Optimization
Vol. 00, No. 00, Month 200x, 1–20

RESEARCH ARTICLE

Self-tuning of the noising methods

Irène Charon and Olivier Hudry, Télécom ParisTech & CNRS - LTCI UMR 5141, 46, rue Barrault, 75634 Paris Cedex 13 - France

(*Received 00 Month 200x; in final form 00 Month 200x*)

The noising methods constitute a family of metaheuristics and generalize the simulated annealing and the threshold accepting methods. One main difficulty with noising methods and more generally with metaheuristics relies in the necessity to tune several parameters. This paper details a way to design noising methods which can tune their parameters themselves, so that there remains only one parameter that the user provides: the CPU time that he or she wants to spend in order to solve his or her problem. Experimental results obtained for four problems show that these self-tuned noising methods succeed in finding good tunings and in providing good results to the studied problems.

Keywords: Metaheuristics, noising methods, simulated annealing, threshold accepting algorithms, automatic tuning, combinatorial optimization.

MSC classification: 90C15, 90C27, 90C31, 90C35, 90C59.

ISSN: 0233-1934 print/ISSN 1029-4945 online
© 200x Taylor & Francis
DOI: 10.1080/0233193YYxxxxxxxx
http://www.informaworld.com

1. Introduction

In 1993, we designed a new metaheuristic for combinatorial optimization problems: the *noising method* [5]. Then, by combining different variants or by designing different "noising-schemes" (see below), we generalized it into a family of methods: the *noising methods* (NM in the following) which share the same basic features (see [10] for a review of the principles and of the applications of NM).

As the other metaheuristics (see for instance [1], [16], [19], [24], [25], [29], [32], [35], [36], [37], [38], [42], [43] or [46] for general presentations upon metaheuristics, for applications and for references), NM are not designed to solve only one problem, but are designed to be applicable more generally to combinatorial optimization problems. Such a problem can be described as follows:

$$\text{Minimize} f(s) \text{ for } s \in S,$$

where S is a finite set of *solutions* and where $f(s)$ gives the value of s according to the objective function f.

As metaheuristics like descents, simulated annealing, tabu search and some others, NM are based on *local* or *elementary transformations*. Such a transformation changes one feature of the current solution without changing its global structure. When applied to a solution s, an elementary transformation changes s into a *neighbour* of s. Thus, for a given elementary transformation, we may define the *neighbourhood* $N(s)$ for each solution s as the set of the neighbours of s that are obtained when the elementary transformation is applied to s.

Thanks to these transformations, we may design a *descent method*. A descent starts with an initial solution s_0 (which can be for instance randomly computed, or found by a heuristic) and then generates a series of solution s_1, s_2, ..., s_i, ..., s_q such that:

(1) for any $i \geq 1$, s_i is a neighbour of s_{i-1}: $s_i \in N(s_{i-1})$;
(2) for any $i \geq 1$, s_i is better than s_{i-1} with respect to f: $f(s_i) < f(s_{i-1})$;
(3) no neighbour of s_q is better than s_q: $\forall s \in N(s_q)$, $f(s) \geq f(s_q)$.

Then s_q is the solution returned by the descent. By (3), s_q is at least a local minimum of f with respect to the elementary transformation. A classic variant of the descent consists in applying descents successively from random initial solutions and in returning the best computed solution: it is the *repeated descents* (RD) method.

In the following, $\Delta f(s, s') = f(s) - f(s')$ will denote the variation of the objective function when we try to move from the current solution s to one of its neighbours s'.

There exist several ways to explore the neighbourhood of the current solution (see [10] for instance). In this paper, we adopt a *systematic* or *cyclic* exploration (see [15]): the neighbours of any solution are ranked in an implicit order and they are all considered in this order once, before being considered for a second time. A neighbour better than the current solution is accepted as soon as it has been discovered, even if there exists another neighbour, not yet considered, which is still better. Other possibilities include a random exploration (as in a classic simulated annealing), or an exhaustive one (as in a classic tabu search) in order to find the best neighbour. It is sometimes possible to improve this systematic exploration by mixing it with an exhaustive one; it is what we shall do for the first two problems described below.

NM are also based on elementary transformations. The main difference with a descent is that, instead of considering the genuine objective function f, we successively optimize perturbed ("noised") functions obtained by adding *noises* to f. These noises are randomly chosen into an interval of which the range decreases during the process down to 0: so, as the process runs, the noised functions get closer and closer to f and, at the end, these is no noise (or no significant noise) and we deal with f itself. From time to time, we may insert a descent with respect to the genuine function f to optimize, in order to stay closer to the values taken by f. Of course, it is necessary to specify some components (what gives the *scheme* of a NM): the probability distribution of the noises (and thus the probability distribution to accept a neighbour of the current solution s), their range, the way of decreasing them, the way of exploring the neighbourhood of s, and so on. These components give to the user many possibilities to get his/her own scheme to design a NM. When these components are properly chosen, we get back the scheme of simulated annealing or of the threshold accepting algorithms designed by Dueck *et alii* [17], [18] (see [9] for details).

From a theoretical point of view, the wider possibilities associated with NM involve the possibility to get better heuristics, what happens usually from a practical point of view too. Nevertheless, as for other metaheuristics, it is still necessary to tune several parameters. Unfortunately, the values of these parameters depend on the problem to solve, and even on the instances of the problem, and thus usually it is quite difficult to find even good values for them (and sometimes, even not too bad ones). The aim of this paper is to describe a first attempt to tune the numerical parameters of NM automatically, by the applied NM itself, so that the user has only one parameter to choose himself or herself: the CPU time that he or she wants to spend in order to solve the considered instance of his or her problem (of course, it is also necessary to define the structural components like the elementary transformation). The other parameters are automatically computed during the run of the algorithm (it is especially the case for the initial *noising-rate*, which plays the same role as the initial temperature in simulated annealing) or fixed in order to simplify the tuning (it is the case for the last noising-rate, which is fixed to 0; see the discussion in the conclusion). Obviously, the more CPU time, the better the expected value of the solution computed by the self-tuned NM.

We tested this self-tuned noising method (STNM) with two different types of noises on four problems: the clique partitioning of a weighted graph, the linear ordering problem, the travelling salesman problem (TSP) in the general case (weights are not necessarily distances), and the Euclidean TSP. Hence, STNM is tested eight independent times with the same program and especially the same way of tuning the numerical parameters.

The paper is organized in six sections. In Section 2, we briefly depict the problems on which STNM has been tested. Then we describe in Section 3 the scheme of the noising methods that we intend to tune automatically; this scheme is based on a procedure that we call the *Core procedure*. Then we show in Section 4 how to tune the parameters of NM automatically. Section 5 is devoted to the experimental results. Conclusions can be found in Section 6.

2. The four studied problems

The four problems studied here arise from graph theory. In this paper, $G = (V, E)$ will denote a graph (undirected for three problems and directed for one). The number of vertices of G will be denoted n, and the vertices will be v_1, v_2, ..., v_n.

As the noising methods belong to the set of metaheuristics defined by neighbourhoods, it is necessary to define such neighbourhoods for the studied problems. Usually, the neighbourhood is defined by an *elementary* (or *local*) *transformation*; then the neighbourhood of a solution s is the set of solutions obtained by applying the elementary transformation to s. We specify the elementary transformations for each problem below. Notice that, from previous experiments (see [10] and references below), NM give usually better results than classic metaheuristics like simulated annealing for these problems.

2.1. *The clique partitioning of a weighted graph problem*

From a chronological point of view, the problem of partitioning a weighted graph into cliques is the first application of the noising methods (see [5]). Since 1993, the noising methods have been applied to this problem several times: see [11], [14], [20], [21], [45]. It appears in different contexts (see [2], [3] or [23] for references and related topics).

