

Generic Simulated Annealing

Chadi Kallab¹, Samir Haddad^{1*}, Jinane Sayah², Mohamad Chakroun³

¹Department of Computer Science and Mathematics, Faculty of Arts and Sciences, University of Balamand, Koura, Lebanon

²Department of Telecom and Networks, Issam Fares Faculty of Technology, University of Balamand, Koura, Lebanon

³Faculty of Computer Science and Electrical Engineering, Universität Rostock, Rostock, Germany

Email: chadi.kallab@fty.balamand.edu.lb, *samir.haddad@balamand.edu.lb, jinane.sayah@balamand.edu.lb,

Mohamad.chakroun@gmail.com

How to cite this paper: Kallab, C., Haddad, S., Sayah, J. and Chakroun, M. (2022) Generic Simulated Annealing. *Open Journal of Applied Sciences*, 12, 1011-1025. <https://doi.org/10.4236/ojapps.2022.126069>

Received: May 28, 2022

Accepted: June 26, 2022

Published: June 29, 2022

Copyright © 2022 by author(s) and Scientific Research Publishing Inc.

This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).

<http://creativecommons.org/licenses/by/4.0/>



Open Access

Abstract

One of the many problems that are considered to be NP-Hard is the Multiple Sequence Alignment one that initially requires, as for any other of its siblings, a specific encoding schema and design of the main functionalities of the heuristics algorithm being implemented and executed. This paper intends to discuss our proposed generic implementation of the Simulated Annealing, inspired for the procedure of cooling and shaping methods of metals. In our algorithm, we attempted to add some executions tracing functionalities in order to help later analysis for initial parameters tuning. On another hand, we also tried to get closer in our attempt to mimic the cooling of metals, but giving it an option to run under different cooling schedules. We proposed a few schedules that seemed to be studied and/or used in many algorithm implementations.

Keywords

Generic, Heuristics, Phylogenies, Bio-Informatics, NP-Hard, Simulated Annealing

1. Introduction

One of the many problems that are considered to be NP-Hard is the Multiple Sequence Alignment one that initially requires, as for any other of its siblings, a specific encoding schema and design of the main functionalities of the heuristics algorithm being implemented and executed. This design was supported by references [1] [2] [3] and [4].

The main issue with that problem, as discussed in our ICeND2013 conference, was to come up with an encoding that would be suitable for the different Heuristics algorithms inspired by papers [5] [6] [7] [8] and [9]. Once the encoding is

agreed on, the remaining part of designing and building the algorithm that will get a solution as close to the optimum as possible would be nonetheless as important. We have selected for our research to focus on the Simulated Annealing heuristic. However, we discovered that to be able to adjust some of the flaws of the standard algorithm, we improved it by adding functionalities like tracing the executions, so that later analysis could help better tune the initial parameters.

Papers [10] [11] [12] [13] come to prove the need for a generic and flexible implementation of Simulated Annealing with different tunable parameters.

In this paper, we'll be discussing the implementation of the Simulated Annealing, with some modifications added to try to shorten the gap between the algorithm and the source that inspired it, which is the structure and behavior of metal cooling and shaping.

2. Initial Algorithm

Standard Simulated Annealing

Simulated Annealing (SA) was initially designed to simulate the cooling of metals into crystalline structure (annealing process). Since the natural annealing process aims to minimize energy in resulting crystals, SA tackles problems from the point of view of minimizing cost/energy.

SA was intended in 1983 to be used with non-linear problems. "SA approaches the global maximization problem similarly to using a bouncing ball that can bounce over mountains from valley to valley." [14] Initially, the temperature is set high enough so that this ball is given enough time to bounce, between and within valleys. The generic annealing algorithm/process suggests that, as time passes, the ball tends to get closer to its optimal location. Therefore, as the temperature cools down, the frequency of the large bounces (between valleys) tends to lower, while that of local bounces (within valleys) increases. However, to avoid falling in a local optimum valley, a probabilistic formula is applied, related to the current temperature gain and cost of bounce from the current location to a new one. In SA, the bounce (change in location) implies to perturb the current solution S into a new one S' . In the SA algorithm, the temperature parameter is denoted as T , and the number of moves (perturbations/bounces) is denoted by M . "It has been proved that by carefully controlling the rate of cooling of the temperature, SA can find the global optimum" [12]. This idea was also further supported by references [6] [7] and [15].

3. Proposed Algorithm

The main idea behind Simulated Annealing was to simulate the annealing of metals, where the metal is mapped into an initial solution, and the atom moves mapped into moves to neighbor solutions, each of which is accepted, with a certain probability, if the cost of the neighbor solution is better than the current and best one found so far. The probability of accepting a solution is \leq a threshold value, directly proportional to the current temperature, and inversely to the dif-

ference in cost between the neighbor and current solutions. As the temperature cools down according to a given schedule, the algorithm is getting us closer to the optimal solution. Therefore, it allows the current solution to perturb more often. This process is repeatedly done for a given maximum number of moves, by applying the cooling rate α to the temperature, and motion rate β .

The steps of the below discussed algorithm are almost very similar to those of the basic standard algorithm, with the difference that some of them, highlighted in blue, offer the possibility to later on trace back each execution and allow better analysis, thus resulting in eventual update of initial parameters. The words/fragments highlighted in green are more of structure and/or behavior enhancements that try to bridge the gap between the initial annealing process and our simulated annealing algorithm.

