Applying River Formation Dynamics to Solve NP-Complete Problems

Pablo Rabanal, Ismael Rodríguez, and Fernando Rubio

Abstract. For obvious practical reasons, NP-complete problems are typically solved by applying heuristic methods. In this regard, nature has inspired many heuristic algorithms to obtain *reasonable* solutions to complex problems. One of these algorithms is *River Formation Dynamics* (RFD). This heuristic optimization method is based on imitating 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. In this chapter, we apply RFD to solve three NP-complete problems, and we compare our results with those obtained by using *Ant Colony Optimization* (ACO).

1 Introduction

NP-complete problems are (strongly) supposed to require exponential time in the worst case to be solved. Fortunately, heuristic methods can be used to obtain sub-optimal solutions in reasonable time. Nature has been a source of inspiration for obtaining many interesting and useful heuristic algorithms (see e.g. [12, 13, 4, 11, 5, 10]). Among them, we would like to highlight *Ant Colony Optimization* (ACO) [7, 5, 6]. This method provides algorithms based on how (natural) ants find the shortest path from the colony to the food source. This efficient method is well-known: (1) Ants release pheromones as they move; (2) ants tend to follow pheromone trails; and thus, (3) paths are reinforced. In the long term, the reinforcement of paths is stronger in short paths than in long paths because ants can traverse the former paths more often per unit of time. Eventually, only a good short path

Pablo Rabanal · Ismael Rodríguez · Fernando Rubio
Dept. Sistemas Informáticos y Computación, Facultad de Informática,
Universidad Complutense de Madrid, 28040 Madrid, Spain
e-mail: prabanal@fdi.ucm.es, {isrodrig, fernando}@sip.ucm.es

R. Chiong (Ed.): Nature-Inspired Algorithms for Optimisation, SCI 193, pp. 333–368.
springerlink.com © Springer-Verlag Berlin Heidelberg 2009

prevails and the rest of paths vanish. A pheromone value is attached to each edge, and ants probabilistically tend to choose those edges where the ratio *'pheromone trail at destination'/'edge cost'* is the highest.

Alternatively, let us suppose that ants' decisions were based on the *gradient* of trail values instead of 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'* (for instance, higher the decrease, higher the probability). Leaving aside for now the important question of how ants could iteratively create paths of *decreasing* pheromone trails (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 trails in such a way that, after some movements, it is impossible not to repeat a node, i.e. a local cycle is followed. Let us note that following a cyclic pheromone trail does not imply that any previous ant actually followed this cycle; in particular, each part of the cycle could have been reinforced by an ant following a different path. 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 the gradient approach because it would require an *ever decreasing* cycle, which is contradictory. Second, let us note that in the standard approach, when an ant finds a shorter path, it needs a lot of movement 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 onwards its edges are preferable (on average) 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.

This alternative approach provides some advantages, but the following question arises: How can we get pheromone trails to decrease along the steps of each path? 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*. Let us consider that a water mass is unleashed at some high point. Gravity will make it follow a path down until it cannot go down anymore. In geological terms, when it rains in a mountain, water tries to find its own way down to the sea. Along the way, water erodes the ground and transforms the landscape, which eventually creates a riverbed. When a strong downward slope is traversed by water, it extracts soil from the ground on the way. This soil is deposited later when the slope is lower. Rivers affect the environment by reducing (i.e. eroding) or increasing (i.e. depositing) the altitude of the ground. Let us note that if water is unleashed at all points of the landscape (e.g., it rains) then the river form tends to optimize the task

of collecting *all* the water and taking it to the sea, which does not imply taking the shortest path from a *given* origin point to the sea. Let us remark that there are *a lot of* origin points to consider (one for each point where a drop falls). In fact, a kind of *combined* grouped shortest path is created in this case. The formation of tributaries and meanders is a consequence of this. However, if water flows from a *single* point and no other water source is considered, then the water tends to form an efficient way to reduce the altitude (i.e., it tends to form a short path between the origin and the destination).

An algorithm based on these ideas called *River Formation Dynamics* (RFD) was presented in [17]. In order to apply the previous scheme, ants are substituted by *drops* and pheromone trails are replaced by *altitudes*. Drops tend to flow down the slope and they modify altitudes in the process. A classical benchmark NP-complete problem, the *Traveling Salesman Problem* (TSP) [9, 1], was considered, which required to adapt the general scheme to this particular problem (for instance, since the general framework implicitly instructs that drops avoid cycles, a change was introduced to allow cycles involving *all* nodes). The applicability of RFD to other NP-complete problems was studied in [18]. Given a cost-evaluated graph, let us consider the problems of (a) finding the minimum spanning tree, and (b) finding the minimum *distances* tree (that is, a tree such that the addition of distances from each node to a given *exit node* is minimal). The standard forms of both problems do not require using heuristic methods because they can be polynomially solved (e.g., by using Kruskal and Dijkstra algorithms, respectively). However, some generalizations of both problems are NP-complete indeed. In particular, let us consider that the cost of taking an edge e depends on the path followed so far. That is, if we traverse e *after* following a path σ then the cost of adding e to the path is $c_{e,\sigma}$; in general, we have $c_{e,\sigma} \neq c_{e,\sigma'}$ for any other path σ'. We denote these graphs as *variable-cost graphs*. The problems of finding a minimum spanning tree or a minimum distances tree for a variable-cost graph (denoted by MSV and MDV, respectively) were defined in [18], where the capability of RFD to solve them was considered. As we will discuss later, both generalizations of the standard problems are NP-complete, and they are applicable to some IT domains (e.g. *formal testing methods* and *routing*). However, to the best of our knowledge, they have not been considered in the literature before.

Since TSP, MSV, and MDV consist in finding some kinds of short paths, the characteristics of RFD commented before (that is, the avoidance of local cycles and the fast reinforcement of shorter paths) make it a suitable choice. Moreover, the geological 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 other drops falling on 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.

In [17] and [18], the performance of RFD to solve TSP and MSV, respectively, was analyzed by comparing the results of RFD with those given by an ACO implementation for the same problem instances. It was observed that the time required by RFD to find good solutions is in general longer than the time required by ACO to find equivalent solutions, though solutions provided by RFD outperformed those given by ACO after some additional time passes. As we will explain later in higher detail, the reasons for these differences lie in the fact that RFD develops a *deeper* exploration of the graph. In this chapter we summarize our previous work on RFD in an integrated fashion. We sketch the main ideas of the algorithm and we recall previous experiments. In addition, we report and analyze other experiments where RFD and ACO are compared for the same instances of the MDV problem. Moreover, we conduct new experiments to study the capability of RFD and ACO to deal with *dynamic* graphs, i.e. graphs where nodes and edges can appear/disappear along time, in the three problems. This will allow us to study the capability of drops and ants to dynamically *adapt* paths found so far to environmental changes. Finally, in order to demonstrate the difficulty of MSV and MDV, we prove the NP-completeness of both problems. In particular, we polynomially reduce 3-SAT to each of them.

The rest of the chapter is structured as follows. Next we describe the main ideas of the RFD algorithm. In Section 3, we formally define the problems we have considered in this chapter to analyze the performance of RFD. Next, in Section 4 we apply RFD and ACO to solve TSP, MSV, and MDV in the case where graphs are static (i.e., they do not change along time), and we report some results. These three problems are revisited in Section 5, where we repeat these experiments in a context where graphs are dynamic. We present our conclusions and lines of future work in Section 6. We prove the NP-completeness of MSV and MDV in the appendix of this chapter.

2 Main Ideas of RFD

In this section we introduce the basic structure of our method based on river formation dynamics. The method 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 downward slope in the edge, which in turn depends on the difference of altitudes between both nodes and the edge 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 and spread around the flat environment until some of them fall in the destination node. This erodes adjacent nodes, which creates new downward 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.

This method provides some advantages over ACO that were briefly outlined before in the introduction. On one hand, local cycles are not created and reinforced because they would imply an *ever decreasing cycle*, which is contradictory. Though ants usually take into account their past path to avoid repeating nodes, they cannot avoid being led by pheromone trails through some edges in such a way that a node must be repeated in the next step. However, *altitudes* cannot lead drops to these situations. Moreover, since drops do not have to worry about following cycles, in general drops do not need to be endowed with *memory* of previous movements, which releases some computational memory and reduces some execution time.[1] On the other hand, when a shorter path is found in RFD, the subsequent reinforcement of the path is fast: Since the same origin and destination are concerned in both the old and the new path, the difference of altitude is the same but the distance is different. So, the (global) gradient of the shorter path makes it preferable to any other path connecting the same nodes. This does not necessarily imply that all *individual* edges in this path are immediately preferred to their respective competitors; this only means that these edges are preferred on an *average*. In particular, bad-gradient steps in the shorter path must be compensated by other good gradients in the path; otherwise, the alternative path cannot be shorter. The erosion/sedimentation process tends to equalize gradients in paths, so bad gradients in the shorter path tend to gain the surplus from other good gradients in the path. In this way, all edges of this path easily tend to be preferable to other edges. On the contrary, when a shorter path is found in ACO, this path is not immediately preferable (not even if considered as a whole) because pheromone trails are still negligible on this path. In particular, the pheromone trail at each individual edge is also negligible, so *all* edges of this path are still far away from being preferable compared to their respective competitors. Thus, the reinforcement of shorter paths is faster in RFD than in ACO. Finally, the erosion process provides a method to avoid inefficient solutions because sediments tend to be cumulated in blind alleys (in our case, in *valleys*). These nodes are filled until eventually their altitude matches those of adjacent nodes, i.e., the valley disappears. This differs from typical methods to reduce pheromone trails in ACO: Usually, the trails of *all* edges are periodically reduced at the same rate. On the contrary, RFD intrinsically provides a *focused* punishment of bad paths where, in particular, those nodes blocking alternative paths are modified.

Let us consider the applicability of RFD to `MSV` and `MDV` (its applicability to `TSP` will be considered below). These problems consist in finding a kind of *combination* of short paths, in particular a tree. After executing RFD for some time, for each node we take the edge with the highest gradient, and we discard the rest of edges. This guarantees that selected edges form a tree: If the formed subgraph includes two or more paths to go from A to B then, for at least one node, there are two

[1] In fact, each drop still needs memory to know the amount of sediments it is carrying. This is a single value so, for each drop, the size of the required memory is in $\mathcal{O}(1)$. On the contrary, if drops were required to record the previously traversed path, then the required memory would be in $\mathcal{O}(n)$, where n is the number of nodes of the graph.

or more outgoing edges (which contradicts the fact that, for each node, only the edge providing the highest gradient is taken). As discussed before, natural rivers do not tend to form solutions where each drop goes to the sea through its shortest path, but they tend to form grouped solutions. This allows RFD to implicitly deal with *path conflicts*, i.e. situations where, at a given node, two drops coming from different origins have different preferences regarding which edge should be taken next (because costs are different for each of them; recall that, in `MSV` and `MDV`, we are considering that costs depend on previously followed paths). In these situations, the tendency of RFD to form grouped solutions implicitly leads to forming paths with a suitable cost tradeoff between available choices: After some steps, the erosion will reinforce more strongly the slopes providing the lowest overall cost. In addition, in the *minimum spanning tree* problem (`MSV`), the tendency of drops to join each other is very appropriate: If drops tend to join the main *flow*, instead of following their respective individual shortest paths, then less edges are added to the tree and the tree cost is reduced.