An instance of this problem is given by an undirected complete graph $G = (V, E, w)$ of which the edges $e \in E$ are weighted by integers $w(e)$ (which can be positive, negative, or equal to 0). A solution s is any partition of V into a certain number k of subsets $V_1, V_2, ..., V_k$; notice that the number of subsets is not given in the instance: it depends on G (more precisely, on n and w). The objective function f is given by the sum of the weights of the edges between the different subsets of the considered partition: more precisely, let k be any integer with $1 \leq k \leq n$ and let $V_1, V_2, ..., V_k$ be any partition of V into k subsets; then we set:

$$f(V) = 0 \text{ and, for } k > 1, f(V_1, V_2, ..., V_k) = \sum_{1 \leq i < j \leq k} \sum_{(x,y) \in V_i \times V_j} w(\{x, y\}).$$

The aim is to determine the optimal value k^* of k and an optimal partition $V_1^*, V_2^*, ..., V_{k^*}^*$ into k^* subsets of V in order to minimize f:

$$f(V_1^*, ..., V_{k^*}^*) = \text{minimum of} f(V_1, ..., V_k) \text{ for } (V_1, ..., V_k) \in \{\text{partitions of } V\}.$$

This problem is NP-hard (see [47]).

The adopted elementary transformation is the one proposed by S. Régnier in [39]: it consists in removing a vertex x from the subset $V(x)$ to which x belongs currently (if $V(x)$ contains only x, then $V(x)$ becomes empty and disappears: the number of subsets decreases by 1) and to add x to another existing subset or to a new subset (which thus contains only x at this moment; in this case, the number of subsets increases by 1). This transformation allows to reach any partition of V from any other partition of V. Notice that the number of neighbours of a partition P of V depends on P: if we do not take into account some special cases (as the one for which some subsets can contain only one vertex), then the number of neighbours of P is about $n \times (k + 1)$, where k denotes the number of subsets of P.

To explore these neighbourhoods, we mix the systematic strategy described above with an exhaustive one. More precisely, if the implicit order involved in the systematic exploration of the vertices is $v_1, v_2, ..., v_n$, we look for the best subset for v_1 (which can be its current subset, or the empty set, or something else), then the best subset for v_2, and so on until v_n, then again for v_1 and so on, until it is not possible to improve the situation by such a move. Such a mixed strategy does not change the features of the descent, but saves time by avoiding to sum up the same weights several times or to move again a same vertex which has just been moved into a subset which was not the best possible.

2.2. *The linear ordering problem*

For this problem, an instance is given by a tournament (i.e., an asymmetric complete digraph) $T = (V, A, w)$ of which the arcs $a \in A$ are weighted by non-negative integers $w(a)$. To any linear order $O = (V, B)$ defined on V, we associate a value $f(O)$ given by the sum of the weights of the arcs with a different orientation in T and in O:

$$f(O) = \sum_{(v,v') \in A-B} w(v, v').$$

The problem consists in finding a linear order O^* (sometimes called a *median order* of T; see [2]) which fits T as well as possible:

$$f(O^*) = \text{minimum of } f(O) \text{ for } O \in \Omega(V),$$

where $\Omega(V)$ denotes the set of the linear orders defined on V. This problem is also NP-hard (for references on the complexity of the computation of median orders, see [26]; see also [2], [13], [30] or [40] for references on the context). NM have been applied formely to the linear ordering problem in [7] and in [12]. A special interesting case is the one (called Slater problem, see [44]) for which all the weights are equal to 1 (then $f(O)$ gives the number of disagreements between O and T); Slater problem is also NP-hard (see [13] for references).

In our experiments, the elementary transformation consists in shifting a vertex from its current place in the considered linear order O to another place in O. More precisely, if O is the linear order $v_1 > ... > v_{i-1} > v_i > v_{i+1} > ... > v_{j-1} > v_j > ... > v_n$, where the notation $v_k > v_{k+1}$ means that the arc between v_k and v_{k+1} is (v_k, v_{k+1}) (the other arcs are obtained by transitivity), then the neighbours of O looks like $v_1 > ... > v_{i-1} > v_{i+1} ... > v_{j-1} > v_i > v_j > ... > v_n$ for some appropriate i and j. Here, the number of neighbours does not depend on O, and is always equal to $(n-1)^2$. To explore the neighbourhood, we also mix two strategies, as for the previous problem: we look for the best place of v_1 in the current linear order, then the same for v_2, and so on until v_n, to do it again from v_1 until no move brings improvement.

2.3. *The travelling salesman problem*

The travelling salesman problem (TSP) is well-known to be NP-hard (see [22], [31] or [33] for references). Given a complete weighted graph, it consists in finding a Hamiltonian cycle with a minimum weight. In this TSP, the weights are not necessarily distances. The adopted elementary transformation is the usual 2-opt proposed by Lin in [34] and which consists in removing two non-adjacent edges in the current Hamiltonian cycle and in adding the two properly chosen edges providing a new Hamiltonian cycle: if the current Hamiltonian cycle is $v_1 v_2 ... v_{i-1} v_i v_{i+1} ... v_{j-1} v_j v_{j+1} ... v_n$, then the application of the 2-opt transformation gives a cyle of the form $v_1 v_2 ... v_{i-1} v_i v_j v_{j-1} v_{j-2} ... v_{i+2} v_{i+1} v_{j+1} v_{j+2} ... v_n$ for properly chosen i and j (in particular, $i \notin \{j-1, j, j+1\}$). Here also the size of the neighbourhood does not depend on the current solution and is equal to $\frac{n.(n-3)}{2}$. Applications of NM to this problem can be found in [6], [8], [27] and [28].

2.4. *The Euclidean travelling salesman problem*

Here we deal with the same problem as in Section 2.3, but the vertices are points in the Euclidean plan and the weight of an edge $\{x, y\}$ is given by the Euclidean distance between x and y. We consider this case separately because we can draw benefit from its specificities. In particular, we follow an idea developped by E. Bonomi and J.-L. Lutton in [4]: the points are contained in a square divided into several equal sub-squares so that the average number of points in each sub-square is about 3; then the application of the 2-opt transformation is limited to vertices v_i and v_j belonging to the same sub-square or to adjacent (i.e., sharing at least a corner) sub-squares. This allows to reduce the number of possible 2-opt (and so to speed up the computations) without damaging the performance of the method (see [4] for details). Thus the size of the neighbourhood depends on the current Hamiltonian cycle and above all on the distribution of the points into the square.

3. The studied noising methods

As said above, NM are based on elementary transformations as descents, but applied to "noised" functions. There are several ways to define these noised functions (see [9] or [10] for more details). In our experiments, we consider two ways:

- noising the variations of f;
- noising the data from which the values of f are computed.

3.1. *Noising the variations*

The most general way (see [10]) of defining the noised functions f_{noised} consists in perturbing the variations of f. When a neighbour s' of s is tried, we do not consider the genuine variation $\Delta f(s, s')$ of f, but a *noised variation* $\Delta f_{noised}(s, s')$ defined by:

$$\Delta f_{noised}(s, s') = \Delta f(s, s') + \rho$$

where ρ denotes the noise (it depends on s and s', but also on the iteration: if we consider twice the same pair (s, s'), ρ is not necessarily the same). The noise ρ follows a specified probability law of which the mean and the standard deviation tend to 0, by decreasing for the standard deviation.

Among the different possible probability distributions, we adopt here a "logarithmic" distribution:

$$\Delta f_{noised}(s, s') = \Delta f(s, s') + r \ln(u),$$

where u is a number uniformly drawn into $]0,\ 1[$ and where r is a decreasing parameter called the *noising rate*. This choice comes from the acceptance criterion used in simulated annealing (SA). Indeed, in SA, a neighbour s' is accepted instead of s with a probability equal to $\min\{1, \exp(\frac{-\Delta f(s,s')}{T})\}$, where T is the temperature. Then, in order to know whether s' is accepted instead of s when $f(s')$ is greater than $f(s)$, we draw a number u uniformly into $]0,\ 1[$ and we compare $\exp(\frac{-\Delta f(s,s')}{T})$ and u: if u is less than $\exp(\frac{-\Delta f(s,s')}{T})$, s' is accepted, otherwise s' is rejected. It is easy to see that it is the same as applying the acceptance criterion $\Delta f_{noised}(s, s') < 0$ with the above definition of $\Delta f_{noised}(s, s')$, if r is chosen to be equal to T (it is why we may consider that NM generalize SA; see [9]). The noising rate r can decrease geometrically or arithmetically from a maximum value r_{max} which depends on the instance to solve and hence which must be tuned.