3.1. Main Procedure

The steps of the algorithm implemented are as follows:

Inputs: Initial Solution S_0 , Initial Temperature T_0 ,
Cooling Rate α Motion Rate β
Initial number of moves M_0 ,
Maximum Annealing time

Precondition: $\alpha < 1$ and $\beta > 1$

Outputs: Optimal Solution $BestS$,

Algorithm:

Assign initial solution " S_0 " to current solution " $CurS$ "

Assign initial temperature " T_0 " to current " T "

Assign initial number of moves " M_0 " to the current " M "

Assign " $CurS$ " to optimal solution " $BestS$ "

Compute cost of " $CurS$ ", and save it in " $CurCost$ "

Compute cost of " $BestS$ ", and save it in " $BestCost$ "

Initialize to zero the Annealing time " $Time$ "

Repeat

Record, in global history, parameters: $\{CurS, BestS, T, M, Time\}$

Update parameters by calling: Metropolis procedure

on $CurS, CurCost, BestS, BestCost, T, M$.

Record in global history that of Metropolis execution

Include in Time number of moves done

Cool down temperature " T " according to rate " α "

Increment number of moves M according to rate " β "

Until " $Time \geq MaxTime$ "

Return " $BestS$ "

3.2. Metropolis Procedure

The steps and parts of the proposed algorithm are as follows, and visualized in **Figure 1**:

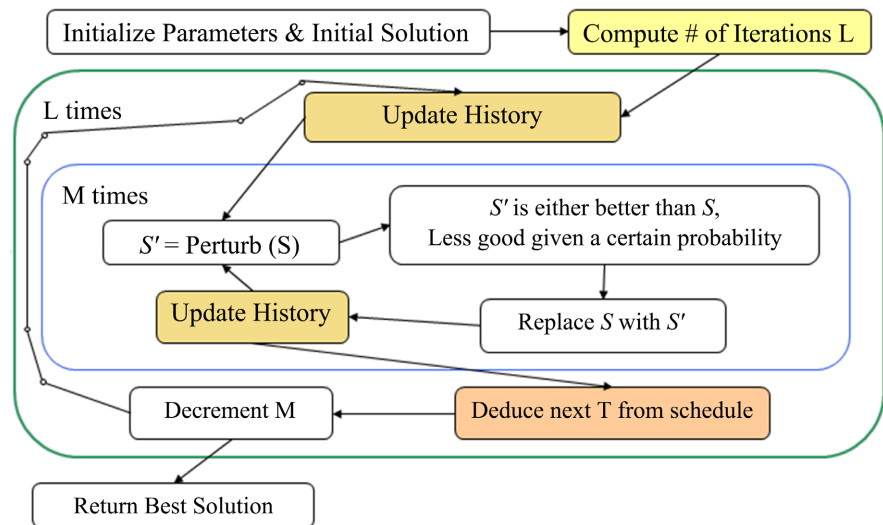


Figure 1. Suggest algorithm flowchart diagram.

Inputs: Current Solution “*CurS*” Cost “*CurCost*”
 Best Solution “*BestS*” Cost “*BestCost*”
 Current Temperature “*T*” Current # Moves “*M*”

Outputs: ---

Algorithm:

Assign number of moves “*M*” to the current variable “*Moves*”

Repeat

Generate a solution neighbor “*NewS*” for “*CurS*”

Compute cost of “*NewS*”, and save it in “*NewCost*”

Record, in local history, parameters: {*CurS*, *NewS*, *BestS*}

Compute “*DeltaCost*” = “*NewCost*” – “*CurCost*”

If “*DeltaCost*” < 0 **then**

{{“*NewS*” has lower cost than “*CurS*”}}

Assign New solution and cost to current

If *NewCost* < *BestCost* **then**

{{*NewS* has lower cost than *BestS*}}

Assign the New solution and cost to best

Else

{{“*NewS*” is higher than that of “*CurS*”}}

If $random() < 1 / e^{(DeltaCost / T)}$ **then**

Assign *new* solution and cost to *current* structure and value

Decrement the value of “*Moves*” by 1

Until “*Moves*” = 0

Per simulation iteration, the Metropolis procedure computes *M* neighbor solutions. Like the “Cost” method, the “Neighbor” function is problem specific. Every problem that needs to run Simulated Annealing needs to implement both “Cost” and “Neighbor”.

Below is an algorithm that helps tune the initial temperature parameter of

the Simulated Annealing, according to the standard and simple cooling schedule.

3.3. Tuning Initial Temperature Procedure

Inputs: Current Solution “*CurS*” Cost “*CurCost*”
Starting Temperature “*T*” # Moves Attempt “*M*”
Result Tolerance “*Threshold*”

Outputs: Initial Temperature “*T₀*”,

Algorithm:

Assign starting temperature “*T*” to initial “*T₀*”
Assign value of $1 - \text{“Threshold”}$ to variable “*TD*”

Do

Assign number of moves “*M*” to # trials “*Attempted*”
Initialize to zero # accepted moves “*Accepted*”
Assign number of moves “*M*” to current “*Moves*”

Repeat

Generate a solution neighbor “*NewS*” for “*CurS*”
Compute cost of “*NewS*”, and save it in “*NewCost*”
Compute “*DeltaCost*” = “*NewCost*” – “*CurCost*”

If “*DeltaCost*” < 0 **then**

Assign *new* solution and cost to *current* structure and value
Increment “*Accepted*” by 1 move

Else-If $\text{random}() < e^{(\text{DeltaCost}/T)}$ **then**

Assign *new* solution and cost to *current* structure and value
Increment “*Accepted*” by 1 move

Decrement the value of “*Moves*” by 1 move

Until “*Moves*” = 0

Compute ratio “*Ratio*” = “*Accepted*” / “*Attempted*”

While “*Ratio*” ≥ “*TD*”

Return “*T₀*”

3.4. Mathematical Design

Customized Simulated Annealing

Various cooling schedules, mentioned later, can be used with a Simulated Annealing optimization. Let T_i be the temperature for iteration i , where i increases from 1 to N . The number of iterations is indirectly determined by the user through: *MaxTime*.

