Towards Applying River Formation Dynamics in Continuous Optimization Problems *

Pablo Rabanal, Ismael Rodríguez and Fernando Rubio

Dpto. Sistemas Informáticos y Programación
Facultad de Informática. Universidad Complutense de Madrid
C/ Prof José García Santesmases, 28040 Madrid, Spain
E-mail: prabanal@fdi.ucm.es, isrodrig@sip.ucm.es, fernando@sip.ucm.es

Abstract. River Formation Dynamics (RFD) is a metaheuristic that has been successfully used by different research groups to deal with a wide variety of discrete combinatorial optimization problems. However, no attempt has been done to adapt it to continuous optimization domains. In this paper we propose a first approach to obtain such objective, and we evaluate its usefulness by comparing RFD results against those obtained by other more mature metaheuristics for continuous domains. In particular, we compare with the results obtained by Particle Swarm Optimization, Artificial Bee Colony, Firefly Algorithm, and Social Spider Optimization.

Keywords: Swarm Intelligence, Metaheuristics, Optimization, River Formation Dynamics.

1 Introduction

Swarm intelligence methods [9] are heuristic problem-solving methods where a set of simple entities interact with each other according to their local information. The goal of these interactions is to collaboratively obtain a good solution to a given problem. Many swarm intelligence metaheuristics have been proposed in the literature, both for discrete combinatorial optimization problems (see e.g. ACO: Ant Colony Optimization [8, 7] or RFD: River Formation Dynamics [19, 21]) and for continuous domain optimization problems (see e.g. PSO: Particle Swarm Optimization [14] or ABC: Artificial Bee Colony [13])

Briefly, River Formation Dynamics (RFD) is a water-based metaheuristic [27] that consists on copying the geological forces that form rivers. When rivers fall through steep slopes, they erode some soil from the ground and transport it within the water. Later, this sediment is deposited in flatter areas of the river, where the water moves more slowly. In this way, the altitude of points traversed

* This work has been partially supported by Spanish project TIN2015-67522-C3-3-R, and by Comunidad de Madrid as part of the program S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union.

by the river iteratively changes, and the whole path tends to form a ever decreasing slope. The river and tributaries courses change along time, and eventually the formed river (together with its tributaries) represents an efficient way to gather all the rain water in some geographical area and send it to the sea. In fact, the final form of the river constitutes an efficient tradeoff between finding short paths from all raining points towards the sea (i.e. finding shortest paths) and forming a small tree of river and tributaries (i.e. finding a small spanning tree): the first goal improves if more tributaries are used, whereas the latter goal encourages collecting water by using *meanders* instead.

RFD fits particularly well in NP-hard problems consisting in creating a kind of tree, as the two tendencies commented before (i.e. finding short paths or small spanning trees) can easily be leant towards either way by means of parameters (see [22]). RFD has been applied to deal with several classical NP-hard optimization problems (see e.g. [20, 21, 23]) and has also been applied to solve industrial problems such as network routing [2, 10], robot navigation [25], VLSI design [6], or optimization in electrical power systems [1]. The interested reader is referred to [24] for a detailed survey covering the main applications of RFD. It is worth noting that, roughly speaking, RFD can be thought as a derivative-oriented version of ACO: In ACO, entities (ants) tend to move to those nodes where some value (pheromone trail) is higher, whereas in RFD, drops tend to move next to those nodes where the *difference* between the values (altitudes) at the origin and the destination nodes is higher (the flow is larger in steeper slopes).

