Finding optimal finite biological sequences over finite alphabets: the OptiFin toolbox

Régis Garnier, Christophe Guyeux, and Stéphane Chrétien

Femto-ST Institute, UMR 6174 CNRS & Laboratory of Mathematics, Besançon

Université de Bourgogne Franche-Comté, France

Email: christophe.guyeux@univ-fcomte.fr

Abstract—In this paper, we present a toolbox for a specific optimization problem that frequently arises in bioinformatics or genomics. In this specific optimisation problem, the state space is a set of words of specified length over a finite alphabet. To each word is associated a score. The overall objective is to find the words which have the lowest possible score. This type of general optimization problem is encountered in e.g 3D conformation optimisation for protein structure prediction, or largest core genes subset discovery based on best supported phylogenetic tree for a set of species. In order to solve this problem, we propose a toolbox that can be easily launched using MPI and embeds 3 well-known metaheuristics. The toolbox is fully parametrized and well documented. It has been specifically designed to be easy modified and possibly improved by the user depending on the application, and does not require to be a computer scientist. We show that the toolbox performs very well on two difficult practical problems.

I. INTRODUCTION

Many biological data can be described as finite sequences over a finite alphabet. For instance, from a computer science viewpoint, a gene is nothing but a sequence of nucleotides A, C, G, T, and a protein is a sequence of amino acids (a.a., a set of 20 elements). In various situations, we can assign a score value to each finite sequence, that can account for its ability to solve a biological problem. For instance, these finite sequences can be the mapped to the descriptions of the protein's spatial configuration in the 3D square lattice; in this case, the score of the considered conformation is simply its energy. The main problem considered in this paper is the one of optimising the score value over all possible sequences. As is well known, most such problems are NP-hard and we often have to resort to efficient heuristics.

The main contribution of the present paper is to propose an handy and efficient toolbox that can solve such problems using a combination of metaheuristics. The toolbox is carefully designed in such a way that the sequences, their updates and the score function are easy to define and work with. Our toolbox – which is still in alpha stage, but in active development – incorporates the state of the art computation standards in computer science. It is mainly written in Python, it is MPI compliant, and it uses XML configuration files. The toolbox can be launched on a collection of computers through the network. After completion, the toolbox provides a report summarising various information concerning its convergence. We provide examples of application to phylogenetics and protein structure predictions.

The remainder of this article is as follows. In the next section, we present the main problem and motivate its relevance to bioinformatics via some illustrative examples. In Section III, 3 currently embedded metaheuristics are presented. The toolbox's architecture is described in Section IV. Application examples are provided in Section V. The article ends with a conclusion, in which the contribution is summarized and future work is outlined.

II. WHAT TYPES OF PROBLEMS ?

A. Main objective

Given an integer $n > 0$ and a finite set $\mathcal{A}$, we consider the set $\mathfrak{S}_n$ of sequences of length n whose elements belong to $\mathcal{A}$. In an equivalent manner, $\mathfrak{S}_n$ is the finite set of n-words $w = w_1...w_n$ whose letters w_i belongs to the alphabet $\mathcal{A}$. We consider a distance $d : \mathfrak{S}_n \times \mathfrak{S}_n \longrightarrow \mathbb{R}^+$, and we suppose that $d\left(\mathfrak{S}_n, \mathfrak{S}_n\right) \subset \mathbb{N}^*$: all distances between words are integer values. Finally, we consider a "scoring function" $s : \mathfrak{S}_n \longrightarrow \mathbb{R}^+$, that is, a function that maps each n-word to a real positive value. Our objective is to design a software that uses various heuristics to find the minimum value of s on $\mathfrak{S}_n$.

Another way of seeing is to construct an undirected graph. Each n-word can be identified with a node of a graph, while two nodes w and w' are connected with an edge if and only if $d(w, w') = 1$. At each node is associated its score. Our goal is then to design a way of moving from a node to the next in the graph, in such a way that the scoring function decreases over successive iterations. As will be clarified later, some sort of continuity will be needed : two nodes close enough must have, most of the times, approximately the same score. Some jumps will be acceptable, but should remain quite rare.

We will now motivate that this kind of problem arises in various instances in the field of bioinformatics by providing three important applications.

B. A first example: core genes and phylogeny

In the first example, we consider a set of species from which we want to infer a biomolecular phylogeny. For each species, we have a list of genes, i.e. a dictionary whose key is the gene name and value the DNA sequence. Although two given species do not necessarily have the same genes, any phylogenetic tree is based on the similarity of their common genes. Indeed, a phylogenetic tree can be computed as follows. Given a set of species, we extract a set of "core" genes (genes

present everywhere in this set of species). Each core gene is multi-aligned using a toolbox such as e.g. Muscle [1], and then the alignments are concatenated. A mutation matrix between nucleotides is then built and the estimation of the tree corresponds to finding the most likely tree that, given this model of evolution, is able to explain the mutations found in the alignment. Statistical techniques like bootstrapping can provide support values on the obtained tree, and these supports provide information on the reliable character of the associated branch. In other words, a branch with a large support value will be considered as trustworthy.

