Computer Physics Communications 223 (2018) 28–33

Contents lists available at ScienceDirect

Computer Physics Communications

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

Conformational Space Annealing explained: A general optimization algorithm, with diverse applications

InSuk Joung [a], Jong Yun Kim [a], Steven P. Gross [b,c], Keehyoung Joo [a,d], Jooyoung Lee [a,b,*]

[a] Center for In Silico Protein Science, Korea Institute for Advanced Study, 02455, Republic of Korea
[b] School of Computational Sciences, Korea Institute for Advanced Study, 02455, Republic of Korea
[c] Department of Developmental and Cell Biology, UC Irvine, Irvine CA, 92697, USA
[d] Center for Advanced Computation, Korea Institute for Advanced Study, 02455, Republic of Korea

ARTICLE INFO

Article history:
Received 12 May 2017
Received in revised form 27 September 2017
Accepted 29 September 2017
Available online 23 October 2017

Keywords:
Conformational space annealing
Parameter optimization problem
Global optimization
Combinatorial optimization
Genetic algorithm

ABSTRACT

Many problems in science and engineering can be formulated as optimization problems. One way to solve these problems is to develop tailored problem-specific approaches. As such development is challenging, an alternative is to develop good generally-applicable algorithms. Such algorithms are easy to apply, typically function robustly, and reduce development time. Here we provide a description for one such algorithm called Conformational Space Annealing (CSA) along with its python version, PyCSA. We previously applied it to many optimization problems including protein structure prediction and graph community detection. To demonstrate its utility, we have applied PyCSA to two continuous test functions, namely Ackley and Eggholder functions. In addition, in order to provide complete generality of PyCSA to any types of an objective function, we demonstrate the way PyCSA can be applied to a discrete objective function, namely a parameter optimization problem. Based on the benchmarking results of the three problems, the performance of CSA is shown to be better than or similar to the most popular optimization method, simulated annealing. For continuous objective functions, we found that, L-BFGS-B was the best performing local optimization method, while for a discrete objective function Nelder–Mead was the best. The current version of PyCSA can be run in parallel at the coarse grained level by calculating multiple independent local optimizations separately. The source code of PyCSA is available from http://lee.kias.re.kr.

1. Introduction

Many important problems can be formulated in terms of an optimization problem. As computers are used to solve more and increasingly challenging problems, improved general global optimization methods are needed to carry out such calculation. The most popular and general optimization method is simulated annealing (SA). However, many other more-complicated methods are also used, including the genetic algorithm (GA) [1], particle swarm optimization (PSO) [2], basin-hopping [3], differential evolution [4], simulated tempering [5], Hamiltonian exchange [6,7], multicanonical sampling [8], and entropic sampling [9]. Here we describe a general yet powerful global optimization method, Conformational space Annealing (CSA). We have been using CSA for solving various hard global optimization problems; for all cases studied so far, CSA has been found to be superior to other methods (see for example [10–18]).

As an optimization method CSA is as general as SA, and CSA's details have been described previously, but the lack of a modular form of the CSA source code has made implementation of CSA to a user-defined problem challenging, thus limiting its popularity. Here, in addition to demonstrating its utility by showing its success on various problems, we provide python (PyCSA) subroutines so that CSA can be easily implemented to user-defined problems.

CSA is a kind of GA, but is conceptually related to SA with key differences discussed below. To apply CSA to an optimization problem, three ingredients must be prepared: a distance measure between two solutions, a local minimization routine, and a crossover procedure to combine two parent solutions to generate daughter solutions.

We start by applying PyCSA to obtain the global minimum solution of two mathematical functions with numerous local minima, allowing us to better describe the method. We find that CSA performs better than SA, or more precisely Monte Carlo with minimization (MCM) [19], a variant of SA, when the energy landscape is of random shape. The performance is similar for a function with

* Corresponding author at: Center for In Silico Protein Science, Korea Institute for Advanced Study, 02455, Republic of Korea.
E-mail address: jlee@kias.re.kr (J. Lee).
URL: http://lee.kias.re.kr (J. Lee).

https://doi.org/10.1016/j.cpc.2017.09.028
0010-4655/

hierarchical energy landscape. The second example of CSA application discussed in this paper is a parameter optimization problem dealing with a given set of decoys, where the goal is to construct the energy landscape by varying parameters so that a desired outcome is achieved. This problem is also known as training.

When there is significant computational complexity, heuristic algorithms are often used instead of exact methods to obtain 'sufficiently good' – or even the best – solutions rapidly. While CSA is a heuristic method – and thus in principle does not *guarantee* the optimality of its solutions – in practice we find that it works so well that it frequently finds optimal solutions.

