Simulated Annealing and Tabu Search for Constraint Solving

Jin-Kao Hao and Jêrome Pannier
LGI2P/EMA-EERIE
Parc Scientifique Georges Besse
F-30000 Nîmes, France
email: {hao, pannier}@eerie.fr

Fifth Intl. Symposium on Artificial Intelligence and Mathematics (AIM'98)
Fort Lauderdale, Florida, USA, January 4-6, 1998

Abstract

In this paper, we present an experimental study of local search for constraint solving. For this purpose, we experiment with two algorithms based on Simulated Annealing (SA) and Tabu Search (TS) for solving the maximal constraint satisfaction problem. These two algorithms were tested on various large random instances going from 100 to 300 variables with 10 to 30 values per variable. Experimental results show that the TS algorithm dominates the SA algorithm on all the tested instances in terms of solution quality and solving speed. We propose empirical arguments to explain the difference of performances.

Keywords: Local Search, Simulated Annealing, Tabu Search, Constraint Solving.

1 Introduction

Constraint solving either for satisfaction or optimization purpose occupies a very important place in Artificial Intelligence (AI) and Mathematics. Informally, constraint solving consists in finding assignment of values to variables while respecting some constraints and eventually optimizing a cost function.

Constraint problems embody many well-known problems such as graph coloring and satisfiability, and numerous practical applications related to resource assignment, planning, scheduling and so on. Constraint problems are in general NP-complete or NP-hard. Therefore, it is unlikely there exists any efficient solution for these problems. Given their theoretical and practical importance, many efforts have been devoted to constraint solving in recent years.

A constraint solving method is either *complete* (exact for optimization) which guarantees the completeness of the resolution or *incomplete* (non-exact for optimization) which sacrifices the completeness to gain efficiency. A complete method for constraint satisfaction explores systematically, though very often implicitly, the whole search space. To do this, most complete methods construct in a step-by-step way a solution and backtrack in case of a failure [7, 17, 20, 25]. These methods use usually various heuristics to guide the choice of the next variable to be instantiated and its value, and employ powerful filtering techniques to achieve different levels of consistency. Similarly, exact methods for constraint optimization are usually based on the branch-and-bound principle and try to eliminate heuristically as many as possible solutions not leading to an optimum [4, 16, 26]. Complete and exact methods have in general exponential time complexity. The solving time required by such a method may thus become prohibitory for large size problems.

An incomplete (non-exact) method does not explore systematically the whole search space. Instead, it tries to examine as rapidly as possible a large number of search points according to a selective or random strategy. Local search (LS) is one of the most popular examples of this family of methods. In general, these methods don't guarantee the completeness of the resolution, but require no exponential time complexity. They constitute in fact a very interesting alternative for practical solving of many hard and large size problems.

Local search has been applied to particular classes of constraint problems such as graph coloring [9, 1, 13] and the maximal satisfiability problem (Max-SAT) [6]. The work presented in [18] was one of the first efforts in AI to apply local search (called "repair method" in [18]) to constraint satisfaction. The proposed "min-conflicts" heuristic realizes a descent following a specific neighborhood. The work presented in [22] gives a successful application of the same heuristic to the satisfiability problem. In [23], the "min-conflicts" heuristic is extended by introducing the "random-walk" strategy which proves to be highly effective. Other extensions and improvements of repair heuristics are reported in [19, 8, 27].

In this work, we are interested in practical solving of general constraint problems by two well-known local search meta-heuristics: Simulated Annealing (SA) and Tabu Search (TS). In particular, we develop SA and TS algorithms for the maximal constraint satisfaction problem. We carry out empirical comparisons of these algorithms on large random instances and study their behaviors. This work constitutes a continuation of the study reported in [11].

The paper is organized as follows. After the introduction of the constraint problems in Section 2, we review briefly the principles of SA and TS in Section 3. We present the SA and TS algorithms for constraint solving in Section 4. We give experimental results in Section 5 and we conclude the paper in the last section.

