A first approach to some basilar questions on the relationship between state-of-the-art GP variants and GEP

Automatic Synthesis of Sorting Algorithms by Gene Expression Programming
+ (Geometric) Semantic Gene Expression Programming
+ Encouraging phenotype variation with a new semantic operator: Semantic Conditional Crossover

David Benedy Pereira Neto

Master Thesis submitted to
Faculdade de Ciências e Tecnologias,
Mestrado em Engenharia Informática

Supervisor: Prof. José Valente de Oliveira

2018
Gambelas, Faro

A first approach to some basilar questions on the relationship between state-of-the-art GP variants and GEP

· Automatic Synthesis of Sorting Algorithms by Gene Expression Programming

· (Geometric) Semantic Gene Expression Programming

· Encouraging phenotype variation with a new semantic operator: Semantic Conditional Crossover

Declaração de autoria de trabalho

Declaro ser o autor deste trabalho, que é original e inédito. Autores e trabalhos consultados estão devidamente citados no texto e constam da listagem de referências incluída.

Agradecimentos

Gostava de agradecer ao meu orientador, Prof. José Valente de Oliveira pelo contributo e papel que teve no desenvolvimento desta tese. Desde as discuções de ideias à forte motivação e confiança que teve nas várias fases do projecto, este não teria sido possivel sem o seu envolvimento.

Agradeço aos meus pais, Rute e Tó, pelo apoio incondicional e força que sempre me deram, neste projecto e em tudo o que me decido envolver.

Abstract

Gene Expression Programming (GEP) is an alternative to Genetic Programming (GP). Given its characteristics compared to GP, we question if GEP should be the standard choice for evolutionary program synthesis, both as base for research and practical application. We raise the question if such a shift could increase the rate of investigation, applicability and the quality of results obtained from evolutionary techniques for code optimization.

We present three distinct and unprecedented studies using GEP in an attempt to develop understanding, investigate the potential and forward the branch. Each study has an individual contribution on its own involving GEP. As a whole, the three studies try to investigate different aspects that might be critical to answer the questions raised in the previous paragraph.

In the first individual contribution, we investigate GEP's applicability to automatically synthesize sorting algorithms. Performance is compared against GP under similar experimental conditions. GEP is shown to be capable of producing sorting algorithms and outperforms GP in doing so.

As a second experiment, we enhanced GEP's evolutionary process with semantic awareness of candidate programs, originating Semantic Gene Expression Programming (SGEP), similarly to how Semantic Genetic Programming (SGP) builds over GP. Geometric semantic concepts are then introduced to SGEP, forming Geometric Semantic Gene Expression Programming (GSGEP). A comparative experiment between GP, GEP, SGP and SGEP is performed using different problems and setup combinations. Results were mixed when comparing SGEP and SGP, suggesting performance is significantly related to the problem addressed. By out-performing the alternatives in many of the benchmarks, SGEP demonstrates practical potential. The results are analyzed in different perspectives, also providing insight on the potential of different crossover variations when applied along GP/GEP. GEP's

compatibility with innovation developed to work with GP is demonstrated possible without extensive adaptation. Considerations for integration of SGEP are discussed.

In the last contribution, a new semantic operator is proposed, SCC, which applies crossover conditionally only when elements are semantically different enough, performing mutation otherwise. The strategy attempts to encourage semantic diversity and wider the portion of the semantic-solution space searched. A practical experiment was performed alternating the integration of SCC in the evolutionary process. When using the operator, the quality of obtained solutions alternated between slight improvements and declines. The results don't show a relevant indication of possible advantage from its employment and don't confirm what was expected in the theory. We discuss ways in which further work might investigate this concept and assess if it has practical potential under different circumstances. On the other hand, in regards to the basilar questions of this investigation, the process of development and testing of SCC is performed completely on a GEP/SGEP base, suggesting how the latest can be used as the base for future research on evolutionary program synthesis.

Resumo

Programação Genética por Expressões (GEP) é uma alternativa recente à Programação Genética (GP). Neste estudo observamos o GEP e colocamos a questão se este não deveria ser tratado como primeira escolha quando se trata de sintetização automática de programas através de métodos evolutivos. Dadas as caracteristicas do GEP perguntamonos se esta mudança de perspectiva poderia aumentar a investigação, aplicabilidade e qualidade dos resultados obtidos para a optimização de código por métodos evolutivos.

Neste estudo apresentamos três contribuições inéditas e distintas usando o algoritmo GEP. Cada uma das contribuições apresenta um avanço ou investigação no campo da GEP. Como um todo, estas contribuições tentam obter cohecimento e informações para se abordar a questão geral apresentada no páragrafo anterior. Na primeira contribuição, investiga-mos

e testamos o GEP no problema da sintese automática de algoritmos de ordenação. Para o melhor do nosso conhecimento, esta é a primeira vez que este problema é abordado com o GEP. A performance é comparada à do GP em condições semelhantes, de modo a isolar as caracteristicas de cada algoritmo como factor de distinção.

As a second experiment, we enhanced GEP's evolutionary process with semantic awareness of candidate programs, originating Semantic Gene Expression Programming (SGEP), similarly to how Semantic Genetic Programming (SGP) builds over GP. Geometric semantic concepts are then introduced to SGEP, forming Geometric Semantic Gene Expression Programming (GSGEP). A comparative experiment between GP, GEP, SGP and SGEP is performed using different problems and setup combinations. Results were mixed when comparing SGEP and SGP, suggesting performance is significantly related to the problem addressed. By out-performing the alternatives in many of the benchmarks, SGEP demonstrates practical potential. The results are analyzed in different perspectives, also providing insight on the potential of different crossover variations when applied along GP/GEP. GEP's

compatibility with innovation developed to work with GP is demonstrated possible without extensive adaptation. Considerations for integration of SGEP are discussed.

Na segunda contribuição, adicionamos ao processo evolutivo do GEP a capacidade de medir o valor semântico dos programas que constituem a população. A esta variante damos o nome de Programação Genética por Expressões Semântica (SGEP). Esta variante trás para o GEP as mesmas caracteristicas que a Programação Genética Semântica(SGP) trouxe para o GP convencional. Conceitos geométricos são também apresentados para o SGEP, extendendo assim a variante e criando a Programação Genética por Expressões Geométrica Semântica (GSGEP). De forma a testar estas novas variantes, efectuamos uma experiência onde são comparados o GP, GEP, SGP e SGEP entre diferentes problemas e combinações de operadores de cruzamento. Os resultados mostraram que não houve um algoritmo que se destaca-se em todas as experiências, sugerindo que a performance está significativamente relacionada com o problema a ser abordado. De qualquer modo, o SGEP obteve vantagem em bastantes dos benchmarks, dando assim indícios de potêncial ter utilidade prática. De um modo geral, esta contribuição demonstra que é possivel utilizar tecnologia desenvolvida a pensar em GP no GEP sem grande esforço na adaptação. No fim da contribuição, são discutidas algumas considerações sobre o SGEP.

Na terceira contribuição propomos um novo operador, o Cruzamento Semântico Condicional (SCC). Este operador, baseado na distância semântica entre dois elementos propostos, decide se os elementos são propostos para cruzamento, ou se um deles é mutato e ambos re-introduzidos na população. Esta estratégia tem como objectivo aumentar a diversidade genética na população em fases cruciais do processo evolutivo e alargar a porção do espaço semântico pesquisado. Para avaliar o potêncial deste operador, realizamos uma experiência prática e comparamos processos evolutivos semelhantes onde o uso ou não uso do SCC é o factor de distinção. Os resultados obtidos não demonstraram vantagens no uso do SCC e não confirmam o esperado em teoria. No entanto são discutidas maneiras em que o conceito pode ser reaproveitado para novos testes em que possa ter potêncial para demonstrar resultados possitivos. Em relação à questão central da tese, visto este estudo ter sido desenvolvido com base em GEP/SGEP e visto a teoria do SCC ser compativel com GP, é demonstrado que

um estudo geral à àrea da sintese de algoritmos por meios evolutivos, pode ser conduzido com base no GEP.

Contents

1. Introduction

1.1 Context

Studied and documented by Charles Darwin in the 19th century, evolution can be described as a generational process in which defining traits of a species are continuously shaped by the process of natural selection in response to characteristics and changes in the environment it's elements live on.

Evolutionary computation, a computer science and artificial intelligence branch, studies how evolution can be applied to find solutions to computational problems. Several approaches derive from this broad concept. Genetic Algorithms are among the best known and studied examples. This approach consists in employing concepts associated with evolution to develop problem-independent (general purpose) optimization algorithms. [2]

Diving deeper in the branch of Evolutionary Computation, there is a particular approach of great interest, Genetic Programming. In GP, a group of executable instructions is combined by means of evolution to generate executable programs. The objective is to search for a final executable composed computer program which excels at a particular given task.

Unlike conventional GA algorithms, where individual chromosome representations are usually linear strings of characters, often with constant length, to represent computer programs (which can be seen as a set of nested instructions) GP algorithms commonly rely on a parse tree structure. Parse trees are ramified structures with different shapes and sizes. Although good to represent functional programs, these structures have the downside of making the process of evolution difficult. This happens because combining and mutating parse trees while maintaining syntactical correctness of the programs they represent is hard. In practice, it is usually necessary to rely on complex genetic operators and/or repair functions to fix

trees representing syntactically incorrect programs. Another challenge with parse tree structures in GP is bloat. In the context of GP, bloat is a phenomenom that occurs when parse tree instances grow in size from accumulating code structures irrelevant to their execution sequence. [3]

Gene Expression Programming is a promising alternative approach to GP which provides features to handle the challenges mentioned above. Furthermore, it has other innovative characteristics which make structural changes in some aspects of the evolution process. GEP main difference to GA and GP, is the structure of its chromosomes. In GEP chromosomes are formed using a language specifically developed for this purpose, Karva language. This language states rules to codify parse trees in linear expressions of fixed size. This allows the element structure to have the simplicity and ease of employment seen in Genetic Algorithms and the functional complexity and purpose of Genetic Programming. [4]

Going back a step to GP and parse trees, another complication with this combination is that considerably small change in the tree structure (e.g. swap in a single leaf) often result in large, and sometimes critical, modification to behavior of represented programs. The opposite situation also applies, i.e., apparently large modification in the structure (e.g. swapping more than half of the leafs) possibly causes no change at all in the execution behavior of the represented program. In other words, it is possible to classify the relation between genotype and phenotype (syntax and semantic) to be fairly complex in GP due to Parse Tree nature. It's possible to say that in GP, behavior, or in other words semantics, is simply not considered during the evolutionary process. [5]

Semantic Genetic Programming is a specialization of conventional Genetic Programming which tackles this problem. It works by introducing awareness of the candidate program behavior in the evolutionary process. To do so, programs semantics is measured and exposed as a characteristic of the element to semantically biased operators, which use this value to either directly or indirectly manipulate candidates or the population as a whole.

1.2 Goals

We strive to discuss, gain insight and provide a first approach to four different but related kinds of questions:

Questions of the first kind: Basilar guiding questions

- Where is GEP positioned relatively to GP as a technology? Is GEP a sub-branch of the Genetic Programming field? Or is it an alternative to the GP algorithm? Is GEP to be considered an enhancement or is it a different concept?
- When handling research or practical applications on evolutionary program synthesis, should GEP be considered and experimented with first before conventional GP?

Questions of the second kind: Application oriented questions

- Can GEP be used to synthesize Sorting Algorithms? Can it be done in the same way as it has been done before with GP?
- How does GEP behave when compared to standard GP approach performing this task?
- How does GEP evolution process affects the generated programs?
- Where is GEP positioned relatively to GP and when should it be used?
- Can unprecedented sorting algorithms be generated automatically with less or no human intervention at all?

Questions of the third kind: Semantic oriented questions

- What are the limitations of GEP?
- Sharing a lot of principles with GP, does GEP share some of its limitations and perks as well?
- Can it be enhanced? In what ways?
- Is GEP compatible with innovation developed on top of GP?
- How can GEP become aware of its individuals (programs) semantics? Can it be done similarly to SGP?
- Can SGEP use semantic space geometry to enhance its results like GSGP does over SGP?

- Considering the new SGEP approach, how does it perform compared to normal GEP, GP and SGP?

Besides, we would like this work to serve as a bridge between two technologies which display great potential and which demonstrate to work very well together. We believe GEP brings easiness of employment which could improve and facilitate the study of Semantics for the purpose of generating computer programs using EC techniques.

Other questions

- Can perform crossover conditionally be a viable strategy in GEP?
- When should this new approach, SCC, be considered for usage?
- Can GEP/SGEP be used as base to develop innovation that works for GEP, GP and other evolution based computer program generation techniques? Is it advantageous relative to GP?

1.3 Contributions

This work offers three distinct contributions. As a whole, these three contributions are bind with the purpose of developing understanding, investigate the potential and forwarding the branch of GEP. Possibly the most central contribution we strive to provide is insight in where GEP positions relatively to GP and if there would be benefit in it being the primary option of choice for future investigation and practical application in place of conventional GP algorithms. We will investigate the quality of results produced, compatibility with existing GP innovation, possible difficulties and considerations of GEP. Individually, each contribution has its own value and is a complete study on its own. We will have a look at each investigation separately in following paragraphs.

As for the three individual contributions, they will merge GEP with concepts previously only applied to GP, for instance program semantics and a new challenge previously only associated with GP, the generation of sorting algorithms. The latest will be associated with all the three experiments, in the first as the main challenge and in the other two as control for performance measurement.

The first contribution is the application of GEP to the automatic synthesis of sorting algorithms. To the best of our knowledge, this is the first time this approach/problem combination is tested. Comparison is done between GEP and GP in terms of performance and quality of solutions.

The second contribution is the development of SGEP and GSGEP, a semantic aware modification of GEP. By following the same principles, this new approach gets GEP up to date with the Semantic and Geometric Semantic branches of investigation that recently branched out of Genetic Programming. GEP and SGEP are faced with a set of benchmarks from 3 different distinct problems and compared to results obtained with GP and SGP.

The third contribution presents a new semantic algorithm, SCC. This algorithm follows the same strategy as the Wang algorithm, but instead of conditioning crossover based on a specific syntactical evaluation of elements, the conditioning is done based on the semantic value of elements. Comparative studies are performed under the same conditions with and without this strategy to observe potential. Development and testing of this operator is done on SGEP alone, suggesting that the approach is capable of being a base for development of innovation that is compatible with both GEP and GP.

1.4 Thesis Organization

The thesis report is divided into 7 chapters. We are currently reading the *Introductory* chapter, which provides a resume of what we will find as well as how it is organized. Next chapter, *State of the art*, includes an overview and brief description of all the topics we consider background to the development of these experiments and relevant to their understanding. While the three research questions share much of the same background, they do not overlap completely. Topics *2.6 Semantic Genetic Programming* and *2.7 Geometry of Genetic Programming* are only relevant for the third and fourth kind of questions.

Chapters 3, 4 and 5 describe each individual contribution: "*Automatic Synthesis of Sorting Algorithms by Gene Expression Programming*", "*(Geometric) Semantic Gene Expression Programming*" and "*A new Semantic Operator: Semantic Conditional Crossover*". Although not having the exact same sections, the structure of the chapters roughly follows the same

sequence. First an introduction to the theory and context of the experiments followed by a description of what was developed and how the practical part of the experiment was setup. Experiment data and results are then presented along with a description and discussion of observations. Finally there is a concluding section reviewing what was done and a resume of most important considerations.

Chapter 6, *Observations and Discussion*, presents a resume of the three experiments and discussion on the results and observations. Considerations regarding the topics binding the experiments and core to this work as a whole are also discussed here.

Finally, chapter 7 presents the *Conclusion* where we briefly describe what was done and present final remarks.

2. State of the art

In this chapter, we describe the essential background concepts to this investigation. At each section we introduce new concepts. Starting from the more fundamental, the topics gradually get more specific until covering the branch where our contributions are built over. If the reader is familiar with any of these concepts, it's reasonable to jump over some of the topics (or the entire chapter, for someone who is closely familiar with the subject). After reading this chapter, one should be fairly more comfortable to proceed to the next chapters where we present the actual contributions.

2.1 Evolution of Species

As first proposed by Charles Darwin in "On the Origin of Species by Means of Natural Selection" (1859), Evolution is the process in which species characteristics are continuously shaped, generation after generation, in response to changes on the surrounding environment. Working as a Geologist and Naturalist, Darwin observed how certain advantageous traits in animals would positively influence their capacity to perform essential tasks to survive and eventually reproduce. Compared to these well-adapted individuals, inapt peers, with less useful, underdeveloped or inadequate traits, would have an harder time surviving and reproducing. Individuals which survive longer tend to generate more offspring and will more likely have their traits passed on to the next generation. On the other hand, traits which would cause individuals to perish or unable to reproduce will tend to disappear as fewer or no individuals will ever inherit them. To this process, Darwin would call Natural Selection.

It is now known that each living being's phenotype (trait) is encoded in their genotype (genome or sequence of genes). Offspring inherit a rearranged combination of part of both

parents genome which, along with mutation, provides them with their individual set of characteristics.

Although Natural Selection filters out which traits are advantageous, other processes are necessary to generate new and unprecedented ones. Different re-combinations of the same genes occur naturally and can create distinct traits by themselves. On the other hand, some genes in parts of the genome, know as non-coding genes, don't explicitly translate into any phenotype at all. When inherited however, what were in the parent non-coding sequences of genes, can sometimes in the child regain an active role, and once again shape the traits of the individual. This allows characteristics to be "hidden" and even skip several generations before showing up once again. Finally, and perhaps the most remarkable way of creating new and diverse traits, is the genome capability to mutate. Mutated genome contains genes that are not inherited from individual's parents. Instead, these genes suffered random and unpredictable modifications which might translate into unprecedented characteristics. Each of these new characteristics can influence the individual ability to survive and will be submitted to environment evolutionary pressure in the same way that inherited characteristics are.

Evolution is responsible for the shaping of every complex life form known. It's very likely that humans owe the ability to think rationally to a long process of evolution. Besides being one of the basis of modern biology, in the next topic we observe how Evolution is used in computer science to automatically solve problems without the need for conventional analytical approaches.

2.2 Genetic Algorithms

Evolutionary Computation (EC) is an Artificial Intelligence branch that uses inspiration in Evolution to develop problem solving algorithms and optimization techniques. In this field, a commonly studied and well-know algorithm is the Simple Genetic Algorithm (SGA).

SGA is a stochastic iterative generational algorithm. The algorithm is initialized by defining a set of candidate solutions that function as a population. Evolution mechanisms are then applied at each iteration to create a new generation of the population. If evolution

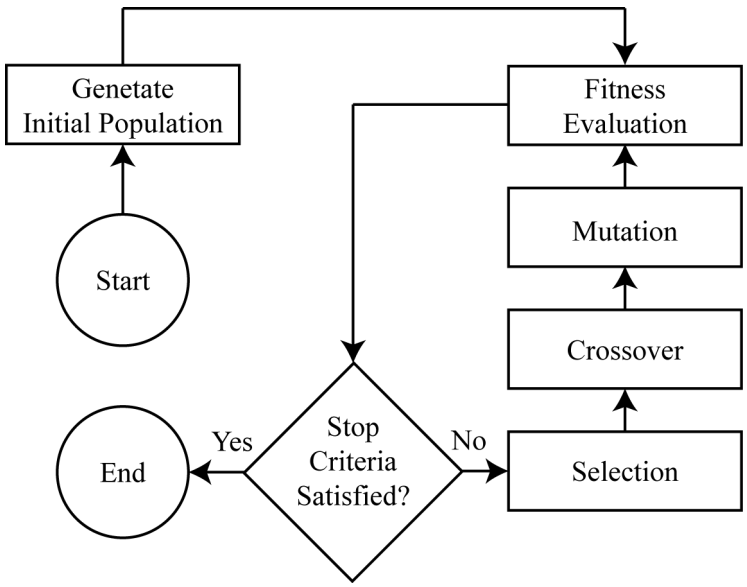


Figure 2.1: Simple Genetic Algorithm flowchart

conditions were correctly set, the new generation might have elements more adapted to solve the proposed problem than the previous. The problem designated for the GA to solve can be understood in evolution terms as the environment to which the population must adapt to survive on. More adapted solutions will correspond to better solutions.

One crucial step for setting up a GA is defining a chromosome representation that matches the problem and allows the elements to evolve at the same time. Creating a function to evaluate the ability of elements to solve the problem is also necessary and usually referred to as the fitness function. Elements that rank higher in the fitness function are considered more apt or fit and are more likely to be selected for reproduction. Reproduction comes in the form of crossover functions which purpose is to create new elements that blend parents characteristics. Before being arranged into a new population, elements can be mutated, which in SGA usually means having some parts of the individual's chromosome representation randomly changed, usually with a very low probability. Finally, the new elements replace the previous population to form a new generation. The process is repeated until a termination condition is met.

The details mentioned can vary greatly depending on the technique employed. Some branches of EC use Evolution in different ways, but the principles of generational adaptation and natural selection are usually present. Flowchart in figure 2.1 displays the flow of

execution of a SGA.

One of the most compelling reasons to apply evolutionary algorithms in practice is the shift this approach brings to the way problems are approached. In Evolutionary Computation, finding solutions to a problem means creating the right conditions for evolution to take place. Understanding of the problem is necessary to create a combination of a chromosome representation and a fitness function that are potentially good enough to allow the right kind of pressure and evaluation to be put on the elements. The remaining knowledge must be on configuring the remaining parameters of evolution and choosing the right operators for selection, crossover, mutation and replication so that the widest possible area of the solution space can be searched and the best possible solutions (global optima) gathered.

EC can be seen as multi purpose tool with the advantage that obtained understanding using this technique can be re-used independently of the problem at hands. The method contrasts with typical approaches which involve analytical development of solutions or optimization techniques which require deep knowledge on the specific problem and the use of, depending on the problem, more or less complex methodologies. Perks and downsides of EC include a tendency for premature convergence to local optima and the fact that in many occasions it might not be the most efficient or better performing optimization technique available.

2.2.1 Two-point crossover (2P)

Two-point crossover is one of the many existing evolutionary operators that can be integrated in a genetic algorithm. This particular form of crossover, which works on string chromosome representations of fixed length is of particular interest for this work as it integrates into our experiments as a control operator to compare with other crossover variations. The other relevant crossover operators will be covered later in this document.

Like most crossover operators, 2P starts with two parent chromosomes picked by a selection operator. These parents are part of a population composed by fixed length string chromosomes. From these two parents the intention is to generate children chromosomes which inherit characteristics from both parents. To do so, 2P randomly points to two positions between the extension of a population chromosome (recall that all chromosomes have

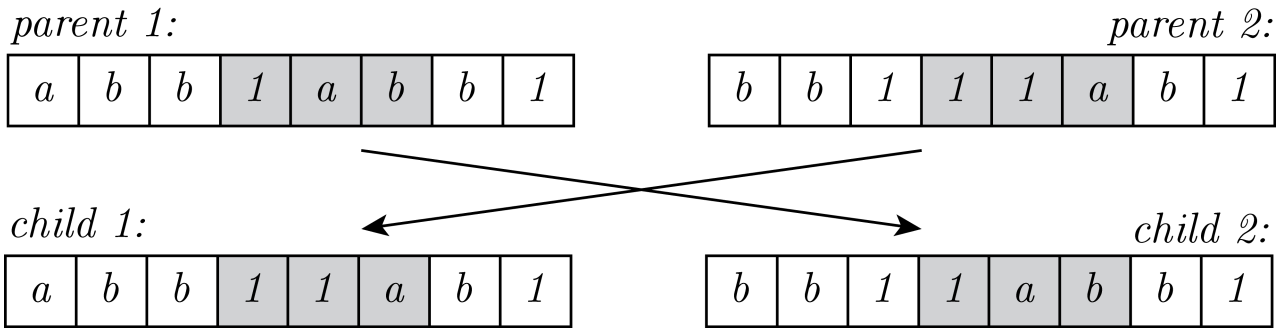


Figure 2.2: Two-point crossover (2P) example. Considering parents 1 and 2 and randomly selected positions 3 and 5, children 1 and 2 represent the new chromosomes resulting from 2P application.

the same length). Two child chromosomes are then generated by copying both parents but swapping with one another the characters contained in region between the two previously selected points. Notice how the child elements will unquestionably inherit syntactical sequences which are characteristic of both parents. Nevertheless, parents semantics is in fact ignored in the case of this operator. Semantic and syntactical characterization and modification will be explained throughout this work.

