Generalization and Completeness of Evolutionary Computation

Doble Grado en Ingeniería Informática - Matemáticas
Bachelor's Thesis

Daniel Loscos Barroso

UNIVERSIDAD COMPLUTENSE
MADRID

Departamento De Sistemas Informáticos y Computación

Ismael Rodríguez Laguna
Narciso Martí Oliet

June 2018

Contents

Page

Abstract & Keywords **5**
Keywords 5

Resumen y Palabras Clave **5**
Palabras Clave 5

Introduction **7**

1 An Overview of Evolutionary Computation **9**
1.1 Brief Introduction to Evolutionary Computation 9
1.1.1 Definitions 10
1.2 State of the Art 11

2 Operational Semantics for Evolutionary Computation **13**
2.1 General Form 13
2.1.1 State and Syntactic Categories 13
2.1.2 Auxiliary Functions 15
2.1.3 Rules 16
2.2 Genetic Algorithms 17
2.2.1 State and Syntactic Categories 18
2.2.2 Auxiliary Functions 18
2.2.3 Rules 19
2.3 Ant Colony Optimization 20
2.3.1 State and Syntactic Categories 20
2.3.2 Auxiliary Functions 21
2.3.3 Rules 22
2.4 Particle Swarm Optimization 22
2.4.1 State and Syntactic Categories 23
2.4.2 Auxiliary Functions 23
2.4.3 Rules 24
2.5 Semantic Analysis of an Example Genetic Algorithm 25
2.5.1 The Algorithm 25
2.5.2 The Semantics 27
2.5.3 A Sample Execution 27

3 The Best Evolutionary Search Heuristic **31**
3.1 No Free-Lunch Theorems 31
3.2 NFL Implications in Evolutionary Computation 32

4 What about complexity? 37
4.1 Turing-Completeness of Evolutionary Computation 37
4.1.1 Discarded proof 38
4.1.2 Turing-completeness of genetic algorithms 38
4.2 Rice's Theorem for GAs and Implications 43

Conclusion 45

Bibliography 46

Abstract & Keywords

The need of a structured framework for evolutionary computation has been acknowledged. In order to achieve this we designed a set of operational semantics and defined a "general form" of evolutionary computation. Our second approach towards a generalization was to study the relationship between different algorithms and the problems they solve from a performance standpoint. Lastly, we tried to analyze the convergence and complexity of evolutionary algorithms. This led to a set of computability results, the main one being that evolutionary computation is Turing-complete.

Keywords

Evolutionary computation, Swarm intelligence, Turing-completeness, Rice's theorem, No-Free-Lunch theorems, Operational semantics, Genetic algorithm, Particle swarm optimization, Ant colony optimization.

Resumen y Palabras Clave

Se ha reconocido la necesidad de crear un marco estructurado para la computación evolutiva. Para llegar a él diseñamos un conjunto de semánticas operacionales y definimos una "forma general" de la computación evolutiva. Nuestro segundo enfoque para llegar a una generalización fue estudiar la relación existente entre distintos algoritmos y los problemas que solucionan desde el punto de vista de su eficiencia. Finalmente, tratamos de analizar la convergencia y complejidad de los algoritmos evolutivos. Esto nos llevó a obtener una serie de resultados sobre su calculabilidad, siendo el más importante la Turing-completitud de la computación evolutiva.

Palabras Clave

Computacion evolutiva, Inteligencia de enjambre, Turing-completitud, Teorema de Rice, Teoremas *No-Free-Lunch*, Semántica operacional, Algoritmo genético, Optimización de enjambre de partículas, Optimización de colonia de hormigas.

Introduction

Evolutionary computation is a chaotic field of knowledge.[1] Multiple papers are published every year detailing new algorithms that when run with specific parameters obtain good results for a set of benchmarks. But very little results are published to give a formal mathematical structure to evolutionary computation. That is why the objective of this investigation was to work towards that goal.

Evolutionary search strategies are often classified by their form, and not by their behavior. Furthermore, this morphological classification is seldom formal. We believe that having a strong framework for the evolutionary computational model will be useful to attempt generalizations and classifications of evolutionary search heuristics. That is why we decided to build a set of operational semantics for different subsets of evolutionary computation and for its "General Form" defined in Section 2.1.

Since evolutionary algorithms are mostly compared by their efficiency we decided to approach this issue in our study of how these algorithms behave. The No Free-Lunch Theorems helped us discuss how useful benchmarks really are and guide this theoretical investigation. Finally, they lead us to a possible geometrical approach to the ordering of evolutionary computation.

The last thing we wanted to study was the convergence and complexity of evolutionary search algorithms. But as soon as that investigation started, we realized that we should first focus on their computability. The results of that investigations are a set of theorems that assure us that evolutionary computation is Turing-Complete and thus, there is a result similar to Rice's Theorem for genetic algorithms and for evolutionary computation.

[1]See "State of the Art", Section 1.2.

1 An Overview of Evolutionary Computation

1.1 Brief Introduction to Evolutionary Computation

Some problems are easier to solve than others. In computation we call this characteristic of problems complexity. In a more technical way, the complexity of a problem is the mathematical relation that exists between the size of the instance considered and the time or memory it takes to solve it.[1]

In a world where optimization problems of very high complexity[2] exist, accepting only the optimal answer may not always be a viable option. Computer scientists have developed different heuristics and strategies to achieve good non-optimal solutions with a reasonable amount of effort.

Evolutionary search strategies are non-deterministic, iterative heuristics to explore the search space based on the assumption that solutions that are close in the search space will yield similar results. Thus, if the search of the following iterations is directed towards the solutions with best results obtained so far, better results are to be expected.

Most of these heuristics are inspired by natural phenomena such as: natural selection [23], ant routing [4], river dynamics [16] and plenty of others. When we use the term *Evolutionary Computation* we are talking about the process of evolutionary search strategies being executed.

The lack of determinism in evolutionary search strategies makes it difficult to define their convergence. Evolutionary algorithms need stop criteria; if none is provided, they will always try to find a better solution navigating through the search space. Common stop criteria are to run the algorithm for a set amount of iterations or until the rate of improvement in solutions drops below a threshold.

Evolutionary heuristics always try to balance deep search by focusing on the areas

[1]Generally we say that the complexity of a problem p is of the order of $f(n)$ ($p \in \Theta(f(n))$), where n is the size of the instance if and only if exists $X \in \mathbb{R}^+, n_0 \in \mathbb{N}$ such that the instances take at most $X * f(n)$ steps to solve for high enough values of n (for all $n > n_0$) and there is no other function $g(x)$ such that g has the same property and $\lim_{x\to\infty} f(x)/g(x) = \infty$. One example of a linear problem ($p \in \Theta(n)$) is to determine the highest value in an unordered set. If the set was ordered by value, the complexity of finding the highest value would instead be constant ($p \in \Theta(1)$).

[2]Problems whose time to solve will outgrow any polynomial on the size of the instance for sizes sufficiently big. For example, problems with exponential complexity fall in this category.

of the search space that seem more promising with wide search by introducing random deviations on the search path with the objective of dodging local optima in the search for the global optimum.

A few definitions will be provided now for readers who aren't already acquainted with evolutionary computation. Then, a brief overview of the state of the art of the field will be presented, focusing on the issues that concern and motivate this investigation.

1.1.1 Definitions

Here we shall define some basic concepts of evolutionary search strategies:

- **Search Space:** Is the set of all the possible solutions to a given problem.
- **Individual:** Is the representation given in the search space to a solution of the problem.
- **Generation:** Is the set of individuals considered in an iteration of the algorithm. Each generation impacts which individuals will be part of the next ones.
- **Fitness:** Is the numerical value given to a solution of the problem to rate its quality. Higher fitness values mean better solutions.
- **Evolution:** Is the process of subsequently considering new generations using the fitness information of the previous ones and striving for optimality.
- **Mutation:** Is a random alteration that affects one or many individuals of a generation. It is meant to make a jump in the search space and drift away from local optima.
- **Genetic Algorithm (GA):** Is the name given to evolutionary search strategies based on natural selection. In the most standard version, the fittest individuals of a generation are selected, crossed with one another to generate new individuals and mutated to create the next generation.
- **Evolution Strategy:** Is a special instance of a GA where the generations only change by mutation and selection of their individuals.
- **Ant Colony Optimization (ACO):** Is an evolutionary heuristic meant to find paths in a graph. The individuals are called ants and leave **pheromones** along the path they follow. The fitness of the solutions (the paths followed by the different ants) determines the amount of pheromones released and ants are more likely to follow paths with higher pheromone levels.
- **Particle Swarm Optimization (PSO):** Is an evolutionary search strategy where individuals are points in a continuous metric space. Individuals are given **weights** depending on their fitness values and then moved to a new location of the search space. The movement is determined by two forces: a gravity function that draws them closer to heavier and nearer solutions and a random drift that acts at their mutation method.

1.2 State of the Art

Evolutionary computation a is relatively new area of knowledge[3]. The first genetic algorithms were presented in the late fifties [6]. During the mid eighties and early nineties evolutionary computation started booming and the rate of publications grew substantially [23, 4, 18]. The state of the art, however, was just an amalgam of heuristics with little to no order between them and some complementary papers discussing the comparative benefits of certain strategies regarding selective pressure [22], mutation rates and other ingredients for evolutionary computation.

In the last five years, the hype generated around big data has also affected evolutionary computation. Dozens of articles applying evolutionary search heuristics to multi-modal optimization and data mining [13, 9, 12] are being published and this has also arisen an interest in distributed evolutionary computation [7].

One characteristic of this particular state of the art is that it is full of papers presenting algorithms that excel when applied to a set of benchmarks [21, 1, 2, 11, 20, 28, 30, 29]. This tendency may fall into the trap of over-fitting[4] algorithms to perform well at the famous set of benchmarks while not being as good for general purposes[5]. Some researchers have already diagnosed this problem [24, 25, 27] and proposed solutions [17].

We are dealing with a chaotic state of the art where attempting to structure the knowledge achieved for evolutionary computation is a titanic task. Generally, evolutionary search strategies are classified syntactically (by their morphological structure) and not semantically (by their behavior). This is clearly seen in De Jong's "Evolutionary Computation: A Unified Approach" [5], perhaps the book that most extensively describes and classifies different evolutionary algorithms. Also, different heuristics tend to be compared only by their benchmarks.

De Jong also points out that "the EC[6] community must continue to work at developing an overarching framework that provides a consistent view of the field" [5, p. 232]. There have been some attempts to classify and generalize subsets of evolutionary computation regarding the philosophy that guides their search [10], but this kind of publications is scarce. More common are surveys that describe and compare certain heuristics [7, 15], but they again fall short in their attempt to order evolutionary computation.

This is what motivates our investigation. We want to put some sort of mathematical order in the field. We believe that a strong mathematical structure for evolutionary computation can lead to a better understanding of evolutionary computation and, in the end, to better algorithms.

[3]It has around sixty years of history.

[4]We say that an algorithm is over-fitted when it is tailored to do great some concrete instances (the benchmarks, for example) at the cost of performing poorly for others.

[5]See Section 3.1 about No Free-Lunch Theorems.

[6]Evolutionary Computation.

2 Operational Semantics for Evolutionary Computation

The first step towards the generalization of evolutionary search strategies that we considered was their structural analysis. To abstract the structure of these algorithms we attempted to design operational semantics for this kind of computation.

