Swarm and Evolutionary Computation 68 (2022) 100982

Contents lists available at ScienceDirect

Swarm and Evolutionary Computation

journal homepage: www.elsevier.com/locate/swevo

Generalization and completeness of stochastic local search algorithms☆

Daniel Loscos, Narciso Martí-Oliet, Ismael Rodríguez*

Dpto. Sistemas Informáticos y Computación. Facultad de Informática, Universidad Complutense de Madrid, 28040 Madrid, Spain

ARTICLE INFO

Keywords:
Stochastic local search
Evolutionary computation
Swarm intelligence
Formal languages
Operational semantics
Generalization
Computability
Turing-completeness

ABSTRACT

We generalize Stochastic Local Search (SLS) heuristics into a unique formal model. This model has two key components: a common structure designed to be as large as possible and a parametric structure intended to be as small as possible. Each heuristic is obtained by instantiating the parametric part in a different way. Particular instances for Genetic Algorithms (GA), Ant Colony Optimization (ACO), and Particle Swarm Optimization (PSO) are presented. Then, we use our model to prove the Turing-completeness of SLS algorithms in general. The proof uses our framework to construct a GA able to simulate any Turing machine. This Turing-completeness implies that determining any non-trivial property concerning the relationship between the inputs and the computed outputs is undecidable for GA and, by extension, for the general set of SLS methods (although not necessarily for each particular method). Similar proofs are more informally presented for PSO and ACO.

1. Introduction

Tackling optimization problems is a necessity in virtually all industrial and scientific areas, yet finding the optimal solution is computationally intractable for many of these problems. Thus, we rather focus on looking for reasonable suboptimal solutions for typical problem instances, normally by means of heuristics. Stochastic Local Search (SLS) algorithms [1], and particularly notable families of them such as Evolutionary and Swarm optimization methods [2,3], can be good choices to tackle many optimization problems. In these methods, a set of simple entities interact with each other according to simple rules, changing across iterations and collaboratively constructing and improving solutions to the problem under consideration. For instance, in a Genetic Algorithm [4], chromosomes (entities) mix with each other and constitute evolving solutions by themselves, whereas in Ant Colony Optimization [5,6] some ants (agents) collaboratively draw a candidate solution (a path or set of paths) on a graph while they iteratively traverse it. SLS heuristics in general, and evolutionary and swarm algorithms in particular, have been applied to a wide variety of domains (see e.g. [7–11]).

During the last years, this field has witnessed an explosion of new methods which, in some cases, have been motivated mainly by the beauty of the natural procedure they try to replicate, rather than by displaying an actually different behavior guaranteeing new interesting mathematical properties during the search [12]. Given the huge number of parameters and alternative steps each method admits, making a new algorithm reach good solutions for a known benchmark (or a significant part of it) is often a matter of using a powerful parameter tuner [13–15] to optimize it for the specific benchmark until it achieves its best, sometimes over-fitted, results —or even just a matter of a long try-and-fail designer time until it works fine (sometimes even by chance, sometimes just because the modifications made it be roughly equivalent to other previously known good algorithm) [16]. Many newly proposed methods seem to be equivalent to others up to some small changes in some algorithm steps, yielding the feeling that we are speaking about a sort of continuous space of algorithms rather than conceptually different algorithms deserving different names.

Aiming to put some order in this field, some classifications of these algorithms based on their morphological structure, natural metaphor, or basic mechanics have been proposed (see e.g. [1,17,18]), as well as some informally defined generalizations embracing just a few specific algorithms (e.g. [19]) and many surveys describing and comparing methods (e.g. [20,21]). Perhaps the framework providing more generality is the Generalized Local Search Machine (GLSM) [22], which models SLS strategies as non-deterministic finite state machines. It illustrates the run-time behaviour and control of SLS algorithms and provides a modelling tool for hybrid and non-cooperative SLS strategies, although it is incomplete: it does not represent termination conditions, neighbourhood relationships, or the definition of the search space or solution set. Additionally, some of the extensions proposed by the authors (namely the ones for cooperative and evolutionary problems) completely alter the framework, instead of slightly modifying some features.

☆ Narciso Martí-Oliet and Ismael Rodríguez are also with Instituto de Tecnología del Conocimiento. Work partially supported by projects TIN2015-67522-C3-3-R, PID2019-108528RB-C22, and by Comunidad de Madrid as part of the program S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union.

* Corresponding author.
E-mail addresses: dloscos@ucm.es (D. Loscos), narciso@ucm.es (N. Martí-Oliet), isrodrig@ucm.es (I. Rodríguez).

https://doi.org/10.1016/j.swevo.2021.100982
Received 22 November 2020; Received in revised form 12 July 2021; Accepted 3 September 2021
Available online 15 September 2021

To the best of our knowledge, the literature lacks a fully formal (and general enough) definition of the common and the different parts of SLS methods, such that any method can be the result of taking the framework and instantiating it in some specific way. Such framework would enable the definition of new and existing algorithms in a common language, easing their classification and direct comparison. By using a common field language, new methods could be developed via a scientific and standardized variation of other known methods, their properties could be more easily investigated, and the results could be seamlessly classified and added into the knowledge base.

In this paper we present a formal language with some fixed structure and some parametric structure such that all SLS methods we are aware of can be the result of instantiating the latter according to the target method. Note that, since all (standard or variant) SLS methods are *algorithms*, any programming language would actually let us implement all of them. Our language, however, will not pursue the generality of programming languages but the opposite: having the *least* generality and freedom needed to generalize stochastic local search, in particular by making the common structure be as large and the parametric one be as small as possible. This way, the language aims to make all basic algorithmic elements visible, atomic, and clear for comprehension, manipulation and testing.

In order to prove the capability of our general formal model to aid in the discovery of new non-trivial properties, we will use it to prove a disheartening meta-property: the general impossibility of checking any non-trivial property regarding the results of SLS methods. One could argue that this impossibility is an obvious consequence of their randomness: since they involve non-deterministic choices, we cannot predict their result with certainty. However, we will show that their unpredictability lies on deeper reasons far beyond their randomness, as their behavior would remain unpredictable even if they behaved in a deterministic way (or, alternatively, even if we knew in advance the results of all coin flips determining all their random choices during their execution). In particular, this implies that their behavior cannot be predicted *in general* by means of statistics, as they would remain unpredictable even in the absence of randomness.

The key to discover the unpredictability of SLS methods will be proving that they are Turing-complete. Rice's theorem [23,24] shows that determining if a program written in a Turing-complete language fulfills a non-trivial property regarding only its observable behavior (i.e. the relationship between its inputs and its outputs) is *undecidable*.[1] Our formal model will let us prove that SLS methods are Turing-complete in general, so Rice's theorem will apply to them as well.[2]

One could argue that the Turing-completeness of such methods could be just a trivial consequence of assuming that they are a too general and free concept —or rather the trivial consequence of representing them with a too general *model*. For instance, suppose our general model of SLS methods includes a post-processing step which maps the best solution found by a particular algorithm upon its termination, along with the problem input, into the actual algorithm's solution to be returned to the user. Suppose this post-processing step can be *any* algorithm. Then, the Turing-completeness would be the trivial consequence of our freedom to make that step be any algorithm.[3] Quite on the contrary, our proof of Turing-completeness will show that we do not need that much freedom at all —regardless of whether it is actually allowed by the model. In fact, the Turing-completeness will arise from the intrinsic basic mechanics of these algorithms, even if each involved step is defined in an extremely simple and minimalist way.