2.3 Genetic Programming

Genetic Programming (GP) is an Evolutionary Computation field where genetic algorithms are used to synthesize computer programs intended to solve a given computational task as best as possible. In other words, GP searches the space of possible programs for the ones that perform best doing the task at hands. To do so, evolution is used similarl

2.3.1 Parse Trees

In GP, programs are usually represented using Parse trees. Tree structures, being nonlinear entities with varying shapes and sizes, are capable of encoding certain levels of functional complexity and therefore ideal to represent the relation between nested instructions that conventionally make up computer programs. In standard GP, instructions can be clustered into two categories:

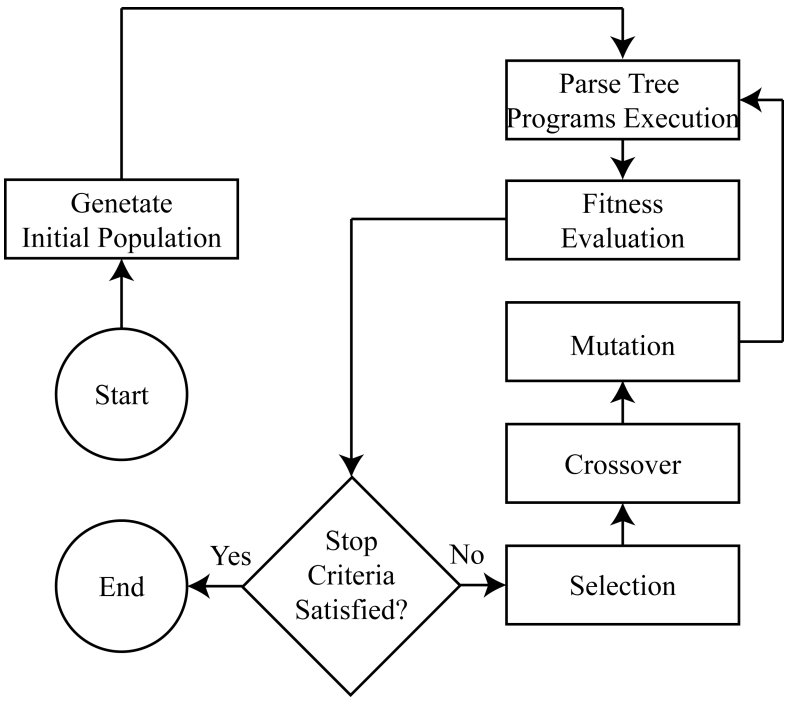


Figure 2.3: Genetic Programming flowchart

- Functions, which have predefined arity (number of parameters) and might be recursively nested;
- Terminals, which are either variables, constants or parameter-less functions (with arity = 0).

For a program to be syntactically correct and execute, all tree's leafs must be composed by terminals. Remaining nodes can be either a function or a terminal. Figure 2.4 displays an example of a computer program codified into a parse tree.

It's considered difficult to apply crossover and mutation on parse trees due to maintaining syntactical correctness of programs. This usually complicates the use and applicability of GP, with most algorithm variations either having operators for mutation with complex methods that prevent the generation of incorrect trees in the first place or needing repair functions to correct their syntax once generated. The process is also susceptible to bloat, which is the accumulation of nodes and leafs without actual role in the execution flow of the program or which are simply irrelevant in the program.

```
i = 1;
while(i < 20){
    i = i + 1;
}
```

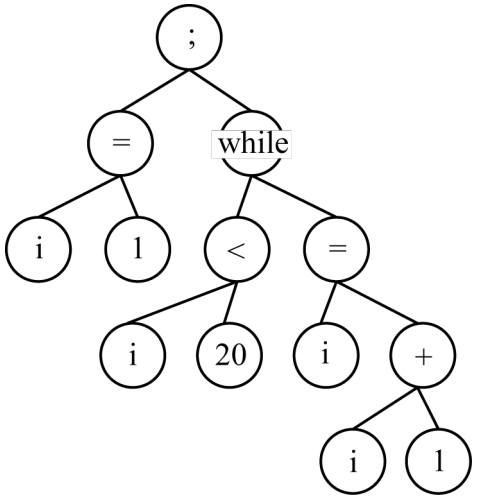


Figure 2.4: On the left a simple computer program and its Parse Tree representation to the right.

2.3.2 Advances and applications of GP

Reference [6] is considered one of the most complete and detailed descriptions of Genetic Programming. The knowledge contained is very often used as base for actual research on GP. Since the book was released, investigation as been made that furthered the field in many directions including theoretical and practical studies on structure, performance, applications and variations including concepts purposely developed or derived from other fields of research. Practically, GP is applied in all sorts of fields where data is large or complex enough that automatically processing is either more practical or an actual necessity.

In [7], Koza mentions many applications of GP where the technique manages to compete with traditional human implementation. Examples of applications include the development of electronic hardware, optical lenses systems, aid mechanisms for other artificial intelligence techniques and all sorts of data analysis. Along with very objective situations where GP is successfully applicable, [7] also presents a very interesting abstraction list describing situations and characteristics of problems where GP might be "especially productive".

Reference [8] mentions how GP is applied in cancer research but also to medicine in general. GP usage includes, among others: cancer research, new drug development, molecular

level research, breast cancer data analysis, radiological data analysis, clinical treatment, etc. When data from modern molecular techniques is not easily processed by common analytical approaches, GP is used solo and in combination with other methods of analysis. Although having a growing record of success applications in this field, GP limitations are reported to limit its usability, namely: Not guaranteeing global optimum nor being possible to tell if a solution is in fact the global best; Sometimes the data available being limited resulting in over fit solutions; Finally the fact that GP, in its stochastic combinatorial approach to problem solving, has a high demand for computational resources.

Another interesting application of Genetic Programming which might be especially relevant in the future is the generation of new algorithms for quantum computation. [9]

2.4 Automatic Synthesis of Sorting Algorithms

When performed by humans the design of sorting algorithms is recognized as a creative, technique, theoretical, and time consuming task, that requires intelligence. As a computer science textbook can teach us, sorting is an ubiquitous task in software, cf., e.g., [10, 11]. Currently there are many sorting algorithms but it seems that no sorting algorithm dominates all the others in all problems and circumstances. This has been a sufficient motivation for studying and analyzing sorting algorithms.

Let $(I_0, \ldots, I_i, \ldots, I_{N-1})$ be a sequence of N items where each item I_i has a key K_i for which a relation of order $(<)$ exists s.t. for any two keys K_i, K_j one has $K_i = K_j$, $K_i < K_j$ or $K_i > K_j$. The relation of order is transitive, i.e., for any three keys K_i, K_j, K_l if $K_i < K_j$ and $K_j < K_l$ then $K_i < K_l$. The comparison sorting problem can be stated as follows. Find a permutation ϵ s.t. $K_{\epsilon(i-1)} \leq K_{\epsilon(i)}$; $0 \leq i \leq N-1$, the sorted sequence being $(I_{\epsilon(0)}, I_{\epsilon(1)}, ..., I_{\epsilon(N-1)})$. As the items can have equal keys, the above permutation is not unique. Depending on the application, one may be interested in an unique permutation with the properties: i) ordering: $K_{\epsilon(i-1)} \leq K_{\epsilon(i)}$, $0 \leq i \leq N-1$; ii) stability: if $i < j$ and $K_{\epsilon(i)} = K_{\epsilon(j)}$ then I_i appears before I_j, in the ordered sequence.

A comparison sorting algorithm is an algorithm that solves instances of the above sorting

problem. Besides stability the following criteria can be used for algorithm characterization and comparison:

- **Time complexity**: measurement that estimates the time an algorithm takes to execute. Estimations are usually based on the number of elementary operations performed. Execution time may vary depending on the input and independently of its size. Worst-case scenario (big-Oh notation) time complexity is usually chosen to describe algorithms in order to cover all possible situations, but depending on the input, some algorithms may perform considerably better if given pre-conditions are met or for average cases. For average or worst case, comparison sorting algorithms performance is never better than O(N n log(N));

- **Space complexity**: measures the amount of memory storage the algorithm requires during execution. Estimations are based on variables and parameters, with each allocation being an elementary unit for measurement. Similarly to time complexity, algorithm's space complexity is usually described in the worst-case scenario (big-Oh notation). In the case of sorting, two distinct algorithmic approaches can be considered, i) the sorting is performed within the target collection of elements, with the algorithm therefore requiring no additional memory; ii) when a copy of the collection's n items or a reference to them must be stored (e.g. pointers);

  It's good practice to perform a so called 'indirect sorting' when each of the collection items occupy a non negligible amount of memory, i.e., sorting a reference of collection's items and not the items themselves in a way that first reference points to the item with smaller key, second reference to the item with second smaller key and so on. In this case, the sorting algorithm is referred to as indirect.

- **Location**: Relative to location, a sorting algorithm can be classified as performing internal or external sorting. On internal sorting, the collection of items is small enough to be completely allocated within the main memory. On the other hand, on external sorting, the collection is too big to be completely allocated inside main memory and the algorithm has to resort to mass memory, e.g. hard drive space. The main difference between the two approaches is that internal sorting has the complete collection available

for access at any given moment. In external sorting however, collection items need to be accessed sequentially or made available in blocks.

- **Adaptability**: A sorting algorithm is said to be adaptable if it's process considers the initial sorting state of the collection, i.e., whether the collection is sorted or mostly sorted to begin with. In contrast, an algorithm that ignores the inital state of the collection is said to be non-adaptable.

- **Deterministic**: A sorting algorithm may be described as i) probabilistic (or stochastic) or ii) deterministic, based on whether or not the sorting process relies on a random (probabilistic) factor. This random factor is introduced on the algorithm using a random number generator.

2.4.1 Integer Sorting

Also a classical computer science subject of study, Integer sorting is a subset of the more embracing Sorting problem. In this problem each item in the sequence has an integer key and the relation of order among the keys also corresponds to that of the integer subset of numbers. Furthermore, it's possible to apply a set of mathematical operations to integer values which conveniently come as standard in most programming languages. Specialized algorithms have been developed to tackle integer sorting in particular. Taking advantage of the mentioned particularities or also knowing the range of values in the sequence, degree of randomness and other particular characteristics of the sequence allows this problem to often be solved more efficiently than using a general purpose comparison sorter [12]. In this work we address Integer Sorting in particular.

2.4.2 Automatic Synthesis

Until recently, sorting algorithms have been designed by humans. Since [1, 13] synthetic sorting algorithms started to emerge with a considerable amount of research work being devoted to this subject or very similarly related problems, e.g., [14], [15], [16], [17], [18], [19].

Reference [14] shows how Push, a programming language developed for use with genetic

and evolutionary computation systems can be used to synthesize complex programs using not so simple control structures, namely sorting a list.

In [15], efficient recursive comparison sorting algorithms have been synthesized using an Object-Oriented Genetic Programming system.

GRAPE (Graph Structured Program Evolution) is employed in [17] and in [16] to evolve sorting algorithms. The technique is confirmed to obtain general sorting algorithms.

Reference [18] focus on the List Search algorithm synthesis using Genetic Programming. This problem shares some of the complexities of synthesizing sorting algorithms. The authors focus on the complexity of evolved algorithms and face the logarithmic-time search problem. The algorithms generated are shown to be general, correct and efficient. The authors refer the wish to find an "algorithmic innovation" not yet invented by humans, which is extremely interesting and also something we take as motivation for this thesis.

In [19] the authors approach the possibility to evolve recursive programs using GP. It's mentioned how recursive programs seem to face the obstacle of fragility when applying search operators to evolve. This is due to a small change in a correct program producing a completely wrong one. A solution is presented as a method for going from a recursive program to a non-recursive and back again. The tests are not performed on the sorting problem but on the very similar Reverse List problem.

In [1], the author investigates and demonstrates how Genetic Programming can be employed to generate iterative sorters with potential indications towards generality. In a follow up work, [13], the subject of generality of sorters synthesized by GP is explored in a greater depth. Kinnear investigates how altering the problem formulation on GP can impact the complexity of resulting algorithms. The results suggest a connection between generated algorithm size and generality, with generality being inversely proportional to size. It's questioned whether this connection extends beyond sampling based problems like sorting. It's also demonstrated that multiple fitness measures (like program size) can improve quality of synthesized programs.

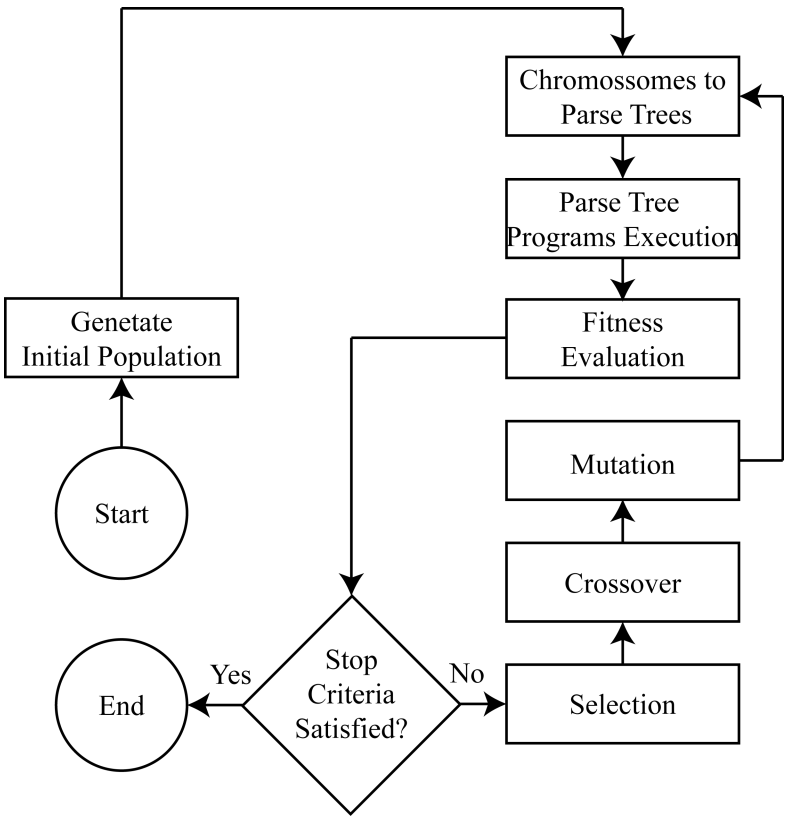


Figure 2.5: Gene Expression Programming flowchart. Notice that in GEP, unlike GP, Crossover and Mutation are performed at chromossomes level and not on Parse Trees. It's only for fitness evaluation purposes that each chromossome must be converted to its executable Parse Tree form. GEP exclusive operators, IS Transposition, RIS Transposition, Gene Transposition and Gene Recombination, if active, are to be applied during the "Crossover" step along with other active crossover operators.

2.5 Gene Expression-Programming

An overview of GEP and some of its mechanisms is provided below. If more details are necessary, [4] provides a complete and detailed description of GEP. GEP's evolutionary process is represented in figure 2.5

Developed by the biologist Cândida Ferreira, Gene expression programming (GEP) is a stochastic population based optimization technique that relies on evolution to create executable programs. GEP's evolutionary process uses mechanisms inspired by processes that take place in biology at the genetic level and attempts to mimic their functionality and role. This technique can be viewed as a sub-branch of EC with many functional similarities

to GP and also sharing its purpose and applicability. Both approaches, GEP and GP, are therefore alternatives to one another. GEP major distinction to GP is the structure of its chromosomes. In GP, chromosomes are parse trees. This remains true throughout the entire process of evolution, for both manipulation/handling and evaluation of individuals (which includes execution). GEP chromosomes however, assume two distinct structures depending on the stage of evolution. For manipulation/modification, chromosomes take the form of a linear string of fixed length, like in GA. This type of structure is easily mutable, combined, transposed, etc... making chromosomes easy to handle and manipulate. For fitness evaluation, chromosomes take the form of a parse tree, decoded from and corresponding to its linear string counterpart, which provides the functional complexity necessary to represent an executable program.

GEP relies on a purposely developed language, Karva, to provide the necessary rules for generating chromosomes and encoding/decoding between the two forms/structures. A chromosome formed by this language is referred to as a k-expression. One remarkable characteristic of this approach is that, by design, the element's linear string form will always represent a syntactically correct parse tree. This tackles one of the greatest challenges of working with GP, which is the necessity to employ repairing functions or specialized genetic operators that prevent syntactically incorrect programs. In the next sub-section we will look in greater detail at how Karva language works. Before moving further, it is important to mention that GEPs advantages are not achieved at a cost of performance or quality of results. In fact, the opposite seems to occur. In [4], the author of GEP presents several comparisons between GEP and GP handling different problems. GEP results surpass GP's, sometimes by orders of magnitude.

2.5.1 Chromossome representation

We mentioned how GEP chromosomes have two forms that can be employed as necessary, linear string of fixed size expression form and the parse tree form. It's important to observe that GEP's expression form might contain more information than the corresponding parse tree form it represents. In genomics, there is a concept of active and inactive DNA sequences. While active DNA sequences have a more straightforward and direct role in shaping the

host characteristics (phenotype), inactive DNA sequences are said to be dormant and not relevantly correspond to shape individual's characteristics. In biology, the role of these inactive sequences (or "Junk DNA" as it is sometimes referred to) is still debatable and trying to be better understood [20]. Inactive sequences most notorious consequence is that they allow characteristic traits to possibly jump generations. In GEP, expressions can have similarly occurring non-coding sections of genes. These non-coding sections are not present in the corresponding parse tree despite being on the expression. Before seeing how this affects GEP's workflow, we will first have a look into the conversion rules between k-expressions and parse trees by the means of the two following examples:

Example Consider the length of the hypotenuse equation (2.1) and Q as the codification for the square root function. This equation is represented by the parse tree to the right and encoded in 2.2 as a k-expression.

$$h = \sqrt{a^2 + b^2} \qquad (2.1)$$

$$Q + * * aabb \qquad (2.2)$$

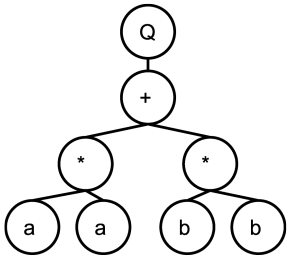


Figure 2.6: Hypotenuse equation, its parse-tree representation and a k-expression codification

The conversion k-expression to parse tree starts with defining the left-most character as tree root, in this case 'Q' (square root's character representation). 'Q' is a Function and therefore we pick the number of characters following correspondent to its arity. The picked characters are then branch down in the same order (left to right) below the parent node. 'Q' is the square root Function, which has arity 1. Picking the next 1 character from where the last evaluation was gets a '+' (sum), so we define this character as the child node from 'Q'. Defined all children of 'Q', next character is evaluated, in this case the '+'. Now the process repeats. '+' has an arity of 2, and so the next not yet used characters are two consecutive '*'s (multiply Function). The characters are placed left to right branching down from the parent '+' and '+"s evaluation is complete. Next evaluation is the first '*'. Arity is 2, so the next 2 unused characters are considered. The next character is the second '*' child of '+', already used, so ignored. Next there is an 'a' Terminal followed by a second 'a'. We place them branching down from the first (and so to the left) '*' node. Next we evaluate the second '*' Function. Arity is 2, and the next two unused characters are the two consecutive 'b' Terminals. We place them branching down from the second '*' node. Next evaluation is the 'a' Terminal. Terminals correspond to leafs and don't branch out. So we move to the next character. Another terminal, same procedure. Actually there are only terminals left until the end of the expression, so the conversion is complete.

Example As a second example, consider the root of the quadratic formula 2.3. This equation can be represented by the parse tree next to it. Observing this parse tree and corresponding k-expression 2.4, it's easier to visualize how the left to right codification rule works.

$$x = -b + \sqrt{\frac{b^2 - 4ac}{2a}} \tag{2.3}$$

$$+ - Q0b/ - * * *2abb4 * ac \tag{2.4}$$

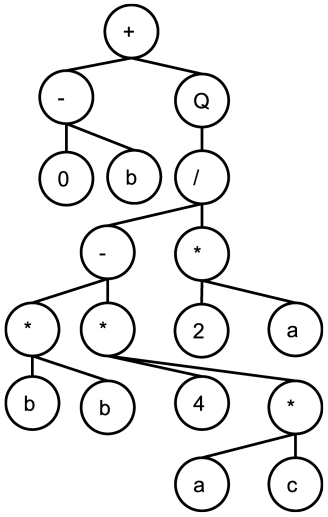


Figure 2.7: Root of quadratic formula, its parse-tree representation and a k-expression codification

2.5.2 K-expressions length and non-coding regions

K-expressions always end with terminals. That's because genes in GEP are composed by two sections, head and tail. Head size is a configurable parameter of evolution. Karva states a specific function 2.5 that based on head size and max-arity of available functions, calculates the size of Gene's tail and the consequent length of the entire expression. The size of all the k-expression of every element in the population is equal and unchangeable through the entire evolution run even though corresponding parse tree sizes can vary.

$$t = h(n - 1) + 1 \tag{2.5}$$

With t being the size of the tail and h the head. n is the maximum arity from the Function set. Head is composed by combinations of functions and terminals while tail can only contain terminals. Depending on the composition of the head, a specific number of elements will be necessary to complete the parse tree. The first elements up to that number that are present

in the parse tree form the active part of the gene. The remaining unused elements form the inactive part. Given formula 2.5, the gene is guaranteed to always have enough terminals to complete the tree, if not in the head, sufficient number is guaranteed to be in the tail. This elegant structure and set of rules is what gives GEP elements the ability to always maintain syntactic correctness.

Example Consider the following configuration. Note that all Functions have airty 2.

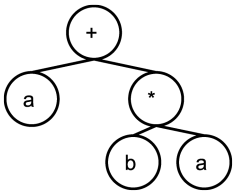
$$h = 3; \qquad (2.6) \qquad Functions = \{+, -, *\} \qquad (2.7)$$

$$n = 2; \qquad (2.8) \qquad Terminals = \{a, b, 2, 4\} \qquad (2.9)$$

$$t = h(n-1) + 1 = 4; \qquad (2.10)$$

Below are two examples of k-expressions and their corresponding parse trees. Tail section is highlighted in bold. In the first case two tail elements were necessary. This gene has an active section of size 5 and inactive of size 2. The active part always correspond to the number of nodes on the parse tree. In the second case, all the tail elements were necessary, with the active part of the gene being size 7, inactive size 0.

+a*__bab4__



-/-__ab24__

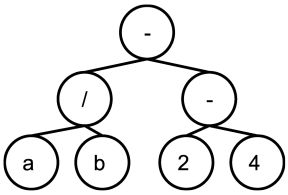


Figure 2.8: Examples of k-expressions and corresponding parse trees

Now lets assume changes in initial conditions to parameter 2.12. Consider function F to have arity 3. This changes n value to 3 and consequently t value to 7.

$$n = 3; \qquad (2.11) \qquad Functions = \{+, -, *, F\} \qquad (2.12)$$

$$t = h(n-1) + 1 = 7; \qquad (2.13)$$

The example below displays a k-expression formed using the the new conditions. 5 tail elements were necessary, and so the active section gets size 8 leaving 2 tail elements inactive.