3.2. *Noising the data*

Another possibility to noise f (and, chronologically, the first one) consists in perturbing f by adding the noises to the data. For instance, if f is a linear function of p variables x_i (with $1 \leq i \leq p$):

$$f(s) = \sum_{i=1}^{p} a_i x_i,$$

with $s = (x_i)_{1 \leq i \leq p}$, we may perturb the data a_i by adding some random noises ρ_i to them in order to design a "noised" function f_{noised}:

$$f_{noised}(s) = \sum_{i=1}^{p} (a_i + \rho_i) x_i.$$

In this kind of NM, when the noised function f_{noised} has been defined, we may apply a descent with respect to f_{noised}, which provides some solution s. Then we compute new noises to add to the data, and we restart a descent with respect to the new noised function, from s. And so on. At each iteration, the noises ρ_i are randomly drawn with a given probability distribution of which the mean and the standard deviation tend towards 0 and the standard deviation decreases during this process: in other words, the added noises vanish progressively and, similarly, the series of noised functions converges towards f. Because of these noises, it may happen that a transformation which would be rejected in a descent (because it would involve an increase of f) is accepted and, conversely, that a transformation which would be accepted in a descent is rejected.

For the problems studied in this paper, the data are given by the weights w of the edges or of the arcs (directed edges) of the considered graph. Then, noising the data simply consists in noising these weights by adding the random noises ρ_i to them. The noises are drawn with a uniform law into an interval $[-r, r]$, where r, the noising rate, decreases arithmetically from an initial value r_{max}, which depends on the data and which must be tuned, down to 0.

As the previous pattern is more general than this one (in fact, we only consider variations to decide whether the scanned neighbour of the current solution is accepted or rejected), we may simulate this way of noising f by the noising of the variations. But il will be necessary to take care with the probability distribution of the noises. For instance for the TSP, as the noises added to the data follow a uniform law, the variation of f involves the sum of four uniform noises (thus the law followed by this sum is not uniform). The situation is still more complex for the partitioning problem and for the linear ordering problem, since the number of terms involved in the computation of the variation is not constant; in these cases, we add to the variation of f as many uniform noises as there are weights involved in the computation of $\Delta f(s, s')$ (notice that, if this number of added noises is large, the sum of the uniform noises tends to follow a Gaussian law).

So, thanks to this possibility to simulate the noising of the data as the noising of the variations, we deal only with the noising of variations in the following, with two types of probability distributions: the one that we call "logarithmic" above, and the one which allows to simulate the noising of the data (and which depends

```
Core($r$, $s$)                (* $r$ is the noising rate, $s$ is the initial solution *)
    for $1 \le i \le 4NS$, do    (* $NS$ denotes the size of the neighbourhood *)
        let $s'$ be the next neighbour of $s$
                            (* “next” with respect to the implicit order *)
                            (* used to explore the neighbourhoods *)
        compute the noised variation $\Delta f_{noised}(s, s')$   (* what involves $r$ *)
        if $\Delta f_{noised}(s, s') < 0$ then replace $s$ by $s'$
    apply an unnoised descent from $s$
    return the solution found by the unnoised descent.
```

Figure 1. Algorithm for the Core procedure. The Core procedure gathers the performed elementary transformations into $4NS$ “noised” transformations, where NS is the size of the neighbourhood, followed by an “unnoised” descent.

on the performed elementary transformation).

3.3. *The Core procedure*

The description of a NM can be based on the *Core procedure* that we depict now. Broadly speaking, the Core procedure consists in applying “noised” transformations, i.e. transformations that are accepted or rejected with respect to the noised variations involved by them, followed by an “unnoised” descent, i.e. a descent applied with respect to the genuine variations of f. Because of the stochastic aspect of NM when the variations are noised, it is not possible to apply noised transformations until a local minimum of f (with respect to the adopted elementary transformation) has been reached. So, we do the following: we try a given number NT of noised transformations, then we apply an unnoised descent until a local minimum is reached, then we try again NT noised transformations, then an unnoised descent, and so on. From several previous experiments (see references in [10]), it appears that choosing $NT = 4NS$ leads usually to good results, where NS denotes the size of the neighbourhood of a solution. In other words, we gather the elementary transformations in order to try $4NS$ noised transformations, then an unnoised descent, then again $4NS$ noised transformations followed by an unnoised descent, and so on.

The Core procedure consists in performing $4NS$ noised transformations followed by an unnoised descent. Core is the base of the self-tuned version of the noising methods (STNM). It needs two parameters: the current noising rate r and an initial solution called s below; this noising rate will decrease after each call to Core. Figure 1 summarized the principles of Core method.

Core helps to design a classic (versus self-tuned) NM. Such a classic NM needs the specification of several numerical parameters:

- the number of performed elementary transformations or, equivalently, the number of times, called *nbCore* in Figure 2, that Core is applied; this parameter is obviously linked to the CPU time that the user can spend to solve the considered instance of his/her problem;
- the initial (and maximum) value r_{max} of the noising rate;
- the final (and minimum) value r_{min} of the noising rate;
- the value by which the noising rate r must decrease after each call to Core.

Figure 2 gives the features of a classic NM in which the noising rate decreases arithmetically.

```
Classic noising method
Parameters given by the user: $r_{max}$, $r_{min}$, $nbCore$

        compute an initial solution $s$
        $bestSol \leftarrow s$
        for $0 \leq i \leq nbCore - 1$, do
            $r \leftarrow r_{max} - i \times \frac{r_{max}-r_{min}}{nbCore-1}$
            $s \leftarrow Core(r, s)$
            if $f(s) < f(bestSol)$ then $bestSol \leftarrow s$
        return $bestSol$
```

Figure 2. The features of a classic noising method (NM). NM is based on the Core procedure, which is applied iteratively with a noising rate r decreasing arithmetically from a maximum value r_{max} down to a minimum value r_{min} which can be equal to 0.

4. The self-tuning of the parameters of the noising methods

4.1. *The main features of STNM*

The aim of the self-tuning is to obtain a NM with only one numerical parameter: the CPU time T that the user wants to spend to solve his/her instance. From previous experiments (see references in [10]), it appears that the most important parameter is r_{max}. If r_{max} is too high, then we waste much time at the beginning of NM: then in this case we must spend much more time than necessary or it is not possible to obtain a good solution. If r_{max} is too low, the behaviour of NM is near the one of a descent and we do not take benefit from the specificities of NM. From the experiments done on the application of the noising methods to the TSP, it appears that it is not always necessary to tune r_{max} very sharply: any value inside $[0.9r^*, 1.1r^*]$, where r^* denotes the best value of r_{max} according to a fine tuning, or even inside $[0.8r^*, 1.2r^*]$, led to very good results in the experiments reported in [8].

On the other hand, the same experiments show that it is never a bad choice to choose r_{min} as equal to 0: other choices allow to save CPU time, but usually not in a qualitatively great extent. So, to design STNM, we definitely set $r_{min} = 0$.

Similarly, we definitely adopt an arithmetical decrease for the noising rate r after each call to Core. Thus the arithmetical decreasing ratio of r is directly given by r_{max} and $nbCore$ (since $r_{min} = 0$) and is equal to $\frac{r_{max}}{nbCore-1}$.