In order to emphasize the fact that the generality of the model is not necessary to achieve the Turing-completeness, we will prove the Turing-completeness for a *particular* instance of our general model representing a particular SLS method: a Genetic Algorithm (GA). This GA will be such that, given a Turing machine to be simulated and its corresponding input, it iteratively constructs individuals representing longer pieces of the *history* (i.e. sequence of traversed configurations) of that Turing machine for that input. The GA will necessarily tend to create such individuals because correctly enlarging that history with subsequent Turing machine steps according to the transitions of the machine will increase their fitness —although the fitness function will *not* embed any Turing machine mechanics and will just check sub-string inclusion.

Despite the fact that we will show the Turing-completeness of our general model for its GA instantiation in particular, note that the Turing-completeness of a general model is implied by the Turing-completeness of any of its instances, so this will indeed prove the Turing-completeness of SLS *in general*. This does not imply that *all* particular SLS methods are Turing-complete. Based on our general model, we will also show how our proof, developed in the context of the GA instance, can be easily adapted to other instances representing popular SLS methods, namely Particle Swarm Optimization (PSO) [28] and Ant Colony Optimization (ACO) [5,6].

The contributions of this paper are the following:

(a) developing the first fully formal model with enough generality to represent any SLS method we are aware of by instantiating its parametric elements, yet keeping the common part of the model as large as possible;
(b) using the formal model to prove the inherent unpredictability (in particular, beyond randomness) of stochastic local search methods in general, which is due to the Turing-completeness of their mechanics at its most basic level; and
(c) providing a formal proof of the Turing-completeness of GA, as well as sketching informal proofs for ACO and PSO.

The rest of the paper is organized as follows. In the next section we describe our general model of SLS heuristics through its operational semantics. Section 3 introduces the particular models for GA, PSO, and ACO to exemplify instances of the general form. The proof of the Turing-completeness of GAs (and thus of SLS) is presented in Section 4, and we informally discuss how to prove the Turing-completeness of other instances of SLS in Section 5. In Section 6 we present our conclusions and lines of future work.

2. General form of stochastic local search

In order to abstract the common structure of SLS strategies, in this section we design a language and a set of operational semantics constituting our *General Form* of SLS. We claim that Genetic Algorithms, Evolution Strategies, Ant Colony Optimization, Iterated Local Search, Stochastic Gradient Descent, Simulated Annealing, Particle Swarm Optimization and every other stochastic local search algorithm we are aware of are instances of this general operational semantics.

The decisions on how to formalize the semantics will be heavily inspired by the operational semantics of **While** by Nielson & Nielson [29, pp. 12–14 and 32–36]. More specific semantics will be provided later for different specific algorithms, but we believe the following semantics are at the lowest abstraction level able to generalize the whole of SLS computation. In the following, we assume that a given algorithm A receives a specific instance p of a problem and explores different solutions. The best solution found by the computation is stored in a variable named *Best*.

[1] By non-trivial we mean fulfilled by neither all programs nor by none. Thus, finding out if programs fulfill properties such as e.g. returning 7 for input 5, returning 7 for some input, computing function $f(x) = 2x$, or halting for all inputs is undecidable.

[2] Note that our goal has nothing to do with showing that the *result* of an SLS method (i.e. the solution it finds to the problem being solved) can be any program written in a Turing-complete language (the ideal output in Genetic Programming [25–27]). Very unrelated to this, we will prove that in general SLS methods *are* Turing-complete by themselves, i.e. for any recursively enumerable function, they can compute it.

[3] Similarly, by abusing the freedom to define *any* fitness function, the fitness function itself could be any computable function.

2.1. State and syntactic categories

The **State** of the computation is a function from variables to values. The relevant variables of the computation constitute the following tuple:[4]

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

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

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

Next we explain the syntactic categories necessary for SLS computation:

Problem must be able to encapsulate the optimization problem that our algorithm must solve. Not only the abstract problem (e.g. nonlinear optimization, TSP[5], etc.) but also the concrete instance and parameters of the problem (e.g. the function to optimize, the graph with edge costs for TSP, etc.). When $Prob \in$ **Problem** is initialized at the beginning of the computation, it will determine how the functions required for the computation of the problem work.

Solution is the category that represents possible solutions of the problem within the computation. For instance, 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 []** 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 guide the search in an SLS algorithm; 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 required to define the auxiliary functions needed for the computation, as well as any other variable needed by the specific algorithm that is being run. For example, the crossover or mutation rate for a GA would belong to this category. **Extra** will be key to the implementation of complex heuristics, as it is the default way to carry lateral effects of computation steps. In hybrid methods, **Extra** will carry all the necessary data to transition between heuristics. The functions would read this information and operate accordingly.

Additionally, let **T** [29, 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 into its corresponding **Problem** instance to be processed), let **Stm** be the set of statements (semantic blocks) that build the SLS computation model, and **Algorithm** be the set of SLS algorithms.

We shall also 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 SLS algorithms, **Algorithm**. Note that we are 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 hereby presented:

$S := S_1; S_2 | \texttt{setProb}(p) | \texttt{generate} | \texttt{nextGen} | \texttt{evaluate} | \texttt{stop} | \texttt{compute}(p)$

We use `compute(p)` to start the execution of our given algorithm 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`, which generates a new set of solutions with the acquired SLS knowledge.

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

2.2. Auxiliary functions

Before we begin to introduce the auxiliary functions that will help us define how the state varies during the computation, let us introduce the following notation: 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 some SLS algorithm, we note that every one of the following functions can also alter the value of **Extra**. Next we present the functions:

- $(\mathbb{C})[\![A]\!]$: **Algorithm** $\longrightarrow$ **Extra**. This is the function that takes an instance A of **Algorithm** and starts the computation by initializing every parameter and variable needed by the heuristic.
- $\mathcal{P}[\![p]\!]_s$: **Pexp** $\times$ **State** $\longrightarrow$ **Problem** $*$. This is the semantic function for **Pexp** that translates 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**[] $*$. It uses the values of *Prob, Sols*, and *FPW* given by the state s to stochastically compute the new value for *Sols*, using the information obtained by the previous iterations of the algorithm, and in such a way that the solutions are concordant to the problem.
- $\mathcal{A}eval[\![]\!]_s$: **State** $\longrightarrow$ **F/P/W** $*$. It 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.
- $\mathcal{B}eval[\![]\!]_s$: **State** $\longrightarrow$ **Solution** $*$. It 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.
- $\mathcal{SC}[\![]\!]_s$: **State** $\longrightarrow$ **T**. This is the stop criterium 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 SLS computation model, different SLS algorithms will implement these functions differently. Just to give an example, let us consider two GAs: the first one returns the value of the best individual of the last generation, and the second one 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.