-/F**2244aba**

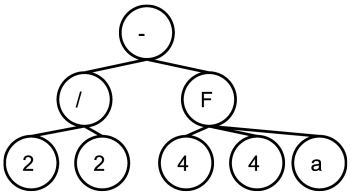


Figure 2.9: K-expression with 2 inactive tail elements and corresponding parse tree

2.5.3 Multigenetic expressions

GEP allows expressions to have more than a single gene, hence called multigenetic expressions. The number of genes per expression is an user defined parameter in the same way as head-size. Each gene encodes a parse tree, so multigenetic expressions represent several parse trees. Multigenetic expressions are therefore useful to apply in problems where multiple outputs are desired, but not only. In the same way as individual genes can have inactive sections, expressions can contain inactive genes. Inactive genes can become active during modification by the means of specific evolutionary operators. Along with GEP, in [4], the author introduces some specific operators to be employ along with GEP. Gene Transposition is one of those operators which randomly selects one of the expression's genes and transpose it to the expression's first position. This would exchange the active parse tree and therefore correspond to a different program. Multigenetic expressions have an impact on evolution dynamics as it provides the possibility for complete programs to be inactive and becoming relevant in later stages of evolution, bringing along their unique characteristics. Below is an example of a multigenetic expression and corresponding encoded parse trees using the conditions described in **Example 3**.

FF-**abab222**+ab**222444b**

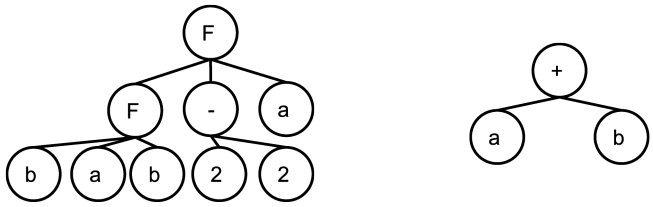


Figure 2.10: Multigenetic k-expression and corresponding parse trees

2.5.4 GEP exclusive operators

GEP can use standard GA and GP operators for selection and replication. For variation operators (mutation, crossover, transposition...) GEP, giving the nature of k-expressions, can employ linear string operators commonly available for GA. The only constrain is maintaining tail section containing only terminals at all times. GP specific operators that employ techniques to maintain syntactical correctness of parse trees are not necessary (neither are repairing functions). The following operators were specifically proposed along with GEP and take advantage of k-expression's structure and features:

- Transposition of insertion sequence of elements (IS)
- Root transposition (RIS)
- Gene transposition
- Gene Recombination

Transposition of insertion sequence of elements (IS)

A random sequence from inside a gene is selected and transposed to a target position on any position of that same gene's head, exception for the first position. The picked gene, its starting position, sequence size and target position are randomly selected. After transposition, the sequence is trimmed as necessary not to surpass head size and to maintain k-expression structure integrity.

Root transposition (RIS)

A random point in a gene's head is chosen and starting from that point, RIS looks for the next Function in that head. If no function is found, the operator does nothing. If a function is found, a sequence starting from there and of random length is copied to the first position of the gene (hence the name root transposition). Elements at the end shift right to accommodate the sequence, while elements at the end of the head are trimmed as necessary to maintain k-expression structure integrity. The picked gene, function search starting position, sequence length and target position are randomly generated.

Gene transposition

This operator is already described in the multigenetic section (2.5.3).

Gene transposition

Gene transposition is a form of crossover. Given two expressions, this operator randomly selects a gene from each and swaps those entire genes to generate children.

2.6 Semantic Genetic Programming (SGP)

SGP is similar to GP with the difference that its evolutionary process has the ability to observe, measure, manipulate and make decisions based on candidate program's semantics.

2.6.1 Program's Semantic

Program's semantic, meaning, or behavior, is the defining characteristic of a program that is derived from the effects produced on all involved resources by its operational procedures from initial to final states of execution. A program's semantic description can be labeled as complete when it contemplates all possible initial states of involved resources and respective consequential state images.

The initial state often includes the program's input. After performing its composing sequence of actions, a returning output based on the initial state is returned. One way to represent

program's semantic can be the complete list of inputs and respective outputs which, in other words, corresponds to the entire set of domain image pairs. This representation alone does not include information on the state changes to involved resources besides input/output, but in practice, and for the examples where spacial, temporal and other complexities are not very relevant, it might suffice as a means of comparing programs in terms of part of their semantic. In other words, it is important to realize that semantics may comprise a lot of distinct and classifiable aspects of a program. In practice, a decision on which of the semantic characteristics should be considered for the task at hands is most likely a necessary step.

Comparing semantics based on domain image pairs

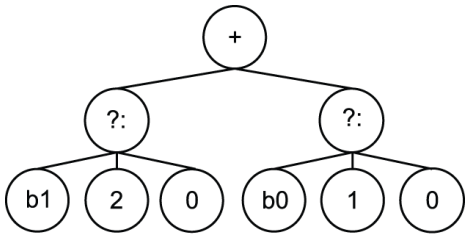
Consider only semantic descriptions based solely on input/output. A part of the complete set of domain image pairs is a semantic representation of all the programs to which the group of pairs verifies and, although not complete, can be used to identify semantic differences/similarities between programs. This sub-set of the complete domain image pairs is important for this study as it has been applied in the form of test-sets for semantic based research experiments, particularly in the recent semantic genetic programming trend of investigation. In this work we make use of this same test-set approach on the experiments introduced to the reader throughout the remaining of this thesis.

It's also important to mention that programs with equal domain image pairs are considered semantically identical, even if syntactically different. Syntactically equal programs however, when applied over the same context, will always have identical input/output pairs, and therefore equal semantics.

Figure 2.11 provides an example of a program and a possible representation of its complete semantic based solely on input/output domain image pairs.

Figure 2.12 provides an example of two programs with the same input/output domain image pairs, but in which an added characteristic is provided: time complexity. If time complexity is to be considered in semantics, then the algorithms are semantically distinct.

$$n = (\, b_1 \,?\, 2 \,:\, 0 \,) + (\, b_0 \,?\, 1 \,:\, 0 \,) \qquad (2.14)$$



Input		Output
b1	b0	n
false	false	0
false	true	1
true	false	2
true	true	3

Figure 2.11: Expression 2.14 is an example of a program. "? :" is the ternary conditional operator and b_0, b_1 Boolean variables. Below is the program's parse tree representation and to the right the complete set of possible input/outputs of the program, which, by covering all possible domain image pairs, can be used as the program's complete semantic description.

Input		Output
b1	b2	10pow
false	false	1
false	true	10
true	false	100
true	true	1000

Time complexity: O(n)

Input		Output
b1	b2	10pow
false	false	1
false	true	10
true	false	100
true	true	1000

Time complexity: O(n^2)

Figure 2.12: Consider the resulting domain image pairs of two hypothetical and distinct algorithms. Although having the same domain image pairs, the algorithms can be considered semantically equal (or not) depending on whether time complexity is considered to be relevant in programs semantic measurement for the task at hands.

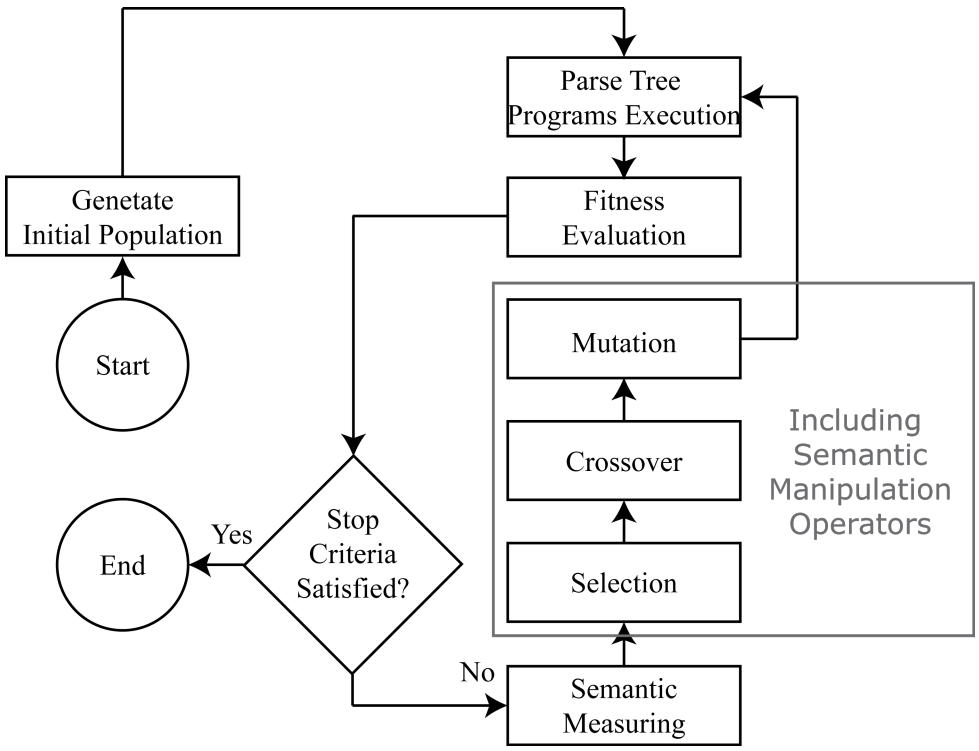


Figure 2.13: Semantic Genetic Programming flowchart. SGP differs from GP in the introduction of Semantic Measuring step and at least one semantic based operator in Selection, Crossover or Mutation. Semantic Measuring step is positioned after the Stop Criteia conditional step, however, depending on the method used, it can be done simultaneously with Fitness Evaluation.

2.6.2 SGP algorithm

Figure 2.13 shows the evolution process of SGP. To understand SGP, its important to realize that there is a gap between GP's intended behavior when employing evolution concepts and the way it actually operates.

In standard GP, to search the space of programs, candidate program's syntax has to be manipulated. Operators that manipulate, however, ignore effects of modification done on program behavior. In other words, GP and its functionality ignore element program's semantics. Although GP has proven to work in many problems to some extend, it's questionable why such approach to evolution would provide the best results considering that what is being searched for is not optimal syntactical composition but optimal program behavior in solving a given problem [21].

Before introducing Semantic Genetic Programming which addresses this problem, the next

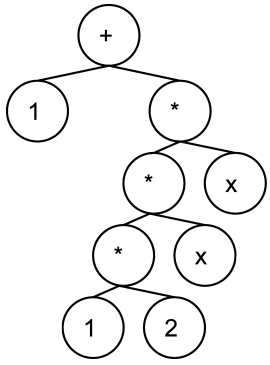
section describes how a parse tree structure can amplify the disparity between what conventional GP operators do and the actual desired effect intended from evolution.

2.6.3 Consequences of GP's unawareness of program's semantic

Recall that in Genetic Programming, Parse Trees are commonly used as population individuals genotype structure (variations exist, e.g. [22]). Due to their nature, considerably small change in this ramified structure (e.g. swapping a single node) often result in critical modification to the semantic of the program it encodes. The opposite situation also applies, i.e., apparently large alteration in the tree composition (e.g. swapping more than half of the leafs) possibly causes no change at all in program semantic.

Closely observing operators exposes how ignoring program's semantics affects evolution. Crossover for instance, has the purpose of passing on relevant characteristics from parents to children in the form of building blocks. When semantics is not considered this might not be the case, with children possibly end up with completely different and disconnected behavior from both parents. As another example, mutation's purpose of introducing small variation is also in question because of this same disparity between syntactic modification and consequences on semantics [23]. Some practical examples of this problem are described in the following example.

Example 4 Consider the following parse tree, the function it encodes and (an exert of) its graphical representation.



$$y = 1 + 2xx \qquad (2.15)$$

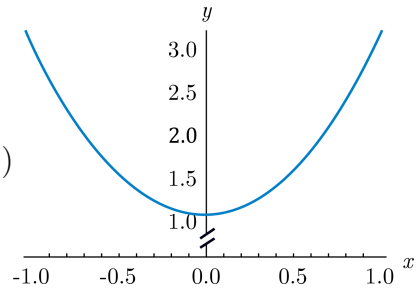
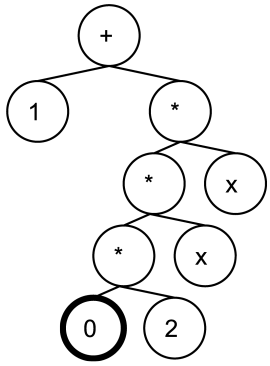


Figure 2.14: An example of a 2 dimensional expression, corresponding parse tree and graphical representation of its semantics.

Now consider the new parse tree below, a possible result from mutation. Notice the highlighted node, it highlights where modifications happened. A single terminal in a leaf node was swapped by another terminal. Syntactically, this is a really small modification, however, in terms of semantics, the program represents a function with considerably distinct image. We can see in the graphical representations how it goes from a parabolic slope to a linear plot.



$$y = 1 \qquad (2.16)$$

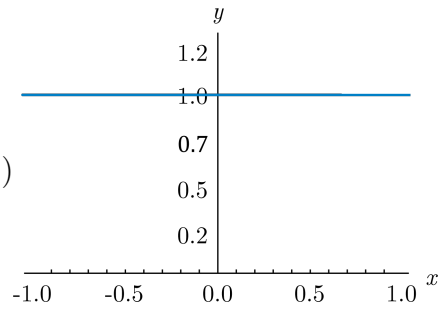
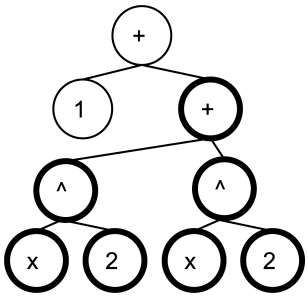


Figure 2.15: Disparity between syntactical and semantic changes: a mutated 2 dimensional parse tree with syntax similar to the originating element but whose semantics is un-proportionally distinct.

The next example represents the opposite situation. This parse tree could result from a crossover which included the tree in the first example (2.17) as one of the parents. Notice the highlighted branches. Syntactically, this is a major change with only two nodes remaining the same: root and left child. Starting on the right root's child, the complete sub branch was replaced by a new, syntactically different, one. When looking at the function the new program represents, however, it has exactly the same meaning as the tree in (2.17), in other words, the trees are semantically identical:



$$y = 1 + x^2 + x^2 \qquad (2.17)$$

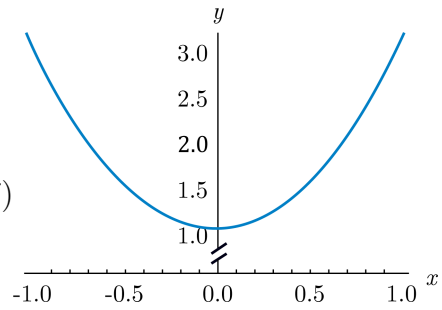
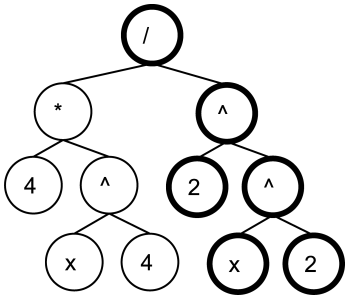


Figure 2.16: Disparity between syntactical and semantic changes: a crossover result 2 dimensional parse tree with very distinct syntax to one of its parents but whose semantics value is equal to that same parent.

As another example to demonstrate the consequences in crossover, consider the tree below, which is semantically similar to the previous one: both corresponding to similar plots of the same convex parabola, only difference being point of interception, which is different by one unit.



$$y = 4x^4/2x^2 \qquad (2.18)$$

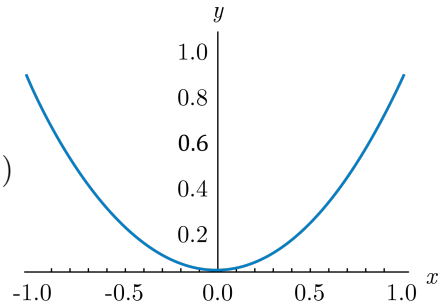
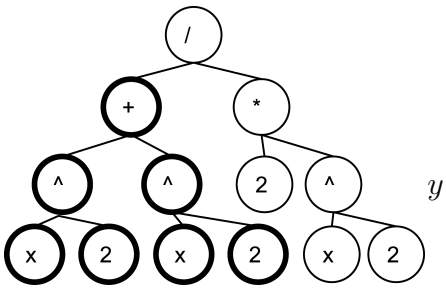


Figure 2.17: A new 2 dimensional parse tree with syntax very distinct to the one in 2.18 but very similar semantics.

A possible crossover between the two previous elements could look like the new tree below. Although having parents very similar semantically and its syntax being a simple combination of theirs, the child's semantics is completely dissimilar from both its parent's.



$$y = (x^2 + x^2)/2 * x^2 \qquad (2.19)$$

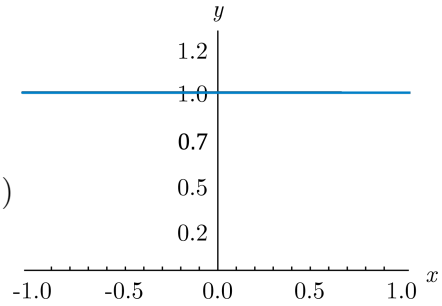


Figure 2.18: Disparity between syntactical and semantic changes: An example of a 2 dimensional parse tree likely to be a result from crossing over 2.17 and 2.18 but whose semantics is completely different from both parents'.

2.6.4 GP to SGP: Measuring and manipulating program semantics

Semantic GP (SGP), is a GP variation that aims to tackle the disparity problem described in the previous subsection. To fix the lack of awareness of program's semantics, this variation introduces two changes over GP's algorithm: Semantic Measuring and Semantic Manipulation.

Semantic Measuring is done individually for each program. Although different methods to perform Semantic Measuring exist, in existing research and depending on the problem being addressed, it is commonly done by facing candidate programs against a training set of tests and collecting the outputs. The outputs are stored associated with the element and used as program's semantic value (sampling approach) [23] [24]. Often, Semantic Measuring can be done simultaneously with Fitness Evaluation, and even using a relatively similar method. Fitness and Semantic values, however, serve very distinct purposes in the evolutionary process. Clarifying this difference helps understanding their roles and how SGP handles both concepts:

fitness

Fitness is handled in the same way as in GP and other evolutionary algorithms. To measure fitness, a fitness function is created, possibly using several results from sample tests ran in the element programs. This samples output serve as input to the fitness function, which should use this data to, as accurately as possible, represent the program aptitude to solve the given problem. Fitness is used to **rank programs relatively to their capability to solve the problem**. Operators then use this fitness value appropriately, considering what it represents. For example selection operators, which chooses a group of elements for crossover based on their fitness.

semantic

To measure semantics, a semantic function can be created. Results from sample tests can be used as well, the difference here is what the value of semantics represents and its purpose. Semantic value represent program's behavior. Its purpose is **positioning programs relative to each other in terms of their meaning**. Ultimately, the semantic value can be used to define a distance between programs in the semantic space [25]. This distance can then be used by operators to make informed decisions on how to manipulate programs knowing their degree of semantic similarity or disparity.

Semantic Manipulation refers to operators handling programs while being aware of their semantics. Basically means the operator is informed on how the program behaves and can adapt its decisions based on it. Semantic Manipulation can be achieved by providing the Semantic Value obtained from Semantic Measuring along with the candidate program(s) to a Semantic Evolutionary Operator. In [24], existing Semantic Operators are divided in 3 broad categories: Diversity methods, indirect semantic methods and direct semantic methods.

- **Diversity methods** refer to manipulations of semantic diversity, mostly done at population level.
- **Indirect semantic methods** still work by altering individuals syntax and semantic use is restricted as a criteria to decide on individuals survivability or other forms of indirect decision making.
- **Direct semantic methods** operate on candidate programs semantics directly, including voluntarily attempting to shape their behavior.

As an example of a semantic operator against non-semantic, consider the case below. Starting with an hypothetical semantic unaware mutation operator, which could easily be found operating on GP:

Given a syntactically correct parse tree, operator selects a random leaf node and swap its terminal by another terminal, also randomly selected, from the pool of terminals.

To make the operator semantically aware, there could be an extension to this operator:

...after performing the swap, the new program semantic value is calculated. Using both programs semantic values, semantic distance between them is calculated. If the distance is larger than a given threshold, the mutation starts over with the original element and swaps again. The process is repeated until an element is obtained which semantic distance to original is shorter than the defined threshold.

In other words, this operator would perform mutation while guaranteeing the elements are not completely apart in terms of semantics. Basically enforcing new variation to be "small". It would classify as an Indirect semantic method as it operates on the element's syntax and the decision process influences the survival of elements to generate its result.

2.6.5 State of the art

In [24], the authors provide a survey of existing research and semantic operators for SGP. Two particular points in time are highlighted as major turns in Semantic GP investigation: The proposal of several new Indirect and Diversity semantic operators and the first contribution of a direct semantic method by [21].
It's referred that up to the time of writing the survey, the only concern shown by existing direct methods was the semantic landscape geometry, referring to Geometric Semantic Genetic Programming which we describe in next section. Altogether, Indirect methods are said to be more mature, given they have been around for longer and have been more deeply studied. Consequently, Indirect methods have so far a more important impact on applications. Direct methods potential is still being discovered. Another important point suggested is that some SGP methods, given the benefits they bring to performance and lack of disadvantages, could become standard in GP packages.
Semantic Genetic Programming, although different in the sense that it introduces semantic awareness and manipulation, serves the same purpose as Genetic Programming, and so, has similar applicability. SGP can improve performance over GP and this performance improve can make the usage of (S)GP more competitive against other existing approaches aiming to

tackle the same problems.

In [26], Semantic Genetic Programming along with the authors developed Root Genetic Programming is used to perform Sentiment Analysis, a branch of text mining. The authors manage to outperform some other state of the art classifiers which are more commonly used to approach this problem. While SGP wasn't able to outperform one of the classifiers, the authors mention it was by a small margin with SGP still being easier to implement.

2.7 Geometry on Genetic Programming

Previously in this work, it's covered why and how semantics can be Measured and Manipulate in the context of GP, referred to as SGP. It's also covered how operators can be created to take advantage of semantics. In this section we cover how recent investigation is using search space geometry to design more efficient semantic operators that improve evolution performance. Before describing how geometry can be used to design operators, some preliminary concepts related to space search and geometry need to be covered:

2.7.1 Search Problem

Given its purpose and the way it operates, GP fits into the search problem category. Actually, sub-parts of the algorithm can by themselves be considered search problems. Consider the main problem of optimization GP tries to solve by generating and evolving candidate solutions but also sub-tasks in the middle of this process performed by some operators (depending on their nature). Formally, and based on the description in [27], a Search Problem can be described as follows:

Given a Search Problem P, a solution set S represents all possible candidate solutions to P. S is assumed to be finite and composed by formal solutions. P's goal is to maximize (or minimize) an objective function:

$$g : S \to R \tag{2.20}$$

Assuming P's objective to be *maximize* g, the *global optima* x^* represents points in S where g is a *maximum* which are collected in set S^* and where:

$$x^* \in S^* \iff g(x^*) = \max_{x \in S} g(x). \tag{2.21}$$

global optima is relative to objective function in terms of definition. A well-defined *objective function* will correspond to a well-defined *global-optima*. While *global optima* is independent of S's structure, it is not the case for *local optima*. There is no pre-defined structure over the solution set imposed by the search problem itself.