When equipped with a heuristic method, one should be concerned with two issues: first, given limited computational resources/time, will the method generate a reasonably good solution? Second, given sufficient time, will the method ultimately obtain the globally optimal solution, and if so, how would the time required for this scale in terms of the problem size? If not, how close is the solution it finds to the globally optimal one? Importantly, the immediate goal of CSA is not to obtain the absolute optimal solution possible, but rather, to obtain many diverse good solutions rapidly. Because of this, the quality of the CSA solution at early stages can be far from ideal. However, as the CSA procedure continues, the globally optimal solution is often obtained as a by-product, leading to superior performance in terms of time complexity relative to other general methods.

Below, we discuss CSA, first in the most general case, and then provide examples of how to apply it to an optimization problem.

2. The CSA algorithm

2.1. Overview of CSA

Global optimization is to find the optimal solution $\boldsymbol{x}^*$ for a given objective function $f(\boldsymbol{x})$ satisfying $f(\boldsymbol{x}) \geq f(\boldsymbol{x}^*)$ for any $\boldsymbol{x}$. CSA can be viewed as a kind of GA since it maintains a conformational population while performing genetic operations. Here, we use the terminologies of "conformation" and "solution" interchangeably. CSA contains two essential ingredients of MCM [19] and SA. As in MCM, CSA deals with only locally minimized conformations. In SA, annealing is performed in terms of temperature, while in CSA, annealing is performed in an abstract conformational space while the diversity of the population is actively controlled.

One of the key features of CSA is that it controls the diversity of its population by (1) defining a distance measure and setting the criterion of the similarity between two conformations as D_{cut}, and by (2) using D_{cut} to control the diversity of the CSA population, while D_{cut} is slowly reduced from an initial value to a final value; hence the name of the method. The way these initial and final values are set and annealed is performed in a manner independent of the problem under investigation.

2.2. Details of CSA

The flowchart of CSA is shown in Fig. 1. In CSA, two populations are maintained, *first bank* and *bank*. The *first bank* contains solutions that are randomly generated and subsequently minimized. It is built at the beginning of CSA and remains unaltered. The *bank* holds the evolving population, and initially it is identical to the *first bank*. We set the size of the *bank* (and *first bank*) initially as N_B and it can be increased by N_B whenever the CSA search is diagnosed as reached no or little change. The optimal value of N_B is problem specific and increase the bank size by N_B at a time. In this research, $N_B = 15$.

Daughter solutions are generated by first selecting N_S (40%–60% of N_B) seeds from *bank* and performing crossover and mutation operations on them. In this work, we set $N_S = 6$. The crossover

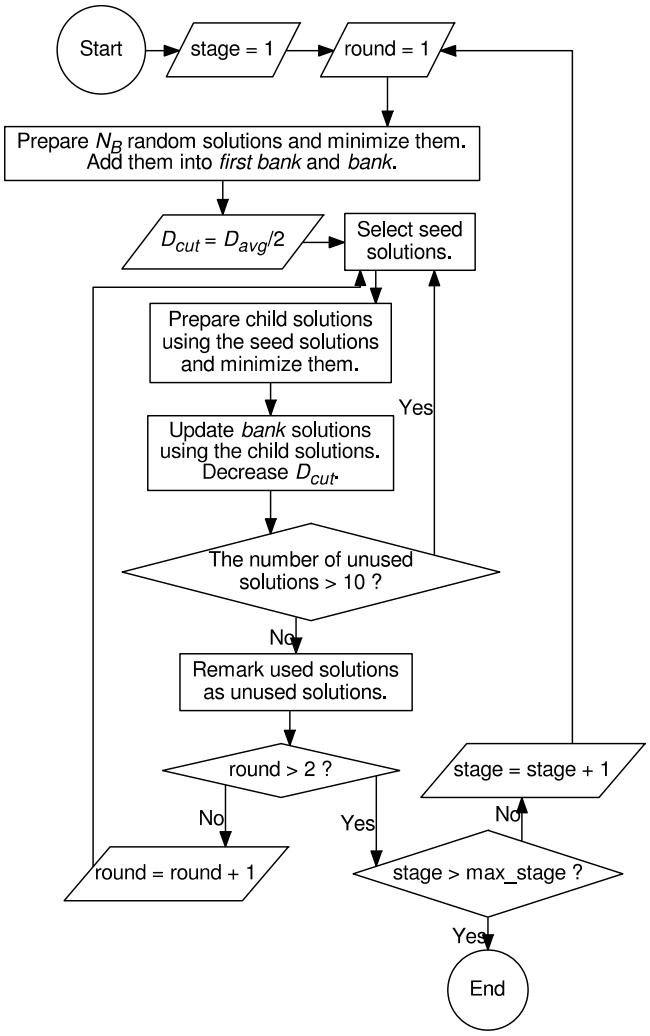

Fig. 1. CSA flow chart.