Hence, the self-tuning will deal with two parameters: the initial value r_{max} of the noising rate r; the number $nbCore$ of calls to Core according to the CPU time fixed by the user. STNM is made of a series $(N_1, N_2, ..., N_q)$ of classic NM. To initialize the series (that is, to define the characteristics of N_1), we apply a preparatory stage (see below). Then, the characteristics of each N_i are defined from the ones of N_{i-1} and from the results obtained during the computations of N_{i-1}. The CPU time devoted to N_i is twice the CPU time allocated to N_{i-1} (except for the last NM of the series, N_q, of which the CPU time is at least twice the one of N_{q-1}). So, as STNM runs, the NM of the series will become longer and longer and the CPU time devoted to the last NM (i.e. N_q) is at least half the total CPU time allocated by the user. To compute the initial noising rate of each N_i, we apply two principles: on the one hand, we compute the average of the noising rates allowing to decrease the current value of f below a given threshold that we detail below; on the other hand, a way to adjust the value of the noising rate to what can be considered as a good initial noising rate.

Broadly speaking, *threshold* gives the value of f from which we can consider that the current Core procedure begins to be truly efficient. At any step of the method, we know the best value *minDesc* computed by a descent since the beginning of STNM. It is usually not difficult to obtain values of f lower than *minDesc*; so we consider only the values of *threshold* lower than *minDesc*. On the other hand, it is difficult (this becomes more and more difficult as STNM runs) to obtain values of f lower than the best value $bestValue$ computed by STNM since its beginning. Thus, it is reasonable from the previous two considerations to choose a value of *threshold* between *minDesc* and $bestValue$. After different trials (see Section 5.3), we adopted the arithmetic mean of *minDesc* and $bestValue$ for *threshold*.

During the whole run of STNM, we keep in memory the best solution computed since the beginning and called *bestSol* (as in Figure 2), as well as its value that we call $bestValue$ (hence equal to $f(bestSol)$). In the following, we do not recall this storage, but it exists obviously.

We detail below the preparatory stage first, then the way of computing the initial noising rate and the number of calls to Core for each NM of the series.

4.2. *The preparatory stage*

The aim of the preparatory stage is mainly (but not only) to compute a first initial noising rate (i.e., the initial rate of N_1). We proceed in several steps.

- We initialize the initial noising rate r_{max} with a value which depends on the values taken by f or, equivalently, on the data. This initial value for r_{max} must be positive. In fact, it is not necessary to choose this value sharply, since r_{max} is going to be tuned. In our experiments, the initial value of r_{max} is given by the average of the absolute values of the weights of the graph that we deal with.
- We apply 10 descents with a random initial solution for each of them. Let *minDesc* be the smallest of the 10 values obtained for f at the end of the 10 descents. In the following, each time that a descent is applied from a random initial solution, *minDesc* is updated. So, *minDesc* gives the best value of f found by a randomly initialized descent since the beginning of STNM.
- From these 10 descents, we compute the average CPU time $\bar{t}$ necessary to apply a descent. We initialize a variable called *adjustment* with a value between 1.1 and 2: *adjustment* gets a value equal to 1.1 if $\bar{t}$ seems small with respect to the total CPU time T allocated by the user (more precisely, if $\bar{t} < T/100$); *adjustment* gets a value equal to 2 if $\bar{t}$ seems large with respect to T (more precisely, if $\bar{t} > T/100$); and *adjustment* varies linearly between 1.1 and 2 for the intermediate values of $\bar{t}$.
- We apply the Core procedure from a solution provided by a descent and, during the noised phase of Core, we compute the percentage π of accepted elementary transformations with respect to the total amount of tried elementary transformations. If π is less than 30 % (see below for a discussion about this percentage), we multiply the noising rate r_{max} by the variable *adjustment* defined above; then we apply Core once again, we compute the new value of π that we multiply once again by *adjustment* if the new value of π is still less than 30 %, and so on, until π becomes greater than 30 %. If on the contrary the first value of π is greater than 30 %, we apply the same process, but by dividing by *adjustment* instead of multiplying by it, until the current value of π becomes smaller than 30 %. So, in both cases, at the end of this process, we get a value of r_{max} for which the accepting percentage π is about 30 %. This value of r_{max} provides the initial noising rate of N_1.

Then the preparatory stage is over and we may perform the NM of the series.

4.3. *The self-tuning of the classic noising methods of the series*

After this preparatory stage, we perform the classic NM of the series $(N_1, N_2, ..., N_q)$.

Each N_i $(1 \leq i \leq q)$ is associated with a variable $nbCore_i$ which specifies the number of calls to Core performed by N_i. The first NM N_1 is very short: $nbCore_1 = 10$. For $1 < i < q$ (that is, for all the N_i but the first and last ones), the CPU time given to N_i is twice greater than the CPU time devoted to N_{i-1}. To reach this aim, we simply double the values of the variables $nbCore_i$: for $1 < i < q$, $nbCore_i = 2 \times nbCore_{i-1} = 10 \times 2^{i-1}$.

For the last NM of the series, i.e. N_q, we do so that it lasts at least twice longer than N_{q-1}. In fact, during the process, we estimate also the CPU time necessary to perform Core; thus we can estimate the CPU time T_i necessary to perform N_i for $2 \leq i \leq q$. When N_{i-1} has been performed, the already consumed CPU time is equal to $\sum_{j=i}^{i-1} T_j = (2^{i-1} - 1)T_1$ and the remaining CPU time is $T - (2^{i-1} - 1)T_1$, where T denotes the total CPU time allocated by the user. To perform N_i, we intend to give a CPU time T_i equal to about $2 \times T_{i-1}$, and the CPU time allocated to the remaining NM of the series becomes $T - (2^i - 1)T_1$. But if the remaining CPU time is less than the CPU time that we want to allocate to the next NM of the series (i.e., if we have: $T - (2^i - 1)T_1 < T_{i+1} = 2T_i = 2^i T_1$), then we give the whole remaining CPU time $T - (2^i - 1)T_1$ to N_i and N_i becomes the last NM N_q of the series. With this strategy, the last NM of the series gets at least half the total CPU time T devoted by the user to solve his/her problem.

To compute the initial noising rates r_{max} of the classic NM, we try to take advantage of the "experience" got during the previous NM of the series. Each NM N_i $(2 \leq i \leq q)$ of the series inherits its initial noising rate from N_{i-1} and computes the initial noising rate of N_{i+1} (except for N_q, since it is the last NM of the series). For each N_i $(1 \leq i \leq q)$, we perform the following steps.

- An initial solution is randomly generated, and a descent is applied to it (and thus $minDesc$ is updated if necessary).
- We compute the value of $threshold$ as the arithmetic mean of the best value $minDesc$ got by a descent and the best value $bestValue$ obtained from the beginning of STNM: $threshold = (minDesc + bestValue)/2$.
- The Core procedure is applied $nbCore_i$ times, with an arithmetic decrease from the initial noising rate r_{max} computed by N_{i-1} (or by the preparatory stage for N_1) down to 0, as specified in Section 4.1. From the moment when the value of the current solution becomes less than $threshold$, we compute the average of the noising rates which allow to improve the current solution during these applications of Core; we call *goodRate* this average value of the noising rates.
- We give the value of *goodRate* to a variable *rate* (*rate* helps to compute the initial value of the noising rate of N_{i+1}) and we adjust it. To do so, we apply Core from *bestSol* with *rate* as its initial noising rate. While the solution obtained by the procedure gives to f a value less than $threshold$, we adjust *rate* by multiplying it by the variable *adjustement* computed in the preparatory stage and by applying Core from the current solution and with *rate* as its initial noising rate. Let r_i be the value of *rate* obtained at the end of this process.
- The initial noising rate of N_{i+1} is given by the average of the rates r_j computed since the beginning, weighted by 2^{j-1} (the importance of each r_j is proportional to the CPU time devoted to N_j). In other words, the initial noising rate r_{max} of

N_{i+1} is given by:

$$r_{max} = \frac{1}{2^i - 1} \sum_{j=1}^{i} 2^{j-1} r_j.$$

- The number of calls to Core is updated as explained above (i.e., it doubles from N_i to N_{i+1}, except for the last NM of the series).

