_{i_2}(t)$ and $x_{i_3}(t)$ are randomly selected individuals such that $i \neq i_1 \neq i_2 \neq i_3$. i_2 and i_3 are chosen based on uniform sampling, and $\beta \in (1, \infty)$ which is a scaling factor for controlling amplification (Storn and Price 1995)

2. After obtaining the mutated offspring which is the trial vector, crossover occurs which is implemented as follows:

$$x_{ij}^{'}(t) = \begin{cases} u_{ij}(t) \ if \ j \in J \\ x_{ij}(t) \ otherwise \end{cases} \tag{3.4}$$

   where $x_{ij}^{'}$ is gene j of chromosome i, and J is the set of chosen perturbation points, which is essentially the set of trial vectors (Storn and Price 1995).

3. Once the offspring is produced from the crossover operator, the evolutionary process continues over iterative generations until termination conditions are met.

DE's main advantage lies in its simplicity, requiring only a few parameters like the mutation scaling factor and crossover rate which makes it easy to implement (Storn and Price 1995). In addition, DE's efficiency with limited parameters make it feasible for use in high-stakes, real-time applications like control system design and power system optimisation, where

quick convergence on reliable solutions is essential for maintaining system stability and performance (Das and Suganthan 2010).

Despite DEs strengths, there are challenges in balancing exploration and exploitation. For problems with highly complex landscapes, DE may converge slowly if its mutation factor is not carefully tuned (Das and Suganthan 2010). Along with this, DEs reliance on population-based differences for mutation can lead to stagnation in situations where diversity is rather low. To address this issue, adaptive differential evolution (ADE) has been developed where mutation and crossover rates adjust dynamically to encourage exploration in later stages of the generation run (Das and Suganthan 2010). Other variations of this algorithm, like multi-objective differential evolution (MODE) extends DE to handle multiple objectives simultaneously which produces a set of pareto-optimal solutions offering the user a range of trade-off options (Xue, Sanderson, and Graves 2003).

Both GP and DE illustrate its versatility within the field of evolutionary algorithms, each tailored to different use cases. While GP excels at evolving flexible and interpretable models for symbolic and algorithmic problems, DEs efficiency in continuous space make it ideal for high-dimensional, real-valued optimisation tasks. Likewise, in situations where computer programs need to be created to approximate an optimisation task, this is where GP shines. These algorithms showcase the adaptability of evolutionary principles to address a wide range of complex problems.

EAs are particularly valuable for problems where evaluating solution quality is straightforward, but generating optimal solutions remains computationally difficult. For such problems, EAs provide an effective approach provided that:

1. the candidate solution can be represented in a modifiable format (e.g., trees, graphs, or vectors)
2. the search space permits incremental improvements through variation operators

This contrasts with traditional methods that may require explicit gradients or problem-specific heuristics (C. Li, (Shoufe), and Zeng 2024).

3.5 Current Research

The field of evolutionary algorithms (EAs) continues to advance rapidly, driven by increasing computational capabilities, the emergence of new application domains, and the demand for more adaptive and scalable optimisation tools. Recent research trends highlight the development of self-adaptive algorithms, the integration of EAs with machine learning, and their application to large-scale and complex systems.

One promising direction is the development of self-adaptive algorithms that can dynamically adjust their evolutionary parameters during the optimisation process. These algorithms eliminate the need for manual tuning, which is often time-consuming and problem-specific. Self-adaptive mechanisms are particularly valuable in solving problems with dynamic environments, applications of which has seen to be used in scheduling problems and vehicle routing during traffic congestion (Dulebenets et al. 2018, Sabar et al. 2019).

Closely related to this is the challenge of maintaining genetic diversity within evolving populations. As previously discussed, initialising a population with sufficient genetic diversity is crucial for effectively exploring the search space and voiding premature convergence to local optima. Over the past decade, several strategies have been proposed to enhance diversity management in evolutionary algorithms. One such method is the Order Distance Vector (ODV) approach, which facilitates population seeding based on an ODV matrix which is a structure that quantifies the diversity differences among elements within a set (Victer Paul et al. 2014). Another recent technique involves the use of k-means clustering during initialisation, where genes are grouped into k clusters. Local and global

optima are then identified within each cluster to ensure a more representative and diverse initial population is generated (Deng, Yang Liu, and D. Zhou 2015).

Significant progress has also been made in improving the performance of crossover operators. Notable among these are the Cut on Worst Gene Crossover (COWGC) and Cut on Worst Left+Right Gene Crossover (COWLRGC) methods. These operators aim to generate offspring by preserving the most beneficial genetic material from both parents, thereby increasing the likelihood of producing fitter individuals (Alkafaween 2018). Another innovative approach is the collision crossover, which conceptualises parent chromosomes as objects moving toward each other. Offspring are generated at the point of collision, simulating an interaction that enhances genetic mixing and exploration of the solution space (Alhijawi and Awajan 2023).

Another interesting area of research is EA's involvement in machine learning and artificial intelligence. Evolutionary algorithms are increasingly used to optimise hyperparameters, architectures, and training processes for machine learning models, such as deep neural networks (Young et al. 2015). Beyond neural network, EAs are finding applications in automating machine learning workflows, including feature selection, model selection, and algorithm design (Nikitin et al. 2022).

With the rise of big data, evolutionary algorithms are being adapted to handle large scale datasets. Traditional EAs face computational challenges when applied to large-scale problems, but advancements in distributed computing, cloud-based frameworks, and GPU acceleration have enabled EAs deployment on massive datasets. EAs are being tailored to solve large-scale optimisation problem in fields such as energy distribution and supply chain, an example of which was its application to schedule the power of distributed systems (Guzek et al. 2014), and another to optimise an integrated system of bioenergy production supply chains (Ayoub et al. 2009).

Finally, the incorporation of fuzzy logic into evolutionary algorithms has opened new avenues for handling uncertainty and imprecision. Fuzzy logic controllers (FLCs) mimic human reasoning by using linguistic rules to guide decision-making. Lin and Gen introduced a fuzzy-based auto-tuning mechanism that dynamically adjusts EA parameters by monitoring changes in the average fitness of parents and offspring population. This adaptive strategy, grounded in linguistic control rules and approximate reasoning, represents a significant step toward more intelligent and responsive evolutionary systems (Mitsuo Gen and Lin 2023).

4. Neuroevolution

This chapter provides the reader with insight into neuroevolution with primary focus on neuroevolution of augmenting topologies (NEAT), a derivative of a Topology and Weight Evolving Artificial Neural Network (TWEANN). An introduction to neuroevolution is given in Section 4.1. The historical background is explored in Section 4.2. Neural architecture search is discussed in Section 4.4. Section 4.3 provides the reader with the basic concepts of neural networks which is followed by the main topic, NEAT, in Section 4.5.

4.1 Introduction

Neuroevolution refers to the optimization of artificial neural networks (ANNs) using evolutionary algorithms (Risi and Togelius 2015). Unlike traditional gradient-based methods which typically rely on backpropagation and gradient descent to train network weights, neuroevolution employs evolutionary processes inspired by Darwinian evolution to evolve both the architecture and parameters of the neural the network. This approach enables capabilities that traditional gradient-based approaches lack, that is, learning neural network topology configurations such as activation functions, hyperparameters, and architectures (Stanley, Clune, et al. 2019). A more detailed comparison with conventional neural network

training techniques in provided in Section 4.3.

At its core, neuroevolution treats the design and training of neural networks as an optimization problem in a high-dimensional space. With reference to the evolutionary principles discussed in Chapter 3, candidate solutions are represented as encoded genetic structures, such as chromosomes or vectors, and are subjected to evolutionary processes like mutation, crossover, and selection. Over successive generations, these processes explore the artificial network's configuration, refining the architectures and weight setting in an attempt to optimize the given task at hand (Risi and Togelius 2015). Neuroevolution offers unique advantages over traditional neural network training methods. For instance, neural networks are traditionally designed manually through trial and error which can become frustrating and cumbersome - neuroevolution does this autonomously.

4.2 Historical Background

The development of neuroevolution can be traced back to the early exploration of both neural networks and evolutionary algorithms as separate disciplines in the mid-20th century (Stanley and Miikkulainen 2002). Evolutionary algorithms, inspired by Darwinian principles of natural selection, were initially applied to optimization problems in the engineering and biology realm. Around the same time, neural networks emerged as a powerful framework to mimic and model cognitive processes (Stanley and Miikkulainen 2002). By the late 1980s and early 1990s, researchers began to combine these fields, laying the groundwork for what is now known as neuroevolution (Stanley and Miikkulainen 2002).

By the 1990s, studies demonstrated that the complexity of an artificial network's topology affects the speed of evaluating the network and its learning rate, and that evolving structures saves times wasted by humans trying to decide what the topology of a neural network should look like for a given problem (Stanley and Miikkulainen 2002). Recurrent Neural

Networks (RNNs), a type of artificial neural network designed to recognize sequential patterns by retaining a "memory" of previous inputs through hidden feedback loops, were further advanced by Peter J. Angeline, Gregory M. Saunders, and Jordan Pollack. Their work in evolving RNNs demonstrated neuroevolution's potential for uncovering complex temporal structures (Angeline, Saunders, and Pollack 1994). Kenneth o. Stanley's introduction of Neuroevolution of Augmenting Topologies (NEAT) algorithm in 2002, an extension of TWEANN, marked a significant milestone, showcasing the ability to evolve both a neural network's topology and weights simultaneously (Stanley and Miikkulainen 2002). NEAT's innovative approach to preserve diversity through speciation and historical marker tracking addressed key challenges that neuroevolution faced such as premature convergence and the difficulty in evolving candidate solutions that represented complex structures (Stanley and Miikkulainen 2002).

The 2000s and 2010s witnessed the rise of more sophisticated algorithms, such as HyperNEAT, which extended NEAT to large-scale, high-dimensions problems by evolving networks that exploit geometric regularities (Stanley, D'Ambrosio, and Gauci 2009). In parallel to this, evolutionary strategies (ES) gained traction in optimizating continuous network parameters, particularly in reinforcement learning. By integrating neuroevolution with reinforcement learning, researchers unlocked new capabilities for solving optimization problems that require agents to adapt dynamically in changing environments (Igel 2003).

Today, neuroevolution continues to grow, benefiting from advancements in distributed systems (and thereby computational power), and hybrid approaches that combine evolutionary algorithms with gradient-based methods (Asseman, Antoine, and Ozcan 2021). The historical progression of neuroevolution reflects a shift from parameter optimization to the co-evolution of architectures and topology parameters, establishing its role as a unique and powerful technique in the artificial intelligence realm.

4.3 Neural Networks in a Nutshell

Neural networks form the backbone of modern artificial intelligence (AI), offering a framework to learn complex relationships from a set of data. They are inspired by the biologicinamely, the cell body, dendrites, and axon as shown in Figure 4.1.

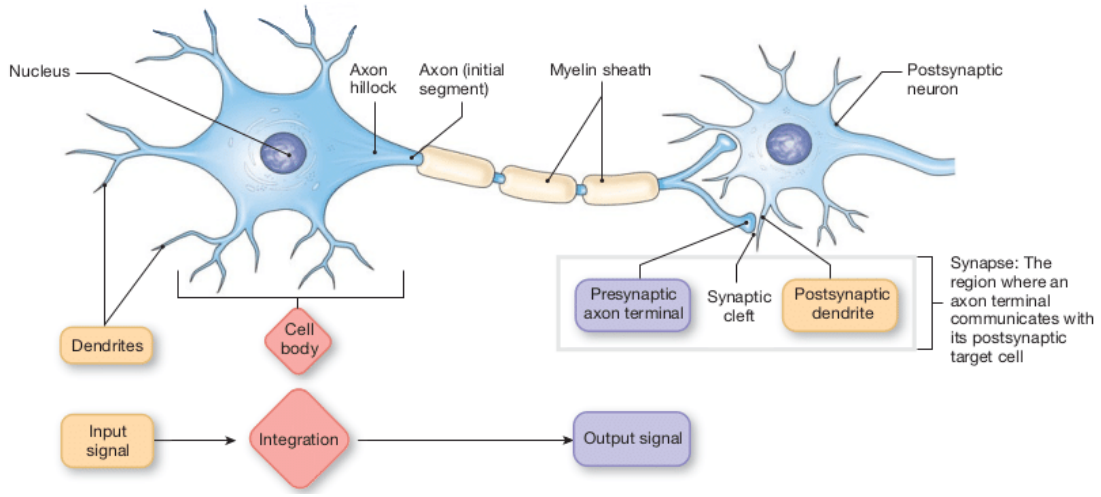


Figure 4.1: Simplified illustration of neuron anatomy (Cook et al. 2021)

These neurons communicate through electrical signals by mean of impulses (aka signal) in the cell wall. The impulses are mediated through the junctions called *synapses* which are located on branches of the cell known as *dendrites* (Gurney 2018). The neurons in turn are interconnected with many other neurons which are sending and receiving a multitude of incoming signals simultaneously. The signals are summed together in some way, but unlike in artificial models, where this is often a simple amplitude-based threshold, biological neurons rely heavily on the frequency of incoming spikes to determine firing. If the combined input (weighted by timing and firing rate) exceeds the neuron's threshold, it generates a voltage response (and thereby *"fire"*) (Gurney 2018). The synapse is categorized as either being an excitory synapse which encourages the subsequent neuron from firing, or it can be an inhibitory synapse which discourages the subsequent neuron

from firing (Bishop 1994). Whether the *synapse* is excitory or inhibitory is dictated by its strength (or weight). Provided that the neuron is *"fired"*, a signal is sent to other neurons via the *axon* and serve as input to the subsequent neuron (Gurney 2018).

4.3.0.1 The Artificial Neuron

At their core, neural networks are computational systems that consist of interconnected nodes that mimic the biological neuron seen in Figure 4.1. Akin to the biological neural system, an artificial neuron is depicted in Figure 4.2.

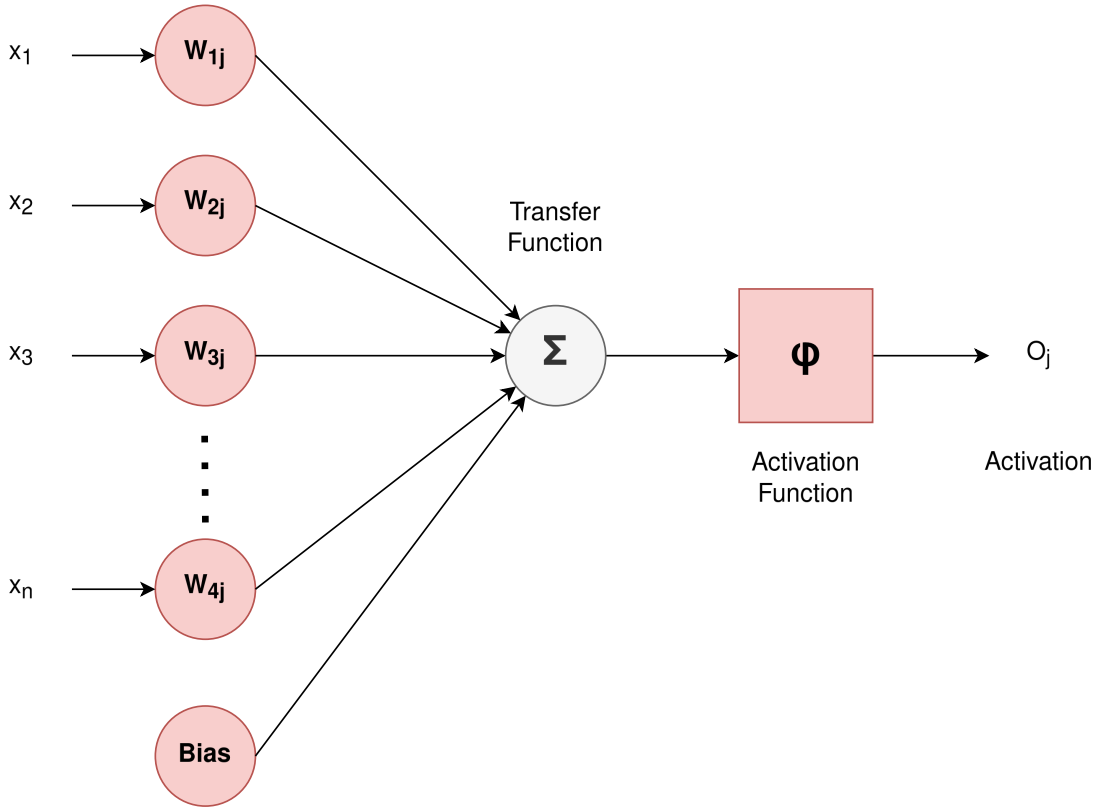


Figure 4.2: Perceptron (Artificial Neuron) model (adapted from Russell and Norvig 2016)

The weighted sum (transfer function) depicted in Figure 4.2 can be represented mathematically as follows (adapted from Suzuki 2011):

$$O_{transfer\_function} = \sum_{i=1}^{n} w_{ij} x_i + b \tag{4.1}$$

where:

- x_i is the input signal
- w_{ij} is the weight of the neuron
- b is the bias
- $O_{transfer\_function}$ is weighted sum/transfer function result

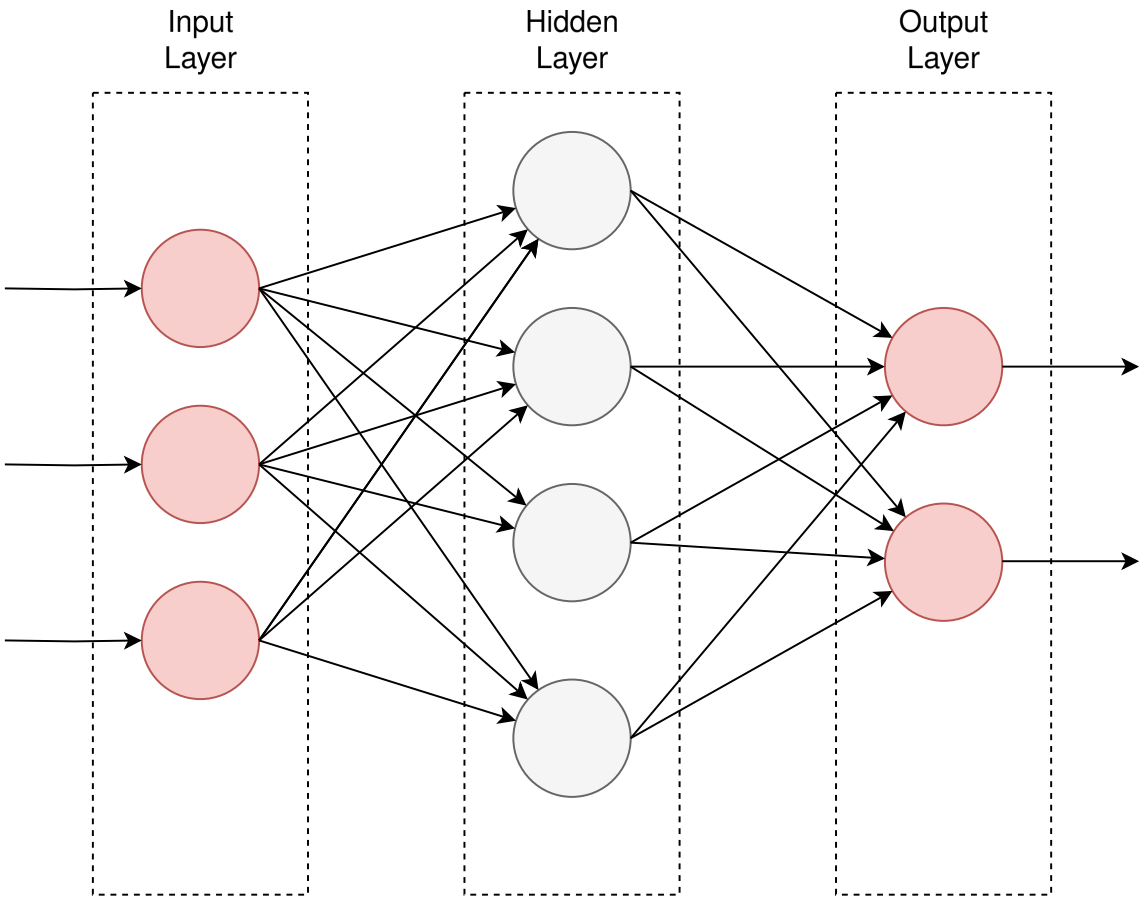


Figure 4.3: Simple Artificial Neural Network (adapted from Nielsen 2015)

The perceptron computes a weighted sum of its inputs plus a bias term. This bias acts like an adjustable threshold, shifting the activation left or right to ensure the neuron can fire even when input signals are weak or zero. Without the bias, the decision boundary (e.g., for classification) would always pass through the origin, severely limiting the network's flexibility (S. Wang, T. Zhou, and Bilmes 2019). Once the weighted sum plus the bias is

calculated, an activation function is applied as follows (adapted from Suzuki 2011):

$$O_j = \phi(O_{transfer\_function}) \tag{4.2}$$

where ϕ is the activation function and $O_{transfer\_function}$ is the result from equation 4.1. These neurons, similiar to the biological neuron are interconnected with other neurons that receive and forward signals. Depicted in Figure 4.3 is the general structure of a layered artificial neural network.

A neural network is made up of 3 distinct layers, namely, the input layer, hidden layer, and output layer as shown in Figure 4.3 (Nielsen 2015). To explain by example, suppose we want to create a neural network to solve the simple XOR problem, that is, given certain bit value combinations, it returns a particular output according to the following truth table:

Table 4.1: The XOR problem truth table

Input A	**Input B**	**Output**
0	0	0
0	1	1
1	0	1
1	1	0

According to the truth table, there are two inputs which corresponds to two input neurons, and likewise, there is one output corresponding to one output neuron. Figure 4.4 illustrates what this network would look like. While mapping of the input and output layers are often simple, the hidden layers can be designed of arbitrary size and can be multilayered (Nielsen 2015). The XOR problem highlights a fundamental limitation of single-layer perceptrons. While a single perceptron can model linearly separable functions like AND and OR, it cannot represent XOR, a non-linear problem where inputs must be classified based on

their parity. This happens because no straight line can separate XOR's truth table outputs, however, this limitation is overcome by multi-layer perceptron (MLP) networks, where hidden layers enable non-linear decision boundaries through successive transformations. Researchers have developed numerous heuristics in order to design what the hidden layers should look like - these heuristics for example can help determine what the trade-off is for the number of hidden layers given the amount of training needed for the network while some researchers even argue that three hidden layers are enough to achieve exponential convergence and mitigate the problem of dimensionality (Nielsen 2015, Shen, Yang, and Zhang 2021).

