10.1109/ICNC.2008.760
Fourth International Conference on Natural Computation

Solving Dynamic TSP by using River Formation Dynamics *

Pablo Rabanal, Ismael Rodríguez, and Fernando Rubio
Dept. Sistemas Informáticos y Computación
Facultad de Informática, Universidad Complutense de Madrid
C/Prof. José García Santesmases, s/n, 28040 Madrid, Spain
prabanal@fdi.ucm.es, {isrodrig,fernando}@sip.ucm.es

Abstract

River Formation Dynamics (RFD) is an heuristic optimization algorithm based on copying how water forms rivers by eroding the ground and depositing sediments. After drops transform the landscape by increasing/decreasing the altitude of places, solutions are given in the form of paths of decreasing altitudes. Decreasing gradients are constructed, and these gradients are followed by subsequent drops to compose new gradients and reinforce the best ones. We apply this method to solve dynamic TSP. We show that the gradient orientation of RFD makes it specially suitable for solving this problem, and we compare our results with those given by Ant Colony Optimization (ACO).

1 Introduction

One of the most paradigmatic NP-complete problems is TSP, the *Traveling Salesman Problem*, which captures, under a very simple problem definition, all the typical difficulties appearing in optimal path construction problems. In particular, this problem harbors an essential difficulty that is common to a big variety of routing problems: Steps that are good in the short term do not necessarily lead to good complete routes. Even more strongly related to routing problems is the *dynamic TSP* variant, that is, TSP under the assumption that nodes (i.e. hosts) and edges (i.e. connections) can appear/disappear along time, which illustrates the fact that networks change along time and routing algorithms must be adapted to this.

Solving TSP or dynamic TSP to find optimal paths in a network requires to face a double difficulty: (a) We are solving an NP-hard problem, and (b) the data required to make decisions is intrinsically distributed. Regarding the first problem, this means that we need a method providing a suitable tradeoff between (semi-)optimality and time consumption. Concerning the second issue, let us note that any solution based on assuming that all relevant network variables can be collected in a single central node is not feasible. Thus, algorithms where decision making is made according to *local* information are preferable.

Evolutionary Computation (EC) methods (see e.g. [1, 6, 9]) provide algorithms that intrinsically face both previous problems. On the one hand, EC methods are based on the assumption that similar solutions *use to* provide comparable solutions *but* that this is not always true, which is appropriate to heuristically solve NP-hard problems. On the other hand, EC methods are based on manipulating very simple entities that evolve according to *local* information, which is suitable for making distributed implementations. This is the case of the *Ant Colony Optimization* (ACO) approach [4, 2, 3]. This method provides algorithms based on copying how (natural) ants find the shortest path from the colony to the nest. The root of this method is the following: A pheromone value is attached to each place, and ants probabilistically tend to choose those edges where the ratio *'pheromone trail at destination'/'edge cost'* is the highest. Alternatively, let us consider that the decision were based on the *gradient* of trail values instead of on the trail values themselves. In particular, let us suppose that ants probabilistically tend to choose the movement providing the highest ratio *'difference of trails between the new place and the current place'/'edge cost'*. Leaving aside by now the important question of how ants could iteratively create paths of *decreasing* pheromone trials (which will be addressed later), what are the differences between this gradient approach and the standard approach?

First, in the standard approach ants can be led by pheromone trials in such a way that, after some movements, it is impossible not to repeat a node, i.e. a local cycle is followed. When an ant finds that it cannot avoid to repeat a node, it is either *killed* or reinserted at the origin node. In both cases, the computational effort required to move it was useless. However, following a cycle is impossible in

*Research partially supported by projects TIN2006-15578-C02-01, PAC06-0008-6995, and MRTN-CT-2003-505121/TAROT.

978-0-7695-3304-9/08 $25.00 © 2008 IEEE
DOI 10.1109/ICNC.2008.760

the gradient approach because it would require an *ever decreasing* cycle, which is contradictory. Second, when an ant finds a shorter path in the standard approach, it needs a lot of movements to *convince* other ants following older well-reinforced paths to join the new path. Technically speaking, reinforcing the new path until pheromone trails are higher than in older paths requires a lot of subsequent steps. On the other hand, if the difference of trails is considered then, when a shorter path is discovered, from this precise moment on its edges are preferable in the overall to the edges of older paths. This is because the difference of pheromone trails between the final destination and the origin is the same in these paths (the origin and the destination are the same indeed), but the cost is lower in the shorter path. So, the ratio *'total difference of trails'*/*'total cost'* is higher in the shorter path. On the contrary, when a shorter path is found in the standard approach, the edges of this path are not preferable yet (not even when considered as a whole) because the amount of pheromones in its edges is still negligible.