2 Constraint Problems

Constraint problems can be conveniently defined with the notion of *constraint network*. A constraint network *CN* is a triplet $< V, D, C >$ where:

- V = $\{V_1, V_2 ... V_n\}$ is a finite set of variables XS;
- D = $\{D_1, D_2 ... D_n\}$ is a finite collection of value domains associated to the variables;
- C = $\{C_1, C_2 ... C_n\}$ is a set of constraints, each being a subset of the Cartesian product of the domains of some variables specifying allowed (or forbidden) value combinations.

Given a constraint network $CN < V, D, C >$, different problems can be defined. First, the associated *constraint satisfaction problem (CSP)* consists in finding one or more assignments of the values of D to the variables of V such that all the constraints of C are satisfied simultaneously[1] [17]. Second, the *maximal constraint satisfaction problem (MCSP)* is an optimization problem which looks for assignments which maximize the number of satisfied constraints (or minimize the number of violated constraints) [4]. Finally, it is possible to associate to each constraint a weight and to define the *weighted maximal constraint satisfaction problem (WMCSP)* for which we look for assignments which maximize the total weight of

[1] Given the incompleteness of local search, we are not interested in the problem of solubility.

satisfied constraints (or minimize the total weight of violated constraints). Clearly, a MCSP is a WMCSP whose constraints have the same weight of 1.

In this work, we are essentially interested in the solving of MCSPs. It is easy to show that the approach presented here is directly applicable to CSPs and WMCSPs.

3 Local Search

3.1 Introduction

Local search, or neighborhood search represents a very important class of incomplete, heuristic-based methods for combinatorial optimization [21]. Traditionally, local search constitutes a powerful tool to tackle well-known hard problems such as the Traveling Salesman Problems and many applications. Over the last ten years, impressive progresses have been achieved due to the discovery of a class of new meta-heuristics. These new meta-heuristics are based on some general concepts and are applicable to a wide range of problems. SA and TS are the most representative examples of these meta-heuristics. Thanks to these modern meta-heuristics, one is able to deal with today much larger classical combinatorial problems as well as many applications intractable before [15].

To apply local search to a problem instance *(S,f)* defined by a search space S and a cost function f, one needs first a neighborhood function.

- **Definition**: Let S be the set of the configurations of a given instance, a *neighborhood* over S is any function $N : S \rightarrow 2^S$. A configuration s is a *local minimum* with respect to N if $f(s) \leq f(s')$ for all $s' \in N(s)$.

A typical LS procedure begins with an initial configuration in S, and carries out a series of *moves* consisting in replacing the current solution by one of its neighbors, often taking into account the cost function f. This iterative process stops and returns the best configuration encountered when some predefined conditions are satisfied, for instance, when a fixed number of iterations or moves is reached or a sufficiently good configuration is found. For a given neighborhood, different LS methods employ different strategies to search the neighborhood.

3.2 Simulated Annealing

Simulated annealing is an advanced LS method which finds its inspiration from the physical annealing process studied in statistical mechanics [14]. A SA algorithm repeats an iterative repairing procedure which looks for better configurations while offering the possibility of accepting in a controlled manner worse configurations. It is this second feature that allows SA to escape from local optima.

More precisely, at each iteration, a neighbor $s' \in N(s)$ of the current configuration s is generated randomly and a decision is then taken to decide whether s' will replace s. If s' is better than s *i.e.* $\Delta = f(s') - f(s) \leq 0$ (for minimization), we move from s to s', otherwise, we move to s' with the probability $e^{(-\Delta)/t}$. This probability depends on two factors: 1) the degree of the degradation Δ (smaller the degradation, greater the accepting probability), and 2) a control parameter t called temperature (higher temperatures lead to higher accepting probabilities and vice versa).

The temperature is controlled by a cooling schedule specifying how the temperature should be progressively reduced. Typically, a SA algorithm stops when a fixed number of non-improving iterations is realized with a temperature or when a limit of iterations is reached.

The performance of a SA algorithm depends critically on the cooling schedule used. There exist theoretical schedules guaranteeing asymptomatic convergence of the algorithm towards an optimal solution. However, these schedules are not practicable since they require an infinite computing time. This is why much simpler schedules are preferred in practice even if they don't guarantee an optimal solution. One example consists in decreasing the temperature by steps, each temperature being kept for a certain number of iterations before being reduced. In such a case, the cooling schedule is entirely defined by 2 functions for reducing the temperature and for defining the number of iterations for each temperature.