partners of a seed are selected from either *bank* or *first bank*. Each daughter solution is minimized and used to update *bank* one at a time. The size of *bank* is kept fixed during this update as a daughter solution either replaces a *bank* solution or is discarded. After the *bank* update is finished, the number of unused solutions as seeds in bank is counted. If it is less than a preset number (11 in this research), one round of CSA iteration is completed, and all *bank* solutions are marked as unused seed solutions and we repeat two additional rounds of CSA iteration to complete one CSA stage. When a CSA stage is finished, we increase the *bank* size as well as the first *bank* size by adding N_B randomly-generated and subsequently-minimized solutions to both *banks*.

2.3. CSA methods

2.3.1. D_{cut} control

At the beginning of each CSA stage, D_{cut} is set as $D_{cut} = D_{avg}/2$ where D_{avg} is the average distance among the *first bank* solutions. After each CSA iteration, D_{cut} is reduced by multiplying it with $R_D = 0.983912$, so that D_{cut} becomes $D_{low} = D_{avg}/5$ after 25 iterations. D_{low} is kept constant afterwards. D_{cut} is set back to $D_{avg}/2$ whenever a new CSA stage begins. We used D_{low} between $D_{avg}/3$ and $D_{avg}/5$. The optimal values of R_D, D_{low} etc., which control the annealing speed of CSA as done by temperature variation in SA, can be problem dependent, but we observe that the performance of

CSA is rather robust. Slower annealing is recommended for more of robust results but at the expense of computational time.

2.3.2. Seed selection

At each CSA iteration, N_S seeds are selected from *bank*. If the number of unused solutions, $N_{unused} \geq N_S$, seeds are selected from unused solutions as follows. One is selected randomly, and then distances are measured between the selected seed and non-selected unused solutions. Among those not closer than the average distance, the best solution is selected as the next seed. By repeating the process of measuring distances between selected seeds and non-selected solutions and selecting the best among not closer than the average distance, the remaining seeds are selected. If $N_{unused} < N_S$, all of N_{unused} are selected and the remaining are selected considering used *bank* solutions applying the same criteria described above. All seeds are marked as used solutions in *bank*. This process of selecting diverse and best-score seeds is to select diverse and more fit daughter solutions.

In CSA, the conformational space near a seed is more extensively searched than the rest of the space since all daughter conformations get the majority of their initial variable values from a seed. When adding N_B conformations to *bank* and *first bank* at the beginning of a new CSA stage, the newly added N_B *bank* conformations are in uphill competition against existing already-settled and low-energy *bank* conformations. To take care of this disadvantage, existing *bank* conformations are not selected as seeds at the first round of CSA. In addition, at the first round, partner conformations for crossover are selected only among the newly added. This is to allow the newly added to evolve to new low-energy basins competing only among daughter solutions perturbed from the newly added. These two restrictions are lifted in the subsequent rounds.

2.3.3. Crossover and mutation methods

For each seed, 30 daughter conformations are generated via crossovers and mutations. In CSA, a daughter solution always takes more information from a seed than from the seed's crossover partner. For this reason, a daughter solution is considered as a perturbed conformation of its seed. The partner solutions may come from either *first bank* or *bank*.

The way crossover is performed using two solutions generally depends on the structure of the variable/solution space, and we provide a guideline here. If the variable space has no particular structure, one can randomly mix two sets of solutions taking more from the seed and less from its partner. If the variable can be represented by a vector, two-point crossover can be performed. Examples include dihedral angle representation of a chain molecule such as proteins or nucleic acids. The idea of the two-point crossover can be generalized to cut a "continuous" fragment from a solution and paste it to overwrite the corresponding part to another solution. Here, the fragment continuity may refer to the topological continuity in the solution space, or spatial continuity, and it is rather problem dependent. If possible, we recommend the user to consider multiple crossover strategies including random, topological and spatial crossovers. Later, we provide examples of these crossover methods, which can serve as guidelines of devising crossovers for user-specific problems.

Random mutation is another popular genetic operator, where a small number of randomly selected variables are perturbed in a random fashion. A good rule of thumb is to design a mutation method where up to about five variables are mutated and the acceptance ratio of the mutation should be about 5% on average.