This customized SA algorithm computes the number of cooling down iterations, beforehand, and tries to make each one as independent of the others as possible. Thus, the temperature, number of moves and time should be as unrelated as possible to the respective values computed in previous iterations.

Let t_i be the time elapsed up to iteration i , and M_i the number of moves for this iteration.

$$t_i = \begin{cases} t_{i-1} + M_{i-1} & \text{if } 1 < i \leq N \\ M_0 & \text{if } i = 1 \end{cases}$$

where: $M_i = \beta * \begin{cases} M_{i-1} & \text{if } 1 < i \leq N \\ M_0 & \text{if } i = 1 \end{cases}$

$$M_i = \beta^i M_0 \text{ for } i = 1, 2, \dots, N$$

$$t_1 = M_0$$

$$t_2 = t_1 + M_1 \Rightarrow t_2 = M_0 + M_1 \Rightarrow t_2 = (1 + \beta) * M_0$$

$$t_3 = t_2 + M_2 \Rightarrow t_3 = (1 + \beta) * M_0 + M_2 \Rightarrow t_3 = (1 + \beta + \beta^2) * M_0$$

$$t_4 = t_3 + M_3 \Rightarrow t_4 = (1 + \beta + \beta^2) * M_0 + M_3 \Rightarrow t_4 = (1 + \beta + \beta^2 + \beta^3) * M_0$$

Similarly:

$$t_k = M_0 * \sum_{j=0}^{k-1} \beta^j$$

$$\Rightarrow t_k = \left(\frac{\beta^k - 1}{\beta - 1} \right) * M_0 \text{ for } k = 1, 2, \dots, N$$

At the end of the simulation- $t_N = MaxTime$

$$MaxTime = \left(\frac{\beta^N - 1}{\beta - 1} \right) * M_0$$

$$\Rightarrow (\beta^N - 1) = \frac{(\beta - 1) * MaxTime}{M_0}$$

Then:

$$\beta^N = 1 + \frac{(\beta - 1) * MaxTime}{M_0} \Rightarrow N = Integer \left(\log_{\beta} \left\{ 1 + \frac{(\beta - 1) * MaxTime}{M_0} \right\} \right)$$

Therefore:

$$N = Integer \left(\frac{1}{\ln(\beta)} * \ln \left(1 + \frac{MaxTime * (\beta - 1)}{M_0} \right) \right)$$

The different schedules, implemented in the following cooling schedule, are illustrated in **Table 1**. These schedules are used in the customized algorithm by referring to the corresponding code, which is an integer value between 1 and 9. For flexibility reasons, code 10 is left to allow the algorithm to handle any externally defined schedule.

3.5. Main Procedure

Inputs: Initial Solution “ S_0 ” Cooling Schedule “*code*”
 Initial Temperature “ T_0 ” Total cooling Rate “ R ”
 Motion Rate “ β ” Initial # moves “ M_0 ”
 Maximum Annealing time “*MaxTime*”

Precondition: “ R ” < 1 “ β ” > 1

“*code*” corresponds to one of the schedules of **Table 1**

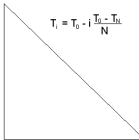
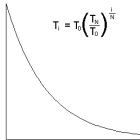
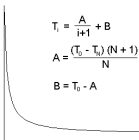
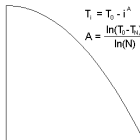
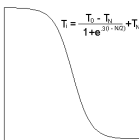
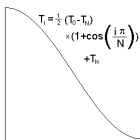
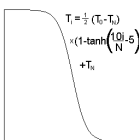
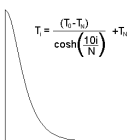
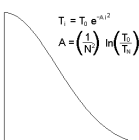
Outputs: Optimal Solution “*BestS*”

Algorithm:

Assign initial solution “ S_0 ” to current “ $CurS$ ”
 Assign initial temperature “ T_0 ” to current “ T ”
 Assign initial number of moves “ M_0 ” to current “ M ”
 Assign current solution “ $CurS$ ” to optimal “ $BestS$ ”
 Compute cost of “ $CurS$ ”, and save it in “ $CurCost$ ”
 Compute cost of “ $BestS$ ”, and save it in “ $BestCost$ ”
 Apply the above mentioned formula for “ N ”
For i **from** 1 **to** “ N ” **do**
 Record, in global history, parameters { $CurS, BestS, T, M, Time$ }
 Update parameters by calling Metropolis
 on { $CurS, CurCost, BestS, BestCost, T, M$ }
 Record in global history that of Metropolis execution
 Assign to “ T ” result of calling CoolingSchedule on ($code, i, N, R, T_0$)
 Set value of “ M ” as “ $M_0 * Power(\beta, i)$ ”
Return “ $BestS$ ”

In comparison with the standard S.A algorithm, this algorithm allows the simulation to occur with a non-scalar cooling schedule, by specifying the code of the cooling schedule. One difference is the absence of the time variable replaced by a loop-iteration index. Another difference is the fact that the temperature and the number of moves are computed independently of previous iterations, but according to the iteration index and some other parameters. Since the global

Table 1. Some suggested cooling schedules.