It is worth pointing out that, so far, RFD has been applied only to problems where the solution space is discrete. In this paper we face the problem of developing a continuous version of RFD. Note that adapting a swarm method to the continuous domain may be relatively straightforward if the entities are the solutions themselves, because most of times the operators defining how each entity affects other entities can be easily generalized from the discrete to the continuous domain. However, it might not be so straightforward if solutions are defined by a structure drawn by the entities over some environment (e.g. ACO or our target method here, RFD), particularly if this structure consists of sequences of steps between consecutive neighbor solutions (points): If a continuous domain is adopted, then these paths contain an infinite amount of points, so an alternative representation would be required to denote them. More importantly, the typical expected outputs of a continuous problem are not naturally represented as a structure over an environment (e.g. a sequence of steps, a path, a round trip, or a tree over a graph). On the contrary, they are typically (and naturally) viewed as a point in a continuous-dimensional space.

Thus, in order to create a continuous version of RFD, it is a sensible choice abandoning our previous RFD view, where the output of the algorithm is a given structure (e.g. path, tree, etc.), and adopting the more natural view that the output is a point in the continuous space. Then, each drop represents a possible solution that moves around the search space. However, rather than guiding the movement of entities in terms of the fitness at each possible destination (as in PSO), each drop will consider the slopes towards other known positions (other

drops). The higher the slope, the higher the probability of moving in such direction. Thus, we consider a gradient-oriented version of continuous optimization metaheuristics.

The rest of the paper is structured as follows. In the next section we present our proposal to adapt RFD to continuous domains. Then, in Section 3 we analyze the usefulness of the approach by comparing its results against those of other metaheuristics. Finally, our conclusions and future work plans are shown in Section 4.

2 Continuous RFD

In the continuous version of RFD we had to decide what the drops would represent, and how they would move through the solution space. The most natural approach is to consider each drop as a solution to the problem, that is, a position in the search space. The evaluation of the position of a drop will be the height at which it is (the value of the solution). To move the drops we use the slopes between them as in the discrete version of the algorithm. In this way, the drops will move with higher probability approaching those drops that are at a lower height (following the slopes of maximum gradient) and, therefore, closer to the optimum.

Next, we describe in detail the RFD scheme for solving minimization continuous problems.

In Figure 1, the main steps of the presented algorithm can be seen.

```
initializeVariables()
initializeDrops()
while (not endingCondition())
    for each drop d
        if numEvals > 0 then
            if noImprove(d) then
                createDrop(d,computeRangeLimit())
            elseif not isBestDrop(d) then
                moveDrop(d)
            else
                moveBestDrop(d)
            end if
        end if
    end for
    computeRangeFactor()
end while
```

Fig. 1. Continuous RFD scheme

2.1 Initialization

In the `initializeVariables()` phase the following variables are initialized:

- `numDrops` represents the number of drops (entities) used.
- `numEvals` represents the maximum number of function evaluations allowed.
- `numStepsNoImprove`: for each drop it indicates the number of steps it can be moved without improving its previous solution. When the drop is moved `numStepsNoImprove` steps without improving, it is created again.
- `accuracy` indicates the precision, i.e. the number of decimal numbers, of the solution.
- `moveLimit` represents the percentage that limits the movement of a drop (it is used in the `moveDrop(d)` phase).
- `rangeFactor` variable allows us to focus (or unfocus) the search in a more concrete (or general) area. It will be explained later in the context of the `computeRangeFactor()` method.
- `time` represents the maximum time of execution measured in seconds.

After that, in `initializeDrops()` each drop is randomly created in the search space. For each dimension i, a random value between the minimum -$min(i)$- and maximum value -$max(i)$- of dimension i is generated defining the drop position $position_d$. In this process the best drop is stored. The best drop will be the drop with the best solution found so far, that is, the drop whose value is the minimum supposing we are minimizing a function f, where this value is the evaluation of function f in the drop position $position_d$, that is, $value = f(position_d)$.

2.2 Main Loop of the Algorithm

After the initialization has taken place, the body of the loop is executed until the `endingCondition()` is satisfied. The execution of the loop finishes when the required accuracy is achieved, when the maximum number of evaluations of the function is reached, or when the time has expired.