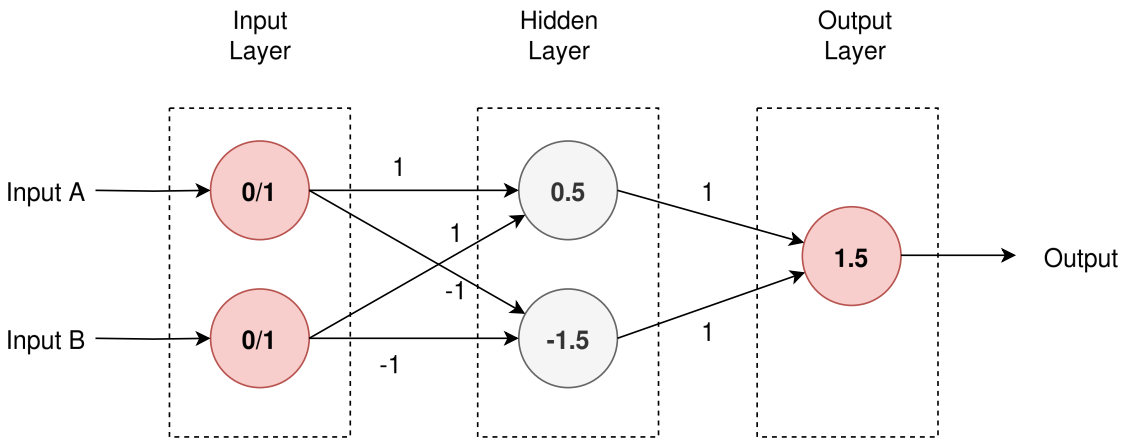


Figure 4.4: XOR Problem Neural Network

Once the neural network structure has been designed, weights are associate with each connection between neurons. To test what output results from a corresponding set of inputs, these inputs are fed into the input layer. The weighted sum of each neuron is calculated in the hidden layer and subjected to an activation function. This is then fed forward as input into the output layer which in response, again, calculates the weighted sum and applied to an activation function producing a final output. This flow of information from the input to the output characterizes the network as a feedforward neural network which essentially means there are no loops within the network (Nielsen 2015). Neural networks that have

loops form part of other distinct models, one of which is recurrent neural networks (RNNs).

4.3.1 The Activation Function

The activation function controls whether the neuron *"fires"*, using amplitude-based thresholds to make this decision. Unlike biological neurons that rely on firing frequency, this artificial approach provides a way to introduce non-linearity into the network. If an activation function is not used in a neural network, the output signal will always be a linear function that is a polynomial of degree one (Sagar Sharma, Simone Sharma, and Athaiya 2017). There are three classes of activation to choose from depending on the problem at hand, namely, a step function, linear function, and non-linear function (Suzuki 2011). To give an overview of the most popular activation functions:

Binary Step Function (*Step Function Category*): This is the simplest activation function that either outputs a 0 or 1 depending on whether some threshold is met. It is mathematically represented as shown in Equation 4.3 and visually illustrated in Figure 4.5 (Sagar Sharma, Simone Sharma, and Athaiya 2017). Binary step functions are typically used when creating binary classifiers as the output maps directly to two options (Sagar Sharma, Simone Sharma, and Athaiya 2017).

$$f(x) = \begin{cases} 1 \; if \; x \geq 0 \\ 0 \; if \; x < 0 \end{cases} \tag{4.3}$$

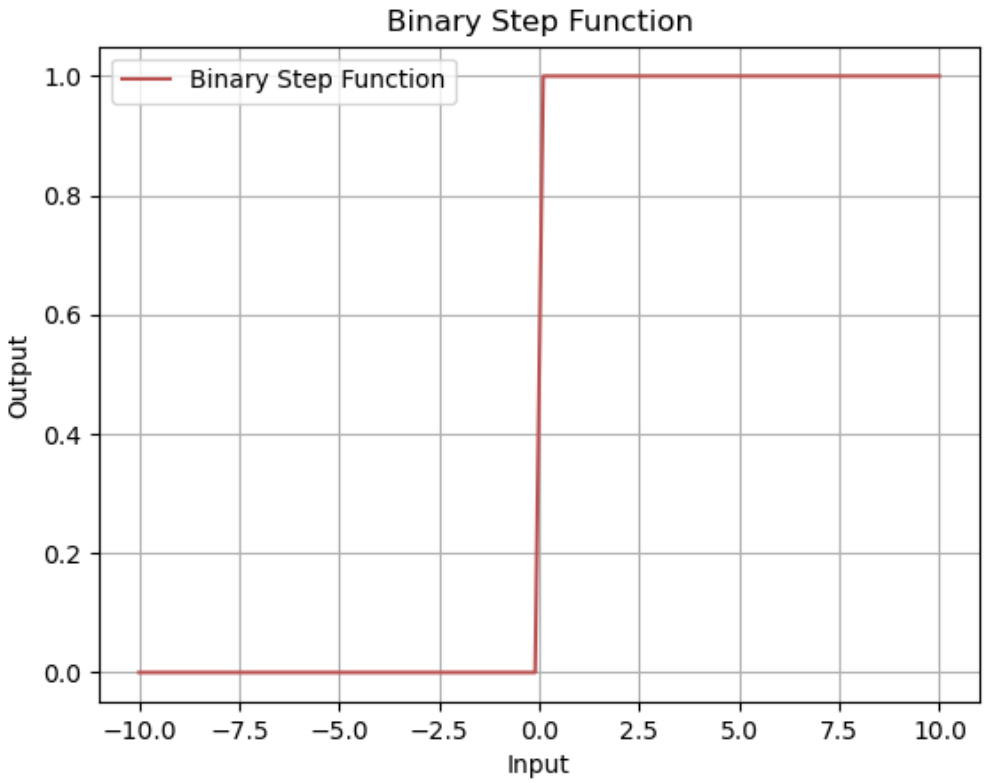


Figure 4.5: Binary Step Activation Function Plot

Linear Activation Function (*Linear Function Category*): This function represents a relationship that is directly proportional to the input. It is mathematically represented as shown in Equation 4.4 and visually illustrated in Figure 4.6 (Sagar Sharma, Simone Sharma, and Athaiya 2017). Linear activation functions are best suited for regression problems where linear relationships are the only concern (Sagar Sharma, Simone Sharma, and Athaiya 2017).

$$f(x) = ax \tag{4.4}$$

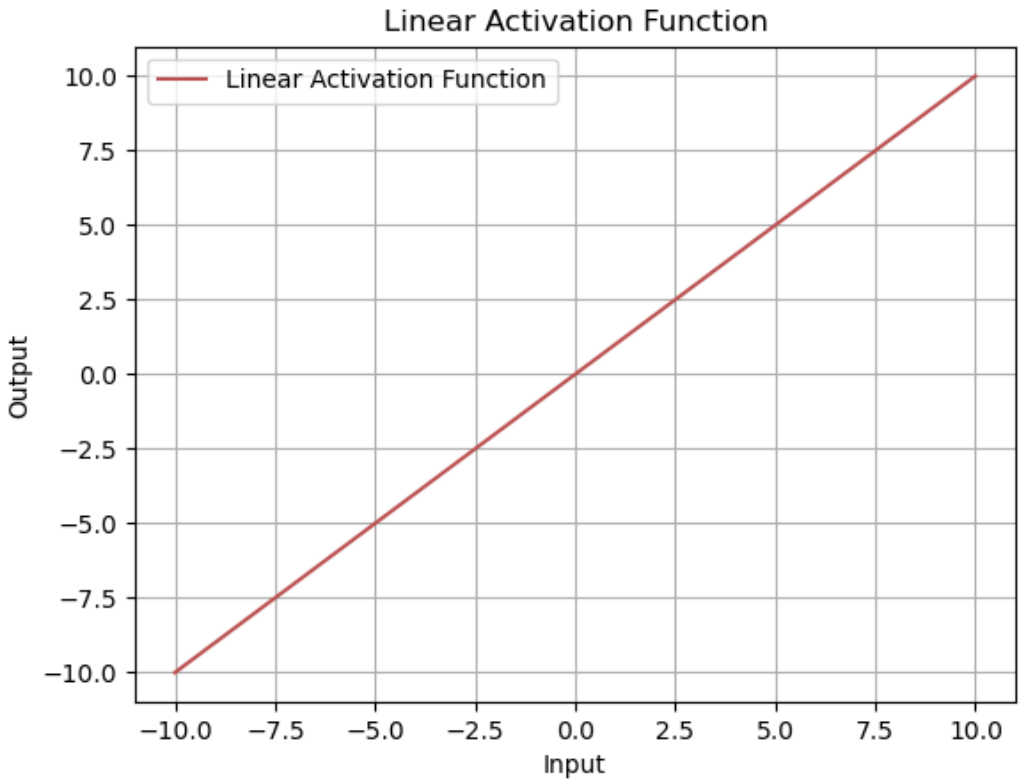


Figure 4.6: Linear Activation Function Plot

Sigmoid Activation Function (*Non-linear Function Category*): This function is the most widely used activation function due to its ability to be continuously differentiable and non-symmetric about zero meaning that the sign of all output values of neurons will be the same (Sagar Sharma, Simone Sharma, and Athaiya 2017). It is mathematically represented as shown in Equation 4.5 and visually illustrated in Figure 4.7 (Sagar Sharma, Simone Sharma, and Athaiya 2017). This activation function brings non-linearity into the neural network having applicability a wide range of tasks such as classification problems, function approximation, etc - essentially problems that involve complex relationships to be learned (Sagar Sharma, Simone Sharma, and Athaiya 2017).

$$f(x) = \frac{1}{e^{-x}} \tag{4.5}$$

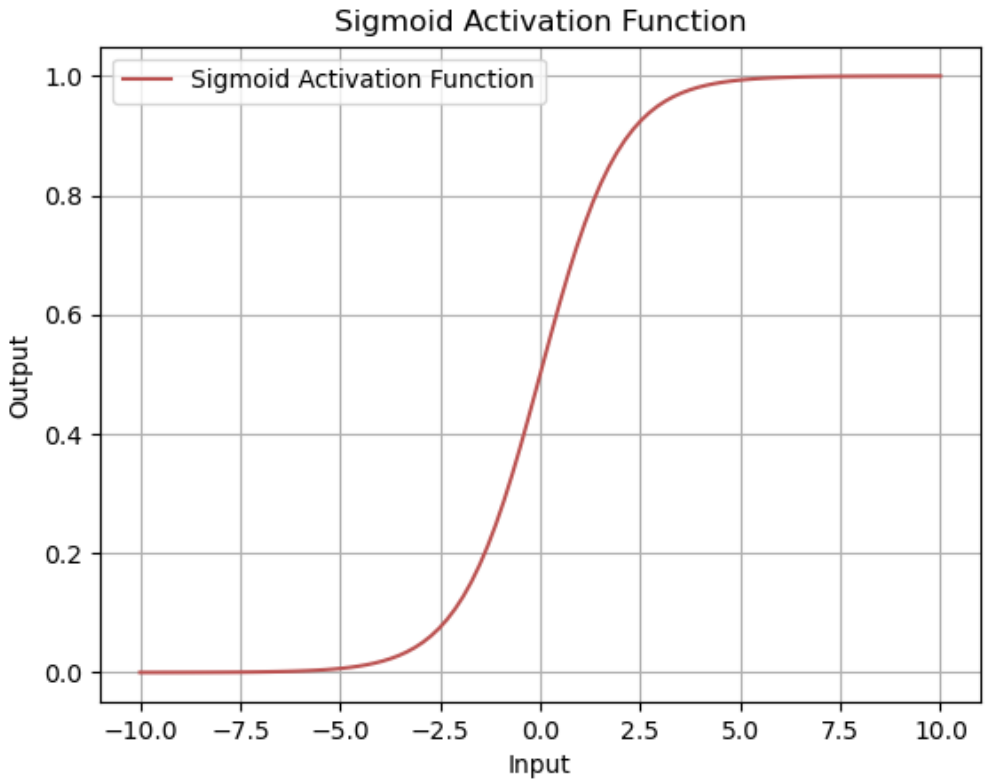


Figure 4.7: Sigmoid Activation Function Plot

There are many other popular activation functions that include *ReLU*, *LeakyReLU*, *Hyperbolic Tangent*, and so on which each have their applicability in a particular task. Choosing the correct activation function has many considerations such as the number of hidden layers in the network, the training method used, hyperparameter tuning, etc. (Sagar Sharma, Simone Sharma, and Athaiya 2017).

4.3.2 Backpropagation

Backpropagation, short for backward propagation of errors, is a fundamental algorithm that is used to train neural networks. Traditionally in early years of artificial neural network research (1950s-1980s), the training of neural networks was not common and made the use of neural networks undesirable (Aggarwal et al. 2018). Since its introduction in the 1980s, backpropagation has become a cornerstone of modern deep learning (Aggarwal

et al. 2018).

Backpropagation's main concept is based on understand how changing the weights and biases in a network impacts the cost function (Nielsen 2015). The cost function measures the network's overall performance across all training examples, serving as the optimisation target during learning. This differs from the loss function, which calculates the error for a single sample. The cost function aggregates these individual losses (typically through averaging) to guide weight adjustments across the entire dataset. The cost function is defined mathematically as (Nielsen 2015):

$$C = \frac{1}{2n} \sum_{x} ||y(x) - a^L(x)||^2 \tag{4.6}$$

where:

- n is the total number of training examples
- x is the training individual example
- $y(x)$ is the desired output
- $a^L(x)$ is the vector of activation outputs when x is the input

There are four main equations that are needed for backpropagation to work (Nielsen 2015):

1. The first equation allows the computation of the error in output layer L:

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{4.7}$$

where:

- δ^L is the error in output layer L
- $\nabla_a C$ is a vector whose components are the partial derivatives $\frac{\partial C}{\partial a_j^L}$ - in other words, the rate of change of C with respect to the output activation
- $\odot$ is the elementwise multiplication operator
- $\sigma'(z_j^L)$ measures how fast the activation function σ changes at z_j^L
- z_j^l is the weighted input of neuron j at layer l

2. The second equation allows the computation of the error δ^l in terms of the error in the following layer:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{4.8}$$

where:

- δ^l is the error in the next layer
- $(w^{l+1})^T$ is the transpose of the weight matric w^{l+1} for the $(1+l)$-th layer

3. Equation 4.7 and 4.8 can be combined to provide the third main equation of backpropagation which allows the computation of the error δ^l for any layer in the network:

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{4.9}$$

4. The fourth and final equation allows the computation of partial derivatives $\frac{\partial C}{\partial w^l_{jk}}$ in terms of δ^l and a^{l-1}:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{4.10}$$

With the above four main equations in mind, the backpropagation algorithm works as follows:

Algorithm 3 Basic Backpropagation Algorithm (Nielsen 2015)

1: **Input** value x: Set the corresponding activation a^1 for the input layer

2: **Feedforward**: For each layer, $l = 1,2,3,...,L$, compute the weighted sum $z^l = w^l a^{l-1} + b^l$ and the activation value $a^l = \sigma(z_l)$

3: **Output error** δ^L: Compute the error of the output layer using $\delta^L = \nabla_a C \odot \sigma'(z^L)$ (Equation 4.7)

4: **Backpropagate the error**: For each layer, $l = L-1, L-2, ..., 2$, computer the error for that layer using $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$ (Equation 4.8)

5: **Output**: The gradient of the cost function is given by $\frac{\partial C}{\partial b^l_j} = \delta^l_j$ and $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ (Equation 4.9 and 4.10 respectively)

In essence, the backpropogation algorithm computes the gradient of the loss function with respect to each weight in the network for a single training example, however, calculating these gradients alone does not update the network. To actually train the model, backpropagation is typically combined with an optimisation algorithm such as stochastic gradient descent (SGD). Gradient descent uses the computed gradients to iteratively adjusts the weights in a direction that minimises the overall cost function. In the case of SGD, this process is performed using small batches or individual training examples, which allows for faster and more scalable learning. This combination of backpropagation and gradient descent forms the foundation of most conventional neural network training mechanisms (Nielsen 2015).

4.4 Neural Architecture Search

Neural Architecture Search (NAS) is a technique in artificial intelligence aimed at automating the design of neural network architectures as opposed to the traditional method of crafting these neural networks by hand and through trial-and-error, both of which are time-consuming and prone to suboptimal outcomes (Ren et al. 2021). NAS addresses these challenges by employing algorithms to discover optimal or near-optimal architectures tailored for a specific task. NAS represents a significant shift towards making machine learning more scalable and accessible, bridging the gap between model development and deployment (Elsken, Metzen, and Hutter 2019).

NAS is based on three core components and are combined to form the NAS methods illustrated in Figure 4.8. (Elsken, Metzen, and Hutter 2019):

- **Search Space**: This component defines the set of possible neural network architectures that NAS can explore and includes various design elements such as layer types, number of layers, connectivity patterns, and the neural networks hyperparameters. A well-structured search space is extremely important as it balances between

being expressive enough to capture high-performing architectures while still being constrained enough to allow efficient exploration (Elsken, Metzen, and Hutter 2019).

- **Search strategy**: This component dictates how the NAS algorithm navigates the search space to identify optimal architectures. Common search strategies include reinforcement learning, evolutionary algorithms, and gradient-descent methods (Yuqiao Liu et al. 2021). Reinforcement learning makes use of a controller model to generate architectural decisions, while evolutionary algorithms mimic natural selection by evolving a population of candidate architecture solutions over successive generations. Gradient-based approaches, such as Differentiable Architecture Search (DARTS), enable more efficient exploration by allowing continuous relaxation of the search space (Yuqiao Liu et al. 2021).
- **Performance Estimation Strategy**: This component assesses how promising a candidate architecture is without fully training it, which is essential due to the high computation cost involved in evaluating each neural network design (Elsken, Metzen, and Hutter 2019). Techniques like early stopping, weight sharing, and surrogate models are used to estimate performance more efficiently (Elsken, Metzen, and Hutter 2019).

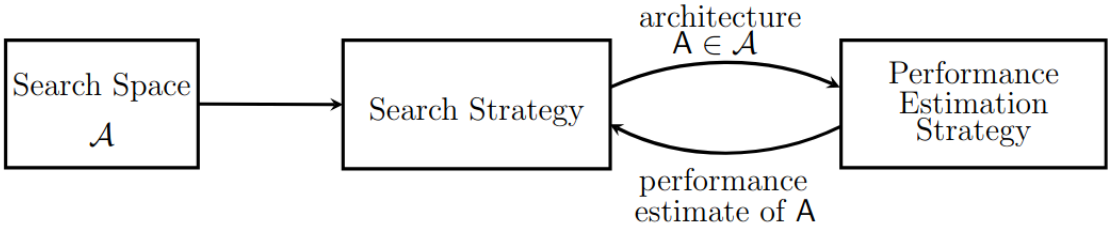


Figure 4.8: Neural Architecture Search methods framework (Elsken, Metzen, and Hutter 2019)

NAS has evolved through the development of many methods, each offering distinct ap-

proaches in navigating the search space. One prominent approach is **reinforcement learning-based** learning, where the generation of a neural network architecture is considered an agent's action and the agent's reward correlated to the performance of a trained architecture on unseen data (Elsken, Metzen, and Hutter 2019). The most influential method is **evolutionary algorithms** (mentioned in Chapter 3) which mimics biological evolution by iteratively apply genetic operators and selecting high-performing architectures. This approach is highly adaptive however can also become computationally intensive fast. More recently, **gradient-based** NAS methods as mentioned previously have gained popularity for their efficiency. DARTS for example, relaxes the discrete search space into a continuous one enabling gradient descent optimization to find optimal architectures far quicker and with reduced computational demands (Yuqiao Liu et al. 2021). These methods differ in scalability, model performance, and efficiently. Reinforcement learning and evolutionary algorithms excel in finding innovative architectures but are computationally expensive while gradient-based approaches offer faster and more resource-efficient search offering more practical application, however, do not innovate as well (Elsken, Metzen, and Hutter 2019).

Despite the promising potential that NAS provides, it still faces several significant challenges. The most pressing issue is its high computational cost (Yuqiao Liu et al. 2021). Traditional NAS methods, especially those making use of reinforcement learning and evolutionary algorithms require the training of thousands of candidate models, consuming computational resources and time exponentially, making NAS undesirable to researchers and organizations with limited infrastructure (Elsken, Metzen, and Hutter 2019). Another issue lies in the design of the search space. Creating a search space that is both expressive and efficient becomes a complex task. A poorly designed search space can either be to restrictive hindering exploration, or too expensive leading to inefficient searches and suboptimal models (Yuqiao Liu et al. 2021).

Recent advancements in NAS have focused on mitigating the challenge of computational demands and improving generalization. One notable development is the use of weight sharing whereby multiple candidate solution architectures share parameters during training. This technique significantly reduces the time needed to train each model therapy cutting computational demands (Pham et al. 2018). Another advancement in the field of NAS is the aforementioned DARTS technique which transforms the discrete search space in a continuous one, allowing for architecture parameters to be optimized efficiently using gradient descent based methods and thereby enhancing the search speed and reducing resource consumption (Yuqiao Liu et al. 2021). Additionally, strategies like proxy models and early stopping have been employed to estimate model performance without fully training each candidate solution architecture, helping mitigate excessive computation while still allowing for the consideration of innovative architectures (Yuqiao Liu et al. 2021). NAS has also extended into real-world application such as computer vision whereby NAS-designed models have achieved state-of-the-art results in image classification and object detection. In natural language processing, NAS has contributed to more efficient transformer architectures, optimizing performance for tasks such as text generation and machine translation (Elsken, Metzen, and Hutter 2019).

4.5 Neuroevolution of Augmenting Topologies

4.5.1 Introduction to NEAT

The Neuroevolution of Augmenting Topologies (NEAT) algorithm is a pioneering method in neuroevolution, known for its powerful ability evolve both the structure and weights of a neural network (Stanley, Clune, et al. 2019). NEAT begins with simple networks and gradually increases its complexity through evolutionary principles. It is based on three core principles in an attempt to solve the challenges that topology and weight evolving artificial neural networks (TWEANNs) face, namely, (1) its unique genetic encoding scheme, (2)

mechanism of speciation, and (3) minimization of dimensionality through incremental growth from minimal structures (Stanley and Miikkulainen 2002).