Code	Name	Graph	Code	Name	Graph
1	Linear		2	Scalar	
3	Hyperbolic		4	Exp.	
5	Sigmoid		6	Cos	
7	Tanh		8	Cosh	
9	Squared Scalar				

initialization phase was replaced by an iteration initialization, we have instead of i . This algorithm uses the Metropolis procedure defined above by the Standard Simulated Annealing procedure.

3.6. Cooling-Schedule Procedure

Inputs: Cooling Schedule “code” Current Iteration Index “ i ”
Number of Cooling Steps “ N ” Initial Temperature “ T_0 ”

Precondition:

“code” corresponds to one of the schedules of **Table 1**

Outputs:

Current Temperature “ T ”

Algorithm:

Case “code” is

$$1: \text{Return } T_0 - \left[\frac{i * (1 - R)}{N} \right]$$

$$2: \text{Return } T_0 * R^{\frac{i}{N}}$$

$$3: \text{Return } T_0 * \left[\frac{(1 - R)(N + 1)}{N(i + 1)} + \frac{R(N + 1) - 1}{N} \right]$$

$$4: A = \frac{\ln(T_0 * (1 - R))}{\ln(N)}$$

$$\text{Return } T_0 - i^A$$

$$5: A = 0.3 * \left(i - \frac{N}{2} \right)$$

$$\text{Return } T_0 * \left[\frac{(1 - R)}{1 + e^A} + R \right]$$

$$6: \text{Return } \left(\frac{T_0}{2} \right) * \left[1 + R + \left\{ (1 - R) * \cos\left(\frac{\pi * i}{N} \right) \right\} \right]$$

$$7: \text{Return } \left(\frac{T_0}{2} \right) * \left[1 + R - \left\{ (1 - R) * \tanh\left(\frac{10i}{N} - 5 \right) \right\} \right]$$

$$8: \text{Return } T_0 * \left[(1 - R) * \text{sech}\left(\frac{10i}{N} \right) \right]$$

$$9: A = \left(\frac{i}{N} \right)^2$$

$$\text{Return } T_0 * R^A$$

$$10: T = \text{CustomizedCoolingSchedule}(i, N, R, T_0)$$

If $T < T_0$ then

Return T

Return $\text{coolingSchedule}(2, i, N, R, T_0)$

3.7. Customized-Cooling-Schedule Procedure

Inputs: Current Iteration Index “ i ” Number of Cooling Steps “ N ”

Total cooling Rate “ R ”Initial Temperature “ T_0 ”*Precondition:*“code” corresponds to one of the schedules of **Table 1***Outputs:*Current Temperature “ T ”*Algorithm:*

The implementation of this algorithm is left for the algorithm or context inheriting from this generic implementation.

3.8. Object-Oriented Procedure

The class diagram of our generic algorithm seems simple, as shown in **Figure 2**, because the genericity itself lies in the implementation of the different methods of the main component used in the algorithm.

The base “SAProblem” component represents the environment offering the possible values and functionalities that the main algorithm’s steps will be relying on to execute generic global and specific tasks.

The second class (in the middle), is the class representation that will be holding the values of the encoded solution.

The third component is the main algorithm offering both flexibility and execution tracing, by saving into current instance variables and some inherited properties a specific set of values per iteration while running.

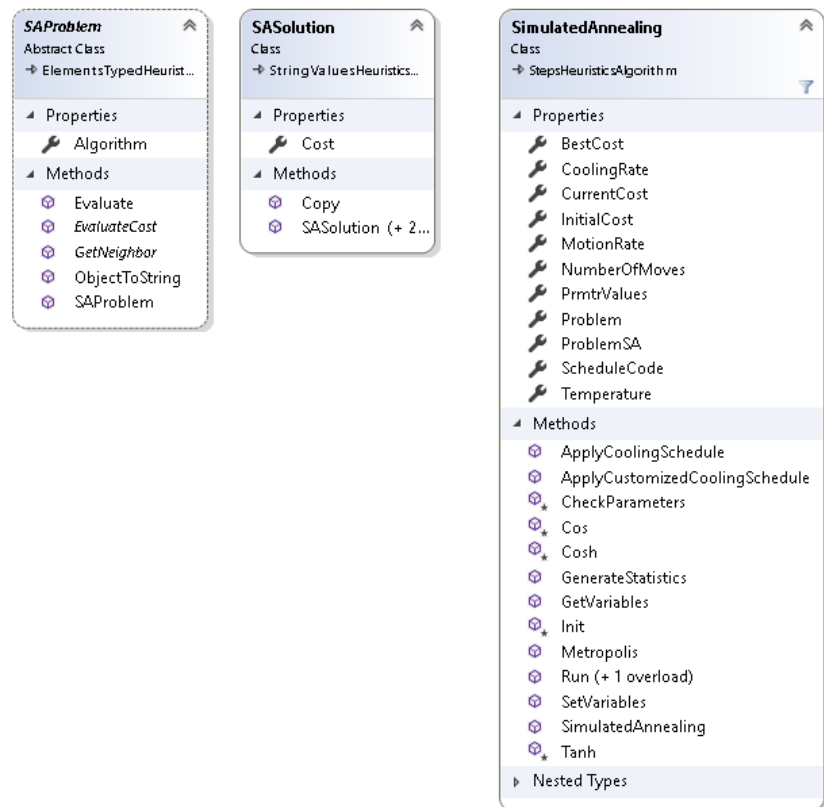


Figure 2. Suggested SA components class diagram.