For the same species, the phylogenetic trees inferred by using two different subsets of common genes are not equal in general. At least, it is unlikely that they share the same branch length. More dramatic, some sister relationship between species may be different (the trees do not have the same topology). Finally, the two obtained trees may share the same topology, but with different support values: some weakness may appear in the first tree in branches considered as trustworthy in the second one. If the given set of species have n genes in common (their core genome has n genes), then we can consider $2^n - 1$ different phylogenetic trees that show a certain variability in terms of topology and supports. To choose a tree that faithfully represents the phylogenetic relationship between the species at hand, we may consider that: (1) it is as well supported as possible, and (2) it is based on a nucleotidic sequence which is as large as possible. If such an approach is chosen, our objective is now to find the largest subset of core genes that leads to the tree that receives the largest support. This can be achieved as follows.

The n core genes are sorted alphabetically, and to each subset we associate a binary word of length n: its i-th character is 1 if and only if the i-th core gene is in the considered subset. The alphabet $\mathcal{A}$ is then equal to $\{0, 1\}$. Given a binary word w, after alignment, we can use a tool like RAxML [2] to infer a tree as well as its supports. The score $s(w)$ of this binary word can be the average value of the percentage of 1's inside w (number of considered core genes) and the lowest support within the tree (an integer ranging between 1 and 100). This provides us with a score we will have to maximise. We can also consider the inverse of this score, so as to turn the optimisation problem into a minimisation one.

C. A second example: self-avoiding walks and protein folding

Nowadays, it is relatively easy to have access to the genome of a given species, and to predict its coding sequences, either by blasting [3] a set of known genes (basis of knowledge) on it or by using *ad hoc* prediction tools like Glimmer for bacterias [4]. It is thus quite easy to obtain the DNA sequence of a gene. A more difficult task is to predict its 3 dimensional shape, since this latter is mostly determined by the physico-chemical properties of its amino acid sequence, their coupling interactions, and so on. The protein prediction tools for conformation usually try to find the best conformation according to an optimization function. To do so, they focus on the hydrophylic/hydrophobic properties of each amino acid,
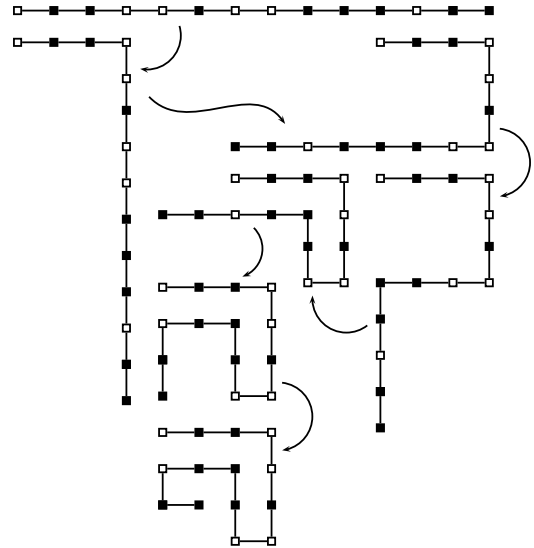

Fig. 1. Protein Structure Prediction by folding SAWs

and they try to find the 3D conformation that optimizes the number of hydrophobic a.a. inside the conformation (as proteins mainly evolve in aqueous solutions like cytoplasms).

In order to solve this optimization problem, a lattice is first considered, like the 2D or 3D square one, and then, a protein conformation in this lattice will be a self-avoiding walk (SAW [5]), i.e. a succession of adjacent nodes of the lattice connected with edges, and such that each node strictly inside the walk has a degree of 2 (as two a.a. cannot lie in the same location), see Figures 1 and 2. Given its sequence of hydrophobicity, finding the best 2D conformation of a protein is not an easy task. When considering the set of self-avoiding walks having n−steps and whose vertices are either black (hydrophobic) or white squares (hydrophylic residues), the authors of [6] have indeed proven that determining the SAWs that maximize the number of neighboring black squares in this set, which is supposed to be the conformation chosen by the protein, is NP-hard.

Given a sequence of amino acids, we will resort to using relevant heuristics for predicting the most probable conformation of the protein. The translation in terms of $\mathfrak{S}_n$ and scoring function of Section II-A is obvious. Each self-avoiding walk is a word on the {North, East, South, West} alphabet, which describes its absolute encoding. The scoring function, here, is the number of neighboring black squares.

D. A third example regarding the protein folding process

The problem described in the previous subsection also leads naturally to the study of various subsets of self-avoiding walks [7], [8], [9]. In the first approach, starting from the straight line, we obtain by a succession of pivot moves of $\pm 90°$ (to choose a node as center of a rotation of the tail) a final conformation being a self-avoiding walk. All the intermediate conformations must be self-avoiding too (see Fig. 1). Such a procedure is one of the two most standard implementations
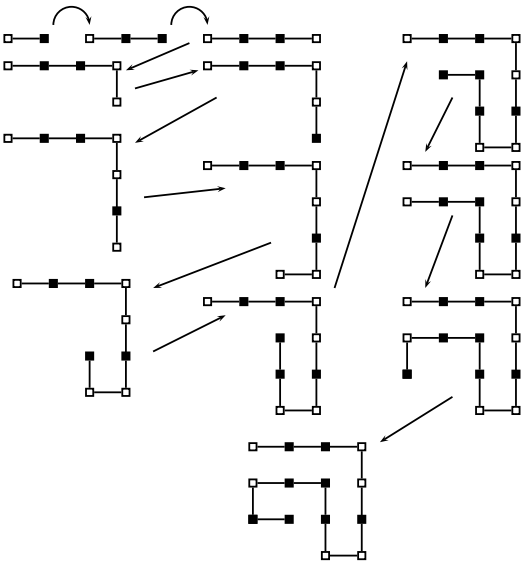
Fig. 2. Protein Structure Prediction by stretching SAWs

Fig. 3. An unfoldable self-avoiding walk

of the so-called "SAW requirement" in the bioinformatics literature, which is for instance presented in [10], [11], [12]. We have previously shown in [9] that, *in this first category of prediction software, it is impossible to reach all the possible conformations.*

The other main approach of protein structure prediction using self-avoiding walks starts with an $1-$step SAW, and at iteration k, a new branch is appended to the current tail of the walk, in such a way that the new $k-$step self-avoiding walk has a larger number of neighboring black squares (see Fig 2). The protein is thus constructed step by step, reaching the best local conformation at each iteration. It is easy to see that such an approach leads to all the possible self-avoiding walks having the length of the considered protein [9].