Let us note that the tendency of ACO methods to form grouped solutions is well known, so similar arguments can be given in the case of ACO. In particular, ACO allows to form short paths from a single node to a single destination. However, combining some short paths departing from different points in such a way that a *tree* is formed is not a natural task for ACO. Let us suppose that two paths coming from different origins join at a given node and then continue together.[2] Ants coming from a departure node can be confused by pheromone trails and go on to the *other* departure node, instead of following to the destination node. Solving this problem requires to use some artificial methods (e.g., using different types of pheromones, using *directed* pheromones, associating ants to specific areas, etc). On the contrary, edge gradients formed by RFD are *intrinsically* directed, and their direction naturally leads to the destination node. This eases the task of constructing trees in RFD. Interestingly, we can adapt RFD to the *minimum distances tree* problem (`MDV`) just by changing a parameter: If we reduce the erosion caused by *high flows*, then the incentive of drops to join each other is partially reduced, and thus each drop tends to follow its own shortest path. For instance, we can achieve this effect by changing the erosion rules in such a way that, if n drops traverse an edge, then they make the effect of e.g. a single drop. In this case, grouped paths are promoted by the method only when they are required to solve path conflicts. Moreover, by considering *intermediate* erosion effects, we can construct trees partially fitting into the objectives of both problems (i.e., a combination of minimum spanning tree and minimum distances tree). This may be a suitable choice for several optimization problems.[3]

Regarding the third NP-complete problem considered in this chapter, `TSP`, let us note that some of the previous considerations apply to this case as well. In particular, the mechanism of focalized punishment of inefficient paths, the avoidance of local

[2] That is, the convergence area reminds the form of a ‘Y’ letter.

[3] For instance, we design a subway network to carry citizens from different areas to downtown in such a way that (a) the time spent by citizens to arrive to downtown is minimized (i.e., we need a minimum distances tree), and (b) the expenses required to build tunnels are minimized (i.e., we need a minimum spanning tree).

cycles, and the fast reinforcement of shorter paths are also good for solving TSP. However, facing this problem requires slightly modifying the general RFD scheme. In particular, let us note that TSP requires finding a *cycle* in the graph, but RFD implicitly avoids cycles. The adaptation of RFD to solve TSP will be addressed later in Section 2.2.