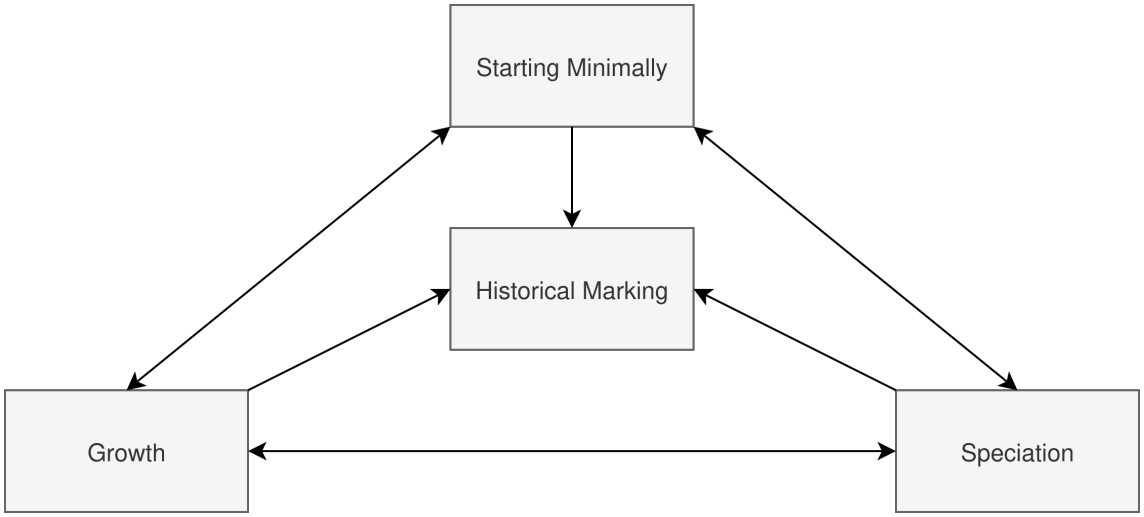


Figure 4.9: NEAT Component Dependencies (adapted from Stanley and Miikkulainen 2002)

Genetic encoding allows the transformation between phenotype, being the actual neural network representation, to genotype, being a linear string subject to genetic operators. Mutations to these genotypes introduce changes such as the addition of new nodes, altered weights, or additional connections, while crossover operations combine features from parent neural networks. These genetic operations are tailored in such away that functional integrity is maintained, ensuring that the offspring networks inherit the best traits. Importantly, NEAT aims to solve the issue of competing conventions whereby candidate solutions can possibly represent the same outcome even though they have different gene configurations. With reference to Figure 4.10, the crossover between two neural networks that are identical representations but are just different structure configurations can lead to a loss in information (Stanley and Miikkulainen 2002). This challenge stems from the isomorphism problem, where functionally equivalent networks may have different node and connection patterns, making meaningful genetic recombination difficult.

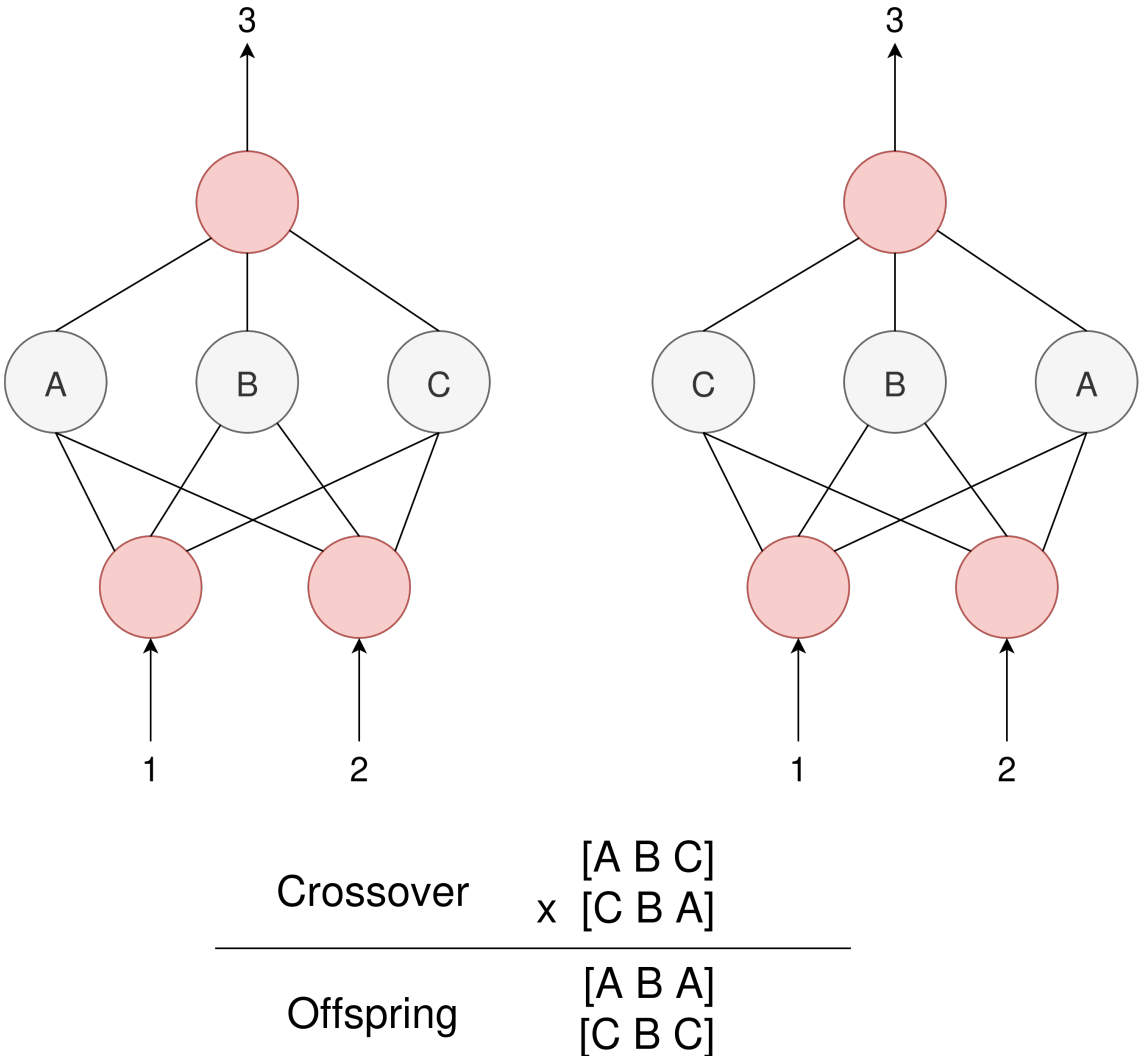


Figure 4.10: Competing Convention between Neural Network Solutions (adapted from Stanley and Miikkulainen 2002)

Speciation preserves unique structures by grouping similar networks into species, protecting novel characteristics from premature elimination. This mechanism prevents innovative offspring from being directly compared with established and well performing networks, which might otherwise out compete them prematurely. By fostering innovation within smaller networks, speciation allows NEAT to better exploit the search space and void stagnation (Stanley and Miikkulainen 2002). While speciation mechanisms exist across evolutionary algorithms (as mentioned in Chapter 3), NEAT implements this through its

distinctive approach of historical markings and compatibility distance, which constitutes its key innovation.

Incremental growth refers to the gradual introduction of nodes and connections through subsequent generations, enabling networks to evolve complexity only when advantageous. This approach avoids computational overhead when dealing with unnecessarily complex networks in early stages and ensures that the algorithm uses these networks as core building blocks in creating innovative candidate solutions (Stanley and Miikkulainen 2002).

4.5.2 Mechanics of NEAT

4.5.2.1 Initialisation of Population

The mechanics of NEAT rely on sophisticated yet efficient process that balance exploration and exploitation in evolving neural networks. The algorithm begins with a population of minimal neural networks, typically consisting of input and output nodes without hidden layers. This simplicity is intentional, as NEAT focuses on incremental growth by introducing new nodes and connection only when they enhance the networks performance. The reason being is that networks that are randomly generated and complex tend to go through the effort of removing connections and nodes that are not needed in the first place (Stanley and Miikkulainen 2002). NEAT's initialisation scheme prevents modifications where fitness functions track network size, as this would unnecessarily complicate fitness evaluation. Starting out minimally ensures that candidate solutions explore the search space in the lowest dimensional weight space first, resulting in dramatic performance gains early on. Another added advantage of starting out simply is the manifestation of speciation. Starting out simply and complexifying over subsequent generations aligns with Darwinian principles (Stanley and Miikkulainen 2002).

4.5.2.2 Phenotype-Genotype Mapping

NEAT's genetic encoding scheme is designed in such away that allows genetic operations to be performed without leading to non-functional neural networks, a problem that traditional TWEANNs face. The neural network (phenotype) is encoded in a linear representation as shown in Figure 4.11.

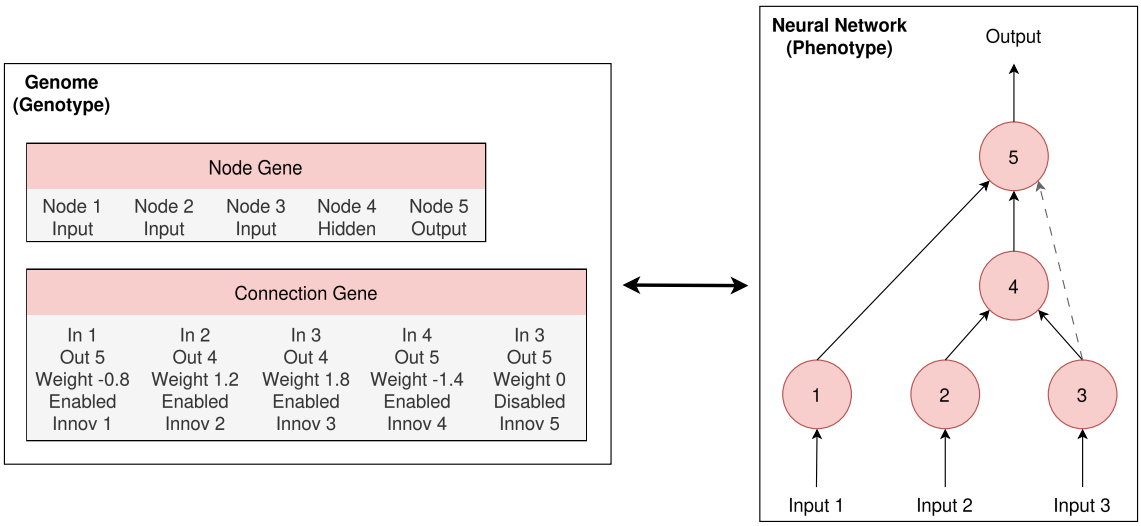


Figure 4.11: NEAT's Genotype-Phenotype Mapping (adapted from Stanley and Miikkulainen 2002)

The mapping works by assigning each node in the neural network representation with an index. The genome of the genotype representation is made up of *node genes* and *connection genes*. Node genes represents input, hidden, and output nodes that make up the network whereas connection genes specify the following information:

- **In**: The outgoing connection of the node.
- **Out**: The incoming connection of the node.
- **Weight**: The weight of the connection.
- **Enabled**: Dictates whether the connection is active or not.
- **Innov**: Serves as a historical marker in tracking corresponding genes.

4.5.2.3 Genetic Operators

NEAT makes use of both mutation and crossover to produce viable offspring. Mutation has the ability to change both the connection weights and network structures. Weight mutation works as normal in any evolutionary algorithm whereby the mutation value is chosen and changed according to some function as explained in Section 3.3.3. Structural mutations can occur in two ways, each of which expands the size of the genome (Stanley and Miikkulainen 2002):

- **Add connection** mutation: This mutation adds a single new connection between two previously unconnected nodes. Figure 4.12 illustrates this by mutating the genome such that a connection is formed between node 1 and 4.
- **Add node** mutation: This mutation splits an existing connection and places a new node where this existing connection use to be. This involves the old connection being disabled. The new connection leading from the old node into the new node receives a weight of 1 while the new connection leading from the new into the other old node receives the same weight as the old connection. Figure 4.13 illustrates this by mutating the genome such that node 6 is added, with two new connections forming between node 1 and 6, and node 6 and 5.

This mutation mechanism allows the initial effect of mutation to be minimised. In traditional mutations as those seen in genetic programming which allows extraneous structures to be added, NEAT essentially forces the structures to be evolved in complex ways later on allowing the network to exploit simple structures first (Stanley and Miikkulainen 2002). The crossover genetic operator is applicable in NEAT as well, however, enabled through a mechanism known as gene tracking through historical markings.

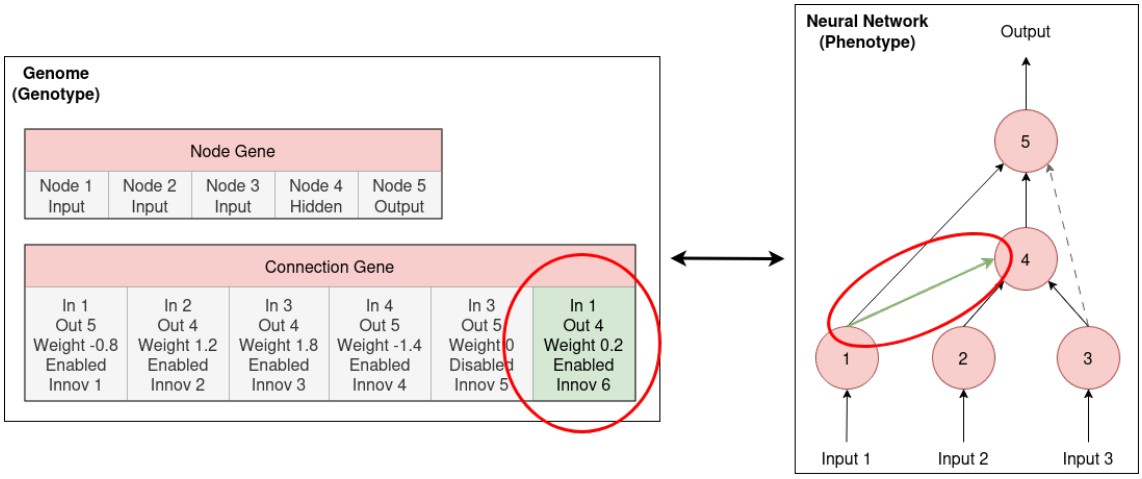


Figure 4.12: NEAT's Add Connection Mutation Mechanism (adapted from Stanley and Miikkulainen 2002)

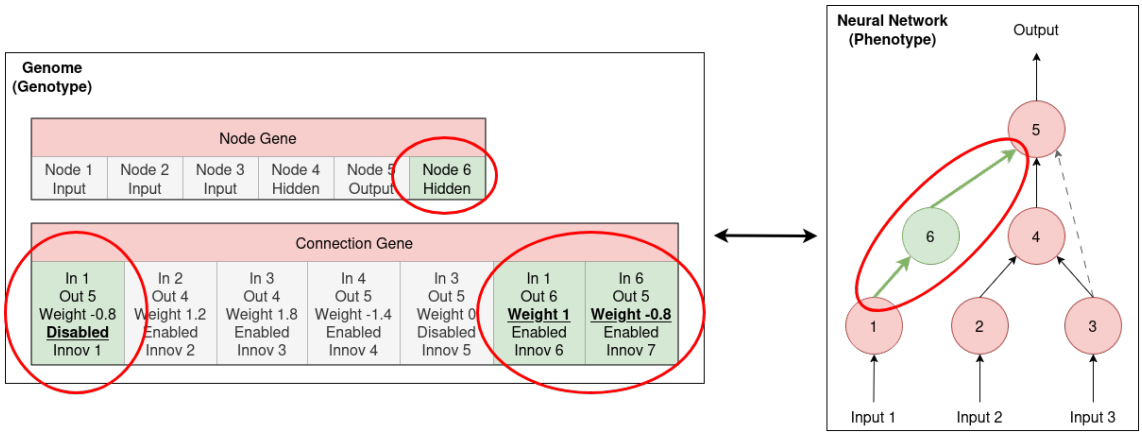


Figure 4.13: NEAT's Add Node Mutation Mechanism (adapted from Stanley and Miikkulainen 2002)

4.5.2.4 Historical Markings

Historical markings is a powerful capability of NEAT that solves the problem of competing conventions. Innovation numbers that are seen in Figure 4.11 enables this capability and represents a chronology of the appearance of every gene. When the initial population is generated, a unique innovation number is assigned to every unique connection. When a new gene appears, a *global innovation number* is incremented and assigned to that gene.

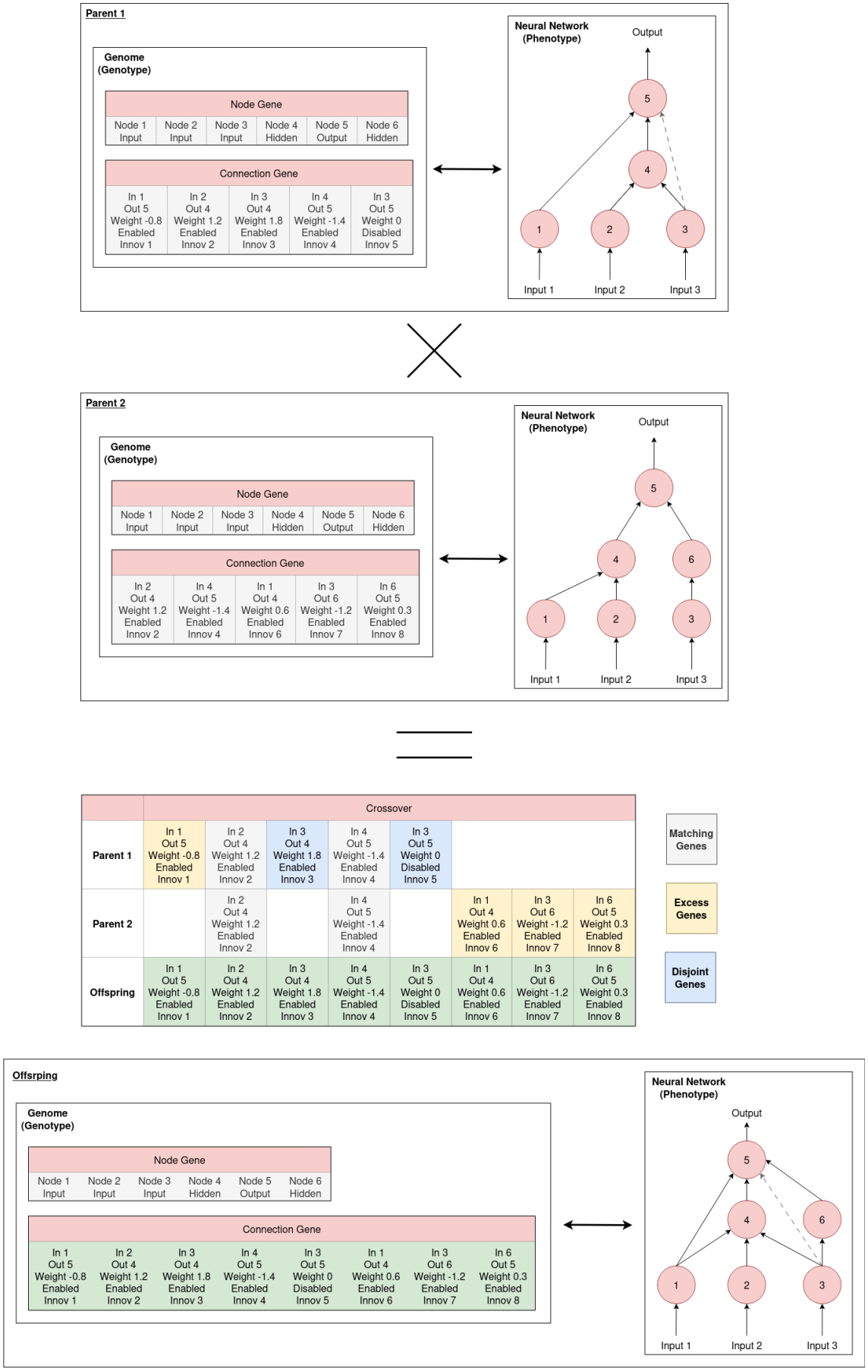


Figure 4.14: NEAT's Crossover Mechanism (adapted from Stanley and Miikkulainen 2002)

There might be a chance mutations may result in the same structural innovation amongst multiple candidate solutions within a generation - to avoid assigning different innovation numbers in this scenario, a list of gene innovations are kept for that generation meaning that if mutations result in the same structural innovation amongst multiple genes, they receive the same innovation number.

Importantly, tracking the historical markings of genes provides a way to exactly know which genes match up when performing crossover. Genes that line up according to their *innovation number* are called *matching* genes. Genes that do not split are categorized as either *disjoint* or *excess* depending what their positions are with regard to the other parent. For example, in Figure 4.14, gene 4 to 6 of parent 1, and gene 1 to 4 of parent 2 form disjoint genes while gene 7 to 8 of parent 2 form genes that are in excess. Matching genes up in this fashion using innovation allows crossover to happen naturally.

4.5.2.5 Speciation

Speciation was introduces in NEAT to allow candidate solutions to compete primarily within their own niches as opposed to the entire population. Doing this means that new topological innovations are protected in a new niche allowing them to optimize through competition within the niche. The idea behind this is to prevent premature convergence and stagnation allowing candidate solutions to exploit the search space thoroughly. Speciating is efficiently enabled through innovation numbers by comparing how many disjoint genes there are between candidate solutions. Essentially, the more disjoint two genomes are, the less historical markers they share and by that extent are less compatible (Stanley and Miikkulainen 2002). This compatibility can be calculated through the following mathematical equation (Stanley and Miikkulainen 2002):

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \tag{4.11}$$

where:

- δ is the compatibility distance between a pair of genomes

- E is the number of excess genes
- D is the number of disjoint genes
- $\overline{W}$ is the average weight difference of matching genes
- N is the number of genes in the larger genome (serving as a normalization)
- c_1, c_2, and c_3 serve as adjustments in the importance of the three factors

The compatibility distance, δ, allows speciation using a threshold δ_t. During the evolution process, a list of species is maintained, and candidate solutions are speciated in each generation. A species is represented by a random genome inside that species known as the *species representative*. Given a genome g in a generation, it is placed in the first species that it is found to be compatible within the species representative of that species - if no species is found, it forms part of a new species (Stanley and Miikkulainen 2002).

Species have the benefit of having access to all candidate solution's fitness function values within that species during reproduction which is a mechanism known as *explicit fitness sharing*. This means that a single candidate solution is unlikely to dominate the species. Speciation involves an adjusted fitness functions as follows (Stanley and Miikkulainen 2002):

$$f_i' = \frac{f_i}{\sum_{j=1}^{n} sh(\sigma(i,j))} \tag{4.12}$$

where:

- f_i' is the fitness of candidate solution i
- f_i is the chosen fitness function for the neural network
- $\sigma(i,j)$ is the compatibility distance seen in Equation 6.7 between organism i and j
- sh is the sharing function that operates as follows:

$$sh(\sigma(i,j)) = \begin{cases} 1 \; if \; \delta(i,j) > \delta_t \\ 0 \; otherwise \end{cases} \tag{4.13}$$

- $\sum_{j=1}^{n} sh(\sigma(i,j))$ reduces to the number of candidate solutions in the same species as

candidate solution i

Making use of this adjusted fitness function means that species reproduce by first eliminating the weakest links. Speciating the population in this regard ensures that the topological innovation is protected.