As stated above, these two approaches cannot reach the same set of conformations. One very bad problem faced by the community is that the difference between the two sets is far from negligible, leading to the necessity of comparing their respective sizes. Clearly, non unfoldable self-avoiding walks, that is, SAWs on which any pivot move introduces an intersection between the head and the tail of the new walk (like in Fig. 3), cannot be reached as the result of performing iterative pivot moves starting with the straight line. Our goal is to generate unfoldable self-avoiding walks. In order to do this, a relevant score will be the number of possible pivot moves, which must reduce to zero in the case of an unfoldable self-avoiding walk.

III. THREE EMBEDDED HEURISTICS

In the present section, we outline the three heuristics that we have embedded within our toolbox. Further details can be found in [13], [14] or in the software documentation[1].
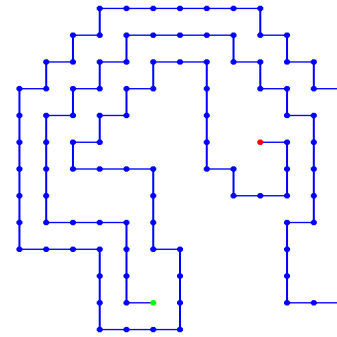
A. Genetic algorithm approach

The population is initialized with a collection of 50 random words. Then, the genetic algorithm iterates until discovering a word whose score is larger than a threshold, or at most for 200 iterations. Each iteration, which produces a new population, consists of the following steps:

1) Repeat 5 times a random pickup of a pair (w^1, w^2) of words and mix them using a crossover approach. In this step, indexes $\{1, \ldots, n\}$ are partitioned into k, $k \leq \frac{n}{2}$, subsets $I_1, \ldots I_k$. A new word w is then defined by $w_i = w_i^1$ if i belongs to some I_j where j is odd; otherwise $w_i = w_i^2$. The obtained words are added to the population P, resulting in population P_c.

2) Mutate 5 words of the population P_c. More precisely, for each of these words w, k randomly selected values of w are replaced by a character randomly picked in $\mathcal{A}$, leading to a new word. The mutated words are added to P_c leading to population P_m.

3) Produce population P_r by adding 5 new random words to P_m.

4) Select the 50 best words in population P_r to form the new population P.

The whole set of produced words with their associated scores contains valuable information about which parts of the words tend to decrease the scores. A Lasso test [15] may then be applied, to rank the word positions according to their impact on the scoring function. In this case, a last genetic algorithm phase is launched on the updated population, in order to mix these candidate words.

B. Particle swarm optimization approach

The particle position is a vector of N values belonging in $\mathcal{A}$, an alphabet of positive numbers. The objective is to define a way to move the particles in the N dimensional search space so that they produce the optimal vectors with respect to the scoring function. A swarm of L particles is then a list of position vectors $(X_1, X_2, \ldots, X_L)$ together with their associated velocities $(V_1, V_2, ..., V_L)$, which are n-dimensional vectors of real numbers between 0 and 1. The latter are

initialized randomly. At each iteration, a new velocity vector is computed as follows:

$$V_i(t+1) = wV_i(t) + \phi_1 \left(P_i^{best} - X_i \right) + \phi_2 \left(P_g^{best} - X_i \right), \quad (1)$$

where w, ϕ_1, and ϕ_2 are weighted parameters setting the level of each three trends for the particle, which are respectively: to continue in its adventurous direction, to move in the direction of its own best position P_i^{best}, or to follow the gregarious instinct to the global best known solution P_g^{best}. Both P_g^{best} and P_i^{best} are computed according to the scoring function. The new position of the particle is then obtained, by applying the sigmoid function [16] on each velocity, and by considering the value of $\mathcal{A}$ the most close to this latter.

C. Simulated annealing approach

After an initialization step, the Simulated Annealing loop is composed by (a) a move in the neighborhood of the current solution, (b) an evaluation of this new position by a real-valued scoring function, then (c) a test, given a well chosen criterion, to store this position as the new best one. Various criteria can be considered, they all have been embedded within the software. A detailed description of the simulated annealing algorithm is provided in Figure 4.

We are now able to describe the software that embeds these metaheuristics, in order to solve the general problem of Section II-A.

IV. SOFTWARE DESCRIPTION

A. Requirement Guidelines

Let us first outline the requirements that were set up before the conception of our toolbox.