As the CPU time at least doubles from N_i to N_{i+1}, the last NM of the series, which should benefit from the best computed noising rate, lasts at least half the total CPU time allocated by the user: thus, what should be the best NM is also the longest. Since the number of NM performed during STNM can be large, and since each N_i starts with a random solution, STNM usually explores a significant part of the set of solutions, and it happens that the best solution is found quite before the last NM of the series. Moreover, the results provided by the successive NM of the series become good quickly (even if, of course, waiting longer usually improves these results).

5. Experiments

STNM has been programmed in C and run on a Sparc 4 Sun workstation. The same program is run for the four problems depicted in Section 2 (except, of course, the parts depending on the problem, as the elementary transformations). For each problem, we tested the two ways of noising the values taken by f described in Section 3.1 and 3.2 and called below STNM-V for the noising of the variations (what allows to design a self-tuned simulated annealing; see Section 3.1) and STNM-D for the noising of the data. Thus, we obtain eight series of independent results.

5.1. *Experimental results*

For each of the four problems, we tried STNM on several thousands of instances with a number of vertices from several tens to several thousands and with different features:

- graphs with random weights drawn with a uniform distribution (and with a uniform law for the orientation of the arcs, in the case of the linear ordering order);
- graphs with random weights following a distribution which is not uniform: if the n vertices are numbered from 0 to $n-1$, then the weight w_{ij} of the edge (or the arc for the linear ordering problem) between i and j is the higher as i and j are near each other (more formally, for $0 \leq i \leq n-1$, $w_{i,i+1} < w_{i,i+2} < ... < w_{i,i+\lfloor n/2 \rfloor}$ and $w_{i,i-1} < w_{i,i-2} < ... < w_{i,i-\lfloor n/2 \rfloor}$, where the additions and the subtractions are done modulo n);
- graphs corresponding to real life instances (especially for the partitioning problem) or simulating real life instances (for the linear ordering problem, see [12]);
- graphs associated with special instances of the four problems and for which it is possible to estimate more or less the optimal values of f: for the TSP and the Euclidean TSP, graphs of which the vertices are located on the crosses of a regular grid or are spread over a square with a uniform distribution, and graphs coming from the library TSPLIB [41] (for these graphs, the number of vertices is specified inside the name of the instance); for the partitioning problem, graphs of

which the vertices are located on a circle or on a torus and of which the weights are given by the Eucidean distance between the considered vertices minus a constant; for the linear ordering problem, tournaments of which the weights are all equal to 1 (Slater problem) or simulates the aggregation of some special sets of (deliberately) biased individual preferences defined on the set of vertices (see [12] for details).

It is not possible to detail all the experimental results that we obtained, and it is not useful, since the qualitative results are globally the same and since, because of the stochastic aspects of NM, there is no optimal tuning to which we could compare the tunings found by the method described above. In order to be more specific in the discussion below, we detail in Figure 3 the results obtained by STNM on one instance of each problem: Part100, Slater200, Tsp100, Grid1024; once again, many other trials of STNM have been done and they all lead to the same qualitative conclusions. The characteristics of these instances are the following:

- Part100 is an instance of the partitioning problem with 100 vertices; the weights are randomly chosen between -100 and 100 with a uniform law; the optimal value is -24296; the CPU time allocated to Part100 is 10 seconds;
- Slater200 is an instance of the linear ordering problem; the number of vertices of the tournament is 200; the orientation of the arcs is drawn randomly with a probability equal to 0.5 for each orientation; all the weights are equal to 1 (thus it is in fact an instance of Slater problem); the optimal value is equal to 8163; the CPU time allocated to Slater200 is 20 seconds;
- Tsp100 is an instance of the travelling salesman problem with 100 vertices; the weights are uniformly chosen between 1 and 100; the optimal value is 282; the CPU time allocated to Tsp100 is 100 seconds;
- Grid1024 is an instance of the Euclidean travelling salesman problem with 1024 vertices located on the crosses of a regular square-grid with a side equal to 0.96875 so that the optimal value is 32; the CPU time allocated to Grid1024 is 100 seconds.

Figure 3 displays the results obtained by 100 applications of STNM-V, of STNM-D, and of repeated descents (RD) to these instances; the three methods have the same CPU time (see the values above). The horizontal axis shows the 100 trials, the vertical axis shows the values of f found by each one of the 100 trials; these ones are sorted so that the values of f are increasing. We may notice also that the efficiency of the self-tuning is about the same for the two ways of noising (the two lines of STNM-D and STNM-V are often superposed, even if STNM-D seems to be slightly better than STNM-V for the Euclidean TSP). Figures 4, 5 and 6 summarize the best, average and worst results found by RD, STNM-D and STNM-V for the four instances of Figure 3. Figure 7 provides the best, average and worst results obtained for STNM-V over 100 trials when applied to several instances from TSPLIB. Sometimes STNM-D and STNM-V found an optimal solution within the allocated CPU time. When so, we specify between parentheses how many times the optimal value has been found in the column “best found value” (notice that STNM-D and STNM-V find optimal solutions more and more often when the CPU times increase); for instance, for Part100, STNM-D and STNM-V found an optimal solution (-24296) 52 times over the 100 trials. These examples show that STNM-D and STNM-V do not always provide optimal solutions (what was obvious to foresee), but anyway pretty good ones, even with short CPU times (what was not so obvious to foresee).

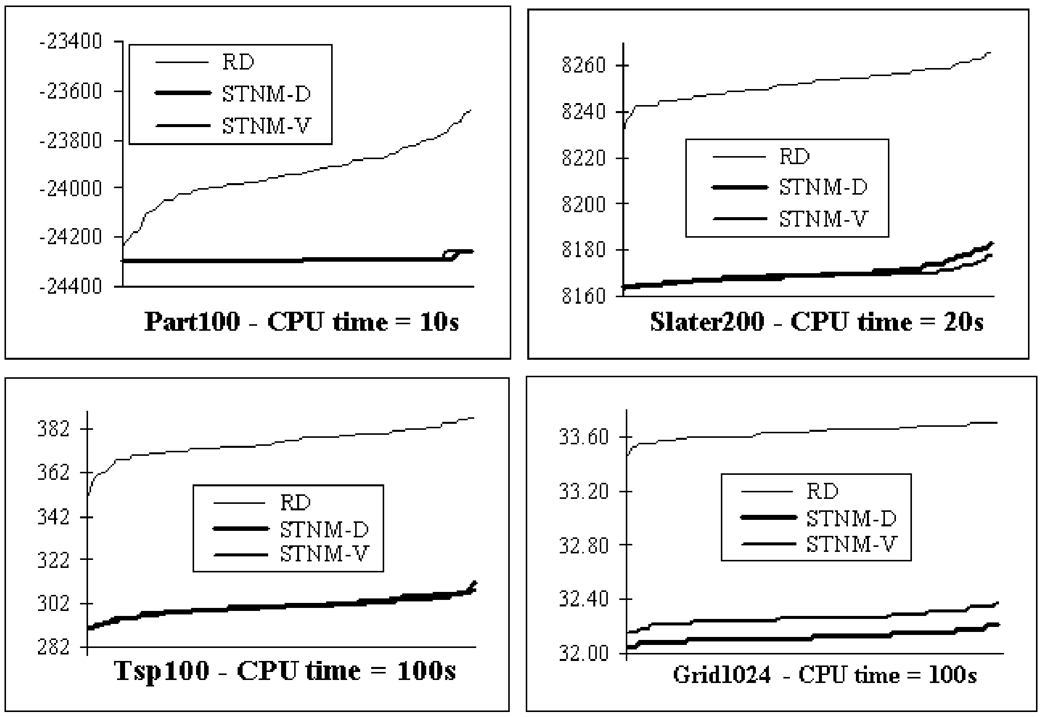

Figure 3. Experimental results for the repeated descents (RD), for the self-tuned noising method when the variations of the objective function f are noised (STNM-V), and for the self-tuned noising method when the data of the instance are noised (STNM-D), for a representative instance of each one of the four studied problems.