4.5.2.6 Minimizing Dimensionality

The last core concept in NEAT is its minimization in dimensionality through incremental growth from minimal structures. When initializing the population at the start of the algorithm, NEAT biases its search around minimal dimensional spaces by initializing candidate solutions with zero hidden nodes. As subsequent generations occur, new structures are introduced incrementally by means of mutation and crossover. This means that the topology innovations that occur are justified. Due to starting out minimally, NEAT outperforms other TWEANN based search algorithms requiring fewer dimension to find optimal solutions (Stanley and Miikkulainen 2002).

4.5.3 Strengths and Challenges of NEAT

The NEAT algorithm has several strengths that have contributed to its widespread use in artificial intelligence research. One of its primary advantages is its efficiency in evolving neural network with minimal complexity. When applied to the double pole balancing problem, NEAT was able to find an optimal solution in as little as 24 generations made up of 150 nodes, a network far more minimal than other traditional TWEANNs (Stanley and Miikkulainen 2002). By starting out with simple networks and adding complexity gradually, NEAT avoids the inefficiencies associated with random initialisation of large, complex networks. This makes NEAT particularly suitable for tasks where the optimal topology is unknown.

Another strength of NEAT is its ability to preserve diversity within the population by means

of its speciation algorithm which ensures that innovative solutions are no prematurely discarded, allowing the algorithm to explore a wide range of architectures. In the original paper of NEAT, to understand the importance of speciation, a graph was produced to illustrate how speciation impacts innovation which is shown in Figure 4.15. As subsequent generations occur, species guide their search collectively towards an optimal solution showcasing its thorough search capability. Fitness sharing within species also prevents over-optimization of a single solution thereby fostering more robust search process. These features make NEAT well-suited for solving complex and high-dimensional problems (Stanley and Miikkulainen 2002).

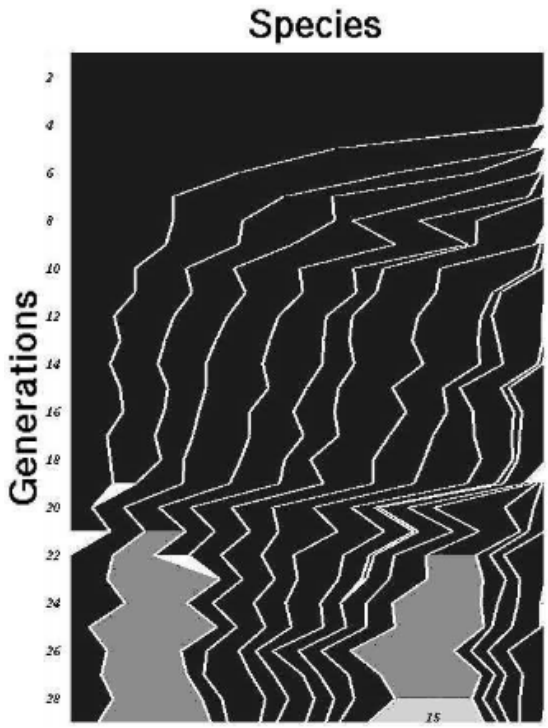


Figure 4.15: Visualizing NEAT's Speciation Mechanism while Solving the Double Pole Balancing Problem (Stanley and Miikkulainen 2002)

Additionally, NEAT offers significant advantage over traditional gradient-based methods, that is, it does not require the objective function or for the resulting neural network to

be differentiable. While conventional training relies on backpropagation and gradient descent, NEAT evaluates candidate networks based soley on their performance. This makes it applicable to a wider range of problems, including those with discrete, noisy, or non-differentiable environments (Stanley and Miikkulainen 2002). As a result, NEAT provides greater flexibility in domains where gradient information is unreliable or simply unavailable.

Despite its strengths, NEAT faces several challenges. One of the most significant is its computational costs. The simultaneous evolution of topology and weights requires high computation feature extraction (Yiming Peng et al. 2018). Additionally, because NEAT evolves arbitrary network topologies rather than layered architectures, it cannot leverage efficient matrix operations for forward propagation. Instead, each neuron must be activated individually following a topological sort, resulting in slower network evaluation. Another challenge is the sensitivity in fine-tuning the hyperparameters such as the mutation rates, speciation thresholds, and population size. Improper tuning can lead to issues such as stagnation, overfitting, or excessive computational overhead (Stanley and Miikkulainen 2002).

Overall, NEAT's strengths far outweigh its limitations, particular in optimization problems requiring innovative architectures or dynamic adaptability. As computational resources improve, many of NEAT's challenges may become less significant, further establishing its role as a powerful tool in neuroevolution.

4.5.4 HyperNEAT, a NEAT extension

Hypercube-based Neuroevolution of Augmenting Topologies (HyperNEAT), an extension of NEAT was developed to address the challenges in problems with inherent geometric or spacial structures. While NEAT excels at evolving neural networks by optimizing both topology and weights, it does not necessarily exploit the spatial relationships present

in real-world tasks such as image recognition (Stanley, D'Ambrosio, and Gauci 2009). HyperNEAT extends NEAT by introducing an indirect encoding mechanisms using Compositional Pattern Producing Networks (CPNNs), which represent neural network connectivity patterns as mathematical functions.

CPPNs allow HyperNEAT to encode complex patterns of connectivity in a compact fashion. Instead of explicitly defining each connection, CPPNs describe how connections between neurons should be formed based on their geometric position. This approach is particularly advantageous for large-scale networks, where explicitly encoding all connection would be computationally prohibitive. HyperNEAT generates neural networks that are not only scalable but also better suited to exploit the inherent spatial structure of the task (Stanley, D'Ambrosio, and Gauci 2009).

CPPNs can be thought of as a phenotype that is a pattern in space. Each coordinate in that space represents some level of expression as an output of a function that encodes the phenotype. The CPPN is a neural network that takes in coordinates and outputs a weight value which maps to a plane as shown in Figure 4.16. HyperNEAT revolves around three main steps, namely, substrate creation, CPPN evolution, and phenotype generation. The HyperNEAT algorithm is shown in Algorithm 4.

Substrate Creation involves the creation of a substrate which is a geometric representation of the neural network, where are nodes are arranged in such a way that reflects the spatial relationship inherent in the task (Stanley, D'Ambrosio, and Gauci 2009). For example, in robotic control tasks, the substrate might reflect the physical layout of sensors and actuators while in image recognition tasks the substrate might be a grid corresponding to pixel locations.

CPPN Evolution involve evolving CPPN networks using the NEAT algorithm. As mentioned, unlike a direct encoding where each connection is explicitly represented, the CPPN operates as a compact function that spits out weight values based on the relationship of underlying nodes in question. The CPPN's evolution is guided by a fitness function that evaluates the performance of the networks generated by the pattern encoded as a result from the CPPN (Stanley, D'Ambrosio, and Gauci 2009).

Phenotype Generation is the creation of the phenotype (the actual neural network) by querying the evolved CPPN for each pair of nodes in the substrate. For every set of nodes, the CPPN takes their spatial coordinates as inputs and returns a weight representing the connections characteristics such as the weight values, whether they exist or not, etc. CPPN is essentially the blueprint that maps out the phenotype (Stanley, D'Ambrosio, and Gauci 2009). The resulting networks inherit the regularities encoded by the CPPN which allows them to better integrate unseen data based on their spatial relationship.

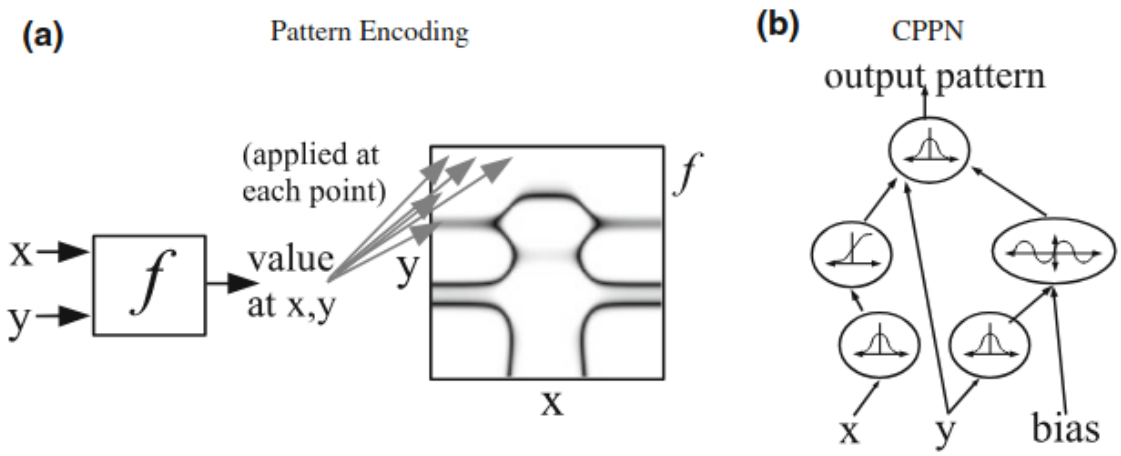


Figure 4.16: CPPN encoding where **a** represents the pattern encoding as a result from output of the CPPN in b (D'Ambrosio, Gauci, and Stanley 2014)

Algorithm 4 HyperNEAT algorithm (Stanley, D'Ambrosio, and Gauci 2009)

1: Choose substrate configuration (i.e, node layout and input-output assignments)

2: Initialize the poopulation of minimal CPPNs with random weights

3: Repeat until a solution is found:

4: For each member of the population

5: Query its CPPN for the weight of each other connection in the substrate. If the absolute value of the output exceeds a threshold magnitude, create the connection with a weight scaled proportionally to the output value

6: Run the substrate as an ANN in the task domain to ascertain fitness

7: Reproduce the CPPNs according to the NEAT method to produce the next generation population

One of HyperNEAT's notable strength resides in its ability to produce networks with inherent symmetries and patterns that mirror the problem domain (which is shown in Figure 4.16). This ability to capture and leverage geometric regularities makes HyperNEAT desirable in domains where spatial relationships are crucial. Although powerful, HyperNEAT can suffer from computational overhead, especially during the querying process of the CPPN to produce neural network solutions of a large scale (Stanley, D'Ambrosio, and Gauci 2009). Applications where HyperNEAT has been used include training agents to play games, learning autonomous agent controllers, robocup simulations, etc (Kowaliw, Bredeche, and Doursat 2014).

5. Gene Expression Programming

This chapter explores the foundational principles of GEP, its integration with neural networks through gene expression programming neural networks, and current research, highlighting GEP's role as a transformative tool in computational intelligence. An introduction to GEP is given in Section 5.1, after which the historical background is discussed in Section 5.2. Section 5.3 provides the reader with the fundamental mechanics behind GEP and it's neural network extension, GEP-NN. The strengths and challenges of GEP are discussed in Section 5.4 and finally, Section 5.5 summarizes recent advancements made in GEP and GEP-NN.

5.1 Introduction

Gene Expression Programming (GEP) is an evolutionary algorithm that blends the strengths of genetic programming and genetic algorithms seen in Chapter 3 in order to solve complex computational problems. Introduced by Candida Ferreira in 2001, GEP is distinguished by its unique encoding method which decouples the genotype (fixed-length chromosome) from the phenotype (expression trees) (Ferreira 2006). This dual representation enables the efficiency and versatility that GEP brings, enabling it to explore complex solution spaces with a balance of exploration and exploitation.

In GEP, candidate solution are represented as fixed length linear chromosomes composed of genes, which are later translated into function programs or expressions. This genotype-to-phenotype mapping is powered by Karva notation and allows GEP to maintain the structural integrity of solutions during genetic operators like crossover and mutation, addressing the key limitation in genetic programming where such genetic operations often disrupt and result in non-functional expressions.

GEP generally excels in tasks requiring symbolic regression, optimization, and automated problem-solving. It has been used in applications ranging from classification problems (CP) to automatic model design problems (AMDP) (Zhong, Feng, and Ong 2017). One of its most innovative applications is in the creation of neural networks using GEP-NN. Gene Expression Programming Neural Networks (GEP-NN) is an extension on GEP whereby the topology and weights of a neural network are evolved using the principles of GEP. The neural network is encoded in three separate domain, namely, the normal GEP genotype domain representing the nodes of the neural net, and two additional domains, *Dw* and *Dt* which encode the weight connection and threshold values respectively (Ferreira 2006).

5.2 Historical Development of GEP

The origins of Gene Expression Programming (GEP) can be traced to Candida Ferreira's groundbreaking in the early 2000s (Ferreira 2006). In attempt to overcome the limitations of genetic algorithms (GA) and genetic programming (GP), Ferreira introduced GEP in 2001 as a novel evolutionary approach that decoupled the representation of solutions in their genotype form from their phenotype functional expression representation. The introduction of this algorithm addressed challenges such as inefficiencies in genetic operations and the difficulty of evolving structurally complex solutions. This established GEP as a versatile tool for solving a wide range of optimization problems (Ferreira 2006).

Ferreira's development of GEP was motivated by limitations of GP, particularly the disruption of functional expressions during evolutionary genetic operators such as crossover and mutation. By encoding solutions as fixed-length linear chromosome, GEP ensured that when these genetic operations were applied, it would not compromise the integrity of functional structures. These chromosomes were then expressed in their phenotype form as parse trees or directed graphs, showcasing its genotype-phenotype mapping flexibility (Ferreira 2006).

The publication of Ferreira's book, *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence*, in 2001 sparked interest in the broader research community. Early experiments demonstrated GEP's superiority in tasks such as symbolic regressions, where it outperformed competing optimization algorithms by producing simpler, more interpretable solutions.

By the mid-2000s, GEP was integrated with neural networks to optimize topology and weights, giving rise to gene expression programming neural networks (GEP-NN). This marked a significant advancement, as GEP's ability to evolve the entire architecture established it as a powerful alternative to traditional training methods like backpropagation (Ferreira 2006).

Initially focused on symbolic regressions, GEP quickly found applications in other fields such as engineering and medicine (H. Malik and Mishra 2016, Kusy, Obrzut, and Kluska 2013). Researchers began adapting GEP to solve multi-objective problems and handle dynamic environments, broadening its usage (Zheng, Jia, and Cao 2012). Today, GEP is recognized as a versatile and scalable evolutionary algorithm and has been seen to be used in many other fields such as the optimization of blasting patterns in mines (Bayat et al. 2022), and its usage in predicting pavement performance, an essential metric used in the rehabilitation and reconstruction of roads (Mazari and Rodriguez 2016).

5.3 Core Concepts in Gene Expression Programming

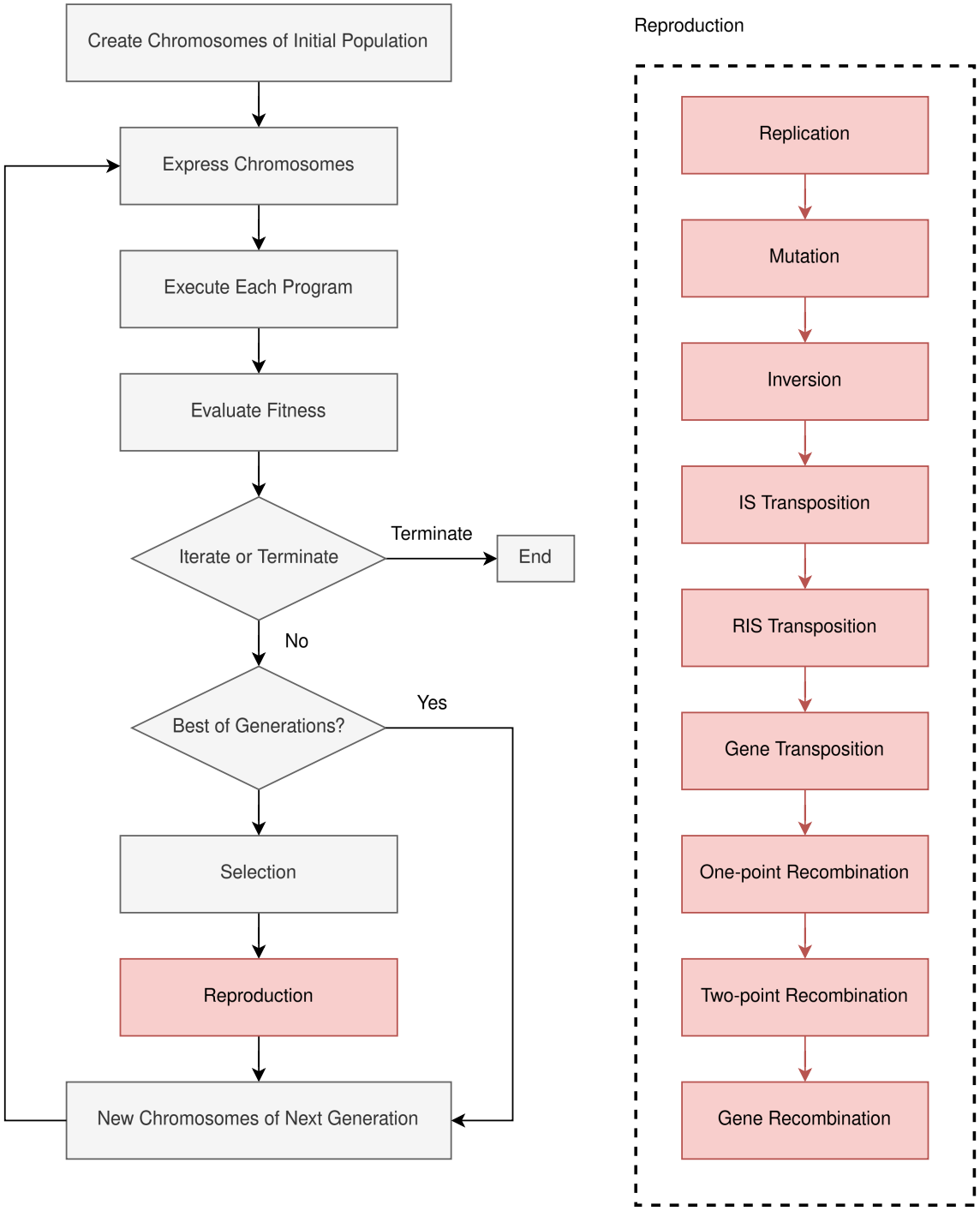


Figure 5.1: Gene Expression Programming Algorithm (adapted from Ferreira 2006)

Gene Expression Programming (GEP) is defined by its unique approach to encoding solutions, translating genetic representations into guaranteed functional outputs, and employing a suite of genetic operators to evolve populations over successive generations mimicking evolutionary processes. Understanding these core concepts is essential in grasping the algorithm that this paper proposes. The basic GEP algorithm can be summarised as shown in Figure 5.1.

5.3.1 The Genome

5.3.1.1 GEP

Understanding how GEP works begins with understanding how the genome is constructed. The chromosome is a linear fixed-length string made up of symbols (genes). The chromosome represents the genotype, and can then be decoded to represent the expression tree which is the phenotype (Ferreira 2006). For example, consider the following algebraic expression:

$$\sqrt{(a \times b) + c} \tag{5.1}$$

This equation can be represented as an expression tree seen in Figure 5.1, representing the phenotype, where Q represents the square root function.

GEP makes use of *Karva Notation* as a genotype-phenotype mapping scheme which maps an expression tree into a *K-expression*. It works by simply reading the expression tree left-to-right and top-to-bottom. The expression tree seen in Figure 5.1 can then be represented in its genotype form (K-expression) as follows:

$$\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \\ Q & + & \times & c & a & b \end{array} \tag{5.2}$$

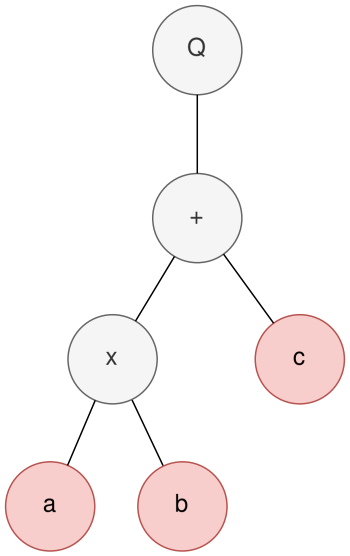


Figure 5.2: Example Expression Tree For Mathematical Equation 5.1

Although GEP primarily uses k-expressions as its mapping scheme, postfix and prefix configurations exist - using the expression tree in Figure 5.2, postfix would be encoded as seen in Equation 5.3, and prefix would be encoded as seen in Equation 5.5.

$$\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \\ a & b & \times & c & + & Q \end{array} \tag{5.3}$$

$$\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \\ Q & + & \times & a & b & c \end{array} \tag{5.4}$$

5.3.1.2 GEP-NN

As mentioned, GEP extends its methodology to encode and evolve neural networks due to its simplicity and plasticity. The general neural network seen in Section 4.3 can be easily translated in an expression tree as shown in Figure 5.3 where a and b represents the inputs $i1$ and $i2$ respectively, and D represents a function with connectivity two which serves as the weighted sum (Ferreira 2006).

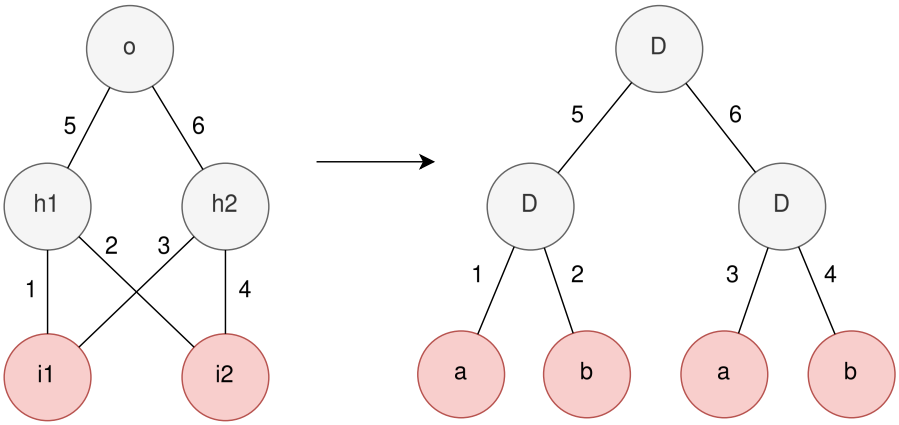


Figure 5.3: Example Neural Network and its Expression Tree Counterpart

5.3.2 Structural Organization of Genes

5.3.2.1 GEP

GEP genes are composed of two different domains, namely, the head and the tail. The head domain is used to encode the function symbols used in the optimization problem while the tail serves as a buffer of terminal symbols in order to guarantee that functional expression trees are generated. The length of the genome is made up of the head and the tail. It should be noted that the head can contain both function symbols and terminal symbols. The length of the head is predefined, and the tail is calculated as follows (Ferreira 2006):

$$t = h \times (n_{max} - 1) + 1 \tag{5.5}$$

where:

- t is the length of the tail
- h is the length of the head (predefined)
- n_{max} is the number of arguments of the function with the most arguments (also called the maximum arity)

As an example with reference to the expression to mathematical Equation 5.1, the function set would be $F = \{Q, \times, +\}$ and the terminal set would be $T = \{a, b, c\}$. The maximum arity n_{max} would be 2 as multiplication and addition both take in two arguments (as opposed

to Q/square root which takes one argument). Provided a head size of 4 was chosen, the tail length would then be $t = 4 \times (2-1) + 1 = 5$. The chromosome length would as a result be $h + t = 4 + 5 = 9$. A chromosome representing Equation 5.1 could look as follows (tail is in bold):

$$\begin{array}{ccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ Q & + & \times & a & \mathbf{b} & \mathbf{c} & \mathbf{a} & \mathbf{b} & \mathbf{a} \end{array} \tag{5.6}$$

The tail acts like a reservoir of terminal symbols meaning that functional expression trees are always a given. Not all terminal symbols are guaranteed to be used in the phenotype representation - those symbols that are not used are referred to as the non-coding region, whilst the symbols that are used is referred as the coding region (Ferreira 2006).