We were successful in our approach and saw that there is a common structure to all evolutionary search heuristics. We call that *General Form.*

Now, the results of our investigation will be presented. The decisions on how to formalize the semantics where heavily inspired by the operational semantics of **While** by Nielson & Nielson [14, pp. 12-14 and 32-36].

2.1 General Form

The goal of this section is to define the operational semantics for the evolutionary computation model. Genetic Algorithms, Evolution Strategies, Ant Colony Optimization, Simulated Annealing, Swarm Algorithms and every other evolutionary search algorithm can be computed with this operational semantics.

More detailed semantics will be provided for the different algorithms, but we believe this is the lowest abstraction level able to generalize the whole of evolutionary computation where a given algorithm A receives an specific instance p of a problem and explores different solutions. The best solution found by the computation is stored in a variable named *Best* and the way this value is outputted is not relevant for us.

2.1.1 State and Syntactic Categories

The **State** of the computation is a function from variables to values. The relevant variables to the computation constitute the following tuple [1]:

$$(Prob, Sols, Best, FPW, Extra) \in \mathbf{Problem} \times \mathbf{Solution}[] \times \mathbf{Solution} \times \mathbf{F/P/W} \times \mathbf{Extra}$$

Since we are aiming to generalize the evolutionary computational model, the categories that define the tuple will be abstract enough to fit any evolutionary search strategy. However we shall define what each category represents and give a glimpse of its structure in

[1]Other variables such array indexes or loop counters are not reflected here. However, they are necessary for the implementation of evolutionary programs.

different evolutionary algorithms.

State is not a constant function, the value associated with each variable may change during the computation. The letter s will represent an instance of **State**. Thus, we shall write $s[A \mapsto b]$ to represent a state where the value associated to A is b and every other variable has the same value as in the state s.

Now, we shall explain the Syntactic Categories necessary for evolutionary computation:

Problem must be able to encapsulate the optimization problem that our algorithm must solve. Not only the abstract problem (eg. TSP[2], nonlinear optimization, etc.) but also the concrete instance and parameters of the problem (eg. function to optimize, cities and paths for TSP, etc.). When $Prob \in$ **Problem** is initialized at the beginning of the computation it will determine how the different functions that the computation requires work.

Solution is the category that represents possible solutions of the problem withing the computation. In a GA **Solution** would represent the codification of the chromosomes; and in an ACO algorithm, **Solution** would be the data type to store the subgraphs that represent the paths followed by ants. **Solution**[] just represents a set of instances of **Solution**. Thus, $Best \in$ **Solution** will be the variable that represents the best solution found by the computation and $Sols \in$ **Solution**[] the set of solutions that are being considered in the current iteration of the algorithm.

The most general category is **F/P/W**, which stands for *Fitness / Pheromones / Weights*. **F/P/W** must be general enough to represent every variable necessary to direct the evolutionary search; e.g. in a swarm based heuristic, $FPW \in$ **F/P/W** would consist of the Weights of each individual, its linear moment, etc.; for a GA, FPW would store the Fitness of every individual, and in an ACO algorithm the Pheromones of each path.

Lastly we introduce the **Extra** category. It will be used to wrap the parameters used to define the auxiliary functions needed for the computation as well as any other variable needed by the specific algorithm that is being run.

Additionally, let **T** [14, p.14] consist of the truth values **tt** (for true) and **ff** (for false), let **Pexp** be the syntactic category used to input the problem into the computation (an instance of **Pexp** will be translated to its corresponding **Problem** instance to be processed), let **Stm** be the set of statements (semantic blocks) that build the evolutionary computation model and **Algorithm** be the set of evolutionary algorithms.

Now, we shall define the meta-variables that will be used to range over constructs of our syntactic categories:

p will range over input problem expressions, **Pexp**.

A will range over the set of evolutionary algorithms, **Algorithm**. Note that we are

[2]Travelling Salesperson Problem: finding the minimally valued route in a positive-valued graph that goes through every single node exactly once.

considering the abstract algorithms, which are independent of the instance being run or the problem at hand.

Lastly, S will range over statements, **Stm**. Also S', S_1 and S_2 will stand for statements.

We assume that the structure for **Pexp** constructs is given elsewhere as it is not relevant for the computation. However, the structure for **Stm** constructs is indeed relevant and given by this:
$S := S_1; S_2$ | `setProb(` p `)` | `generate` | `nextGen` | `evaluate` | `stop` | `compute(` p `)`

`compute( p )` starts the execution of our given A by running `setProb( p )` which translates the problem from **Pexp** to **Problem** and then calls `generate` to create the original set of candidate solutions, `evaluate` to rate them and `stop` to decide whether the algorithm has finished or if the process must be iterated by calling `nextGen` to generate a new set of solutions with the acquired evolutionary knowledge.

The meaning of the statements is further detailed by the following functions and rules.

2.1.2 Auxiliary Functions

Before we begin to introduce the auxiliary functions that will help us define in which way the state varies during the computation, let us define a notation that will help us simplify the presentation of the following functions. Let **Space** be any space relevant to the computation where variables or syntactic constructs may range (e.g. **F/P/W**) and let **Space** not be **Extra**. Then, we define **Space*** as **Space** $\times$ **Extra**.

Since any step of the computation and any change of the state may involve a change in **Extra** for at least one evolutionary algorithm, we state here that every one of the following functions can alter the value of **Extra**. That way we don't have to say it for each of the functions:

- $\mathbb{C}[\![A]\!]$: **Algorithm** $\longrightarrow$ **Extra**
  Is the function that takes and instance A of **Algorithm** and starts the computation by initializing every parameter and variable that the heuristic needs.
- $\mathcal{P}[\![p]\!]_s$: **Pexp** $\times$ **State** $\longrightarrow$ **Problem***
  Is the semantic function for **Pexp** to translate the problem from the input form to the computing form that can be stored as a variable value. The previous state of the computation is used to adapt the problem to the algorithm that is being run.
- $\mathcal{G}[\![]\!]_s$: **State** $\longrightarrow$ **Solution[]***
  This function uses the value of *Prob* given by the state s to generate a new random value for *Sols* concordant to the problem stored.
- $\mathcal{N}ext[\![]\!]_s$: **State** $\longrightarrow$ **Solution[]***
  Uses the values of *Prob*, *Sols* and *FPW* given by the state s to compute stochastically the new value for *Sols* using the information obtained by the previous iterations of the algorithm and in a way that the solutions are concordant to the problem.

- $\mathcal{A}eval[\![]\!]_s : \textbf{State} \longrightarrow \textbf{F/P/W*}$
  Uses the values of *Prob*, *Sols* and *FPW* given by the state s to compute the new value for *FPW* determined by the problem stored and the information obtained by the previous iterations of the algorithm.

- $\mathcal{B}eval[\![]\!]_s : \textbf{State} \longrightarrow \textbf{Solution*}$
  Uses the values of *Prob*, *Sols*, *Best* and *FPW* given by the state s to compute the new value for *Best*, which is the best solution found so far.

- $\mathcal{SC}[\![]\!]_s : \textbf{State} \longrightarrow \textbf{T}$
  This is the stop criteria function. It analyses the whole state and returns **`tt`** if the stop criteria has been met and **ff** if it has not.

Since this is an abstraction of the evolutionary computation model, different evolutionary algorithms will implement these functions differently. Just to give an example, consider two SGA's: the first may return the value of the best individual of the last generation, and the second the best individual amongst all generations. The former has a $\mathcal{B}eval[\![]\!]_s$ function that ignores *Best* whereas the latter does consider it in every generation. Similarly, the parameters of the algorithm directly affect most of these functions.

2.1.3 Rules

Here is the list of rules for Evolutionary Computation in its general form. The execution begins at a initial state s with the [`compute`] rule, by inputting the problem p as a parameter: $\langle \texttt{compute(}\ p\ \texttt{)}, s\rangle$.

The values of the initial state s are not relevant. They will be changed by the inputed problem and no previous information will affect the computation in any way.

We introduce a special notation in the same way as we did for defining the functions. Let *Var* any variable of the *State* other than *Extra*. We define *Var** as $Variable \times Extra$.

Operational Semantics for Evolutionary Computation: General Form

[comp^1] $$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}$$

[comp^2] $$\frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

[set problem] $\langle \texttt{setProb(}\ p\ \texttt{)}, s\rangle \Rightarrow s[Prob^* \mapsto \mathcal{P}[\![p]\!]_s]$

[generate] $\langle \texttt{generate}, s\rangle \Rightarrow s[Sols^* \mapsto \mathcal{G}[\![]\!]_s]$

[next generation] $\langle \texttt{nextGen}, s\rangle \Rightarrow s[Sols^* \mapsto \mathcal{N}ext[\![]\!]_s]$

[evaluate]	$\langle \texttt{evaluate}, s\rangle \Rightarrow s[FPW^* \mapsto \mathcal{A}eval[\![]\!]_s, Best^* \mapsto \mathcal{B}eval[\![]\!]_s]$	
[stop$^{\text{tt}}$]	$\langle \texttt{stop}, s\rangle \Rightarrow s$	if $\mathcal{SC}[\![]\!]_s = \mathbf{tt}$
[stop$^{\text{ff}}$]	$\langle \texttt{stop}, s\rangle \Rightarrow \langle \texttt{nextGen}; \texttt{evaluate}; \texttt{stop}, s\rangle$	if $\mathcal{SC}[\![]\!]_s = \mathbf{ff}$
[compute]	$\dfrac{\langle \texttt{setProb(} p \texttt{)}, s[Extra \mapsto \mathbb{C}[\![A]\!]]\rangle \Rightarrow s'}{\langle \texttt{compute(} p \texttt{)}, s\rangle \Rightarrow \langle \texttt{generate}; \texttt{evaluate}; \texttt{stop}, s'\rangle}$	

For those unfamiliar with Operational Semantics, the meaning of the statements is specified by a transition system with two kinds of configurations:

$\langle S, s\rangle$ representing that the statement S is to be executed from the state s .

s represents a terminal state. These final states often come in the $s[A \mapsto b]$ notation.

A valid transition from configuration α to configuration β is represented by $\alpha \Rightarrow \beta$. The transition system $\dfrac{\gamma \Rightarrow \delta}{\alpha \Rightarrow \beta}$ indicates that we can only transition from configuration α to configuration β if a valid transition from γ to δ can be made.

2.2 Genetic Algorithms

Now we will break down the computation of genetic algorithms in a similar fashion as we modeled the general form of evolutionary computation.

Thus, a more detailed semantics will be provided for genetic algorithms. The goal is to establish a set of operational semantics that are able to showcase the structure of genetic algorithm's computation while being general enough to be applied to particular instances of the genetic algorithms. The representation of the chromosomes, selection method, mutation rate and implementation, stop criteria and other particularities of the genetic algorithm instance should not be relevant to the semantics.

Also, the semantics should be general enough to cover degenerated instances of the genetic algorithm such as evolutionary strategies, where the selection and cross steps are trivial.

The statements and functions added to this semantics are meant to show the importance of sequentially modifying a set of individuals to guide the search. The memory of genetic algorithms resides on its population, as fitness values of an iteration are not relevant for the next generation.

2.2.1 State and Syntactic Categories

The **State** is defined as it was in the general form:

$$(Prob, Sols, Best, FPW, Extra) \in \mathbf{Problem} \times \mathbf{Solution[]} \times \mathbf{Solution} \times \mathbf{F/P/W} \times \mathbf{Extra}$$

However, some syntactic categories can be further specified:

Solution is the category that represents possible solutions of the problem. In this case **Solution** would represent the codification of a single chromosome. **Solution**[] just represents a set of instances of **Solution**. Thus, $Best \in$ **Solution** will be the fittest chromosome found by the computation and $Sols \in$ **Solution**[] the current generation and every other solution needed to compute crosses and mutations.