We used the following crossover/mutation methods to generate daughters:

- Crossover between a seed and a bank conformation: For a seed, its partner is selected from bank in a random fashion (excluding the seed itself). First, we copy the seed as a daughter and then replace some of the daughter's variables with the corresponding ones from the partner to perform crossover. The size of crossover is set in a random fashion between 1 and half of the total number of variables.
- Crossover between a seed and a first bank conformation: The same procedure as above is performed but the partner of a seed is selected from first bank. The size of crossover is set between 1% and 20% of the total number of variables.
- Mutation of a seed: 1, 2 or 3 variables of a seed are mutated within 20% of the variable range from their current values in a random fashion.

For a seed, each of the above three methods are used 10 times. Therefore, with $N_S = 6$, a total of 180 ($= 6 \times (10 + 10 + 10)$) daughters are generated, each of which are subsequently energy minimized using a local minimizer (see next section). We note that, as discussed above, during the first CSA round of each CSA stage, the partner solutions are selected only from the last N_B solutions in *bank* that are newly added.

2.3.4. Local minimization methods

Since all solutions are locally energy minimized in CSA, we need a local minimizer, which can be problem specific. We used L-BFGS-B [20–22], Powell [23], TNC [24], sequential least squares programming (SLSQP) and Nelder–Mead [25] algorithms, which are implemented in Python's scipy module (version 0.12.0). The default setting of each minimizer was used. Note that, intentionally, we did not use any analytic forms of functional derivatives for local minimization because, in a general case, analytic forms of functional derivatives may not be available.

When using the Nelder–Mead method, we constructed the initial vertices of a simplex by perturbing a given solution up to 10% of the variable range. For example, if the value of a given variable is 1.0 and its range is $(-10.0, 10.0)$, a new variable for a vertex is randomly chosen in the range of $(-1.0, 3.0) = (1.0–2.0, 1.0+2.0)$. The termination of local minimization is controlled by the number of maximum iterations we set or the default tolerance values. The number of maximum iterations was set to 1000.

2.3.5. Bank update

All daughter solutions are used to update *bank* one at a time as follows. For each daughter solution α, its distances to all the *bank* solutions are measured to identify its closest solution a. If the distance between them, $D(\alpha, a)$, is less than or equal to D_{cut} and α is more optimal than a, α replaces a. If $D(\alpha, a) > D_{cut}$ and α is more optimal than the worst solution b in *bank*, α replaces b. Otherwise, α is discarded.

3. Applying CSA

3.1. Multidimensional test functions

Various test functions in analytic form are widely used to test the performance of global optimization algorithms [26]. Here, we used two: the Ackley function and the Eggholder function (see Fig. 2 for their 2D versions). The energy landscape of Ackley is hierarchical while that of Eggholder is close to a random surface. Ackley is hierarchical because its functional form is a sum of a high frequency low-barrier surface and a smooth convex-down no-barrier surface. Both functions are multidimensional and have many local minima. In both cases, the variable can be expressed as $\boldsymbol{x} = (x_1, x_2, \ldots, x_D)$, where D represents the dimension of the function.

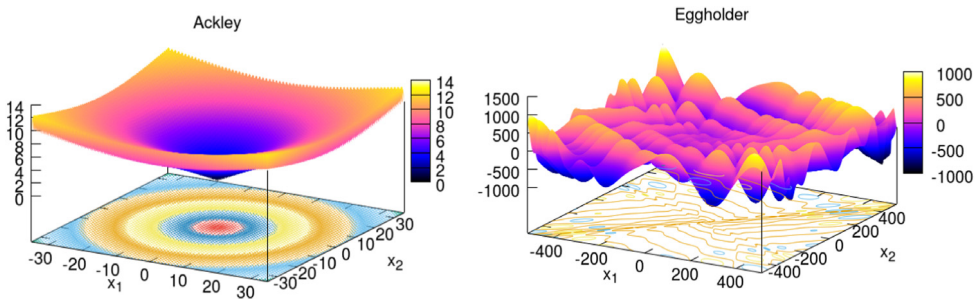

Fig. 2. Energy landscapes of the Ackley function (left) and the Eggholder function (right) are shown for $D = 2$.

The D-dimensional Ackley function [27] is defined as

$$f_{Ackley}(\boldsymbol{x}) = -20\exp\left(-0.02\sqrt{D^{-1}\sum_{i=1}^{D}x_i^2}\right) - \exp\left(D^{-1}\sum_{i=1}^{D}\cos(2\pi x_i)\right) + 20 + e.$$

The energy landscape of Ackley is characterized by a funnel-like envelop where its global minimum is located at $\boldsymbol{x}^* = (0, \ldots, 0)$ with the function value of zero. The range of all variables are subjected to $-35 \leq x_i \leq 35$ in this test. Four separate dimensions ($D = 2$, 10, 20, and 50) were tested. In all cases, the global optimality is assumed when the function value drops within 1×10^{-4}.