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
```

This 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(s). Next, all nodes of the graph are initialized (`initializeNodes()`). This consists of two operations. On 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 departing from the same initial nodes traverse the same sequences 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 of 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:

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

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

$$decreasingGradient(i,j) = \frac{altitude(j) - altitude(i)}{distance(i,j)} \tag{2}$$

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 of a drop moving through an edge with 0 gradient is set to some (non null) value. This enables drops to spread around a flat environment, which is required, 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 inversely proportional to the increasing gradient, and it will be reduced during the execution of the algorithm by using a method similar to the one followed by *Simulated Annealing* (see [12, 8]). This new feature improves the search of good paths. Let us note that solutions found during the first steps tend to bias the exploration of the graph afterwards. This is because previously formed paths tend to be followed by subsequent drops. Enabling drops to climb increasing slopes with some low probability allows us to find alternative choices and enables the exploration of other paths in the graph. This partially decouples the method from its behavior in the first steps. Actually, the probability of climbing increasing slopes encapsulates most of the dependency of the method on previous solutions in a single value. As usual in heuristic search algorithms, this dependency must provide a suitable tradeoff between past solutions and alternative choices. Let us note that allowing climbing up gradients does not invalidate the argument that local cycles are avoided in practice in our method: After following a sequence of downward gradients, completing a cycle requires to climb up all the altitude lost so far, and the probability of climbing up a high gradient is negligible.

The climbing up mechanism works as follows. Given a drop d located at node k, we randomly decide whether d can climb upward gradients according to the following probability:

$$P(d) = \frac{1}{notClimbingFactor} \tag{3}$$

where *notClimbingFactor* is a variable that is initially set to 1 and is slightly increased after each loop iteration. Every N iterations, this variable is not increased, but *decreased* (this is done to emulate the *tempering* process used in Simulated Annealing). When the drop is *allowed* to climb up and it decides its next move, it can also choose moving upwards – as well as moving downwards or going through a null gradient, as the rest of drops can. In particular, the transition rule of a climbing drop is defined as follows:

$$P_k(i,j) = \begin{cases} \frac{decreasingGradient(i,j)}{total} & \texttt{if} \ \ j \in V_k(i) \\ \frac{\omega/|decreasingGradient(i,j)|}{total} & \texttt{if} \ \ j \in U_k(i) \\ \frac{\delta}{total} & \texttt{if} \ \ j \in F_k(i) \end{cases} \tag{4}$$

where

$$total = \left(\sum_{l \in V_k(i)} decreasingGradient(i,l) \right) + \sum_{l \in U_k(i)} \left(\frac{\omega}{|decreasingGradient(i,l)|} \right) + \sum_{l \in F_k(i)} \delta$$

and $V_k(i)$, $U_k(i)$ and $F_k(i)$ are the sets of nodes that are *neighbors* of node i that can be visited by the drop k and are connected to k through a down, up, and flat gradient, respectively. We consider that δ and ω are parameters of the algorithm. On the other hand, if a drop fails to be considered as a climbing drop then it only considers taking down or flat gradients.

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. The reduction of the altitude of this node depends on the current gradient between A and B. In particular, the erosion is higher if the downward gradient between A and B is high. The altitude of the eroded node A is modified as follows:

$$altitude(A) := altitude(A) - erosion(A,B)$$

$$erosion(A,B) = \frac{paramErosion}{(numNodes - 1) \cdot numDrops} \cdot decreasingGradient(A,B)$$

where *paramErosion* is a parameter of the erosion process, *numNodes* is the number of nodes of the graph, and *numDrops* is the number of drops used in the algorithm. On the contrary, if the edge is flat or increasing then a small erosion is performed. However, the altitude of the final node (i.e., the *sea*) is never modified and it remains equal to 0 during the execution.

Once the erosion process finishes, the altitude of all nodes of the graph is slightly increased (`depositSediments()`). The objective is to avoid, after some iterations, the erosion process leading to a situation where all altitudes are close to 0, which would make gradients negligible and would ruin all formed paths. In particular, the altitude of a node N is increased according to the following expression:

$$altitude(N) := altitude(N) + (erosionProduced/(numNodes - 1))$$

where *erosionProduced* is the sum of erosions introduced in all graph nodes in the previous phase, and *numNodes* is the number of nodes of the graph.

We also enable individual drops to *deposit* sediment on nodes. This happens when all movements available for a drop imply climbing 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 is transporting. This increases the altitude of the current node in proportion to the cumulated sediment carried by the drop, which in turn is proportional to the erosions produced by the drop in previous movements. If a drop gets blocked at node N, then the altitude of N is increased as follows:

$$altitude(N) := altitude(N) + paramBlockedDrop \cdot cumulatedSediment$$

where *paramBlockedDrop* is a parameter and *cumulatedSediment* is the amount of sediment carried by the drop.

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

Some additional improvements can be introduced to this basic scheme (see [17] for details). For instance, we can gather drops to reduce the number of individual movements, we can consider the formation of *lakes* in blind alleys, etc.

2.2 Adapting RFD to TSP

Though the basic RFD scheme is suitable for solving `MSV` and `MDV` as it is, some adaptations are required to apply RFD to `TSP`. We present them in this section (further details can be found in [17]). We can define `TSP` as follows: Given a set of cities and the costs of traveling from any city to any other city, compute the cheapest round-trip route that visits each city exactly once and then returns to the starting city. Note that, since we are looking for a cyclic tour, it is irrelevant what concrete node is the origin of the tour: The cycle A-B-C-D-A has the same length as B-C-D-A-B or C-D-A-B-C.

The adaptation of our method to `TSP` has several similarities with the way ACO is applied to this problem. In ACO, endowing ants with the capability of *remembering* all nodes traversed so far is necessary to avoid repeating nodes. Let us recall that the use of *gradients* strongly minimizes these situations in RFD. In particular, only sequences of low-probability climbing movements can lead drops to follow local cycles.[4] Thus, in RFD memories are not as useful as in ACO, and not using them is preferable in the general case because less operations are performed (e.g., this is the case in `MSV` and `MDV`). However, in `TSP` memories are actually required to identify round-trips: Any path not including all nodes is *not* a round-trip, and so it should not be reinforced. Thus, drops must be endowed with memory indeed.

Another difference with the general scheme of RFD is that altitudes are not eroded after each individual drop movement. On the contrary, altitudes of all traversed nodes are eroded all together when the drop actually completes a round-trip (as ants actually 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 less probable in RFD than in ACO). In RFD, there is an additional reason 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. Let us note that in this case the computations performed to move the drop should not be considered as *lost* despite the fact that it did not find a solution. This is because the cumulation of sediments increases the node altitude, and thus gradients coming from adjacent nodes are flattened. This eventually avoids that other drops fall in the same *blind alley*.

[4] This may happen e.g. at the earliest steps of the algorithm, where the environment is still almost flat.

Other peculiarities of the adaptation of RFD to TSP are specific to RFD and do not appear in ACO. Our method provides an intrinsic method to avoid drops traversing 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* fixed 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 introducing other additional nodes for different technical reasons. Let us suppose that a solution A-B-C-D-E-A' is found (A' being the clone of A playing the role of *destination* node). Drops create a decreasing gradient along this path. In particular, the altitude of B is higher than the altitude of C, C is higher than D, and D is higher than E. Let us also suppose that there exists an edge from B to E. Since the difference of altitude between B and E is the addition of the differences between B and C, C and D, and D and E, the decreasing gradient from B to E could be so big that drops *prefer* to go directly from B to E. However, in that case drops will fail in finding a solution: C, D, and E would not be included in this path, but solutions must traverse all nodes. In order to avoid the altitude of adjacent nodes wrongly deviating paths, an auxiliary node will be created at each *edge* of the graph. These new nodes, called *barrier* nodes, are introduced as follows: If there is an edge connecting (standard) nodes X and Y, this edge is replaced by an edge connecting X and a new *barrier* node xy, as well as another edge connecting xy and Y. If a drop traverses all standard nodes (i.e., it finds a solution) then barrier nodes traversed in the path are eroded exactly as standard nodes are. For instance, let us consider the solution A-B-C-D-E-A' given in the previous example. This solution actually traverses nodes A-ab-B-bc-C-cd-D-de-E-ea'-A', ab being the barrier node appearing between A and B, and so on. These barrier nodes will be eroded when finding this solution. However, the barrier node between B and E, be, is not eroded by drops following this path. In fact, if a drop moves directly from B to E and next moves to A', then it will not find a solution, so node be will not be eroded by this alternative path. Thus, the altitude of node be will remain high and drops at B will not prefer moving to be. That is, be imposes a *barrier* between B and E.