Instance	best value	best found value	average	worst	CPU time
Part100	−24296	−24237	−23934.96	−23677	10 s
Slater200	8163	8231	8252.41	8266	20 s
Tsp100	282	352	375.79	387	100 s
grid1024	32	33.48	33.64	33.71	100 s

Figure 4. Best, average and worst results for RD over 100 trials, for a representative instance of each one of the four studied problems.

Instance	best value	best found value	average	worst	CPU time
Part100	−24296	−24296 (52)	−24292.29	−24254	10 s
Slater200	8163	8164	8169.98	8183	20 s
Tsp100	282	291	300.5	311	100 s
grid1024	32	32.05	32.12	32.20	100 s

Figure 5. Best, average and worst results for STNM-D over 100 trials, for a representative instance of each one of the four studied problems.

Instance	best value	best found value	average	worst	CPU time
Part100	−24296	−24296 (52)	−24290.98	−24254	10 s
Slater200	8163	8163 (1)	8168.33	8178	20 s
Tsp100	282	291	299.67	308	100 s
grid1024	32	32.15	32.25	32.36	100 s

Figure 6. Best, average and worst results for STNM-V over 100 trials, for a representative instance of each one of the four studied problems.

Graph	best value	best found value	average	worst	CPU time
ei151	426	426 (83)	426.17	427	2 s
kroC100	20749	20749 (94)	20749.56	20769	2 s
lin105	14379	14379 (75)	14389.52	14449	2 s
ch150	6528	6528 (42)	6539.68	6555	20 s
bier127	118282	118282 (80)	118298.84	118799	60 s
kroA200	29368	29368 (56)	29378.82	29445	60 s
kroB200	29437	29437 (44)	29439.82	29472	300 s
pr144	58537	58537 (18)	58761.3	59865	300 s
kroB150	26130	26130 (2)	26132.08	26136	900 s
rd400	15281	15281 (2)	15298.69	15340	900 s
tsp225	3916	3916 (13)	3939.09	3959	900 s
pcb442	50778	50785 (1)	50800.41	50827	1800 s
vm1084	239297	240162	241284.71	242563	3600 s

Figure 7. Best, average and worst results for STNM-V over 100 trials, for several instances of the TSP from TSPLIB.

Because of the NP-hardness of the studied problems, it is difficult to know an optimal solution and, thus, to measure how far the solution found by STNM is with respect to an optimal one (moreover, this concerns more the efficiency of NM, which is not discussed in this paper – see the references given in [10] for such discussions –, than the efficiency of a self-tuning of these methods, which is what we study here). When an optimal solution is known, we see that the solution found by STNM is quite often near it or equal to it. Moreover, for the linear ordering problem, we applied STNM-D and an exact branch-and-bound algorithm (the one described in [12] and freely available at the address http://www.infres.enst.fr/∼charon/tournament/median.html; this software includes a part devoted to STNM for the linear ordering problem) to 5790 instances with different characteristics and with a number of vertices up to 100 (what is already large for an exact method; to be more specific, dealing with an instance similar to Slater200 but with only 31 vertices needed an average CPU time of 2635 seconds; the exponential increase of CPU time was such that we could not try larger tournaments with the same features as Slater200 when we dealt with Slater problem in [12]): in these experiments, STNM found exact solutions for 5784 instances; for the 6 remaining instances, the solutions found by STNM were almost optimal (and a second run of STNM succeeded in finding an optimal solution).

5.2. *Global conclusions from the experiments*

We can draw the following qualitative conclusions from the experiments (once again arising from several thousands of instances for each problem, but not detailed here): the initial noising rate r_{max} of the last classic NM N_q of the series can be considered as the one found manually by an expert after successive trials and usually leads to very good solutions (optimal or almost).

It is noticeable that, for a given instance of a given problem, the tuning of r_{max} is steady and does not present accident (i.e., a tuning which would lead to an inefficient NM), as shown by Figure 3; the probability to fail seems very low.

About the efficiency of STNM, as the automatic tuning of r_{max} by STNM leads to values that we could obtain by a manual tuning done by an expert, we usually obtain solutions quite similar to the ones provided by a classic NM which would be well tuned, and thus much better than the ones of repeated descents (what was not at all obvious at the beginning).

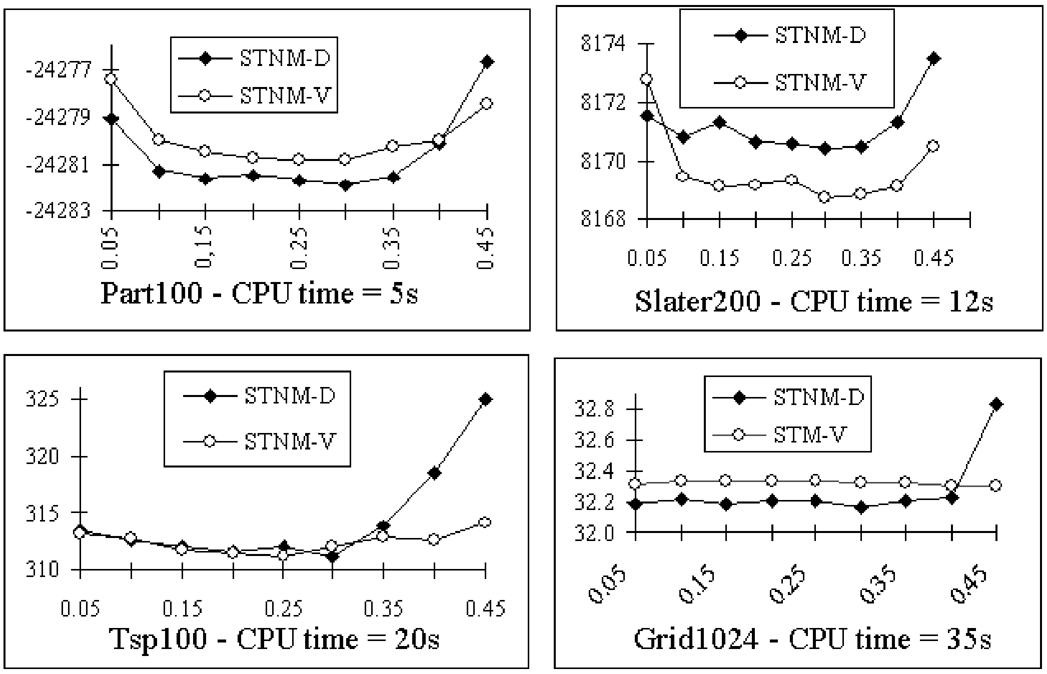

Figure 8. Study on the choice of 30 % to limit the percentage π of accepted elementary transformations. The horizontal axis corresponds to the tried values instead of 30 %; the vertical axis shows the average of the values taken by the objective function f over 100 trials, for a representative instance of each one of the four studied problems.

About the CPU time consumed by STNM with respect to the one consumed by a classic NM, let's say N, with a good initial noising rate (specifically, with the noising rate provided by STNM at its end), the design of STNM (doubling the CPU time of each classic NM of the series) is such that we are sure to reach a solution of same quality by STNM as by N within at most twice the CPU time allocated to N, since the last NM of the series applied by STNM gets at least half the total CPU time allocated to STNM. The experiments show in fact that, quite often, STNM finds a comparable solution within the same CPU time. But in the case of a classic NM N, it is then necessary to tune the parameters (at least r_{max}), which always takes much CPU time because, as NM is not deterministic, it is necessary to make statistics and thus to run N several times before finding a good tuning.

5.3. *Discussion on the choice of 30 % for the percentage of accepted transformations*

In designing STNM, we limit several times the percentage π of accepted elementary transformations to 30 %. We tried other values instead of 30 % to know how robust the choice of 30 % is. Figure 8 shows the effect of other values for the previous four instances Part100, Slater200, Tsp100 and Grid1024. But, once again, we tried many other instances for the four problems, with different characteristics: the results are quite similar to the ones displayed by Figure 8. The horizontal axis of Figure 8 corresponds to the tried values instead of 30 %: from 0.05 to 0.45, with a step of 0.05; the vertical axis shows the average of the values taken by f over 100 trials.