The D-dimensional Eggholder function is defined as

$$f_{Eggholder}(\boldsymbol{x}) = \sum_{i=1}^{D-1}\left[-(x_{i+1}+47)\sin\sqrt{|0.5x_i + x_{i+1} + 47|} - x_i\sin\sqrt{|x_i - x_{i+1} - 47|}\right].$$

The energy landscape of Eggholder is more or less random and the optimal solution is reported only for $D = 2$ with $f(\boldsymbol{x}^*) \approx -959.64$ and $\boldsymbol{x}^* \approx (512, 404.2318)$. Optimal solutions for $D > 2$ have not been yet reported. The range of all variables are subjected to $-512 \leq x_i \leq 512$ in this test. Four separate dimensions ($D = 2, 5, 10$, and 20) were tested in this study, and the optimal solutions found in this study are shown in Table 1. The global optimality of each D is assumed when the function value drops below a preset value consistent in five significant digits with the corresponding number in Table 1 (-959.64 ($D = 2$), -3719.7 ($D = 5$), -8291.2 ($D = 10$) and $-17\,455$ ($D = 20$)). Five independent CSA runs were carried out for each D. The global minima of the Eggholder function were confirmed by the five separate runs, each finding the identical minimum and no lower function values were obtained.

3.1.1. The comparison with SA

Initially, we intended to compare the performance of CSA to that of SA. However, SA was found to underperform compared to MCM, a variant of SA, that combines SA with local minimization. Therefore, we decided to use MCM for the performance comparison. The description of the MCM implementation is provided in Supplementary Sections 1 and 2. For the performance evaluation, we used the number of function evaluations required to find the optimal solution. The average number of function evaluation (N_{eval}) is plotted in Fig. 3. For the Ackley function, where the energy landscape is relatively simple, CSA and MCM performed similarly. However, for the Eggholder function, where the energy landscape is much more complex, a significant performance difference was observed as shown in Fig. 3. For more difficult cases of $D = 5$, 10 and 20, CSA outperformed MCM with a large difference in N_{eval}. For

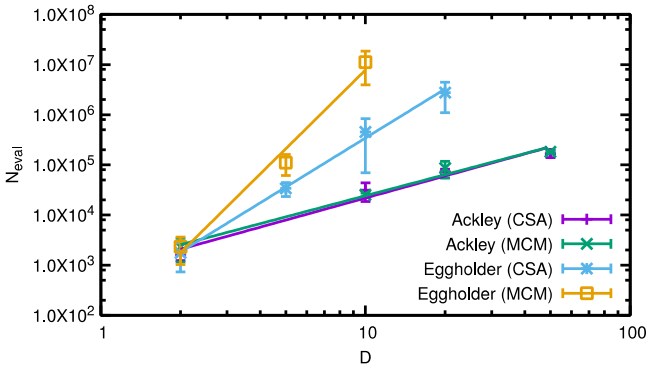

Fig. 3. The relative performance of CSA and MCM is estimated in terms of the average number of function evaluation (N_{eval}) required to obtain the optimal solution.

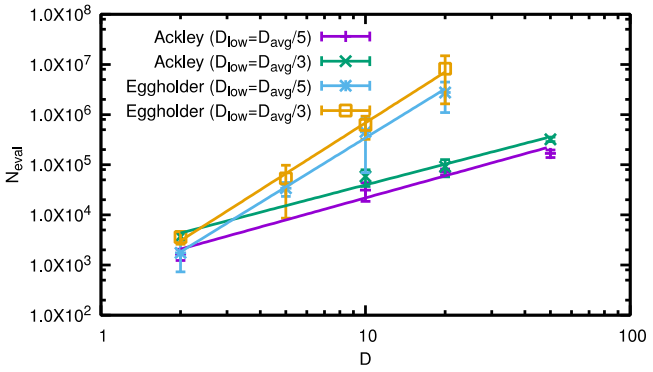

Fig. 4. The effect of D_{low} on N_{eval}. For local minimization L-BFGS-B was used.

$D = 20$, none of the five MCM runs succeeded in finding the global minimum solution even after 1×10^8 function evaluations. Based on these experiments, we conclude that CSA performs better than MCM for conformational searching of complex energy landscape. For local minimization, L-BFGS-B was used for both CSA and MCM. D_{low} was set to $D_{avg}/5$ in CSA.

3.1.2. The effect of D_{low}

As discussed in the section of D_{cut} control, the default value of D_{low} is set as $D_{low} = D_{avg}/5$ in CSA. A smaller or larger value of D_{low} is expected to lead to a faster or more robust convergence of a CSA run. We tested the case of $D_{low} = D_{avg}/3$ and the results are shown in Fig. 4. The regression lines show the relationship between the function dimension, D, and D_{low}. The result using $D_{low} = D_{avg}/5$ is consistently better than that using $D_{low} = D_{avg}/3$ in terms of N_{eval}. We note that the slopes of the regression lines appear similar to each other for each function.