GEP chromosomes do not necessarily need to contain only one gene - multigenic systems can easily be represented, for example:

$$\begin{array}{cccccccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ Q & + & \times & a & \mathbf{b} & \mathbf{c} & \mathbf{a} & \mathbf{b} & \mathbf{a} & + & + & Q & b & \mathbf{a} & \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{a} \end{array} \tag{5.7}$$

These two genes represent two respective sub-expression trees. By incorporating mutligenic systems, complex genomes can be created by smaller genomes, a common occurrence seen in nature (Ferreira 2006). For optimization problems with multiple outputs such as classification problems, the different sub-expression trees are seen as autonomous agents working together to each produce a particular output for a set of given inputs (Ferreira 2006). Sub-expression trees can also be combined using a linking function to form a larger expression tree as shown in Figure 5.4 (using Equation 5.7). Multigenic chromosomes guide the evolution in modular structures, a small building block in order to create more complex structure.

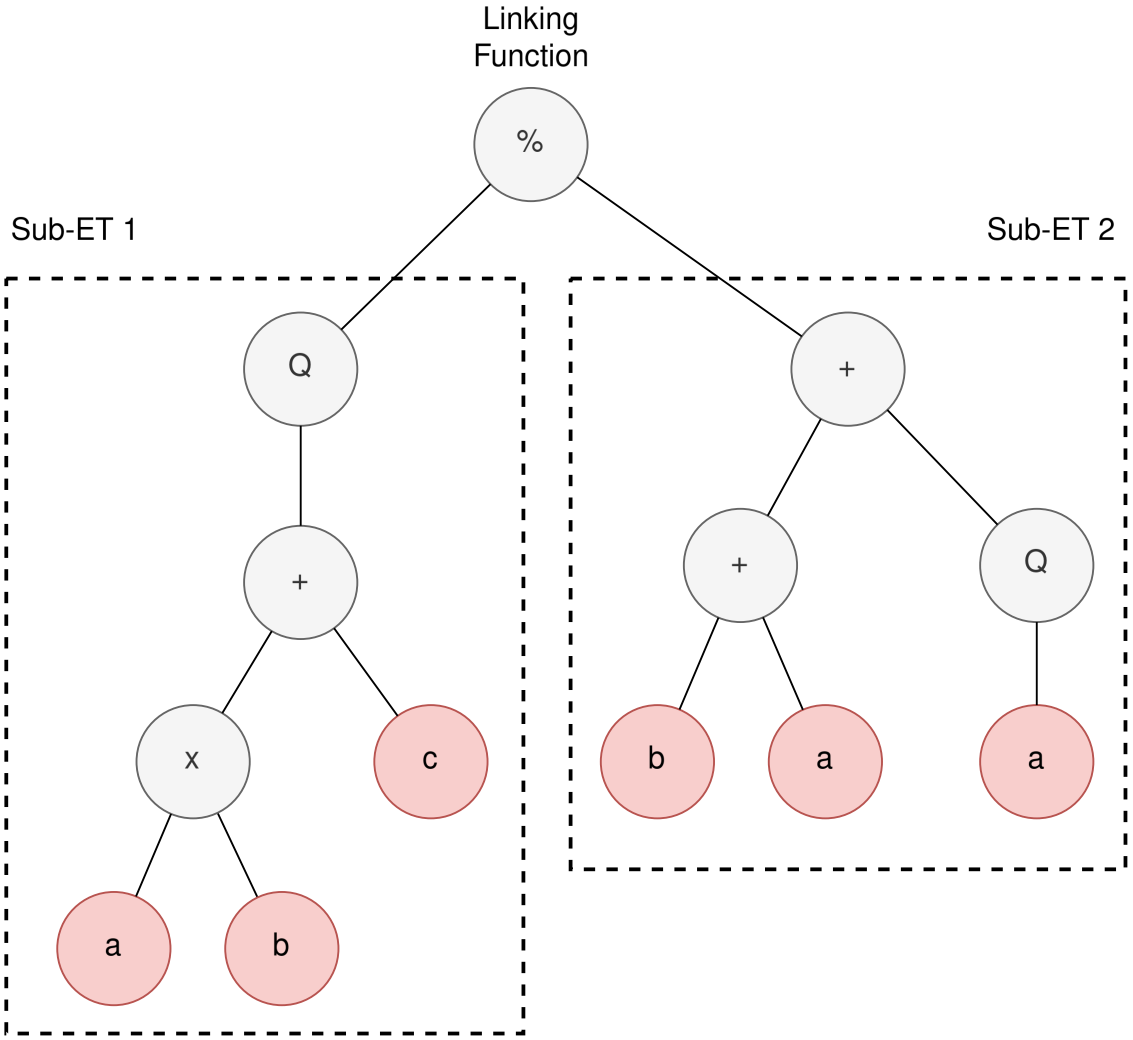


Figure 5.4: Example of Linking Function to Connect Sub Expression Trees

5.3.2.2 GEP-NN

When using GEP genomes to represent neural networks, two additional domains are required in order to encode the weights of the connectivities between nodes, and the thresholds used when applying an activation function (Ferreira 2006). These two domains, denoted, *Dw* and *Dt*, are encoded in their own regions within the chromosome. Depending on the problem at hand in which a neural network needs to be designed for, these additional domains are characterized by the outcome. In some instances, the threshold domain is not required, especially when using activation functions that are non-linear. The chromosome

for the expression tree seen in Figure 5.3 would look as follows:

$$\begin{array}{cccccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 0 & 1 & 2 \\ D & D & D & a & b & a & b & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} & \mathbf{6} \end{array} \tag{5.8}$$

where position 0 to 6 represents the neural network gene region, and the positions in bold are the weights. Typically, the *Dw* gene stores indices and is used to fetch indices from a lookup array as necessary. This array could look as $W = \{-1.2, 0.8, -0.3, 1.4, 0.5, -2.0\}$. The length of the *Dw* domain is calculated using $Dw_{length} = h \times n_{max}$ and the length of *Dt* is calculated using $Dt_{length} = h$ (Ferreira 2006).

5.3.3 Genotype Operators

GEP employs a variety of genetic operators to balance exploration and exploitation. These operators are tailored to its fixed-length genotype and ensure syntactically correct phenotypes. These genetic operators are modified in various ways to extend its application in the GEP-NN domain.

5.3.3.1 Mutation

Mutation in GEP is one of the most efficient genetic operators allowing candidate solutions to adapt very quickly. For a given mutation p_m, two one-mutations operations are applied to the chromosome. If the mutation point happens to be in the head, the mutation can result in symbols from both the function and terminal set whereas if it occurs in the tail, only symbols from the terminal set can be chosen (Ferreira 2006). More importantly, a single mutation can have a profound effect on the phenotype, especially if the mutation involves change from function set symbols to terminal set symbols as the entire expression tree structure would be altered. In the GEP-NN counterpart, mutation is able to also occur in the *Dw* and *Dt* domains using novel schemes for mutating constants (Ferreira 2006).

5.3.3.2 Inversion

GEP Inversion is similar to the inversion scheme seen in Section 3.3.3.2 with the difference that it operates within the head of the chromosome which ensures that no matter what start and termination points are chosen, syntactically correct programs will always be produced (Ferreira 2006). In GEP-NN, inversion is applicable to both the *Dw* and *Dt* and works as implied for a given inversion rate, p_i. Inversion in this domain ensures the circulation of weights and thresholds within the genetic pool.

5.3.3.3 Transposition

Transposition moves a segment of a gene to another location within the same chromosome in an attempt to create repetitive sequences within the phenotype. There are three kinds of transposable elements in GEP (Ferreira 2006):

1. **Insertion Sequence (IS) Elements** - Fragments beginning with either a function or terminal that transpose to the head of genes, except the root.
2. **Root Insertion Sequence (RIS) Elements** - Fragments beginning with only a function that transpose to the root genes.
3. **Entire genes** that transpose to the beginning of chromosomes.

During the transposition of IS elements of transposition rate, p_{is}, any sequence within the element can become an IS element. This element is then copied at the place of origin and inserted at the randomly chosen point in the gene head (except the start position) (Ferreira 2006). Its worth noting that once the IS element is copied, it can only be inserted strictly within the gene head to ensure functional expressions.

During the transposition of RIS elements of transposition rate, p_{ris}, a sequence beginning with a function is chosen, and inserted in a randomly chosen start and end position. RIS differs from IS transposition in its ability to have the root of the chromosome as the start point (Ferreira 2006).

During gene transposition of transposition rate, p_{gr}, entire genes inside the chromosome are transposed and in effect make repeatable structures amongst chromosomes that consist of more than one expression tree (Ferreira 2006).

Transposition can also be applied to GEP-NN genes in the *Dw* and *Dt* domain, with the added advantage that any sequence in the domain can be chosen as the transposition element (Ferreira 2006). This ensures that new weight and thresholds get tested at different connectivity locations within the neural network.

5.3.3.4 One-point Recombination

One-point recombination involves two randomly chosen parents to be paired side by side and split up, producing two offspring. Its recombination rate is specified as p_{1r}. It works exactly as what was seen with single point crossover in Section 3.3.3.2 (Ferreira 2006). One-point recombination can have monstrous effects disrupting old building blocks and continually forming new ones.

5.3.3.5 Two-point Recombination

Two-point recombination involves two randomly chosen parents to be paired side by side and split at two points randomly, producing two offspring. Its recombination rate is specified as p_{2r}. Two-point recombination is far more disruptive than one-point recombination since genetic material is combined more thoroughly thereby destroying existing building blocks and creating new ones. When dealing with GEP-NN chromosomes that consist of multiple sub-NNs, two-point recombination is restricted within the sub-NN in order to allow fine grain control and ensure that weights and thresholds amongst other sub-NNs are kept intact (Ferreira 2006).

5.3.3.6 Gene Recombination

Gene recombination operates in the same fashion as gene transposition, that is, entire genes are exchanges between two parent chromosomes, at a gene recombination rate, p_{gr}. The

intent of gene recombination is not to create genes, but instead test different configuration of existing genes, especially in scenarios where they are collectively put together through a linking function (Ferreira 2006). Likewise, in GEP-NN chromosomes, gene recombination serves as mechanism to exchange entire sub-NNs as a balance mechanism between exploration and exploitation.

5.3.4 Population and Fitness

As in any evolutionary algorithm, the initial population of candidate solutions are generated through a randomized approach. In the case of GEP, these chromosomes are generated in a way that is adhered to the head and tail constraints mentioned, that is, the head would consist of symbols drawn from the function and terminal set whereas the tail would consist of symbols drawn only from the terminal set (Ferreira 2006). After the population is initialized, a fitness function is applied to each individual. According to Ferreira, for populations of a small size or set of fitness cases not broad enough, the population in its entirety might produce a fitness of 0. To mitigate this issue, GEP employs that a start-up control mechanism is used to in which an initial population is generated until a single viable solution is obtained. In doing so, GEP evolves candidate solutions that are descendants of a sole viable founder.

Fitness evaluations measure how well an individual solves the given problem, guiding the selection of individuals for reproduction. GEP proposes a few fitness functions that can be used depending on the task at hand. A few fitness functions will be discussed that are applicable to different optimization problems.

5.3.4.1 Fitness Functions for Symbolic Regression

Symbolic regression tasks involves finding a symbolic expression that excels at numeric prediction. There are four main fitness functions that Ferreira discusses which are summarised as follows:

1. **Number of Hits** Fitness Function - This fitness functions is designed for models that need to perform well for numerous fitness cases within a certain error, whether it is absolute or relative (Ferreira 2006). The fitness of an individual candidate solution i for fitness case j is evaluated by the formula:

$$f_{ij} = \begin{cases} 1 \; if \; E_{ij} <= p \\ \\ 0 \; otherwise \end{cases} \tag{5.9}$$

where p is the precision and E_{ij} is the error of candidate solution i and fitness case j (Ferreira 2006). Depending on whether the error is relative or absolute is defined by the following equations respectively:

$$E_{ij} = |P_{ij} - T_{ij}| \tag{5.10}$$

$$E_{ij} = |\frac{P_{ij} - T_{ij}}{T_j} \times 100| \tag{5.11}$$

where P_{ij} is the predicted value for candidate solution i and fitness case j, and T_j is the target value for fitness case j.

2. **Precision and Selection Range** Fitness Function - This fitness function operates by means of a selection range whereby if the candidate solutions fitness performs well it contributes to the overall fitness and if it does poorly it weakens the overall fitness (Ferreira 2006). The fitness can use either relative error or absolute error as follows:

$$f_i = \sum_{j=1}^{n} (R - E_{ij}) \tag{5.12}$$

where E_{ij} refers to Equation 5.10 if using relative error, and E_{ij} refers to Equation 6.6 is using absolute error.

3. **Mean Squared Error** Fitness Function - This fitness function is one of the most widely used and is based on the standard mean squared error, E_i (Ferreira 2006). The mean squared error is calculated as follows:

$$E_i = \frac{1}{n} \sum_{j=1}^{n} (E_{ij})^2 \tag{5.13}$$

where E_{ij} refers to Equation 5.10 if using relative error, and E_{ij} refers to Equation 6.6 is using absolute error. In order for a perfect fitness to occur, P_{ij} needs to equal T_j for all fitness cases meaning that the mean squared error ranges from 0 to infinity. Its worth noting that this fitness function can't be used for evolutionary selection mechanisms that involve proportionate selection as the fitness in this scheme needs to increase. In such as situation, the following fitness evaluation needs to be used ranging its fitness number from 0 to 1000:

$$f_i = 1000 \times \frac{1}{1+E_i} \tag{5.14}$$

4. **R-square** Fitness Function - This fitness function is based on the standard R-square returning the square of the Pearson product moment correlation coefficient, R_i, and is calculated as follows (Ferreira 2006):

$$R_i = \frac{n\sum_{j=1}^{n}(T_j P_{ij}) - (\sum_{j=1}^{n} T_j)(\sum_{j=1}^{n} P_{ij})}{\sqrt{[n\sum_{j=1}^{n} T_j^2 - (\sum_{j=1}^{n} T_j)^2][n\sum_{j=1}^{n} P_{ij}^2 - (\sum_{j=1}^{n} P_{ij})^2]}} \tag{5.15}$$

When $R = 1$, there is an optimal positive linear correlation, when $R = -1$, an optimal negative linear correlation between P and T, and when $R = 0$, there is no correlation. The fitness is then calculated using the equation below, meaning that the fitness ranges from 0 to 1000.

$$f_i = 1000 \times R_i^2 \tag{5.16}$$

5.3.4.2 Fitness Functions for Classification and Logic Synthesis

Classification optimization problems focus on finding a model that accurately maps inputs to predefined categories or classes whereas logic synthesis focuses on optimizing logic circuits to implement Boolean functions efficiently. GEP has applications in both with a wide selection of fitness functions that Ferreira mentions:

1. **Number of Hits** Fitness Function - This fitness function corresponds to the number of samples correctly classified. The fitness of an individual candidate solution, i, can be evaluated as follows (Ferreira 2006):

$$f_i = h \tag{5.17}$$

where h is the number of fitness cases evaluated correctly.

2. **Hits with Penalty** Fitness Function - This fitness function mitigates the chances of favoring candidate solutions that end up in a local optima. It works by recording the number of true positives (TP_i) and true negatives (TN_i). The fitness function is then calculated as follows (Ferreira 2006):

$$f_i = \begin{cases} 0 \; if \; TP_i = 0 \; OR \; TN_i = 0 \\ \\ h \; otherwise \end{cases} \tag{5.18}$$

where h is the number of fitness cases evaluated correctly, that is $h = TP_i + TN_i$. The fitness, f_i has a maximum value as per Equation 5.17.

3. **Sensitivity/Specificity** Fitness Function - This fitness function is based on sensitivity and specificity indicators. The sensitivity indicator reflects the probability that a fitness case detects a true positive in proportion to a positive fitness case. The specificity indicator reflects the probability that a fitness case detects a true negative in proportion to a negative fitness cases (Ferreira 2006). These two indicators are calculated as follows:

$$SE_i = \frac{TP_i}{TP_i + FN_i} \tag{5.19}$$

$$SP_i = \frac{TN_i}{TN_i + FP_i} \tag{5.20}$$

This sort of fitness function is useful in applications that involve highly unbalanced training sets since the result of multiplying both these indicators results in an index forcing the discovery of models that have both a high sensitivity and specificity probability (Ferreira 2006). The fitness of candidate solution, i, would be calculated as follows:

$$f_i = 1000 \times SS_i \tag{5.21}$$

where $SS_i = SE_i \times SP_i$.

4. **Positive Predictive Value / Negative Predictive Value** Fitness Function - This fitness function is based on the positive predictive value (PPV) and negative predictive

value (NPV) indicators (Ferreira 2006). PPV reflects the percentage of true positive cases and NPV reflects the percentage of false negatives cases. These two indicator are calculated as follows for candidate solution, i:

$$PPV_i = \frac{TP_i}{TP_i + FP_i} \; where \; TP_i + FP_i \neq 0 \tag{5.22}$$

$$NPV_i = \frac{TN_i}{TN_i + FN_i} \; where \; TN_i + FN_i \neq 0 \tag{5.23}$$

Multiplying both these indicators results in the following equation:

$$PN_i = PPV_i \times NPV_i \tag{5.24}$$

Evaluating the fitness of candidate solution, i, is done as follows:

$$f_i = 1000 \times PN_i \tag{5.25}$$

Again, this fitness function ranges from 0 to 1000 with 1000 being the best solution.

5.3.5 Selection Mechanisms

There are many selection mechanisms that can be used in evolutionary computing, a few of which have been discussed in Section 3.3.4. The selection operator chooses which candidate solutions are to be reproduced according to its fitness. GEP is not restricted to any specific selection mechanisms, however, in many of the optimization tasks that GEP was applied to in the original paper, elitism was used which produced favourable results (Ferreira 2006).

5.4 Strengths and Challenges

Gene Expression Programming is a powerful evolutionary algorithm that excels in solving complex problems through adaptive modeling and optimization. One of its greatest strengths lies in its genome representation of solutions. Unlike traditional Genetic Algorithms (GAs) and Genetic Programming (GP) (seen in Section 3.4.1 and 3.4.3 respectively),

GEP decouples the genotype and phenotype, enabling better exploration capabilities in the solution space. These separations allow for the evolution of highly complex expression while maintaining simplicity in genetic operations. GEP effectively mitigates one of the major challenges that GP struggles with which is enforcing complex rules in order to ensure that the evolved program is syntactically functional. During evolutionary processes of GP, special care needs to be taken in the way in which candidate solutions produce offspring, mutated, etc., all of which require often complex computation. By representing solutions as linear chromosomes that are later expressed as expression trees, GEP combines the advantages of linear and tree-based models, facilitating the discovery of sophisticated solutions without the complications of direct tree manipulations seen in GP. This structure enables the algorithm to evolve solutions that are both powerful and adaptable.

GEP also presents certain challenges. One of the primary limitations is the computational cost associated with evolving large populations over many generations, especially when dealing with high-dimensional data or complex spaces (Zhong, Feng, and Ong 2017). As the complexity of the problem increases, so does the demand for computational resources, potentially making GEP less practical for time-sensitive tasks. As the case with many evolutionary algorithms, GEP may struggle with premature convergence, where population loses diversity and becomes trapped in a local optima, preventing the discovery of better candidate solutions. Maintaining genetic diversity within the population is critical but difficult, requiring well-balanced genetic operators to avoid stagnation. Parameter tuning such as selecting population size, mutation rates, function sets, etc., can be complex and problem-dependent, often times requiring trial and error to optimize performance. Many GEP adaptions make use of self adaption in an attempt to mitigate this problem, however, not much research has been done in this domain (Zhong, Feng, and Ong 2017). Balancing exploration and exploitation in the evolutionary search process remains a delicate task, often influencing the efficiency and success of GEP applications. Addressing these

challenges requires a thoughtful algorithm design and problem specific customization to fully leverage GEP's flexibility and power.

5.5 Current Research

As expressed in a comprehensive review article by Zhong, Feng, and Ong 2017, there have been numerous advancements in the research of GEP. These advancements can be categorized in three different areas of research in the context of this literature review:

Adaption Design in GEP is a mechanism that evolves control parameters such as population size, chromosome length, mutation rate, etc. within the evolutionary algorithm in order to improve optimization performance. There have been two adaptive GEP approaches, namely, AdaGEP and Adapative GEP. **AdaGEP** is a proposed extension of the original GEP algorithm that dynamically adjusts the number of active genes in chromosomes using a binary *genemap* (E. Bautu, A. Bautu, and Luchian 2007). Genes marked with a zero within this chromosome indicate that the subtree is ignored when being decoded while the being marked with one means that the subtree is active. This *genemap* is evolved with the traditional GA genetic operators. The optimum number of genes used to solve the problem is evolved on the fly aiding to its increased performance over traditional GEP. **Adaptive GEP** makes use of feedback heuristic to modify operator execution probabilities (Mwaura and Keedwell 2009). In other words, candidate solutions with higher fitness have increased probability of mutating and generating offspring as solutions with lower fitness are less likely to mutate and produce offspring.