The **F/P/W** category will represent every variable necessary to direct the evolutionary search and the parameters used to define the auxiliary functions needed for the computation. $FPW \in$ **F/P/W** now consists of the fitness of every individual (which must be recomputed in every generation).

We will use the same meta-variables of the general form, the only difference will be on the structure for **Stm** constructs, now given by:

$$\begin{aligned} S := S_1; S_2 \mid & \ \texttt{setProb(}\ p\ \texttt{)} \mid \texttt{generate} \mid \texttt{select} \mid \texttt{cross} \mid \texttt{mutate} \mid \texttt{nextGen} \\ \mid & \ \texttt{evaluate} \mid \texttt{stop} \mid \texttt{compute(}\ p\ \texttt{)} \end{aligned}$$

We see that new statements are being considered: `select`, `cross` and `mutate` structure the process that was previously abstracted by `nextGen`. This is meant to showcase how the evolution is performed in GAs: fitness-biased selection of individuals to cross, generation of new individuals and mutation of the population. The meaning of the statements is further specified by the following functions and rules.

2.2.2 Auxiliary Functions

- $\mathbb{C}[\![A]\!]$: **Algorithm** $\longrightarrow$ **Extra**
  As seen in the general form.

- $\mathcal{P}[\![p]\!]_s$: **Pexp** $\longrightarrow$ **Problem***
  As seen in the general form.

- $\mathcal{G}[\![]\!]_s$: **State** $\longrightarrow$ **Solution[]***
  As seen in the general form.

- $\mathcal{S}[\![]\!]_s$: **State** $\longrightarrow$ **Solution[]***
  Uses the values of $Prob$, $Sols$ and FPW given by the state s to compute the new value for $Sols$ consisting of the individuals of the previous generation selected for the crossover stage.

- $\mathcal{C}[\![]\!]_s$: **State** $\longrightarrow$ **Solution[]***
  Uses the values of $Prob$, $Sols$ and the FPW given by the state s to compute the new value for $Sols$ determined by a combination of the result of the crossover operation applied to the selected individuals and some of those original individuals.

- $\mathcal{M}[\![]\!]_s$: **State** $\longrightarrow$ **Solution[]***
  Uses the values of *Prob*, *Sols* and *FPW* given by the state s to compute the new value for *Sols* consisting of the individuals already present in *Sols* after modifying (mutating) some of them.

- $\mathcal{A}eval[\![]\!]_s$: **State** $\longrightarrow$ **F/P/W***
  As seen in the general form: for every iteration, calculates the fitness values of the individuals and drops the old ones.

- $\mathcal{B}eval[\![]\!]_s$: **State** $\longrightarrow$ **Solution***
  As seen in the general form.

- $\mathcal{SC}[\![]\!]_s$: **State** $\longrightarrow$ **T**
  As seen in the general form.

Since this is an abstraction of the genetic algorithm computation model, different genetic algorithms will implement these functions differently: Consider a GA with elitism where, in each generation, some of the individuals will be selected as the elite by $\mathcal{S}[\![]\!]_s$; the $\mathcal{C}[\![]\!]_s$ operation will force them into the next value of *Sols* and $\mathcal{M}[\![]\!]_s$ will not modify them. whereas in an elite-less GA, every individual will be subject to modifications by $\mathcal{C}[\![]\!]_s$ or $\mathcal{M}[\![]\!]_s$ and may not be selected by $\mathcal{S}[\![]\!]_s$.

2.2.3 Rules

Here is the list of rules for Genetic Algorithm Computation. In the same way as the general form, the execution begins at a initial state s with the [compute] rule inputting the problem p as a parameter: $\langle \texttt{compute(}\ p\ \texttt{)}, s\rangle$. However, some additional rules have been introduced to showcase the stages of the computation of every new generation.

Operational Semantics for Evolutionary Computation: Genetic Algorithm

[comp^1] $$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}$$

[comp^2] $$\frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

[set problem] $\langle \texttt{setProb(}\ p\ \texttt{)}, s\rangle \Rightarrow s[Prob^* \mapsto \mathcal{P}[\![p]\!]_s]$

[generate] $\langle \texttt{generate}, s\rangle \Rightarrow s[Sols^* \mapsto \mathcal{G}[\![]\!]_s]$

[selection] $\langle \texttt{select}, s\rangle \Rightarrow s[Sols^* \mapsto \mathcal{S}[\![]\!]_s]$

[crossover] $\langle \texttt{cross}, s\rangle \Rightarrow s[Sols^* \mapsto \mathcal{C}[\![]\!]_s]$

[mutation]	$\langle \texttt{mutate}, s\rangle \Rightarrow s[Sols^* \mapsto \mathcal{M}[\![]\!]_s]$	
[next generation]	$\langle \texttt{nextGen}, s\rangle \Rightarrow \langle \texttt{select}; \texttt{cross}; \texttt{mutate}, s'\rangle$	
[evaluate]	$\langle \texttt{evaluate}, s\rangle \Rightarrow s[FPW^* \mapsto \mathcal{A}eval[\![]\!]_s, Best^* \mapsto \mathcal{B}eval[\![]\!]_s]$	
[stop$^{\text{tt}}$]	$\langle \texttt{stop}, s\rangle \Rightarrow s$	if $\mathcal{SC}[\![]\!]_s = \mathbf{tt}$
[stop$^{\text{ff}}$]	$\langle \texttt{stop}, s\rangle \Rightarrow \langle \texttt{nextGen}; \texttt{evaluate}; \texttt{stop}, s\rangle$	if $\mathcal{SC}[\![]\!]_s = \mathbf{ff}$
[compute]	$\dfrac{\langle \texttt{setProb(}\ p\ \texttt{)}, s[Extra \mapsto \mathbb{C}[\![A]\!]]\rangle \Rightarrow s'}{\langle \texttt{compute(}\ p\ \texttt{)}, s\rangle \Rightarrow \langle \texttt{generate}; \texttt{evaluate}; \texttt{stop}, s'\rangle}$	

2.3 Ant Colony Optimization

In this section we will see how we can particularize the operational semantics given for the general form evolutionary of computation when we are dealing with an Ant Colony Optimization algorithm.

ACO algorithms generate a completely new population on each generation. However, the previous iterations influence how the new generation will be generated. These strategies are opposed to their GAs counterparts where some individuals may survive but only the last generation influences the new generation, which is the direct result of a set of operations over the previous population.

We may, therefore, establish that the memory of ACO resides on the pheromones and the memory of GAs lies on the individuals. Even so, the difference in their computation models is relatively small and the overall philosophy is quite similar.

2.3.1 State and Syntactic Categories

The **State** is defined as it was in the general form:

$(Prob, Sols, Best, FPW, Extra) \in \mathbf{Problem} \times \mathbf{Solution}[] \times \mathbf{Solution} \times \mathbf{F/P/W} \times \mathbf{Extra}$

Now, some of the syntactic categories will be further specified:

Problem is as described in the general form. The only variation is that now we are dealing with optimal routing search in a graph, so it will at least represent the graph of the problem as well as other relevant data for the computation.

Solution is the category that represents possible solutions of the route finding problem: A data type to store ordered subgraphs. Metaphorically this subgraphs are the paths that different “ants” will take. We often refer to this solutions as “ants” by association. **Solution**[] represents a set of instances of **Solution**. Thus, $Best \in$ **Solution** will be the most efficient ant found by the computation and $Sols \in$ **Solution**[] the set of ants generated and simulated in the current iteration of the algorithm.

F/P/W’s core will be the representation of the pheromones dropped by the ants of previous iterations. Thus, $FPW \in$ **F/P/W** will store optimality values for more than one iteration! Additionally it will represent the parameters needed for the computation of auxiliary functions.

We will use the same meta-variables of the general form, the only difference will be on the structure for **Stm** constructs, now given by:

$S := S_1; S_2 \mid$ `setProb(` p `)` $\mid$ `generate` $\mid$ `nextGen` $\mid$ `simulate` $\mid$ `evaluate` $\mid$ `stop` $\mid$
$\mid$ `compute(` p `)`

In this case the only new statement considered is `simulate`. This decision was made to highlight the relevance of the fitness evaluation for these methods. While the generation of new individuals is pheromone-biased and relatively simple, it is in the fitness evaluation of the chosen paths (simulation) where pheromones are dropped for the following generation.
The meaning of the statements is given by the following functions and rules.

2.3.2 Auxiliary Functions

- $\mathbb{C}[\![A]\!]$: **Algorithm** $\longrightarrow$ **Extra**
  As seen in the general form.

- $\mathcal{P}[\![p]\!]$: **Pexp** $\longrightarrow$ **Problem***
  As seen in the general form.

- $\mathcal{G}[\![]\!]_s$: **State** $\longrightarrow$ **Solution**[]*
  As seen in the general form.

- $\mathcal{N}ext[\![]\!]_s$: **State** $\longrightarrow$ **Solution**[]*
  Uses the values of $Prob$ and FPW given by the state s to compute the new value for $Sols$ determined by the problem stored and the information obtained by the previous iterations of the algorithm (except for the first iteration, obviously). Note that $Sols$ is no longer relevant for this function.

- $\mathcal{A}eval[\![]\!]_s$: **State** $\longrightarrow$ **F/P/W***
  As seen in the general form. In this particular case this function carries out two different tasks: to compute the fitness of every ant in $Sols$ and to store the pheromones dropped by it.

- $\mathcal{B}eval[\![]\!]_s$: **State** $\longrightarrow$ **Solution***
  As seen in the general form.

- $\mathcal{SC}[\![]\!]_s$: **State** $\longrightarrow$ **T**
  As seen in the general form.

Again, different ant colony algorithms may implement these functions differently.

2.3.3 Rules

The rules for Ant Colony Optimization Computation are the same as the ones of the general form.

Operational Semantics for Evolutionary Computation: Ant Colony Optimization

[comp^1] $$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}$$

[comp^2] $$\frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

[set problem] $$\langle \texttt{setProb(} p \texttt{)}, s\rangle \Rightarrow s[Prob^* \mapsto \mathcal{P}[\![p]\!]]$$

[generate] $$\langle \texttt{generate}, s\rangle \Rightarrow s[Sols^* \mapsto \mathcal{G}[\![]\!]_s]$$

[next generation] $$\langle \texttt{nextGen}, s\rangle \Rightarrow s[Sols^* \mapsto \mathcal{N}ext[\![]\!]_s]$$

[evaluate] $$\frac{\langle \texttt{simulate}, s\rangle \Rightarrow s'}{\langle \texttt{evaluate}, s\rangle \Rightarrow s'[Best^* \mapsto \mathcal{B}eval[\![]\!]_s]}$$

[simulate] $$\langle \texttt{simulate}, s\rangle \Rightarrow s[FPW^* \mapsto \mathcal{A}eval[\![]\!]_s]$$

[stop$^{\text{tt}}$] $$\langle \texttt{stop}, s\rangle \Rightarrow s \qquad \text{if } \mathcal{SC}[\![]\!]_s = \mathbf{tt}$$

[stop$^{\text{ff}}$] $$\langle \texttt{stop}, s\rangle \Rightarrow \langle \texttt{nextGen}; \texttt{evaluate}; \texttt{stop}, s\rangle \qquad \text{if } \mathcal{SC}[\![]\!]_s = \mathbf{ff}$$

[compute] $$\frac{\langle \texttt{setProb(} p \texttt{)}, s[Extra \mapsto \mathbb{C}[\![A]\!]]\rangle \Rightarrow s'}{\langle \texttt{compute(} p \texttt{)}, s\rangle \Rightarrow \langle \texttt{generate}; \texttt{evaluate}; \texttt{stop}, s'\rangle}$$

2.4 Particle Swarm Optimization

The last particularization of the operational semantics for evolutionary computation that we are going to develop in this chapter is the operational semantics for PSO algorithms.