Though adopting this alternative approach provides some advantages, it leads to the following question: If formed paths are not sequences of high pheromone trails but sequences of *decreasing* pheronome trails, how pheromone values must be changed after an ant moves? We can find an answer to this question by giving the ant metaphor up and getting some inspiration from another nature-based phenomenon: The *river formation dynamics*.

In [11] an algorithm based on these ideas called RFD (*River Formation Dynamics*) was presented. In this paper we adapt RFD to deal with TSP in *dynamic* networks, i.e. networks where nodes and connections may appear/disappear along time. This problem has several applications, both inside the IT domain (e.g., routing in faulty/congested networks) and outside it (e.g., traffic planning). We identify and reinforce the characteristics of RFD that improve its adaptability to graph changes. Moreover, the Geology metaphor provides another characteristic that is important in this regard. Let us note that the *erosion* process provides a method to *punish* inefficient paths as well as to avoid blocked paths: If a path leads to a node that is lower than any adjacent node (i.e., it is a blind alley) then the drop will deposit its sediment, which will increase the altitude of the node. Eventually, the altitude of this node will match the altitude of its neighbors, which will avoid that other drops fall in this node. If the ground reaches this level, other drops will be allowed to cross this node from one adjacent node to another. Thus, paths will not be interrupted at this point. This provides an implicit method to avoid inefficient behaviors of drops, as well as to find alternative paths when an old established path is interrupted by the disappearance of an edge or a node.

We present a general RFD scheme for finding short paths in dynamic graphs. Besides, we adapt this general scheme to dynamic TSP and we compare the results of RFD with those of ACO for the same case studies. Let us remark that only some improvements, out a big set of choices, have already been applied to the basic RFD scheme (both in the static and in the dynamic case). Thus, we think that these results are not only interesting but also promising.

The rest of the paper is structured as follows. Next we describe the behavior of RFD. In Section 3 we apply both RFD and ACO to solve dynamic TSP instances. Finally, in Section 4 we present our conclusions.

2 River Formation Dynamics method

In this section we introduce the basic structure of our method. It works as follows. Instead of associating pheromone values to edges, we associate *altitude* values to nodes. *Drops* erode the ground (they reduce the altitude of nodes) or deposit the sediment (increase it) as they move. The probability of the drop to take a given edge instead of others is proportional to the gradient of the down slope in the edge, which in turn depends on the difference of altitudes between both nodes and the distance (i.e. the *cost* of the edge). At the beginning, a flat environment is provided, that is, all nodes have the same altitude. The exception is the destination node, which is a *hole*. Drops are unleashed at the origin node, which spread around the flat environment until some of them fall in the destination node. This erodes adjacent nodes, which creates new down slopes, and in this way the erosion process is propagated. New drops are inserted in the origin node to transform paths and reinforce the erosion of promising paths. After some steps, good paths from the origin to the destination are found. These paths are given in the form of sequences of decreasing edges from the origin to the destination.

2.1 Basic algorithm

The basic scheme of the RFD algorithm follows:

```
initializeDrops()
initializeNodes()
while (not allDropsFollowTheSamePath())
      and (not otherEndingCondition())
   moveDrops()
   erodePaths()
   depositSediments()
   analyzePaths()
end while
```

The scheme shows the main ideas of the proposed algorithm. We comment on the behavior of each step. First, drops are initialized (`initializeDrops()`), i.e., all drops are put in the initial node. Next, all nodes of the graph are initialized (`initializeNodes()`). This consists of two operations. On the one hand, the altitude of the destination node is fixed to 0. In terms of the river formation

dynamics analogy, this node represents the *sea*, that is, the final goal of all drops. On the other hand, the altitude of the remaining nodes is set to some equal value.

The `while` loop of the algorithm is executed until either all drops find the same solution (`allDropsFollowTheSamePath()`), that is, all drops traverse the same sequence of nodes, or another alternative finishing condition is satisfied (`otherEndingCondition()`). This condition may be used, for example, for limiting the number of iterations or the execution time. Another choice is to finish the loop if the best solution found so far is not surpassed during the last n iterations.