3.3 Tabu Search

Tabu Search is another advanced LS meta-heuristic. Compared with SA where randomness is extensively used, TS is based on the belief that intelligent searching should embrace more systematic forms of guidance such as memorizing and learning. A typical TS algorithm begins with an initial configuration s in S and then proceeds iteratively to visit a series of locally best configurations following the neighborhood function N. At each iteration, a *best* neighbor $s' \in N(s)$ is sought to replace the current configuration even if s' is no better than the current configuration in terms of the cost function. To avoid the problem of possible cycling and to allow the search to go beyond local optima, TS introduces the notion of *tabu list*, one of the most important components of the method.

A tabu list is a special short term memory that maintains a selective history H, composed of previously encountered solutions or more generally pertinent attributes of such solutions. A simple TS strategy based on this short term memory H consists in preventing solutions of H from being reconsidered for next k iterations (k, called tabu tenure, is problem dependent). Now, at each iteration, TS searches for a best neighbor from this dynamically modified neighborhood $N(H,s)$, instead of $N(s)$ itself. Such a strategy prevents Tabu from being trapped in short term cycling and allows the search process to go beyond local optima.

Tabu restrictions may be overridden under certain conditions, called *aspiration criteria*. Aspiration criteria define rules that govern whether a solution may be included in $N(H,s)$ if the solution is classified tabu. One widely used aspiration criterion consists of removing a tabu classification from a move when the move leads to a solution better than the best obtained so far. Other forms of criteria may be defined over subsets of solutions that belong to common regions or that share specified features. Aspiration constitutes an important element of flexibility in TS.

There are other interesting and important techniques available such as intensification and diversification [5]. In this paper, we show that a basic Tabu algorithm based on the above mentioned elements may be very effective and robust.

TS uses an aggressive search strategy to exploit its neighborhood. Therefore, it is crucial to have special data structures and techniques which reduce the effort of finding best moves and allow fast updatings after a move.

4 SA and TS Algorithms for Constraint Solving

In order to present our SA and TS algorithms for constraint problems, we describe first the common components of the two algorithms: problem encoding (configuration), search space, neighborhood and cost function.

4.1 Common Components

For the constraint problems MCSP, CSP and WMCSP, a simple and natural configuration structure can be defined as follows:

- **Configuration**: Given a constraint network $< V, D, C >$, a configuration s is any assignment defined by $s = \{< V_i, v_i > \mid V_i \in V$ et $v_i \in D_i\}$.

Therefore, the search space S is composed of all the possible assignments of the network. It is easy to see there are $\prod_{i=1}^{n} |D_i|$ configurations in S. From this configuration, a neighborhood function can be introduced as follows.

- **Neighborhood**: Let s be a configuration in S, the *neighborhood* $N : S \to 2^S$ is such an application that for each $s \in S$, $s' \in N(s)$ if and only if s and s' are different at the value of a single *conflicting* variable[2].

Using this neighborhood, a neighbor in $N(s)$ can be easily obtained by changing simply the current value of a conflicting variable in s. A move can be thus characterized by the couple $< variable, value >$. It should be clear that the size of this neighborhood varies during the search according to the number of conflicting variables.

- **Cost function**: The cost function f to be minimized is defined as follows:

$$f(s) = \sum_{i=1}^{|C|} p_i * \chi(C_i),\ C_i \in \mathrm{C} \text{ and } p_i = \text{the weigth of } C_i$$

$$\chi(C_i) = \begin{cases} 1 \text{ if } C_i \text{ is vioalted} \\ 0 \text{ otherwise} \end{cases}$$

For a MCSP (a WMCSP respectively), $f(s)$ corresponds to the number (weighted number respectively) of violated constraints by the configuration s. A CSP can then be simply dealt with as a MCSP having a optimal cost of zero.

4.2 SA Algorithm

We present now the remaining components of the SA algorithm: configuration evaluation, functions to determine temperatures and step lengths. The general algorithm is given in Figure 1.