From the experimental results, it appears that other choices than 30 % are possible, but not basically better. It would be risky to adopt a too large value, typically larger than 40 %: such a percentage would lead to too large initial noising rates

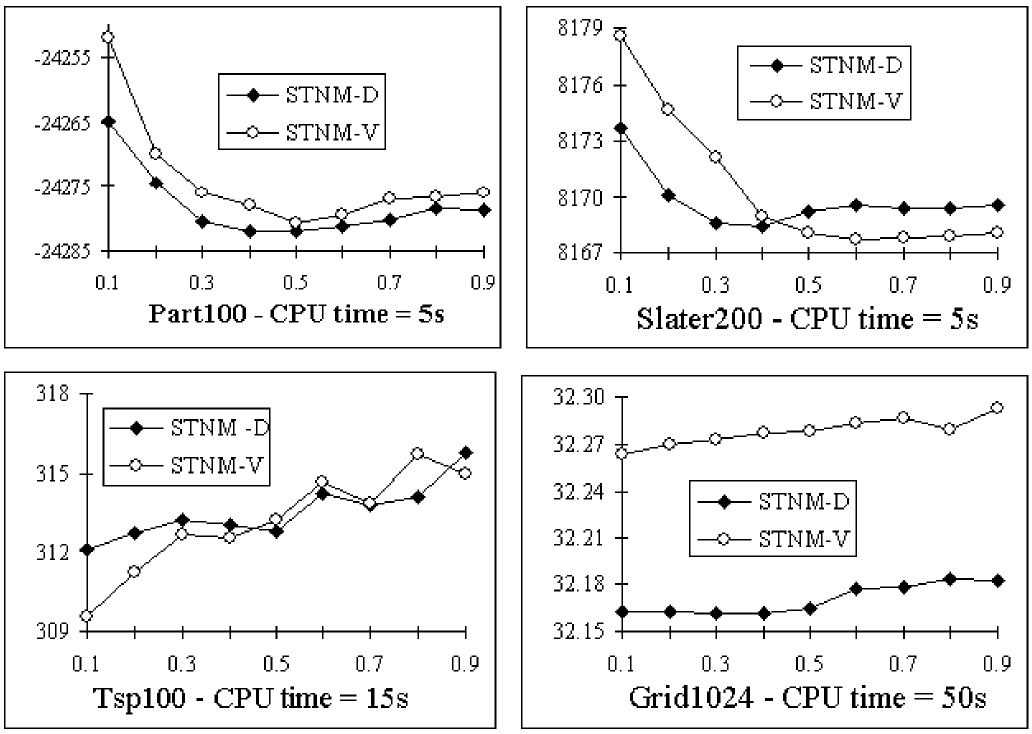

Figure 9. Evolution of the average of the values taken by the objective function f over 100 trials when the parameter α (providing the value of the parameter *threshold*: $threshold = \alpha \times minDesc + (1-\alpha) \times bestValue$, where $minDesc$ and $bestValue$ denote respectively the best value found by a descent and the best value found by STNM since the beginning) varies from 0.1 to 0.9 with a step equal to 0.1, for a representative instance of each one of the four studied problems.

and, then, too many unfavourable transformations would be accepted, what wastes CPU time. Similarly, a too low percentage, let's say, below 10 %, can be a worse choice than 30 % by restricting the values of the initial noising rate too much. But, comparing the values of Figure 8 to the ones of Figure 3 shows that, even for "bad" percentages, the results obtained are quite good (and in all cases much better than the ones provided by repeated descents). Thus the choice of a sharp percentage does not seem to be crucial and a value around 30 % seems reasonable.

5.4. *Expression of threshold in function of minDesc and bestValue*

As described in Section 4, *threshold* is a variable involved in the computation of the initial noising rate r_{max} of the classic NM of the series performed by STNM. After different attempts in order to try several formulae, we adopted a linear expression for *threshold* in function of *minDesc* and $bestValue$:

$$threshold = \alpha \times minDesc + (1-\alpha) \times bestValue$$

with $0 \le \alpha \le 1$ (hence, the arithmetic mean adopted above corresponds with $\alpha = 0.5$). The lower α, the lower *threshold* and thus the lower r_{max} (more precisely, the lower the increase of r_{max}). Figure 9 shows the evolution of the average of the values taken by f over 100 trials when STNM is applied to the four previous instances with α varying from 0.1 to 0.9 with a step equal to 0.1.

For the first two problems, the choice of the arithmetic mean seems quite appropriate. For the other two, it remains a good choice, as well as other possibilities, except for STNM-V when applied to the TSP: here it seems that a value lower than 0.5 for α would be better. Notice anyway that, for certain instances of TSP, low values of α, and thus low values of *threshold*, lead STNM-V to bad results, comparable with the ones of repeated descents. On the contrary, choosing a value greater than 0.5 for α usually does not lead to bad results in average, but may consume CPU time useless: in this case, at the beginning of the method, r_{max} is too large and the noised functions are too far from f to be interesting; it is then necessary to wait that the noising rate reaches lower values during the noising process to compute interesting solutions. It is why we decided to adopt the arithmetic mean to compute *threshold*.

6. Conclusion

It is not an easy task to design a metaheuristic with only one parameter, the CPU time that the user wants to spend in order to solve his or her problem, and this field is still in its infancy. The difficulty arises from different sources. A first source relies in the difficulty to imagine ways to automate the tuning, especially if we want to obtain a method that we could apply to several problems; and even for a given problem, it is surely not possible to design a method utterly automatic and simultaneously optimal for all the instances. A second difficulty arises from the validation of the automatic tunings: experiments are always open to criticism, especially with stochastic methods since there is no optimal tuning in this case; comparing the results (especially the CPU time) with manually tuned methods is not fair, since it is impossible to estimate the CPU time necessary to manually tune the parameters (which depends more on the ability of the user than on the method itself); while theoretical studies usually fails to give practical information on how to choose the precise values of numerical parameters (and even sometimes on how to design more structural components of metaheuristics). By this study, we try to show that the noising methods can be automatically tuned, and even that they can tune themselves during their runs.

The game is worth the candle. Indeed, metaheuristics are powerful tools to tackle difficult problems; but a main problem arises from the necessity to tune their parameters, while it is usually difficult to get an idea of their appropriate values (sometimes, even broadly). In some cases, it is possible to substitute a parameter to another which can be easier to tune (as it has been suggested to compute the initial temperature of simulated annealing; see [1] for references); but it is nonetheless necessary to tune this new parameter. The tuning of the parameters can be long, tedious and should be done for each instance of the problem (or, at least for each family of instances sharing common features). It often consumes much CPU time that we could devote to solve the instance better. We think that a next step in the development of metaheuristics will precisely be the design of automatically tuned schemes or, still better, of self-tuned schemes. These automatic tunings can be used directly to run the considered methods, as we did in STNM; but it is also possible to use them as a help to an expert: in this case, automatically tuned methods can be run to compute broad values of the parameters that an expert can start with and sharpen, without wasting time in order to find these starting values.

By this paper, we try to contribute to this aim of developing the field of self-tuned metaheuristics. It is surely possible to improve several aspects. This could be done by polishing different aspects that we introduced in STNM. This could be done also by introducing new aspects in our STNM. For instance, the final noising rate has been set to 0 because it is more convenient. But, in many cases, it is useless to make the noising rate decrease until 0: we only waste time. Thus we could improve STNM by tuning also this parameter. Another possibility would be to deal with some variants of NM. For instance, we recalled at the beginning that the noising methods provide a generalization of simulated annealing (SA). Thus, it is possible to design STNM in order to give a self-tuned version of SA. Even if the design of STNM-V is a first attempt in this direction, we did not try to draw benefit from the specificities of SA in this paper. Many possibilities are still open to investigations; some of them will be the object of our further research. We hope that the reader will find, by this preliminary study, that it is worth doing and that the automatic tuning of metaheuristics (of course, not only the noising methods) is a field which deserves consideration. Indeed, it is very convenient to be able to apply a metaheuristic without tuning any parameter!