The first step of the loop body consists in moving the drops across the nodes of the graph (`moveDrops()`) in a partially random way. The following *transition rule* defines the probability that a drop *k* at a node *i* chooses the node *j* to move next:

$$P_k(i,j) = \begin{cases} \frac{decreasingGrad(i,j)}{\sum_{l \in V_k(i)} decreasingGrad(i,l)} & \texttt{if } j \in V_k(i) \\ 0 & \texttt{if } j \notin V_k(i) \end{cases} \quad (1)$$

where $V_k(i)$ is the set of nodes that are *neighbors* of node *i* that can be visited by the drop *k* and *decreasingGrad(i,j)* represents the negative gradient between nodes *i* and *j*, which is defined as follows:

$$decreasingGrad(i,j) = \frac{altitude(j) - altitude(i)}{distance(i,j)} \quad (2)$$

where *altitude(x)* is the altitude of the node *x* and *distance(i,j)* is the length of the edge connecting node *i* and node *j*. Let us note that, at the beginning of the algorithm, the altitude of all nodes is the same, so $\sum_{l \in V_k(i)} decreasingGradient(i,l)$ is 0. In order to give a special treatment to flat gradients, we modify this scheme as follows: We consider that the probability that a drop moves through an edge with 0 gradient is set to some (non null) value. This enables drops to spread around a flat environment, which is mandatory, in particular, at the beginning of the algorithm.

In fact, going one step further, we also introduce this improvement: We let drops climb *increasing* slopes with a low probability. This probability will be inverse proportional to the increasing gradient, and it will be reduced during the execution of the algorithm by using a similar method to the one used in *Simulated Annealing* (see [10]). This new feature improves the search of good paths.

In the next phase (`erodePaths()`) paths are eroded according to the movements of drops in the previous phase. In particular, if a drop moves from node A to node B then we erode A. That is, the altitude of this node is reduced depending on the current gradient between A and B. The erosion is higher if the down slope between A and B is high. If the edge is flat or increasing then a small erosion is performed. The altitude of the final node (i.e., the *sea*) is never modified and it remains equal to 0 during all the execution.

Once the erosion process finishes, the altitude of all nodes is slightly increased (`depositSediments()`). The objective is to avoid that, after some iterations, the erosion process leads to a situation where all altitudes are close to 0, which would make gradients negligible and would ruin all formed paths. In fact, we also enable drops to *deposit* sediment in nodes. This happens when all movements available for a drop imply to climb an increasing slope and the drop fails to climb any edge (according to the probability assigned to it). In this case, the drop is blocked and it deposits the sediments it transports. This increases the altitude of the current node. The increment is proportional to the amount of cumulated sediment.

Finally, the last step (`analyzePaths()`) studies all solutions found by drops and stores the best one.

2.2 Adapting the Method to TSP

In this section we show how the previous general scheme is particularized to the TSP problem. Further details can be found in [11].

The adaptation of our method to TSP has several similarities with the way ACO is applied to this problem. Like ants, each drop must remember the nodes traversed so far in previous movements. In particular, this is necessary to avoid repeating nodes. In spite of the fact that the use of gradients strongly minimizes these situations in RFD, sequences of (improbable) climbing movements can still lead a drop to a local cycle. Besides, altitudes are not eroded after each individual drop movement. On the contrary, the altitudes of traversed nodes are eroded all together when the drop actually completes a round-trip (as ants do when they increase pheromone trails). This is another reason to keep track of the sequence of traversed nodes. As in ACO, when a drop finds that all adjacent nodes have already been visited, the drop disappears (again, this is much less probable in RFD than in ACO). There is an additional reason in RFD why a drop may disappear: When all adjacent nodes are *higher* than the actual node (i.e., the node is *valley*) and the drop fails to climb any up gradient, the drop is removed and it deposits the sediment it carries at the current node. Note that, in this case, the computations performed to move the drop should not be considered as *lost* despite of the fact that it did not found a solution. This is because the cumulation of sediments increases the node altitude, and thus gradients coming from adjacent nodes are flatted. This eventually avoids that other drops fall in the same *blind alley*.

The adaptation of RFD to TSP has other peculiarities that do not appear in ACO. Our algorithm provides an intrinsic method to avoid that drops traverse cycles, but the goal of TSP consists in finding a *cycle* indeed. In order to allow drops to follow a cycle involving *all* nodes, the origin node (which, as we said before, can be any fix node) is *cloned*

as well as all edges involving it. The original node plays the role of *origin* node and the cloned node plays the role of *destination* node. In this way, drops can form decreasing gradients from the origin node to the destination node (for us, from a node to *"itself"*). Finally, the adaptation of RFD to TSP requires to introduce other additional nodes for different technical reasons. The number of these additional nodes, called *barrier* nodes, belongs to $O(n^2)$ where n is the number of nodes of the original graph. The reasons for introducing these nodes are explained in detail in [11]. In the next section, ACO and RFD will be compared in terms of graphs with the same number of *standard* nodes. That is, if an experiment deals with a 30 nodes graph, then the quality of solutions and the consumed time are computed and compared in the case where ACO deals with the original 30-nodes graph and RFD deals with the corresponding adapted *bigger* graph.