- **Configuration evaluation**: Each time a neighbor $s' \in N(s)$ is picked randomly, the cost difference $\Delta = f(s') - f(s)$ must be calculated. To do this, it is sufficient to re-examine only the constraints containing the modified variable. This has a time complexity of $O(|V| * |D|)$ in the worst case. In practice, this complexity is much lower and bounded by the density of the constraint network.

- **Temperature function** σ: A temperature function allows a series of decreasing positive values to be calculated. After having tested several functions, the following one seems to be appropriate σ: $T \to T$ where $T \subset \mathbf{R}$ is the set of possible temperatures. More precisely, let $t \in T$ be the current temperature, one calculates (at iteration i) the next temperature by $\sigma(t) = t*(1-\frac{A}{i})$ where A is a control parameter. With this function, the reduction rate of temperatures is slowed down progressively when the search progresses.

[2] A variable is said to be conflicting if it is implied in some unsatisfied constraints.

- **Step length function** ω: This function is defined as follows: ω: $L \rightarrow L$ where $L \subset \mathbf{Z}^+$ is the set of possible lengths. Formally, let l be the current length, the next length is determined by $\omega(l) = l * (1 + \frac{A}{i})$ where A is the same parameter as the one used in the temperature function. Using this function, the number of iterations per temperature increases progressively when the search progresses (when the temperature decreases).

SA Algorithm

```
begin
| Pick randomly an initial configuration s0;
| Determine an initial temperature t0;
| Determine an initial step length l0;
| nb_iter ← 0; nb_mv ← 0 ;
| s ← s0; s* ← s; t ← t0;
| l ← l0 ; l_count ← l;
| while ( (nb_mv < max) and (f(s) ≠ 0) )
| do
| |  while (l_count ≠ 0) do
| |  |  Pick randomly s' ∈ N(s) such as s' ≠ s;
| |  |  Δ = f(s') - f(s);
| |  |  if ( (Δ ≤ 0) Or ( (Δ > 0) and (Probability e^((-Δ)/t) is verified) ) ) then
| |  |  |  s ← s';
| |  |  |  if ( f(s) ≤ f(s*) ) then
| |  |  |  |_ s* ← s;
| |  |  |_ nb_mv ← nb_mv + 1;
| |  |  nb_iter ← nb_iter + 1;
| |  |_ l_count ← l_count − 1;
| |  t ← σ(t);
| |  l ← ω(l);
| |_ l_count ← l;
| output(s*);
end
```

Figure 1 : *SA algorithm for constraint solving*

Let us make three remarks about the SA algorithm. First, the algorithm stops when the given number of moves (max) is reached or when the cost value zero (0) is found (the given instance is then satisfiable). Second, in order to decide if a worse neighbor is accepted, a random number $r \in$ (0,1] is generated and then compared with the probability $e^{(-\Delta)/t}$. Third, an iteration here does not always lead to a move. Indeed, the number of iterations for a move increases in general when the temperature decreases and when there are fewer better neighbors in the neighborhood.

4.3 TS Algorithm

We introduce now the components of the TS algorithms: configuration evaluation, tabu list, and aspiration criterion. The general TS algorithm is given in Figure 2.

- **Configuration evaluation**: At each iteration, TS looks for a best neighbor in *N(s)* to make a move. It is therefore essential to be able to evaluate the neighbors in *N(s)* quickly. To do this, we use a technique inspired by [3]. This technique is based on a $|V|$*$|D|$ matrix δ where each element $\delta[i,j]$ indicates the cost variation (called move value) if the corresponding move $< V_i, v_j >$ is made. Therefore, the cost *f(s')* of any *s'* $\in$ *N(s)* can be obtained by a simple summation of the cost *f(s)* and the value of the corresponding element in δ. To obtain a best neighbor, it is now sufficient to search the δ matrix in time $O(|N(s)|)$. After a move, the matrix can be updated in time $O(|V|*|D|)$ in the worst case.

- **Tabu list**: Recall that a move is characterized by a couple $< V_i, v_i >$, V_i and v_i being respectively a variable and a value for the variable. When a move replacing v_i by v_i' for V_i is carried out, the couple $< V_i, v_i >$ is recorded and the value v_i is forbidden to be re-assigned to V_i for the next k iterations[3]. In order to implement the tabu list, we use a $|V|*|D|$ matrix T where each element $T[i,j]$ corresponds to a possible move $< V_i, v_j >$. Each time a move $< V_i, v_j >$ is realized, $T[i,j]$ is set to the current number of iterations plus the tabu tenure k. In this way, it is very easy to know if a move is tabu or not by simply comparing the current iteration with that recorded in T.