- We need to assess the efficiency of the meta-heuristics (MH) for various models of solution spaces. Therefore, we focus on measuring and keep track of the performances, record events or messages (logfiles) related to our algorithms. In other words, introspection is preferred to speed.
- We need to allow for easy application to new problems the users might come up with. Similarly, new algorithms and heuristics must be easy to embed, as well as batteries of tests.
- As we intend to work on multi-cores or clusters and with scoring functions that putatively need a long computation time, we look for a scalable, ready, and reliable network solution. Furthermore, as the heuristic methods under consideration need various amount of resources during computation, we need a convenient scalability mechanism. For this purpose, we chose the Message Passing Interface (MPI) 2.2.
- Various existing scripts and third party applications need to be embedded or reused without code duplication, and both must be launchable for testing purposes or specific needs. Therefore, we need an easy to use but reliable shell script launcher, with basic communication and logging capabilities.
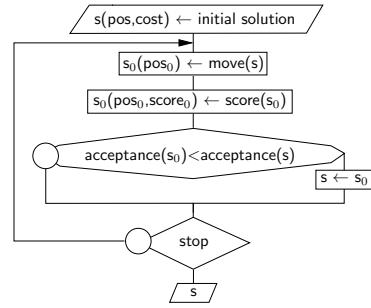
(a) Generic threshold class algorithm
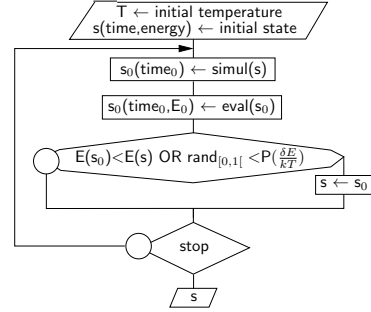
(b) Metropolis algorithm
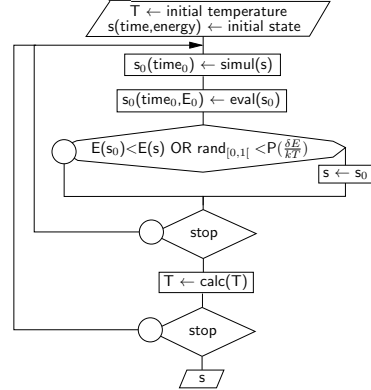
(c) Simulated annealing algorithm

Fig. 4. Simulated annealing as a threshold class algorithm.

- Finally, targeted end-users like bioinformaticians are typically familiar with the use of terminals and code scripting, but they are not necessarily computer scientists. This is why the Python language has been chosen. But with a C-like flavor syntax when possible, to produce codes easy to read for everyone.

B. Architecture Guideline

Software architecture is meant to convert requirements in a small list of well-known design patterns. Objectives of the latter is to provide a coherent approach that allows to easily reuse, maintain, and upgrade the proposal. Indeed, a pattern is used whenever possible, while they are reused as soon as possible to reduce their number, leading to an architecture of low complexity as depicted in Fig. 5. It is described hereafter.
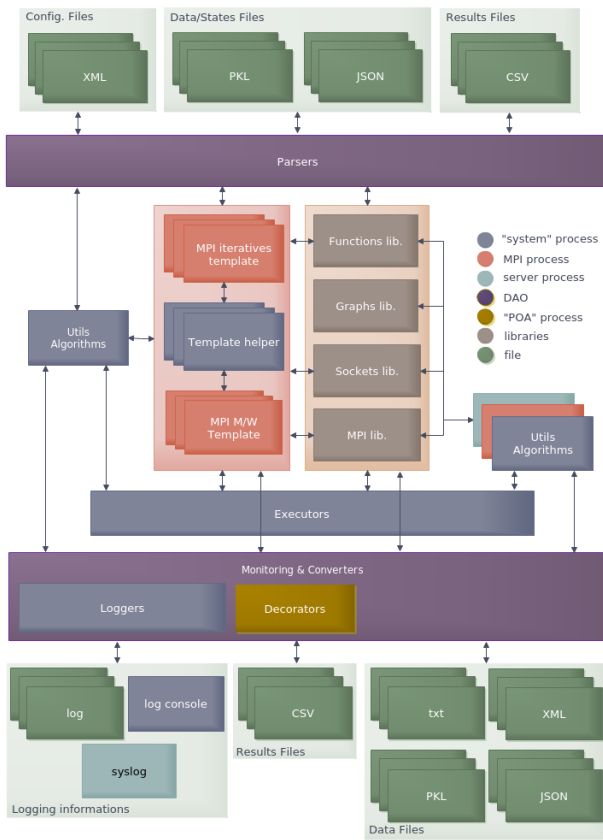
Fig. 5. Detailed architecture

Fig. 6. Dependency Injection, Factory and Mapping Example (slightly simplified)

1) Introspection – Decorators: Oriented Object Programming (OOP) is the main chosen paradigm of the framework but, as Python offers convenient ways to use it, a kind of Aspect Oriented Programming (AOP) has been preferred to separate some technical functions from domain ones. Python decorator (wrapper) has been the chosen pattern, which is in this language a simple decoration on a class, function, or method. Basically a decorator provides a way to intercept communication to or from a designated function. So inputs, outputs, and capability of a decorated class could be analyzed (loggers), validated (contracts), have methods replaced (monkey patching) or enhanced without modifying its code. Without the necessity to launch tests again on the decorated domain specific code. The function domain is also more readable and simpler to write. Note that a decorator can be shared with as much domain functions as needed. So decorators maximise the reuse of codes, leading to a decrease of both the testing time and the failure risk.