In PSO algorithms individuals of the solutions population move around the search space trying to find the optimal solution. This process is equivalent to that of GAs since we can see each movement as a new generation of solutions generated after the previous population by following a set of rules.

Again, the memory of PSO lies on the individuals and not so much on the weights that have to be recalculated on each generation. However, the way the new positions of the particles are computed is structurally different from that of GAs.

Thus, we consider it an interesting example to illustrate how the same philosophy of GAs can be implemented in a different computational model and still have its operational semantics included in the general form of evolutionary computation.

2.4.1 State and Syntactic Categories

The **State** is defined as it was in the general form:

$(\mathit{Prob}, \mathit{Sols}, \mathit{Best}, \mathit{FPW}, \mathit{Extra}) \in \textbf{Problem} \times \textbf{Solution[]} \times \textbf{Solution} \times \textbf{F/P/W} \times \textbf{Extra}$

However, some of syntactic categories can be further specified:

In this case **Solution** will represent the position, direction and speed of a particle in the search space. **Solution**[] represents a set of **Solution** particles. Thus, *Best* $\in$ **Solution** will be the optimal point found by the computation so far (ignoring its momentum) and *Sols* $\in$ **Solution**[] the current position and momentum of each particle of the swarm.

The **F/P/W** category will represent every variable necessary to direct the evolutionary search and the parameters used to define the auxiliary functions needed for the computation. *FPW* $\in$ **F/P/W** now consists of the fitness-based weights of every particle (which must be recomputed in every generation).

We will use the same meta-variables of the general form, the only difference will be on the structure for **Stm** constructs, now given by:

$S := S_1; S_2$ | `setProb(` p `)` | `generate` | `divert` | `aim` | `move` | `nextGen`
| `evaluate` | `stop` | `compute(` p `)`

Stm is changed in a similar fashion as it did for GAs: `divert`, `aim` and `move` are introduced to detail the form in which `nextGen` does its job. First `divert` introduces a random influence in the future movement, then `aim` targets the movement towards the objective and finally `move` combines those two forces to set the new position of the particle.

The `divert` and `aim` are actually interchangeable for the computation, but for the functions and rules that follow, we will assume that `divert` is prior to `aim`.

2.4.2 Auxiliary Functions

- $\mathbb{C}[\![A]\!]$: **Algorithm** $\longrightarrow$ **Extra**
  As seen in the general form.

- $\mathcal{P}[\![p]\!]_s$: **Pexp** $\longrightarrow$ **Problem***
  As seen in the general form.

- $\mathcal{G}[\![]\!]_s$: **State** $\longrightarrow$ **Solution[]***
  As seen in the general form.

- $\mathcal{D}[\![]\!]_s$: **State** $\longrightarrow$ **Solution[]***
  Uses the values of *Prob*, given by the state s to compute new random momentum values for *Sols* while not changing any positions. These momentum changes are called diversions.
- $\mathcal{A}[\![]\!]_s$: **State** $\longrightarrow$ **Solution[]***
  Uses the values of *Prob*, *Sols* and *FPW* given by the state s to compute attractions between particles of *Sols*, then merges this attraction with the momentum values obtained by $\mathcal{D}[\![]\!]_s$ and sets a new momentum value for each particle in *Sols*.
- $\mathcal{M}[\![]\!]_s$: **State** $\longrightarrow$ **Solution[]***
  Uses the values of *Sols* given by the state s to compute the new positions of every particle in *Sols* based on its previous position and momentum (direction and speed).
- $\mathcal{A}eval[\![]\!]_s$: **State** $\longrightarrow$ **F/P/W***
  As seen in the general form: every iteration it calculates and updates the fitness values of the individuals based on their position inside the search space. The fitness of every particle determines its weight for the attraction stage.
- $\mathcal{B}eval[\![]\!]_s$: **State** $\longrightarrow$ **Solution***
  As seen in the general form.
- $\mathcal{SC}[\![]\!]_s$: **State** $\longrightarrow$ **T**
  As seen in the general form.

Again, different particle swarm algorithms may implement these functions differently.

2.4.3 Rules

Here is the list of rules for PSO Computation. In the same way as the general form, the execution begins at a initial state s with the [compute] rule inputting the problem p as a parameter: $\langle \texttt{compute(}\ p\ \texttt{)}, s\rangle$. However, some additional rules have been introduced to showcase the stages of the computation in each iteration.

Operational Semantics for Evolutionary Computation: Particle Swarm Optimization

[comp^1] $$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}$$

[comp^2] $$\frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

[set problem] $\langle \texttt{setProb(}\ p\ \texttt{)}, s\rangle \Rightarrow s[Prob^* \mapsto \mathcal{P}[\![p]\!]_s]$

[generate] $\langle \texttt{generate}, s\rangle \Rightarrow s[Sols^* \mapsto \mathcal{G}[\![]\!]_s]$

[divert] $\langle \texttt{divert}, s\rangle \Rightarrow s[Sols^* \mapsto \mathcal{D}[\![]\!]_s]$

[aim] $\langle \texttt{aim}, s\rangle \Rightarrow s[Sols^* \mapsto \mathcal{A}[\![]\!]_s]$

[move] $\langle \texttt{move}, s\rangle \Rightarrow s[Sols^* \mapsto \mathcal{M}[\![]\!]_s]$

[next generation] $\langle \texttt{nextGen}, s\rangle \Rightarrow \langle \texttt{divert}; \texttt{aim}; \texttt{move}, s'\rangle$

[evaluate] $\langle \texttt{evaluate}, s\rangle \Rightarrow s[FPW^* \mapsto \mathcal{A}eval[\![]\!]_s, Best^* \mapsto \mathcal{B}eval[\![]\!]_s]$

[stop$^{\mathrm{tt}}$] $\langle \texttt{stop}, s\rangle \Rightarrow s$ if $\mathcal{SC}[\![]\!]_s = \mathbf{tt}$

[stop$^{\mathrm{ff}}$] $\langle \texttt{stop}, s\rangle \Rightarrow \langle \texttt{nextGen}; \texttt{evaluate}; \texttt{stop}, s\rangle$ if $\mathcal{SC}[\![]\!]_s = \mathbf{ff}$

[compute] $$\frac{\langle \texttt{setProb(}\ p\ \texttt{)}, s[Extra \mapsto \mathbb{C}[\![A]\!]]\rangle \Rightarrow s'}{\langle \texttt{compute(}\ p\ \texttt{)}, s\rangle \Rightarrow \langle \texttt{generate}; \texttt{evaluate}; \texttt{stop}, s'\rangle}$$

2.5 Semantic Analysis of an Example Genetic Algorithm

To finish this chapter, a semantic analysis within our framework is performed on an example genetic algorithm. An execution of the algorithm through our operational semantics ends this section.

2.5.1 The Algorithm

The algorithm is designed to count from zero to the inputted integer. For example, if the input is 5, the output should be $0, 1, 2, 3, 4, 5$. To keep things simple, we can assume that the algorithm only works correctly for inputs lower than 1000.

An implementation in pseudo-code of this algorithm (where we assume `X` is the input) could be:

```
0.   INPUT X
1.      int input = read(X);
2.      int[1000] candidates, sol, best;
3.      bool[1000] blacklist;
4.      for (int i = 0; i < input; i++)
5.          candidates[i] = i; blacklist[i] = true;
6.      sol[0] = 0; int nsol = 1;
7.      int fpw = eval(sol, best, nsol, blacklist, input);
8.      while (fpw < input+1)
```

```
    sol[nsol] = random(candidates,blacklist);
    fpw = eval(sol, best, nsol, blacklist);
RETURN enumerate(best,input)
```

In the previous code, some functions have been used but not yet defined. The function `read(X)` parses the input given by the user and translates it to the corresponding integer number. The function `enumerate(list,n)` takes a list of elements and outputs `list[0]`, `list[1]`, ..., `list[n]`. The function `random(list,booleanlist)` returns a random element `list[i]` from `list` such that `booleanlist[i] == true`.

The implementation of `eval(sol, best, nsol, blacklist)` is described below. Even though the return argument is an integer, the input variables modified inside the function remain modified in the environment that calls the function:

```
INPUT Sol Best Nsol Blist In
  i = 0;
  while (Sol[i] == i && i < Nsol)
    i++;
  if (i == Nsol)
    Nsol++;
    for (int j = 0; j < In; j++)
      Blist[j] = true;
  else
    Blist[Sol[i]] = false;
  for (int k = 0; k < i; k++)
    Best[k] = Sol[k];
RETURN i (Best Nsol Blist)
```

Here is how this algorithm works: First it translates the input `X` from its original syntax to an integer data type manageable by the program. Then, we fill the array `candidates` up with the integers from 0 to `X` both included and set to true the corresponding positions in the blacklist array.

The population of this GA only has one member: `sol` that stores the numbers found so far in the correct order. The variable `nsol` serves as an index to signal the index of the first unconfirmed position. Since we force `sol[0]` to be 0, it has the correct value and we can set `nsol` to 1.

Then, the first evaluation comes in, which does several things: It sets `fpw` to the amount of confirmed correct positions of `sol`. If every position so far has been confirmed as correct it augments `nsol` by one and resets the blacklist to allow every candidate. Otherwise, it does not change `nsol` and blacklists the candidate considered in `sol[nsol]` by setting its value to false in the blacklist. Also the `fpw` confirmed positions from `sol` are copied to `best`.

The stop criteria is to have `input+1` correct positions in `sol`, thus completing the count from zero to the input. If the criteria is met, then an enumeration of the numbers is outputted. If it has not been met yet the first unconfirmed position of `sol` is assigned a new random, not-blacklisted value from `candidates` and a new evaluation takes place.

2.5.2 The Semantics

Now we will present how this algorithm fits our operational semantics for GAs:

First, the input `X` in its natural form belongs to **Pexp**, while its processable representation (`input` plus `candidates`) is an instance of **Problem**.

The *State* here will be the set of every variable displayed in lines `0` to `11` (except`i` which only serves as an index). *Prob* will be the combination of `input` and `candidates`. *Sols* will only have one element, `sol` and naturally *Best* will be represented by `best`. *FPW*, the fitness values will only store one value, the fitness of `sol` represented by `fpw`. Finally *Extra* will consist of the blacklist `blacklist` and the index `nsol`. Knowing the data types of the `variables` we can easily describe the data structures that constitute **State**.

According to the changes in the *State* that happen throughout the computation we can determine the lines of code that correspond to the statements of **Stm**. Thus, the following statements provide the semantics for:

- `compute(` p `)`: lines `0` to `11`.
- `setProb(` p `)`: lines `0`,`1`,`4` and `5`.
- `generate`: line `6`.
- `evaluate`: line `7` first and `10` later.
- `stop`: lines from `8` to `11`.
- `nextGen`: line `9`.
- `mutate`: line `9` (it implements `nextGen`).