[4] Other auxiliary variables such as array indexes and loop counters are not reflected here, although they would be necessary for the implementation of SLS algorithms.

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

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

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

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

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

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

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

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

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

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

Fig. 1. Operational semantics for the General Form.

Readers should also be aware that many of these functions, specially $\mathcal{A}eval[\![]\!]_s$, $\mathcal{B}eval[\![]\!]_s$ and $\mathcal{SC}[\![]\!]_s$, could be overloaded with *daemon actions* such as local search phases, automatic parameter modifications, and other mechanisms to improve solutions. For the sake of generality and notation simplicity, no terms or functions are given to *specifically* denote daemon actions in our design, as they can be easily integrated within the other elements of the language.

2.3. Rules

Finally, we introduce the rules of our SLS computation model in its General Form. The execution begins at an initial state s with the [compute] rule by inputting the problem p as a parameter: $\langle \texttt{compute}(p), s\rangle$.

The values of the initial state s are not relevant, as they will be changed by the input 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** be any variable of **State** other than **Extra**. We define **Var*** as **Variable** $\times$ **Extra**.

As usual in operational semantics, the meaning of the statements is specified by a transition system with two kinds of configurations:

- $\langle S, s\rangle$ represents 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 rule $\frac{\alpha \Rightarrow \beta}{\gamma \Rightarrow \delta}$ indicates that we can only transition from configuration γ to configuration δ if a valid transition from α to β can be made.

The rules for the General Form are defined in Fig. 1 and have the following purposes: [comp] rules allow to concatenate instructions carrying on the state modifications; [set problem] loads the instance to run onto the state of the computation; [generate], [next generation], and [evaluate] run their corresponding statements to generate the initial solutions, the following generation, and evaluating the fitness of their generation, respectively; and the [stop] rules either finish the computation and return the final state or trigger the next iteration of the process, depending on the evaluation of the stop function at the current state. Finally [compute] is the first rule we apply to start the computation, and all the others are applied as we unravel it.

3. GAs, PSO & ACO As instances of the general form

Instances of the General Form for particular algorithms (GA, ACO, PSO) are introduced in this section.

3.1. Genetic algorithms

Now that the general model has been presented, more detailed semantics, based on it, will be provided for genetic algorithms. The goal is now to showcase the structure of the computation of genetic algorithms while keeping the semantics general enough to be applied to particular instances of the genetic algorithms (i.e. different variants of GA). The representation of the chromosomes, selection method, mutation rate, stop criteria, and other particularities of the genetic algorithm instance should not be relevant to the semantics. This semantics is general enough to cover degenerated instances of the genetic algorithm such as e.g. evolutionary strategies, where the selection and crossover 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 a generation are no longer relevant after the creation of the next generation.

The **State** is defined as in the General Form. However, some syntactic categories can be further specified. In this case, **Solution** will represent the codification of a single chromosome and **Solution []** 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 together with every other solution needed to compute crossovers and mutations. Also, *FPW* $\in$ **F/P/W** now consists of the fitness of every individual (which must be recomputed for every generation).

We will use the same meta-variables of the General Form, the only difference being the structure for **Stm** constructs:

$S := S_1; S_2 | \texttt{setProb}(p) | \texttt{generate} | \texttt{select} | \texttt{cross} | \texttt{mutate} | \texttt{nextGen} | \texttt{evaluate} | \texttt{stop} | \texttt{compute}(p)$

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

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

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

$$[\text{next generation}] \quad \langle \texttt{nextGen}, s\rangle \Rightarrow \langle \texttt{select}; \texttt{cross}; \texttt{mutate}, s'\rangle$$

Fig. 2. Instantiation of the operational semantics for Genetic Algorithms.

Three new statements (`select`, `cross`, and `mutate`) are considered to structure the process that was previously abstracted by `nextGen` in a single step. This showcases how the evolution is performed in GAs: fitness-biased selection of individuals to cross over, generation of new individuals, and mutation of the population. The meaning of the statements is further specified by the following functions and rules.

- $\mathcal{S}[\![]\!]_s$: **State** $\longrightarrow$ **Solution**[] $*$. This uses the values of *Prob, Sols*, and *FPW* given by the state s to compute the new value of *Sols* consisting of the individuals of the previous generation selected for the crossover stage.
- $\mathcal{C}[\![]\!]_s$: **State** $\longrightarrow$ **Solution**[] $*$. It uses the values of *Prob, Sols*, and *FPW* given by the state s to compute the new value of *Sols*, which is determined by a combination of the result of the crossover operation applied to the selected individuals and some of the original individuals.
- $\mathcal{M}[\![]\!]_s$: **State** $\longrightarrow$ **Solution**[] $*$. It uses the values of *Prob, Sols*, and *FPW* given by the state s to compute the new value of *Sols* consisting of the individuals already present in *Sols* after modifying (mutating) some of them.

Since this is an abstraction of the genetic algorithm computation model, different genetic algorithms will implement these functions differently. For instance, 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. On the contrary, 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 $\mathcal{S}[\![]\!]_s$. The definitions of the other functions used in the following rules remain unmodified with respect to the General Form.

To complete the semantics, we take the rules from the General Form and modify them by redefining one rule ([next generation]) and adding three new ones: [selection], [crossover], and [mutation] are introduced to represent the corresponding namesake stages where a GA performs its population transformations in each generation. The changes over the operational semantics of the General Form are presented in Fig. 2.

3.2. Ant colony optimization

In this section we particularize the operational semantics from the General Form to represent Ant Colony Optimization algorithms. ACO algorithms generate a completely new population on each generation. However, the previous iterations influence how the new generation is created. We may, therefore, establish that the memory of ACO resides on the pheromones rather than on the individuals.

For this model, the **State** is also defined as it was in the General Form, although some of the syntactic categories are further specified as follows.

Problem is basically as described in the General Form. The only variation is that now we are dealing with some form of 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.

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

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

Fig. 3. Instantiation of the operational semantics for Ant Colony Optimization.

Solution is the category that represents possible solutions of the considered route finding problem, so it is a data type to store ordered subgraphs. These subgraphs are formed from the paths taken by ants. Note that sometimes the best solution may be the composition of paths traversed by several different ants. **Solution []** represents a set of instances of **Solution**. Thus, *Best* $\in$ **Solution** will be the most efficient subgraph found by the computation and *Sols* $\in$ **Solution []** the set of subgraphs generated and simulated by ants in the current iteration of the algorithm.

The core of **F/P/W** 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 any performance-based parameters needed for the computation of auxiliary functions. Of course, the pheromone values do not need to be scalar, and vectors are a natural way to implement multi-pheromone ACO heuristics [30].

We will use the same meta-variables of the General Form, the only difference being the structure for **Stm** constructs, now given by:

$S := S_1; S_2 | \texttt{setProb}(p) | \texttt{generate} | \texttt{nextGen} | \texttt{simulate} | \texttt{evaluate} | \texttt{stop} | \texttt{compute}(p)$

In this case, the only new statement is `simulate`. This decision is made to highlight the relevance of the fitness evaluation for these methods. Although 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. To make use of this statement, we introduce the [simulate] rule and modify [evaluate] to include the simulation step.