2) Framework – low coupling, helpers, and DAO:

a) Coupling and cohesion, Liskov principle: Coupling property is a quality metric that describes the interdependence degree between software modules. Cohesion, for its part, is another metric that describes how much elements of a given module are related on. Our objective was to increase as much as possible these two metrics, for each module within the software. As all MJs use both *move* and *score* concepts
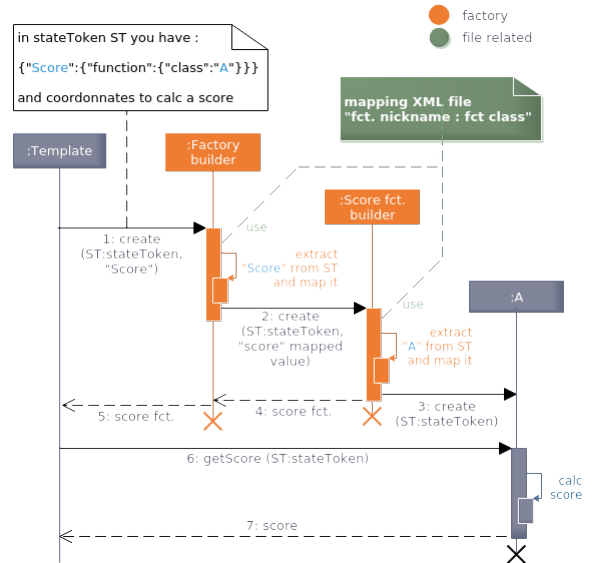
(we called them "field functions"), separated libraries have been built for these functions. Indeed, the generalization of a concrete *move* function leads to a common interface with all other *move* functions. Some characteristics are also shared by all *score* functions, which leads to a second interface. So, each member of our class of functions respects the so-called Liskov substitution principle: if S is a subtype of T, then objects of type T can be replaced with objects of type S.

b) Factory, generic programming, and mapping: If we have substitutive functions but call a specific type of them in the main MH template, we need to change code of template each time we change experiment conditions. The coupling must be lower. We could call a function that is interface compatible. So we need a tool to select function in a cohesive collection from a XML hashtables and return instance of this function. This is provided by the factory and dependency injection pattern.

As no reference to effective calculus function remains in the main MH code, we abusively speak about *templates*, in a C++ terminology. Launcher scripts are mostly strategy patterns, *i.e.*, they could be launched by any elemnts of the templates. By doing so, the targeted quality metric was genericity: it measures the ability for effective types of data to be defined dynamically, *i.e.*, only when instantiated. With this low-decoupling approach, the framework allows a zero-modification reuse of old field functions with a new template, and *vice versa*.

c) Template helpers: MH templates often share same concepts but with specific implementation. Associated functions of common concept are outsourced. This approach has been chosen as we wanted not to allow any lambda user to practice a top-down programming. Furthermore, the objective was to discourage a "scripting" bottom up style for templates,

as authors considered that users will want to plan neither unit nor integration tests on templates. By doing so, code review is more easy to achieve.

d) DAO and loggers: As stated previously, various formerly existing shell scripts were at the beginning of this software, and we tried to group them in a unique benchmarkable and reliable environment. Scripts usually pipe applications, while inputs and outputs of these applications are usually files. To be readable, all these files must follow specific standards. As a consequence, conversions, sorting, or subset selection are sometimes required regarding these files. To prevent from issues that may be raised by such a variety of inputs and outputs, two priorities have been kept in mind: an improved and stringent logging and a file system (Data Access Object, *i.e.*, DAO pattern) helpers.

3) Maintainability – Third-party libraries policy: Authors of this software have *a priori* considered that Python is not the most used language, and so developers may not be familiar with its ecosystem. This lack of background does not allow all developers to sort available libraries according to their maturity or their reliability, while various libraries from the Python ecosystem are still in an early reversion designed from scratch. Consequently, we decided to use as few dependencies as possible, leading to a very simple proposal easy to improve, customize, and fork.

4) Scalability – MPI 2.2 and sockets libraries, server MPI factory: Let us firstly recall that Message Passing Interface (MPI) is a concurrency inter-process communication system based on message-passing. The message-passing paradigm implies than process code execution is message-driven, *i.e.*, the type and content of message select the code to invoke for its receiver. MPI is a Language Independent Specifications (LIS), point to point, and multicast communication protocol. It mainly provides a set of basic tools for synchronisation and data exchange, being an ISO 5th, 6th, and 7th level middleware and application framework using TCP sockets. APIs are provided for C, C++, and Fortran languages so, in Python, a binding library or a MPI standard implementation is needed. This latter is described hereafter.

a) MPI4py and OpenMPI: MPI for Python (MPI4py) is based on the standard MPI 2.2 C++ bindings and it provides a similar interface. But MPI4py adds an unique Python capability, pickling, to send/receive objects in RAM or HDD buffers. The widely used middleware OpenMPI 2 has been chosen, as this is a software of production quality which is fully MPI 2.2 and 3.1 compliant. It allows process spawning, network and process fault tolerance, safe threads, a good portability, and so on. Last, but not least, it is well documented and supported.

b) Client/server mode and libraries: MPI4py over OpenMPI third party library was a prerequisite as most of existing scripts use it. However, after an intensive evaluation, its low reliably with long series of connections, spawning, or big message exchanges was quite disappointing. Unfortunately, no good alternative has been found.