The avid reader will have noticed that we have not associated any lines of code with S_1;S_2, `select` or `cross`. The first case makes sense for every algorithm, since that statement is just a linking tool for other statement. The statements `select` or `cross` are not necessary in this case to implement `nextGen`, so they exist only as identity functions that do not change the state s within line `9`.

2.5.3 A Sample Execution

Now, we will execute the algorithm for an input TWO that translates into the instance "count to two". Note that since random elements are involved, this exact execution path may not always happen for the same input. We assume we were given a random seed that from $\{0, 1, 2\}$ chooses 1 for the first and second rolls and that chooses 2 between $\{0, 2\}$ on its third roll as our random number generator. We also assume the starting *State* s_0 is completely undefined.

Before we begin, we will introduce the following notation for referencing the values stored in a state s_i: 'u' will represent an undefined integer, 'ub' an undefined boolean, 't' will represent *true* and 'f' $false$. Thus, the initial state would be written as

$s_0 = (\mathit{Prob}, \mathit{Sols}, \mathit{Best}, \mathit{FPW}, \mathit{Extra})$ = ([u,[u,u,u]], [u,u,u],[u,u,u],u,[u,[ub,ub,ub]])

Given these conditions, the execution of our algorithm would be:

Computation Trace With Input *TWO*:

```
1 $\langle \texttt{compute(}TWO\texttt{)}, s_0\rangle \Rightarrow \langle \texttt{generate}; \texttt{evaluate}; \texttt{stop}, s_1\rangle$
    1.1 $\langle \texttt{setProb(}TWO\texttt{)}, s_0\rangle \Rightarrow s_1 = s_0[\mathit{Prob} \mapsto [2,[0,1,2]], \mathit{Extra} \mapsto$ [u,[t,t,t]] ]

2 $\langle \texttt{generate}; \texttt{evaluate}; \texttt{stop}, s_1\rangle \Rightarrow \langle \texttt{evaluate}; \texttt{stop}, s_2\rangle$
    2.1 $\langle \texttt{generate}, s_1\rangle \Rightarrow s_2 = s_1[\mathit{Sols} \mapsto [0,\mathrm{u},\mathrm{u}], \mathit{Extra} \mapsto$ [1,[t,t,t]] ]

3 $\langle \texttt{evaluate}; \texttt{stop}, s_2\rangle \Rightarrow \langle \texttt{stop}, s_3\rangle$
    3.1 $\langle \texttt{evaluate}, s_2\rangle \Rightarrow s_3 = s_2[\mathit{FPW} \mapsto 1, \mathit{Best} \mapsto$ [0,u,u]$, \mathit{Extra} \mapsto$ [2,[t,t,t]] ]

4 $\langle \texttt{stop}, s_3\rangle \Rightarrow \langle \texttt{nextGen}; \texttt{evaluate}; \texttt{stop}, s_3\rangle$

5 $\langle \texttt{nextGen}; \texttt{evaluate}; \texttt{stop}, s_3\rangle \Rightarrow \langle \texttt{evaluate}; \texttt{stop}, s_4\rangle$
    5.1 $\langle \texttt{nextGen}, s_3\rangle \Rightarrow \langle \texttt{select}; \texttt{cross}; \texttt{mutate}, s_3\rangle$

    5.2 $\langle \texttt{select}; \texttt{cross}; \texttt{mutate}, s_3\rangle \Rightarrow \langle \texttt{cross}; \texttt{mutate}, s_3\rangle$
        5.2.1 $\langle \texttt{select}, s_3\rangle \Rightarrow s_3$

    5.3 $\langle \texttt{cross}; \texttt{mutate}, s_3\rangle \Rightarrow \langle \texttt{mutate}, s_3\rangle$
        5.3.1 $\langle \texttt{cross}, s_3\rangle \Rightarrow s_3$

    5.4 $\langle \texttt{mutate}, s_3\rangle \Rightarrow s_4 = s_3[\mathit{Sols} \mapsto [0,1,\mathrm{u}]\ ]$

6 $\langle \texttt{evaluate}; \texttt{stop}, s_4\rangle \Rightarrow \langle \texttt{stop}, s_5\rangle$
    6.1 $\langle \texttt{evaluate}, s_4\rangle \Rightarrow s_5 = s_4[\mathit{FPW} \mapsto 2, \mathit{Best} \mapsto$ [0,1,u]$, \mathit{Extra} \mapsto$ [3,[t,t,t]] ]

7 $\langle \texttt{stop}, s_5\rangle \Rightarrow \langle \texttt{nextGen}; \texttt{evaluate}; \texttt{stop}, s_5\rangle$

8 $\langle \texttt{nextGen}; \texttt{evaluate}; \texttt{stop}, s_5\rangle \Rightarrow \langle \texttt{evaluate}; \texttt{stop}, s_6\rangle$
    8.1 $\langle \texttt{nextGen}, s_5\rangle \Rightarrow \langle \texttt{select}; \texttt{cross}; \texttt{mutate}, s_5\rangle$

    8.2 $\langle \texttt{select}; \texttt{cross}; \texttt{mutate}, s_5\rangle \Rightarrow \langle \texttt{cross}; \texttt{mutate}, s_5\rangle$
        8.2.1 $\langle \texttt{select}, s_5\rangle \Rightarrow s_5$

    8.3 $\langle \texttt{cross}; \texttt{mutate}, s_5\rangle \Rightarrow \langle \texttt{mutate}, s_5\rangle$
        8.3.1 $\langle \texttt{cross}, s_5\rangle \Rightarrow s_5$

    8.4 $\langle \texttt{mutate}, s_5\rangle \Rightarrow s_6 = s_5[\mathit{Sols} \mapsto [0,1,1]\ ]$

9 $\langle \texttt{evaluate}; \texttt{stop}, s_6\rangle \Rightarrow \langle \texttt{stop}, s_7\rangle$
    9.1 $\langle \texttt{evaluate}, s_6\rangle \Rightarrow s_7 = s_6[\mathit{Extra} \mapsto$ [3,[t,f,t]] ]
```

10 $\langle \texttt{stop}, s_7 \rangle \;\Rightarrow\; \langle \texttt{nextGen}; \texttt{evaluate}; \texttt{stop}, s_7 \rangle$

11 $\langle \texttt{nextGen}; \texttt{evaluate}; \texttt{stop}, s_7 \rangle \;\Rightarrow\; \langle \texttt{evaluate}; \texttt{stop}, s_8 \rangle$
  11.1 $\langle \texttt{nextGen}, s_7 \rangle \;\Rightarrow\; \langle \texttt{select}; \texttt{cross}; \texttt{mutate}, s_7 \rangle$

  11.2 $\langle \texttt{select}; \texttt{cross}; \texttt{mutate}, s_7 \rangle \;\Rightarrow\; \langle \texttt{cross}; \texttt{mutate}, s_7 \rangle$
    11.2.1 $\langle \texttt{select}, s_7 \rangle \;\Rightarrow\; s_7$

  11.3 $\langle \texttt{cross}; \texttt{mutate}, s_7 \rangle \;\Rightarrow\; \langle \texttt{mutate}, s_7 \rangle$
    11.3.1 $\langle \texttt{cross}, s_7 \rangle \;\Rightarrow\; s_7$

  11.4 $\langle \texttt{mutate}, s_7 \rangle \;\Rightarrow\; s_8 = s_7[Sols \mapsto [0,1,2]\;]$

12 $\langle \texttt{evaluate}; \texttt{stop}, s_8 \rangle \;\Rightarrow\; \langle \texttt{stop}, s_9 \rangle$
  12.1 $\langle \texttt{evaluate}, s_8 \rangle \;\Rightarrow\; s_9 = s_8[FPW \mapsto 3, Best \mapsto \text{[0,1,2]}, Extra \mapsto \text{[4,[t,t,t]]}\;]$

13 $\langle \texttt{stop}, s_9 \rangle \;\Rightarrow\; s_9$

The previous transitions are justified by RNG[3] and the following rules and transitions:

1 $\mapsto$ [compute] & 1.1
  1.1 $\mapsto$ [set problem]

2 $\mapsto$ [comp^2] & 2.1
  2.1 $\mapsto$ [generate]

3 $\mapsto$ [comp^2] & 3.1
  3.1 $\mapsto$ [evaluate]

4 $\mapsto$ [stop$^{\text{ff}}$]

5 $\mapsto$ [comp^2], 5.1, 5.2, 5.3 & 5.4
  5.1 $\mapsto$ [next gen]
  5.2 $\mapsto$ [comp^2] & 5.2.1
    5.2.1 $\mapsto$ [selection]
  5.3 $\mapsto$ [comp^2] & 5.3.1
    5.3.1 $\mapsto$ [crossover]
  5.4 $\mapsto$ [mutation]

6 $\mapsto$ [comp^2] & 6.1
  6.1 $\mapsto$ [evaluate]

7 $\mapsto$ [stop$^{\text{ff}}$]

8 $\mapsto$ [comp^2], 8.1, 8.2, 8.3 & 8.4
  8.1 $\mapsto$ [next gen]
  8.2 $\mapsto$ [comp^2] & 8.2.1
    8.2.1 $\mapsto$ [selection]
  8.3 $\mapsto$ [comp^2] & 8.3.1
    8.3.1 $\mapsto$ [crossover]
  8.4 $\mapsto$ [mutation]

9 $\mapsto$ [comp^2] & 9.1
  9.1 $\mapsto$ [evaluate]

10 $\mapsto$ [stop$^{\text{ff}}$]

11 $\mapsto$ [comp^2], 11.1, .2, .3 & .4
  11.1 $\mapsto$ [next gen]
  11.2 $\mapsto$ [comp^2] & 11.2.1
    11.2.1 $\mapsto$ [selection]
  11.3 $\mapsto$ [comp^2] & 11.3.1
    11.3.1 $\mapsto$ [crossover]
  11.4 $\mapsto$ [mutation]

12 $\mapsto$ [comp^2] & 12.1
  12.1 $\mapsto$ [evaluate]

13 $\mapsto$ [stop$^{\text{tt}}$]

[3]Random Number Generation.

This example GA is very similar to the one that we will use to prove of the Turing-completeness of genetic algorithms in Section 4.1.2., but before we address the computability of evolutionary algorithms, we will consider their performance of and what can we learn from it.

3 The Best Evolutionary Search Heuristic

Now that we have shown that there is a common structure to evolutionary computation, we will focus on the performance of evolutionary search strategies.

Every year a large number of papers get published on this field [21, 1, 2, 11, 20, 28, 30, 29]. Some of them claim to have developed an algorithm that excels in performance when applied to a set of benchmarks.

When trying to put a certain order into the chaotic state of the art of evolutionary computation a group of algorithms outperforming consistently another group could provide information. Perhaps even a generalization of each of those two groups could be made and their implicit differences stated.

In 1995, David H. Wolpert and William G. Macready published their article *No Free-Lunch Theorems in Search* [24] in which they stated that under a certain set of conditions and with no extra information about the problem at hand, no algorithm is better than any other. Two years later, they provided similar results for black-box[1] optimization [25]. However, in 2005, they publish *Coevolutionary Free Lunches* [26] where they acknowledge the existence of heuristics for self-play that do not fall under the NFL[2] theorems.

In the following section we shall provide a detailed look into the NFL theorems, what do they really mean [27] and how they are relevant for evolutionary computation and what can they tell us about the generalization of evolutionary search strategies.