Since we are talking about a generic implementation of the algorithm and its components, we implemented a few problems, as shown in **Figure 3** and **Figure 4**, which follow the guidelines set by the components diagrams of **Figure 2**; but also by the suggested encoding in reference paper 1, trying to cover different complexities. This set is divided into 3 groups relating all to a base class holding the math formulas: the first group is working with {2, 8, 10 and 16} radix applied to the default “Formula01” functionality; while the second group is a collection of problems handling different math formulas. The last group is the implementation of the encoding suggested in reference paper 1 about phylogenies.

In order to illustrate the generic implementations applied on these problems, we have developed a quick straight forward windows application, in which we dynamically modified the parameters of a default algorithm based on the suggested generic implementation, instead of writing many versions inheriting for our main implementation.

The first step, **Figure 5**, is to specify which problem the algorithm will work on. This list of problems is dynamically built from the set of problems included in the package, not only the one discussed above.

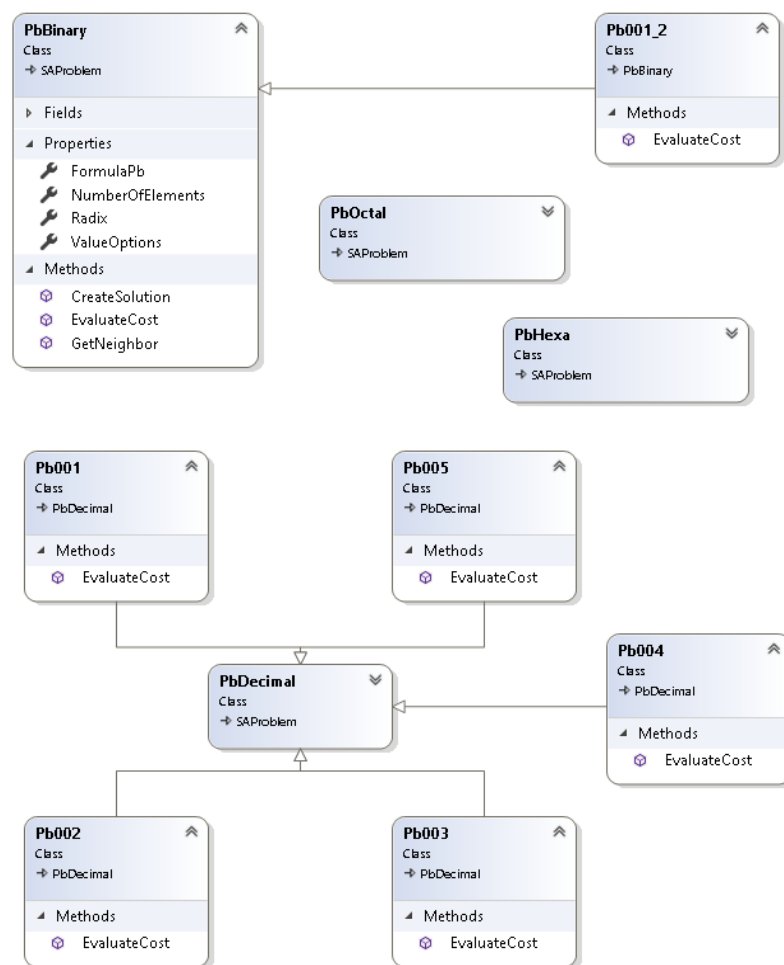


Figure 3. Some supporting problem implementations.

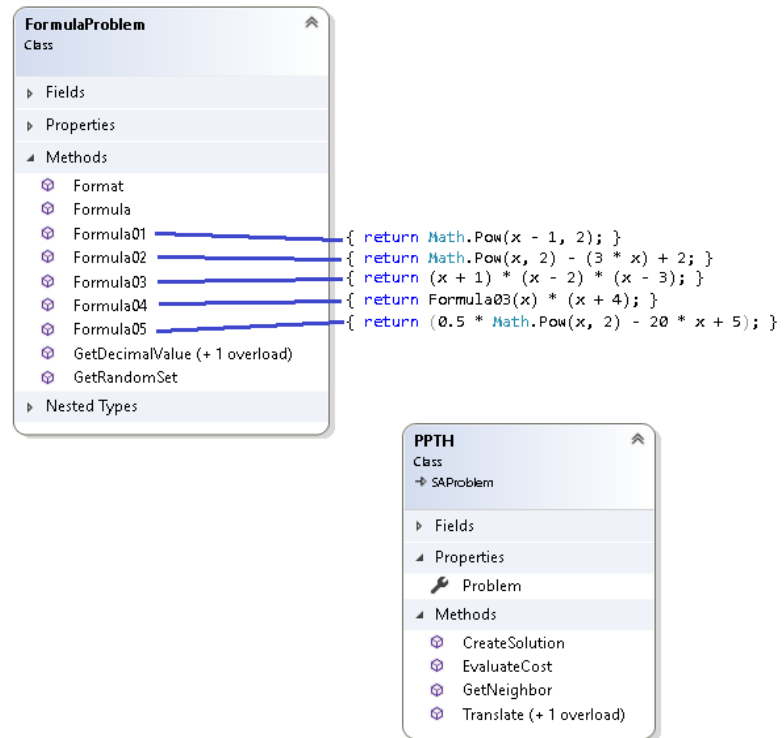


Figure 4. Utility class & parsimony problem implementation.