As a consequence, we decided to focus on the MPI 2.2 client/server (CS) mode, while minimizing as possible the coupling between MPI4py and the other codes within the software. Indeed, only a MPI-helper library requires it.

c) MPI process server factory: Using OpenMPI name server, a socket helper library and the OpenMPI helper, we allow a MPI process to launch another MPI 2.2 subprocess (Fig. 7). And to free resources after results return. This server XML configured factory with socket communication channel is a standalone script (singleton).

d) MPI most used pattern – Master and Workers: In this parallel computation pattern, a main thread called the master splits a given problem and merges partial results effectively computed by a set of forked processes, which are called the workers. This pattern can rely on a message-passing architecture to broadcast data to workers and regroup partial obtained results. This is typically a good choice when a problem can be sliced in a set of same smaller and simpler ones, splitting and merging times being small when compared with sub-problems computation ones.

5) Reuse with low-coupling – Executor library: The MPI server factory needs to execute MPI and Python codes in a shell, and with a maximum of security. As a set of applications and scripts has been added that use XML configuration files for their environment, a shell launcher has been required, which should follow an "easy to use" philosophy like in Java 5 executor with easy contracting and centralized logging abilities.

6) Introspection – general architecture and dataflow programming: According to the observation granularity, the proposed framework uses various classical software architectures. Some components as the MPI factory server induce their own main architecture, and the classical architecture erosion (gap between models and implementation) is observed. This is why, as much as possible, Separation of Concerns (SoC) is used at high level structures.

But a significant choice has been forced and preserved over versions concerning the "state" data of each MH process. Each problem (that is, the combination of a MH script, field functions like the move or score ones, and their parameters) is defined in an unique XML file. This XML is validated by XSD; it is parsed, and then converted in a Python dictionary, *i.e.*, a hash-table implementation. So (almost) all the needs start in an unique entity. The choice has been made to preserve this entity, and to add in it all what MH templates require to be instantiated. In other words, this is principally a memory for current coordinates and score when moving on the solution space. This state token, which is a dataflow programming-inspired architecture, is passed to each function and all of them return new version of the token.

With this approach, MH templates can be modeled by graphs, thus simplifying both software monitoring and tuning. Additionally, the check-pointing strategy that corresponds to store this token leads to various interesting properties, like the possibility to restore a computation from a given saved point, to study the convergence speed, to benchmark various physical platforms on a same computation, or to launch in parallel various instances to maximize the chance to find interesting
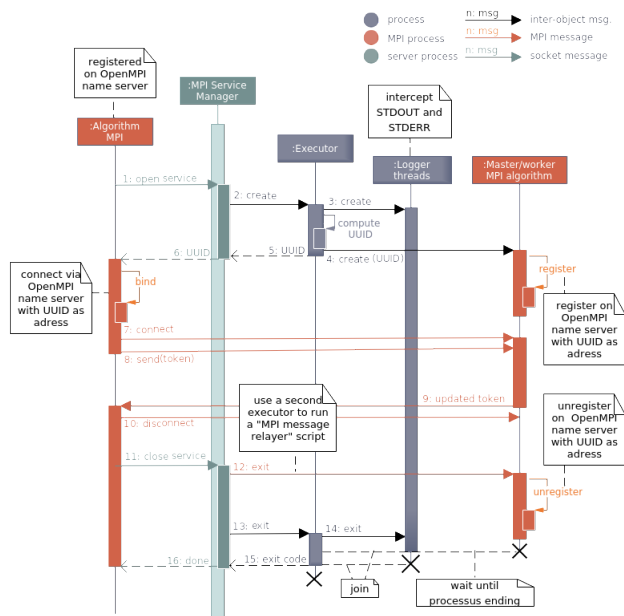
Fig. 7. Multi-level MPI 2.2 with Server MPI Factory and Executor Class functioning Principles (slightly simplified)

solutions (or to study the dispersion of solutions). We thus improve both introspection and reliability.

C. How to construct a MH in practice

As a framework, a toolbox for computing resources is provided, together with a methodology to create MH. In broad outline some simple templates are created, helper are written with DOA and helpers, then when needed, MPI processes are nested with MPI helper and standalone MPI C/S factory. Figure 8 illustrates a SA evolution build by iterating SA and greedy local descent basic algorithms. The first step was to construct both SA and local descent in an one-core version. They are then linked by a constructor to apply a descent on each solution selected by the SA Metropolis criterion. In a second step, the descent is replaced by a MPI version: some descents are launched in Workers and the Master returns the best Worker result. The simple Executor is replaced by a call to the MPI server factory. Then, in a third step, the simple SA execution itself is injected in a Worker and a Master, created to make a synchronisation after each Markov chain.

On Figure 8, dashed arrows represent data circulation, and then moves of the token instances.

V. APPLICATIONS

A. Protein folding

The first illustrative example of the effectiveness of the proposed software is related to the protein folding problem presented in Section II-D. More specifically, given a sequence length, the objective is to find a self-avoiding walk of this length which is non unfoldable: to apply a pivot move at each internal node always leads to a new walk that does not satisfy the self-avoiding requirement.

Accession Nb	Name	Nb. of genes
NC_024082.1	Cylindrotheca closterium	257
NC_014808.1	Thalassiosira oceanica CCMP1005	138
NC_025313.1	Cerataulina daemon	195
NC_028052.1	Pelargonium cotyledonis	271
NC_015403.1	Fistulifera solaris	192
NC_024084.1	Leptocylindrus danicus	155

TABLE I
FAMILY NUMBER 1 (PELARGONIUM COTYLEDONIS AS OUTGROUP).

We considered the number of nodes on which a pivot move can be applied as the scoring function. When this score is zero, then a non unfoldable self-avoiding walk has been obtained. As a first basic experiment, we considered the SA meta-heuristic on short sequences, to reduce the computation time. Obtained results are depicted in Figure 9, which emphasize the convergence ability of the proposed software on the considered problem.