3.1 No Free-Lunch Theorems

The NFL theorems [24] state that $\forall a$:

$$\sum_{f \in F} P(f(\overrightarrow{d})|f, m, a) = |Y|^{|X|-m}$$

where X is the finite search space of solutions, Y is the finite space of fitness values for solutions, $f : X \to Y$ is a fitness function, $f(\overrightarrow{d}) = \{f(d_1), f(d_2), ..., f(d_m)\}$ (abusing the notation), F is the finite set of all possible mappings between X and Y, m is the total number of unique fitness evaluations done so far, $\overrightarrow{d} \in D$ is the set of the m points

[1]Optimization of unknown functions where no information about them is available a priori.
[2]No Free-Lunch.

explored so far, $D = \mathcal{P}(X)$ and $a : D \to X$ is the algorithm that chooses the next point to explore. Without loss of generality we can assume that $a(\overrightarrow{d})$ will never return a point in $\overrightarrow{d}$.

This means that the average fitness of every single algorithm over all possible fitness functions after the same amount of iterations is constant. Thus we can expect every algorithm to perform the same when applied to all possible fitness functions. Since we don't know the nature of the fitness function we are dealing with, no search strategy can be deemed, a priori, better than any other **if every f is as likely to be our black-box**.

Note that another important assumption is made: both X and Y are finite. This may seem theoretically irrational, but is technically true when dealing with machine computation.

The results of [24] are extended in [25] with a set of results designed specifically for optimization problems. Perhaps, the most interesting one is the extension on NFL for variable fitness functions, which states that $\forall f_1 \in F$ and $\forall a$:

$$\sum_{t^m \in T} P(f_m(\overrightarrow{d})|f_1, t^m, m, a) \text{ is constant.}$$

Where T is the set of all bijective functions $t : F \to F$, $f_{i+1} = t_i(f_i)$, $f_m = t^m(f_1)$ and t^m is the functional composition of m t_i functions $\in T$, which is also bijective from F to itself. Note that the successive do no depend on the observations $\overrightarrow{d}$ made so far and that t_i can equal the identity function for every $i \in \mathbb{N}$ to make f constant across every iteration

This, however, does not imply that every algorithm performs the same when applied once to every possible $f \in F$. Also, non bijective functions from F to F can bias the fitness function and open the door for free-lunches[3].

In [26] a free-lunch example is given for self-play. They use the subtleties we just mentioned to achieve this. First they provide a training oracle f to the algorithm and then they let it face off the real problem g, with the correspondence between those two functions not being bijective. The result is that the information obtained from evaluations of f and knowing the correspondence between the possible f's and g's, the algorithm can choose better points for g.

3.2 NFL Implications in Evolutionary Computation

Perhaps the single most important implication of the NFL theorems is that if an algorithm a_1 outperforms another algorithm a_2 for a subset $H \subset F$ then a_2 outperforms a_1 for the complementary subset of F.

This implication motivates the idea presented in [25] of the geometrical view of the NFL theorems. Assuming F is represented by a subspace of a bigger dimension space K and the set of all possible algorithms A is another subspace of K defined by any set of

[3]Algorithms that, in average and without previous knowledge, perform better than others.

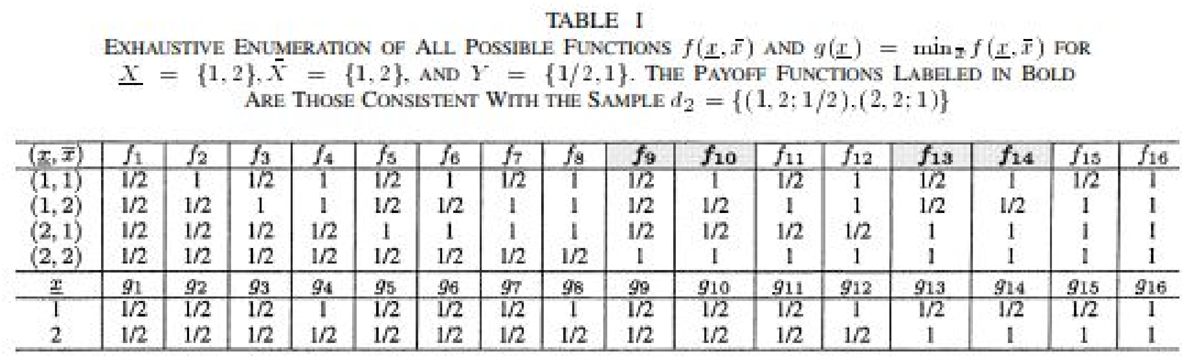

TABLE I

EXHAUSTIVE ENUMERATION OF ALL POSSIBLE FUNCTIONS $f(\underline{x}, \bar{x})$ AND $g(\underline{x}) = \min_{\bar{x}} f(\underline{x}, \bar{x})$ FOR $\underline{X} = \{1, 2\}, \bar{X} = \{1, 2\}$, AND $Y = \{1/2, 1\}$. THE PAYOFF FUNCTIONS LABELED IN BOLD ARE THOSE CONSISTENT WITH THE SAMPLE $d_2 = \{(1, 2; 1/2), (2, 2; 1)\}$

$(\underline{x}, \bar{x})$	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	$\mathbf{f_9}$	$\mathbf{f_{10}}$	f_{11}	f_{12}	$\mathbf{f_{13}}$	$\mathbf{f_{14}}$	f_{15}	f_{16}
(1, 1)	1/2	1	1/2	1	1/2	1	1/2	1	1/2	1	1/2	1	1/2	1	1/2	1
(1, 2)	1/2	1/2	1	1	1/2	1/2	1	1	1/2	1/2	1	1	1/2	1/2	1	1
(2, 1)	1/2	1/2	1/2	1/2	1	1	1	1	1/2	1/2	1/2	1/2	1	1	1	1
(2, 2)	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1	1	1	1	1	1	1	1

$\underline{x}$	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8	g_9	g_{10}	g_{11}	g_{12}	g_{13}	g_{14}	g_{15}	g_{16}
1	1/2	1/2	1/2	1	1/2	1/2	1/2	1	1/2	1/2	1/2	1	1/2	1/2	1/2	1
2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1/2	1	1	1	1

Figure 3.1: This table included in [26] represents all possible training functions and their answer to every input during training. After two experiments (sample d_2) only four candidate functions remain (labeled in bold). Since we know which g_i functions correspond to our f candidates, we can conclude that 2 is a more promising point than 1. See [26] for further details.

points that satisfy that $\forall a_1, a_2 \in A$:

$$\sum_{f \in F} \langle \vec{f}, \vec{a_1} \rangle = \sum_{f \in F} \langle \vec{f}, \vec{a_2} \rangle$$

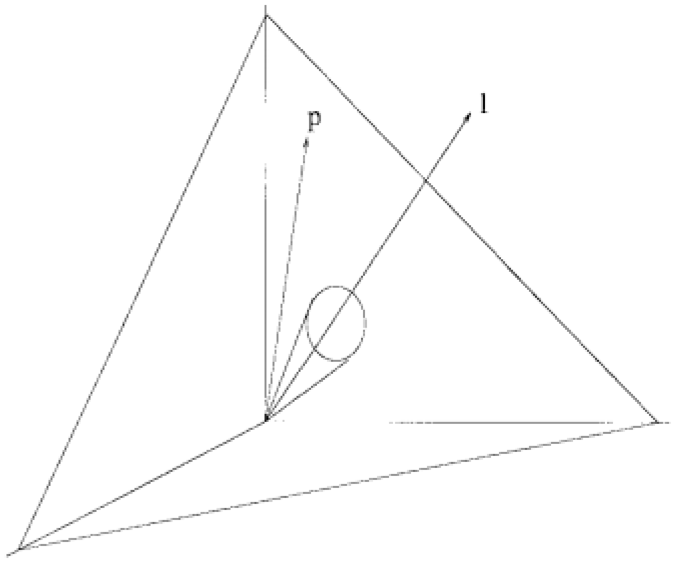

Figure 3.2: This is an image included in [25]. In this representation the function space K is tridimensional, F is the surface of the simplex shown in the figure, p is the vector representing the problem and points to its function f, vector i is the uniform prior over this space and different algorithms a give different vectors $\vec{a}$ lying in the cone surrounding i. The algorithm that will perform best will be the algorithm in the cone having the largest inner product with f. See [25] for further details.

Thus, a theoretical measure of how adequate an algorithm is for a certain problem is established as a dot product between the position vectors of the algorithm and the problem $\langle \vec{f}, \vec{a} \rangle$. The paper also recognizes the limitations of this approach since it is very difficult to establish an actual geometry and dimension for F an A. Two algorithms that may intuitively appear similar, may behave very differently when applied to the same f.

If we were able to formalize, even for a very specific set F, the topology and measure notions of F and A, that result would provide a lot of information towards the mathematical classification and generalization of evolutionary search heuristics. Applying different algorithms to the same benchmarks provides punctual information about these geometric notions, but considering the cardinal of A, any finite number of measures will still constitute a null-measure set.

Thus, we would need different techniques to map these spaces and their metrics. Until that formalization is figured out, assuming that previous benchmarks are useful to predict the behavior of an algorithm with a different problem is, at best, wishful thinking.

When applying the NFL theorems to the actual analysis of algorithm performance in the real world two facts make the theorem practically useless: The NFL is based on the fitness per unique evaluation (or oracle call), but does not take into account the time it takes for different algorithms to decide the next point to check. As the authors point out in [27], if when dealing with a black-box optimization problem, every possible f had the same probability of being the function we are trying to optimize, then NFL holds; but in real life this is hardly ever the case and

$$\sum_{f \in F} P(f(\vec{d})|f, m, a) \times P(f)$$

is not the same for every $a \in A$, where $P(f)$ is the probability of f being the function we are optimizing.

In [26], we see an example of this reasoning. An auxiliary function is used to narrow down $P(f)$ (or $P(g)$ in this case) and thus produce a free-lunch heuristic. This is a very clever technique, and applying it formally to a macro scale would suppose a breakthrough in optimization.

To sum up, all these results and open questions make very unlikely the idea that a mathematical equivalence between all evolutionary algorithms can be achieved. But obtaining enough information to classify evolutionary algorithms as instances of meta-heuristics based on their performance and not on their structure still seem far fetched.

We believe, however, that investigating the mathematical relationship between the algorithms and the problems they aim to solve is key towards this generalization and classification effort.

Another major conclusion (even if it is not related to the generalization) is that the use of a specific fixed set of benchmarks to measure performance is a bad idea: suppose we want a good algorithm to solve TSP. If we measure the performance of different algorithms with always the same set of benchmark instances; then, we do not get information about its behavior for other instances of TSP. Also if it verifies the conditions for the NFL theorems we know that its average performance for every other possible problem (including the other instances of TSP) decreases as the benchmark score increases.

Now suppose a random benchmark that chooses an unpredictable instance of TSP among a subset of TSP instances (e.g. graphs with 23 to 25 cities). Then we could statistically determine which algorithms perform best for that subset of TSP instances without having to worry about it being over-fitted for one specific instance. In [17] a system to implement random benchmarking is proposed to solve this problem.

Since no other conclusions could be obtained from this approach, in the next section we will take the computability and complexity one.

4 What about complexity?

We say that a sequence X_n of values converges towards X if $lim_{n \to \infty} X_n = X$. In computation this notion is often applied to states, and we say that a computation converges to a state s whenever that computation stops with s as its final state.

Evolutionary search heuristics are non-deterministic and often applied to black-box optimization problems. This makes the common notions of convergence difficult to apply to these kind of algorithms.