In the main loop, three different strategies are used to move each drop if `numEvals>0`:

- If the drop has not improved in the last `numStepsNoImprove` steps (that is, `noImprove(d)`), then the drop is randomly created in the search space (`createDrop(d,computeRangeLimit())`) depending on the `rangeFactor` variable. This variable *narrows* the dimensions where the drop can be created. Each dimension i is *reduced* in the `computeRangeLimit()` function as follows:

  $$range(i) = (max(i) - min(i)) * \texttt{rangeFactor}$$

  These ranges will limit where the drop can be created. To create the new position of the drop, first we choose a drop (cd) in a random manner depending on its values. Those drops with lower values will be selected with higher

probability. Second, and having into account the position of the chosen drop ($position_{cd}$) and the dimension limit $range$, we compute the position of the drop d. For each dimension i we randomly choose a value for $position_d(i)$ in range

$$[position_{cd}(i) - (range(i)/2), position_{cd}(i) + (range(i)/2)]$$

without exceeding the limits of the dimensions.

- If the drop is not the best drop at this moment (`not isBestDrop(d)`) then it is moved (`moveDrop(d)`) depending on the slopes between the drop d and the rest of the drops. First, these slopes are computed as follows:

$$slope(d, ad) = (position_d - position_{ad})/distance(d, ad)$$

where ad is another drop and $distance(d, ad)$ is the Euclidean distance between both drops. There exist two exceptional cases to deal with cases with non-decreasing slopes: (a) When $slope(d, ad) = 0$ we assign an epsilon slope: $slope(d, ad) = \epsilon$; (b) When $slope(d, ad) < 0$ we assign

$$slope(d, ad) = 0.1/(Abs(slope) + 1)$$

in order to allow drops climbing ascendent slopes (with a very low probability). Second, one drop is chosen as destination (dd) in a random way depending on the slopes: The higher the slope, the higher the probability of choosing that drop as destination. $slope_{dd}$ will represent the slope between drop d and dd. Third, once the destination is selected, we compute the direction in which the drop will be moved. For each dimension i,

$$direction(i) = position_{dd}(i) - position_d(i)$$

Fourth, the new position of d is calculated. In particular, for each dimension i we have:

$$newPosition_d(i) = position_d(i) + step(i)$$

where $step(i) = direction(i) * slope_{dd}$. However, there is a limit given by the expression

$$limit(i) = (max(i) - min(i)) * (\texttt{moveLimit}/100)$$

If $|step(i)| > limit(i)$ then $step(i) = limit(i)$ if $step(i) >= 0$, and $step(i) = -limit(i)$ in other case. Of course, the movement cannot exceed the limit values $min(i)$ and $max(i)$. Fifth, once the new position is known, the new value is computed: $newValue = f(position_d)$. Sixth and finally, if the new position improves the previous one, the drop is moved to the new position: $position_d = newPosition_d$. In other case, the drop remains in its previous position.

- In `moveBestDrop(d)` phase, the best drop is moved using the golden spiral. In this case, we forget about the river analogy, and we take inspiration

from [18] to analyze the surroundings of the current position. By using the golden spiral we analyze with more probability nearby points, but trying to find a good tradeoff with positions located farther. More precisely, we modify two randomly chosen dimensions i and $j = (i+1)\ mod\ dimensions$ according to the following expression:

If $exp\ mod\ 4 = 0$ or $exp\ mod\ 4 = 1$ then:

$$newPosition_d(i) = position_d(i) + addend$$

Else:

$$newPosition_d(i) = position_d(i) - addend$$

If $exp\ mod\ 4 = 0$ or $exp\ mod\ 4 = 3$ then:

$$newPosition_d(j) = position_d(j) + addend$$

Else:

$$newPosition_d(j) = position_d(j) - addend$$

where $addend = 1/\varphi^{exp}$, $\varphi = (1+\sqrt{5})/2$ is the golden ratio, and exp takes values from 1 to `accuracy` $* 10$ (increasing it one by one) for each pair of dimensions i and j. The rest of dimensions remain unchanged. Again, the drop is moved only if the new position improves the previous position.