2.7.2 Search Space

A *metric space* is an ordered pair (M, d), with M being a set provided with a metric or distance d that is a real-valued map on $M \times M$, such that for any $x, y, z \in M$ the following axioms hold: [28]

$$d(x, y) >= 0, \text{ non-negativity property;} \tag{2.22}$$

$$d(x, y) = 0 \iff x = y, \text{ identity of indiscernible property;} \tag{2.23}$$

$$d(x, y) = d(y, x), \text{ symmetry property;} \tag{2.24}$$

$$d(x, z) <= d(x, y) + d(y, z), \text{ triangle inequality;} \tag{2.25}$$

2.7.3 Geometric Distance

The notion of distance varies and redefines according to the geometric environment being considered [27]. This is especially important in the context of GP as problems being faced require different types of chromosome representations that fit and span throughout different geometric environments. In Euclidean geometry, distance between two points $p1$ and $p2$ with

respective coordinates x_{p1} and x_{p2} in a plane is given by:

$$d(p1, p2) = \sqrt{(x_{p1} - x_{p2})^2 + (y_{p1} - y_{p2})^2} \tag{2.26}$$

This formula represents the shortest path between two points in the Euclidean space. If we change the nature of our geometric environment, the notion of distance naturally follows. Another classical example of a distance is Manhattan's. This distance is applied when Taxicab geometry is considered. Taxicab geometry uses the city of Manhattan as a metaphor. In some parts of the city, taxicabs movement is limited by the displacement of city blocks, resulting in movement that is exclusively horizontal or vertical. This kind of geometry is also commonly found in graphical computation where many algorithms are designed to perform on screens which are typically composed by grids of pixels. Manhattan distance is given by:

$$d(p1, p2) = |x_{p1} - x_{p2}| + |y_{p1} - y_{p2}| \tag{2.27}$$

Both Manhattan and Euclidean distance formulas are particular cases (simplifications) of the Minkowski distance formula in which parameter p is respectively replaced with values 1 or 2:

$$d(p1, p2) = (\sum_{i=0}^{n-1} |x_{p1} - x_{p2}|^p)^{1/p} \tag{2.28}$$

Another example, the Hamming Distance, represents the number of positions in which characters or symbols differ between two strings of equal length. This concept is frequently used in fields like cryptography, coding theory and computer science [29].

2.7.4 Abstract shapes

When a geometric space is not defined, it is impossible to describe distance in objective terms. There is no context. It is when a specific geometric space is considered that, within the context of that geometric space, distance becomes objectively describable and becomes possible to measure and calculate. Therefore, it is possible to say distance is an abstract concept that relies on context to be objective and measurable. The same thing happens

to geometric shapes. Only when a specific geometric space is defined can segments, circles, parabolas, etc, be objectively described and measured. Shapes are therefore abstract concepts in the same way that distance is. Nevertheless, it is possible to describe abstract shapes based on other abstract concepts like distance and other abstract shapes. In other words, it is possible to describe shapes independently from the metric employed. These *abstract shapes*, studied in *metric geometry*, are defined based on axioms with specific properties independent of the metric [27].

2.7.5 Balls and Segments

Two particular *abstract shapes*, the Ball and the Segment, are very adequate to describe mutation and crossover, respectively, and thus interesting to the context of this investigation. With r being a real number defining the radius and c the central point, a *closed and filled ball* defined in the metric space (M, d) is given by:

$$B_d(c; r) = \{p \in M | d(c, p) \leq r\}, \tag{2.29}$$

The line segment defined in metric space (M, d) is given by:

$$[p_1; p_2]_d = \{p_3 \in M | d(p_1, p_3) + d(p_2, p_3) = d(p_1, p_2)\}, \tag{2.30}$$

p_1 and p_2 are the extreme points of the segment that respect symmetry property, $[p_1; p_2]_d = [p_2; p_1]_d$. The distance between the pair of extremes is called the *length* and given by $l([p_1; p_2]_d) = d(p_1, p_2)$ [27].

In figure 2.19 there are representations of balls and segments on the Hamming, Eucledian and Manhattan spaces

2.7.6 Geometric Mutation

Geometric mutation fits the abstract ball shape and can be described the following way:

- A mutation operator is said to be geometric when in a metric space (M, d), all the

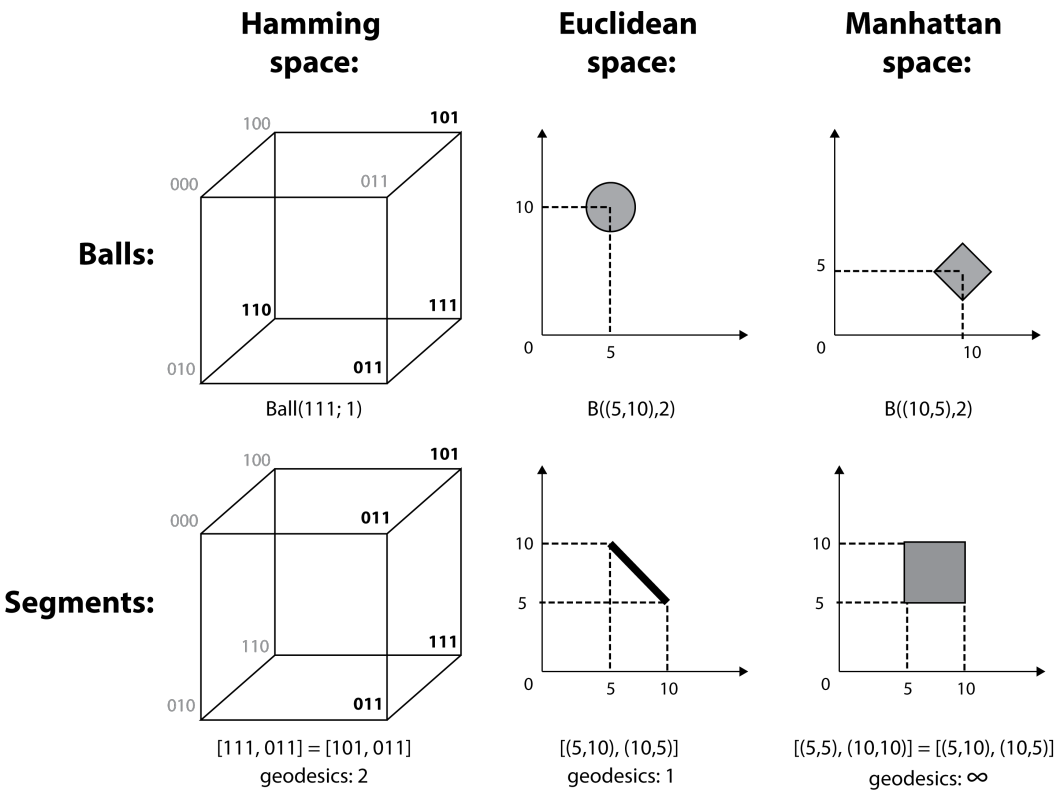


Figure 2.19: Balls and Segments on Hamming, Eucledian and Manhattan spaces. Based on the examples found on [27].

derived modifications from an initial element are inside the ball $B_d(c; r)$, where c corresponds to the original element and r to a specified radius.

Figure 2.20 shows an example of geometric mutation area represented by a ball over bi-dimensional Euclidean space. Point $p1$ forms the center of a ball with *radius* r equal to 2. This point represents the original element being mutated. In this case, being on a bi-dimensional Euclidean space, the ball is a circle. The closed space delimited by the circle represents the area in which points are correct geometric mutations. To exemplify, point $p2$ could be a possible result from mutating $p1$ while $p3$, being outside, could not.

Figure 2.21 shows a different example. This time the considered geometry is Taxicab's. The distance metric implied by Taxicab geometry is the Manhattan distance, given by function $M()$. The original element is represented by point $p1$ and radius is 3. From the image we can observe the naturally formed ball around $p1$. Curiously, the ball resembles an Euclidean square, although being distinct abstract shapes. Point $p2$, being "inside" the ball, is a

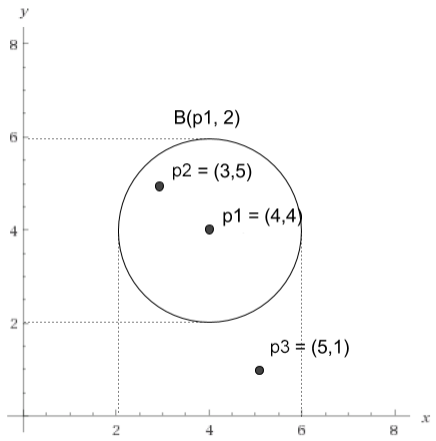


Figure 2.20: Geometric mutation represented by a ball over a bi-dimensional Euclidean space.

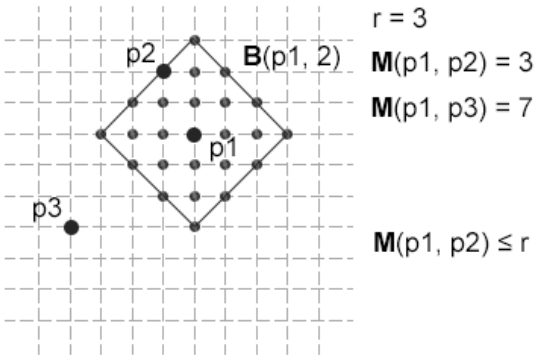


Figure 2.21: Geometric mutation represented by a ball over Manhattan space.

possible result from mutating $p1$ with a geometric mutation operator. Point $p3$, on the other hand, falls outside the ball and doesn't qualify as geometric mutation.

2.7.7 Geometric Crossover

Geometric crossover fits the Segment abstract shape. It is described as follows:

- A crossover operator is said to be geometric when in a metric space (M, d), all the generated children coincide with points in the segment to which parents are the extremities.

Figure 2.22 shows the example of geometric crossover over a bi-dimensional Euclidean space. Point $p1$ and $p2$ represent the positioning of parent elements elected for crossover. Connecting

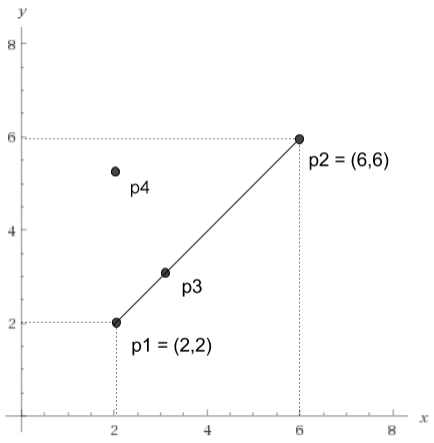


Figure 2.22: Geometric crossover represented by a segment over a bi-dimensional Euclidean space.

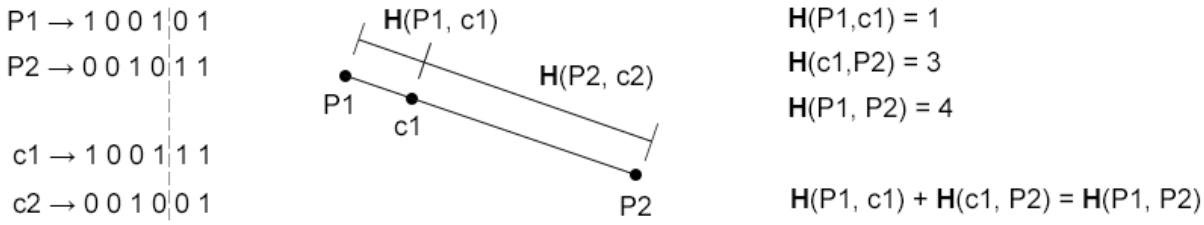


Figure 2.23: Geometric crossover represented by a segment over Hamming space.

these two points is a segment, which in the case of bi-dimensional Euclidean space is a bi-dimensional line. All the points which coincide with the line are possible geometric children resulting from a geometric crossover. Point $p3$ is an example of a possible children, while $p4$, being outside the line, is not.

Figure 2.23 shows another crossover example, but this time considering Hamming geometry. To parents P1 and P2 correspond the fixed length binary strings displayed on the right. Bellow c1 and c2 are children generated using a typical crossover operator. $H()$ is a function that calculates Hamming distance between two points. The operator is geometric as the children produced respect 2.30, recalling, the Segment equation given by distance.

2.7.8 Importance of the Geometric approach

From the concepts observed so far, some notions can be highlighted:

- The Geometric approach works over GP at an abstract level, in other words, the theory

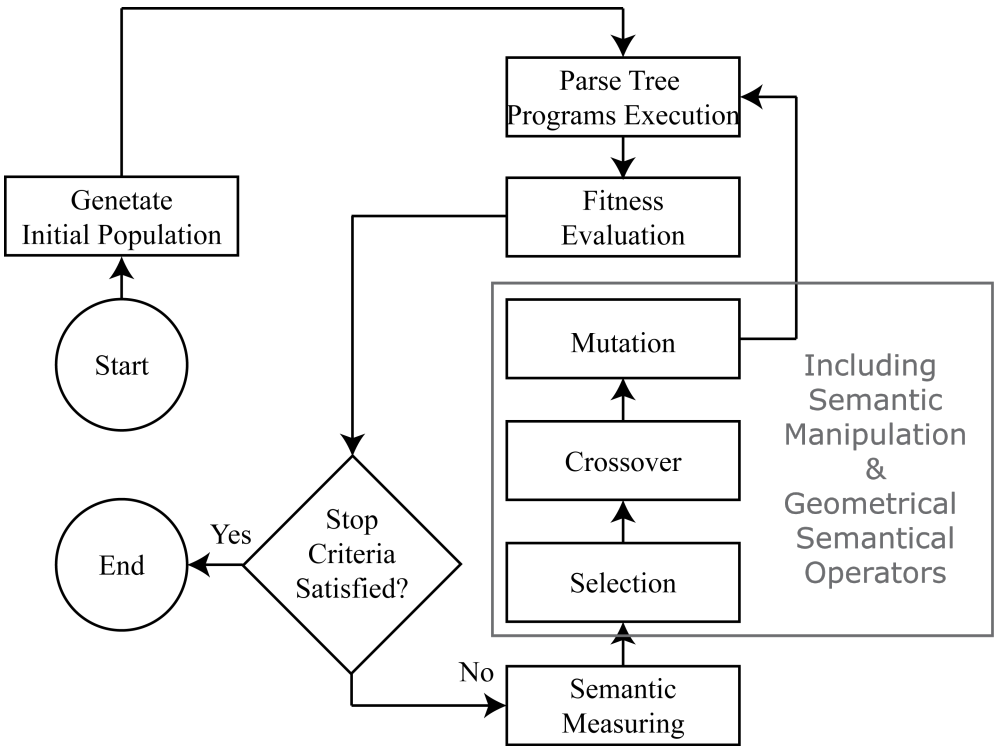


Figure 2.24: Geometric Semantic Genetic Programming flowchart. This algorithm differs from SGP by having at least one of its operators using a geometric semantic based strategy.

is independent of representation, and so can be applied independently of the problem being addressed.

- Under this approach, Crossover and Mutation operate in a simple abstract landscape with generalized guidelines for operator design.

2.7.9 Geometric Semantic Genetic Programming (GSGP)

Recall how semantic operators in GP guarantee that modifications occur respecting some criteria based on the semantics of involved programs. Also recall the roles of fitness value and semantic value in SGP. With the introduction of geometry to the evolutionary process, new relationships between involved concepts can be established. Figure 2.24 portraits GSGP evolutionary process.

Fitness and distance

Fitness can now be measured as the distance between a program's output and the defined target output. If more than one test is used then fitness is given by the distance between output vector and target output vector. The distance function must be adequate for the geometric space employed.

Semantics and distance

The difference in programs semantics can now be seen as the distance between their output vectors, hence called semantic distance. Similarly to fitness, the distance metric corresponds to the geometric environment adequately selected for the chosen chromosome representations.

Semantic Geometric Operators

Semantic Geometric Operators are operators that consider the semantic distance between points and elements in the semantic space in their decision making process.

Crossover Landscape Shape

Demonstrated in [21], when semantic crossover operators employ geometry, the semantic fitness landscape, derived from the definition of semantic distance, always assumes a particularly convenient shape: a cone. Visualized by the evolutionary algorithm and operator, this shape happens to be really easy to search and optimize as the global optima corresponds to the tip of the cone. The distance from the element to the tip corresponds to the element's fitness value.

2.7.10 KLX

In [5] the authors present an indirect approximately geometric semantic crossover operator. This operator is commonly referred to as the Karwiec Lichocki Crossover (KLX). The idea behind this operator is exploiting the conical semantic-fitness landscape to obtain the best possible fitness-distance correlation. Like the geometric theory which it bases itself upon,

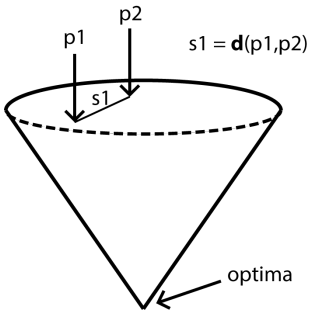


Figure 2.25: Fitness landscape viewed by GP/GEP when employing geometric operators. $p1$ and $p2$ represent examples of elements and segment $s1$ their semantic distance. The tip of the cone marks the semantic behavior correspondent to best attainable fitness.

this operator's applicability is independent of the problem being addressed and chromosome representation used.

In other words, knowing parents position in the semantic space, knowing that semantically similar children to parents are the ones lying closer or in the segment between them (parents) on the fitness-semantic space and knowing that the shape formed by the relation between fitness-semantics is a cone, the operator attempts to obtain children as closer while as equidistant to both parents as possible.

The ideal spot for the perfect geometric children is the exact middle point between the segment that separates both parents. The algorithm tries to obtain offspring which are as close as possible to this spot. In practice, this is achieved by electing one of children out of an offspring pool C that minimizes the following expression:

$$\overbrace{d(\mathbf{s}(p1),\mathbf{s}(c)) + d(\mathbf{s}(c),\mathbf{s}(p2))}^{\textbf{criteria1}} + \overbrace{|d(\mathbf{s}(p1),\mathbf{s}(c)) - d(\mathbf{s}(c),\mathbf{s}(p2))|}^{\textbf{criteria2}} \tag{2.31}$$

With $s()$ giving element's semantic value and $d()$ the distance, $p1$ and $p2$ being the parent elements and c being the child from pool C currently under evaluation. The expression corresponds to a sum of two individual parcels, each representing one of the two crucial criteria:

- **criteria1:** the sum of distances to both parents
- **criteria 2:** penalty for equidistance (distance to one of the parents being larger than

to the other).

In the experiment conducted in [5], the operator doesn't perform worse than standard GP. While the obtained fitness for the experiments is better, when set on fair computational conditions the operator does not outperform GP. KLX is extremely semantic oriented and ignores syntax which makes it a good representation of the geometric semantic approach. It has been used as control and in group tests of operators to represent the semantic geometric approach. In this work we will use an adaptation of this operator along with our implementation of GEP/SGEP.

2.7.11 Brood Selection Fitness (BSF)

For control and comparison purposes we introduce and use in the experiments a crossover we name Brood Selection Fitness. BSF is similar in principle and implementation to KLX. It makes use of brood selection in the same way, but instead of using the programs semantic value for decision making and candidate program comparison, it uses fitness.

3. Automatic Synthesis of Sorting Algorithms by Gene Expression Programming

This chapter refers to this thesis's Contribution 1. A study of its own and part of the group of bonded contributions.

To the best of our knowledge, while GEP has been successfully applied to address many problems, it has never been applied to the generation of Sorting Algorithms. Attempting this experiment with GEP is interesting in two distinct ways. First, solving this problem has already been done using GP. GEP is an alternative with a different nature, pros and cons to GP. Comparing both approaches when being applied to a common problem can add empiric insight on how they position relatively to each other. Secondly, GEP's evolutionary process is different in many aspects to that of GP (section 2.5). Observing how this difference reflects on generated solutions to this problem can be particularly curious. Sorting algorithms have been extensively studied in computer science and many different methods and variations exist. Creating or artificially generating a new variation of sorting algorithm is unlikely, but nevertheless an exciting (perhaps remote) possibility when trying this problem for the first time with a stochastic optimizer that often produces solutions that deviate from conventional human thought process.

To perform a fair experiment it was necessary to create as similar as possible contextual conditions for GP and GEP in order to isolate the algorithms semantics as the distinctive factor separating the results. As mentioned, synthesizing sorters has already been performed

with GP. [1] is a detailed investigation on how sorters can be generated with GP. We used the practical results from this experiment as control to conduct our own experiment. The experiment we propose consists in replicating the contextual conditions as best as possible, with the difference that the algorithm employed was GEP instead of GP. To perform the experiment and to match GEP's implementation to what was done in [1], a series of steps were taken.

Recall that GEP and GP implementations are problem independent algorithms, however, they require some problem specific methods to be provided, namely a set of functions and terminals, and a relevant fitness function. In the next sections we describe the experiment's conditions before presenting the results and discussion.

3.1 Functions and Terminals - the building blocks of a sorter

In GP and GEP, the approach to obtain a final solution involves creating an effective way to measure and classify a candidate program in its ability to solve the problem. Equally important is defining the elementary constituents that will compose each of the population's programs. Failing to provide adequate elementary instructions will hamper and even impossibilitate the evolution of adequate final solutions. In [1], interesting deliberation is put into developing a set of Terminals and Functions capable of providing adequate building blocks to evolve successful sorting algorithms. The sets were created observing the most common operations from comparative and swapping types of sorters. In swapping sorters, values in the sequence are never actually altered, instead, existing values swap positions among themselves in order to sort the sequence. This is arguably the most intuitive and conventionally used type of sorting algorithm with instances including Bubble-sort, Quick-sort and Bucket-sort. GP and GEP using the following set of terminals and functions will naturally inherit some characteristics associated with Swapping types of sorters considering their elementary constituents. The Terminals and Functions are the following:

3.1.1 len (Terminal)

`len` corresponds to the length of the sequence provided for sorting. This value is maintained throughout program execution.

3.1.2 index (Terminal)

`index` value is manipulated by `dobl` functions and assume the iteration index value. Before terminating execution, `dobl` reverts `index` to the value it had when the `dobl` intruction began executing. This strategy allows blocks of `dobl` instructions to behave like nested loops, which are extremely important to evaluate and manipulate sequences.

3.1.3 dobl (Function)

`dobl(start, end, work)` is an iterative function with 3 parameters. When executing, it loops from `start` to `end` doing the operation specified in `work`. `index` terminal is assigned the value of `start` and is incremented each iteration until either `end` or `len` is reached. After executing and returning `index` to its original value, `dobl` returns the minimum value between `len` and `end`.

3.1.4 swap (Function)

`swap(x, y)` swaps value in sequence's position `x` with the value in position `y`. Returns the value of `x`.

3.1.5 wibigger (Function) and wismaller (Function)

`wibigger(x, y)`, and `wismaller(x, y)`, are comparison functions. In both functions, the values in positions `x` and `y` in the sequence are compared. In `wibigger` case, the index of larger argument is returned, while in `wismaller` is the index of the smaller argument.

3.1.6 e1p (Function), e1m (Function) and em (Function)

`e1p(x)` increments and returns the value obtained from executing `x`. `e1m(x)` decrements and returns the value obtained from executing `x`. `em(x)` returns (-1) times the value obtained from executing `x`.

3.2 Fitness Function

For this problem the fitness function needs to accurately classify candidate programs in their ability to sort provided sequences in a continuous way. In [1] the strategy adopted was to measure the disorder of a sequence before and after execution. The difference between initial and final disorder represents the ability of a program to sort the given sequence. Program fitness is obtained by evaluating 15 or more sequences (depending on the Test-Set). Since the fitness is based on several sample tests, we can classify this fitness function as one using a sampling approach.

Neighbor Inversions

To implement the fitness function based on mentioned description we resorted to the notion of neighbor inversion which is based on the normal notion of inversion [10]:

Let $(a_1, a_2, \ldots, a_n)$ be a permutation of the sequence $(1, 2, \ldots, n)$. The pair (a_i, a_{i+1}) is called a neighbor inversion of the permutation if and only if $i < i + 1$ and $a_i > a_{i+1}$. Neighbor inversion pairs represent pairs of consecutive sequence elements that are out-of-order. A sequence can be classified as a completely sorted sequence when it is a permutation with no neighbor inversions.