Let us note that barrier nodes must be taken into account in the initialization and sedimentation phases of the algorithm. In the initialization phase, we must set the height of barrier nodes. This height will be the same as the height of the rest of nodes of the graph. Regarding the sedimentation phase, it will be necessary to increase their heights in the same way as any other node.

3 Formal Definition of MSV and MDV

Though the TSP problem is well-known, a precise definition of the other two problems considered in this chapter, MSV and MDV, is required. In this section we formally define them, and we briefly consider their applicability to some computational problems. As we said in the introduction, finding a minimum spanning tree (i.e. a

tree embracing all nodes where the addition of costs of all edges in the tree is minimal) or a minimum distances tree (i.e. a tree where the addition of distances from each node to the exit node through edges in the tree is minimal) are polynomial time problems, and thus heuristic methods are not necessary in this case. However, if we assume that the cost of including an edge in the tree depends on the *rest* of edges included in the tree, then these problems are not that simple. In particular, let us consider that the cost of an edge depends on the path we have to traverse in the tree before taking it. More technically, we assume that the cost of a path of edges $e_1,\ldots,e_n$ from a given origin node o to a given destination node d depends on the evolution of a *variable* through the path. Initially, a value v_o is assigned to this variable at node o. Then, the cost added to the path due to the inclusion of edge e_1 is an amount depending on v_o. After traversing e_1, the value of the variable is updated to a new value v_1. Next, the cost of adding e_2 to the path depends on v_1. After taking e_2, the value of the variable is updated again, and the process continues so on until we obtain the whole cost of the path $e_1,\ldots,e_n$.

Following this idea, we can define a variable-cost graph by attaching some information to a standard graph. Let us consider a set of *origin* nodes (in particular, this set could include *all* nodes of the graph). Then, (1) we assign an *initial value* to each origin node; (2) we assign a *cost function* to each edge. Depending on the value of the variable just before traversing the edge, taking the edge adds a different cost; and (3) we assign a *transformation function* to each edge. Given the value of the variable before traversing the edge, it returns the new value after taking it.

Let us suppose that a variable-cost graph defined in these terms is given. On one hand, a *minimum distances tree* is a tree connecting each origin node with the destination node in such a way that the addition of costs of all *paths* from each origin node to the destination is minimal. Since the returned solution is a tree, paths departing from different origin nodes could share some edges (in particular, different sequences of edges could share some suffixes). Let us note that, in general, the cost of a shared edge is different for each path because the value of the variable when the edge is reached may be different for each path. On the other hand, a *minimum spanning tree* is a tree connecting all origin nodes with the destination node in such a way that the addition of costs of all *edges* included in the tree is minimal. In this case, the cost of an edge e in a tree t is computed as follows. Let us consider all the paths of t connecting an origin node with the destination node and including edge e. The cost of e in t is the *average* of the cost of e for all of these paths. Let us note that, in both problems, trees are not required to include all nodes from the original graph, but only those actually used to connect origin nodes to the destination node. In particular, if all nodes are considered origin nodes then the resulting tree must include all nodes indeed.

Definition 1. A *variable-cost graph* is a tuple $G = (N,O,d,V,A,E)$ where:

- N is a finite *set of nodes*,
- $O \subseteq N$ is the set of *origin nodes*,
- d is the *destination node*,
- $V = \{v_1,\ldots,v_n\}$ is a finite set of *values*,

- $A : O \longrightarrow V$ is the *initial value function*, that is, a function assigning an initial value to each origin node.
- E is the *set of edges*. Each edge $e \in E$ is a tuple (n_1, n_2, C, T) where $n_1, n_2 \in N$ are the *origin* and *destination* nodes, respectively, and
  - $C : V \longrightarrow \mathbb{N}$ is the *cost function* of e. Given a value in V denoting the current value of the variable, it returns the cost of traversing e.
  - $T : V \longrightarrow V$ is the *transformation function* of e. Given the current value of the variable, it returns the new value assigned to the variable if e is traversed.

Paths are sequences of edges departing at an origin node and arriving to the destination node. Formally, a path of G is a sequence of edges $\sigma = (e_1, \ldots, e_k)$ with $e_i = (n_i, n'_i, C_i, T_i) \in E$ for all $1 \leq i \leq k$ such that $n_1 \in O$, $n'_k = d$, and for all $1 \leq i \leq k-1$ we have $n'_i = n_{i+1}$. The *cost* of σ, denoted by $c(\sigma)$, is equal to

$$C_1(A(n_1)) + C_2(T_1(A(n_1))) + C_3(T_2(T_1(A(n_1)))) + \ldots + C_k(T_{k-1}(\ldots(T_2(T_1(A(n_1))))\ldots))$$

The term denoting the cost of traversing e_i in the previous expression, that is $C_i(T_{i-1}(\ldots(T_2(T_1(A(n_1))))\ldots))$, will be denoted by $c_{e_i}(\sigma)$. In a notation abuse, we will write $e \in \sigma$ if $e = e_i$ for some $1 \leq i \leq k$.

We say that $G' = (N', O, d, V, A, E')$ with $N' \subseteq N$ and $E' \subseteq E$ is a *tree* of G if for all $o \in O$ there exists a single path $\sigma = (e_1, \ldots, e_k)$ of G' departing from o, that is, such that $e_1 = (o, n, C, T)$ for some n, C, T. For each $o \in O$, we denote by σ_o the unique path of G' departing from o.

The *distances cost* of G', denoted by $dc(G')$, is equal to $\sum_{o \in O} c(\sigma_o)$. The *spanning cost* of G', denoted by $sc(G')$, is equal to $\sum_{e' \in E'} \frac{\sum\{c_{e'}(\sigma_o) | o \in O, e' \in \sigma_o\}}{|\{c_{e'}(\sigma_o) | o \in O, e' \in \sigma_o\}|}$. □

Now we are provided with all the needed machinery to formally define the problems considered in this chapter. As it is usual in Complexity theory, these minimization problems are defined in terms of their equivalent decision problems.

Definition 2. The problem of the *minimum distances tree for a variable-cost graph*, denoted by `MDV`, is stated as follows: Given a variable-cost graph G and a natural number $K \in \mathbb{N}$, is there any tree G' of G such that $dc(G') \leq K$?

The problem of the *minimum spanning tree for a variable-cost graph*, denoted by `MSV`, is stated as follows: Given a variable-cost graph G and a natural number $K \in \mathbb{N}$, is there any tree G' of G such that $sc(G') \leq K$? □

The previous problems generalize the classical *minimum spanning tree* and the *minimum distances tree* problems to the case where the cost of traversing each edge depends on the path traversed before taking the edge. The past path is abstracted by the *value* of the variable, which particularizes the cost of each edge for each path. Let us note that, in formal terms, we do not need to consider *several* variables in the problem definition because the dependence on past paths can be denoted by using a single variable. Though several variants of the minimum spanning tree and the minimum distances tree problems have been studied in the literature, as far as we are concerned the variant problems proposed in this chapter have not been considered. Hence, their properties must be analyzed. A proof of the NP-completeness

of both problems is presented in the appendix of this chapter. In fact, variants of both problems where only graphs fulfilling $N = O$ are considered (that is, where *all* nodes are origin nodes) are NP-complete as well. These alternative problems are also considered in the appendix.

As a matter of fact, the generalization introduced in MDV and MSV (that is, the use of variable-cost graphs instead of fix-cost graph) increases their applicability to new interesting scenarios. In fact, we recently came across these problems because we were constructing some *testing derivation algorithms* (for an introduction to *Formal Testing Techniques*, see e.g. [14, 16, 2, 15, 20]). Typically, the goal of a testing methodology is to interact with the analyzed system so that all system states are reached *at least once*. If previous system configurations can be *restored* then we can explore a part of the system, then go back to a previously traversed point, and next go on through a different way. Thus, the problem of reaching all states consists in creating a tree embracing all states. Since the time required to go from state s to state s' depends on the previous activities of the system (available resources, values of variables, etc), composing the optimal tree reaching all states at least once requires taking past activities into account. We can use a variable-cost graph to denote how the execution time of each activity depends on the current values of variables. Thus, if we assume that previous configurations can be *restored* then finding a tree which allows reaching all states in minimum time essentially consists of solving MSV for this graph. There exist other related testing problems whose basic structure fits into this problem scheme as well.

Next we consider an applicability example of MDV. Let us consider that a local area network (LAN) is constructed on top of a given existing networking infrastructure. The transmission cost of a given connection (i.e. *edge*) depends on the kind of information being transmitted (e.g., low connections are unacceptable for a real-time video stream, but may be suitable for low priority packets). Besides, the kind of information being transmitted depends on the kind of sender machine. Thus, a variable-cost graph can be used to define communication costs in the existing infrastructure. Let us suppose that we want to design a networking tree that allows all nodes to communicate with a central dispatcher in such a way that average communication costs are minimized. Finding this tree consists of solving MDV. Other applicability scenarios of MSV and MDV may be considered as well (for instance, this is the case of the *subway design problem* we briefly sketched before).

4 Applying RFD and ACO to MDV, MSV, and TSP

In this section we describe the application of our approach to solve MDV, MSV, and TSP, and we report some experimental results. Only static graphs will be considered in this section. Experiments where nodes and edges appear/disappear along time will be considered in the next section. In both sections, we compare the results obtained by using ACO methods and the solutions obtained by using our method. All experiments were performed in an Intel Core Duo T7250 with a 2.00 GHz processor. The basic aspect of our application interface can be seen in Figure 1 (left);