Cooperative Coevolutionary (CC) is a framework aimed at solving large-scale optimization problems by breaking them into simpler subproblems. Each subproblem is then solved independently by making use of separate evolutionary algorithms with distinct subpopulations (Zhong, Feng, and Ong 2017). CC creates two problems to overcome -

figuring out an effective way to decompose the problem, and finding a way to evaluate candidate solutions effectively. Sosa-Ascencio et al. proposed a new algorithm, CC-GEP, which encorporates GEP evolving two subpopulations, one representing the main program, and the other evolving automatically defined functions (ADFs) (Zhong, Feng, and Ong 2017). Furuholment et al. applied the concept of CC-GEP to the distributor's pallet packing problem which is an optimization problem to maximize the volume a set of distinct boxes certain dimensions on pallets, and the 3D process plant problem, both of which incorporated fitness sharing amongst the subpopulations to leverage a fine balance of exploration and exploitation of the search space.

Parallel Design in GEP is an approach to circumvent the issue of slow iterative evolution processes for large-scale and complex optimization problems. There are two parallelization models that researchers have applied GEP - the master-slave model and the island model (Zhong, Feng, and Ong 2017). The master-slave model operates by incorporating a master thread that manages the main program, while multiple slave threads handle subtasks concurrently. pGEP, an example of parallelization use in GEP, works my utilizing a master thread to manage GEP, with GPUs performing parallel fitness evaluations on the candidate solutions (Zhong, Feng, and Ong 2017). The island model, similar to the idea of CC, divides the population into subpopulations and each evolve independently by different processors. Subpopulations make use of fitness sharing amongst individuals to better enhance exploration and exploitation. Parallel niche gene expression programming (PNGEP) introduced by X. Li et al. 2005, uses niching techniques to maintain diversity, exchanging the best individuals through a shared pool amongst subpopulations (Zhong, Feng, and Ong 2017). Other models of parallelism include multiagent models whereby autonomous agents are used for task distribution.

Other advancements presented in this survey included constant creation design mechanisms to better provide highly accurate constant values, GEPs application in automatic model

design problems (AMDPs), and other problem specific optimization problems (Zhong, Feng, and Ong 2017). Another interesting, although not fully explored improvement in GEP is a study conducted by X. Li et al. 2005, whereby the encoding scheme was investigated, forming a new algorithm coined prefix gene expression programming (P-GEP). As opposed to making use of the bread-width like encoding scheme used when converting from expression trees to karva notation, prefix notation is used, an example of which was seen in Equation 5.5. Prefix notation has two major characteristics (X. Li et al. 2005):

- **Substructure Preserving characteristics**: Due to the nature in which prefix notation constructs the genome string, there is a more direct correlation between the genotype and phenotype. Nodes from the same sub-tree appear adjacent to one another meaning that subroutines can be better recognized as segments within the chromosomes thereby accounting for tighter genetic linkages. Having tightly linked genes positioned adjacent to one another means that the typical destructive genetic operators have less of monstrous effect thereby preserving structure and incorporating small change instead (X. Li et al. 2005).
- **Inherent Hierarchy in Forming the Solution**: Because nodes from the same sub-tree appear adjacent to one another means that the hierarchies seen in the linear representation form incrementally and built upon the lower ones naturally (X. Li et al. 2005). Figure 5.5 illustrates this inherent hierarchy, and as well, shows that the genotype and phenotype both encode functional complexity similarly. Put simply, the linear chromosome representation and the expression tree both share the same level of expressiveness.

Based on the experimental results of P-GEP, there are considerable improvements compared to the original GEP algorithm. By making use of prefix notation, the traditional GEP genetic operators are more protective for substructures which aids more refined search

capabilities. GEP has two major shortcomings:

- Constructing expression trees require frequent heap operations effecting the efficiency of the program. The larger the expression tree, the larger the number of heap operations.
- Expression trees are non-linear meaning that the larger the tree, meaning that non-linear memory address allocations become vastly complex

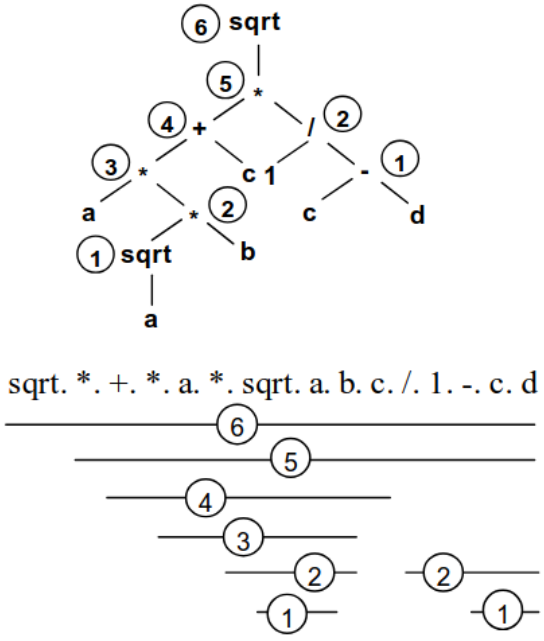


Figure 5.5: Illustration of hierarchy of P-GEP (X. Li et al. 2005)

S_GEP, an extension of *P-GEP*, has a unique characteristic to overcome the above shortcomings, which is its ability to evaluate the linear chromosomes using a stack (YuZhong Peng et al. 2014). In order to evaluate the chromosome using a stack, the effective length of the gene needs to be calculated which represents the coding region. The effective gene length, *el*, is calculated as shown in Algorithm 5.

The effective gene length can then be used to evaluate the chromosome using a stack which is then visualized in Figure 5.6. The effective gene is read in Polish notation fashion whereby the symbols are read right to left. As terminals are read, they are pushed onto the

stack, and when a function symbol is encountered, it takes the terminals on the stack as input. The result is then pushed back on the stack and the process repeated until a single result remains. This mechanism can be summarized using Algorithm 6.

Algorithm 5 Effective gene length (adapted from YuZhong Peng et al. 2014)

Input: String of gene symbols
Output: eL
Initialize variables $Len = 1$ and $count = 0$
while scanning gene symbols starting from left to right
    $Len = Len + 1$;
    if current gene symbol is function:
        $count = count - 1$
    else:
        $count = count - 1$
    if $count < 0$
        return $eL = Len$ as output

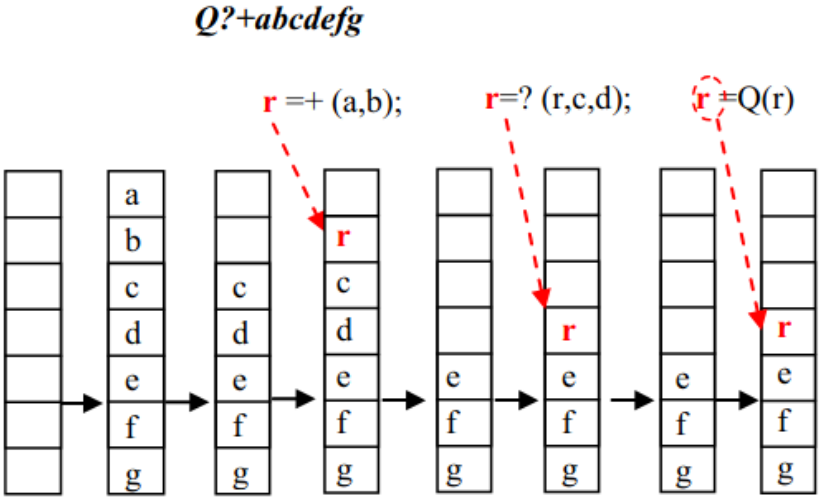


Figure 5.6: Evaluation of GEP chromosome using stack (YuZhong Peng et al. 2014)

S_GEP proved to show favourable results by outperforming traditional GEP in terms of its computation efficiency (YuZhong Peng et al. 2014). Although applied primarily to regression problems, *S_GEP* poses as a powerful technique in general application, however, in the context of this research, GEP-NN specifically.

Algorithm 6 S_GEP Algorithm (adapted from YuZhong Peng et al. 2014)

```
1: Calculate eL using Algorithm 5
2: while reading effective gene from right to left using eL
3:     if current gene is a terminal symbol:
4:         push terminal symbol on stack
5:     else if current gene is function symbol:
6:         pull terminal symbols from stack and feed to function symbol
7:         push result value to stack
8: pull value from stack and return
```

6. GEP-NEAT

This chapter covers GEP-NEAT, a blend between NEAT (covered in Chapter 4.5) and gene expression programming (covered in Chapter 5). An introduction to GEP-NEAT is given in Section 6.1. The proposed algorithm is explained in-depth in Section 6.2, after which the prototype implementation is discussed in Section 6.3. Finally, the prototype is applied to a few problems to benchmark the algorithm's performance in Section 6.4.

6.1 Introduction

As discussed in Chapter 4, neuroevolution explores innovative strategies for optimising artificial neural networks (ANNs) using evolutionary algorithms (see Chapter 3). Unlike conventional training methods such as backpropagation and gradient descent, which rely on differentiable loss functions and gradient-based updates, neuroevolution leverages Darwinian principles to evolve the topology of the neural network along with its weights and biases. This allows for the discovery of novel topologies, activation functions, and hyperparameters that may not be easily accessible through traditional approaches.

A prominent example of this is the NEAT algorithm (Chapter 4.5), which evolves both the structure and weights of neural weights simultaneously. NEAT's speciation mechanism

preserves diversity and protects innovative structures during evolution, but one of its main challenges lies in its computational cost, largely due to the need for topological sorting during network evaluation, which makes the genotype-phenotype mapping and forward propagation computationally expensive. Nevertheless, NEAT's ability to operate without requiring differentiable functions makes it particularly well-suited for environments where gradient information is unaccessable.

Building on these ideas, Gene Expression Programming (GEP) and its extension GEP-NN offer alternative evolutionary approaches that combine the global search capabilities of genetic algorithms with the expressive, tree-based representations of genetic programming. GEP introduces a novel genotype-to-phenotype mapping using Karva notation, enabling the evolution of complex, non-linear structures such as expression trees.

The proposed algorithm, GEP-NEAT, integrates the representational power of GEP with the structural evolution principles of NEAT. It introduces several enhancements, including a reworked system for tracking innovation numbers, inspired by Richard Dawkins' concept of the meme. This mechanism enables more effective selection and mutation by identifying and preserving useful substructures across generations. Furthermore, GEP-NEAT addresses a key limitation in earlier GEP-based models by introducing a method for embedding bias information directly into the chromosome, thereby improving the expressiveness and learning capacity of evolved networks.

6.2 Proposed Algorithm

6.2.1 Evolutionary Life Cycle

The goal of GEP-NEAT is to integrate the expressive representational capabilities of Karva notation from GEP with the neuroevolutionary mechanisms of NEAT, particularly its speciation and innovation tracking strategies. This hybrid approach aims to evolve neural

network populations more effectively by combining the strengths of both paradigms.

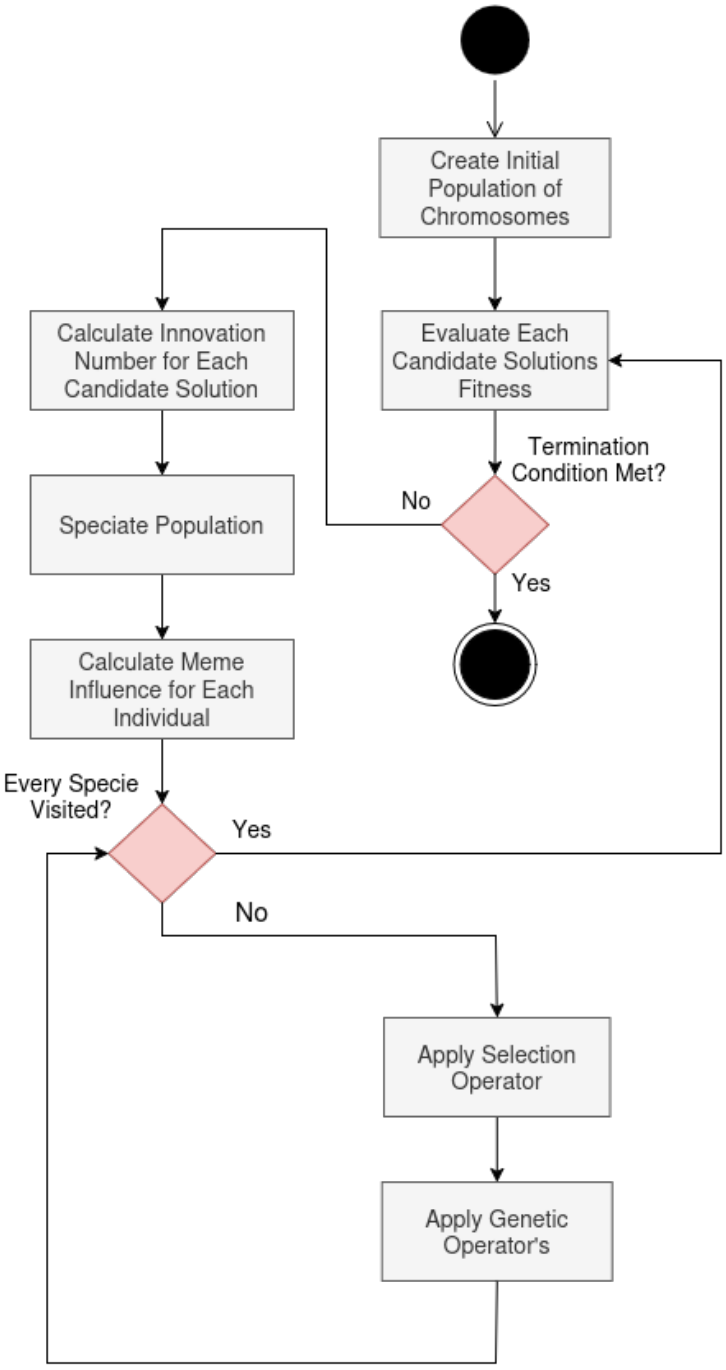


Figure 6.1: Evolutionary Lifecycle of GEP-NEAT

At a high level, both NEAT and GEP follow a similar evolutionary cycle, that is, intialising a population, representing chromosomes, evaluating fitness, applying genetic operators,

and iterating until a termination condition is met. The two approaches however diverge significantly in their internal mechanisms and innovations. NEAT introduces a sophisticated speciation mechanism, where historical gene information is tracked using innovation numbers. These are stored in a lookup table and used to calculate compatibility distance between individuals, allowing the population to be divided into species that evolve in parallel within their own niches (Stanley and Miikkulainen 2002).

In contrast, GEP adheres more closely to the traditional genetic algorithm framework but distinguishes itself through a rich set of genetic operators that promote diversity and facilitate the emergence of reusable substructures, or building blocks, within chromosomes (Ferreira 2006). Its use of Karva notation ensures syntactically valid expressions, making it particularly well-suited for evolving complex, non-linear representations.

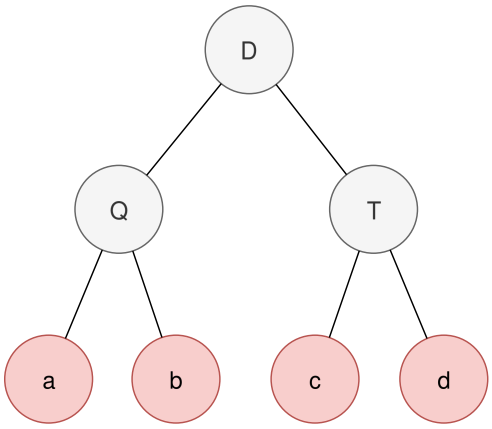
GEP-NEAT draws inspiration from both of these methodologies to form a unified framework, as illustrated in Figure 6.1. It leverages NEAT's innovation tracking and speciation to maintain diversity and protect novel structures, while adopting GEP's flexible chromosome representation to enable the evolution of expressive and structurally diverse neural networks.

6.2.2 Representation

NEAT-GEP draws inspiration from the genotype-to-phenotype mapping introduced in Gene Expression Programming (GEP), particularly through the use of Karva notation, however, several modifications have been made to enhance the representation, starting with the adoption of prefix tree-style encoding. In traditional GEP, k-expressions are converted into expression trees by reading the linear string from left to right and top to bottom. While this method ensures syntactic correctness, it can become computationally expensive, especially when evaluating large populations or complex expressions since the expression tree must first be constructed from the expression string during each fitness evaluation.

As discussed in Chapter 5, alternative encoding schemes such as prefix and postfix notation have been explored to address this limitation. On such approach, P-GEP, encodes linear chromosomes in prefix form. This representation improves the preservation of meaningful substructures during operations like crossover and mutation, which in turn supports more efficient and targeted exploration of the search space.

Figure 6.2 illustrates the differences between standard Karva notation and prefix notation, highlighting how subtrees - shown in purple, blue, and green - are affected by genetic operations. In standard Karva notation, these substructures are often disrupted during crossover, whereas prefix notation allows them to be preserved. For example, when a single-point crossover is applied between gene 3 and 4, the prefix-encoded chromosome maintains the integrity of its subtrees, whereas the same operation on a Karva-encoded chromosome may fragment them.



	0	1	2	3	4	5	6
GEP Notation Encoding	D	Q	T	a	b	c	d
Prefix GEP Notation Encoding	D	Q	a	b	T	c	d

Figure 6.2: GEP Encoding Scheme versus P-GEP Encoding Scheme

To illustrate how a GEP-NEAT chromosome is structured and functions, this chapter introduces a Simplified example. Consider a linear chromosome with a head size of $h = 3$ and a maximum function arity of 3. Let the function set be D, T, where function D takes two inputs and T takes three, and the terminal set be a, b, c. According to GEP's formulation, the tail size is calculated using the formula:

$$t = h \times (n_{max} - 1) + 1 = 3 \times (3 - 1) + 1 = 7 \tag{6.1}$$

This results in a genome length of $h + t = 3 + 7 = 10$. In the original GEP formulation, both weight and threshold domains were encoded within the linear string. While threshold domains are useful in networks with binary or step activation functions, they can be restrictive in more expressive architectures. Therefore, in GEP-NEAT, only a weight domain is encoded, allowing for greater flexibility in network behaviour. The weight domain length is defined as:

$$D_w = h \times n_{max} = 3 \times 3 = 9 \tag{6.2}$$

A lookup weight array is used, where each gene in the weight domain serves as an index into this array. For example, a weight array of size 5 might be defined as:

$$[-1.2, 2.3, 3.2, 4.4, -0.5, 6.0] \tag{6.3}$$

A sample chromosome constructed using this configuration looks as follows:

$$\begin{array}{cccccccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ D & a & T & a & b & c & a & b & a & 5 & 2 & 4 & 1 & 2 & 3 & 3 & 2 & 6 \end{array} \tag{6.4}$$

Based on the logic of GEP, to reiterate, genes 0 to 5 represent the actual expression tree, while genes 6 to 8 serve as buffer symbols that ensure syntactic correctness but do no contribute to the functional output. One notable limitation of the original GEP-NN implementation is the absence of bias nodes. While this omission had minimal impact in the original research likely due to the relatively constrained problems it had been applied to, its effect becomes more pronounced in larger and more complex solution spaces. In

such cases, the lack of biasing can hinder the network's ability to shift activation thresholds, potentially leading to suboptimal performance.

To address this shortcoming, the first proposed novelty in GEP-NEAT is the explicit incorporation of bias into the chromosome structure. This is achieved by modifying the expression tree representation such that each function symbol encapsulates two components:

- **bias enabled**: This stores a boolean value to determine whether a bias is enabled on the function symbol.
- **bias weight**: This stores the bias weight.

Continuing with the illustrative example, let us now consider the bias values associated with the functional symbols in the expression string presented in Equation 6.4:

- Function Symbol D
  - **bias enabled**: true
  - **bias weight**: 0.4
- Function Symbol T
  - **bias enabled**: false
  - **bias weight**: 2.3

This candidate solution corresponds to the expression tree, and by extension, the neural network depicted in Figure 6.3. Similar to the forward propagation process in conventional neural networks, input values are assigned to the terminal a, b, and c, after which the expression tree is evaluated to compute the final network output.

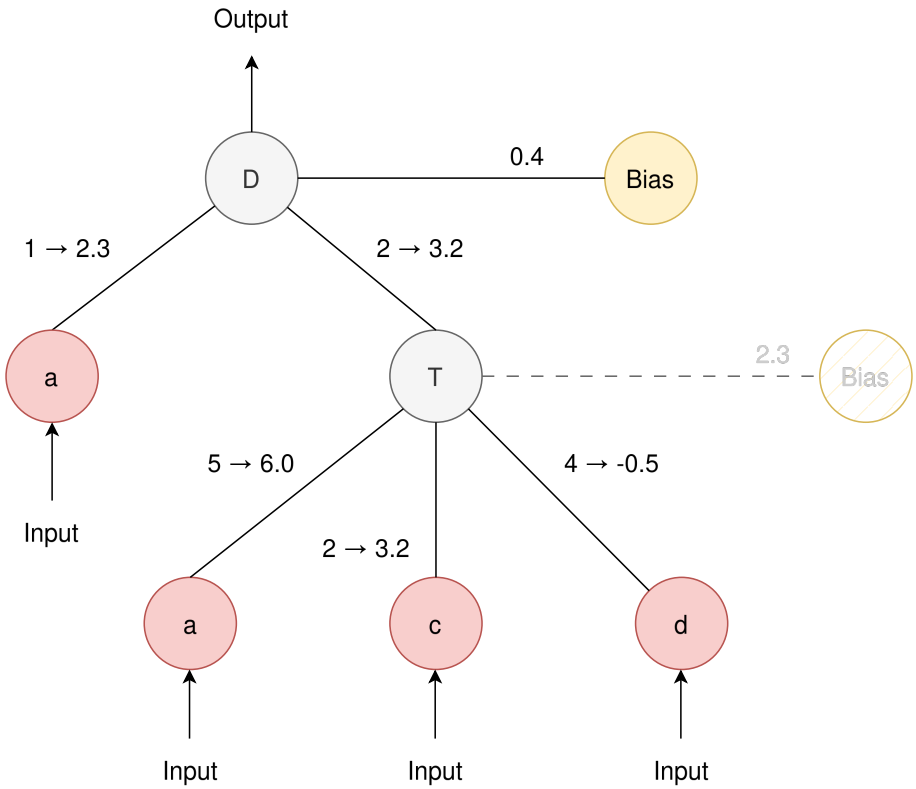


Figure 6.3: Example GEP-NEAT Candidate Solution Expression Tree of Linear String in Equation 6.4

As illustrated in Figure 6.3, the expression tree is decoded from the linear chromosome using prefix notation. Although a more detailed explanation is provided later in the chapter, it is import to note that weight values from the weight domain are assigned in a depth-first manner, progressing from leaves of the tree upward, similar to the evaluation strategy used in symbolic regression. Each weight value corresponds to an index in the predefined weight lookup array. Additionally, as previously mentioned, each function symbol in the tree is associated with a bias node. In this example the function symbol D has its bias enabled, while T does not, demonstrating how bias can be selectively applied within the network structure.