B. Phylogeny

We have first considered phylogenetic issues, with the family listed in Table I. This family is constituted by 6 genomes, with a number of detected genes that ranges from 138 to 271, and a core genome of 122. The phylogeny with the alignment of these core genes leads to a small weakness in one branch (bootstrap lower than 95). To wonder whether some genes may be responsible of such weak uncertainty, we have firstly launched the genetic algorithm, which has stopped after 29 iterations, in systematic investigation mode, leading to 2 topologies:

- Topology 0 depicted in Fig. V-B, that has occurred 27 times. The best obtained tree has a lowest bootstrap of 96, while in average the lowest bootstrap is equal to 86.
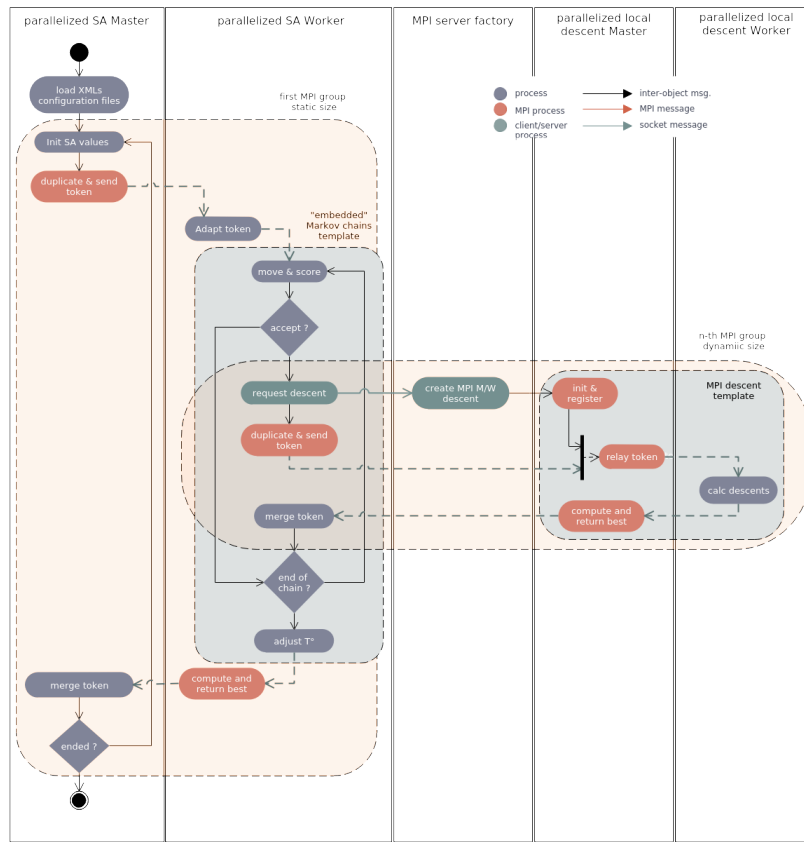
Fig. 8. 2-level MPI 2.2 application : SA with Basin Hopping algorithm (simplified)

- Topology 1, for its part, has occurred twice, with a non supported branch of 64 in its best tree. It is the same than Top. 0, except that the clade in newick format ((*C.closterium*,*F.solaris*),(*C.daemon*,*L.danicus*)) becomes ((((*C.closterium*,*F.solaris*),*C.daemon*),*L.danicus*).

The PSO, for its part, has rapidly found a first well supported phylogenetic tree in a third different topology, and with all supports equal to 100, namely: ((((*C.closterium*,*F.solaris*),*L.danicus*),*C.daemon*). However, the PSO has used only 47.5% of the core genes to reach such a tree. According to our stop criterion, this tree has not been returned by the algorithm. Indeed, this example illustrates the ability of the particle swarm optimization algorithm to more globally visit the whole space at the beginning, in order to discover regions of interest.

The simulated annealing, for its part, raised these 3 topologies. A remarkable element is that these 3 topologies have the whole bootstraps equal to 100. Furthermore, Topology 2 appears as the best one according to the produced result (it was Topology 0 according to the GA, while PSO has not succeeded in separating these two topologies). Obviously, both PSO and SA have converged to local minima that are not global ones if we consider that both minimum bootstraps and proportion of core genes must be maximized. Launching them again with other initial values and parameters may select other optimal positions in the cube.

VI. CONCLUSION

In this article, we have detailed the general principles at the bases of an efficient software we propose for the bioinformatician community. This software iterates on finite words on a finite alphabet, such that each word has a score. Using one of the three most classical metaheuristics, this software is able to discover the optimums in a distributed manner. The software is easy to configure, to use, and to update. It can resolve a variety of bioinformaticians problems, like to measure the effects of genes on a phylogeny.