Table 1
Optimal solutions of the Eggholder function are listed.

D	approx. $\boldsymbol{x}^*$	approx. $f(\boldsymbol{x}^*)$
2	(512.0000, 404.2318)	−959.641
5	(485.590, 436.124, 451.083, 466.431, 421.959)	−3719.72
10	(480.85, 431.37, 444.91, 457.55, 471.96, 427.50, 442.09, 455.12, 469.43, 424.94)	−8291.24
20	(481.0, 431.5, 445.1, 457.9, 472.4, 428.0, 443.0, 456.5, 471.7, 427.3, 442.5, 456.3, 471.6, 427.1, 442.5, 456.3, 471.7, 427.3, 442.8, 456.9)	−17 455.9

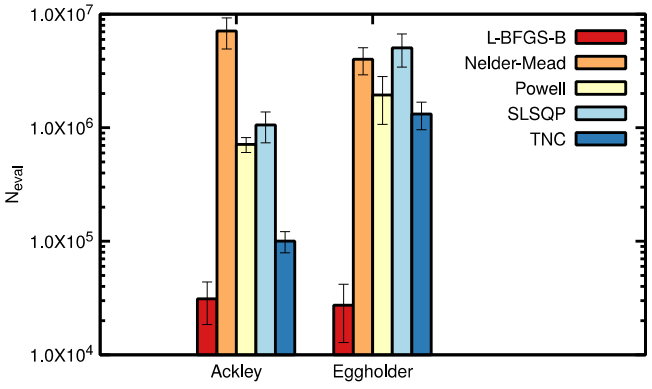

Fig. 5. The effect of local minimization method on N_{eval} is shown. $D = 10$ is used for Ackley function and $D = 5$ for the Eggholder function. In all cases, we used $D_{low} = D_{avg}/5$.

3.1.3. The effect of local minimization method

We tested the effect of using various local minimization methods. In Fig. 5, we observe that L-BFGS-B performed best with the least value of N_{eval} for both functions. For benchmarking, $D = 10$ for Ackley function and $D = 5$ for the Eggholder function were used in Fig. 5.

3.1.4. Details of CSA implementation

In terms of the python programming, PyCSA class (csa) needs to be properly inherited in the user provided subroutines. A user should provide three problem-specific subroutines, minimizef, perturbvarf, and comparef, respectively for minimizing a given solution, performing perturbation of a seed by genetic operations, and measuring the distance between two given solutions. For the Eggholder function, the three subroutines are provided in Supplementary Section 3. For local minimization, we used L-BFGS-B adopted from the python's scipy module. For perturbation, two crossover operators and a random mutation operator were used. Finally, in addition, the user should provide a subroutine for generating initial random solutions (randvarf). Here, random values in the variable range are assigned. The finalized inherited class can be found in csa_example.py.

Now, the customized class should be imported by another python script and its run method should be called (see runcsa_single.py) to perform the CSA optimization. However, this simple script will run only the first CSA stage. To implement the complete protocol shown in Fig. 1, one needs to write a script implementing the flow (see runcsa.py). The run method can be performed in a highly parallel fashion at the coarse-grained level. To use multiple CPUs, the script should be combined with an MPI executor like mpirun. Users are encouraged to check the source code provided in the supplementary information, where line-by-line comments are provided.

3.2. Parameter/weight optimization of a function

In the problem of parameter optimization of a function, one wants to determine its parameters so that one can obtain more desired results. The parameter optimization problem can be considered as a training problem.

As an example of parameter optimization, we consider a protein model selection problem out of given decoy structures using an energy function. In particular, we consider the Rosetta decoy set [28] that contains 59 proteins ($N_{target} = 59$), each with 100 decoy protein models with variable structural similarities to its native structure. Various energy functions have been suggested to discriminate protein structures so that they can be used to select more native-like structures out of given decoys. They include dDFIRE [29], DOPE [30], GOAP [31], RF_HA_SRS [32], and RW-plus [33]. Using each of them, the lowest energy decoy is identified for each protein of the Rosetta decoy set [28], and the results are summarized in Table 2. Here, TM-score [34] is used to measure the similarity between a decoy and its native protein structure. TM-score ranges between 0.0 and 1.0, where 1.0 corresponds to a perfect match.