Another limitation of the original GEP-NN algorithm lies in its restricted use of activation

functions. As previously mentioned, the original implementation employed a threshold domain for each function symbol, where a neuron would output 0 if its weighted sum of its inputs was below the associated threshold, and 1 otherwise, effectively implementing a binary step function. Additionally, the only weighted sum operations used were linear in nature, represented by symbols such as D, T, and Q, with arities, 2, 3, and 4 respectively.

While this approach was sufficient for the original problem domains, it proved inadequate for more complex tasks such as symbolic regression and classification. Subsequent research demonstrated that by modifying these function symbols to represent arithmetic operations like subtraction, multiplication, and division, the algorithm's expressiveness could be significantly improved (W. Wang, Q. Li, and Qi 2008). This however still did not address the broader limitation of lacking non-linear activation functions, which are a fundamental component of modern neural networks.

To overcome this, the next proposed novelty in GEP-NEAT framework is the introduction of configurable activation functions. At each function node, the weighted sum - computed as the sum of all inputs plus the bias - is passed through an activation function specific to that symbol. This allows each function symbol to encode not only its arity and operation but also its own activation behaviour. The function set is fully configurable, enabling the inclusion of a diverse range of activation functions (e.g., sigmoid, tanh, ReLU), each associated with different arities. This design mirrors the flexibility of traditional neural networks, allowing GEP-NEAT to learn which activation functions are most effective for a given problem domain.

6.2.3 Initial Population

In the original GEP algorithm, Candida mentions that in order to speed up the evolutionary process of finding a suitable candidate, a start-protocol is employed which is that the an intial population is generated until a viable solution is find such that the fitness is not

minimum. The purpose of this from GEP's persepctive is that every offspring is part of a lineage from a sole founder as a means to accelerate the exploitation vs exploration scheme Ferreira 2006. The obvious implication of this is that there runs a risk of candidate solutions stagnating around local optima.

In the case of NEAT, the concept of starting out minimally is introduced. NEAT does this by generating a population such that there are no hidden nodes forcing the resulting structures to be as minimal as possible. In doing so, the idea is that structural innovations are intended which means that solutions in the lowest dimentional space is searched for which leads to performance gains (Stanley and Miikkulainen 2002). Mentioned in the original paper of NEAT, typical algorithms don't do this because they run the risk of topological innovations being prematurely killed. NEAT solves this by protecting structural innovations through historical gene tracking by means of innovation numbers.

GEP-NEAT does not inherit the start-up control mechanism that GEP makes use of in order to mitigate the chance that candiate solutions may prematurley stangate around local optima. Instead, GEP-NEAT adapts the *starting out minimally* dependency compoennt NEAT mentions by generating individuals that have only one function symbol which is in the root of the linear chromosome during the intialisation phase. An example of this, is the linear string shown in Equation 6.5, and its phentotype representation shown in Figure 6.4.

$$\begin{array}{cccccccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ D & a & b & a & b & c & a & b & a & 5 & 2 & 4 & 1 & 2 & 3 & 3 & 2 & 6 \end{array} \tag{6.5}$$

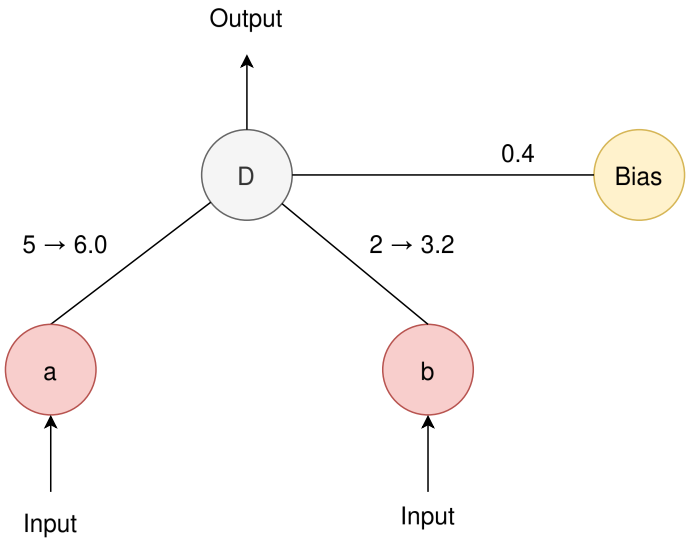


Figure 6.4: Example GEP-NEAT Candidate Solution Expression Tree of Linear String in Equation 6.4 Generated Through Initialisation Phase

In doing so, the initial population generated all have genotypes that represent networks with only a single input and output layer (no hidden layer). This aligns with the reasoning NEAT imposes whereby structural innovations are intended and the simplest solution space is explored first.

6.2.4 Evaluating Candidate Solutions

Section 6.2.2 details how candidate solutions represent linear strings in their genotype form and translate using prefix notation into the respective expression tree (neural networks). Converting between genotype and phenotype for complex lienar strings and iteratively over many generations can incur expensive computational efforts.

S_GEP, an extension of P-GEP has the added benefit of evaluating the expression string without the need to convert it to an expression tree. Although there is no research on its use in GEP-NN, the idea to encorate it in GEP-NEAT is that forward feeding values in an expression tree already involves computationally expensive operations, so this would greatly benefit the calculations of fitness when doing this repeatedly throughout many

candidate solution samples over many generations. Although S_GEP was premis was covered in Chapter 5.5, with the added concept of proper activation functions and biases, the algorithm is changed slightly. Firstly, in order to evaluate the expression tree, values are replaced with the terminal values so that mathematic operation can commence. For example, if the network is being evolved to solve the XOR problem, then their would be 4 test values that would need to be fed into the network, that being, 00, 01, 10, and 11. For each case, the inputs would replace terminal symbols, so for example, a would be the first digit and b the second. The next step would be that the effective gene length is required in order to know what part of the linear string is the coding region. This is calculated using Algorithm 7.

Algorithm 7 Effective gene length (adapted from YuZhong Peng et al. 2014)

1: **Input**: String of gene symbols
2: **Output**: eL
3: Initialize variables $Len = 1$ and $count = 0$
4: **while** scanning gene symbols starting from left to right:
5: $Len = Len + 1$;
6: **if** current gene symbol is function:
7: $count = count - 1$
8: **else**:
9: $count = count - 1$
10: **if** $count < 0$:
11: return $eL = Len$ as output

Once the effective gene length is known, the linear string can be evaluated using the concept of polish notation as the encoding has been done in prefix notation. The linear string is evaluated using Algorithm 8.

Once the linear string is evaluated, it is then fed into a fitness function to calculate a fitness value. Fitness functions are problem specific, for example, in both NEAT and GEP, the

Algorithm 8 GEP-NEAT Evaluation Algorithm (inspired from YuZhong Peng et al. 2014)

```
Calculate eL using Algorithm 7
set stack array variable to an empty list
set weight_index variable to 0
while reading effective gene from right to left using eL:
    if current gene is a terminal symbol:
        push terminal symbol on stack
    else if current gene is function symbol:
        set weighted_sum variable to 0
        for number of function symbol inputs:
            pop value from stack and multiply with weight from weight lookup array at index weight_index
            add result value to weighted_sum
            increment weight_index by 1
        if bias is enabled for current function symbol:
            add function symbol's bias value to weighted_sum
        apply function symbol's activation function to weighted_sum
        push result value onto stack
pull value from stack and return
```

XOR uses the folloiwng fitness function:

$$f_i = \sum_{j=1}^{C_t} (M - |C_{(i,j)} - T_{(j)}| \tag{6.6}$$

where:

- f_i is fitness for candidate solution i
- C_t is the number of test cases
- M is the maximum value for a fitness case
- $C_{(i,j)}$ is the value returned by individual i for case j
- $T_{(j)}$ is the target value for case j

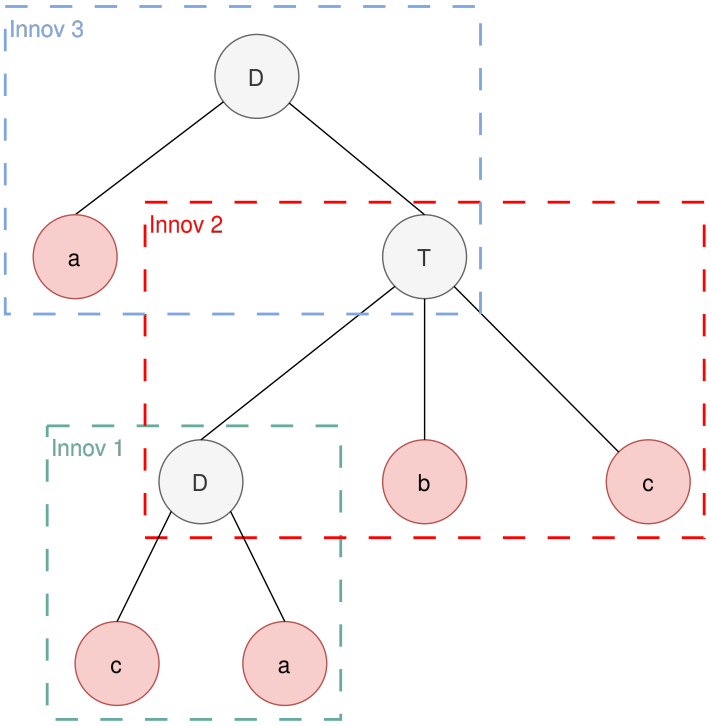
Other problems such as the cartpole problem have fitness functions based on a reward scheme whereby for every episode thta the cartpole balances the pole, it receives an abritary incentive reward value and otherwise, a zero value.

6.2.5 Speciation

The next step of the algorithm is to do two things, that is, calculate innovation numbers and speciate. NEAT uses innovation numbers to track the historical lineage of genes to in order to preserve innovation, in particular, innovative structures. NEAT assigns innovation numbers by keeping track of a global table and whenever an new, unseen connection between two nodes appear, the global innovation number is incremented. Innovation numbers in this was as seen in Chapter 4.5 allows two chromosomes to be compared by calculating how the difference in number of innovation numbers they share and further allows speciation.

GEP-NEAT adopts the concept of innovation numbers by redefining what they represent. Innovation numbers are redefined to represent sub-tree information. In other words, each subtree represents a unique innovation number. Subtree can then reference reference other innovation numbers. To visually illustrate, refer to Figure 6.5. The tree is traversed in the same manner when evaluating the expression tree. As values are popped onto the stack,

each subtree is analysed and checked against a global innovation number lookup table. If the subtree exists in the lookup table, that innovation number is used, otherwise, the subtree is added to the lookup table as a new entry and incremented innovation number value. Algorithm 9 shows this mechanism.



Innovation Number Lookup Table

Innovation Number	Subtree (Prefix Notation)
1	D c a
2	T 1 b c
3	D a 2

Figure 6.5: Innovation Number Assignment on GEP-NEAT Expression Tree

Innovation numbers serves as a system to represent subtree configuration, which in turn can be used to represent entire expression trees. Every expression tree in this regard can be expressed using its *innovation number composition* which is a list comprising of innovation number that make up their tree. Each innovation number with that composition list stores the subtree that the innovation number represents with its respective weights. This would mean that this composition list is keyed on the innovation number with its value being the a tuple representing its subtree and weight respective weights. The *innovation number composition* is calculated using Algorithm 10. For example, the *innovation number composition* for the expression tree seen in Figure 6.5 has the corresponding *innovation number composition* seen in Figure 6.6.

Algorithm 9 GEP-NEAT Innovation Number Algorithm

```
1: Calculate eL using Algorithm 7
2: set stack array variable to an empty list
3: while reading effective gene from right to left using eL:
4:     if current gene is a terminal symbol:
5:         push terminal symbol on stack
6:     else if current gene is function symbol:
7:         push current function symbol to stack
8:         access index [function symbol's arity plus 1] from stack to get subtree
9:         set innovation_number variable to 0
10:        if innovation number does not exist in lookup table:
11:            set innovation_number to length of lookup table minus 1
12:        else:
13:            set innovation_number variable to found value in lookup table
14:        push innovation_number to stack
```

Algorithm 10 GEP-NEAT Innovation Number Composition Algorithm

```
1: Calculate eL using Algorithm 7
2: set weight_index variable to 0
3: set stack array variable to an empty list
4: while reading effective gene from right to left using eL:
5:     if current gene is a terminal symbol:
6:         push terminal symbol on stack
7:     else if current gene is function symbol:
8:         set weights array variable to empty list
9:         for number of function symbol inputs:
10:            retrieve value from weight lookup array indexed at current input and add to
    weight array
11:            increment weight_index by 1
12:        append current symbol to stack
13:        access index [function symbol's arity plus 1] from stack to get subtree
14:        get innovation number using subtree from lookup table
15:        append innovation number with its corresponding subtree and weights to the
    innovation_number_composition
16:        push result value onto stack
17: return innovation_number_composition
```

	Expression Tree	D a T D c a b c 1 2 3 4 5 6 7		
Innovation Number Composition	Innovation Number	1	2	3
	Subtree	D c a	T 1 b c	D a 2
	Weights	(1, *2)*	*(3, 4, 5)*	(6, 7)

Figure 6.6: Innovation Number Composition Expression Tree in Figure 6.5

Similarly to how NEAT can represent a candidate solution entirely in terms of its innovation numbers, so can GEP-NEAT. GEP-NEAT adopts the same speciation mechanism as NEAT, that is, using a compatibility distance function. Inspired from NEAT, GEP-NEAT introduces the following compatibility distance function:

$$\delta = \frac{c_1 NM}{N} + c_2 \cdot \overline{W} \tag{6.7}$$

where:

- δ is the compatibility distance between a pair of genomes
- NM is the number of non-matching genes
- $\overline{W}$ is the average weight difference of matching gene arrays
- N is the number of genes in the larger genome (serving as a normalization)
- c_1 and c_2 serve as adjustments in the importance of the two factors

In the original NEAT paper, although disjoint and excess genes were defined distinctively as two different components, they had no unique implication as whole. For this reason, GEP-NEAT recognizes matching and non-matching genes. On this note, the average weight works slightly differently. Two genes representing the same subtree configuration can have different weights assosiated between the same connections, hence, in order to calculate the average weight difference:

$$\overline{W} = \frac{\sum_{j=0}^{N} |W_{aj} - W_{bj}|}{N} \tag{6.8}$$

where:

- $\overline{W}$ is the average weight difference of the subtree

- N is the number of child nodes in the subtree
- W_{aj} is the weight of subtree 'a' of child node j
- W_{bj} is the weight of subtree 'b' of child node j

The number of matching and non-matching genes can then be calculated easily by converting an expression tree to its *innovation number composition* counterpart, and analysing which genes are in common and which are not. Figure 6.7 shows that the matching genes between expression tree 1 and 2 are $\{2\}$, and the non-matching genes are $\{1,3,4,5\}$.

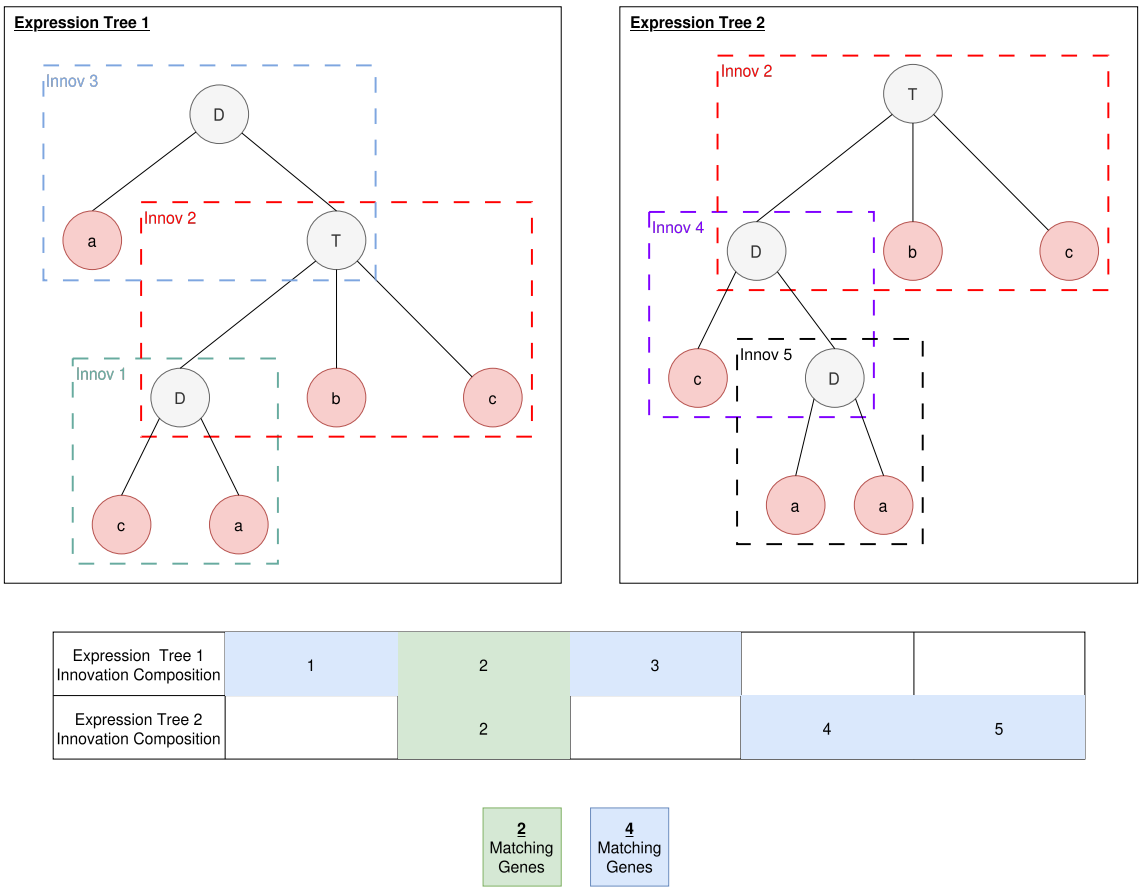


Figure 6.7: Comparing Expression Trees Using Innovation Number Compositions

Speciation occurs within every evolutionary generation by speciating individuals and follows algorithm 11. Essentially, during each generation, the population is shuffled to give every individual an opportunity to be the representative individual of a species (in an attempt to better group individuals in species group). Every individual is then evaluated against every specie's representative individual and provided its compatibility distance is

less than the compatibility threshold δ_{ct}, it will then belong to that species. This process continues until every individual is part of a specie group.

Algorithm 11 GEP-NEAT Speciation Algorithm

```
Initialise empty list of species
Shuffle the population
for every individual in population:
    if species is empty:
        add individual as first specie group
    else:
        for every specie in the species list:
            calculate compatibility distance between representative individual in current specie and current individual
            if compatibility distance is less than compatibility threshold δct:
                add individual to current specie
                break to next individual
```

6.2.6 Meme Influence

Richard Dawkins is a British evolutionary biologist and best known for his gene-centered view of evolutionary, most famously in his 1976 book *The Selfish Gene*. In this book, Dawkins argues that genes are the primary units of natural selection, driving evolution through replication and competition (Dawkins 1981). Dawkins introduces the term "**meme**" in the *The Selfish Gene* as a cultural analog to biological genes. A meme is a unit of cultural information such as an idea, behavior, or trend, that spreads from person to person via imitation. Importantly, memes are subject to variation, competition and selection, much like genes that drives evolution.

Seeing that subtree configurations are tracked by means of an innovation lookup table throughout the evolutionary cycle, GEP-NEAT takes advantage of this by introducing

a new concept called the *meme influence*. The *meme influence* and *innovation number contribution* are two additional attributes tracked within each innovation number and is used to rank sub-tree configurations. This attribute is calculated as shown in Algorithm 12. The algorithm works by first calculating what each innovation numbers contribution is with regard to the individuals adjusted fitness (to the specie it belongs to). This is done using Equation 6.9. The *innovation contribution* value is then added to the result stored in every innovation number that makes up the individual's innovation number contribution. In doing this, information is being built up to describe which innovation numbers have contributed to the highest fitness according to adjusted fitness within each specie. Equation 6.9 assumes that each innovation number contributes equally to the success of a candidate individual, however, by tracking and incrementing each innovation numbers contribution collectively by all individuals within the population, innovation numbers that generally form part of higher fit individuals will tend to have higher innovation contributions. After the innovation contribution for every innovation number has been updated, the next step is to normalize these innovation contribution values so that they can be represented in a way that they can be ranked. Each innovation contribution value is then normalised according to Equation 6.10. With the normalised value calculated, the meme influence value of individual i is calculated using Equation 6.11 where α is the decay rate. The meme influence is calculated using a shifting average moving function in order to encourage current trends for a period of time until it fades away (Haynes, Corns, and Venayagamoorthy 2012). The outcome of the meme influence calculation is to provide a mechanism in which innovation numbers can be ranked within every generation.

$$innovation\,contribution = \frac{individual\,ad\,justed\,fitness}{length(individual\,innovation\,number\,composition)} \tag{6.9}$$

$$normalised\,value_i = \frac{innovation\,contribution_i - min\,innovation\,contribution}{max\,innovation\,contribution - min\,innovation\,contribution} \tag{6.10}$$

$$meme\,influence_i = (\alpha \times normalised\,value) + ((1-\alpha) \times meme\,influence_i) \tag{6.11}$$

Algorithm 12 GEP-NEAT Meme Influence Algorithm

for specie in species:
    for individual in specie:
        calculate *innovation contribution* using Equation 6.9
        for each innovation number in the current individual's innovation number composition:
            innovation_lookup_table[innovation number][innovation contribution attribute] += *innovation contribution*

for each innovation number in innovation_lookup_table:
    calculate the normalised value for the current innovation number using Equation 6.10
    calculate innovation number contribution using Equation 6.9
    calculate the *meme influence* for the current individual using Equation 6.11
    append the *meme influence* value as attribute to the current innovation number

6.2.7 Selection

Both GEP and NEAT make use of well studied selection operators; NEAT uses tournament selection while GEP makes use of roulette wheel selection. The problem with this kind of selection mechanism is that the only information that the selection operator makes use of is fitness and has no strong relation to the innovation of structures that make up an individual. GEP-NEAT tracks a metric known as the *meme influence* as an additional attribute alongside each innovation number giving a way in which innovation numbers can be ranked. GEP-NEAT introduces a new selection operator called *Meme Influence Roulette Wheel Selection (MIRWS)* which is based on roulette wheel selection with the difference being that the *meme influence* values are used as opposed to raw fitness scores of individuals. The new adjusted fitness scores based on the *meme influence* values are calculated using Algorithm 13 and thereafter follows the generic roulette wheel based selection algorithm.

6.2.8 Genetic Operators

Once individuals are selected within every specie during every generation of the evolutionary process, selected individuals undergo an opportunity to undergo mutation and crossover to produce offspring. The following genetic operators are available to GEP-NEAT each with their own mutation/crossover probabilities. Mutation, the most important operator, is implemented in various domains. Since the genotype is largely designed on the premis of Karva notation, a large majority of the mutation operators are adapted from GEP with others based on the introduced concept of meme influence.