Formal definition

The fitness function f is given by the following formula:

$$f = \frac{1}{S} \sum_{i=1}^{S} \frac{dd(i)}{Mdd(i)}, \tag{3.1}$$

S is the number of sequences to test the candidate programs with. $dd()$ is the difference in disorder and $Mdd()$ is the maximum possible difference in disorder. These functions are given by (3.2) and (3.3) respectively. Notice that $0 \leq f \leq 1$.

$$dd(i) = nt(d(s_{original}(i)) - d(s_{result}(i))), \tag{3.2}$$

$nt(x)$ stands for *negative threshold*, returning 0 if $x \leq 0$, otherwise returns x. Given by 3.4, $d()$ computes the number of sorted pairs. $s_{original}$ and s_{result} represent the sequence being evaluated by the i-th candidate program prior and post execution, respectively.

$$Mdd(i) = l(s_{original}(i)) - 1 - d(s_{original}(i)) \tag{3.3}$$

$l(s)$ gives the length of sequence s while this value minus 1 is the number of distinct consecutive pairs in the sequence. Thus, $l(s_{original}) - 1$ is the number of pairs on the original sequence.

$$d(s) = \sum_{k=1}^{l} int(s_k \leq s_{k+1}) \tag{3.4}$$

s_k and s_{k+1} being the values at position k and $k+1$ of sequence s. $int()$ represents a type cast from boolean. $int(x)$ returns 1 when x is true and 0 otherwise.

3.3 Evolutionary Parameters

This section describes GEP parameter configurations for this experiment. Table 3.1 shows parameters which exist on both GEP and GP. These parameter were set to be equal to the ones employed in [1]. Assigning the same values to these parameters guarantee evolution will be configured with similar starting conditions, isolating each of the algorithms individual characteristics, and so increasing fairness of comparison. Multi-gene GEP, as described in has exclusive parameters, *head size* and *number of Genes* and exclusive operators, *Transposition of insertion sequence of elements (IS)*, *Root transposition (RIS)*, *Gene transposition* and *Gene Recombination*. GEP exclusive operators frequency of activation and parameters

GEP parameters matching GP's	
Parameter	Value
Population-size	1000
Selection	Tournament Selection Pool Size 2
Probability of mutation	0.01
Probability of crossover	0.8
Replacement factor	0.95
Maximum number of generations (1)	49
GEP exclusive parameters	
Parameter	Value
Head Size	16
Number of Genes	4
Probability of IS	0.1
Probability of RIS	0.1
Probability of Gene Transposition	0.1
Probability of Gene Recombination	same as probability of crossover

Table 3.1: Experiment 1 Parameters

are described separately in the lower part of the table. These values were based on recommendations proposed on [4] and some empirical intuition built after running the algorithm a few times and observing the results. Note that, for this experiment, GEP exclusive operators are assumed to be part of the algorithm's way of operating and part of its nature. For these reason, exclusive operators were employed alongside other conventional operators while considering comparison conditions to be similar nonetheless.

3.3.1 Headsize and number of genes discussion

Intuition obtained from individual (or very few) runs of a genetic algorithm is hardly significant to justify the decision to fine tune a genetic parameter and, given the stochastic nature of GEP, probably misleading. This is actually expressed as one of the conclusions in [1] for the case of GP when trying to generate sorters. Statistic observation from larger batch of runs is the correct way to decide on the adjustments to perform on parameters. Such statistic studies come at the cost of computational resources and time expended performing the tests. Performing these tests can, from my perspective, be seen as an investment for

the refinement of the obtained program solutions and give a statistic clue on whether the configuration being employed reaches the algorithms potential to solve the problem at hands. In practical situations, the decision is probably not only as whether or not these tests should be done, but to what extent. It must be considered what parameters should be studied and what range and precision of each parameter should be tested. Would a change of a certain range to a particular parameter produce any considerable change to the obtained solutions? At this point, some theoretical knowledge on the role of each parameter is probably of greater importance, as it can *a priory* provide an idea of to what extend can the calibration of a parameter affect the quality of the results and whether or not is it worth to invest effort in tweaking the parameter.

GEP introduces two extra parameters, headsize and number of genes. Having two extra parameters, when compared to GP, means extra effort is required in configuration that must be considered when pondering whether GEP is the right algorithm to deal with a given problem.

While in Experiment 1 headsize and number of genes were not obtained using an experimental batch of tests and still produced promising results, in Experiment 2, for some of the sub-experiments, not all the results were this promising. On the latest case, one is left wondering if optimizing these two GEP parameters would make a difference in the obtained results. While our work provides an overview of GEP's performance dealing with several problems without optimizing these parameters, it must be considered as a bottom threshold for GEP's potential when handling the respective benchmarks. Future work might do parameter optimization and investigate the difference.

3.3.2 Headsize and number of genes employed

A few initial evolutionary runs were made with experimental values of 12 for headsize and 3 for number of genes. After adapting the values for the sake of experimentation, with parameters 16 and 4 respectively, the implementation obtained converging programs with surprisingly high frequency. Eventually, we ran a complete set of runs (20) with these same parameters for the first test-set (A) and the obtained results already promising enough that no further tweaks seemed necessary.

Headsize defines the number of terminals and functions that compose a karva-expression head and, derived from it, tail size and complete expression length (formula 2.5 on section 2.5.2). With the head size of 16 and assuming the maximum arity of 3 given by the employment of Function *dobl*(), genes all have a tail of size of 33. This also caps the number of nodes the corresponding parse trees can have to no more than 49. The number of genes, 4, means that each karva-expression is composed by 4 parse trees of maximum-size 49, 1 active at a time.

3.3.3 Termination criteria

Evolution process was configured to stop at convergence. Convergence is characterized by evolving an element which attains the best fitness possible. In this case, convergence would be correctly sorting all proposed test sequences. To prevent the evolution process from running for long periods, or even indefinitely, a threshold sets the maximum number of generations before a run stops. If at generation 49 no converging element is found, the run terminates and the fittest element found up to that generation is returned.

3.4 Test-Sets

5 distinct test-sets compose the experiment, named from A to E alphabetically. Between each test-set, the sequences used to evaluate fitness vary in number and maximum length. Maximum sequence length is maintained through the entire evolution run. The test-sequences are randomly generated and its number of elements can be equal or smaller than this maximum-length. In all test-sets but one, new sequences are randomly-generated at each iteration. The programs facing the test-sets with new sequences are expected to become more general. In these cases, The programs will be pressured to be able to sort sequences in a more abstract way and not be able to specialize in particular sequences as easily. Table 3.2 display each test-set parameters.

Test-set	M	l	New sequences each gen.
A	15	30	yes
B	15	30	yes
C	25	30	yes
D	15	50	yes
E	15	30	no

Table 3.2: Test-Sets for Experiment 1 as defined in [1]. For fitness evaluation purposes, M indicates the number of testing sequences, l maximum sequence length and "New sequences each gen." whether new sequences are generated and replaced previous ones as population iterates in generations. If this parameter is defined as "no", sequences are generated once for the original population and reused for all subsequent generations.

3.5 Steady State Genetic Programming

In [1], SSGP is used instead of conventional GP. Steady State concept was first introduced to conventional Genetic Algorithms in [30] and later introduced to GP in [31]. As mentioned in [31], "Steady State" in GA refers to the use of a "continuous" unique pool as population. Elements leave the pool as necessary to give way to new selected elements. The exchange of elements in the pool responds to crossover and mutation, although the concept seems extensible to other kinds of operators and variations that would need to manipulate the pool. The pool must be kept free of duplicates. Apart from removing the need for new population synchronization at every generation, some evidence in Steady State performance benefits can be find in [32] and [33].

3.6 Generating sorters with SSGP and results from [1]

Before presenting the results obtained with GEP, in this section we describe the part of the results obtained in [1] using SSGP which are relevant for this comparison. 20 runs of each test-set were performed. The resulting table display a count of how many of the runs for each test-set generated at least one converging program (i.e. a program that sorts all the proposed test-sequences and obtains best fitness possible). Other measures of success are recorded on columns '90%' and '75%'. These columns show how many of the runs generated a program which removes the respective percentage of disorder. Further generality tests are

performed in the experiment. Column # GEN RUNS (#GR) displays out of the 20 runs how many found a converging program that also solved extra generality tests. Finally, column AVG INDS 100% (AI100%) shows the average number of elements that had to be handled for runs which generated converging programs.

					# Successful runs (out of 20)			
Test Set	Significant Feature	M	l	#GR	100%	> 90%	> 75%	AI100%
A	baseline	15	30	4	12	12	12	25250
B	baseline	15	30	3	10	11	13	23095
C	more tests	25	30	5	8	8	9	20451
D	longer tests	15	50	5	8	9	10	14830
E	no new tests each generation	15	30	3	5	8	8	7173

Table 3.3: Results of application of Steady State Genetic Programming to the evolution of sorting algorithms as presented in [1], Table 1.

Further details on how the results on table 3.3 were obtained can be found in [1]. Of particular interest is the description of the thought process for the setup of all conditions necessary to successfully generate sorting algorithms using GP.

3.7 Results

Table 3.4 shows equivalent results to table 3.3 using GEP instead of SSGP. Experiment conditions were made to be as similar as possible to what was described in the previous sections. GEP's algorithm, besides 3 different exclusive operators also has 2 extra parameters referring to the size and composition of its expressions. These parameters were not accurately optimized, we simply used standard and empirically tested values. This means that GEP performs "at least" as good as results in table 3.4, but with optimization could perform possibly better.

3.8 Discussion

GEP converged considerably more often than GP. For baseline Test-Sets A and B, GEP excelled at the first by converging in all 20 runs and missed by 1 the second. SSGP scored

					# Successful runs (out of 20)			
Test Set	Significant Feature	M	l	#GR	100%	> 90%	> 75%	AI100%
A	baseline	15	30	4	20	20	20	20000
B	baseline	15	30	3	19	19	19	17000
C	more tests	25	30	5	20	20	20	20000
D	longer tests	15	50	1	18	19	19	21000
E	no new tests each generation	15	30	3	15	19	20	15000

Table 3.4: Results of application of Gene Expression Programming to the evolution of sorting algorithms

slightly more than half, with 12 and 10 convergences respectively. A very expressive advantage for GEP.

For test-set C, when more tests per evaluation were added, GEP scored perfectly again, which probably means the extra evaluations either guided the evolution more generally or simply didn't provided enough of an extra challenge to reduce performance. Maybe both. In contrast, SSGP seemed to struggle slightly more by obtaining a worst result than with the baseline Test-Sets.

On test-set D, with longer tests GEP sees a slightly worst performance that for the previous with 18 convergences. Nevertheless an excellent result. The difference is however not expressive enough to confidently suggest longer tests might create extra struggle for the algorithm. When comparing to SSGP however, we can still see GEP notoriously stands out with, again, more than twice the convergences. This longer tests generated less General Run convergences than previous test-sets and than SSGP.

Finally for test-set E, where tests were repeated through iterations, the results were notoriously less convergent for both algorithms. This confirms the importance varying challenges can have when it comes to both these algorithms. GEP still converged 3 times more than SSGP.

In terms of generality runs, the number of convergences was similar except for the test-set with longer tests, where GEP scored considerably worst than SSGP. The results in this columns are referent to the 100% column, so considering GEP converged more, the generality results become even worst: out of 8 convergences, 5 passed generality tests, while for GEP, out of 18, 1 passed generality tests. There are several possible explanations and

factors possibly involved. GEP bloat limitation preventing the necessary number of elements to create a more general structure, which is related to head size parameter, might be one explanation, although this goes against the inverse-proportionality relation between generality and program size observation from [1]. The stochastic nature of the experiment might be another factor and even by itself explain the reasons, although by the expressivity of the results, there is no indication that that is the case. Future work might investigate this phenomena and its causes.

Another interesting observation is that GEP has 4 times the number of parse trees being "carried" by each element, although each tree is most likely smaller than those in SSGP and not necessarily being ever considered/processed. Execution time recording wasn't performed in this experiment neither. Time comparison under similar conditions for both algorithms could be of interest, especially if a decision must be made on using one of the approaches for practical applications. Would be interesting to see future comparisons including this measurement.

3.9 Conclusion

In this experiment we over-viewed GEP as an alternative to GP. With coinciding applications but having different natures, each algorithm comes with its own characteristics. From previous studies, GEP is presented as being the better performer while introducing bloat control and eased implementation from guaranteeing always correct expressions by design.

GEP's bloat controlling mechanism, which is based on a user defined size for the elements of a particular evolution run, when incorrectly configured and depending on the problem and terminal/function set, can exclude desirable solutions. In other words, if a desirable solution exists that requires more elementary instructions than those possible to fit inside the limits set for the k-expression instance, than this solution is, a priory, out of reach for GEP's evolutionary process to find. On a different perspective, this same bloat control can be seen as evolutionary pressure to search for more compact or simplified solutions, which can be attained by intentionally deciding to exclude those too big. This situation results in GEP's head size parameter being a slight complication to the user, which must be aware

and consider it's possible effects. Developing ways to help setting this parameter, or possibly doing it automatically, is, in our perspective, an important topic for future investigation.
In terms of Sorting algorithms, we saw that previous work has investigated how these algorithms can be synthesized using GP. Notoriously in [1], a very complete study is done on the subject with very interesting practical results. To the best of our knowledge however, no attempts have been made to apply GEP into solving this problem. We replicated the conditions in this study and apply it to our own implementation of GEP in order to compare the two algorithms. In initial experiments, we observed that GEP is indeed capable of synthesizing sorting algorithms. Furthermore we verify that, under similar conditions and isolating the algorithm's nature, GEP obtained convergence in nearly all runs, and so, outperfors GP. The number of convergences is often more than twice of GP's so the advantage is really expressive. Execution time was not taken in consideration. From these observations, we have a new indication (to reaffirm existing studies with similar conclusions) that GEP's by design element structure characteristics work and manage to outperform its direct alternative (GP) in terms of convergence rate.

4. (Geometric) Semantic Gene Expression Programming

This chapter refers to Contribution 2 of this thesis. A study of its own and part of the group of bonded contributions.

In this study we investigated if the recent research made into semantic and geometric approaches to GP can be adapted and applied with positive results to GEP. The experiment will consist in running SGEP, the new proposed semantic aware variation of GEP, applied to several benchmarks from 3 different problems in order to provide an overview of how it performs and compares against settled alternatives, namely GP, conventional GEP and SGP. Before describing the experiments and all the steps taken towards their preparation, execution, results and discussion, the first sections cover the theory behind the implementation of SGEP and also present GSGEP (the geometric variation of SGEP).

4.1 Considering semantics in GEP: Why?

The original version of GP, although proven to work and having positive results when applied in practice, does not emulate evolution's genotype/phenotype roles properly, in the sense that elements semantic is ignored when it comes to passing on parent's characteristics or allowing slight modification through mutation. Semantic is demonstrated to be unintentionally and disproportionately altered in these scenarios. This problem has been addressed by the development of a dedicated algorithm which brings semantic consideration to GP: SGP. Furthermore, SGP opened the possibility to the development of the geometric ap-

proaches to semantic element handling which demonstrates considerable improvements over conventional SGP. Both SGP and its geometric enhanced variation show great potential backed by several performed studies. GEP is another variation of GP that aims to tackle a different set of GP problems. GEP guarantees bloat control and handles difficulties related to parse trees manipulation. It also includes additional characteristics, like multiple chromosomes and non-coding regions that have been shown to improve obtained results. In this study we introduce the Semantic Gene Expression Programming (SGEP) algorithm, which aims to combine in its evolutionary process all the positive characteristics that SGP and GEP demonstrate over GP.

4.2 Considering semantics in GEP: How?

In section 2.6.4 we saw that Semantic Measuring and Semantic Manipulation are the two necessary features needed to allow GP to consider program's semantic in its evolutionary process. SGP differs from GP by implementing these features in its evolutionary process. Our hypothesis, given GEP's algorithmic similarity to GP's, is that semantic measuring and manipulation can be added to GEP in a similar way and obtain the same desired characteristics in its evolutionary process. GEP employs a different data-structure to encode its elements, has an extra set of rules and consequent alterations in the evolutionary process. Below we describe why none of these differences prevents GEP's algorithm from getting semantic measuring and manipulation similarly to GP. However, in the case of Semantic Manipulation some adaptation might or might not be needed depending on the type of semantic operators one decides to include in the evolutionary process.

4.2.1 Semantic Measuring in GEP

In GP, Semantic Measuring can be done very similarly to fitness evaluation and often, depending on the fitness function and semantic function, can be done simultaneously. Usually, for Semantic Measuring, before selection, existing programs are ran against a set of tests. The vector of outputs produced by these tests is used as the program's semantic value.
This exact same process can be replicated in GEP without further adaptation. GEP pro-

grams can be presented to and run the same tests, in the same way, GP programs can. In GEP programs have an encoded form, but ultimately, can represent the exact same code structures given and identical set of functions and terminals. In GEP, the size of programs is limited differently, but not the method of executing the programs.

Adding semantic measure means that each element store an extra value (value not necessarily numeric, typically it can often be a vector of results) to represent its semantic state. Adding an extra parameter storing semantic value has no interference in the evolutionary run. Computational cost might be negligible or not depending on application conditions, the problem being addressed and the dimensions of vector of results and its data structure. In SGP's algorithmic flow, Semantic Measuring usually takes place immediately before, after or simultaneously to fitness evaluation. This is usually done consecutively for every element of the population before moving to a different stage of evolution. In some cases, operators may need to perform Semantic Measuring in different stages of evolution. This can happen, for example, in cases where programs are transformed or new programs generated within the context of the operator. In these cases, the new program's semantic value may need to be assessed for decision making in later stages of the operator and trigger immediate semantic measuring of these locally generated new programs.

Apart from these situations, GEP's algorithm is fundamentally similar to GP in the necessary mechanisms to incorporate Semantic Measuring in the same way it is done in GP.

4.2.2 Semantic Manipulation in GEP

Evolutionary operators are designed to intentionally manipulate the evolutionary process, usually with a specific desired effect in mind (e.g. preventing premature convergence of programs to local optima by increasing randomness of child if parents are too syntactically similar). In normal GP and GEP, information available to operators doesn't include program's semantics, neither regarding the population as whole nor of programs individually. When this information becomes available through the introduction of semantic measuring, new possibilities arise for operator design. In other words, it's now possible to use the semantic value of programs to create operators that make decisions and affect the evolutionary process in ways not previously possible.

Adapting operators

Existing semantic operators designed to work with SGP have been divided into three categories according to the way they employ semantic values: diversity control, indirect and direct (2.6.4) semantic operators. Depending on which of these three types of semantic operators is being considered, the differences between GP and GEP may or may not be an obstacle to the direct integration of operators from SGP into the semantic variation of GEP (SGEP).

One crucial difference between GEP and GP is the nature of its chromossomes. GP's chromossomes are parse trees. GEP chromossomes are karva-expressions, which ultimately are parse-trees encoded in a linear string of fixed length structure.

Semantic diversity operators do population control to manipulate semantic diversity and decide on individual's relevance. These operators work at population level. The elements themselves are usually not modified, simply excluded or replaced to compose the population differently.

Indirect semantic operators use semantics to decide on elements survivability and election at operator level. Indirect operators are mostly crossover and mutation operators. Modification if needed is done syntactically.

Both diversity and indirect types of operators can be directly integrated into GEP without adaptation, as their way of operation is not affected by GEP's distinct characteristics and fit similarly in the evolutionary process.

In the case of indirect operators, syntactic modification done at parse tree level is usually performed in a stochastic manner with resulting elements then filtered based on some criteria involving the elements semantic. It's consider that the election and choice of syntactically modified elements is where operator theory takes effect. For this kind of syntactical modification, it is assumed that parse tree modifications methods can be replaced by k-expression modification methods without changing the theoretical value or effect of the operator.

Direct semantic operators use semantic knowledge to perform direct manipulation and construction at the level of parse trees (reference [24] provides examples). Unlike the indirect and population control variations, existing instances of direct semantic operators might need

adaptation and reconsidering to integrate with GEP. Karva-expressions enforce size and structural limits on the encoded parse-trees. Considering these limits, when designed directly, some form of logic must be created to prevent resulting parse-trees from disrespecting Karva language enforced rules. In other words, it is necessary to guarantee that semantically engineered trees produced by direct operators can be converted back into correct k-expressions. Future work can investigate ways to adapt existing direct semantic operators to work with GEP/SGEP.

4.2.3 Semantic Gene Expression Programming (SGEP)

SGEP is what we chose to name a new type of algorithm that combines GEP with SGP. More than an algorithmic combination, SGEP intends to combine the advantages and characteristics of both GEP and SGP over conventional GP: Stronger semantic correlation on modification and reproduction between related population elements, always correct resulting chromosomes from modification and reproduction, non-coding regions capable of transporting hidden characteristics across generations and bloat control. From an algorithmic point of view, SGEP starts as a standard GEP's algorithm and is augmented with Semantic Measuring and Manipulation which are inserted in its evolutionary flow. Semantic Measuring is added as a step that can be performed along with fitness evaluation or using any strategy that manages to capture a representation of program's semantics in the same way it is done in SGP. Semantic Manipulation is achieved by employing a combination (or at least one) semantic operator besides or along conventional syntax only based operators. Figure 4.1 shows the workflow of SGEP.

4.2.4 New possibilities

Up to this point it was examined the possibility to integrate existing operators which may or may not need adaptation. Conventional direct semantic operators were designed to operate at parse tree level, so in terms of GEP, this would be done after decoding k-expressions. The design of new operators would have to make sure new operators can be converted back and still respect k-expression enforced length limits and head/tail proportions.

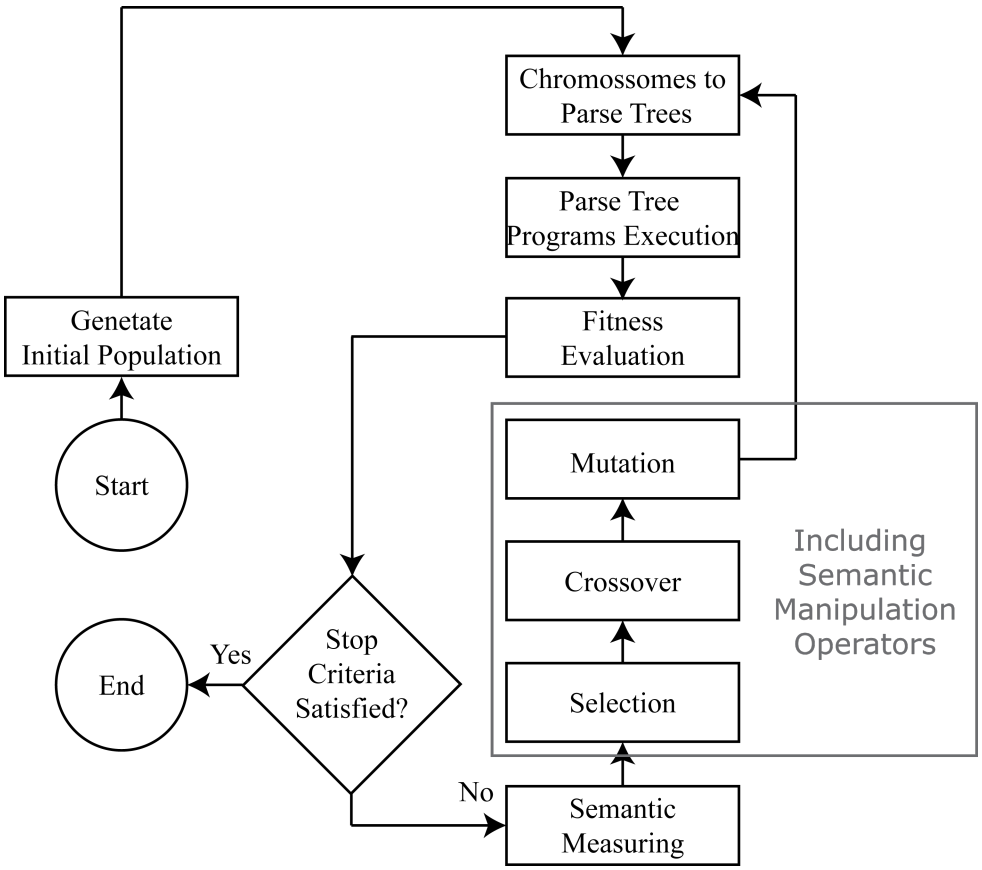


Figure 4.1: Semantic Gene Expression Programming flowchart. SGEP builds on the GEP's algorithm by adding a Semantic Measuring step and including semantic based operators

GEP evolutionary process, although sharing many similarities with GP, has also many characteristics of its own. From multi-genetic expressions to hidden sequences, bloat control and always correct parse trees when decoded from k-expressions. GEP has the possibility to bring building blocks and provide genetic variation at different stages of evolution in ways that are not easily achievable nor natural to conventional GP.