The most common take on this problem is to talk about convergence in probability: A sequence X_n of random variables converges in probability towards X if

$$\forall \varepsilon > 0 \lim_{n \to \infty} \mathcal{P}(|X_n - X| > \varepsilon) = 0.$$

This implies that a stochastic optimization algorithm converges in probability to the global optimum if the probability of it not finding the optimum in a set amount of iterations goes to zero when the number of iterations goes to infinity.

With this notion we can establish that a random optimization algorithm that chooses for each iteration an element from a discrete search space X (where every element of X has a non-zero probability to be chosen) converges in probability to the global optimum.

We decided that studying the convergence and complexity of different evolutionary search strategies could shed some light over the problem of classifying and generalizing evolutionary algorithms.

What we discovered, however, is that evolutionary computation is Turing-complete. Rice's Theorem can be applied to evolutionary computation and thus semantic properties of evolutionary algorithms are undecidable.

4.1 Turing-Completeness of Evolutionary Computation

The goal of this section is to prove that evolutionary computation is Turing-complete, which means that any program that can run on a Turing Machine (TM) can also be run using an evolutionary algorithm.

We achieve this by proving the Turing-completeness of genetic algorithms, which are a subset of all evolutionary algorithms. As far as we know, this is something that has not been done yet and that has serious implications for our investigation.

Before we begin with the proof, we want to clarify a possible misconception: Some people may think that a genetic algorithm for genetic programing that uses in its genome a set of Turing-complete instructions is able to compute every possible program and is, thus, Turing-complete. This is not true since the GA is not doing any computational work and the program derived from this algorithm is not running outside of the fitness function. Therefore, this proves that we can use a GA to find any possible program but not that the GA itself can run any possible program.

Our goal will be to find a GA that solves the problem of finding the complete computation history of any given Turing machine.

4.1.1 Discarded proof

Our first approach to the problem was to include a Universal Turing Machine (UTM)[1] as described in [3, p. 20] inside the fitness function of the GA. *Prob* would be a representation of the TM whose computation history we want to find, *Sols* would be a set of computation histories from the initial state and stored symbols (the representation used is not relevant) and *Best* would be the computation history that has reached a furthest correct state. The fitness function would use its UTM to verify the correctness of the computation history and assign a fitness proportional to the number of correct steps achieved from the initial state.

The idea for the crossover and mutation functions is that they would force the individuals from *Sols* to copy *Best*'s history up to the first incorrect step and only generate mutations from that point onwards.The algorithm would stop whenever the computation of the simulated TM stopped.

Even though this GA would fit into our theoretical model shown in Section 2.2, other descriptions of what a GA is may put limitations on the fitness function that prevents us from running arbitrary code in them. In other words, the Turing-completeness of the algorithm relied too much in the fitness function for our comfort.

We wanted to design a GA that was Turing-complete and where the Turing-completeness was more heavily aligned with the structure of the computation. Looking into the undecidability of Post's Correspondence Problem (PCP) as described in [8, pp. 392-403] we came up with a more elegant, simpler and universal GA.

4.1.2 Turing-completeness of genetic algorithms

Before we begin with the description of the Turing-complete GA, we will introduce some definitions and results that will help us build it.

[1]A Turing Machine able to compute what another TM computes when given a certain input.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a Turing Machine as described in [8, p. 319], let $w \in \Sigma^*$ be its input, and let T be the set of (a, b) tuples (or "tiles") used to represent (M, w)[2] for the Modified Post Correspondence Problem (MPCP) as shown in [8, pp. 392-403].

Then, T is a finite set of tuples deterministically generated. The set of tuples T can only be generated if the TM never moves left from the initial position and never writes blanks. Luckily, for every TM, there is an equivalent TM with these restrictions.

In the way that T is generated we will have a tuple to represent the initial state and input of (M, w), a closing tuple to represent the end of the computation, tiles to represent the accepting states of M and that simulate the consumption of every symbol in the tape, a (x, x) tuple for each $x \in \Gamma$, a $(\#, \#)$ tuple for the separator symbol # and a tuple to represent every transition in δ. Furthermore, if the first tuple for our MPCP partial solution is the one that represents the initial state, then there is one and only one tuple $t \in T$ able to extend that partial solution. This allows the extension of the partial solution to effectively emulate the computation of (M, w).

To prove the undecidability of PCP, the halting problem[3] of Turing machines is reduced to MPCP and MPCP is reduced to PCP. Thus, if PCP was decidable, so would be the halting problem (which is undecidable). This proof uses the tuples in T to establish that MPCP has a solution if and only if M stops with w as its input. Moreover, the shortest possible solution for MPCP will represent the complete computation history (states, pointer and symbols stored in each step of the computation) of M with input w.

With these ingredients we can design a GA that tries to solve MPCP by subsequently finding the next tuple (the next step of the computation) until its MPCP is solved. We will prove that our GA converges, for every step, to the next step of the computation. This means that the GA can successfully emulate the behavior and tape status of every possible (M, w) tuple.

The idea for this GA is to mimic genetic programming and attempt to design a program that arranges the tuples t_i from T into incrementing partial solutions of MPCP until a complete solution is found.[4] To keep things as simple as possible, the chosen population will be just one individual. This individual is represented as a linked list of instructions we will call *tiles*. We should interpret $tile_i$ as the instruction to place $t_i \in T$ next.

The fitness function always returns the amount of correctly arranged tiles and stores the best partial solution found so far (it is only necessary to return this solution at the end of the computation). Both the selection and crossover stages are unnecessary and will not change anything in the individuals. The changes from the mutation stage only affect the end of the program that the GA is building: if the last instruction is incorrect, it is substituted by another random one; if it is correct, a new random one is added after

[2]Turing Machine M when run with input w.

[3]The halting problem is to determine whether an arbitrary program with an input will finish its computation or continue to run forever.

[4]The algorithm we will now describe is very similar to the example one given in Section 2.5.2 and we encourage the reader to revisit it to have a clearer picture of this one.

it expanding the program.

With these elements we can guarantee, for every step, its convergence in probability to the next correct one. However, introducing a blacklist of tiles via *Extra* to filter the ones already discarded in the current step of the MPCP, we can guarantee regular convergence. To accelerate the convergence of the implementation, the instruction $tile_0$ (correspondent to the initial tile from T) cannot be introduced randomly into the individual.

Finally, the GA will only stop when the closing tile has been placed correctly. Let's break this GA down:

Data Structures

- **Tile:** is a tuple of strings. Represents the instruction to place the tuple (tile) of the MPCP instance that has those strings as its elements.
- **Problem:** is an array of elements of type **Tile** and an integer. Which represent the set T of tiles for the MPCP codification of the inputted (M, w) and $|T|$ respectively.
- **Solution:** consists of a linked list of **Tile** elements. Which represent the ordered instructions to arrange the tuples of MPCP.
- **F/P/W:** is just one integer. It counts the number of correct instructions chained so far in the only element of *Sols*.
- **Extra:** is an array of booleans. Represents a blacklist of tiles that resets every time a new correct one is added to the partial solution and filters out the incorrect ones tried so far for the current iteration.

Function Definitions

- $\mathbb{C}[\![A]\!]$:
  Returns an instance of **Extra** with every value set to true.
- $\mathcal{P}[\![p]\!]_s$:
  Processes (M, w) to create T as in [8, pp. 392-403], then codes every tuple in T as instances of **Tile** and gives a **Problem** instance with the initial tile in the first position of the array, the closing tile in the second position and all the others in the next $|T| - 2$ positions. The integer value of **Problem** is set to $|T|$.
- $\mathcal{G}[\![]\!]_s$:
  Gives a linked list with 2 elements, the first one is the first element of the array *Prob*, the second one is chosen randomly among all other positions of *Prob* with index lower than the integer stored. Also returns an instance of **Extra** equal to *Extra* with the corresponding position set as false.
- $\mathcal{S}[\![]\!]_s$:
  Always returns the only element in *Sols* and *Extra*.
- $\mathcal{C}[\![]\!]_s$:
  Always returns the only element in *Sols* and *Extra*.

- $\mathcal{M}[\![]\!]_s$:
  If *FPW* equals the number of tiles in *Best*, appends a new random tile from *Prob* (with index $i \in [1, N-1]$, where N is the integer stored and the first index of the array is 0), and then returns the augmented list of tiles and an instance of **Extra** with every value set to true except the value with index i (this resets the blacklist). Otherwise, a new tile from *Prob* is chosen randomly with index $i \in [1, N-1]$ where $Extra[i]$ is true. The returned values are a linked list of tiles consisting of *Best* with $Prob[i]$ at the end; and an instance of **Extra** equal to *Extra* excepting the value with index i, that is set to false.

- $\mathcal{A}eval[\![]\!]_s$:
  Concatenates all the strings from the tiles of *Sols* into two strings a and b. a is the concatenation of the first elements of each tuple in the order given by *Sols* and b is analogous for the second elements.
  If *FPW* has not been initialized but b cannot be expressed as a followed by other characters, the function returns 1 and *Extra*.
  If *FPW* has not been initialized and b can be expressed as a followed by other characters, the function returns 2 and an instance of **Extra** with every element set to true.
  Otherwise, the function returns *FPW* and *Extra* if b cannot be expressed as a followed by other characters and $b+1$ and an instance of **Extra** with every element set to true if it can.

- $\mathcal{B}eval[\![]\!]_s$:
  Returns the first *FPW* tiles of *Sols* linked in the same order.

- $\mathcal{SC}[\![]\!]_s$:
  Only stops if the number of tiles in *Best* equals *FPW* and the last tile of *Best* equals $Prob[1]$.

Proof of Turing-completeness

Theorem 4.1.1. *There is a Turing-complete genetic algorithm: There is a genetic algorithm that is able to simulate every step of the computation of (M, w) for every Turing machine M and input w.*

Proof. To prove this theorem, we will consider the GA described above and use induction over the number of steps computed. Let a be the concatenation of the first elements of each tile in *Best* respecting their order and let a_i, b_i the strings between the $(i-1)$th and the (i)th separators in a and b respectively.

The induction hypothesis (IH) will be the following: if the computation of (M, w) does n transitions; then, at some point, b will have $n+2$ separators #, no more symbols after the $(n+2)$th separator and each $b_i \subset b$ will correctly represent the state, pointer, and tape of M after i computation steps.

Induction Basis: For $n = 0$ we have to prove that at some point $\exists b_0 \subset b$ such that b_0 represents the original configuration of (M, t). This is granted after the first evaluation, since the first tile of the chromosome is determined to be the first one of the MPCP by the algorithm.

Induction Step: For $n = k+1$ we use the (IH) and see how the algorithm behaves at that point in time. b_k is a string with a finite set of symbols. The construction of the tile set T seen in [8, pp. 392-403] implies that one and only one tile can augment a partial solution of MPCP. Also, there are no tiles whose first or second elements contain 0 symbols.

Thus, if the GA described always finds the next tile to augment its current partial solution stored in *Best*, after adding $|b_k|^5 + 1$ tiles we will have added at least one more separator to a (to match the one present in b), which implies adding it to b since the only tiles that have a separator are $(\#, \#)$ and $(q_f\#\#, \#)$.

All that remains to be proven is that the tiles added have correctly computed b_k and that the GA described always finds the next tile to augment *Best*. The first thing is given by [8, pp. 392-403] and the second one is proven in the following lemma.