In future work, we intend to improve the following elements. First of all, the XML "problem" configuration file must have a more common structure, to allow state token to be used by more than one specific MH template. The objective is as follows: a problem, started to be resolved by a SA, should be finalized by a PSO, and *vice versa*. An improvement of the execution speed must be achieved, among other things by substituting some of our functions by more efficient and reliable ones provided by third-party libraries. Finally, the number of tests, contracts, and code factoring must be enlarged, together with logs, error management, and documentation, to help bug tracking, maintainability, safety, and data integrity.
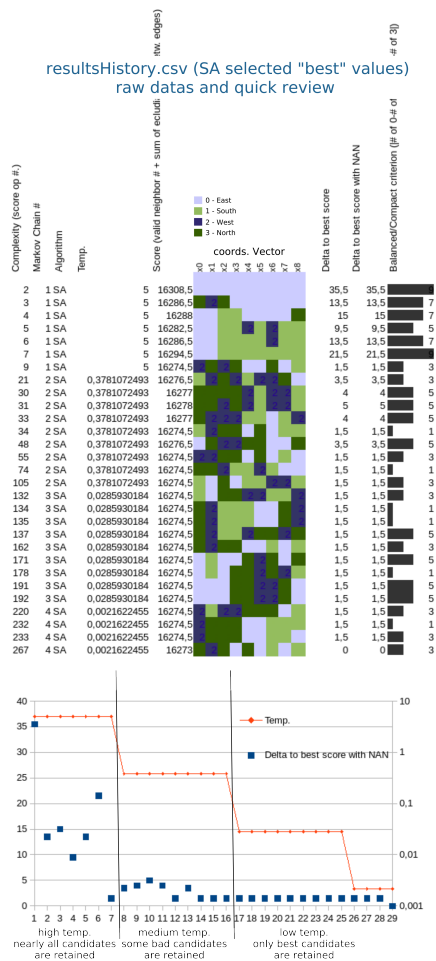
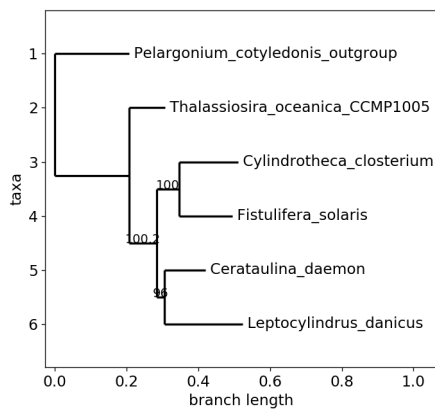Fig. 9. Example of output exploitation using the SA on a short size problem.

Fig. 10. First obtained supported topology.

References

[1] Robert C Edgar. Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic acids research*, 32(5):1792–1797, 2004.

[2] Alexandros Stamatakis, Thomas Ludwig, and Harald Meier. Raxml-iii: a fast program for maximum likelihood-based inference of large phylogenetic trees. *Bioinformatics*, 21(4):456–463, 2005.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, October 1990.

[4] Arthur L Delcher, Kirsten A Bratke, Edwin C Powers, and Steven L Salzberg. Identifying bacterial genes and endosymbiont dna with glimmer. *Bioinformatics*, 23(6):673–679, 2007.

[5] KA Dill. Theory for the folding and stability of globular proteins. *Biochemistry*, 24(6):1501–9–, March 1985.

[6] Pierluigi Crescenzi, Deborah Goldman, Christos Papadimitriou, Antonio Piccolboni, and Mihalis Yannakakis. On the complexity of protein folding (extended abstract). In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 597–603, New York, NY, USA, 1998. ACM.

[7] Jacques Bahi, Nathalie Côté, Christophe Guyeux, and Michel Salomon. Protein folding in the 2D hydrophobic-hydrophilic (HP) square lattice model is chaotic. *Cognitive Computation*, 4(1):98–114, 2012.

[8] Jacques Bahi, Nathalie Côté, and Christophe Guyeux. Chaos of protein folding. In *IJCNN 2011, Int. Joint Conf. on Neural Networks*, pages 1948–1954, San Jose, California, United States, July 2011.

[9] Christophe Guyeux, Nathalie M.-L. Côté, Wojciech Bienia, and Jacques Bahi. Is protein folding problem really a NP-complete one? first investigations. *Journal of Bioinformatics and Computational Biology*, *(*):***–***, 2013. Accepted manuscript. To appear.

[10] Md. Kamrul Islam and Madhu Chetty. Clustered memetic algorithm for protein structure prediction. In *IEEE Congress on Evolutionary Computation* [17], pages 1–8.

[11] Ron Unger and John Moult. Genetic algorithm for 3d protein folding simulations. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 581–588, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[12] Trent Higgs, Bela Stantic, Tamjidul Hoque, and Abdul Sattar. Genetic algorithm feature-based resampling for protein structure prediction. In *IEEE Congress on Evolutionary Computation* [17], pages 1–8.

[13] Bassam AlKindy, Christophe Guyeux, Jean-François Couchot, Michel Salomon, Christian Parisod, and Jacques M. Bahi. Hybrid genetic algorithm and lasso test approach for inferring well supported phylogenetic trees based on subsets of chloroplastic core genes. *International Conference on Algorithms for Computational Biology, AlCoB 2015*, pages 1–15, 2015.

[14] Reem Alsrraj, Bassam Alkindy, Christophe Guyeux, Laurent Philippe, and Jean-François Couchot. Well-supported phylogenies using largest subsets of core-genes by discrete particle swarm optimization. In *CIBB 2015, 12th Int. Meeting on Computational Intelligence Methods for Bioinformatics and Biostatistics*, pages ***–***, Naples, Italy, sep 2015.

[15] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society (Series B)*, 58:267–288, 1996.

[16] Mojtaba Ahmadieh Khanesar, Mohammad Teshnehlab, and Mahdi Aliyari Shoorehdeli. A novel binary particle swarm optimization. In *Control & Automation, 2007. MED'07. Mediterranean Conference on*, pages 1–6. IEEE, 2007.

[17] *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, Barcelona, Spain, 18-23 July 2010*. IEEE, 2010.