In the last step of the loop, the `computeRangeFactor()` method modifies the variable `range_factor`. If the solution has not been improved in the last $numEvalsOneLoop$ evaluations of function f, then:

$$\texttt{rangeFactor} = \texttt{rangeFactor} * 2$$

that is, the range where a drop can be created is duplicated. In other case, the range is halved:

$$\texttt{rangeFactor} = \texttt{rangeFactor}/2$$

$numEvalsOneLoop = numDrops + accuracy * 10$ is the number of evaluations of function f in one loop of the algorithm, because we have $numDrops$ evaluations for every movement or creation of a drop, and $accuracy * 10$ evaluations when moving the best drop. Let us remark that the values of `range_factor` are limited to $10^{-\texttt{accuracy}}$ as minimum, and 1 as maximum.

After creating (`createDrop(d,computeRangeLimit())`) or moving a drop (`moveDrop(d)` or `moveBestDrop(d)`) we compare if the best solution has been improved, and if it is the case then the new best solution is stored.

Problem	PSO	ABC	FF	SSO	RFD
f_1	-15	-15	-14.99	-15	-15
f_2	-0.79	-0.79	-0.785	-0.802	-0.716
f_3	-30662.821	-30664.923	-30662.032	-30665.538	-30665.538
f_4	-6958.369	-6958.022	-6950.114	-6961.008	-6961.814
f_5	24.475	26.58	28.54	24.306	24.704
f_6	-0.749	-0.75	-0.749	-0.75	-0.75
f_7	0.05416	0.05398	0.05417	0.05394	0.08615
f_8	963.925	962.642	965.428	961.999	961.719

Table 1. Results for a benchmark of constrained optimization problems

3 Experiments

In order to assess the usefulness of our approach, we have conducted a set of experiments to compare the performance of RFD against that obtained by other more mature metaheuristics. In particular, we consider two types of case studies. First, we consider a benchmark of eight well-known optimization problems obtained from [16]. Then, we consider three real-world optimization problems dealing with concrete engineering problems. In particular, we deal with the tension/compression spring design problem [4], the welded beam design problem [3], and the speed reducer design problem [12]. All cases can be described as optimization problems where a minimization has to be done subject to fulfill a given set of constraints. For all the case studies, we compare the results obtained by RFD with the results obtained by Particle Swarm Optimization (PSO [14]), Artificial Bee Colony (ABC [13]), Firefly Algorithm (FF [28]), and Social Spider Optimization (SSO [5]).

Table 1 summarizes the results obtained with the first benchmark, while Table 2 summarizes the results obtained with the three real-world engineering optimization problems. In all cases, the results correspond with the average of 30 independent executions of each algorithm. Regarding the parameter tuning, in the case of RFD the number of drops (`numDrops`) used was 50; `numStepsNoImprove` was set to 10; the `accuracy` value varies between 4 and 8, depending on the problem; the `moveLimit` value used in the experiments was 90; the initial value of `rangeFactor` is 1 in all cases; while the `time` was set to values from 10 to 300 seconds. Regarding the other metaheuristics, we have used the configurations described in [5].

Problem	PSO	ABC	FF	SSO	RFD
Tension/compression	0.0148631	0.0128507	0.0129307	0.0127649	0.0127486
Welded-beam	2.01115	2.16736	2.19740	1.74646	1.727833
Speed reducer	3079.262	2998.063	3000.005	2996.113	2994.805