The list of functions remains unchanged from the General Form, but their specifications are worth noting:

- $\mathcal{N}ext[\![]\!]_s$: **State** $\longrightarrow$ **Solution**[] $*$. This uses the values of *Prob* and *FPW* given by the state s to compute the new value of *Sols* determined by the stored problem and the information obtained by the previous iterations of the algorithm (except for the first iteration). Note that *Sols* is no longer relevant for this function.
- $\mathcal{A}eval[\![]\!]_s$: **State** $\longrightarrow$ **F/P/W** $*$. It is as seen in the General Form, although in this particular case this function carries out two different tasks: computing the fitness of every ant in *Sols* and storing the pheromones dropped by it.

Again, different ant colony algorithms may implement these functions differently; e.g.: $\mathcal{B}eval[\![]\!]_s$ could consider solutions not present in *Sols*, such as the densest pheromone path, as potential best solutions. Finally, the rules for Ant Colony Optimization are the same as the ones of the General Form with the modifications mentioned to include `simulate`, as shown in Fig. 3.

3.3. Particle swarm optimization

The last particularization of the operational semantics for SLS computation we present in this paper is the operational semantics for Particle Swarm Optimization algorithms.

In PSO individuals of the solution population move around the search space trying to find the optimal solution. This process is similar 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. However, the way the new positions of the particles are computed is structurally different from that of GAs. Again, the memory of

PSO lies on the individuals and not so much on the weights that have to be recalculated on each generation. Thus, this is 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 our General Form of SLS computation.

The **State** is defined as it was in the General Form, although some syntactic categories are further specified as follows.

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

The **F/P/W** category represents every variable necessary to direct the search in the SLS scheme. *FPW* ∈ **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 being the structure for **Stm** constructs, now given by:

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

Stm is changed in a similar fashion as it was for GA: `divert`, `aim`, and `move` are introduced to detail the way in which `nextGen` works. 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 order of `divert` and `aim` statements is actually interchangeable for the computation, but for the functions and rules that follow, we will assume that `divert` precedes `aim` in every iteration. To make use of these statements, we will add their namesake rules. Similarly as for GA, additional functions are required:

- $\mathcal{D}[\![]\!]_s$: **State** ⟶ **Solution**[] ∗. This function 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** ⟶ **Solution**[] ∗. It 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** ⟶ **Solution**[] ∗. It 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** ⟶ **F/P/W** ∗. This is as seen in the General Form although, in this particular case, the function computes 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.

Once more, different particle swarm algorithms may implement these functions differently. For example, different neighbourhood topologies [31] would yield different $\mathcal{D}[\![]\!]_s$ and $\mathcal{A}[\![]\!]_s$ functions. Similarly as in the previous instances, the list of rules for PSO adds new ones to showcase the stages of the computation in each iteration, as well as a modification in [next generation] to introduce them, as it is depicted in Fig. 4.

The previous three examples (GA, ACO, PSO) are illustrative of the power of the General Form semantics to model the semantic behavior of different algorithms of stochastic local search. We will use them to prove a significant semantic property: the Turing-completeness of SLS strategies in general. In the next section, we will prove this property through one of the instances of our SLS model.

Note that, given that Turing-completeness, *any* algorithm of any kind (not just SLS) can be emulated by our SLS model —if properly codified as input of the SLS instance proved to be Turing-complete. We trivially infer that, in particular, any *SLS algorithm* can be codified into and emulated by our SLS model. Yet the purpose of our SLS model is not just to enable the representation of any SLS method, but also doing so through a natural use of the model parameters. The model instances defined in the previous sections for GA, ACO, and PSO illustrate this capability, and similarly natural instances can be created for variants of these methods and others.

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

Fig. 4. Instantiation of the operational semantics for Particle Swarm Optimization.

We already mentioned how the framework could deal with PSO algorithms with different neighbourhood relationships, as well as with multi-pheronome ACO algorithms. Clearly, algorithms originally designed as a modification of others are expected to be defined in a similar way (e.g. River Formation Dynamics (RFD) [32] could be defined in a very similar way as ACO).

Many popular algorithm variations are easy to define as well. For instance, for multi-population algorithms [33], a population identifier can be added as a parameter for individuals and be handled by the different functions. For multi-modal or multi-objective optimization [34–36], *Best* may easily become a vector, and $\mathcal{B}eval[\![]\!]_s$ can manage the identification of different objectives. In co-evolutionary algorithms [37], we could borrow the previous techniques to handle multi-population, and *Extra* could include the structures needed to manage the global fitness of the model. Similar techniques can also cover the instantiation of decomposition-based strategies [38]. Some trajectory-based heuristics such as Simulated Annealing [39] or Iterated Local Search require a single individual population. Our General Form also accommodates techniques to handle constrained optimization problems [40], in particular by including penalization or blacklisting parameters into $\mathcal{A}eval[\![]\!]_s$ (independently of the heuristic being instanced).

Of course, these techniques for the mentioned heuristics are not the only ways to instantiate our framework —they are just an illustration of its versatility. As a final example, consider the SEMO, FEMO, and GEMO multi-objective algorithms presented in [36]: All of them keep a population of Pareto optimal individuals, therefore making *Best* = *Sols* at the end of each iteration. Here we see how having *Best* be a vector easily accommodates for multi-objective optimization. We can also divide the "next generation" stage of these methods in two steps: selection and mutation. Since the three algorithms only differ in which individuals are mutated, their instantiation would be the same except for a slightly different $\mathcal{M}[\![]\!]_s$ function. This way our model captures their common general structure while adapting for their particularities.

4. Turing-completeness of stochastic local search heuristics

In this section we prove the Turing-completeness of SLS computation, which means that any program that can run on a Turing Machine (TM) can also be run by launching an SLS strategy. We achieve this by proving the Turing-completeness of an instance of these strategies: genetic algorithms. As far as we know, this property has not been proved before and has serious implications for the investigation of semantic properties. Note that the convergence of some SLS methods has been studied in the literature under specific conditions and configuration settings (see e.g. [41–46]). Due to the Turing-completeness of SLS methods and Rice's theorem, we conclude that very strong and exceptional hypotheses will be required, in general, to achieve this kind of semantic

knowledge —in the same way as proving that a program produces its expected outputs is undecidable in general, but it can be proved for some subsets of programs if we heavily constrain their form in some way (e.g. programs with restricted loop structures or no loops at all).

Introducing strong constraints can let us reason about desirable semantic properties in some SLS settings; e.g. polynomial-time approximation to maximizing submodular functions [47,48] or convergence of SA to the global optimum under strong ergodicity [42]. However, restrictions on the expressivity of the method (when viewed as a computation model) will always come with them. Turing-completeness also disables *in general* the use of Statistics to predict the behavior of SLS methods. Even if we defined a GA that simulated a universal Turing machine, run it for many instances and calculated its average outputs or execution times; this would not tell us anything about what to expect, in general, if the same GA were run for new untested instances. It would be equivalent to trying to predict the behavior of a C++ program for all inputs just by observing what it does for some and calculating the average of its outputs or execution times. Statistics are, however, very useful indeed for the analysis of SLS methods when additional constraints do limit their scope.