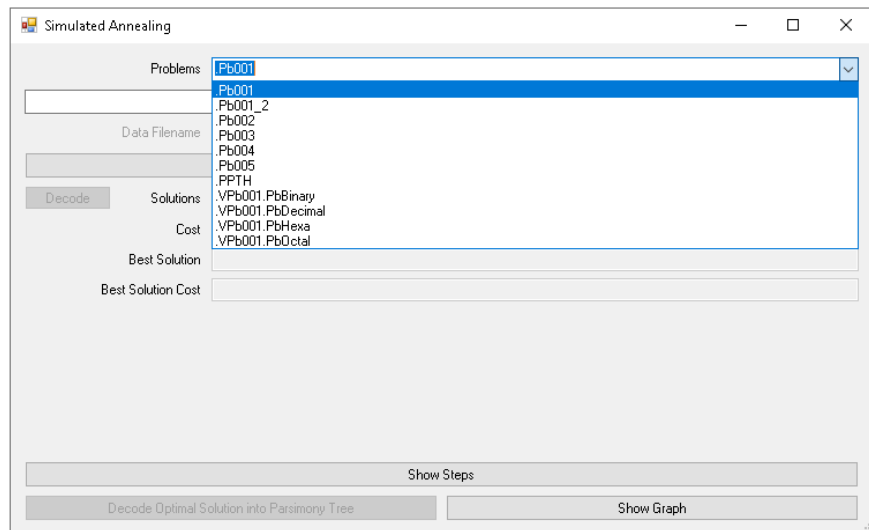


Figure 5. Win App, SA execution—problem selection.

The second step, **Figure 6**, would be to specify some of the properties required by the algorithm. The problem itself has no parameters, since the real action lies in the algorithm's method. Even though default values were given to those parameters, we advise to modify them to be able to view effective results, and/or test different cooling schedules.

The third and final step, as shown in **Figure 7**, is to simply run the algorithm. In the above screenshot, we are showing the generated result of an execution on

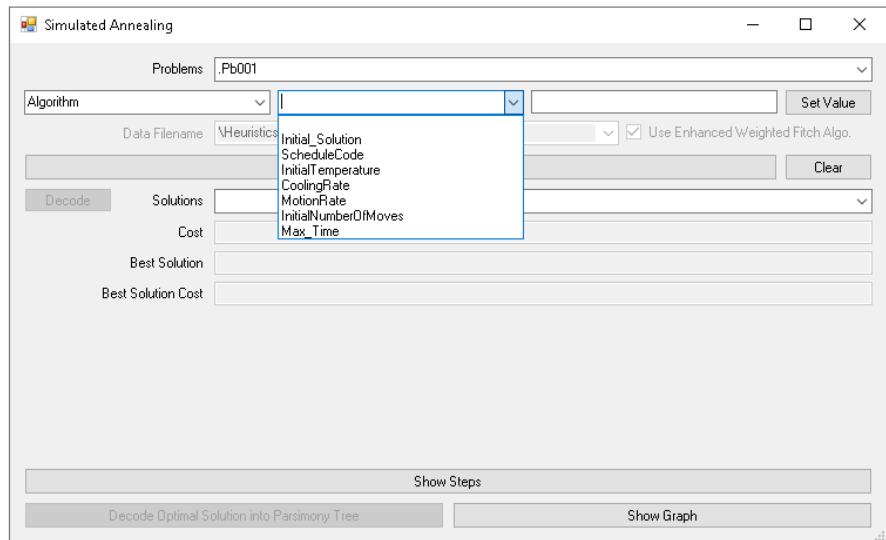


Figure 6. Win App, SA execution—parameterization.

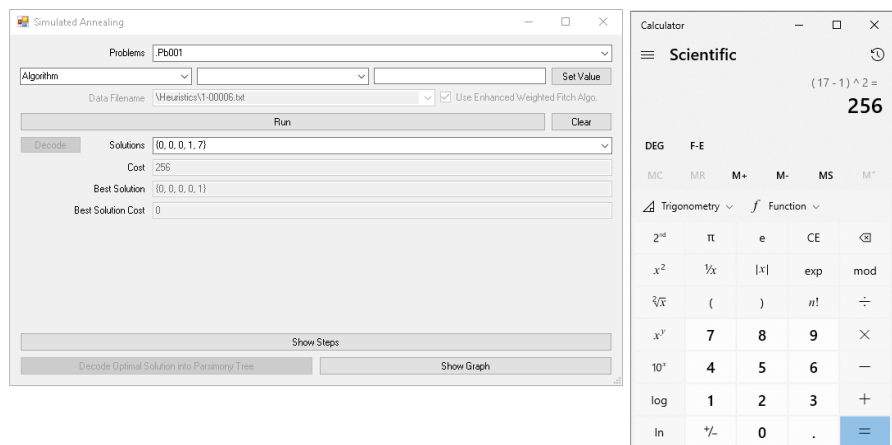


Figure 7. Win App, SA execution—sample run.

problem Pb001, along with the actual value corresponding to the last current solution compared to the cost evaluated during the last iteration of the algorithm. The next section of the screen shows the optimal solution it was able to detect with a cost of 0, which is mathematically the actual best cost. For more complex problems, running the algorithm for more time will most probably change the best cost value converging towards a more suitable optimum.