- **Aspiration criterion**: In some cases, a move classified tabu may prevent interesting, non visited configurations from being considered. To overcome this problem, a simple aspiration criterion is used: the tabu status of a move is removed if the move leads to a neighbor configuration which is better than the best configuration ever found so far.

TS Algorithm

```
begin
  Pick randomly an initial configuration s0;
  nb_mv ← 0;
  s ← s0; s* ← s;
  while ( (nb_mv < max) and (f(s) ≠ 0) )
  do
    Pick a best neighbor s' ∈ N(s) characterized by < V_i, v'_j >
    among the non-tabu moves and the moves verifying the aspiration criterion ;
    T[i,j] = nb_mv + k ; /* move < V_i, v_j > becomes tabu, v_j being the lost value */
    s ← s' ;
    Update the δ matrix ;
    if ( f(s) ≤ f(s*) ) then
      s* ← s;
    nb_mv ← nb_mv + 1;
  output(s*);
end
```

Figure 2 : *TS algorithm for constraint solving*

Let us make two remarks about the TS algorithm. First, this algorithm uses the same stop condition as SA. Second, contrary to SA, each TS iteration leads always to a move.

5 Experimentation and Results

In this section, we present experimental results of the SA and TS algorithms on the maximal constraint satisfaction problem. We introduce first the test instances and the protocol used to fix the parameters of the SA and TS algorithms.

[3] Note that classifying $< V_i, v_i >$ tabu will forbid more than one configurations to be visited for the period defined by the move's tabu tenure k.

5.1 Tests

Test instances used in this work correspond to random, binary constraint networks generated according to a standard model [24]. A network class is defined by $< n, d, p_1, p_2 >$ which has n variables, d values per variable, $p_1.n.(n-1)/2$ constraints taken randomly from $n.(n-1)/2$ possible ones (p_1 is called the density), and $p_2.d^2$ forbidden pairs of values taken randomly from d^2 possible ones for each constraint (p_2 is called the tightness). For each given class $< n, d, p_1, p_2 >$, different instances can be generated using different random seeds.

A constraint network may be under-constrained, or over-constrained. A phase transition in solubility occurs in between when the network is critically constrained [2, 10, 24]. Under-constrained networks tend to be easily satisfiable (cost $f = 0$) and consequently not interesting for optimization. Over-constrained networks are usually unsatisfiable (cost $f > 0$). Critically constrained networks may or may not be satisfiable and are usually hard to solve from a satisfaction point of view. These different regions are characterized by a factor called *constrainedness* [12]:

$$\kappa = \frac{n-1}{2} p_1 \log_m \left(\frac{1}{1-p_2}\right)$$

$\kappa = 1$ delimits under- ($\kappa < 1$) and over- ($\kappa > 1$) constrained networks. Networks with $\kappa \approx 1$ corresponds to critically-constrained ones.

For the purpose of this work, only over- ($\kappa > 1$) or critically ($\kappa \approx 1$) constrained networks are interesting. Note however that, from an optimization point of view, little is known about the difficulty in finding optimal solutions in these regions.

5.2 Parameter Tuning

It is well known that SA and TS are sensitive to parameter tuning. To obtain good values for the parameters of SA and TS, the following protocol was followed. For each parameter, an interval of reasonable size is first determined. Then a best value is sought for a limited number of moves (50,000 moves in this study). This value is finally used for the final experiments (200,000 moves).

The main parameter for the TS algorithm is the tabu tenure k. The first step of the parametric study gave the interval of [5..50]. Different values of the interval with a step of 5 were then tested and the best one was finally selected. We also experimented several functions to tune dynamically the tabu tenure. However, we have not found a satisfactory function.

The tuning for SA proved to be more complicated for SA since there are three inter-dependent parameters: the initial temperature t_0, the initial step length l_0 and the control parameter A used in the temperature function and step function. For the tested instances, a preliminary study showed that $A = 1{,}000$ was a stable value whatever the values taken by t_0 and l_0. This value was thus used to tune t_0 and l_0.