Note that proving the Turing-completeness of GAs has nothing to do with the Turing-completeness of the *solutions* constructed by some GAs (such as some GAs dealing with Genetic Programming [25–27]). It consists in showing the existence of a GA which can *simulate* any program (or technically, Turing machine) for any input it could receive. That is, when the initial configuration of that GA codifies that Turing machine and its input in some form, the final solution of the GA provides the output of that Turing machine for that input.

Achieving Turing-completeness requires representing all the internal memory of a program no matter how much it grows. Therefore chromosomes will be required to grow arbitrarily in the GA constructed for our proof, like they do in GA domains such as e.g. Genetic Programming when no size bounds are set. Given the way GAs behave exactly like programs in our proof, constraining the sizes of chromosomes would not make GAs easy to predict though, in the same way as predicting the output of programs using only a polynomial amount of memory, or executing only for an exponential number of steps, is decidable but PSPACE-hard (and thus, intractable if P≠PSPACE) or EXPTIME-hard (and hence intractable in any case) respectively.

Note that if a computation model includes some component allowed to be *any program* then the model will very probably be Turing-complete. We could argue that GAs are general enough to let some components, such as its fitness function, be any program. If achieving the Turing-completeness required exploiting that total freedom to define some component in *any* way, then one could wonder if the Turing-completeness exists *only* for GAs with that high level of sophistication. Were that the case, many categories of simpler but widely used GAs could be non-Turing-complete and easily predictable, giving the Turing-completeness of GAs *in general* a marginal impact in practice. Quite on the contrary, our proof will show that GAs are Turing-complete even when all their components are defined in remarkably simple ways. In particular, the fitness function will just check for substring inclusion. This simplicity will show that, if we wished to define a particular category of non-Turing-complete and efficiently-predictable GAs by constraining the way GA components are defined, then constraints would be so strong as to nearly nullify the resulting category.

4.1. Proving the Turing-completeness of genetic algorithms

In this section we prove the Turing-completeness of GA by constructing a GA capable of simulating any given Turing Machine for any given input, i.e. a GA being universal for Turing machines. Technically, we will present a GA that solves the problem of finding the complete computation *history* (i.e. sequence of all configurations iteratively reached during the computation) of any given Turing machine for any given input. Before we begin with the description of this GA, we introduce some definitions and results that will help us build it.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a Turing Machine (TM) as defined in [49, p. 327], that is: Q is the set of states; Σ is the input alphabet; Γ is the tape alphabet; δ is the transition function; $q_0 \in Q$ is the initial state; $B \in \Gamma$ is the blank symbol; and $F \subseteq Q$ is the set of final states (accepting states). The Post Correspondence Problem (PCP) consists in, given a finite set W of pairs (a, b) of finite strings, finding out if, for some finite sequence of pairs (where there may be several occurrences of the same pair), the string obtained by consecutively reading the first components of the pairs and the string read on the second ones coincide. The Modified Post Correspondence Problem (MPCP) adds the constraint that the first pair of the sequence is fixed. In order to show the undecidability of MPCP, in [49, pp. 401–412] any pair (M, w), where M is any Turing Machine and $w \in \Sigma^*$ is its input, is mapped into a set of MPCP pairs T such that M halts for w iff the answer of MPCP for T is yes. We will use this deterministically generated finite set T of pairs in our construction next.

This set of pairs 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 [49]. Given the way T is constructed, we will have a pair to represent the initial state and input of (M, w), a closing pair to represent the end of the computation, pairs to represent the accepting states q_f of M and that will simulate the consumption of every symbol in the tape, a (x, x) pair for each $x \in \Gamma$, a $(\#, \#)$ pair for the separator symbol # (these symbols separate the representation of each full TM configuration from the one reached in the next computation step), and a pair to represent every transition in δ. Furthermore, if the first pair of a MPCP partial solution is the one that represents the initial state of the machine, then there is one and only one pair $t \in T$ able to extend that partial solution. This allows the extension of the partial solution to effectively emulate the computation of (M, w).

In order to prove the undecidability of PCP, in [49] the halting problem for Turing machines is reduced to MPCP, and MPCP is reduced to PCP. Thus, if PCP were decidable, so would be the halting problem (which is undecidable). As mentioned before, this proof uses the pairs 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 of M for input w.

With these ingredients, we can design a GA that tries to solve MPCP for input T by sequentially finding the next pair (the next step of the computation) until MPCP is solved. We will prove that, for every computation step of the represented TM, our GA converges 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 is that this GA will mimic the way genetic programming builds structured individuals (in this case just linearly structured, i.e. sequences with different sizes) in order to design an individual that arranges the pairs t_i from T into incremental partial solutions of MPCP until a complete solution is found. We will employ the operational semantics for GAs to describe this algorithm. Its population will consist of just one individual. This individual is represented as a sequence of elements we will call *tiles*, and we should interpret $tile_i$ as the computational representation of $t_i \in T$ (i.e. the representation of t_i in the GA domain).

The fitness function always returns the number of correctly arranged tiles and stores the best partial solution found so far (returning this solution is necessary only 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 tile sequence of the individual, i.e. the end of the program execution history the GA is building: if the last tile is incorrect, then it is substituted by another random one; if it is correct, then a new random one is added afterwards, expanding the sequence.

With these elements we can guarantee, for every execution step, that the GA will always have convergence in probability to add the next correct execution step. However, by 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. In order to accelerate the convergence of the implementation, the special pair of T denoting the beginning of the sequence may have its corresponding tile removed from those that can be randomly added to the individual. Finally, the GA will only stop when the tile for the closing pair has been placed correctly.

Note that this is an extremely simple GA, in particular with a single individual population that only varies via mutation. This makes it akin to simple trajectory-based methods like Simulated Annealing (SA) or Iterated Local Search (ILS).

Now we define the data structures (not to be confused with the actual GA variables being instances of them; boldface and italics are used, respectively, to distinguish them) and the functions used in our operational semantics:

Data Structures

- **Tile** is a pair of strings. It is used to represent pairs $t_i \in T$.
- **Problem** is an array of elements of type **Tile** and an integer, which represent the set T of pairs in the MPCP codification of the input (M, w) and $|T|$ respectively.
- **Solution** consists of a linked list of **Tile** elements, which represents an ordered arrangement of the pairs that code the instance of MPCP.
- **F/P/W** is just one integer. It counts the number of correct tiles chained so far in the only element of *Sols*.
- **Extra** is an array of Boolean values. It 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 [49, pp. 401–412], then codes every pair 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*[6], and the second one is chosen randomly among all other positions of *Prob* with index lower than the integer stored. It also returns an instance of **Extra** equal to *Extra* with the corresponding position set to 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*.
- $\mathcal{M}[\![]\!]_s$ is defined as follows. If *FPW* equals the number of tiles in *Best*, then it appends a new random tile from *Prob* (with index $i \in [1, N - 1]$, where N is the stored integer, 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 for 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 a modified version of *Best* with $Prob[i]$ at the end, and an instance of **Extra** equal to *Extra* except for the value with index i, which is set to false.
- $\mathcal{A}eval[\![]\!]_s$ concatenates all the strings from the tiles of *Sols* into two strings a and b, where a is the concatenation of the first elements of each tile 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 symbols, then the function returns 1 and *Extra*.
  If *FPW* has not been initialized and b can be expressed as a followed by other symbols, then 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 returns $FPW + 1$ 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 $Prob[1]$ (the closing tile as defined in $\mathcal{P}[\![p]\!]_s$).