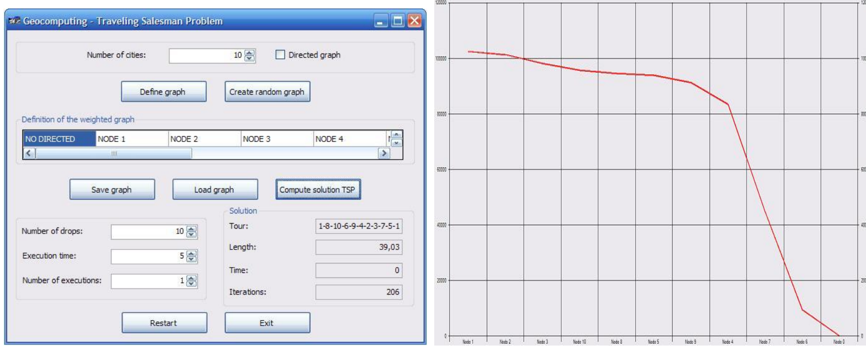

Fig. 1 Application interface (left) and altitudes in a TSP solution (right)

the picture on the right shows the altitudes of each of the nodes after solving an instance of `TSP`.

Next we introduce the values of parameters of RFD and ACO used in these experiments. Parameters depend on the problem we are executing. The number of drops and ants used to solve `MDV` and `MSV` was detected not to be very significant; they are set to 10 in both cases. However, we observed that these values did influence results when `TSP` was solved. In this case, the number of drops is set to 50 and the number of ants is 1000. In RFD, the initial altitude of the nodes is 1000 in the case of `MDV` and `MSV`, and 10000 in the case of `TSP`. Furthermore, parameters *paramErosion* and *paramBlockedDrop* (see Section 2.1) are set to 1 for all problems, and δ and ω are set to 1 and 0.1. The parameter *notClimbingFactor* is increased by 0.01 after every loop iteration, and after every 100 iterations it is decreased by 0.5. For ACO, the initial amount of pheromone in edges is set to 1000 and the amount of pheromone deposited by an ant after a movement is 100. The evaporation rate is set to a standard value, 0.5. The α and β parameters, considered as in [7], are set to 4 and 2, respectively, when `MDV` and `MSV` are considered, and they are equal to 1 and 5 when `TSP` is solved.

4.1 Static MSV

Next we present the results obtained when `MSV` is solved by both methods. In the case of RFD, we have directly applied the method presented in Section 2, while in the case of ACO we have used an implementation inspired by [3]. Three randomly generated variable-cost graphs with 100, 200, and 300 nodes were considered.[5] In these graphs, each node is connected to approximately 40% of the rest of nodes. Variables can take up to 10 possible values. Cost functions and transformation functions attached to edges are randomly generated. In particular, features such as

[5] All graphs used in experiments in this paper can be downloaded from `http://kimba.mat.ucm.es/~prabanal/`.

monotonicity or injectivity are not required in these functions. Figure 2 shows the results of an experiment where the input of both algorithms was the graph with 100 nodes. The graph shows the cost of the solution found by each algorithm for each execution time (in seconds). Analogously, Figure 3 contains the results obtained by using the graph with 200 nodes as the input of both algorithms, while Figure 4 shows the results for the 300 nodes graph. All figures show the evolution of the algorithms in a *single* execution, but the same basic shape has been obtained for most of the executions. In order to report solutions that are not biased by a single execution, each algorithm was executed thirty times for each of the graphs. Table 1 summarizes the arithmetic mean (Avg), the variance (Var), and the best solution (Best) found by each method among all thirty executions.

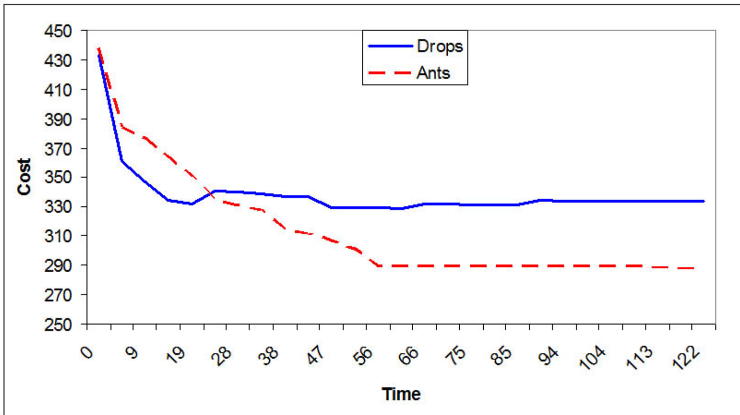

Fig. 2 `MSV` results for a randomly generated variable-cost graph with 100 nodes

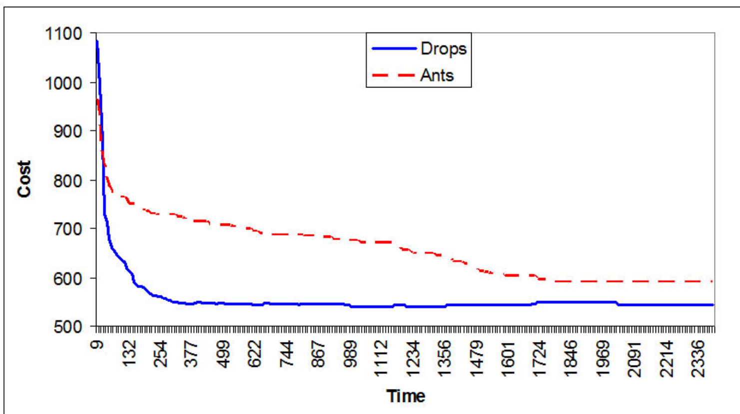

Fig. 3 `MSV` results for a randomly generated variable-cost graph with 200 nodes

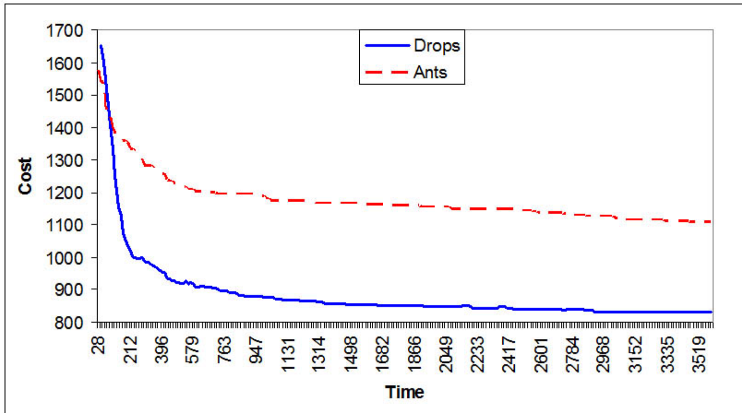

Fig. 4 MSV results for a randomly generated variable-cost graph with 300 nodes

Table 1 Summary of MSV results. Static case.

Graph size	Avg RFD	Avg ACO	Var RFD	Var ACO	Best RFD	Best ACO
100 nodes	300.20	273.30	193.73	17.33	262.75	263.03
200 nodes	538.76	577.54	160.33	365.01	506.36	542.32
300 nodes	829.22	1128.73	128.40	641.79	802.44	1069.99

The results presented in the previous table show that average solutions found by the RFD method are better than the solutions found by ACO. Moreover, the variance is also lower in our algorithm than in the ACO algorithm, with the only exception of the smallest graph.

4.2 Static MDV

Next we study the application of ACO and RFD to solve MDV. Let us recall that changing a single parameter which defines the erosion caused by drops, makes RFD solve either MSV or MDV: If n drops traversing an edge per unit of time erode the ground, as if the erosion effect of a single drop were multiplied by n, then constructed trees will approximately solve MSV. However, if n drops have the effect of a *single* drop, then trees tend to solve MDV.

The comparison results obtained for the MDV problem are similar to the case of MSV. That is, RFD again obtains better solutions (on average), but solutions provided by ACO in short times are better. Times required by RFD to surpass ACO solutions are now a little bit longer. Figures 5, 6, and 7 show the results of experiments performed with 100, 200, and 300 nodes graphs. Again, in all these cases the figures show the evolution of the algorithms in a *single* execution. Following the same methodology as in the previous example, each algorithm was executed thirty

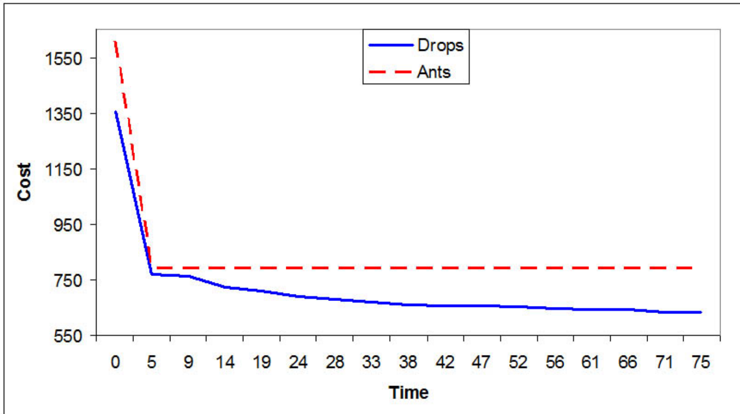

Fig. 5 MDV results for a randomly generated variable-cost graph with 100 nodes

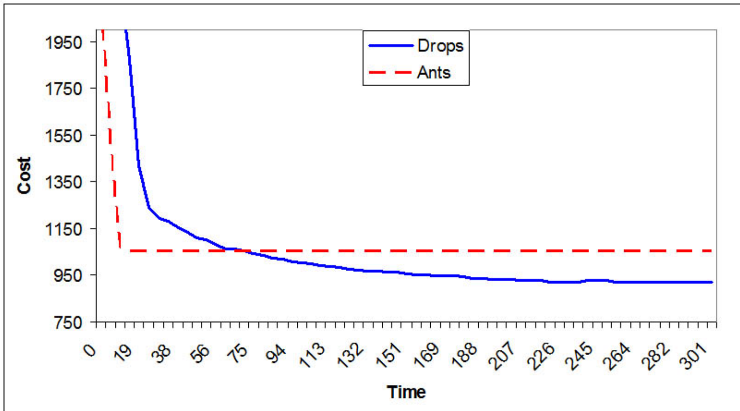

Fig. 6 MDV results for a randomly generated variable-cost graph with 200 nodes

Table 2 Summary of MDV results. Static case.

Graph size	Avg RFD	Avg ACO	Var RFD	Var ACO	Best RFD	Best ACO
100 nodes	613.81	711.90	60.92	4026.26	600.14	587.52
200 nodes	884.05	933.20	111.41	5275.66	854.78	771.64
300 nodes	1362.05	1414.31	244.50	8804.20	1330	1239.73

times for each of the graphs. Table 2 summarizes the same parameters as those considered before for MSV.

As in the case of MSV, the results presented in the previous table show that average solutions found by the RFD method are again slightly better than those found by ACO. Moreover, the variance is lower again. However, now RFD needs more time

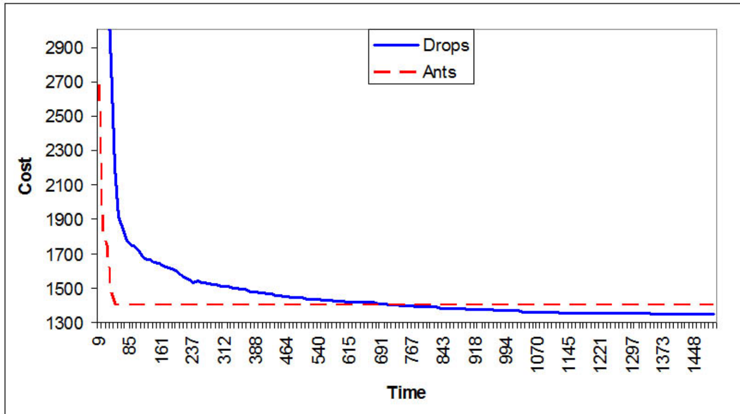

Fig. 7 MDV results for a randomly generated variable-cost graph with 300 nodes

to surpass the quality of the solutions found by ACO. In fact, despite the fact that the mean of solutions is better in RFD than in ACO, the best solution found in all the thirty executions is usually found by ACO. Thus, although the results obtained by RFD are good, the advantage over ACO is not as significant as in the case of MSV. The main reason is that RFD intrinsically promotes the formation of *grouped* paths instead of individual paths, which reduces the size of constructed trees (as required by MSV). In particular, let us note that the incentive of drops to join the main flow is *structurally* provided by the use of *gradients* in RFD: If a drop d moving at a high altitude falls into a strongly eroded flow then, due to the fast reinforcement of shorter paths in RFD, subsequent drops traversing the same area as d will quickly tend to join this flow as well. This feature helps RFD to properly solve MSV (but it is not that useful for solving MDV).

4.3 Static TSP

We apply RFD and ACO to solve some instances of TSP, and we report the collected results. Let us recall that handling TSP by means of RFD requires taking into account the adaptations previously sketched in Section 2.2.

Figures 8 and 9 show the results of two experiments. The first one shows the results for a graph taken from the TSPLIB library [19]. TSPLIB is a library of sample instances for the TSP that has become a standard benchmark for this problem. Nodes are defined by means of points in a 2D plain, and there is an edge between all pairs of nodes. Thus, fully connected graphs are represented. The distance of each edge is the euclidean distance between both points of the plain.

Figure 9 contains a larger case example where we consider a graph of 100 nodes. This graph has been randomly generated assuming that each node is connected with only a few other nodes (between 10 and 20 connections per node) as it is the most common case in complex networks. The basic shape shown in the figures is also obtained with other benchmark examples.

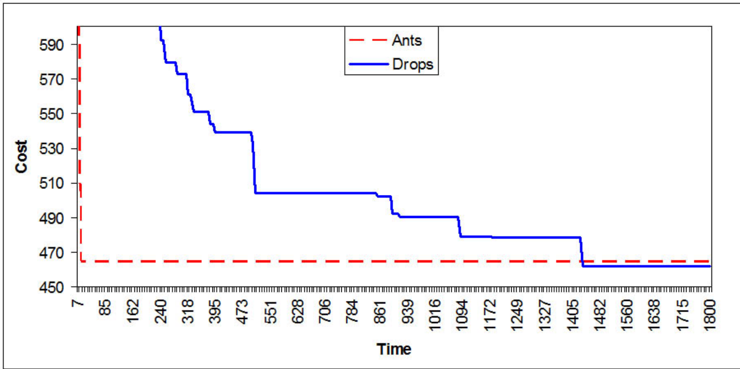

Fig. 8 TSP results for the TSPLIB eil51 graph

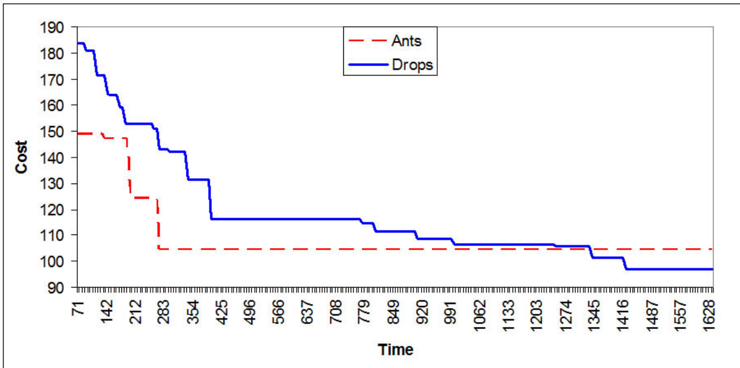

Fig. 9 TSP results for a 100 nodes graph

Table 3 Summary of TSP results. Static case.

Graph size	Avg RFD	Avg ACO	Var RFD	Var ACO	Best RFD	Best ACO
eil51	458.08	457.97	51.19	3.05	441.9	454.42
100 nodes	101.36	102.61	197	12.06	84.15	96.55

Note that the shapes shown in these figures are analogous to the case of MSV and MDV: ACO finds solutions faster, while RFD finds better solutions after some time. However, in this case it seems harder for RFD to surpass ACO solutions. The main reason is that drops need memory to register the traversed path in TSP, so one of the advantages of RFD with respect to ACO (that is, the absence of memories) is lost.

Following the same methodology as in the previous example, each algorithm was executed thirty times for each of the graphs. Table 3 summarizes the same parameters as in previous tables.

4.4 Summarizing Results

We extract the following conclusions from the experimental results obtained for the three considered problems. In all problems, we observe that ACO usually provides good solutions earlier than RFD. However, after some additional time passes, the solutions provided by RFD usually surpass the quality of those given by ACO. 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. First, let us note that the erosion-sedimentation mechanism of RFD provides a dynamic dual mechanism to promote/punish good/bad paths or parts of paths. This process allows not only to locally reinforce good parts of paths, but also to locally punish bad sequences. This is an active *try and fail* mechanism which enables a more exhaustive exploration of the graph. Besides, as we said before, the construction of paths of decreasing altitudes implicitly avoids the formation of local cycles, which in turn avoids inefficient movements of drops. Moreover, this forces drops not to be concentrated in local areas but to spread around the graph, which allows a wider and more homogeneous search. In addition, let us recall that the use of gradients makes *shorter* paths preferable overall from the time they are discovered, which accelerates the process of reinforcing them. This increases the competitiveness of shorter paths: At a given time, *several* new shorter paths may have attractive gradients, even if they are still young. This helps new shorter paths strongly compete with each other, which again enables a deeper exploration of the graph.