Table 2. Results for a benchmark of real-world optimization engineering problems

In order to appropriately compare the metaheuristics, we perform a statistical test. In general, a Friedman test can be used to check whether the hypothesis that all methods behave similarly (the null hypothesis) holds or not. However, since the number of metaheuristics under consideration is low, using a Friedman aligned ranks test is more recommended in this case. This test does not rank methods for each problem separately (as Friedman test does), but construct a global ranking where values of all methods and problems are ranked together. In Friedman aligned ranks test, for each problem the difference of each method with respect to the average value for all methods is considered, and next all values of all problems are ranked together. Table 3 shows the results of applying an Aligned Friedman test, considering five metaheuristics and eleven case studies (that is, putting together all the case studies from both benchmarks). As it can be seen, RFD obtains the highest overall score, with a very small difference over SSO. In fact, a more detailed analysis using Holm's procedure shows that there is not an statistical relevant difference between RFD and SSO. Although the null hypothesis can not be rejected to differentiate RFD and SSO when considering both benchmarks together, we can try to analyze each case study independently. In this case, it is worth to mention that SSO outperforms RFD in the first set of examples, while RFD outperforms SSO in the case of the real-world engineering optimization problems. That is, RFD behaves better when the problems are harder.

4 Conclusions

We have provided a first approach to adapt RFD to deal with continuous domain optimization problems. The results we have obtained are promising. In particular, RFD obtains competitive results against ABC, PSO, FF, and SSO. However, there is still plenty of space for improvement. Let us remark that in our current approach we do not take profit from a basic RFD issue in discrete domains: erosion. In fact, our main line of current work is integrating erosion into continuous RFD. The basic idea is to use an alternative fitness function f' recording erosion, where this f' function is computed by using a data structure that records information about all the positions that have been explored so far by the algorithm.

Metaheuristic	Ranking
RFD	21,5455
SSO	22,7273
ABC	26,7273
PSO	29,9545
FF	39,0455

Table 3. Ranking Aligned Friedman Results

In addition to including erosion, we are also working on improving the performance of the algorithm by providing a parallel implementation of our metaheuristic. In this sense, we are using the parallel functional language Eden (see e.g. [15, 11, 17]) to extend our library of parallel versions of metaheuristics (see [26]) to deal with RFD.

Acknowledgments

The authors would like to thank Alberto de la Encina for valuable suggestions about the development of a version of RFD to deal with continuous domain optimization problems.

References

1. Hatim G. Abood, Victor Sreeram, and Yateendra Mishra. Optimal placement of PMUs using river formation dynamics (RFD). In *2016 IEEE International Conference on Power System Technology (POWERCON)*, pages 1–6, Sept 2016.
2. Saman Hameed Amin, HS Al-Raweshidy, and Rafed Sabbar Abbas. Smart data packet ad hoc routing protocol. *Computer Networks*, 62:162–181, 2014.
3. Leticia C Cagnina, Susana C Esquivel, and Carlos A Coello Coello. Solving engineering optimization problems with the simple constrained particle swarm optimizer. *Informatica*, 32(3), 2008.
4. Carlos A Coello Coello. Use of a self-adaptive penalty approach for engineering optimization problems. *Computers in Industry*, 41(2):113–127, 2000.
5. Erik Cuevas and Miguel Cienfuegos. A new algorithm inspired in the behavior of the social-spider for constrained optimization. *Expert Systems with Applications*, 41(2):412–425, 2014.
6. Satyabrata Dash, Sukanta Dey, Deepak Joshi, and Gaurav Trivedi. Minimizing area of VLSI power distribution networks using river formation dynamics. *Journal of Systems and Information Technology*, 20(4):417–429, 2018.
7. Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