Proof of Turing-completeness

Theorem 1. *There is a genetic algorithm that, given any Turing Machine M and any input w, is able to simulate every step of the computation of M for w. Thus, genetic algorithms are Turing-complete.*

Proof. To prove this result, we will consider the GA described above and use induction over the number of computed steps. Let a be the concatenation of the first elements of each tile in *Best* respecting their order, b the concatenation of the second elements and let a_i, b_i be 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 a computation of (M, w) takes 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, w). This is granted after the first evaluation, since the first tile of the chromosome is determined to be the first pair 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 pair set T seen in [49, pp. 401–412] 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 described GA always finds the next tile to augment its current partial solution stored in *Best*, then after adding $|b_k| + 1$ tiles (where $|b_k|$ is the number of symbols present in string b_k) 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 proved is that the added tiles have correctly computed b_k, and that the described GA always finds the next tile to augment *Best*. The first property is given in [49, pp. 401–412], whereas the second one is proved in the following lemma.

Lemma 1. *The GA described above always finds the next tile to augment its current partial solution stored in Best.*

Proof. The construction of T implies that, for every partial solution c of the MPCP instance, there exists one and only one $t_i \in T$ such that c followed by t_i is also a partial solution [49, pp. 401–412]. By 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. □

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

Analogously, it can be proved that our GA halts iff (M, w) does, which makes the Halting Problem undecidable for SLS computation in

[6] The representation of the problem to solve, in this case the list of all MPCP tiles.

general; but this is much simpler to prove by directly applying the properties of T as follows. 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, then $\mathcal{SC}[\![]\!]_s$ will return **tt** and our GA will halt. If it is not that kind of tile, then the GA will keep on adding tiles that eliminate the symbols of the tape like (Xq_f, q_f), (Xq_fY, q_f), or (q_fY, q_f) until the only viable option is a $(q_f\#\#, \#)$ tile. A more detailed explanation of this elimination process can be seen in [49, pp. 401–412].

It is worth noting that the proposed universal GA is particularly simple. Essentially, it lies in just two key ideas: (a) when the mutation operator changes a chromosome (the only one in the population), it just modifies its ending part, which is replaced or expanded by some sequence randomly taken from a given finite set of possible sequences (the tiles); and (b) the fitness function, as well as the condition to decide between replacing or expanding in (a), just checks if some sequence is a prefix of another one and for how long. The mechanics of this GA are particularly simple, and by no means they embed the internals of the behavior of a TM (tape, pointer moves, states, etc.). The TM to be simulated by the GA is actually received in the form of the finite set mentioned in (a). Note that using any Universal Turing Machine (UTM)[7] requires receiving the TM to be simulated by it codified into some arbitrary notation, and similarly, simulating any TM with our GA requires receiving the TM codified into an appropriate notation, in particular that finite set mentioned in (a). Although this set can be manually defined for each simulation case, there is actually an automatic way to obtain it from a TM defined with its most typical notation: converting it into a set of MPCP tiles as described in [49].

Note that we could trivially make a deterministic universal GA from our GA by removing its randomness. We could do it just by making the GA try the different tiles to be added to the chromosome in any arbitrary deterministic order, instead of randomly (the blacklist could be trivially used to know which tile must be tried next).

Finally, we show that a more memory-efficient (but less simple) implementation can be created by eliminating from *Sols* those tiles whose information is no longer relevant to the evolution of the solution. If some symbols on the first parts of the tiles have already been completely matched in the concatenation of the second parts, then they no longer provide information to determine the next tile to be appended to *Sols*. Moreover, if a symbol in the concatenation of the first parts has at least two separator symbols # to its right in the concatenation (not necessarily together or right next to it), then eliminating it still preserves the entirety of the TM configuration in the last computation step (recall that each full TM configuration is surrounded by a # symbol at each side). Therefore, we can safely eliminate from *Sols* all the tiles whose symbols in the first parts have already been completely matched and have at least two # symbols to their right in the concatenation of first parts.

This implementation would require careful modifications in *Extra* and the functions previously defined, in particular to keep track of how many symbols from the concatenation of the second parts of *Sols* need to be skipped by $\mathcal{A}eval[\![]\!]_s$ (because they already matched symbols that have been removed from the concatenation of the first parts), as well as to take into account that we no longer keep the full computation history, but just the TM configuration at the last step and part of the configuration at the current one. Note that, by removing tiles this way, *Sols* will never contain symbols from the representation of more than two TM configurations. The stop condition would now require that the last tile is $Prob[1]$ and that the concatenations of the first and second parts match if we skip the already-matched symbols of the concatenation of the second parts. The remaining functions would be trivially adapted.

By trimming unnecessary tiles this way, the representation size of *Sols* (and thus, the representation size of our only individual in the GA) does not grow with the number of simulated TM steps as in our original construction, but it is just proportional to the maximum amount of memory used by the simulated TM during its execution. This independence between the size of the chromosome and the number of simulated TM steps is not required at all to achieve the Turing-completeness of GAs. However, it is needed for other Complexity-related properties, such as e.g. the property that GAs where chromosomes have polynomial size can simulate any TM running in polynomial space. Note that this would not be possible if the full history was stored in the chromosome, as polynomial-space TMs can take exponential time (i.e. have *exponential-size* histories). On the contrary, tile-trimming GAs with polynomial-size chromosomes can simulate all TMs running in polynomial space, so even very basic semantic properties of GAs with polynomial-size chromosomes are PSPACE-hard to decide.

5. Proving Turing-completeness for other instances of the general form

The Turing-completeness proof presented in the previous section can be adapted to work with other instances of SLS computation. Of course, not all instances of SLS algorithms are Turing-complete; however, we claim that most other popular methods different from GA are too. Next we informally illustrate how we could prove it for the other two methods instantiated in this paper.

Let us consider Ant Colony Optimization. From a TM and input (M, w) we can build the set T of MPCP tiles as before. We can construct an ACO instantiation such that, if it is applied to that MPCP instance, then it necessarily tends to solve it, thus developing on the fly the execution of M for w. Since Turing-completeness requires unlimited memory, an infinite graph to be traversed by the ants is necessary in ACO, or more precisely, a finite graph (in particular, a tree) that can be iteratively extended during the ACO execution up to any needed finite size on demand, without limit. Each edge represents taking some tile of the MPCP instance, and each path from the root to any leaf represents a partial solution (sequence of tiles). In the beginning of the execution of ACO, this tree just consists of a single node. When an ant reaches a leaf, the ant finishes its journey and the path is reinforced via pheromones only if the tile sequence of the path consists only of legal moves (i.e. if one of the two strings read through the sequence of tiles is a prefix of the other, as we did for GA). In that case, right after removing the ant new edges are created from the leaf (thus no longer a leaf), one for each available tile, which will eventually let subsequent ants go one step further.