With the introduction of SGEP, new theoretical possibilities might now be available for exploration in terms of semantic operator design. From now on, it's possible to create dedicated semantic operators that take advantage of Karva language structure and SGEP's evolutionary algorithm to explore new strategies and to manipulate the evolutionary process in ways that were not possible up to this point.

4.2.5 SGEP operators employed in this experiment

In this contribution, experiments have a focus on the role of crossover operators. No mutation operators are used in order to better isolate crossover operator's role in results. Nonetheless, this shouldn't be taken as an indication that SGEP benefits exclusively crossover operators. SGEP opens the same possibilities for mutation operator design that it does for crossover (or other variety of operators). In principle, adapting or developing new mutation operators with SGEP in mind might benefit evolution and provide good practical results in the same way that this study demonstrates for crossover. Future work can investigate possible new mutation strategies based on semantic decisions to modify k-expressions.

4.3 Semantic geometry in SGEP

Semantic geometry uses understanding of the semantic space of programs to develop more efficient strategies for operator design. When considering SGEP, theory on geometric semantic operator design can be directly derived and used as it is in GSGP over SGP. This is possible due to program execution and semantic measuring working the exact same way for both algorithms.

One thing to consider however, are the practical consequences to the semantic space of program size limitation enforced by k-expressions. Limiting the size of programs also implies

limiting the universe of possible programs developed by a given combination of functions and terminals and therefore also possibly limiting the space of possible semantics.
The semantic-fitness landscape seen by an evolutionary algorithm for semantic geometric operators has been demonstrated to be cone shaped, with the volume including all possible program semantics and the tip being the optimum. Distance from the optimum to each program corresponds to the fitness value of said program. [21]. In the case of GEP, limiting the programs size might have two types of geometric consequences on the space of available solutions attainable: First, by excluding some of the semantics possible with limitless number of instructions, the cone becomes a sub-section of the initial shape, consequently limiting the volume of the new shape. Secondly, if placed in space, the achievable tip may shift place to a different point inside where the original cone was. This means the original optimal semantic-fitness solution might no longer be attainable with just a sub-set of the possible initial programs. Formalizing this concept and defining a size-limitation parameter which doesn't cause a shift on the position of the semantic-fitness landscape can be topics for future investigation.

4.3.1 Geometric Semantic Gene Expression Programming (GSGEP)

GSGEP is what we chose to name a new variation of SGEP where semantic geometry based operators are employed in the evolutionary process. GSGEP intends to bring to SGEP the advantages that studying and observing the semantic differences between programs as points in a geometric space brought to GSGP over SGP. In a Geometric Semantic environment, a distance can be defined between program's semantics and with the semantic space characterized to develop operators that more efficiently optimize program in an evolutionary system. Figure 4.2 displays GSGEP evolutionary process flowchart.

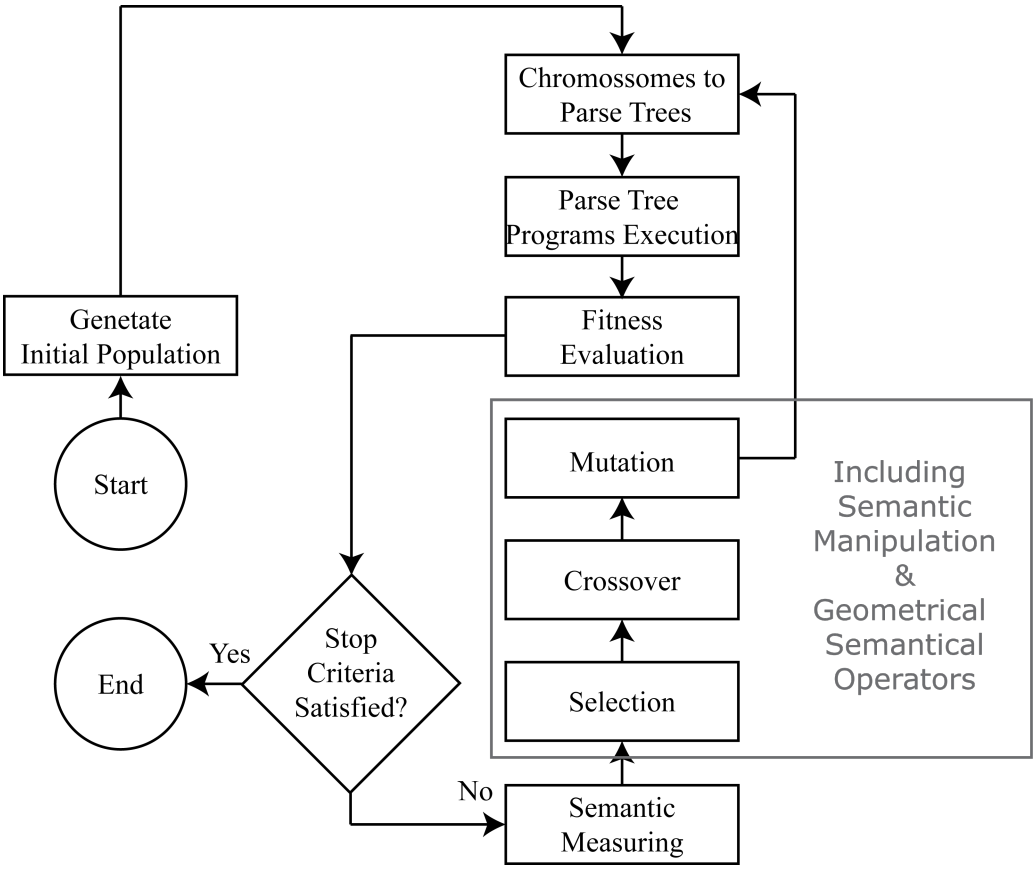


Figure 4.2: Geometric Semantic Gene Expression Programming flowchart. GSGEP works similarly to SGEP but employs geometric semantic operators which brings geometric semantic decision making possibilities to a GEP based evolutionary process.

4.4 Experiment conditions

In these set of experiments, we test the semantic aware version of GEP along the Approaching Geometric Semantic Crossover operator (KLX). This operator was picked due to being in the indirect class of semantic operators and therefore possible to be employed in GEP as is, without adaptation. The same GEP implementation employed in Experiment 1 was modified and Semantic Awareness was added as described in previous paragraphs. Semantic manipulation is performed by our implementation of KLX. To this algorithm we will refer to as an SGEP implementation. Since KLX operator is an approximated geometric crossover operator, the implementation could actually be referred to as GSGEP but we will still call it SGEP.

The experiments will consist in gathering data from GEP and SGEP solving 3 distinct problems, one of them with two variations: Symbolic Regression (Single Variable and Double Variable), Boolean Function Synthesis and finally Synthesis of Sorting Algorithms.

SGEP will be compared against SGP for a combination of the problems and against normal GEP for the complete set. This data should provide a first overview of the algorithms capabilities and potential. Several crossover operators are used throughout the experiments, these operators are described in the State of the Art chapter of this document. NoCX refers to an evolutionary process where no crossover operator was used at all.

These experiments were conducted in a quad core Intel i7, 2.00GHz, 4Gb of Ram laptop. Although I didn't perform any accurate time measurements, it took around 15 days to a month to run the entirety of the benchmarks. It's important to mention that during this time the experiments were performed intermittently and the computer was often in battery saving or low performance conditions.

4.4.1 SGEP and SGP comparison

Comparison between SGEP and SGP will be performed using problems Symbolic Regression (both variations) and Boolean Function Synthesis. As base for comparison we use results obtained in similar experiments performed with SGP. The experiment conditions were set to be as similar as possible to the ones employed in the control studies. Similarly to contribution

1, we consider GEP exclusive operators to be part of the algorithm's nature (SGEP's), and so, kept active and triggering with a certain probability.
Symbolic Regression (Single Variable) and Boolean Function Synthesis conditions, benchmarks and results used for this comparison are based on [34], a study comparing existing geometric semantic crossover operators. For Symbolic Regression (Double Variable) benchmarks and control results are based on [35], an investigation on GSGP which demonstrates the possibility of developing an alternative algorithm guaranteed to find optimal solutions in less than infinite time. In sections 4.4.3 and 4.4.4, parameters and benchmarks employed are described in detail.

4.4.2 SGEP and GEP comparison

To compare SGEP with GEP, both algorithms were ran under as similar as possible conditions. Results for both approaches were generated (no external control studies for this experiment). The comparison includes all 3 problems' and both variations of Symbolic Regression's benchmarks.
SGEP results are obtained using semantic crossover KLX. GEP results use non-semantic crossover operators 2P and BSF. The usage of semantics is All remainder conditions are kept identical in attempt to isolate usage of semantics in the decision making process as the differentiating factor in results.
Apart from isolating SGEP's, GEP's and GP's algorithms nature, this comparison also overlaps with isolating different crossover operators. Therefore, the results are also insightful on positioning each of the crossover operators employed relative to each other under the context of SGEP, GEP and GP.

4.4.3 Evolutionary Parameters

Evolutionary parameters used to configure GEP are described in table 4.1. Depending on the problem at hands and target algorithms being compared, the parameters were adapted to create conditions as similar as possible to the ones employed in the respective comparing set of results.

For the boolean function synthesis problem, brief trial and error was done in attempt to calibrate GEP's IS, IRS and Gene Transposition operators, hence appearing different from the standard recommendations mentioned in [4].

4.4.4 Benchmarks and Test-Sets

The list of benchmarks and test-sets used to perform the experiments is described in table 4.2. For Symbolic Regression (Single Variable) and Boolean Function Synthesis, benchmarks are the same as the ones in [34]. For Symbolic Regression (Double Variable), benchmarks are the same as in [35]. For Sorting Algorithm Synthesis, the test-sets are retrieved from [1]. Function and Terminal sets are described in table 4.1.

4.5 Results and Observations

Table 4.3 contains results for each benchmark performed by each of the following combinations: GEP without crossover (displayed as NoCX), GEP with 2P (displayed as 2P), GEP with BSF (displayed as BSF), SGEP with KLX (displayed as KLX), GSGP with KLX (displayed as GSGP KLX) and GSGP using geometric crossover (displayed as GSGP). GSGP KLX and GSGP are represented by the results in [34] and [35] studies respectively and are a base for comparing against the other combinations.

Synthesis of Sorting Algorithms problem contains results for the same GEP and SGEP combinations used for the other problems. For control purposes however, Steady State GP results from [1] are used instead (displayed as GP (SSGP)).

Some combinations of benchmarks/crossovers are not available for some problems and thus marked with "n.a.".

The different combinations of algorithms, operators and benchmarks allow observing the table from different perspectives:

4.5.1 Crossovers under GEP: NoCX vs 2P vs BSF

Among the three variations corresponding to non-semantic GEP, NoCX, with few exceptions, obtained the worst results. This is expected as crossover is a crucial part of the evolutionary process. The magnitude of difference is however not that different. This might be explained by the employment of GEP exclusive operators, which although less active than normal crossover (around 0.1 probability of activation compared to 0.8 depending on the problem), might compensate for the lack of combinatorial variation among elements. 2P and BSF obtained similar results for most situations with none standing out considerably.

4.5.2 GEP vs SGEP

In general, SGEP paired with KLX, shows considerably better results in most situations when compared to any of the three GEP combinations. The exception is for Boolean Function Synthesis problem where results were mixed and often worst, though not significantly. This is a good indication of the potential of combining GEP with Semantics.

For the Synthesis of Sorting Algorithms problem, although obtaining similar but slightly worst average results, KLX converged more often for the different test-sets (94) than the other approaches (67, 92, 89 for NoCx, 2P and BSF respectively). The difference in convergence wasn't large compared to 2P, but the results were already close to perfect for the latest in all test-sets, which sets a limit on the potential for improvement (and which nevertheless occurred). Therefore, in this perspective, there is a positive indication of SGEP's potential.

For Boolean Function Synthesis problem, results between 2P, BSF and KLX are very similar, with BSF obtaining a not at all expressive advantage.

Finally for Symbolic Regression, both for single variable and double variable instances of the problem, SGEP with KLX stood out positively from GEP. In 8 out of the 13 benchmarks, SGEP with KLX results were better than the other 3 GEP options. For all the cases where KLX didn't perform better, the difference to the best performer is rather minimal. Once again, a good indication of SGEP's potential.

4.5.3 SGEP vs GP

It was already demonstrated in this thesis' contribution 1 that GEP outperforms GP under similar conditions when applied to the Synthesis of Sorting Algorithms problem (considering the set of functions, terminals and remaining parameters employed). Again under similar conditions, the experiment was repeated, this time with the other combinations of GEP/crossover operators (NoCX, 2P and BSF) and SGEP (KLX). When compared to conventional GP (SSGP), GEP and SGEP always outperformed GP, even when no crossover is used (NoCX).

4.5.4 SGEP vs SGP

SGEP compared to SGP had mixed results depending on the problem.

For Boolean Function Synthesis, GEP and SGEP greatly under-performed compared to GSGP's both studies. In [35] for most benchmarks, GSGP converged. In contrast, GEP combinations and SGEP results were far from it. The difference to GSGP KLX results which didn't converged is often of several orders of magnitude. At first we thought a possible cause for such under-performance might be GEP's element size limitation. While it helps prevent bloating, when solutions for a problem require a larger number of functions and terminals to perform the task at hands with acceptable or better results, if GEP element-size limits is not set accordingly, it might limit the quality of results. To investigate if this was the case we ran a limited set of the Boolean Function Synthesis experiments with Head Size parameter set to 100. The results were similar to the experiments with a smaller head size, which lead us to believe there are other factors influencing the results. Future work can investigate the origin of such discrepancy in results.

For Symbolic Regression with a single variable, results were mixed. Compared to GSGP KLX [34], SGEP KLX under-performed in all benchmarks except Keijzer4. Compared to GSGP [35], SGEP outperformed GSGP in 3 out of 5 benchmarks. The differences were quite significant, particularly when SGEP under-performed. Given this mixed results, it is possible to say SGEP can improve results at times. Results from this benchmark seem to indicate how SGEP and GSGP algorithms performance can differ based on their differences

in nature. Despite all their shared similarities, in terms of performance, algorithms seem to be complementary to one another.

For Symbolic Regression (Double Variable), GEP and SGEP greatly outperformed GSGP in all 4 benchmarks. In one case by two orders of magnitude. Results for this problem are a favorable indication of SGEP potential of outperforming SGP.

All in all, when compared the performance in results for both algorithms, SGEP under-performed in one problem, outperformed in another and in Symbolic Regression had mixed performance against SGP. There seems to be an indication for complementarity between these approaches.

4.6 Conclusions

This experiment demonstrates how GEP can be enhanced to consider candidate program's semantics. This is done by adding semantic measuring and manipulation, similarly to how SGP is built over GP. This new algorithm, SGEP, contains the advantages of both, otherwise distinct, ramifications of GP: GEP's karva elements, bloat control, always correct expressions, hidden sequences of genome and overall simplicity of implementation, and SGP's semantic awareness and integration with the expanding and promising Geometric Semantic approach to GP. In practice, SGEP was applied to a broad set of benchmarks and test-sets corresponding to 3 different problems and 1 variation. The obtained results allow SGEP to be positioned against its alternative approaches in terms of performance: GP, GEP, and GSGP (the geometric variant of SGP). SGEP showed mixed results depending on the problem. In general, it managed to over-perform conventional GEP and GP, especially in the Synthesis of Sorting Algorithms problem. Against GSGP, results were very distinct with SGEP under-performing greatly in the Boolean Function Synthesis problem, outperforming greatly in Symbolic Regression (Double Variable) problem and having mixed results in Symbolic Regression (Single Variable) variation. The results seem to indicate a complementary positioning between SGP and SGEP related to the problem being addressed. In practical terms, another factor in which SGEP differs from its alternatives is in implementation challenges. This factor should be considered in practice when deciding which approach to

apply to a problem. While bloat control and always correct expressions may ease SGEP application to a problem, one must carefully consider the values for extra parameters. For example Head Size, which limits the size of elements, when defined too small, might limit the complexity of structures attainable. If set too big it may lead to an increase in computing time necessary for evolution. Allocating some time and computer resources to identify an as good as possible combination for these parameters can possibly minimize this issue (combination which may vary depending on the problem being addressed). Overall, SGEP indicates potential to improve over GEP. Research exists suggesting semantics should be standard on GP implementation packages. In this study indications are identified that the same thing should apply to GEP.

SGEP parameters matching GP's	
Parameter	Value
Population-size	1024
	SR2,SORT: 1000
Selection	Tournament Selection Pool Size 7
	Tournament Selection Pool Size 4
Probability of crossover	1
Probability of mutation	0
Replacement factor	0.8
	SORT: 0.95
Number of runs	100
	SORT: 20
Maximum number of generations (1)	100
	SORT: 50
Fitness function	SR1, SR2: Euclidean distance
	BOOL: Hamming distance
	SORT: Number of inversions
Optimal fitness	Minimum
	SORT: Maximum
Termination condition	Max. # of gen. or find converging element (fitness 0)
Functions and Terminals	SR1: x,+,-,*,/,sin,cos,exp,log
	SR2: same as SR1 plus terminal x2
	BOOL: D1..D8, and, or, nand, nor
	SORT: len, index, dobl, swap, wibigger, wismaller, e1p
BSF and KLX pool size:	10

SGEP exclusive parameters	
Parameter	Value
Head Size	12
	SORT: 16
	SR2: 18
Number of Genes	4
Probability of IS	0.1
	BOOL: 0.3
Probability of RIS	0.1
	BOOL: 0.3
Probability of Gene Transposition	0.1
	BOOL: 0.2
Probability of Gene Recombination	same as probability of crossover

Table 4.1: Defined set of parameters for Experiment 2. SR1, SR2, BOOL and SORT respectively refer to Symbolic Regression (Single Variable), Symbolic Regression (Double Variable), Boolean Function Synthesis and Sorting Algorithm Synthesis problems.

Symbolic Regression - Single variable		
Benchmark	Function	Range
Septic	$x^7 - 2x^6 + x^5 - x^4 + x^3 - 2x^2 + x$	[-1,1]
Nonic	$\sum_{i=1}^{9} = x^i$	[-1,1]
R1	$(x+1)^3/(x^2 - x + 1)$	[-1,1]
R2	$(x^5 - 3x^3 + 1)/(x^2 + 1)$	[-1,1]
R3	$(x^6 + x^5)/(x^4 + x^3 + x^2 + x + 1)$	[-1,1]
Nguyen6	$\sin(x) + \sin(x + x^2)$	[-1,1]
Nguyen7	$\log(x+1) + \log(x^2 + 1)$	[0,2]
Keijzer1	$0.3x \sin(2\pi x)$	[-1,1]
Keijzer4	$x^3 \mathrm{e}^{-x} \cos x \sin x (\sin^2 x \cos x - 1)$	[0,10]

Symbolic Regression - Two variables		
Benchmark	Function	Range
Nguyen9	$\sin x_1 + \sin x_2^2$	$[0,1]^2$
Nguyen12	$x_1^4 - x_1^3 + (x_2^2/2) - x_2$	$[0,1]^2$
Pg1	$1/(1 + x_1^{-4}) + 1/(1 + x_2^{-}4)$	$[-5,5]^2$
Vl1	$\mathrm{e}^{-(x_1-1)^2}/(1.2 + (x_2 - 2.5)^2)$	$[0,6]^2$

Boolean Function Synthesis		
Benchmark	Problem	#Bits (Fitness Cases)
PAR5	Even parity	5 (32)
PAR6	Even parity	6 (64)
PAR7	Even parity	7 (128)
MUX6	Multiplexer	6 (64)
MUX11	Multiplexer	11 (2048)
MAJ7	Majority	7 (128)
MAJ8	Majority	8 (256)
CMP6	Comparator	6 (64)
CMP8	Comparator	8 (256)

Sorting Algorithm Synthesis			
Test-Set	Description	# Tests	Max length
A	baseline	15	30
B	baseline	15	30
C	more tests	25	30
D	longer tests	15	50
E	no new tests each generation	15	30

Table 4.2: Description of benchmarks and test-sets employed in the experiments. Each table corresponds to a different problem. For symbolic regression, the training-set is composed by 20 equidistant points distributed by the given range while test-set is composed by 20 uniformly drawn from the same range. For the Boolean Function Synthesis problem, training and test sets are composed by all the possible fitness cases generated by the number of defined #Bits. For Sorting Algorithm Synthesis, tests are randomly generated with # Tests defining the number of tests and Max length their size limit.