Tracing back the execution of the execution mentioned we can see in **Figure 8** that the algorithm reached 80377 as the best solution at the end of the initial step.

As illustrated in **Figure 9**, while looking in more depth at the changes and switching between current and best solutions over iterations of the main and of the metropolis procedures, we have to highlight the fact that different cooling schedule will yield arrival to the optimum at various steps. Thus allowing us to tune and select a more suitable cooling schedule for later runs.

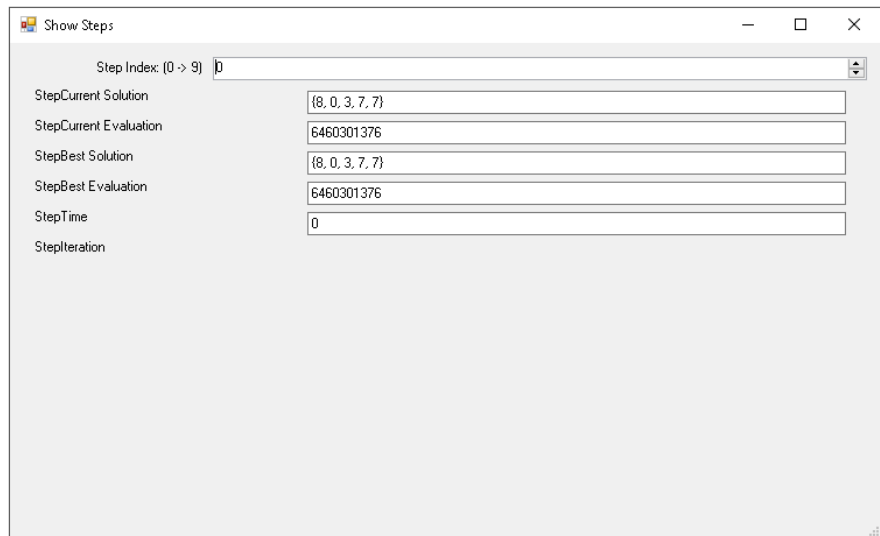


Figure 8. SA tracing functionality—initial iteration.

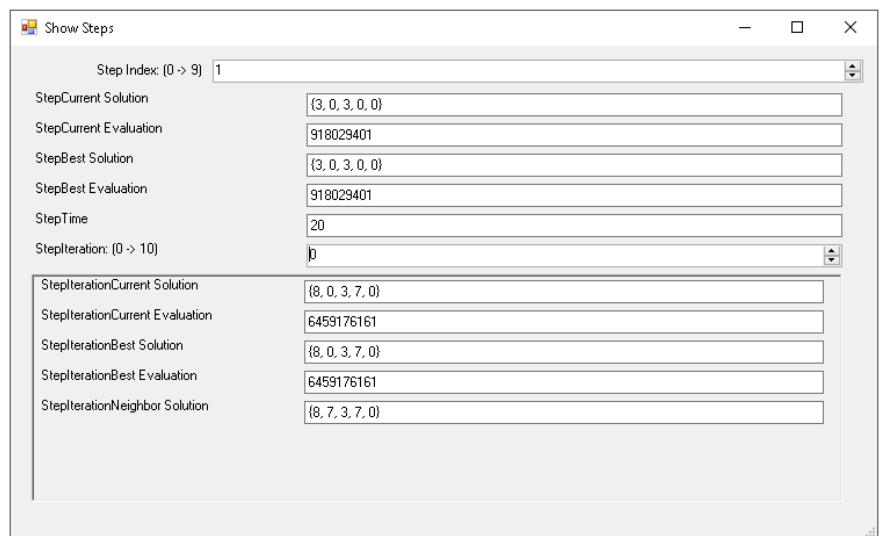


Figure 9. SA tracing functionality—internal iteration.

4. Conclusions

This research can be developed by trying to design and implement solutions to the below issues raised during our research, and supported by references [16] [17] [18] [19] [20] but also [21] [22] [23]:

- Fixed Input Parameters:
 - Even with a parameters tuning {initial temperature T_0 , motion and cooling rates and initial number of jumps M_0 , among others}, we still end up with fixed values for the algorithm's inputs.
- Randomness doesn't handle potentially repetitions:
 - In Neighbor method
- Initial solution may be far from the optimal one, thus the algorithm will take more time.