6.2.8.1 Neuron Mutation

This mutation mechanism with mutation rate ρ_{neuron}, mutates the neuron portion of the expression string based on n-point mutation where n random genes are chosen within the expression string. If the point of mutation resides on a function symbol, within the root,

Algorithm 13 GEP-NEAT Meme Influence Roulette Wheel Selection Algorithm

```
1: for each individual in the population:
2:     initialise total_individual_fitness variable to 0
3:     for each innovation number in the current individual's innovation number composition:
4:         total_individual_fitness += current innovation numbers meme influence value
5:     set the current individuals adjusted fitness to the total_individual_fitness
6:
7: intialise selected_individual array to empty list
8: for number of individuals required:
9:     choose a random number between 0 and the total adjusted fitness based on the meme influence value
10:    set current variable to 0
11:    for every individual in the population:
12:        current += current individuals adjusted fitness
13:        if current >= random number chosen:
14:            append current individual to selected_individual list
15: return selected_individual
```

the resulting mutated gene has to be a function symbol. If the mutation resides anywhere else within the head of the expression tree, the symbol can mutated into either a function or terminal symbol, however, if the mutation point resides within the tail, only a terminal symbol may be chosen.

6.2.8.2 Weight Mutation

This mutation mechanism with mutation rate ρ_{weight}, mutates the weight portion of the expression string based on n-point mutation where random n weight genes are chosen and then mutated such that values are randomly uniformaly chosen from the index size of the weight lookup array.

6.2.8.3 Weight Lookup Mutation

This mutation mechanism with mutation rate ρ_{weight_lookup}, mutates the weight lookup array based on n-point mutation where random n weight genes are chosen and then mutated according to a uniform distribution.

6.2.8.4 Bias Toggle Mutation

This mutation mechanism with mutation rate ρ_{bias_toggle}, mutates the *bias toggle* attribute of a randomly chosen function symbol within a chromosome by peforming essentially a bit flip based on true/false logic.

6.2.8.5 Bias Mutation

This mutation mechanism with mutation rate ρ_{bias}, mutates the *bias value* attribute of a randomly chosen function symbol within a chromosome by performing mutation based on uniform distrbution based on the bias range hyperparameter.

6.2.8.6 Mutation

hello

6.3 Prototype Implementation

Richard Dawkins,

6.4 Model Validation and Results

Hello

Bibliography

Aggarwal, Charu C et al. (2018). *Neural networks and deep learning*. Volume 10. 978. Springer (cited on page 70).

Ahn, Chang Wook (2006). *Advances in evolutionary algorithms*. Springer (cited on page 40).

Alhijawi, Bushra and Arafat Awajan (Feb. 2023). "Genetic algorithms: theory, genetic operators, solutions, and applications". In: *Evolutionary Intelligence* 17, pages 1–12. DOI: 10.1007/s12065-023-00822-6 (cited on page 57).

Alkafaween, Esra'a (2018). "Novel Methods for Enhancing the Performance of Genetic Algorithms". In: *CoRR* abs/1801.02827. arXiv: 1801.02827. URL: http://arxiv.org/abs/1801.02827 (cited on page 57).

Angeline, Peter J, Gregory M Saunders, and Jordan B Pollack (1994). "An evolutionary algorithm that constructs recurrent neural networks". In: *IEEE transactions on Neural Networks* 5.1, pages 54–65 (cited on page 61).

Asseman, Alexis, Nicolas Antoine, and Ahmet S Ozcan (2021). "Accelerating deep neuroevolution on distributed FPGAs for reinforcement learning problems". In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 17.2, pages 1–17 (cited on page 61).

Ayoub, Nasser et al. (2009). "Evolutionary algorithms approach for integrated bioenergy supply chains optimization". In: *Energy Conversion and Management* 50.12, pages 2944–2955 (cited on page 57).

Bäck, Thomas, Joost N Kok, and G Rozenberg (2012). *Handbook of natural computing*. Springer, Heidelberg (cited on pages 44, 46–53).

Bautu, Elena, Andrei Bautu, and Henri Luchian (2007). "Adagep-an adaptive gene expression programming algorithm". In: *Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007)*. IEEE, pages 403–406 (cited on page 111).

Bayat, Parichehr et al. (2022). "Blasting pattern optimization using gene expression programming and grasshopper optimization algorithm to minimise blast-induced ground vibrations". In: *Engineering with Computers*, pages 1–10 (cited on page 95).

Bell, Okezue (2022). *Applications of Gaussian Mutation for Self Adaptation in Evolutionary Genetic Algorithms*. arXiv: 2201.00285 [cs.NE]. URL: https://arxiv.org/abs/2201.00285 (cited on page 40).

Bishop, Chris M (1994). "Neural networks and their applications". In: *Review of scientific instruments* 65.6, pages 1803–1832 (cited on page 63).

Blickle, Tobias and Lothar Thiele (1996). "A comparison of selection schemes used in evolutionary algorithms". In: *Evolutionary Computation* 4.4, pages 361–394 (cited on page 46).

Camburn, Bradley et al. (2017). "Design prototyping methods: state of the art in strategies, techniques, and guidelines". In: *Design Science* 3, e13 (cited on page 26).

Cilliers, Michael (2022). "Maintaining population diversity in evolutionary algorithms via epigenetics and speciation". eng. PhD thesis. University of Johannesburg (cited on page 44).

Cook, Blake et al. (Mar. 2021). *Neural Field Models: historical perspectives and recent advances*. DOI: 10.48550/arXiv.2103.10554 (cited on page 62).

Cullinan, Michéle, Duncan Coulter, and Jacques van Appel (2023). "CLISDE: An Agent-Oriented Cladistic Island Genetic Algorithm". In: *2023 10th International Conference on Soft Computing & Machine Intelligence (ISCMI)*. IEEE, pages 27–33 (cited on page 47).

D'Ambrosio, David B, Jason Gauci, and Kenneth O Stanley (2014). "HyperNEAT: The first five years". In: *Growing Adaptive Machines: Combining Development and Learning in Artificial Neural Networks*, pages 159–185 (cited on page 90).

Das, Swagatam and Ponnuthurai Nagaratnam Suganthan (2010). "Differential evolution: A survey of the state-of-the-art". In: *IEEE transactions on evolutionary computation* 15.1, pages 4–31 (cited on pages 32, 55).

Dawkins, Richard (1981). "Selfish genes and selfish memes". In: *The mind's I: Fantasies and reflections on self and soul*, pages 124–144 (cited on page 136).

Deng, Yong, Yang Liu, and Deyun Zhou (Oct. 2015). "An Improved Genetic Algorithm with Initial Population Strategy for Symmetric TSP". In: *Mathematical Problems in Engineering* 2015, pages 1–6. DOI: 10.1155/2015/212794 (cited on page 57).

Diaz-Gomez, Pedro A and Dean F Hougen (2007). "Initial population for genetic algorithms: A metric approach." In: *Gem*, pages 43–49 (cited on page 45).

Dulebenets, Maxim A et al. (2018). "A self-adaptive evolutionary algorithm for the berth scheduling problem: Towards efficient parameter control". In: *Algorithms* 11.7, page 100 (cited on page 56).

Eiben, A. E. and James E. Smith (2015). *Introduction to Evolutionary Computing*. 2nd. Springer Publishing Company, Incorporated. ISBN: 3662448734 (cited on pages 29, 36, 37, 42, 48, 51–53).

Elsken, Thomas, Jan Hendrik Metzen, and Frank Hutter (2019). "Neural architecture search: A survey". In: *Journal of Machine Learning Research* 20.55, pages 1–21 (cited on pages 73–76).

Engelbrecht, Andries P (2007). *Computational intelligence: an introduction*. John Wiley & Sons (cited on page 13).

Ferreira, Cândida (2006). *Gene expression programming: mathematical modeling by an artificial intelligence*. Volume 21. Springer (cited on pages 14, 93–109, 120, 126).

Gen, Mitsuo and Lin Lin (2023). "Genetic Algorithms and Their Applications". In: *Springer Handbook of Engineering Statistics*. Edited by Hoang Pham. London: Springer London, pages 635–674. ISBN: 978-1-4471-7503-2. DOI: 10.1007/978-1-4471-7503-2_33. URL: https://doi.org/10.1007/978-1-4471-7503-2_33 (cited on page 58).

Gregar, Jan (2023). "Research design (qualitative, quantitative and mixed methods approaches)". In: *Research Design* 8 (cited on page 27).

Gurney, Kevin (2018). *An introduction to neural networks*. CRC press (cited on pages 62, 63).

Guzek, Mateusz et al. (2014). "Multi-objective evolutionary algorithms for energy-aware scheduling on distributed computing systems". In: *Applied Soft Computing* 24, pages 432–446 (cited on page 57).

Haynes, David, Steven Corns, and Ganesh Kumar Venayagamoorthy (2012). "An exponential moving average algorithm". In: *2012 IEEE Congress on Evolutionary Computation*. IEEE, pages 1–8 (cited on page 137).

Hevner, Alan R et al. (2004). "Design science in information systems research". In: *MIS quarterly*, pages 75–105 (cited on pages 21, 22).

Hinterding, Robert (1995). "Gaussian mutation and self-adaption for numeric genetic algorithms". In: *Proceedings of 1995 IEEE International Conference on Evolutionary Computation* 1, pages 384–. URL: https://api.semanticscholar.org/CorpusID:30756203 (cited on page 40).

Igel, Christian (2003). "Neuroevolution for reinforcement learning using evolution strategies". In: *The 2003 Congress on Evolutionary Computation, 2003. CEC'03.* Volume 4. IEEE, pages 2588–2595 (cited on page 61).

Koc, Hatice et al. (2021). "UML diagrams in software engineering research: a systematic literature review". In: *Proceedings*. Volume 74. 1. MDPI, page 13 (cited on page 25).

Kowaliw, Taras, Nicholas Bredeche, and René Doursat (2014). *Growing Adaptive Machines*. Springer (cited on page 91).

Koza, John R (1994). "Genetic programming as a means for programming computers by natural selection". In: *Statistics and computing* 4, pages 87–112 (cited on pages 32, 51, 53).

Kramer, Oliver (Jan. 2017). *Genetic Algorithm Essentials*. ISBN: 978-3-319-52155-8. DOI: 10.1007/978-3-319-52156-5 (cited on pages 33–37).

Kusy, Maciej, Bogdan Obrzut, and Jacek Kluska (2013). "Application of gene expression programming and neural networks to predict adverse events of radical hysterectomy in cervical cancer patients". In: *Medical & biological engineering & computing* 51, pages 1357–1365 (cited on page 95).

Larranaga, Pedro et al. (Jan. 1999). "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators". In: *Artificial Intelligence Review* 13, pages 129–170. DOI: 10.1023/A:1006529012972 (cited on page 39).

Li, Changhe, Changhe/Han Li (Shoufe), and Sanyou Zeng (2024). *Intelligent Optimization*. Springer (cited on pages 34, 38, 40, 41, 48, 50, 56).

Li, Changhe, Shoufei Han, et al. (2024). "Basics of Evolutionary Computation Algorithms". In: *Intelligent Optimization: Principles, Algorithms and Applications*. Singapore: Springer Nature Singapore, pages 63–92. ISBN: 978-981-97-3286-9. DOI: 10.1007/978-981-97-3286-9_4. URL: https://doi.org/10.1007/978-981-97-3286-9_4 (cited on pages 13, 29).

Li, Xin et al. (2005). "Prefix gene expression programming". In: *Late Breaking Paper at the Genetic and Evolutionary Computation Conference (GECCO), Washington, DC* (cited on pages 112–114).

Links, Alex Fraser and In MemoriamObituaryRemembranceMemory BookDust (2002). "Alex S. Fraser". In: *IEEE Transactions on Evolutionary Computation* 6.5 (cited on page 31).

Liu, Yuqiao et al. (2021). “A survey on evolutionary neural architecture search”. In: *IEEE transactions on neural networks and learning systems* 34.2, pages 550–570 (cited on pages 74–76).

Luke, Sean and Liviu Panait (2002). “Lexicographic parsimony pressure”. In: *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 829–836 (cited on page 53).

Maaranen, Heikki, Kaisa Miettinen, and Antti Penttinen (2007). “On initial populations of a genetic algorithm for continuous optimization problems”. In: *Journal of Global Optimization* 37, pages 405–436 (cited on page 35).

Malik, Hasmat and Sukumar Mishra (2016). “Application of gene expression programming (GEP) in power transformers fault diagnosis using DGA”. In: *IEEE Transactions on Industry Applications* 52.6, pages 4556–4565 (cited on page 95).

Malik, S and S Wadhwa (2014). “Preventing premature convergence in genetic algorithm using DGCA and elitist technique”. In: *International Journal of Advanced Research in Computer Science and Software Engineering* 4.6 (cited on page 44).

Mazari, Mehran and Daniel D Rodriguez (2016). “Prediction of pavement roughness using a hybrid gene expression programming-neural network technique”. In: *Journal of Traffic and Transportation Engineering (English Edition)* 3.5, pages 448–455 (cited on page 95).

Meyer-Nieberg, Silja and Hans-Georg Beyer (2007). “Self-adaptation in evolutionary algorithms”. In: *Parameter setting in evolutionary algorithms*. Springer, pages 47–75 (cited on page 41).

Mirjalili, Seyedali (2018). *Evolutionary Algorithms and Neural Networks: Theory and Applications*. 1st. Springer Publishing Company, Incorporated. ISBN: 3319930249 (cited on pages 30, 40).

Mitchell, Melanie (1998). *An introduction to genetic algorithms*. MIT press (cited on pages 44, 47).

Mwaura, J and E Keedwell (2009). "Adaptive gene expression programming using a simple feedback heuristic". In: *Proceedings of the AISB, Edinburgh, UK* (cited on page 111).

Neri, Ferrante and Carlos Cotta (2012). "Memetic algorithms and memetic computing optimization: A literature review". In: *Swarm and Evolutionary Computation* 2, pages 1–14 (cited on page 41).

Nielsen, A (2015). *Neural networks and deep learning* (cited on pages 64–66, 71–73).

Nikitin, Nikolay O et al. (2022). "Automated evolutionary approach for the design of composite machine learning pipelines". In: *Future Generation Computer Systems* 127, pages 109–125 (cited on page 57).

Nunamaker Jr, Jay F, Minder Chen, and Titus DM Purdin (1990). "Systems development in information systems research". In: *Journal of management information systems* 7.3, pages 89–106 (cited on page 26).

O'Neill, Michael (2009). *Riccardo Poli, William B. Langdon, Nicholas F. McPhee: A Field Guide to Genetic Programming: Lulu. com, 2008, 250 pp, ISBN 978-1-4092-0073-4* (cited on page 51).

Olivier, Martin S (2009). *Information technology research: A practical guide for computer science and informatics*. Van Schaik (cited on pages 24–27).

Peng, Yiming et al. (2018). "NEAT for large-scale reinforcement learning through evolutionary feature learning and policy gradient search". In: *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 490–497 (cited on page 88).

Peng, YuZhong et al. (2014). "An improved gene expression programming approach for symbolic regression problems". In: *Neurocomputing* 137, pages 293–301 (cited on pages 114–116, 128, 129).

Petrowski, Alain and Sana Ben Hamida (Jan. 2016). *Evolutionary Algorithms* (cited on page 31).

Pham, Hieu et al. (2018). “Efficient neural architecture search via parameters sharing”. In: *International conference on machine learning*. PMLR, pages 4095–4104 (cited on page 76).

Rachmawati, Lily and Dipti Srinivasan (2009). “Multiobjective evolutionary algorithm with controllable focus on the knees of the Pareto front”. In: *IEEE Transactions on Evolutionary Computation* 13.4, pages 810–824 (cited on page 37).

Rafajłowicz, Wojciech (2018). “Horizontal Gene Transfer as a Method of Increasing Variability in Genetic Algorithms”. In: *Artificial Intelligence and Soft Computing*. Edited by Leszek Rutkowski et al. Cham: Springer International Publishing, pages 505–513. ISBN: 978-3-319-91253-0 (cited on page 40).

Ren, Pengzhen et al. (2021). “A comprehensive survey of neural architecture search: Challenges and solutions”. In: *ACM Computing Surveys (CSUR)* 54.4, pages 1–34 (cited on page 73).

Risi, Sebastian and Julian Togelius (2015). “Neuroevolution in games: State of the art and open challenges”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 9.1, pages 25–41 (cited on pages 59, 60).

Rothlauf, Franz (2009). “Representations for evolutionary algorithms”. In: *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 3131–3156 (cited on page 33).

Russell, Stuart J and Peter Norvig (2016). *Artificial intelligence: a modern approach*. Pearson (cited on pages 14, 63).

Sabar, Nasser R et al. (2019). “A self-adaptive evolutionary algorithm for dynamic vehicle routing problems with traffic congestion”. In: *Swarm and evolutionary computation* 44, pages 1018–1027 (cited on page 56).

Satman, Mehmet Hakan and Emre Akadal (2020). “Machine coded compact genetic algorithms for real parameter optimization problems”. In: *Alphanumeric Journal* 8.1, pages 43–58 (cited on page 50).

Segura, Carlos et al. (2016). "The importance of proper diversity management in evolutionary algorithms for combinatorial optimization". In: *NEO 2015: Results of the Numerical and Evolutionary Optimization Workshop NEO 2015 held at September 23-25 2015 in Tijuana, Mexico*. Springer, pages 121–148 (cited on pages 45, 46).

Sharma, Sagar, Simone Sharma, and Anidhya Athaiya (2017). "Activation functions in neural networks". In: *Towards Data Sci* 6.12, pages 310–316 (cited on pages 67–70).

Shen, Zuowei, Haizhao Yang, and Shijun Zhang (2021). "Neural network approximation: Three hidden layers are enough". In: *Neural Networks* 141, pages 160–173 (cited on page 66).

Shukla, Anupriya, Hari Mohan Pandey, and Deepti Mehrotra (2015). "Comparative review of selection techniques in genetic algorithm". In: *2015 international conference on futuristic trends on computational analysis and knowledge management (ABLAZE)*. IEEE, pages 515–519 (cited on pages 43, 44).

Slowik, Adam and Halina Kwasnicka (2020). "Evolutionary algorithms and their applications to engineering problems". In: *Neural Computing and Applications* 32, pages 12363–12379 (cited on page 30).

Stanley, Kenneth O, Jeff Clune, et al. (2019). "Designing neural networks through neuroevolution". In: *Nature Machine Intelligence* 1.1, pages 24–35 (cited on pages 59, 76).

Stanley, Kenneth O, David B D'Ambrosio, and Jason Gauci (2009). "A hypercube-based encoding for evolving large-scale neural networks". In: *Artificial life* 15.2, pages 185–212 (cited on pages 61, 89–91).

Stanley, Kenneth O and Risto Miikkulainen (2002). "Evolving neural networks through augmenting topologies". In: *Evolutionary computation* 10.2, pages 99–127 (cited on pages 14, 60, 61, 77–88, 120, 126).

Storn, Rainer and Kenneth Price (1995). "Differential evolution-a simple and efficient adaptive scheme for global optimization over continuous spaces". In: *International computer science institute* (cited on page 54).

Suzuki, Kenji (2011). *Artificial neural networks: methodological advances and biomedical applications*. BoD–Books on Demand (cited on pages 63, 65, 67).

Tharwat, Alaa and Wolfram Schenck (2021). "Population initialization techniques for evolutionary algorithms for single-objective constrained optimization problems: Deterministic vs. stochastic techniques". In: *Swarm and Evolutionary Computation* 67, page 100952. ISSN: 2210-6502. DOI: https://doi.org/10.1016/j.swevo.2021.100952. URL: https://www.sciencedirect.com/science/article/pii/S2210650221001140 (cited on page 35).

Victer Paul, P. et al. (2014). "A new population seeding technique for permutation-coded Genetic Algorithm: Service transfer approach". In: *Journal of Computational Science* 5.2. Empowering Science through Computing + BioInspired Computing, pages 277–297. ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2013.05.009. URL: https://www.sciencedirect.com/science/article/pii/S1877750313000744 (cited on page 56).

Wang, Shengjie, Tianyi Zhou, and Jeff Bilmes (2019). "Bias also matters: Bias attribution for deep neural network explanation". In: *International Conference on Machine Learning*. PMLR, pages 6659–6667 (cited on page 64).

Wang, Weihong, Qu Li, and Xing Qi (2008). "Gene Expression Programming Neural Network for Regression and Classification". In: *Advances in Computation and Intelligence*. Edited by Lishan Kang et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pages 212–219. ISBN: 978-3-540-92137-0 (cited on page 125).

Wirsansky, Eyal (2024). *Hands-On Genetic Algorithms with Python*. Sciendo. DOI: 10.0000/9781805121572. URL: https://doi.org/10.0000/9781805121572 (cited on pages 32, 37, 42).

Xue, Feng, Arthur C Sanderson, and Robert J Graves (2003). "Pareto-based multi-objective differential evolution". In: *The 2003 Congress on Evolutionary Computation, 2003. CEC'03.* Volume 2. IEEE, pages 862–869 (cited on page 55).

Young, Steven R et al. (2015). "Optimizing deep learning hyper-parameters through an evolutionary algorithm". In: *Proceedings of the workshop on machine learning in high-performance computing environments*, pages 1–5 (cited on page 57).

Yu, X. and M. Gen (2010). *Introduction to Evolutionary Algorithms*. Decision Engineering. Springer London. ISBN: 9781849961295. URL: https://books.google.co.za/books?id=rHQf_2Dx2ucC (cited on page 34).

Zainuddin, Farah Ayiesya, Md Fahmi Abd Samad, and Durian Tunggal (2020). "A review of crossover methods and problem representation of genetic algorithm in recent engineering applications". In: *International Journal of Advanced Science and Technology* 29.6s, pages 759–769 (cited on page 38).

Zheng, Yifei, Lixin Jia, and Hui Cao (2012). "Multi-objective gene expression programming for clustering". In: *Information Technology and Control* 41.3, pages 283–294 (cited on page 95).

Zhong, Jinghui, Liang Feng, and Yew-Soon Ong (2017). "Gene expression programming: A survey". In: *IEEE Computational Intelligence Magazine* 12.3, pages 54–72 (cited on pages 94, 110–113).

Zhou, Aimin et al. (2011). "Multiobjective evolutionary algorithms: A survey of the state of the art". In: *Swarm and Evolutionary Computation* 1.1, pages 32–49. ISSN: 2210-6502. DOI: https://doi.org/10.1016/j.swevo.2011.03.001. URL: https://www.sciencedirect.com/science/article/pii/S2210650211000058 (cited on page 37).