Symbolic Regression - Single variable						
Benchmark	NoCX	2P	BSF	KLX	GSGP KLX [34]	GSGP [35]
Septic	3.15 ±0.39	2.15 ±0.28	2.269 ±0.32	2.00 ±0.28	**0.266 ± 0.052**	n.a
Nonic	58.79 ±10.72	41.77 ±8.68	43.64 ±7.88	39.63 ±7.61	**0.223 ± 0.033**	n.a.
R1	1.63 ±0.20	1.32 ±0.17	1.300 ±0.18	1.22 ±0.16	**0.177 ± 0.041**	2.71 ±0.27
R2	0.94 ±0.07	0.89 ±0.07	0.91 ±0.06	0.89 ±0.06	**0.163 ± 0.032**	0.50 ±0.06
R3	0.21 ±0.05	0.17 ±0.03	0.16 ±0.03	0.14 ±0.02	**0.038 ± 0.007**	0.19 ±0.02
Nguyen6	0.79 ±0.12	0.69 ±0.13	0.76 ±0.13	0.69 ±0.12	**0.027 ± 0.013**	n.a.
Nguyen7	0.18 ±0.03	0.13 ±0.02	0.13 ±0.02	0.10 ±0.02	**0.053 ± 0.014**	n.a.
Keijzer1	0.40 ±0.03	0.37 ±0.03	0.36 ±0.03	0.35 ±0.03	**0.134 ± 0.022**	0.35 ±0.01
Keijzer4	0.33 ±0.02	0.31 ±0.02	0.31 ±0.02	0.31 ±0.02	**0.455 ± 0.093**	0.96 ±0.03

Symbolic Regression - Two variables						
Benchmark	NoCX	2P	BSF	KLX	GSGP KLX [34]	GSGP [35]
Nguyen9	0.13 ±0.04	0.11 ±0.03	0.13 ±0.03	**0.09 ± 0.02**	n.a.	0.65 ±0.04
Nguyen12	0.18 ±0.02	0.15 ±0.01	0.15 ±0.01	**0.14 ± 0.01**	n.a.	0.37 ±0.02
Pg1	0.54 ±0.07	0.43 ±0.07	0.46 ±0.07	**0.34 ± 0.07**	n.a.	1.25 ±0.08
Vl1	0.04 ±0.01	**0.03 ± 0.00**	**0.03 ± 0.00**	**0.03 ± 0.00**	n.a.	1.03 ±0.02

Boolean Function Synthesis						
Benchmark	NoCX	2P	BSF	KLX	GSGP KLX [34]	GSGP [35]
PAR5	12.80 ±0.18	12.51 ±0.20	12.32 ±0.21	12.24 ±0.23	2.033 ±0.510	**0.0 ± 0.00**
PAR6	28.94 ±0.21	28.59 ±0.28	28.59 ±0.28	28.48 ±0.23	13.367 ±0.792	**0.0 ± 0.00**
PAR7	61.16 ±0.23	60.67 ±0.28	60.89 ±0.25	60.70 ±0.29	40.967 ±1.047	**0.13 ± 0.12**
MUX6	9.16 ±0.75	8.28 ±0.67	7.87 ±0.74	6.65 ±0.76	5.667 ±0.642	**1.13 ± 0.37**
MAJ7	20.74 ±0.55	19.26 ±0.53	19.44 ±0.59	18.61 ±0.48	0.133 ±0.153	**0.0 ± 0.00**
MAJ8	43.32 ±0.88	40.42 ±0.80	40.38 ±0.86	39.49 ±0.80	**0.000 ± 0.000**	**0.0 ± 0.00**
CMP6	5.84 ±0.38	5.02 ±0.41	4.72 ±0.39	4.63 ±0.39	0.533 ±0.221	**0.0 ± 0.00**
CMP8	28.01 ±1.58	26.04 ±1.30	25.12 ±1.37	25 ±1.82	7.300 ±0.956	**0.0 ± 0.00**

Sorting Algorithm Synthesis						
Benchmark	NoCX	2P	BSF	KLX	GP (SSGP) [1]	
Test-Set A	12.627 ±1.33	14.576 ±0.49	**14.696 ± 0.47**	14.211 ±0.85	n.a.	
Test-Set B	13.606 ±1.00	14.671 ±0.51	**15 ± 0.00**	14.573 ±0.67	n.a.	
Test-Set C	21.899 ±1.99	24.805 ±0.30	24.485 ±0.80	**25 ± 0.00**	n.a.	
Test-Set D	11.225 ±1.53	14.476 ±0.73	**14.943 ± 0.09**	13.618 ±1.18	n.a.	
Test-Set E	12.490 ±1.49	**14.609 ± 0.46**	13.873 ±1.03	13.014 ±1.34	n.a.	

Sorting Algorithm Synthesis - Convergence						
Benchmark	NoCX	2P	BSF	KLX	GP (SSGP)	
Test-Set A	14	18	18	**19**	12	
Test-Set B	16	19	19	**20**	10	
Test-Set C	13	19	**20**	19	8	
Test-Set D	10	18	17	**19**	8	
Test-Set E	14	**18**	15	17	5	

Table 4.3: Experiment results. Each problem is presented in a different table. Different algorithm/operator combinations are organized through columns. Each different benchmark/test-set occupies a different row. Tables show the average of best fitness obtained out of 100 evolutionary runs followed by respective 0.95% confidence interval. For the Symbolic Regression and Boolean Function Synthesis the objective was to minimize, so less fitness is better. For Sorting Algorithm Synthesis it's maximize, higher fitness being better. The last table shows the number of convergent runs out of 20 for Sorting Algorithm Synthesis experiments.

5. Encouraging phenotype variation with a new semantic operator: Semantic Conditional Crossover

This chapter refers to Contribution 3 of this thesis. A study of its own and part of the group of 3 bonded contributions.

In this study we propose a semantic variation of the Wang genetic algorithm [36]. The Wang genetic algorithm is a modification of the GA process that encourages genotype diversity in the population by replacing crossover with mutation when a selected pair of parents are too syntactically similar. This strategy has been shown to promote the evaluation of a wider range of solutions and reduce premature convergence. Wang genetic algorithm doesn't fit in the role of traditional genetic operators. Nevertheless, the implementation of its characteristics is arguable modular and localized enough that it can be easily added or removed without fundamentally change the genetic algorithm. In this study, we will therefore refer to the specificities of the Wang algorithm as an operator.
The variation we propose, Semantic Conditional Crossover (SCC), is similar in concept to Wang's operator: Crossover is performed only when parent elements are different enough. When it's not the case, mutation is performed. SCC however, has a crucial distinction in the way parents difference is measured. While in Wang's operator, elements difference is measured based on syntax, we propose comparing their semantics. The objective of this modification is to encourage phenotype variation in the population, as opposed to genotype variation. To evaluate the semantic difference between elements, SCC, requires access to their

semantic values. This implies that SCC is only compatible with an evolutionary algorithm where semantic measuring of elements is performed. SCC is therefore compatible with SGEP and SGP. In the following sections SCC is described in further detail. To investigate its potential, a practical experiment is also described. The work ends with discussion on the operator's applicability.

5.1 Semantic Conditional Crossover Operator for SGP and SGEP

crossover and mutation operators as arguments

Unlike traditional operators, SCC is a conditional step, which means the operator consists in performing a decision (more on this next) that leads to one of two possible options: either crossing over the elements provided, or mutating one of them. This requires two additional complementary operators to be provided to work along SCC, one crossover and one mutation operator. Notice SCC behavior is independent of which operators are chosen and can be combined with different operators for different strategies. This choice is therefore handed to the person composing the evolutionary algorithm. In a functional perspective, these complementary crossover and mutation operators can be seen as parameters provided to SCC. As an example, consider an implementation of SCC provided with 2P crossover and semantic mutation. In this case, if the condition results in the mutation option, semantic mutation is applied to one of the elements, else, 2P crossover takes place.
When constructing the evolutionary process, element modifications don't have to be limited to SCC. After selection and after SCC taking place, the resulting elements can be proposed to other forms of modification, e.g. a different mutation operator.

SCC's semantic based condition

In evolution terms, positioning elements based on their semantic value corresponds to classifying them as more or less alike in terms of phenotype.
The main characteristic of SCC is a conditional step that compares two elements semantics.

This condition leads to either crossover, if the elements are semantically different enough or mutation otherwise, as described in previous section. While Wang's operator encourages syntactical variation in the population, SCC, by restricting crossover from elements too semantically similar and promoting the introduction of possibly new building blocks in the population by mutation, will in theory introduce phenotypical variety into the population.
More precisely, the condition in SCC is the following: given two provided parent elements coming from selection, if the semantic distance between elements is larger than a predefined threshold, the designated crossover operator proceeds as normal and two offspring elements replace parents in the new population (or proceed to further modification, depending on evolutionary process configuration). Otherwise, one of the parents is mutated with a certain probability and re-introduced, the other parent is re-introduced directly as is. Pseudo-code for this process is provided in Algorithm 1.

5.2 Experiment Conditions

To test SCC, we ran a similar group of tests to the ones used previously this thesis, in section 4.4, but integrating SCC combined with the list of crossover operators used before. Results from experiment 2 are used as base for comparison for this new new experiment's. Apart from integrating SCC, all other conditions are set to be as similar as possible. The same base algorithms will be used for the same crossover operators: GEP paired with Two Point Crossover operator (2P) and BSF crossover operator and SGEP paired with the Approximately Geometric Semantic Crossover operator KLX. The same initial population was used for both experiments (randomly generated with equal initial seeds).
These experiments were conducted the same quad core Intel i7, 2.00GHz, 4Gb of Ram laptop. Again, no accurate time measuring was performed. This time the entirety of the benchmarks with the larger set of experiments took between a month and a month and a half to complete. It's important to mention that during this time the experiments were performed intermittently and the computer was often in battery saving or low performance conditions.

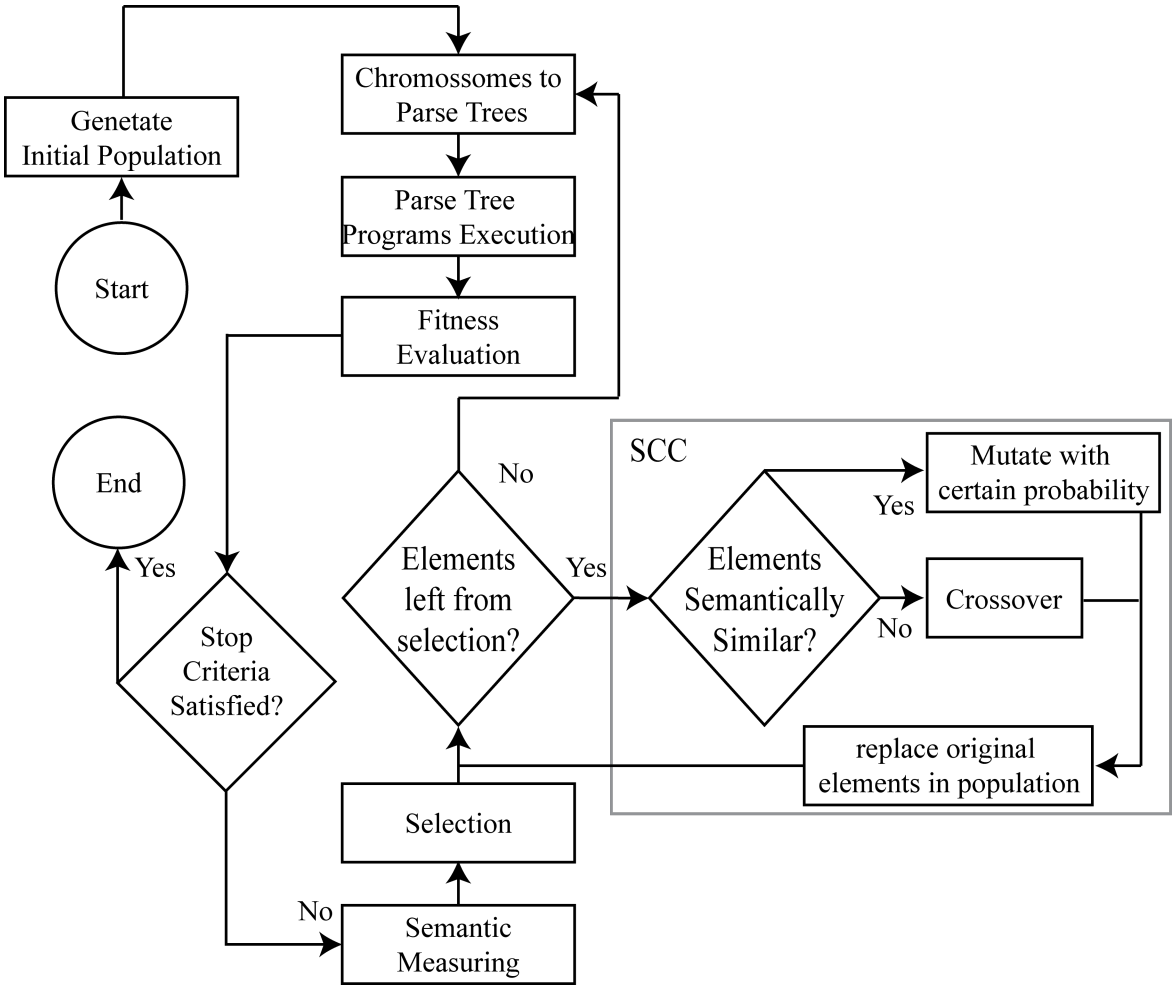


Figure 5.1: Flowchart illustrating SGEP evolutionary algorithm paired with SCC operator. SCC integrates as a module and can be easily replaced with other modification operators without affecting the evolutionary process flow.

```
input : Population P of size N;
Max number of generations: G;
Semantic distance calculation method: d;
Semantic distance reference threshold: T;
Defined selection operator: s;
Probability of Crossover: pCX;
Crossover operator designated to work with SCC: cX;
Mutation operator designated to work with SCC: m;
P ← generate initial population;
for g ← 1 to G do
    Evaluate fitness of each element in P;
    if element in P is convergent then
        break loop;
    end
    Evaluate semantic of each element in P;
    //Depending on the problem, this step can be done along with fitness evaluation;
    P2 ← initialize replacement population;
    if proceed under pCX probability then
        while Number of elements in P2 < N do
            //SCC operator starts here
            p1, p2 ← select 2 elements from P using s;
            sd ← calculate semantic distance between p1 and p2 using d;
            if sd < T then
                n1 ← new element mutated from p1 using m under certain probability;
                n2 ← copy of p2 ;
            else
                offspring ← crossover between p1 and p2 using cX;
                n1, n2 ← two different childs from offspring ;
            end
            Add n1 and n2 to population P2 ;
        end
        //SCC concluded
    end
    optionally perform other forms of modification to P2;
    P ← P2 under a certain replacement factor;
end
```

Algorithm 1: Example of an Evolutionary Process with Semantical Conditional Crossover. Some details specific to the evolutionary algorithm are omitted for simplicity and abstraction. For example in SGEP, conversion of k-expression chromosomes to parse trees would have to be performed before fitness evaluation. d method is dependent on the problem at hands. Notice that deciding if SCC performs using pCX is optional and should vary with the strategy intended.

5.2.1 Benchmarks and Parameters

Experiments were ran with problems Symbolic Regression single variable, Symbolic Regression double variable, Boolean Function Synthesis and Synthesis of Sorting Algorithms. The list of Functions and Terminals is therefore the same as in Contribution 2, found along the algorithm configuration parameters in 4.1. Benchmarks and Test-Sets can be found in 4.2.

5.2.2 Configuration Parameters

The initial parameters are the same employed in Experiment 2, and thus, described in 4.1. The remaining parameters necessary to configure SCC are displayed in table 5.1, which include semantic distance threshold and probability of mutation. For simplicity, the Semantic Distance Threshold and Probability of Mutation values associated with SCC were obtained from empirical decision upon observing several individual runs.

SCC parameters		
Parameter	Problem	Value
Semantic Distance Threshold	SR1	0.9
	SR2	0.9
	BOOL	10
	SORT	0.6
Probability Mutation inside SCC	SR1	0.3
	SR2	0.3
	BOOL	0.6
	SORT	0.6

Table 5.1: Experiment 3 SCC parameters

5.3 Interpreting results

Results obtained using SCC are displayed in table 5.2. To ease the comparison between SCC and non-SCC experiments, table 5.3 displays side-by-side results from experiment 2 (No SCC) and experiment 3 (using SCC).

Results for the NoCX experiment (probability of crossover is defined to be 0 and no crossover operator is employed) are exactly the same with and without SCC. Results with NoCX oper-

ator for this comparison are therefore considered irrelevant and omitted from the comparison table.

It's important to mention that parameters used to configure SCC were not optimized. Consequently, the obtain results correspond to a guaranteed minimum potential for the algorithm/operator/parameters combination. Fine tuning the parameters could improve the results obtained at the cost of extra experimental effort and resources. In practical applications o, we suggest parameter optimization cost and benefit to be considered before large scale use of the algorithms mentioned in this work.

5.4 Results and Discussion

Using SCC improved and worsen some of the results. In all situations the same order of magnitude was maintained. The differences were, in most cases, of a small fraction of the total value, suggesting the potential of this operator to be for slightly pushing improvements instead of revolutionizing the potential of solutions obtained. The differences were, however, not statistically relevant enough to produce any strong indications.

In some of the benchmarks, using SCC resulted in a change of which crossover was the most effective. Considering also the different problems addressed, the employment of SCC doesn't seem to shift results consistently or significantly, on the contrary, improvements seemed to occur without a noticeable pattern across the different tests. This verification is somehow expected given the stochastic nature of mutation adding to the extra probabilistic nature introduced by the concept of SCC. The operator seems to be able to introduce variation in the range of results obtained, but nevertheless in the experiments conducted, not sufficient or significant. Further work can investigate the potential and relevance of this operator under different conditions.

5.4.1 Adaptive solutions for setting the threshold parameter is SCC

SCC introduces variation by the means of mutating elements instead of crossing them over depending on their similarity. However, variation in population might be more desirable in some stages of evolution than others. Although dependent on the strategy and problem being faced, variation is commonly more desirable in the earlier stages of evolution than on the later. For the Wang algorithm, some approaches have been proposed.

In [37] an adaptive mechanism is proposed: considering the difference between parent elements to be given by a continuous factor $d\sigma(0,1)$, threshold T is gradually adjusted each generation according to the following criteria: $T[i+1] = Q.T[i]$ where i is a generation and $Q\sigma(0,1)$. Q is set by the user and is called cooling ratio.

Despite the changes mentioned above considering the transitions of evolution phases, it still doesn't provide indication on how to set the initial value for T. Although choosing an adequate T being a problem dependent task, when poorly chosen, it can significantly worse performance. The authors in [38] propose a new mechanism for setting T based on the average difference degree among all selected parent elements of a given generation. This value is referred as d. After remaining identical for a user defined g number of generations, d is recalculated used to redefine T. Between each new calculation of d, cooling ratio technique is applied to adjust T, similarly to the previous mentioned mechanism.

Similar to Wang's algorithm, SCC can benefit from the mentioned adaptive mechanisms. As in SCC itself, the mechanism parents similarity calculation method needs to shift to semantic based in order for phenotype changes to be leading factor for threshold parameter adaptation. Future research can experiment mixing SCC with adaptive solutions already existent for the Wang algorithm.

5.4.2 Future Work

The level of "intensity" of this operator can be easily altered. Changing the probability of mutation, threshold for distance condition and number of parents mutated (mutating both parents instead of one when elements are not similar) may change the way and range of

variety introduced. Further investigation can examine variation in potential of SCC under different conditions in attempt to understand whether the concept has potential to push results more significantly.

5.5 Conclusion

In this study we propose a semantic variation of Wang's algorithm. In the original Wang algorithm, elements are proposed for crossover conditionally instead of probabilistically. The strategy is to, in order to increase variation, only perform crossover when candidate parents are different enough. If, on the other hand, parents are too similar, one of the parents is mutated. In Wang's algorithm, calculating elements difference is done by a form of syntactical comparison specific for linear strings. Our variation of this algorithm, the SCC operator has SGP and SGEP approaches in mind and works by performing semantic comparison instead of syntactic.

In evolution terms, positioning elements based on their semantic value instead of syntax corresponds to phenotype comparison instead of genotype. Using phenotype for positioning was demonstrated to more accurately mimic evolution and better map element's relevant characteristics.

To test this algorithm, a practical experiment is described. SGEP is used as the base algorithm and faces the usage of SCC against its absence. Results show SCC usage provides a slight discrepancy in the results obtain, with both improvements and declines in obtained values. In the experiment performed, the results don't have enough statistical significance to provide a sustained indication of their quality. Further studies can investigate whether the same concept with adaptations or in different experiments can obtain more expressive results. The operator contains 2 parameters that need to be set but are arguable trivial to understand, which might facilitate its applicability and usage, especially if semantic measuring is already taking place in the evolutionary process.

Employing SCC makes the most sense if the evolutionary algorithm already performs semantic evaluation. In this case SCC requires no fundamental changes in the evolutionary process, making integration especially easy. Furthermore, it might be a simple way to push

results as its parameters are arguably trivial to understand and configure.

The experiment associated with this contribution was performed with a GEP implementation, thus indicating no obstacles in developing state of the art technology that is suitable for the field of genetic programming under GEP's principles instead of GP's.

Symbolic Regression - Single variable				
Benchmark	NoCX	2P	BSF	KLX
Septic	3.15 ±0.39	2.24 ±0.31	**2.07 ± 0.28**	2.26 ±0.30
Nonic	58.79 ±10.72	43.41 ±7.39	48.19±9.24	**36.06 ± 6.89**
R1	1.63 ±0.20	**1.30 ± 0.19**	1.36 ±0.21	1.34 ±0.18
R2	0.94 ±0.07	**0.88 ± 0.07**	0.90 ±0.06	**0.88 ± 0.07**
R3	0.21 ±0.05	0.17 ±0.03	0.17 ±0.03	**0.15 ± 0.03**
Nguyen6	0.79 ±0.17	0.71 ±0.12	0.72 ±0.11	**0.69 ± 0.11**
Nguyen7	0.18 ±0.03	0.14 ±0.03	0.15 ±0.03	**0.12 ± 0.02**
Keijzer1	0.40 ±0.03	0.38 ±0.03	0.38 ±0.03	**0.36 ± 0.03**
Keijzer4	0.33 ±0.02	0.31 ±0.02	0.31 ±0.02	**0.30 ± 0.02**

Symbolic Regression - Two variables				
Benchmark	NoCX	2P	BSF	KLX
Nguyen9	0.13 ±0.04	0.11 ±0.03	0.12 ±0.03	**0.07 ± 0.02**
Nguyen12	0.18 ±0.02	0.15 ±0.01	0.15 ±0.01	**0.14 ± 0.01**
Pg1	0.54 ±0.07	0.44 ±0.07	0.48 ±0.07	**0.37 ± 0.06**
Vl1	0.04 ±0.01	**0.03 ± 0.00**	**0.03 ± 0.00**	**0.03 ± 0.00**

Boolean Function Synthesis				
Benchmark	NoCX	2P	BSF	KLX
PAR5	12.80 ±0.18	12.53 ±0.19	12.60 ±0.19	**12.17 ± 0.23**
PAR6	28.94 ±0.21	28.59 ±0.28	28.53 ±0.25	**28.52 ± 0.23**
PAR7	61.16 ±0.23	60.90 ±0.26	60.98 ±0.23	**60.89 ± 0.23**
MUX6	9.16 ±0.75	8.05 ±0.72	8.09 ±0.65	**7.16 ± 0.66**
MAJ7	20.74 ±0.55	19.41 ±0.52	19.11 ±0.55	**18.58 ± 0.47**
MAJ8	43.32 ±0.88	40.42 ±0.82	40.67 ±0.88	**39.87 ± 0.76**
CMP6	5.84 ±0.38	5.01 ±0.42	5.39 ±0.42	**4.42 ± 0.36**
CMP8	28.01 ±1.58	26.12 ±1.46	25.48 ±1.32	**24.16 ± 1.19**

Sorting Algorithm Synthesis				
Benchmark	NoCX	2P	BSF	KLX
Test-Set A	12.627±1.33	**14.799 ± 0.23**	14.265±0.71	14.392±0.61
Test-Set B	13.606±1.00	14.579±0.66	**14.595 ± 0.51**	13.902±0.81
Test-Set C	21.899±1.99	24.272±1.14	**24.322 ± 0.80**	23.663±1.44
Test-Set D	11.225±1.53	14.575±0.58	**14.929 ± 0.11**	13.705±1.10
Test-Set E	12.490±1.49	13.566±1.21	13.874±1.03	**14.885 ± 0.18**

Sorting Algorithm Synthesis - Convergence				
Benchmark	NoCX	2P	BSF	KLX
Test-Set A	14	**19**	17	17
Test-Set B	16	**19**	18	16
Test-Set C	13	**19**	18	18
Test-Set D	10	18	**19**	17
Test-Set E	16	16	**19**	**19**

Table 5.2: Experiment 3 results, obtained under the same conditions as experiment 2 except for the employment of SCC with conditioning activated