- Different cooling schedules might give better results than others at times and less good at other times; thus we will need to have a comprehensive and flexible schedule, or a functionality allowing us to run different schedules at different iterations.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

- [1] Kallab, C. (2013) Generic Encoding and Phylogenies. *Proceedings of the ICeND 2013 Conference*, Kuala Lumpur, 4-6 March 2013, 142.
- [2] Kim, J. and Warnow, T. (1999) Tutorial on Phylogenetic Tree Estimation. <https://www.semanticscholar.org/paper/Tutorial-on-Phylogenetic-Tree-Estimation-Kim/aa81aac8c1e762196e36ae8169fe20980f294fd1>
- [3] Moret, B., Bader, D. and Warnow, T. (2002) High-Performance Algorithm Engineering for Computational Phylogenetics. *Journal of Supercomputing*, **22**, 99-111. <https://www.mendeley.com/catalogue/f94b325a-d512-34d1-8335-b735011b1cf0>
- [4] Opper, D. Parsimony Phylogenetic Trees. <http://www.icp.ucl.ac.be/~opperd/private/parsimony.html>
- [5] Stamatakis, A., Ott, M. and Ludwig, T. (2005) RAXML-OMP: An Efficient Program for Phylogenetic Inference on SMPs. *International Conference on Parallel Computing Technologies*, Krasnoyarsk, 5-9 September 2005, 288-302. https://www.researchgate.net/publication/221185791_RAXML-OMP_an_efficient_program_for_phylogenetic_inference_on_SMPs https://doi.org/10.1007/11535294_25
- [6] Thinkquest 2001: International Internet Challenge. Genetic Engineering, the Creation Website. <https://link.springer.com/article/10.1023/A:1013193211174>
- [7] Felsenstein, J. (1982) Numerical Methods for Inferring Evolutionary Trees. *The Quarterly Review of Biology*, **57**, 379-404. <https://doi.org/10.1086/412935>
- [8] Fitch, W. (1971) Toward Defining the Course of Evolution: Minimum Change for a Specified Tree Topology. *Systematic Zoology*, **20**, 406-416. <https://doi.org/10.2307/2412116>
- [9] Hendy, M.D. and Penny, D. (1982) Branch and Bound Algorithms to Determine Minimal Evolutionary Trees. *Mathematical Biosciences*, **59**, 277-290. <https://www.sciencedirect.com/science/article/abs/pii/002555648290027X> [https://doi.org/10.1016/0025-5564\(82\)90027-X](https://doi.org/10.1016/0025-5564(82)90027-X)
- [10] Aarts, E.H.L. (1989) Simulated Annealing: An Introduction. *Statistica Neerlandica*, **43**, 31-52. <https://doi.org/10.1111/j.1467-9574.1989.tb01245.x>
- [11] Davis, L. (1987) Genetic Algorithms and Simulated Annealing.
- [12] Ingber, L. (1993) Simulated Annealing: Practice versus Theory. *Mathematical and Computer Modelling*, **18**, 29-57. [https://doi.org/10.1016/0895-7177\(93\)90204-C](https://doi.org/10.1016/0895-7177(93)90204-C)
- [13] Locatelli, M. (2000) Simulated Annealing Algorithms for Continuous Global Optimization: Convergence Conditions. *Journal of Optimization Theory and Applications*, **104**, 121-133. <https://doi.org/10.1023/A:1004680806815>
- [14] Pandey, A., Banerjee, S. and Sahoo, G. (2014) Applications of Meta Heuristic Search

- Algorithms in Software Testing: An Investigation into Recent Trends. *Advances in Computer Science and Information Technology*, **1**, 61-64.
- [15] Affenzeller, M. and Mayrhofer, R. (2004) Generic Heuristics for Combinatorial Optimization Problems. https://www.researchgate.net/publication/2902641_Generic_Heuristics_for_Combinatorial_Optimization_Problems
- [16] Sharman, K.C. (1988) Maximum Likelihood Parameter Estimation by Simulated Annealing. *ICASSP-88, International Conference on Acoustics, Speech, and Signal Processing*, Volume 1, 2741-2744.
- [17] Fleischer, M. (1995) Simulated Annealing: Past, Present, and Future. *Winter Simulation Conference Proceedings*, Arlington, 3-6 December 1995.
- [18] Yao, X. and Li, G. (1991) General Simulated Annealing. *Journal of Computer Science and Technology*, **6**, 329-338. <https://doi.org/10.1007/BF02948392>
- [19] Suppakitnarm, A., Seffen, K.A., Parks, G.T. and Clarkson, P.J. (2007) A Simulated Annealing Algorithm for Multiobjective Optimization. *Engineering Optimization*, **33**, 59-85. <https://doi.org/10.1080/03052150008940911>
- [20] Kim, J., Pramanik, S. and Chung, M.J. (1994) Multiple Sequence Alignment Using Simulated Annealing. *Bioinformatics*, **10**, 419-426. <https://doi.org/10.1093/bioinformatics/10.4.419>
- [21] Yang, W.B. and Wang, Y.D. (2012) Improved Simulated Annealing Algorithm for GTSP. *International Conference on Automatic Control and Artificial Intelligence (ACAI)*, Xiamen, 3-5 March 2012, 1202-1205.
- [22] Khairuddin, R. and Zainuddin, Z.M. (2019) A Comparison of Simulated Annealing Cooling Strategies for Redesigning a Warehouse Network Problem. *Journal of Physics Conference Series*, **1366**, Article ID: 012078. <https://doi.org/10.1088/1742-6596/1366/1/012078>
- [23] Peprah, A.K., Appiah, S.K. and Amponsah, S.K. (2017) An Optimal Cooling Schedule Using a Simulated Annealing Based Approach. *Applied Mathematics*, **8**, 1195-1210. <https://doi.org/10.4236/am.2017.88090>