Lemma 4.1.1.1. *The GA described above always finds the next tile to augment its current partial solution stored in Best.*

Proof. The construction of T gives us that for every partial solution c of the MPCP instance exists one and only one $tile_i \in T$ such that c followed by t_i is also a partial solution [8, pp. 392-403]. Applying this property to *Best* once it has been initialized we get that there is one and only one $Prob[i] \in Prob$ such that the concatenation of *Best* and $Prob[i]$ is also a partial solution.

The mutation step chooses a random, not initial and not blacklisted tile from *Prob*. In the first iteration after modifying *Best* the probability of choosing the correct tile is $\frac{1}{|Prob|-1}$. The divisor decreases by one unit for every blacklisted tile, and each erroneous choice blacklists one tile. In the worst case scenario, after $|Prob| - 2$ iterations, the probability of choosing the right tile is 1. The expected number of iterations to find the right tile is $\frac{|Prob|^2-3|Prob|+2}{2}$.

□

This completes the proof of the induction step. Thus, we have proven that the GA described can correctly emulate every step of the computation of any given (M, w).

□

Analogously it can be proven that our GA halts if and only if (M, w) does, which makes the Halting Problem undecidable for evolutionary computation; but this is much simpler by applying the properties of T:

If (M, w) halts, then a tile that represents an accepting state in its first string will be added to our partial solution. If this is a $(q_f\#\#, \#)$ tile, $\mathcal{SC}[\![]\!]_s$ will return **tt** and our GA will halt. If it isn't that kind of tile, then the GA will keep on adding tiles like (Xq_f, q_f), (Xq_fY, q_f) or (q_fY, q_f), that eliminate the symbols of the tape until the only viable option is a $(q_f\#\#, \#)$ tile. A more detailed explanation and examples can be seen in [8, pp. 392-403].

[5] $|b_k|$ is the number of symbols present in the string b_k.

4.2 Rice's Theorem for GAs and Implications

In this section we will prove that there are no decidable non-trivial semantic properties for the set of all evolutionary algorithms. This means that for every property $\mathcal{P}$ regarding only the result (and not the structure) of an evolutionary search strategy such as there is an algorithm that satisfies it and one that doesn't; determining whether a given evolutionary algorithm satisfies $\mathcal{P}$ or not is undecidable.

Rice's Theorem [19] states that "All non-trivial semantic properties of programs are undecidable." It is often also stated as "Every non-trivial property of the Recursively Enumerable languages is undecidable"[8, p.388]. Since evolutionary computation is Turing-complete, and Rice's Theorem is a consequence of Turing-completeness; a Rice's Theorem restricted to evolutionary computation exists. We wanted, however, to provide a detailed proof tailored with the aid of our operational semantics from Section 2. Please note that this is not strictly necessary.

Theorem 4.2.1. *All non-trivial semantic properties of genetic algorithms are undecidable.*

Proof. Let $\mathcal{P}$ be a non-trivial semantic property of genetic algorithms. Let's assume there is a function g able to decide whether $\mathcal{P}$ holds for any GA. Without loss of generality we can assume that g returns NO for a genetic algorithm that never halts[6]. Then we would be able to decide the halting problem for Turing machines.

To do this we only have to consider a GA a that satisfies $\mathcal{P}$ and modify it properly (since $\mathcal{P}$ is non-trivial and g returns NO for our never-stopping GA, we can assure that such a exists). Given (M, w) the way to determine whether (M, w) halts or not is to take a and add to it all the elements that made the GA from the previous section Turing-complete. We will also need to add a single boolean to **Extra**.

This modified $a*$ is a GA that first uses all the added data structures to simulate (M, w) in the way we described in the previous section, the only difference is that whenever the $\mathcal{SC}[\![]\!]_s$ function would return **tt** for the first time, it returns **ff** instead but changes the boolean value we added to *Extra*. This changes the parts of the state we were using for the simulation and the functions behavior from the Turing simulation to the originals from a.

This way, $a*$ behaves first as a Turing simulator and then, after halting, as a. Thus, $\mathcal{P}$ holds for $a*$ if and only if (M, w) halts, so $g(a*) = YES$ if and only if (M, w) halts (and $g(a*) = NO$ if and only if it does not halt).

This is absurd because it contradicts the undecidability of the Halting Problem for Turing machines. Thus, no such decision function g can exist for any non-trivial semantic property. □

[6]The only other option is that it returns YES, which makes for an analogous proof where the chosen GA a would not satisfy $\mathcal{P}$.

Theorem 4.2.2. *All non-trivial semantic properties of evolutionary search algorithms are undecidable.*

Proof. Let $\mathcal{P}$ be a non-trivial semantic property of evolutionary search algorithms. If there was a function f able to decide whether $\mathcal{P}$ holds for any evolutionary search algorithm or not, then f would be deciding that property for genetic algorithms, since those are a subset of evolutionary search algorithms. The previous theorem makes this is absurd. $\square$

The existence of a Rice's Theorem for evolutionary computation makes it that much harder to make a semantic classification for evolutionary search heuristics. Also, seeing that the halting problem for genetic algorithms with variable length chromosomes is undecidable, trying to determine the complexity of this kind of algorithms is just impossible.

However, these issues can still be interesting to approach in subsets of evolutionary computation where perhaps no Rice's Theorem exists. In any case, much work still needs to be done in the formalization of evolutionary computation.

Conclusion

To summarize the work done: First we constructed a set of operational semantis for evolutionary computation to use them as a framework to analyze evolutionary search strategies. Then we took a look at the performance analysis being done on evolutionary computation and the No-Free-Lunch Theorems. Lastly, a set of computability results arose when we tried to study the complexity and convergence of evolutionary algorithms.

Our first conclusion, a consequence of Rice's theorem, is that the only possible generalization for the whole of evolutionary computation has to be a syntactic one. Determining whether a given evolutionary algorithm meets a given semantic-only non-trivial property is undecidable. Regarding the syntactic generalization, we have proposed a set of operational semantics that may turn out to be useful for this goal.

From the NFL theorems we can conclude that using a set of known benchmarks to determine the efficiency of a genetic algorithm is likely to lead to over-fitting and not necessarily provide any information regarding the use of those algorithms for other purposes. In the same way as [17], we want to encourage the use of blind random benchmarks from a big pool of instances in a way that every possible test's results become public to prevent the marketing of generally-useless over-fitted heuristics. This is a direct consequence of the NFL theorems: if an algorithm verifies the NFL theorems conditions, having the best performance at a set of benchmarks B implies having the average worst performance across the set $\bar{B}$ of every other possible problem.

The other big conclusion from our analysis of the NFL theorems is that there could be a geometric structure inherent to all evolutionary algorithms and the problems they solve. This mathematical structure could provide information useful for the classification and generalization of subsets of evolutionary computation. Nowadays, the state of the art is far from close to these kinds of results, but an effort in such a direction seems very promising to us.

Finally, we wanted to note that there are subsets of evolutionary computation whose Turing-completeness is not yet decided and subsets where Rice's theorem does not hold. Perhaps a semantic approach to the generalization and classification of these areas can be done. But trying to resolve those problems exceeded the scope of this work.

Bibliography

[1] M. Alden and R. Miikkulainen. Marleda: Effective distribution estimation through markov random fields. Technical Report TR-13-18, Department of Computer Science, The University of Texas at Austin, Austin, TX, November 2013.

[2] M. Alden, A.-J. van Kesteren, and R. Miikkulainen. Eugenic evolution utilizing a domain model. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 279–286, 2002.

[3] S. Arora and B. Barak. *Computational Complexity, A Modern Approach*. Cambridge University Press, 2009.

[4] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. pages 134–142, Paris, France, 1991. Actes de la Première Conférence Européenne sur la Vie Artificielle, Elsevier Publishing.

[5] K. A. De Jong. *Evolutionary Computation: A Unified Approach*. MIT press, 2006.

[6] A. S. Fraser. Simulation of genetic systems by automatic digital computers i. introduction. *Australian Journal of Biological Sciences*, 10(4):484–491, 1957.

[7] Y.-J. Gong, W.-N. Chen, Z.-H. Zhan, J. Zhang, Y. Li, Q. Zhang, and J.-J. Li. Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. *Applied Soft Computing*, 34:286–300, 2015.

[8] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Pearson Education, 2001.

[9] H. Ishibuchi, N. Akedo, and Y. Nojima. Behavior of multiobjective evolutionary algorithms on many-objective knapsack problems. *IEEE Transactions on Evolutionary Computation*, 19(2):264–283, 2015.

[10] Q. Kang, J. An, L. Wang, and Q. Wu. Unification and diversity of computation models for generalized swarm intelligence. *International Journal on Artificial Intelligence Tools*, 21(03):1240012, 2012.

[11] S. Kirkpatrick, C. D. G. Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[12] B. Li, J. Li, K. Tang, and X. Yao. Many-objective evolutionary algorithms: A survey. *ACM Computing Surveys (CSUR)*, 48(1):13, 2015.

[13] A. Mukhopadhyay, U. Maulik, S. Bandyopadhyay, and C. A. C. Coello. A survey of multiobjective evolutionary algorithms for data mining: Part i. *IEEE Transactions on Evolutionary Computation*, 18(1):4–19, 2014.

[14] H. R. Nielson and F. Nielson. *Semantics With Applications: A Formal Introduction.* Wiley Professional Computing, 1999.

[15] J. L. Olmo, J. R. Romero, and S. Ventura. Swarm-based metaheuristics in automatic programming: a survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 4(6):445–469, 2014.

[16] P. Rabanal, I. Rodríguez, and F. Rubio. Using river formation dynamics to design heuristic algorithms. In *International Conference on Unconventional Computation*, pages 163–177, Kingston, ON, Canada, 2007. Springer.

[17] P. Rabanal, I. Rodríguez, and F. Rubio. Assessing metaheuristics by means of random benchmarks. In *International Conference on Computational Science 2016 (ICCS 2016), Procedia Computer Science, vol.80*, pages 289–300, 2016.

[18] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 25–34, New York, NY, USA, 1987. ACM.

[19] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.

[20] S. Salcedo-Sanz, J. D. Ser, I. Landa-Torres, S. Gil-López, and A. Portilla-Figueras. The coral reefs optimization algorithm: An efficient meta-heuristic for solving hard optimization problems. In *Proceedings, 15th Applied Stochastic Models and Data Analysis (ASMDA2013)*, pages 751–758, 2013.

[21] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[22] D. Whitley. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[23] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85, 1994.

[24] D. H. Wolpert and W. G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, July 1995.

[25] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[26] D. H. Wolpert and W. G. Macready. Coevolutionary free lunches. *IEEE Transactions on Evolutionary Computation*, 9:721–735, 2005.

[27] D. H. Wolpert and W. G. Macready. What the no free lunch theorems really mean: How to improve search algorithms. In *Ubiquity Symposium: Evolutionary Computation and the Processes of Life*, pages 2:1–2:15, New York, NY, USA, 2013. ACM.

[28] X. S. Yang and S. Deb. Cuckoo search via lévy flights. In *2009 World Congress on Nature Biologically Inspired Computing (NaBIC)*, pages 210–214, Dec 2009.

[29] Y. Zhong, J. Lin, L. Wang, and H. Zhang. Hybrid discrete artificial bee colony algorithm with threshold acceptance criterion for traveling salesman problem. *Information Sciences*, 421:70 – 84, 2017.

[30] Y. Zhou, Q. Luo, H. Chen, A. He, and J. Wu. A discrete invasive weed optimization algorithm for solving traveling salesman problem. *Neurocomputing*, 151:1227 – 1236, 2015.