Symbolic Regression - Single variable						
Benchmark	2P	2Pc	BSF	BSFc	KLX	KLXc
Septic	**2.15 ± 0.28**	2.24 ±0.31	2.269 ±0.32	**2.07 ± 0.28**	**2.00 ± 0.28**	2.26 ±0.30
Nonic	**41.77 ± 8.68**	43.41 ±7.39	**43.64 ± 7.88**	48.19 ±9.24	39.63 ±7.61	**36.06 ± 6.89**
R1	1.32 ±0.17	**1.30 ± 0.19**	**1.30 ± 0.18**	1.36 ±0.21	**1.22 ± 0.16**	1.34 ±0.18
R2	0.89 ±0.07	**0.88 ± 0.07**	0.91 ±0.06	**0.90 ± 0.06**	0.89 ±0.06	**0.88 ± 0.07**
R3	**0.17 ± 0.03**	**0.17 ± 0.03**	**0.16 ± 0.03**	0.17 ±0.03	**0.14 ± 0.02**	0.15 ±0.03
Nguyen6	**0.69 ± 0.13**	0.71 ±0.12	0.76 ±0.13	**0.72 ± 0.11**	**0.69 ± 0.12**	**0.69 ± 0.11**
Nguyen7	**0.13 ± 0.02**	0.14 ±0.03	**0.13 ± 0.02**	0.15 ±0.03	**0.10 ± 0.02**	0.12 ±0.02
Keijzer1	**0.37 ± 0.03**	0.38 ±0.03	**0.36 ± 0.03**	0.38 ±0.03	**0.35 ± 0.03**	0.36 ±0.03
Keijzer4	**0.31 ± 0.02**	**0.31 ± 0.02**	**0.31 ± 0.02**	**0.31 ± 0.02**	0.31 ±0.02	**0.30 ± 0.02**
Symbolic Regression - Two variables						
Benchmark	2P	2Pc	BSF	BSFc	KLX	KLXc
Nguyen9	**0.11 ± 0.03**	**0.11 ± 0.03**	0.13 ±0.03	**0.12 ± 0.03**	0.09 ±0.02	**0.07 ± 0.02**
Nguyen12	**0.15 ± 0.01**	**0.15 ± 0.01**	**0.15 ± 0.01**	**0.15 ± 0.01**	**0.14 ± 0.01**	**0.14 ± 0.01**
Pg1	**0.43 ± 0.07**	0.44 ±0.07	**0.46 ± 0.07**	0.48 ±0.07	**0.34 ± 0.07**	0.37 ±0.06
Vl1	**0.03 ± 0.00**	**0.03 ± 0.00**	**0.03 ± 0.00**	**0.03 ± 0.00**	**0.03 ± 0.00**	**0.03 ± 0.00**
Boolean Function Synthesis						
Benchmark	2P	2Pc	BSF	BSFc	KLX	KLXc
PAR5	**12.51 ± 0.20**	12.53 ±0.19	**12.32 ± 0.21**	12.60 ±0.19	12.24 ±0.23	**12.17 ± 0.23**
PAR6	**28.59 ± 0.28**	**28.59 ± 0.28**	28.59 ±0.28	**28.53 ± 0.25**	**28.48 ± 0.23**	28.52±0.23
PAR7	**60.67 ± 0.28**	60.90 ±0.26	**60.89 ± 0.25**	60.98 ±0.23	**60.70 ± 0.29**	60.89±0.23
MUX6	8.28 ±0.67	**8.05 ± 0.72**	**7.87 ± 0.74**	8.09 ±0.65	**6.65 ± 0.76**	7.16±0.66
MAJ7	**19.26 ± 0.53**	19.41 ±0.52	19.44 ±0.59	**19.11 ± 0.55**	18.61 ±0.48	**18.58 ± 0.47**
MAJ8	**40.42 ± 0.80**	**40.42 ± 0.82**	**40.38 ± 0.86**	40.67 ±0.88	**39.49 ± 0.80**	39.87±0.76
CMP6	5.02 ±0.41	**5.01 ± 0.42**	**4.72 ± 0.39**	5.39 ±0.42	4.63 ±0.39	**4.42 ± 0.36**
CMP8	**26.04 ± 1.30**	26.12 ±1.46	**25.12 ± 1.37**	25.48 ±1.32	25 ±1.82	**24.16 ± 1.19**
Sorting Algorithm Synthesis						
Benchmark	2P	2Pc	BSF	BSFc	KLX	KLXc
Test-Set A	14.576 ±0.49	**14.799 ± 0.23**	**14.696 ± 0.47**	14.265 ±0.71	14.211 ±0.85	**14.392 ± 0.61**
Test-Set B	**14.671 ± 0.51**	14.579 ±0.66	**15 ± 0.00**	14.595 ±0.51	**14.573 ± 0.67**	13.902±0.81
Test-Set C	**24.805 ± 0.30**	24.272 ±1.14	**24.485 ± 0.80**	24.322 ±0.80	**25 ± 0.00**	23.663±1.44
Test-Set D	14.476 ±0.73	**14.575 ± 0.58**	**14.943 ± 0.09**	14.929 ±0.11	13.618 ±1.18	**13.705 ± 1.10**
Test-Set E	**14.609 ± 0.46**	13.566 ±1.21	13.873 ±1.03	**13.874 ± 1.03**	13.014 ±1.34	**14.885 ± 0.18**
Sorting Algorithm Synthesis - Convergence						
Benchmark	2P	2Pc	BSF	BSFc	KLX	KLXc
Test-Set A	18	**19**	**18**	17	**19**	17
Test-Set B	**19**	**19**	**19**	18	**20**	16
Test-Set C	**19**	**19**	**20**	18	**19**	18
Test-Set D	**18**	**18**	17	**19**	**19**	17
Test-Set E	**18**	16	15	**19**	17	**19**

Table 5.3: Experiment 2 and 3 results side by side for easier comparison. 2P, BSF, KLX performed with GEP without and with conditioning activated. (Conditioning marked with c

6. Discussion

6.1 GEP as primary evolutionary option for program synthesis

With many alternative optimization techniques, evolutionary computation has in many cases had its main advantage in its simplicity of employment. An option where the understanding of necessary concepts and the task of setting it up to handle the problem at hands are sometimes more accessible than the alternatives. The knowledge for solving problems partially shifts from analytical understanding of the problem itself to understanding of evolution concepts that allow the automatic search for an evolved solution.

In existing research, the application of evolution for the automatic generation and optimization of executable code is a majority of times focused exclusively on GP. GP was the first developed form of such approach which works. Besides, GP has demonstrated positive results in practice, is extensively documented and a lot of research has been done around the basic algorithm. It's plausible to say GP algorithm is seen as the standard evolutionary option for program synthesis.

By design, GP has some characteristics that are in practice less desirable and which complicate its usage. Having its elements encoded in expression trees, GP consequently inherits some pitfalls. Associated problems with this approach include the occurrence of broken expressions which need repairing operators or complex methods to assure correctness and, the lack of mechanisms to limit evolved elements size which result in synthesized programs usually being bloated and containing a great number of irrelevant code structures. The bloat problem also usually results in extra work being necessary preparing and trimming the ob-

tained solutions. In a way, these issues distance GP from simplicity and straight forwardness of application usually associated and valued in evolutionary computation.

GEP provides an alternative to GP which handles some of these problems. GEP also provides other advantages and packs some interesting features in its evolutionary process. K-expressions, the name associated with GEP's elements, are easier to handle while still providing complexity similar to that of expression trees. This is possible due to their structure which encodes entire expression trees in linear strings of fixed size. By design, these elements manage to mitigate the issues observed and associated with GP: broken expressions and the overly-complex bloated results. Having fixed length, the limiting mechanism that is able to control bloat and overly-complex programs also provides the possibility to pressure for more compact solutions which in turn reduce the necessity for "trimming" of irrelevant sequences of the program. In terms of performance, GEP has also demonstrated benefits. Existing research shows the algorithm manages to outperforms GP by a considerable margin in several different problems. This set of apparent advantageous characteristics and evidence in mind, we believe it's important to understand whether GEP should be the standard option of choice for evolutionary based optimization of executable code, both for research and practical applications. Could a change in perception of the standard option in the field of evolutionary program synthesis benefit from GEP as a primary approach? Given the reduced implementation challenges that GEP provides, the field could be perceived as having a more manageable learning curve and having less restrictive challenges. We are referring especially to broken expressions and bloated solutions, but not only. With a more straightforward approach and better results, maybe more interest could be placed in evolutionary program generation, including more research and applications. On the other hand, a tool sometimes considered out of reach for some, could become more easily adopted.

GEP usage has some compromises though. As mentioned, GP being around for longer and being a more studied approach has already a more extensive set of documentation, experiments and documented applications. For the same reasons, there is certainly more available empirical and practical knowledge from potential applicants than with GEP. Not only so, but GEP comes with the burden of some extra parameters that need to be understood and set properly to avoid some pitfalls. From this study we verified and discussed the

later consideration, however, we don't consider it to be a major setback nor a difficult one to handle. In our practical experiments, even with un-optimized parameters, the results were mostly positive compared to GP variants, indicating that even sparing the effort on optimizing the parameters, there is a good chance GEP will over-perform GP. A parameter-less GEP variation would be of great interest for future work, something like was done to GA in [39].

In this work, we employed GEP in three distinct studies and situations which allow a perspective on the challenges of primary GEP usage. The three experiments apart from contributing to position GEP relative to GP, also are relevant and provide contribution and results for other branches each in its own way and apart from the main question binding the experiments. Following is a recap of the contributions/experiments:

6.2 Automatic Synthesis of Sorting Algorithms by Gene Expression Programming

First, GEP is applied to generate Sorting Algorithms under similar conditions to what as been done previously using GP in [1]. We implemented GEP and adapted it to work for this problem. Results confirm that GEP is capable of generating sorting algorithms and strongly outperform (Steady State) GP doing so, obtaining close to perfect results in the proposed test-sets in terms of convergence rates. The results obtained were done without optimizing the extra parameters of GEP, suggesting this optimization in not at all times necessary if better results are to be expected from GEP comparing to GP. This indication also adds to the previously good results in other problems, building confidence that when applied to new problems, GEP has a good probability of being able to outperform GP.

6.3 (Geometric) Semantic Gene Expression Programming

Second, adapting GEP to considering candidate program's semantics. Comparing elements using semantics has demonstrated to more closely correspond to what is expected of evolution. Practical results confirm the improvements. Investigation on the semantic space created by evaluating individual programs semantics gave origin to the geometric semantic approach and a new algorithm, GSGP. In our study we looked into the possibility and challenges of adapting these recent innovations to GEP. This process led to two new semantic and geometric semantic approaches to GEP (SGEP and GSGEP).

A practical experiment was performed comparing results between GP, SGP, GEP and SGEP. The experiment included combinations of different crossover operators and benchmarks from 3 different problems (Symbolic Regression (single and double variable variations), Boolean Function Synthesis and Sorting Algorithm Synthesis). The results can be observed from several perspectives: in terms of GEP and its new variation SGEP, both are competitive against GP and SGP. Both SGEP and SGP performed better more often than not considering their non semantic counterparts. When comparing SGEP and SGP, the results were mixed, and highly related to the problem at hands. GEP and SGEP did best at synthesizing sorting algorithms and regression with two variables, mixed with GP and SGP at symbolic regression and worst at Boolean function synthesis. We suspect some pitfall might have occurred for the Boolean Function Synthesis problem, possibly related to the definition of a too small value for the head size parameter. Future work can revisit GEP with this problem.

Although having mixed results, GEP proved compatible without complicated adjustments to the semantic approach already applied in GP. Semantic measuring and manipulation were successfully introduced in its evolutionary process. Converting existing semantic operators to work with GEP might require direct access to the elements parse tree and afterwards possibly need repair to convert back into a valid k-expression. This is not the case for many operators which don't manipulate parse trees directly which include all indirect and population diversity semantic operators. Furthermore, developing new strategies and operators that specifically take advantage of the linear encoding of parse trees seen in GEP's k-expressions

might open new possibilities for operator development.

6.4 Encouraging phenotypic variation with a new semantic operator: Semantic Conditional Crossover

Third, a new semantic variation of the Wang algorithm was proposed: Semantic Conditional Crossover. This operator shows slight variation on benchmark results when in use compared to when absent and replaced by simple crossover. The results are however not very expressive, with further investigation being needed to better understand the potential of the operator and its concept. The usage of the operator is quite uncomplicated but requires candidate programs semantic value, making it specially straightforward to include on an evolutionary process where semantic measuring is already taking place, the case of SGEP and SGP.
The operator was developed and introduced using the SGEP implementation. Studies of performance were done using this same algorithm while handling benchmarks from 3 different problems (the same as the ones employed on experiment 2) with and without SCC for comparison. This experiment, besides introducing a new operator/algorithm variation and demonstrating its performance potential, also demonstrate how GEP (in this case SGEP) can be used as primary platform for developing and testing innovation that is theoretically compatible and targeting both GP, GEP and program evolutionary program synthesis in general.

6.5 Consideration on GEP elements size and possible pitfall

During the development of these experiments, possibly the biggest struggle with GEP was configuring some of its exclusive parameters. Head-size parameter defines the number of positions where both terminal set and function set instructions can be placed. A tail size is calculated accordingly to form the complete expression blueprint that will be instantiate by every population element during the run. That said, head-size parameter controls the

number of instructions resulting solutions can contain. This same mechanism is capable of preventing bloat to a certain extent. The smaller the head-size the fewer bloat structures are capable of forming around relevant parts of code. The problem with this mechanism is that by reducing the possible number of instructions, we are also limiting the space of possible solutions at reach of GEP to search. Being an optimization technique which automatically searches for solutions, one of its advantage is the possibility to explore schemes that are not typically thought by humans. In a sense however, with this consequence of head size, GEP forces the user to have an idea of what the solution we expect the algorithm to find would look like, and so we might be in some way rejecting all the solutions we can't even imagine possible from actually materializing. At least in terms of size. If a really good solution takes one more instruction than what we imagine a really good solution would look like (plus the margin we might possibly defined for our overestimation of the unimaginable), than this solution is out of reach and GEP will not generate it.

In practice this problem is possibly not very relevant for many applications. Being able to define a Terminal and Function Sets and a fitness function, most likely implies sufficient knowledge and formed expectations on what a reasonable size for a resulting problem might be, although this might be depend greatly on the problem.

Another consideration is that the alternative, which is allowing programs to grow indefinitely as needed, is accompanied with the prospect of bloat, which can slow down the run of elements, the processing of final solutions and therefore the entire search for an acceptable solution. The risk of incorrectly setting this parameter is most likely more worth it than the alternative and its consequences.

With this in mind, by using GEP we might never know a priory what a good limit to element size would be, but having an automatic way to suggest a value in a statistically informed manner would be of interest for future work. In [39] motivation for a parameter-less GA and the actual algorithm are described. Perhaps future work could take inspiration from the parameter-less GA and develop a variation of GEP where head-size and the other exclusive parameters could be automatically filled or suggested.

It's also important to mention that all experiments performed here didn't had GEP parameters optimized statistically. The results were promising and competitive to GP nevertheless.

This seems a good indication that to over-perform GP, configuring parameters with empirical knowledge and based on rough estimations might be sufficient. Nevertheless, for better results, some form of beforehand statistical optimization is recommended.

7. Conclusion

In this thesis, three distinct contributions were presented using GEP. The initial objective was to experiment with, forward and branch out this technique which appears as a very promising alternative to GP.

In the first study, we faced GEP with the Sorting Algorithm Synthesis problem for the first time. Problem which has already been tackled by GP. Under similar conditions, GEP proved capable of generating sorting problems and also outperformed GP by a large margin doing so. The results assert GEP's position established in previous research as a simpler, yet frequently better performing approach than GP.

In the second study we question if investigation developed specifically for GP can be promptly adapted to GEP given the algorithm's similarities and shared theoretical principles. We focused on SGP, a GP variation that sees candidate programs semantics measured and exposed to purposefully developed semantic operators. These extra steps map the roll of phenotype as seen in biological evolution, making SGP a more accurate model of evolution than GP, having the roles of genotype and phenotype arguable more objectively separated. In practical terms, SGP has in previous studies outperformed GP in several problems. The concept is taken a step further by the Geometric Semantic approach built on top of it, which is currently a trending investigation topic with interesting results. Following the same principles of semantic and geometric semantic, we proposed two new approaches, SGEP and GSGEP along with a comparative study in convergence rate facing it with GP, SGP and conventional GEP. The implementation of semantic measuring and manipulation in GEP was very similar to the process for doing so in GP. An extra process might be needed to adapt existing semantic operators that directly manipulate execution trees. On the other hand, the development of semantic operators that take advantage of GEP's simplified element

codifications is now a possibility. Overall the transition was smooth and an indication that developments made over GP can likely be applied directly or easily adapted to GEP. In terms of performance: SGEP comparing to GP an SGP, the approach had mixed results highly correlated to the problem. When comparing to normal GEP, SGEP shown considerably better results in many benchmark instances. Overall results are often good and promising for this approach.

In the third study we proposed a new semantic operator: SCC. The operator strategy is to check parents semantic similarity after selection. If elements are too similar, instead of being proposed for conventional crossover, mutation is applied to one of them in order to increase semantic variation in the population. This strategy was first introduced in the Wang algorithm. SCC differs from the Wang algorithm by using the semantic value of programs instead of a syntactical measurement for comparison. To test the new operator in action, we performed experiments with it in the same GEP/SGEP environment and group of tests as in experiment 2. The results demonstrate slight variations running the benchmarks (both improvements and declines). Comparing the results of using this operator with it's absence showed no significant or relevant change, and thus pointing the necessity of making further studies to assess the potential of this operator and its concept. In terms of practical application, the operator is especially straightforward to include if semantic measurement is already present, which is the case of SGEP and SGP. Very relevant to this investigation is the fact that the development and testing of the algorithm was done completely on a GEP/SGEP implementation, and thus demonstrating how GEP environment can be used as primary platform for developing and testing innovation which is theoretically compatible and targeting both GP, GEP and evolutionary program synthesis approaches in general.

Bibliography

[1] K. E. Kinnear, Jr., "Evolving a sort: Lessons in genetic programming," in *Proceedings of the 1993 International Conference on Neural Networks*, vol. 2, (San Francisco, USA), pp. 881–888, IEEE Press, 28 March-1 April 1993.

[2] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs (3rd Ed.)*. London, UK, UK: Springer-Verlag, 1996.

[3] D. Ashlock, *Evolutionary computation for modeling and optimization*. Springer Science & Business Media, 2006.

[4] C. Ferreira, "Gene expression programming: a new adaptive algorithm for solving problems," *Complex Systems*, vol. 13, no. 2, pp. 87–129, 2001. cite arxiv:cs/0102027Comment: 22 pages, 17 figures.

[5] K. Krawiec and P. Lichocki, "Approximating geometric crossover in semantic space," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pp. 987–994, ACM, 2009.

[6] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*, vol. 1. MIT press, 1992.

[7] J. R. Koza, "Human-competitive results produced by genetic programming," *Genetic Programming and Evolvable Machines*, vol. 11, pp. 251–284, Sep 2010.

[8] W. P. Worzel, J. Yu, A. A. Almal, and A. M. Chinnaiyan, "Applications of genetic programming in cancer research," *The International Journal of Biochemistry & Cell Biology*, vol. 41, pp. 405–413, 2009.

[9] L. Spector, H. Barnum, H. J. Bernstein, and N. Swamy, "Quantum computing applications of genetic programming," *Advances in genetic programming*, vol. 3, pp. 135–160, 1999.

[10] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[11] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[12] Y. Han and M. Thorup, "Integer sorting in o (n/spl radic/(log log n)) expected time and linear space," in *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, pp. 135–144, IEEE, 2002.

[13] K. E. K. Jr., "Generality and difficulty in genetic programming: Evolving a sort.," in *ICGA* (S. Forrest, ed.), pp. 287–294, Morgan Kaufmann, 1993.

[14] L. Spector, J. Klein, and M. Keijzer, "The push3 execution stack and the evolution of control.," in *GECCO* (H.-G. Beyer and U.-M. O'Reilly, eds.), pp. 1689–1696, ACM, 2005.

[15] A. Agapitos and S. M. Lucas, "Evolving efficient recursive sorting algorithms," in *in Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pp. 6–21, IEEE Press, 2006.

[16] S. Shirakawa and T. Nagao, "Evolution of sorting algorithm using graph structured program evolution.," in *SMC*, pp. 1256–1261, IEEE, 2007.

[17] S. Shirakawa, S. Ogino, and T. Nagao, "Graph structured program evolution," in *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation* (D. Thierens, H.-G. Beyer, J. Bongard, J. Branke, J. A. Clark, D. Cliff, C. B. Congdon, K. Deb, B. Doerr, T. Kovacs, S. Kumar, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, R. Poli, K. Sastry, K. O. Stanley, T. Stutzle, R. A. Watson, and I. Wegener, eds.), vol. 2, (London), pp. 1686–1693, ACM Press, 7-11 July 2007.

[18] K. Wolfson and M. Sipper, "Evolving efficient list search algorithms.," in *Artificial Evolution* (P. Collet, N. Monmarché, P. Legrand, M. Schoenauer, and E. Lutton, eds.), vol. 5975 of *Lecture Notes in Computer Science*, pp. 158–169, Springer, 2009.

[19] A. Moraglio, F. Otero, C. Johnson, S. Thompson, and A. Freitas, "Evolving recursive programs using non-recursive scaffolding," in *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pp. 1–8, June 2012.

[20] E. Pennisi, "Encode project writes eulogy for junk dna," *Science*, vol. 337, no. 6099, pp. 1159–1161, 2012.

[21] A. Moraglio, K. Krawiec, and C. G. Johnson, "Geometric semantic genetic programming," in *International Conference on Parallel Problem Solving from Nature*, pp. 21–31, Springer, 2012.

[22] W. Kantschik and W. Banzhaf, *Linear-Tree GP and Its Comparison with Other GP Structures*, pp. 302–312. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

[23] T. PAWLAK, "Semantic genetic programming," 2012.

[24] L. Vanneschi, M. Castelli, and S. Silva, "A survey of semantic methods in genetic programming," *Genetic Programming and Evolvable Machines*, vol. 15, no. 2, pp. 195–214, 2014.

[25] K. Krawiec and T. Pawlak, "Locally geometric semantic crossover," in *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pp. 1487–1488, ACM, 2012.

[26] M. Graff, E. S. Tellez, H. J. Escalante, and S. Miranda-Jiménez, "Semantic genetic programming for sentiment analysis," in *NEO 2015*, pp. 43–65, Springer, 2017.

[27] A. Moraglio, *Towards a geometric unification of evolutionary algorithms*. PhD thesis, University of Essex, 2008.

[28] B. Choudhary, *The Elements of Complex Analysis*. J. Wiley, 1992.

[29] R. W. Hamming, "Error detecting and error correcting codes," *Bell Labs Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[30] G. Syswerda, "Uniform crossover in genetic algorithms," in *Proceedings of the third international conference on Genetic algorithms*, pp. 2–9, Morgan Kaufmann Publishers, 1989.

[31] C. W. Reynolds, "An evolved, vision-based behavioral model of coordinated group motion," *From animals to animats*, vol. 2, pp. 384–392, 1993.

[32] G. Syswerda, "A study of reproduction in generational and steady-state genetic algorithms," in *Foundations of genetic algorithms*, vol. 1, pp. 94–101, Elsevier, 1991.

[33] L. Davis, "Handbook of genetic algorithms," 1991.

[34] T. P. Pawlak, B. Wieloch, and K. Krawiec, "Review and comparative analysis of geometric semantic crossovers," *Genetic Programming and Evolvable Machines*, vol. 16, no. 3, pp. 351–386, 2015.

[35] T. P. Pawlak, "Geometric semantic genetic programming is overkill," in *European Conference on Genetic Programming*, pp. 246–260, Springer, 2016.

[36] R. L. Wang, "A genetic algorithm for subset sum problem," *Neurocomputing*, vol. 57, pp. 463–468, 2004.

[37] R.-L. Wang and K. Okazaki, "An improved genetic algorithm with conditional genetic operators and its application to set-covering problem," *Soft computing*, vol. 11, no. 7, pp. 687–694, 2007.

[38] Z. Q. Chen, R. Wang, R.-V. Sanchez, J. V. de Oliveira, and C. Li, "An adaptive genomic difference based genetic algorithm and its application to memetic continuous optimization," *Intelligent Data Analysis*, vol. 22, no. 2, pp. 363–382, 2018.

[39] G. R. Harik and F. G. Lobo, "A parameter-less genetic algorithm," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1*, pp. 258–265, Morgan Kaufmann Publishers Inc., 1999.