3 Applying RFD to Dynamic TSP

In this section we describe the application of our approach to solve dynamic TSP and we report some experimental results. We compare the results found by using ACO methods and the solutions found by using our method. All the experiments were performed in an Intel T2400 processor with 1.83 GHz.

In these experiments we follow the next procedure. First, we calculate the solution of an instance of the problem by using both algorithms. Next, we introduce one of the following changes in the graph: (a) We delete an edge that is common to the solutions found by both methods; (b) We delete a node of the graph; (c) We add a new node.

Once a change is introduced, we calculate a solution for this instance of the problem. Let us remark that we do not restart the algorithms, but we keep the state of the methods (that is, the amount of pheromone at each edge and the altitudes of the nodes, respectively) just before the change.

In Figures 1-4 we show the results of experiments with a graph of 100 nodes. These figures show the best solution found by each method for each execution time before and after the changes are introduced. These graphs have been randomly generated assuming that each node is connected with only a few other nodes (between 10 and 20 connections per host) as it is the most common case in complex networks. The basic shape shown in the figures is also obtained with other benchmark examples. Next we comment the experimental results.

Deleting Edges

In this experiment we can see how RFD and ACO recalculate the solution after deleting an edge of the graph (i.e. after removing a connection of the network). Figure 1 shows the results of an experiment where the input of both algorithms

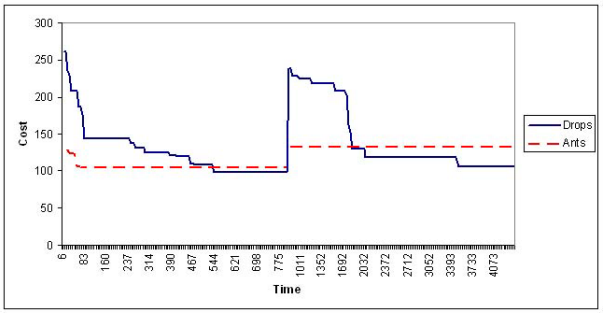

Figure 1. Deleting edges

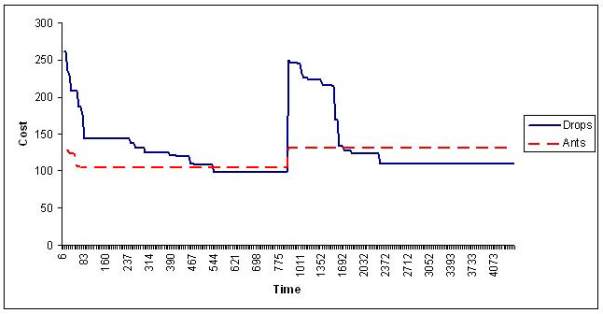

Figure 2. Deleting nodes

was a graph with 100 nodes.

It is specially remarkable that the difference between the best solutions obtained by RFD and ACO is increased after the change is introduced. That is, RFD has more capacity to reconstruct good solutions after a change happens.

Deleting Nodes

In this experiment we show how both methods recalculate the solution after deleting a node of the graph (i.e. after a host of the network breaks down). Figure 2 shows the results obtained for the graph with 100 cities.

Note that the results are similar to those obtained when removing one edge.

Adding Nodes

Finally, we show how both methods recalculate their solutions after adding a new host in the network. When we are using our algorithm and we add a new node, we have to set its altitude to some value. This initial altitude should ease as much as possible the task of finding new solutions. In particular, the altitude of the new node is set to the average of its neighbors. We use this altitude because it is the value that is less aggressive with its environment, and in fact we observed that it allows to find good solutions.

On the other hand, there are a lot of different strategies to treat dynamic graphs in ACO (see e.g. [7, 8, 5]. When a change is introduced in the graph, it is necessary to reset part of the pheromone information.