The previous considerations are common to all three problems, though the differences of RFD and ACO is affected by an additional factor in the cases of `MSV` and `MDV`. As we said before, when RFD solves `TSP`, drops keep a memory of previously traversed nodes, though no memory is used in RFD when `MSV` or `MDV` are solved. This contrasts with ACO, where memories are used in the three problems. The implicit avoidance of cycles of RFD allows us not to provide drops with any memory in `MSV` and `MDV`, but ants still need memories in both problems because they can fall in cycles indeed. Though cycles are avoided by formed gradients in RFD, gradients are still weak during the first steps of the algorithm. Thus, drops *do* follow some cycles during these early steps. In the long term, when gradients are stronger, drops avoid cycles without maintaining any memory structure, which boosts their performance with respect to ACO. Thus, the absence of memory in `MSV` and `MDV` also promotes the general characteristic we already pointed out before: Solutions given by ACO are better in the short term, but RFD surpasses solutions given by ACO after some time.

5 Applying RFD and ACO to MDV, MSV, and TSP in Dynamic Graphs

In this section we reconsider the previous problems in a scenario where graphs are *dynamic*, that is, nodes and edges can change along time. In particular, we are interested in studying the capability of each method to *adapt* previous solutions to

new configurations after each change is introduced. In these experiments, we follow the following procedure for each of the considered problems. First, we calculate the solution of an instance of the problem by using either ACO or RFD. Next, we introduce one or more 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 the 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 most cases we observe that, after some time, the quality of the solutions of our method surpasses the quality of the solutions provided by ACO. In fact, the time needed to surpass ACO solutions is shorter than in the static case. Moreover, it is specially remarkable that for some graphs ACO does not find a solution at all after changing the graph (but, in all of these cases, RFD does).

5.1 Dynamic MSV

Let us start by considering the `MSV` problem. As we did in the static case, we will compare the performance of RFD and ACO by using the same input graphs. In particular, we consider the same graphs introduced in the static case, and in each experiment we introduce several changes at the same time. In the case of the 100 nodes graph, we remove two nodes, we remove two edges (one of them belongs to the solution found by ACO and another one to the solution found by RFD), and we add other two new nodes. The results can be seen in Figure 10. In the case of the 200 nodes graph (see Figure 11) we proceed in an analogous way: We remove four nodes as well as four edges (where two edges belong to the solution given by ACO and the other two edges belong to the solution provided by RFD), and we add

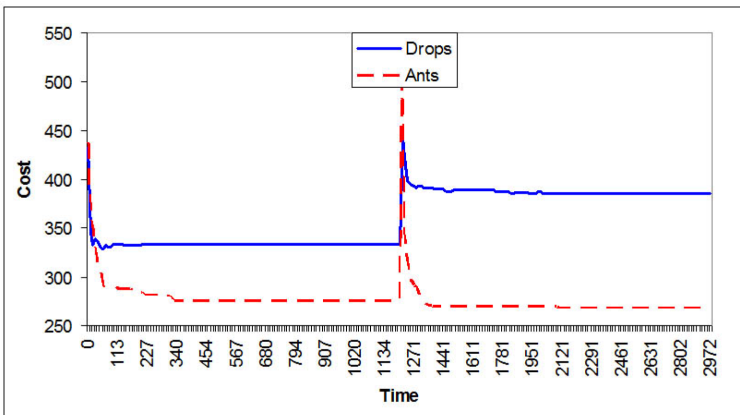

Fig. 10 `MSV` results for a dynamic randomly generated variable-cost graph with 100 nodes

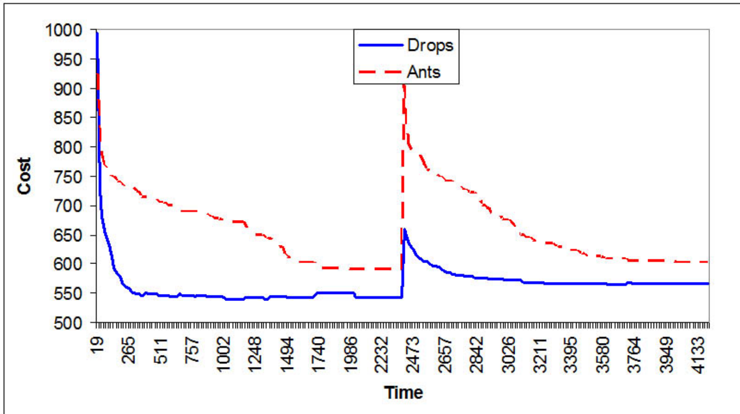

Fig. 11 `MSV` results for a dynamic randomly generated variable-cost graph with 200 nodes

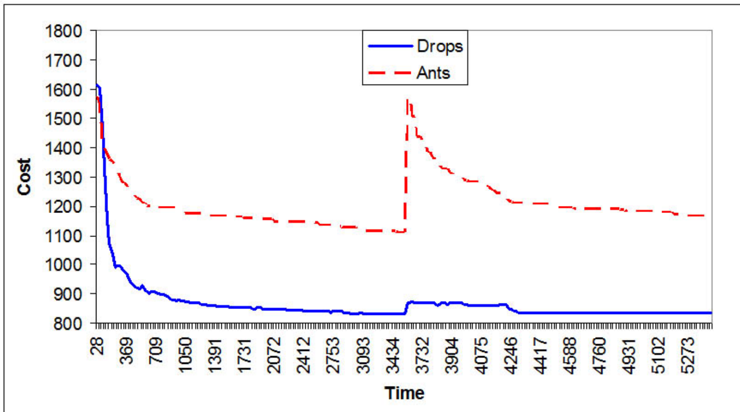

Fig. 12 `MSV` results for a dynamic randomly generated variable-cost graph with 300 nodes

two new nodes. We proceed analogously in the 300 nodes graph (see Figure 12): six nodes and six edges are removed, and next two new nodes are added.

In all cases we see that the time needed to react to the modifications is similar both in ACO and RFD. However, the results obtained by using RFD are better than those given by ACO in all graphs but the smallest one. Thus, the advantages of RFD in the static case are still valid in the dynamic case, while the main disadvantage of RFD in the static case is minimized in dynamic frameworks.

Table 4 summarizes observed results after executing each algorithm thirty times for each graph.

Let us remark that the results are somehow similar to those obtained in the static case. That is, ACO finds better solutions in the smallest graph, while RFD obtains better results in the larger graphs. Moreover, the time needed to react to the changes in both cases is very similar.

Table 4 Summary of MSV results. Dynamic case.

Graph size	Avg RFD	Avg ACO	Var RFD	Var ACO	Best RFD	Best ACO
100 nodes	383.53	265.08	11.72	14.23	376.45	258.68
200 nodes	559.15	602.90	3.87	172.93	556.94	578.41
300 nodes	835.10	1121.66	1.38	200.71	834.56	1058.86

5.2 Dynamic MDV

Next we analyze the performance of RFD and ACO when MDV is solved in environments where the graph changes along time. Again, we consider the same graphs introduced in the static case and, for each experiment, we introduce several changes at the same time. The changes introduced are the same as in the MSV case.

The results can be seen in Figures 13, 14, and 15, where experiments with the graphs of 100, 200, and 300 nodes, respectively, are considered. The situation now is similar to the MSV case: (i) the time needed to react to the modifications is similar both in ACO and in RFD; (ii) the results obtained using RFD are better than in ACO. In fact, we can see that RFD has a bit more advantage over ACO in the dynamic case than the static case.

Following the same methodology as in the previous case, each algorithm was executed thirty times for each of the graphs. Table 5 summarizes the same parameters as in previous tables.

Notice that in the dynamic case the advantage of RFD over ACO has increased with respect to the static case: The average results and the variance are better, and the best results are usually found by RFD. However, the main advantage with respect to the static case is that the time needed by RFD to surpass the quality of the solutions found by ACO is much less than in the static case. In fact, both RFD and ACO require similar times to react to changes.

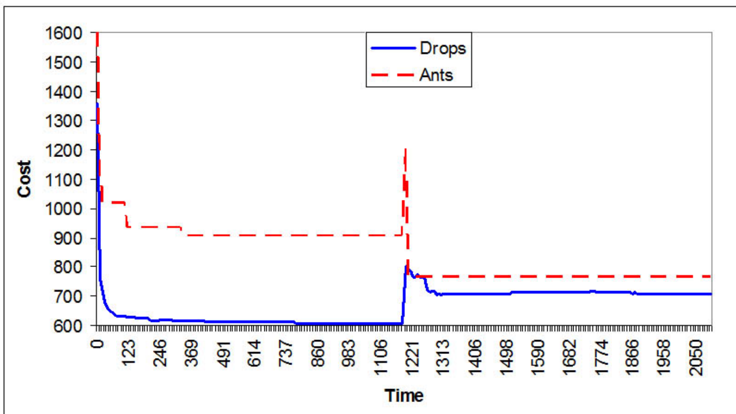

Fig. 13 MDV results for a dynamic randomly generated variable-cost graph with 100 nodes

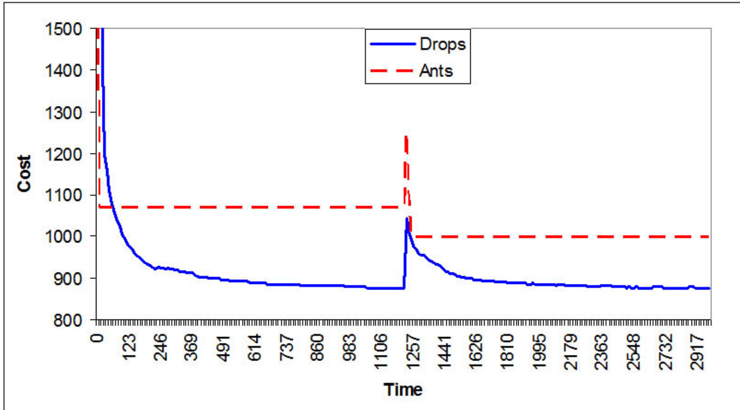

Fig. 14 MDV results for a dynamic randomly generated variable-cost graph with 200 nodes

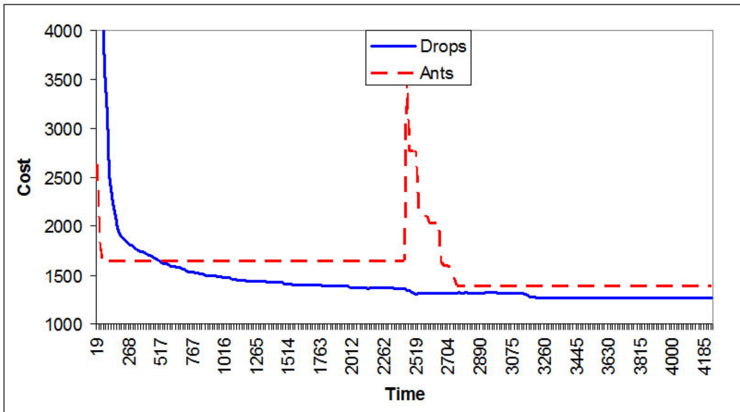

Fig. 15 MDV results for a dynamic randomly generated variable-cost graph with 300 nodes

Table 5 Summary of MDV results. Dynamic case.

Graph size	Avg RFD	Avg ACO	Var RFD	Var ACO	Best RFD	Best ACO
100 nodes	699.49	753.35	65.53	5216.30	681.69	618.25
200 nodes	878.02	1241.58	10.47	18653.47	873.37	1012
300 nodes	1264.53	1473.34	1.18	16106.36	1263	1275

5.3 Dynamic TSP

Next we use ACO and RFD to solve TSP in the case where changes are introduced in the graph. Figure 16 shows the results after removing one edge that was common to solutions found by both ACO and RFD, considering the 100 nodes graph.

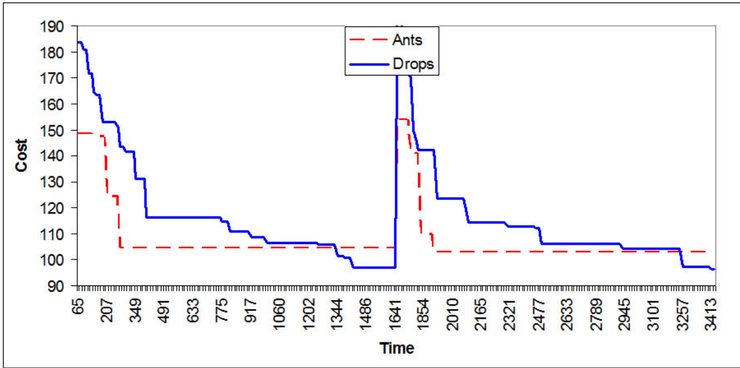

Fig. 16 `TSP` results for the 100 nodes graph, after removing an edge

Next, Figure 17 shows the results after removing one node in the same graph, and Figure 18 considers the case where several changes (in particular, deleting two common edges, adding two new edges, adding two new nodes, and deleting other two nodes) are introduced at the same time in the same graph. Next, Figure 19 shows the results after introducing a new node in the TSPLIB `eil51` graph. Finally, Figure 20 shows the results after introducing several changes (in particular, the same as in the case of the 100-nodes graph) at the same time in `eil51`.

Let us remark that, in one of these experiments, ACO did not find any solution to the problem after several modifications were introduced. That is, ACO did not find a way to either integrate the modifications within its previous solution or compose a completely new solution where the modifications were considered. 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. This is a good choice indeed, because the time needed by ACO to provide reasonable solutions when executed from scratch is actually shorter. However, this

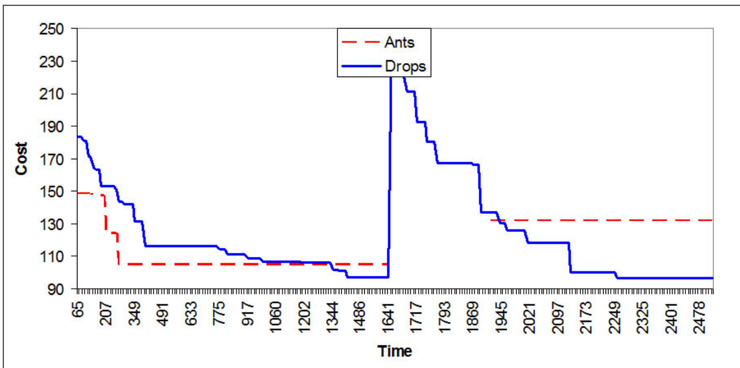

Fig. 17 `TSP` results for the 100 nodes graph, after removing a node

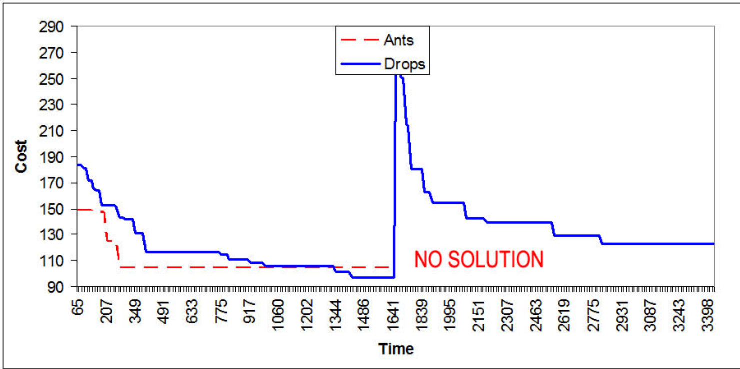

Fig. 18 `TSP` results for the 100 nodes graph, after performing several modifications together

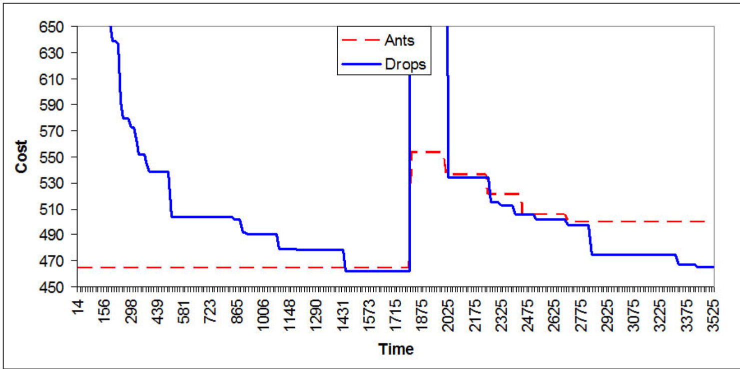

Fig. 19 `TSP` results for the TSPLIB `eil51` graph, after adding a node

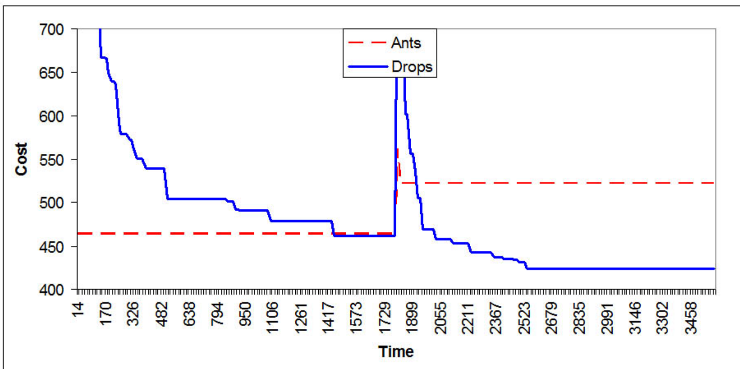

Fig. 20 `TSP` results for the TSPLIB `eil51` graph, after performing several modifications together

shows that the adaptability of ACO to changing landscapes (which is the point of considering the dynamic case) is worse than the adaptability of RFD.

Regarding the cases where ACO finds a solution after the modifications, we observe that ACO finds them faster than RFD, but RFD finds better solutions after some time. Moreover, it is worth pointing out that, in general, the time needed by RFD to surpass ACO is smaller after a modification is introduced than in the case where we compute the initial solution from scratch. However, the advantage of RFD over ACO in terms of quality of results was clearer in `MSV` and `MDV`. The reason in that `TSP` is harder for RFD to solve than `MSV` and `MDV`, as now drops need to keep memories of the path traversed so far. Anyway, it is remarkable that RFD can always react to changes in the graph, while that is not the case for ACO.

Next we study in higher detail the case where several changes are introduced at the same time, considering both the randomly generated graph of 100 nodes and the `eil51` graph from TSPLIB. Following the same methodology as in previous examples, each algorithm was executed thirty times for each graph. Table 6 summarizes the same parameters as in previous tables. The '–' symbol denotes that no solution was found in this case. Interestingly, ACO did not find any solution for the 100 nodes graph in all the thirty executions.

Table 6 Summary of `TSP` results. Dynamic case.

Graph size	Avg RFD	Avg ACO	Var RFD	Var ACO	Best RFD	Best ACO
eil51	436.40	441.76	57.39	28.63	427.68	436.35
100 nodes	149.86	-	199.49	-	123.11	-

5.4 Summarizing Results

We extract the following general conclusions from experiments where RFD and ACO were applied to solve `MSV`, `MDV` and `TSP` in dynamic graphs:

(1) In both static and dynamic graphs, RFD usually obtains better solutions in the long term;
(2) In dynamic graphs, ACO and RFD require similar time to react to changes, with the only exception of `TSP`, where ACO is faster than RFD.
(3) Once RFD has found a good solution in the static case and when several changes were introduced later in the graph, RFD works faster than in the static case because it can exploit the slopes created before; and
(4) RFD always obtains a solution after a modification is introduced, while sometimes ACO cannot adapt the solution constructed before the graph was modified to the new scenario (in particular `TSP`).

These features are again 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.

The fact that sometimes ACO does not find a new solution for `TSP` after a change is introduced deserves additional comments. First, let us note that finding