This way, ACO encourages ants to follow legal paths according to the tiles and extend them. Pheromone trails are reinforced only on the correctly computed tiles (edges). We assume that each newly created edge starts the execution with a 0 pheromone trail, pheromones do not disappear after any number of iterations, and probabilities to select each edge are directly proportional to the amount of pheromones in all available edges. We can see that, in this setting, there is convergence in probability to eventually reach any step of the TM computation. By using the same stop criterion used for the Turing-complete GA, we again achieve Turing-completeness.

Proving the Turing-completeness of Particle Swarm Optimization is more closely related to the proof for GA than the one for ACO. In addition to the standard adaptations of PSO to deal with a discrete domain, we need the solution vectors to be arbitrarily enlarged on demand to represent arbitrarily long representations of the full TM memory state, as we did for GA. Moreover, similarly as the mutation operator of GA was restricted to affect only the last part of the solution (i.e. the part it grew from), the two forces governing the movement of particles in PSO are defined to affect only specific scopes as follows: The force that makes particles move closer to the best particle will make all particles necessarily copy all the components (tiles) of the best solution; and the force that makes particles randomly move will make particles create random additions in a new component (tile) of its vector, used to let the sequences of MPCP tiles enlarge. This way, the particle that copies the

[7] A Turing Machine that, given any TM and an input, simulates the computation of the latter TM for that input, see e.g. [50, p. 20].

former best solution (i.e. becomes it) and next randomly adds the suitable addition to its vector (which must be the legal tile) becomes the new best particle, the one to be followed next. Again, this process converges in probability to eventually creating the sequence of tiles representing the corresponding execution of the TM.

Similarly to the GA case, blacklists can be introduced in the previous ACO and PSO constructions to turn the convergence in probability into regular convergence. This way, the maximum number of steps up to termination would be bounded by a function over the number of steps actually needed by the TM to run for its input. Also, unnecessary elements of the constructed trees and solution vectors of ACO and PSO, respectively, could be trimmed as we did in the alternative GA construction described in the last paragraphs of the previous section.

6. Conclusions and future work

We have presented a fully formal language which, by means of its common structure and its parametric constructions, lets us define any stochastic local search method to the best of our knowledge. In order to illustrate its flexibility, its instances for three algorithm families with very different mechanics (GA, PSO, ACO) have been presented. By using our common and standardized language in the definition of new and existing algorithms, the functional relationships between algorithmic choices and problem features could be more easily investigated. Results from different researchers, working with different algorithms, could be seamlessly classified and added into a common knowledge pool.

We have also formally proved Turing-completeness of the GA instance, which implies Turing-completeness of the general model it instantiates, and thus Turing-completeness of stochastic local search methods in general. Besides, we have sketched how this Turing-completeness proof can be easily adapted to the other two instances presented, which supports the idea that many other important families will be Turing-complete too.

By Rice's theorem, Turing-completeness of a method implies undecidability of all its non-trivial semantic properties (i.e. the ones concerning the relations it computes between its inputs and its outputs, and being fulfilled neither always nor never). This undecidability means that, for a given property, no algorithm exists that decides whether the property is satisfied or not. This result constitutes an important limitation to any attempt to predict the behavior of these methods. For example, it implies the undecidability, in general, of whether GA, ACO, or PSO instances, allegedly designed to solve some optimization problem, will eventually produce solutions of some desired kind (e.g. optimal solutions, solutions always reaching some performance ratio, solutions reaching some performance ratio on average, solutions fulfilling some given structural condition, etc.). These semantic properties may still be proved for particular cases (e.g. a given SLS method solving a particular problem or problem instance), as well as for subsets of cases disabling Turing-completeness by constraining the general freedom of the method or the problem. However, as we pointed out in the paper, removing Turing-completeness (e.g. by not letting GA chromosomes grow unboundedly) is not enough, by itself, to enable *efficient* property detection in general, as it may only turn the undecidability into intractability (e.g. PSPACE-hardness).

Regarding our lines of future work, we wish to use our model to develop a fully formal taxonomy of stochastic local search methods. Different levels of that taxonomy would be reached by instantiating the General Form up to a higher or lower level of abstraction. Additionally, we wish to investigate the complexity of semantic property identification in general when chromosomes cannot grow *at all*. We suspect that we would still have PSPACE-hardness in this case, and note that this intractability would be consistent with known convergence results in the SLS field. For instance, SA is guaranteed to reach the optimum solution if we can proceed at a slow enough pace, in particular taking exponential time in some bad cases [51]. This makes the heuristic search take, in practical terms, as much time as an exhaustive search over the corresponding (exponential-size) search space.