Figure 3 shows the results of an experiment where the

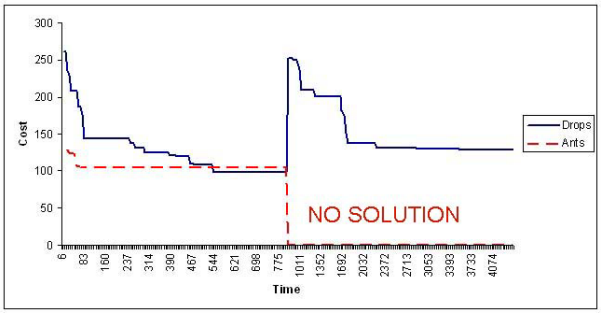

Figure 3. Adding nodes

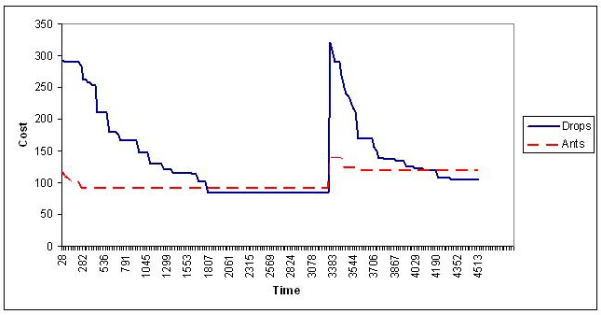

Figure 4. Introducing several changes

input of both algorithms was the graph with 100 nodes. Let us remark that, in this case, ACO did not find any solution to the problem after the node is added. However, RFD obtains a solution. Hence, in this case the advantage of RFD over ACO is quite relevant. Obviously, we could completely restart the computation of ACO from the beginning.

Introducing several changes

Now, instead of introducing only one change in the graph, we use a more dramatic approach: We introduce several modifications at the same time. The changes we have introduced in these experiments consist in simultaneously making these modifications: Deleting two edges that are common to the solutions found by both methods; Adding two new edges; Adding two new nodes (with its corresponding edges to other nodes); and deleting two old nodes.

We consider an experiment where a randomly generated graph of 150 nodes is used. The average degree of nodes is 15. As we show in Figure 4, the basic shape of the results is basically the same as in the previous examples. That is, ants are better in the short term but drops obtain better results in the medium term. Thus, the basic results are the same even if several modifications are performed at the same time.

4 Conclusions

We extract the following general conclusions from our experimental results: (1) In both static and dynamic graphs, ACO works faster than RFD, but in the long term RFD obtains better solutions in both cases; (2) RFD works faster in the dynamic case than in the static case; and (3) RFD always obtains a solution after a modification is introduced, while sometimes ACO cannot adapt the solution constructed before changing the graph to the new scenario. These features are a consequence of the fact that the exploration of the graph is *deeper* in RFD than in ACO, which in turn is due to the differences between both methods.

A relevant aspect to be considered is that sometimes ACO does not find a solution after introducing changes in the graphs. When ACO converges to a solution, pheromone trails not involved in the solution sequence are negligible, i.e. only the edges included in the solution sequence have significant pheromone trails. Hence, no information about other alternatives is kept for being used when the graph changes. In particular, resetting missing pheromone trails to some default values does not recover this information. On the other hand, when RFD converges to a solution, all nodes are provided with some altitude, which gives them a relative position in the graph. That is, providing a solution does not require to completely reset part of the graph information. Thus, this information helps RFD to react to subsequent graph changes.

References

[1] T. Bäck and H. Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1):1–23, 1993.

[2] M. Dorigo. *Ant Colony Optimization*. MIT Press, 2004.

[3] M. Dorigo and L. Gambardella. Ant colonies for the traveling salesman problem. *BioSystems*, 43(2):73–81, 1997.

[4] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Tran. Systems, Man and Cybernetics, Part B*, 26(1):29–41, 1996.

[5] C. J. Eyckelhof and M. Snoek. Ant systems for a dynamic TSP - ants caught in a traffic jam. In *ANTS Workshop, LNCS 2463*, pages 88–99. Springer, 2002.

[6] D. Fogel. What is evolutionary computation? *Spectrum, IEEE*, 37(2):26, 28–32, 2000.

[7] M. Guntsch and M. Middendorf. Pheromone modification strategies for ant algorithms applied to dynamic TSP. In *Applications of Evolutionary Computing, LNCS 2037*, pages 213–222. Springer, 2001.

[8] M. Guntsch, M. Middendorf, and H. Schmeck. An ant colony optimization approach to dynamic TSP. In *Genetic and Evolutionary Computation Conference*, pages 860–867. Morgan Kaufmann, 2001.

[9] K. Jong. *Evolutionary computation: a unified approach*. MIT Press, 2006.

[10] S. Kirkpatrick, C. G. Jr., and M. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671, 1983.

[11] P. Rabanal, I. Rodríguez, and F. Rubio. Using river formation dynamics to design heuristic algorithms. In *Unconventional Computation, LNCS 4618*, pages 163–177. Springer, 2007.