any solution is harder for TSP than for MSV or MDV (in particular, given a graph, there are *less* round-trips than spanning trees). 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 available 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 resetting part of the graph information. Thus, this information helps RFD react to subsequent graph changes.

6 Conclusions and Future Work

In this chapter we have studied the application of the River Formation Dynamics approach to three NP-complete problems. One of them, TSP, is well-known and has been extensively studied in the literature. The other problems, MSV and MDV, are novel generalizations of other known tree-construction problems. Let us note that RFD is conceptually related to both ACO methods and other gradient-oriented Evolutionary Computation (EC) approaches. On one hand RFD is, in a rough sense, a gradient-driven variant of ACO. On the other hand, the gradient orientation of RFD reminds of methods like Hill Climbing (HC) or Genetic Algorithms (GA) which traverse a space of solutions by seeking solutions with higher fitness. However, there is a big difference between RFD and these methods. RFD modifies the points of a given structure (a *graph*) by iteratively traversing and transforming these points. In this way, the structure is iteratively transformed as well, and finally the formed structure constitutes the returned solution. In HC and GA, the traversed structure is the *space of solutions* itself, which is of exponential size in general. Hence, only a small proportion of these points can be traversed. In these cases, aiming at iteratively *modifying* this space is unfeasible.

Other features of RFD make it different from other EC approaches, specifically ACO. The main ones are the mechanism of focalized punishment of inefficient paths, the avoidance of local cycles, and the fast reinforcement of shorter paths. These characteristics are a consequence of the natural tendency of RFD to form paths that are intrinsically *directed* towards a given final destination. As commented in previous sections, these features are specially suitable for dealing with MDV and MSV problems. Interestingly, a simple parameter change allows RFD to solve either of these problems. RFD's intrinsic features also make it an interesting choice for solving TSP, though in this case one of the advantages of RFD with respect to ACO is lost (in particular, the possibility of *not* endowing drops with memory to register the path traversed so far).

Since RFD is a young method, there is still enough room for introducing both general improvements and purpose-specific variants. Out of a long list of choices, we are specially interested in simulating the *speed* of the drops. Intuitively, when a drop falls through a strong downward slope its speed increases. This gives the

drop some *kinetic energy* allowing it to climb up slopes later. Thus, the speed can be thought as a kind of drop *credit*. In this way, a second derivative mechanism would be introduced in RFD. We are also interested in developing a *hybrid* RFD-ACO system. Altitudes and pheromone trails would be simultaneously represented and considered by *drop-ant* hybrids. The rate of influence of each approach could change along time depending on the suitability of each approach for each execution stage. In this way, we could take the best characteristics of both approaches.

Acknowledgements. The authors would like to thank the anonymous reviewers for valuable suggestions on improving this chapter.

Research partially supported by the MCYT project TIN2006-15578-C02-01, the Junta de Castilla-La Mancha project PAC06-0008-6995, and the Marie Curie project MRTN-CT-2003-505121/TAROT.

Appendix: Proving MDV, MSV $\in$ NP-complete

In this appendix we prove that the novel problems considered in this chapter, `MDV` and `MSV`, belong to the `NP-complete` class. This implies that exponential times are (very probably) required to optimally solve them. Thus, sub-optimally solving them by means of heuristic algorithms like those considered in this chapter is an appropriate choice. The proof is structured as follows. First, we prove that both problems belong to the `NP` class. Next, we prove that a well-known NP-complete problem, `3-SAT`, can be polynomially reduced to each considered problem, which implies that they belong to the `NP-complete` class. At the end of this appendix, we study variants of `MDV` and `MSV` where only graphs fulfilling $N = O$ (that is, graphs where *all* nodes are origin nodes) are considered, showing that these variants are also NP-complete.

Lemma 1. `MDV` $\in$ `NP` and `MSV` $\in$ `NP`.

Proof. We consider `MDV` $\in$ `NP`; proving `MSV` $\in$ `NP` is similar. We prove that `MDV` can be solved in polynomial time by a non-deterministic algorithm. Given a variable-cost graph G and a natural number $K \in \mathbb{N}$, this algorithm non-deterministically constructs a subgraph G' of G and next deterministically checks whether (a) G' is a tree of G, and (b) we have $dc(G') \leq K$. Both operations are performed in polynomial time with respect to the size of G and the size of K (measured in bits). Given a subgraph G' of G, checking whether G' is a tree of G requires polynomial time. Next, if G' is a tree of G, calculating $dc(G')$ requires traversing all paths connecting each origin node to the destination node and adding the costs of all of these paths. The length of each of these paths is polynomial, so calculating the cost of a path requires polynomial time. Since G' is a tree, for each origin node there exists a single path connecting it to the destination node. Thus, the number of paths to be considered is polynomial. Hence, we can check whether the property $dc(G') \leq K$ holds or not in polynomial time. □

In order to prove the NP-completeness of MDV and MSV, we construct a polynomial reduction of a known NP-complete problem to each of these problems. In particular, we consider the well-known 3-SAT problem. Next we introduce some notions related to this problem as well as the problem itself.

Definition 3. The 3-SAT problem is stated as follows: Given a propositional logic formula φ expressed in conjunctive normal form where each disjunctive clause has at most 3 literals, is there any valuation ν satisfying φ?

Let $\varphi \equiv (l_{11} \vee l_{12} \vee l_{13}) \wedge \ldots \wedge (l_{k1} \vee l_{k2} \vee l_{k3})$ be an input for 3-SAT. We denote by $\mathtt{props}(\varphi) = \{p_1, \ldots, p_n\}$ the set of propositional symbols appearing in φ. We denote the i-th disjunctive clause of φ by c_i, that is, $c_i \equiv l_{i1} \vee l_{i2} \vee l_{i3}$.

We say that c_i *holds when* p_j *is equal to* $x \in \{\top, \bot\}$, formally denoted by $\mathtt{h}(p_j, x, c_i)$, if for all valuation ν fulfilling $\nu(p_j) = x$ we have that c_i evaluates to $\top$. That is, $\mathtt{h}(p_j, \top, c_i)$ iff $l_{im} \equiv p_j$ for some $1 \leq m \leq 3$, and $\mathtt{h}(p_j, \bot, c_i)$ iff $l_{im} \equiv \neg p_j$ for some $1 \leq m \leq 3$. □

Theorem 1. 3-SAT $\in$ NP-complete. □

We prove MDV, MSV $\in$ NP-complete as follows (next we consider MDV; the same arguments are given in the case of MSV). Given an input φ of 3-SAT, we show that we can construct an input (G, K) of MDV from φ in polynomial time in such a way that the solution of 3-SAT for φ is *yes* iff the solution of MDV for the variable-cost graph G and the natural number K is *yes*. By the definition of the NP-complete class, this implies MDV $\in$ NP-complete. In particular, if we were able to solve MDV in polynomial time then we could solve the NP-complete problem 3-SAT in polynomial time as well: We could just transform φ into (G, K), next call the algorithm that solves MDV in polynomial time, and finally return the answer given by it.

Before formally presenting the construction of (G, K) from φ, let us informally introduce it. Each origin node of the constructed graph G represents a *disjunctive clause* of φ. From each of these origin nodes, edges iteratively lead through some nodes representing each *proposition symbol* appearing in φ. Each of these proposition nodes is connected to the next proposition node through two edges. One of them represents valuating the corresponding proposition symbol to $\top$, while the other edge represents giving it the $\bot$ value. Depending on the origin node where we come from (that is, depending on the disjunctive clause we are considering), taking the edge that evaluates the proposition symbol to true or to false adds a different cost to the path. This cost is 1 *unless* the proposition valuation represented by the edge allows to make true the disjunctive clause *for the first time* in the path. In this case, the edge adds 0 to the overall path cost. In order to keep track of this information, the value of the *variable* of the variable-cost graph G codifies the considered clause, as well as whether this clause necessarily holds (according to the valuation represented by the path traversed so far). In particular, variable values follow the form $v_{j,w}$ where j is an index denoting a clause and $w \in \{already\top, notyet\top\}$. A value $v_{j,w}$ denotes that the current path departed at an origin node denoting the j-th clause of φ, and $w = already\top$ denotes that the j-th clause must be true regardless of the

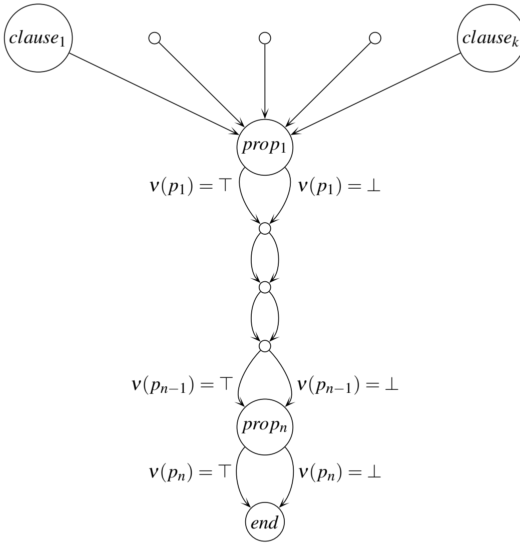

Fig. 21 Structure of the variable-cost graph G

valuation of the remaining proposition symbols (because the valuation implicitly defined by the path traversed so far necessarily makes it true). Otherwise, we consider $w = notyet\top$. After the last proposition node is traversed, the destination node of G is reached. The structure of G is depicted in Figure 21.

Recall that MDV seeks a tree where the addition of costs from each origin node to the destination node is minimal. On the other hand, MSV seeks a tree where the addition of average edge costs is minimal. Let us note that, given the variable-cost graph G, a tree of G can include only *one* of the edges that connect each proposition node to the next proposition node (otherwise, it would not be a tree). Hence, given G, trees computed by both problems represent valuations of proposition symbols. Since MDV searches the cheapest tree connecting all origin nodes to the destination node, MDV actually seeks a tree that allows making as many clauses true as possible. In particular, we will prove that the cost of the cheapest tree found by MDV is under a given threshold if and only if *all* clauses are true under the constructed valuation, that is, iff φ holds. Moreover, due to the specific form of G, finding a tree where the addition of costs from each origin to the destination is minimal is equivalent to finding a tree where the addition of *average* costs of edges is minimal: Due to the definition of G, for both problems the tree cost is minimized if the valuation represented by the tree makes true as many clauses as possible. Hence, we can also define a threshold such that the cost of the minimum tree for MSV is under it iff φ is satisfiable.