Here we consider a linear combination of the above five energy functions with weights w's, $E_{Combined} = \sum_i w_i E_i$ so that $E_{Combined}$ can be used as a better discriminator for selecting more native-like decoys. Index i refers to each energy term, and the weight of dDFIRE is set to 1.0 with remaining four weights are parameters to vary for better discrimination. Here we performed the parameter optimization using the following objective function, f_{weight}:

$$f_{weight}(\boldsymbol{w}) = -\frac{1}{N_{target}} \sum_{j}^{N_{target}} TM\,(j, \boldsymbol{w}) + \sum_{i}^{N_{param}} k(w_i - 1.0)^2.$$

Here, $N_{target} = 59$ is the number of targets, $N_{param} = 5$ is the number of weight parameters, $\boldsymbol{w}$ is the weight vector, and $TM(j, \boldsymbol{w})$ is the TM-score of the lowest $E_{Combined}$ decoy of protein j. The second term of the objective function is a regularization term keeping the weight parameters close to 1.0, and we set $k = 1.0 \times 10^{-6}$. The range of each weight parameter was set (0.01,100.0) in this example. Considering the potential scale difference among five energy terms, the search for weights was performed in a logarithmic scale.

We benchmarked the efficiency of finding the best solution using five local minimizers and two settings for D_{low}, and the results are shown in Fig. 6. For each minimizer, five CSA runs were performed. The lowest function value found is −0.529156644 and all five runs produced the same value. The optimized weights are $w_{dDFIRE} = 1.0$, $w_{DOPE} = 0.963$, $w_{GOAP} = 0.0246$, $w_{RF\_HA\_SRS} = 1.14$, and $w_{RWplus} = 0.895$, and the contribution of the second term to f_{weight} is almost negligible (only about 0.01%). The first term of f_{weight} converged relatively fast with multiple solutions that produced the identical lowest value for the first term. Many additional function evaluations were required to obtain the best solution with the lowest second term. The best result was obtained by using Nelder–Mead with $D_{low} = D_{avg}/5$, for which about 1.4×10^5 function evaluations was performed. We note that the objective function for this parameter optimization is discontinuous and gradient-based local minimization methods generally performed less efficiently than Nelder–Mead or Powell.

With the combined energy function $E_{Combined} = \sum_i w_i E_i$, the improvement of TM-score is 0.0143 compared to the best energy model, GOAP as shown in Table 2. The transferability of the combined energy function to other decoy sets not included in the

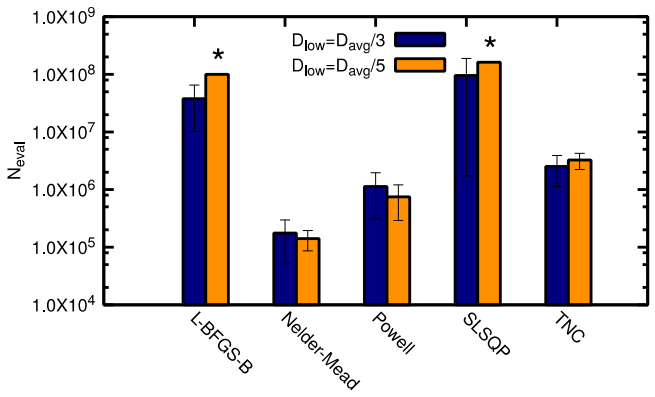

Fig. 6. The effect of the local minimization method and D_{low} on the number of function evaluations (N_{eval}) for the parameter optimization problem. Asterisk indicates that the runs were terminated prematurely.

Table 2
Average TM-score of the selected models.

Energy term	Avg. TM-score
dDFIRE	0.4782
DOPE	0.5125
GOAP	0.5149
RF_HA_SRS	0.4631
RWplus	0.5038
$E_{Combined}$	0.5292

training is beyond the scope of this research. For a practical application of parameter optimization, consideration of the transferability should take into account of a cross validation procedure.

4. Conclusion

Here, we have presented an implementation of a tool for global optimization employing conformational space annealing (CSA), designed to solve a general optimization problem. Prerequisites for applying CSA to a user-defined problem are a local minimizer, crossover/mutation methods, and a distance measure between two solutions. For the three functions tested in this work for global optimization, the Ackley function, the Eggholder function and the weight optimization function, we used readily available local minimizers, namely, L-BFGS-B, Nelder–Mead, Powell, TNC and SLSQP. For the crossover method, we used a simple multiple point crossover method, and we used the Euclidean distance to define a distance measure between two given solutions.

We demonstrated that CSA outperforms Monte Carlo with minimization, an improved variant of simulated annealing for searching global minimum solutions of rugged energy landscapes. Benchmarking showed that L-BFGS-B works well for the optimization of continuous functions. In case of the discontinuous objective function, the Nelder–Mead local minimization method performed better than L-BFGS-B.