Finally, we wish to systematically investigate what restrictions in the definition of the *elements* of a heuristic, beyond the representation size of algorithm entities, make the difference between enabling Turing-completeness for that method and not enabling it, with special attention to non population-based heuristics due to their simplicity.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] H. H. Hoos, T. Stützle, Stochastic Local Search Algorithms: An Overview, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1085–1105. 10.1007/978-3-662-43505-2_54
[2] Handbook of evolutionary computation, T. Back, D.B. Fogel, Z. Michalewicz (Eds.), IOP Publishing, Bristol, UK, 1997.
[3] J. Kennedy, R.C. Eberhart, Y. Shi, Swarm intelligence, Morgan Kaufmann, 2001.
[4] D.E. Goldberg, Genetic algorithms, Pearson Education, 2006.
[5] M. Dorigo, G. Di Caro, Ant colony optimization: a new meta-heuristic, in: Proceedings of the 1999 Congress on Evolutionary Computation, CEC'99, 2, IEEE, 1999, pp. 1470–1477.
[6] M. Dorigo, M. Birattari, T. Stutzle, Ant colony optimization, IEEE Comput Intell Mag 1 (4) (2006) 28–39.
[7] F. Ducatelle, G. Di Caro, L.M. Gambardella, Using ant agents to combine reactive and proactive strategies for routing in mobile ad-hoc networks, Int J Comput Intell Appl 5 (2) (2005) 169–184.
[8] D. Karaboga, B. Akay, A modified artificial bee colony (ABC) algorithm for constrained optimization problems, Appl Soft Comput 11 (3) (2011) 3021–3031.
[9] C. Qiu, C. Wang, X. Zuo, A novel multi-objective particle swarm optimization with k-means based global best selection strategy, International Journal of Computational Intelligence Systems 6 (5) (2013) 822–835.
[10] P. Pinto, A. Nägele, M. Dejori, T. Runkler, J. Sousa, Using a local discovery ant algorithm for bayesian network structure learning, Trans. Evol. Comp 13 (4) (2009) 767–779.
[11] I. Rodríguez, P. Rabanal, F. Rubio, How to make a best-seller: optimal product design problems, Appl Soft Comput 55 (2017) 178–196.
[12] K. Sörensen, Metaheuristics—the metaphor exposed, International Transactions in Operational Research 22 (1) (2015) 3–18.
[13] F. Hutter, H.H. Hoos, K. Leyton-Brown, T. Stützle, Paramils: an automatic algorithm configuration framework, Journal of Artificial Intelligence Research 36 (2009) 267–306.
[14] M. López-Ibánez, J. Dubois-Lacoste, L.P. Cáceres, M. Birattari, T. Stützle, The irace package: iterated racing for automatic algorithm configuration, Oper. Res. Perspect. 3 (2016) 43–58.
[15] F. Hutter, H.H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: International conference on learning and intelligent optimization, Springer, 2011, pp. 507–523.
[16] P. Rabanal, I. Rodríguez, F. Rubio, Assessing metaheuristics by means of random benchmarks, in: International Conference on Computational Science 2016 (ICCS 2016), Procedia Computer Science, vol.80, 2016, pp. 289–300.
[17] K.A. De Jong, Evolutionary computation: A Unified approach, MIT press, 2006.
[18] D. Molina, J. Poyatos, J.D. Ser, S. García, A. Hussain, F. Herrera, Comprehensive taxonomies of nature- and bio-inspired optimization: Inspiration versus algorithmic behavior, critical analysis and recommendations, 2020.
[19] Q. Kang, J. An, L. Wang, Q. Wu, Unification and diversity of computation models for generalized swarm intelligence, Int. J. Artif. Intell. Tools 21 (03) (2012) 1240012.
[20] Y.-J. Gong, W.-N. Chen, Z.-H. Zhan, J. Zhang, Y. Li, Q. Zhang, J.-J. Li, Distributed evolutionary algorithms and their models: a survey of the state-of-the-art, Appl Soft Comput 34 (2015) 286–300.
[21] J.L. Olmo, J.R. Romero, S. Ventura, Swarm-based metaheuristics in automatic programming: a survey, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 4 (6) (2014) 445–469.
[22] H.H. Hoos, T. Stützle, Stochastic local search: Foundations and applications, Elsevier, 2004.
[23] H.G. Rice, Classes of recursively enumerable sets and their decision problems, Trans Am Math Soc 74 (1953) 358–366.
[24] N. Cutland, Computability: An introduction to recursive function theory, Cambridge University Press, 1980.
[25] J.R. Koza, Hierarchical genetic algorithms operating on populations of computer programs, in: N.S. Sridharan (Ed.), Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89, 1, Morgan Kaufmann, Detroit, MI, USA, 1989, pp. 768–774.
[26] A. Teller, Turing completeness in the language of genetic programming with indexed memory, in: Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence, IEEE Press, 1994, pp. 136–141.

[27] A. Naidoo, N. Pillay, Using genetic programming for Turing machine induction, in: EuroGP'08 Proceedings of the 11th European conference on Genetic programming, Springer-Verlag, 2008, pp. 350–361.
[28] J. Kennedy, Particle Swarm Optimization, in: Encyclopedia of machine learning, Springer, 2011, pp. 760–766.
[29] H.R. Nielson, F. Nielson, Semantics With Applications: A Formal Introduction, Wiley Professional Computing.
[30] M. Sekara, M. Kowalski, A. Byrski, B. Indurkhya, M. Kisiel-Dorohinicki, D. Samson, T. Lenaerts, Multi-pheromone ant colony optimization for socio-cognitive simulation purposes, in: Proceedings of the International Conference on Computational Science, ICCS 2015, 2015, pp. 954–963.
[31] Q. Liu, W.W.H. Yuan, Z. Zhan, Y. Li, Topology selection for particle swarm optimization, Inf Sci (Ny) 363 (2016) 154–173.
[32] P. Rabanal, I. Rodríguez, F. Rubio, Using river formation dynamics to design heuristic algorithms, in: International Conference on Unconventional Computation, Springer, 2007, pp. 163–177.
[33] E. Cantú-Paz, Topologies, migration rates, and multi-population parallel genetic algorithms, in: GECCO'99, Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1, Morgan Kaufmann Publishers Inc., 1999, pp. 91–98.
[34] S. Das, S. Maity, B. Qu, P. Suganthan, Real-parameter evolutionary multimodal optimization - a survey of the state-of-the-art, Swarm Evol Comput 1 (2) (2011) 71–88.
[35] R. Marler, J. Arora, Survey of multi-objective optimization methods for engineering, Struct. Multidiscip. Optim. 26 (6) (2004) 369–395.
[36] M. Laumanns, L. Thiele, E. Zitzler, Running time analysis of multiobjective evolutionary algorithms on pseudo-boolean functions, IEEE Trans. Evol. Comput. 8 (2) (2004) 170–182.
[37] L. Jiao, H. Wang, R. Shang, F. Liu, A co-evolutionary multi-objective optimization algorithm based on direction vectors, Inf. Sci. 228 (2013) 90–112.
[38] Q. Zhang, H. Li, MOEA/D: a multiobjective evolutionary algorithm based on decomposition, IEEE Trans. Evolutionary Computation 11 (6) (2007) 712–731.
[39] S. Kirkpatrick, C.G. Jr, M. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671.
[40] D. Karaboga, B. Basturk, Artificial bee colony (ABC) optimization algorithm for solving constrained optimization problems, in: Foundations of Fuzzy Logic and Soft Computing, 12th International Fuzzy Systems Association World Congress, IFSA 2007, in: Lecture Notes in Computer Science, 4529, Springer, 2007, pp. 789–798.
[41] G. Rudolph, Finite markov chain results in evolutionary computation: a tour d'horizon, Fundam Inform 35 (1–4) (1998) 67–89.
[42] D. Mitra, F. Romeo, A. Sangiovanni-Vincentelli, Convergence and finite-time behavior of simulated annealing, Adv Appl Probab 18 (3) (1986) 747–771.
[43] A.W. Johnson, S.H. Jacobson, On the convergence of generalized hill climbing algorithms, Discrete Appl. Math. 119 (1–2) (2002) 37–57.
[44] I. Trelea, The particle swarm optimization algorithm: convergence analysis and parameter selection, Inf. Process. Lett. 85 (6) (2003) 317–325.
[45] C. Lin, Q. Feng, The standard particle swarm optimization algorithm convergence analysis and parameter selection, in: Third International Conference on Natural Computation, ICNC 2007, Volume 3, 2007, pp. 823–826.
[46] M. Jiang, Y. Luo, S. Yang, Stochastic convergence analysis and parameter selection of the standard particle swarm optimization algorithm, Inf. Process. Lett. 102 (1) (2007) 8–16.
[47] C. Qian, Y. Yu, K. Tang, X. Yao, Z.-H. Zhou, Maximizing submodular or monotone approximately submodular functions by multi-objective evolutionary algorithms, Artif Intell 275 (2019) 279–294.
[48] T. Friedrich, F. Neumann, Maximizing submodular functions under matroid constraints by evolutionary algorithms, Evol Comput 23 (4) (2015) 543–558.
[49] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to automata theory, languages and computation, Pearson Education, 3rd edition.
[50] S. Arora, B. Barak, Computational Complexity, A Modern Approach, Cambridge University Press.
[51] A. Nolte, R. Schrader, Simulated annealing and its problems to color graphs, in: European Symposium on Algorithms, Springer, 1996, pp. 138–151.