Theorem 2. MDV $\in$ NP-complete and MSV $\in$ NP-complete.

Proof. First, we prove MDV $\in$ NP-complete. Due to Lemma 1, if we polynomially reduce 3-SAT to MDV then MDV $\in$ NP-complete. Let φ denote a conjunctive

normal form $\varphi \equiv c_1 \wedge \ldots \wedge c_k$ where $\mathtt{props}(\varphi) = \{p_1, \ldots, p_n\}$. We construct a variable-cost graph $G = (N, O, d, V, A, E)$ as follows:

- $N = \{clause_1, \ldots, clause_k, prop_1, \ldots, prop_n, end\}$,
- $O = \{clause_1, \ldots, clause_k\}$,
- $d = end$,
- $V = \{v_{j,w} | 1 \leq j \leq k \wedge w \in \{already\top, notyet\top\}\}$
- For all $clause_i \in O$ we have $A(clause_i) = v_{i,notyet\top}$.
- $E = \{(clause_i, prop_1, C, T) | 1 \leq i \leq k \wedge \forall v \in V : (C(v) = 0 \wedge T(v) = v)\}$

$$\bigcup$$

$$\left\{ \begin{pmatrix} prop_i, \\ prop_{i+1}, \\ C_i^x, \\ T_i^x \end{pmatrix} \middle| \begin{array}{l} 1 \leq i \leq k-1 \wedge x \in \{\top, \bot\} \wedge \\ C_i^x(v_{j,notyet\top}) = \left\{ \begin{array}{l} 0 \ \mathtt{if} \ \mathrm{h}(p_i, x, c_j) \\ 1 \ \mathtt{otherwise} \end{array} \right\} \wedge \\ C_i^x(v_{j,already\top}) = 1 \wedge \\ T_i^x(v_{j,notyet\top}) = \left\{ \begin{array}{ll} v_{j,already\top} & \mathtt{if} \ \mathrm{h}(p_i, x, c_j) \\ v_{j,notyet\top} & \mathtt{otherwise} \end{array} \right\} \wedge \\ T_i^x(v_{j,already\top}) = v_{j,already\top} \end{array} \right\}$$

$$\bigcup$$

$$\left\{ \begin{pmatrix} prop_n, \\ end, \\ C_n^x, \\ T_n^x \end{pmatrix} \middle| \begin{array}{l} x \in \{\top, \bot\} \wedge \\ C_n^x(v_{j,notyet\top}) = \left\{ \begin{array}{l} 0 \ \mathtt{if} \ \mathrm{h}(p_n, x, c_j) \\ 1 \ \mathtt{otherwise} \end{array} \right\} \wedge \\ C_n^x(v_{j,already\top}) = 1 \wedge \\ T_n^x(v_{j,notyet\top}) = \left\{ \begin{array}{ll} v_{j,already\top} & \mathtt{if} \ \mathrm{h}(p_n, x, c_j) \\ v_{j,notyet\top} & \mathtt{otherwise} \end{array} \right\} \wedge \\ T_n^x(v_{j,already\top}) = v_{j,already\top} \end{array} \right\}$$

We show that constructing G from φ requires polynomial time. This property is a consequence of the following conditions:

(a) $|N|$ is equal to the number of clauses of φ plus the number of proposition symbols of φ plus 1 (the *end* node), which is polynomial with respect to the size of φ.
(b) $|V|$ is equal to the number of disjunctive clauses of φ multiplied by 2. Thus, for each edge in E, defining functions C and T by means of extensional arrays (relating each input value with its output value) requires polynomial size and time.
(c) $|E|$ is equal to the number of clauses plus the number of propositions multiplied by 2, which is polynomial with respect to the size of φ.

Finally, we prove that the *answer* of MDV for G and a given threshold is *yes* iff φ is satisfiable. In particular, we prove that φ is satisfiable iff there exists a tree G' of G such that $dc(G') \leq k * (n-1)$. We consider each implication of this statement:

$\Rightarrow$: Let us note that a tree G' of G must include all edges connecting each node $clause_i$ with $prop_1$. All of these edges have 0 cost. Besides, for each pair of edges connecting each node $prop_i$ with node $prop_{i+1}$, the tree G' must include exactly one of these edges. Let us consider a valuation ν such that for all $1 \leq i \leq n$ we have $\nu(p_i) = \top$ if G' includes the edge $(prop_i, prop_{i+1}, C_i^\top, T_i^\top)$ and $p_i = \bot$

if G' includes $(prop_i, prop_{i+1}, C_i^{\perp}, T_i^{\perp})$. For all $clause_i \in O$, the cost of the path from $clause_i$ to end in G' is $n-1$ if v makes c_i true, and n otherwise. This is because if v makes c_i true then all edges in the path but *one* add 1 cost to this path. The exception is the edge that makes c_i true for the first time, which adds 0 cost. If φ is satisfiable then there exists a valuation v' making *all* clauses c_i true. Thus, there exists a way to choose the edges connecting each $prop_i$ with $prop_{i+1}$ in such a way that, for all $clause_i$, the unique path from $clause_i$ to end has $n-1$ cost. In this case, $dc(G') = k*(n-1)$.

$\Leftarrow$:Let us consider a valuation v defined as in the previous case. If the cost of G' is $k*(n-1)$ then the cost from each $clause_i$ to end must be $n-1$. This implies that, for each $1 \leq i \leq n$, v makes the clause c_i true. Hence, φ is satisfiable.

We prove $\texttt{MSV} \in \texttt{NP-complete}$ by following very similar arguments. In particular, we can construct a polynomial reduction of 3-SAT to MSV by using the same variable-cost graph G defined before. In this case, we argue that φ is satisfiable iff there exists a tree G' of G such that $sc(G') \leq n-1$. Let us recall that, in MSV, the cost of each edge e of G' is the average cost of e for all paths traversing e. Due to the structure of G, it is easy to check that $sc(G') = \frac{dc(G')}{k}$. Thus, φ is satisfiable iff $sc(G') = \frac{k*(n-1)}{k} = n-1$. □

It is worth pointing out that the goal of the previous construction is proving the NP-completeness of MDV and MSV, not providing a suitable graph construction to solve 3-SAT by means of RFD or ACO. In particular, if the variable-cost graph G were used to find solutions to 3-SAT by means of RFD, then we would need to introduce a *barrier node* at each edge connecting a node $prop_i$ with $prop_{i+1}$ (see details about barrier nodes in [17]).

Finally, we study the relation of MSV with other NP-complete problems. Let us note that MSV is a generalization of the *Minimum Steiner Tree* problem. This NP-complete problem is stated as follows: Given a (non-variable) cost-evaluated graph G and a subset S of its nodes, find the minimum spanning tree including (at least) all nodes in S. MSV generalizes this problem by considering *variable-cost* graphs instead of fixed-cost graphs (in particular, the set of origin nodes O in MSV corresponds to the set S given in the previous problem definition). Thus, we may argue that the NP-completeness of MSV is a consequence of the NP-completeness of the Minimum Steiner Tree problem. However, the NP-completeness of MSV (as well as the NP-completeness of MDV) does not lie *only* in the fact that a given *subset* of nodes is required to be included in the tree. In fact, even if only trees including *all* nodes are considered, the NP-completeness of MDV and MSV is met – though, in this case, the Minimum Steiner Tree problem would *not* be NP-complete, because it would be equivalent to the (standard) Minimum Spanning Tree problem (which can be polynomially solved).

Let $\texttt{MSV}'$ and $\texttt{MDV}'$ be problems defined as MDV and MSV, respectively, but with the following difference: Only graphs fulfilling $N = O$ (that is, graphs where all nodes are origin nodes) are considered. We prove $\texttt{MSV}', \texttt{MDV}' \in \texttt{NP-complete}$ by using very similar arguments as before. In particular, let us consider the same

construction as in the proof of Theorem 2, but now nodes $prop_1, \ldots, prop_n, end$ are also included in the set of origin nodes O. A new variable value $nullCost \in V$ is assigned, as initial value, to all of these new nodes, that is, $A(a) = nullCost$ for all $a \in \{prop_1, \ldots, prop_n, end\}$. If $nullCost$ is the current variable value, then taking the next edge is costless and the value $nullCost$ remains after taking any edge. Formally, for all transition $(n_1, n_2, C, T) \in E$ we have $C(nullCost) = 0$ and $T(nullCost) = nullCost$. For the rest of variable values, the behavior of edges remains exactly as defined in the proof of Theorem 2. By using the same arguments as those given in that proof, we infer $\mathtt{MSV}', \mathtt{MDV}' \in \mathtt{NP\text{-}complete}$. Thus, the presence of *variable-cost* edges is a sufficient condition for the NP-completeness of both problems.

References

1. Applegate, D.L., Bixby, R.E., Chvátal, V., Cook, W.J.: The Traveling Salesman Problem: A Computational Study. Princeton University Press, Princeton (2006)
2. Brinksma, E., Tretmans, J.: Testing transition systems: An annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 187–195. Springer, Heidelberg (2001)
3. Bui, T., Zrncic, C.: An ant-based algorithm for finding degree-constrained minimum spanning tree. In: GECCO, pp. 11–18. ACM Press, New York (2006)
4. Davis, L., et al.: Handbook of genetic algorithms. Van Nostrand Reinhold New York (1991)
5. Dorigo, M.: Ant Colony Optimization. MIT Press, Cambridge (2004)
6. Dorigo, M., Gambardella, L.: Ant colonies for the traveling salesman problem. BioSystems 43(2), 73–81 (1997)
7. Dorigo, M., Maniezzo, V., Colorni, A.: Ant system: optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man and Cybernetics, Part B 26(1), 29–41 (1996)
8. Fleischer, M.: Simulated annealing: past, present, and future. In: Proceedings of the 27th conference on Winter simulation, pp. 155–161 (1995)
9. Gutin, G., Punnen, A.: The Traveling Salesman Problem and Its Variations. Kluwer, Dordrecht (2002)
10. Jong, K.: Evolutionary computation: a unified approach. MIT Press, Cambridge (2006)
11. Kennedy, J., Eberhart, R.C.: Particle swarm optimization. In: Proceedings of IEEE International Conference on Neural Networks, Piscataway, NJ, pp. 1942–1948 (1995)
12. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by Simulated Annealing. Science 220, 671–680 (1983)
13. Langton, C.: Studying artificial life with cellular automata. Physica D 22, 120–149 (1986)
14. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines: A survey. Proceedings of the IEEE 84(8), 1090–1123 (1996)
15. López, N., Núñez, M., Rodríguez, I.: Specification, testing and implementation relations for symbolic-probabilistic systems. Theoretical Computer Science 353(1–3), 228–248 (2006)
16. Petrenko, A.: Fault model-driven test derivation from finite state models: Annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 196–205. Springer, Heidelberg (2001)

17. Rabanal, P., Rodríguez, I., Rubio, F.: Using river formation dynamics to design heuristic algorithms. In: Akl, S.G., Calude, C.S., Dinneen, M.J., Rozenberg, G., Wareham, H.T. (eds.) UC 2007. LNCS, vol. 4618, pp. 163–177. Springer, Heidelberg (2007)
18. Rabanal, P., Rodríguez, I., Rubio, F.: Finding minimum spanning/distances trees by using river formation dynamics. In: Dorigo, M., Birattari, M., Blum, C., Clerc, M., Stützle, T., Winfield, A.F.T. (eds.) ANTS 2008. LNCS, vol. 5217, pp. 60–71. Springer, Heidelberg (2008)
19. Reinelt, G.: TSPLIB 95. Research Report, Institut für Angewandte Mathematik, Universität Heidelberg, Heidelberg, Germany. Tech. rep. (1995), http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/
20. Rodríguez, I., Merayo, M., Núñez, M.: HOTL: Hypotheses and observations testing logic. Journal of Logic and Algebraic Programming 74, 57–93 (2008)