References

[1] E.H.L. Aarts and J.K. Lenstra (eds.), Local search in combinatorial optimization, Wiley, New York, 1997.

[2] J.-P. Barthélemy and B. Monjardet, *The median procedure in cluster analysis and social choice theory*, Mathematical Social Sciences 1, 1981, 235-267.

[3] J.-P. Barthélemy and B. Leclerc, *The median procedure for partitions*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 19, 1995, 3-34.

[4] E. Bonomi and J.-L. Lutton, *The N-city travelling salesman problem: statistical mechanics and the Metropolis algorithm*, SIAM Review 26, 1984, 551-568.

[5] I. Charon and O. Hudry, *The noising method: a new combinatorial optimization method*, Operations Research Letters 14, 1993, 133-137.

[6] I. Charon and O. Hudry, *Mixing different components of metaheuristics*, in: I.H. Osman and J.P. Kelly (eds), Metaheuristics: Theory and Applications, Kluwer Academic Publishers, Boston, 1996, 589-603.

[7] I. Charon and O. Hudry, *Lamarckian genetic algorithms applied to the aggregation of preferences*, Annals of Operations Research 80, 1998, 281-297.

[8] I. Charon and O. Hudry, *Application of the noising method to the Travelling Salesman Problem*, European Journal of Operational Research 125/2, 2000, 266-277.

[9] I. Charon and O. Hudry, *The noising methods: a generalization of some metaheuristics*, European Journal of Operational Research 135 (1), 2001, 86-101.

[10] I. Charon and O. Hudry, *The noising methods: a survey*, in P. Hansen, C.C. Ribeiro (eds), Essays and Surveys in Metaheuristics, Kluwer Academic Publishers, 2002, 245-261.

[11] I. Charon and O. Hudry, *Noising methods for a clique partitioning problem*, Discrete Applied Mathematics 154 (5), 2006, 754-769.

[12] I. Charon and O. Hudry, *A branch and bound algorithm to solve the linear ordering problem for weighted tournaments*, Discrete Applied Mathematics 154, 2006, 2097-2116.

[13] I. Charon and O. Hudry, *A survey on the linear ordering problem for weighted or unweighted tournaments*, 4OR 5 (1), 2007, 5-60.

[14] I. Charon and O. Hudry, *Application of the "descent with mutations" metaheuristic to a clique partitioning problem*, proceedings of 2007 IEEE International Conference on Research, Innovation and Vision for the Future, Proceedings of the Institute of Electrical and Electronics Engineers, Vietnam, 2007, 29-35.

[15] K. A. Dowsland, *Simulated annealing*, in C. Reeves (ed), Modern heuristic techniques for combinatorial problems, McGraw-Hill, London, 1995, 20-69.

[16] J. Dréo, A. Pétrowski, É. Taillard and P. Siarry, *Metaheuristics for Hard Optimization, Methods and Case Studies*, Springer, 2006.

[17] G. Dueck and T. Scheurer, *Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing*, Journal of Computational Physics 90, 1990, 161-175.

[18] G. Dueck and J. Wirsching, *Threshold accepting algorithms for multi-constraint 0-1 knapsack problems*, Technical paper, TR 89 10 016, IBM Heidelberg Scientific Center, Germany, 1989.

[19] F.W. Glover and G.A. Kochenberger, *Handbook of Metaheuristics*, Kluwer Academic Publishers, International series in operations research and management science, Boston Hardbound, 2003.

[20] D. Guillaume and F. Murtagh, *An application of XML and XLink using a graph-partitioning method and a density map for information retrieval and knowledge discovery*, in: D.M. Mehringer, R.L. Plante, and D.A. Roberts (eds), Astronomical data analysis software and systems VIII, ASP Conf. Ser. 172, ASP, San Francisco, 1999, 278-282.

[21] D. Guillaume and F. Murtagh, *Clustering of XML documents*, Computer Physics Communications 127, 2000, 215-227.

[22] G. Gutin, A.P. Punnen, *Traveling Salesman Problem and its Variations*, Kluwer, Dordrecht, 2002.

[23] P. Hansen and B. Jaumard, *Cluster analysis and mathematical programming*, Mathematical Programming 79, 1997, 191-215.

[24] P. Hansen, C.C. Ribeiro (eds), *Essays and Surveys in Metaheuristics*, Kluwer Academic Publishers, 2002.

[25] H.H. Hoos, T. Stützle, *Stochastic Local Search*, Foundations and Applications, Morgan Kaufmann / Elsevier, 2004.

[26] O. Hudry, *NP-hardness results on the aggregation of linear orders into median orders*, to appear in the Annals of Operations Research.

[27] C.-P. Hwang, *Global and local search heuristics for the symmetric travelling salesman problems*, PhD thesis, University of Mississippi, 1996.

[28] C.-P. Hwang, B. Alidaee and J.D. Johnson, *A tour construction heuristic for the travelling salesman problem*, Journal of the Operational Research Society 50, 1999, 797-809.

[29] T. Ibaraki, K. Nonobe, M. Yagiura (eds.), *Metaheuristics: Progress as Real Problem Solvers*, Operations Research/Computer Science Interfaces, vol. 32, Springer, Berlin, Heidelberg, New York, 2005.

[30] M. Jünger, *Polyhedral combinatorics and the acyclic subdigraph problem*, Heldermann Verlag, Berlin, 1985.

[31] M. Jünger, G. Reinelt and G. Rinaldi, *The Traveling Salesman Problem*, Handbooks in OR and MS vol. 7, M.O. Ball et al. (eds), Elsevier Science, 1995, 225-330.

[32] G. Laporte and I.H. Osman, *Metaheuristics: a bibliography*, Annals of Operations Research 63, 1996.

[33] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Schmoys (ed.), *The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization*, John Wiley and Sons, 1985.

[34] S. Lin, *Computer solutions for the travelling salesman problem*, Bell Syst. Tech. J. 44, 1965, 2245-2269.

[35] I.H. Osman and J.P. Kelly (eds), *Meta-heuristics: theory and applications*, Boston, Kluwer Academic Publishers, 1996.

[36] D.T. Pham and D. Karaboga, *Intelligent optimisation techniques. Genetic Algorithms, Tabu Search, Simulated Annealing and Neural Networks*, Springer, 2000.

[37] M. Pirlot, *General local search methods*, European Journal of Operational Research 92 (3), 1996, 493-511.

[38] C. Reeves (ed), *Modern heuristic techniques for combinatorial problems*, McGraw-Hill, 1995, London.

[39] S. Régnier, *Sur quelques aspects mathématiques des problèmes de classification automatique*, I.C.C. Bulletin 4, 1965, Rome.

[40] G. Reinelt, *The linear ordering problem: algorithms and applications*, Research and Exposition in Mathematics 8, Heldermann Verlag, 1985, Berlin.

[41] G. Reinelt, *TSPLIB - A Traveling Salesman Problem Library*, ORSA Journal on Computing 3 (4), 1991, 376-384, http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.

[42] M.G.C. Resende and J. Pinho de Sousa (eds.), *Metaheuristics: Computer Decision-Making. Applied Optimization*, vol. 86, Kluwer Academic Publishers, Boston, USA, 2003.

[43] S. Sait and H. Youssef, *Iterative computer algorithms with applications in engineering*, IEEE Computer Society Press, 1996.

[44] P. Slater, *Inconsistencies in a schedule of paired comparisons*, Biometrika 48, 1961, 303-312.

[45] V. Sudhakar and C. Siva Ram Murthy, *A modified algorithm for the graph partitioning problem*, Integration, the VLSI journal 22, 1997, 101-113.

[46] S. Voss, S. Martello, I.H. Osman and C. Roucairol, *Metaheuristics: advances and trends in local search paradigms for optimization*, Kluwer Academic Publishers, 1998.

[47] Y. Wakabayashi, *Aggregation of binary relations: algorithmic and polyhedral investigations*, PhD Thesis, Augsbourg, 1986.