The source code provided in this work is written in python, so that it can be easily combined with existing source codes. Parallelization of the calculation is already implemented in the provided module, therefore, if necessary, the computation can be easily accelerated using massively parallel computing. The source code together with the examples shown above are available at http://lee.kias.re.kr.

Acknowledgments

The financial support by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2008–0061987) is greatly appreciated. Authors also acknowledge Center for Advanced Computation (CAC) in the Korea Institute for Advanced Study (KIAS) for providing computing resources (Linux Cluster System).

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.cpc.2017.09.028.

References

[1] M. Mitchell, An Introduction To Genetic Algorithms, MIT press, 1998.
[2] J. Kennedy, Encycl. Mach. Learn., Springer, 2011, pp. 760–766.
[3] D.J. Wales, J.P.K. Doye, J. Phys. Chem. A 101 (1997) 5111–5116.
[4] R. Storn, K. Price, J. Global Optim. 11 (1997) 341–359.
[5] R.H. Swendsen, J.-S. Wang, Phys. Rev. Lett. 57 (1986) 2607.
[6] H. Fukunishi, O. Watanabe, S. Takada, J. Chem. Phys. 116 (2002) 9058–9067.
[7] Y. Sugita, A. Kitao, Y. Okamoto, J. Chem. Phys. 113 (2000) 6042–6051.
[8] B.A. Berg, T. Neuhaus, Phys. Rev. Lett. 68 (1992) 9.
[9] J. Lee, Phys. Rev. Lett. 71 (1993) 211.
[10] J. Lee, H.A. Scheraga, S. Rackovsky, J. Comput. Chem. 18 (1997) 1222–1232.
[11] J. Lee, A. Liwo, H.A. Scheraga, Proc. Natl. Acad. Sci. 96 (1999) 2025–2030.
[12] J. Lee, I.-H. Lee, J. Lee, Phys. Rev. Lett. 91 (2003) 80201.
[13] K. Lee, C. Czaplewski, S.-Y. Kim, J. Lee, J. Comput. Chem. 26 (2005) 78–87.
[14] K. Joo, J. Lee, I. Kim, S.J. Lee, J. Lee, Biophys. J. 95 (2008) 4813–4819.
[15] K. Joo, J. Lee, J.-H. Seo, K. Lee, B.-G. Kim, J. Lee, Proteins Struct. Funct. Bioinform. 75 (2009) 1010–1023.
[16] J. Lee, S.P. Gross, J. Lee, Phys. Rev. E 85 (2012) 56702.
[17] I.H. Lee, Y.J. Oh, S. Kim, J. Lee, K.J. Chang, Comput. Phys. Comm. 203 (2016) 110–121.
[18] J. Lee, I.-H. Lee, I. Joung, J. Lee, B.R. Brooks, Nature Commun. 8 (2017) 15443.
[19] Z. Li, H.A. Scheraga, Proc. Natl. Acad. Sci. 84 (1987) 6611–6615.
[20] R.H. Byrd, P. Lu, J. Nocedal, C. Zhu, SIAM J. Sci. Comput. 16 (1995) 1190–1208.
[21] C. Zhu, R.H. Byrd, P. Lu, J. Nocedal, ACM Trans. Math. Software 23 (1997) 550–560.
[22] J.L. Morales, J. Nocedal, ACM Trans. Math. Software 38 (2011) 7.
[23] M.J.D. Powell, Comput. J. 7 (1964) 155–162.
[24] S.G. Nash, SIAM J. Numer. Anal. 21 (1984) 770–788.
[25] J.A. Nelder, R. Mead, Comput. J. 7 (1965) 308–313.
[26] M. Jamil, X.-S. Yang, Int. J. Math. Model. Numer. Optim. 4 (2013) 150–194.
[27] T. Bäck, H.-P. Schwefel, Evol. Comput. 1 (1993) 1–23.
[28] J. Tsai, R. Bonneau, A.V. Morozov, B. Kuhlman, C.A. Rohl, D. Baker, Proteins Struct. Funct. Bioinforma 53 (2003) 76–87.
[29] H. Zhou, Y. Zhou, Protein Sci. 11 (2002) 2714–2726.
[30] M.-Y. Shen, A. Sali, Protein Sci. 15 (2006) 2507–2524.
[31] H. Zhou, J. Skolnick, Biophys. J. 101 (2011) 2043–2052.
[32] D. Rykunov, A. Fiser, BMC Bioinformatics 11 (2010) 1.
[33] J. Zhang, Y. Zhang, PLoS One 5 (2010) e15386.
[34] Y. Zhang, J. Skolnick, Proteins Struct. Funct. Bioinforma. 57 (2004) 702–710.