8. Marco Dorigo and Gianni Di Caro. Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 IEEE Congress on Evolutionary Computation, CEC'99*, volume 2, pages 1470–1477. IEEE, 1999.
9. Russell C. Eberhart, Yuhui Shi, and James Kennedy. *Swarm Intelligence*. Morgan Kaufmann, 2001.
10. Koppala Guravaiah and R Leela Velusamy. Energy efficient clustering algorithm using RFD based multi-hop communication in wireless sensor networks. *Wireless Personal Communications*, 95(4):3557–3584, 2017.
11. Mercedes Hidalgo-Herrero, Yolanda Ortega-Mallén, and Fernando Rubio. Analyzing the influence of mixed evaluation on the performance of Eden skeletons. *Parallel Computing*, 32(7-8):523–538, 2006.
12. Majid Jaberipour and Esmaile Khorram. Two improved harmony search algorithms for solving engineering optimization problems. *Communications in Nonlinear Science and Numerical Simulation*, 15(11):3316–3331, 2010.
13. Dervis Karaboga and Bahriye Akay. A modified artificial bee colony (ABC) algorithm for constrained optimization problems. *Applied Soft Computing*, 11(3):3021–3031, 2011.
14. James Kennedy. Particle swarm optimization. In *Encyclopedia of machine learning*, pages 760–766. Springer, 2011.
15. Ulrike Klusik, Ricardo Peña, and Fernando Rubio. Replicated workers in Eden. In *Constructive Methods for Parallel Programming (CMPP'00)*. Nova Science, 2000.
16. JJ Liang, Thomas Philip Runarsson, Efrén Mezura-Montes, Maurice Clerc, Ponnuthurai Nagaratnam Suganthan, CA Coello Coello, and Kalyanmoy Deb. Problem definitions and evaluation criteria for the cec 2006 special session on constrained real-parameter optimization. *Journal of Applied Mechanics*, 41(8):8–31, 2006.
17. Rita Loogen. Eden–parallel functional programming with Haskell. In *Central European Functional Programming School*, pages 142–206. Springer, 2011.
18. Natalia López, Manuel Núñez, Ismael Rodríguez, and Fernando Rubio. Introducing the golden section to computer science. In *Proceedings First IEEE International Conference on Cognitive Informatics*, pages 203–212. IEEE, 2002.
19. Pablo Rabanal, Ismael Rodríguez, and Fernando Rubio. Using river formation dynamics to design heuristic algorithms. In *International Conference on Unconventional Computation, UC'07*, pages 163–177. Springer, 2007.
20. Pablo Rabanal, Ismael Rodríguez, and Fernando Rubio. Solving dynamic TSP by using river formation dynamics. In *Fourth International Conference on Natural Computation (ICNC'08)*, pages 246–250. IEEE, 2008.
21. Pablo Rabanal, Ismael Rodríguez, and Fernando Rubio. Applying river formation dynamics to solve NP-complete problems. In *Nature-inspired algorithms for optimisation*, pages 333–368. Springer, 2009.
22. Pablo Rabanal, Ismael Rodríguez, and Fernando Rubio. Applying RFD to construct optimal quality-investment trees. *J. Universal Computer Science*, 16(14):1882–1901, 2010.
23. Pablo Rabanal, Ismael Rodríguez, and Fernando Rubio. Studying the application of ant colony optimization and river formation dynamics to the steiner tree problem. *Evolutionary Intelligence*, 4(1):51–65, 2011.
24. Pablo Rabanal, Ismael Rodríguez, and Fernando Rubio. Applications of river formation dynamics. *Journal of computational science*, 22:26–35, 2017.
25. Grzegorz Redlarski, Mariusz Dabkowski, and Aleksander Palkowski. Generating optimal paths in dynamic environments using river formation dynamics algorithm. *Journal of Computational Science*, 20:8–16, 2017.

26. Fernando Rubio, Alberto de la Encina, Pablo Rabanal, and Ismael Rodríguez. A parallel swarm library based on functional programming. In *International Work-Conference on Artificial Neural Networks*, pages 3–15. Springer, 2017.
27. Fernando Rubio and Ismael Rodríguez. Water-based metaheuristics: How water dynamics can help us to solve NP-hard problems. *Complexity*, 2019.
28. Xin-She Yang. Firefly algorithm, levy flights and global optimization. In *Research and development in intelligent systems XXVI*, pages 209–218. Springer, 2010.