To tune t_0 and l_0, two intervals were first identified: [1..3] for t_0 and [1600..2500] for l_0. Then a number of selected combinations (limited to 15 in this study) were tested and the best of them was finally chosen.

5.3 Results

We present in this subsection comparative results of SA and TS on random instances generated according to the above mentioned model[4]. To carry out our experimental studies, both

[4] Our instance generator is available from the first author of the paper.

individual instances of different classes and different instances of a same class were used. These instances belong to 3 groups characterized by the number of varibales in a constraint network: 100, 200 et 300 with 10 to 30 values per variable.

Each algorithm was run 10 times on each instance, each run being given 200,000 moves. The parameters of each algorithm were fixed according to the protocol explained above. Two criteria are used to carry out our experimental studies:

- **Quality** : the quality of the final result, *i.e.* the minimal cost value in terms of unsatisfied constraints, found by an algorithm;

- **Running time**: the CPU time used by an algorithm to carry out a given number of moves (200,000 in this study). Note that using iterations instead of moves as a comparison criterion will be unfair with respect to SA.

Table 1 presents comparative results of SA and TS on 15 instances (generated with random seed = 0) belonging to different classes of networks. (Results on different instances of a same class are given below in Table 3.) These classes are chosen in such a way that their instances are not easily satisfiable ($f = 0$) and the cost value of these instances is not too high ($f < 30$).

Problem	κ	TS				SA					TS-SA		
		k	cost			t_0	l_0	cost					
			min.	ave.	max.			min.	ave.	max.	min.	ave.	max.
100.10.15.25	0.93	30	0(4)	0.6	1(6)	2	2000	0(3)	0.8	2(1)	0	-0.2	-1
100.10.20.25	1.24	25	19(10)	19	19(10)	2.5	2000	19(4)	20	22(1)	0	-1	-3
100.15.10.45	1.64	25	11(4)	11.7	13(1)	2	2000	12(1)	13.3	14(4)	-1	-1.6	-1
100.15.20.30	1.3	20	25(1)	26.4	28(2)	2.5	2000	26(3)	27.7	29(4)	-1	-1.3	-1
100.15.30.20	1.22	20	18(2)	19	20(2)	2	2000	22(2)	23.7	25(3)	-4	-4.7	-5
200.18.20.13	0.96	25	4(3)	5.6	8(1)	2	2200	5(4)	5.9	8(1)	-1	0.3	0
200.20.12.22	0.99	35	8(1)	9.6	11(2)	2	2000	11(2)	12.3	14(1)	-3	-2.7	-3
200.20.18.14	0.9	15	0(1)	1.5	3(1)	2.5	1800	2(5)	2.8	4(3)	-2	-1.3	-1
200.20.26.11	1	25	9(1)	11.6	14(1)	2	2300	11(3)	13	15(1)	-2	-1.4	-1
200.20.20.15	1.08	30	21(1)	23.1	25(1)	2.3	2000	23(2)	26.7	29(1)	-2	-3.6	-4
300.20.10.18	0.99	40	14(1)	17	20(1)	2	2100	16(1)	18.4	21(1)	-2	-1.4	-1
300.25.18.10	0.92	25	2(2)	2.8	3(8)	2	2000	2(4)	3.3	5(3)	0	-0.5	-2
300.28.12.16	0.94	35	9(1)	11	12(3)	2	2000	11(2)	12	14(5)	-2	-1	-2
300.30.16.12	0.9	25	4(1)	5.7	7(2)	2.3	2000	4(1)	6.6	8(3)	0	-1.1	-1
300.30.20.10	0.93	35	11(3)	12	14(1)	2	2000	11(2)	13.3	18(1)	0	-1.3	-4

Table 1 : *Comparative results of SA and TS for 200,000 moves*

Let us notice first that the instances used here are quite large compared with those usually used in the literature. The optimal costs of these instances are unknown since they cannot be solved to optimality by current exact algorithms due to their large size. Nevertheless, we have an evident lower bound which is 0.

For each instance, we indicate the constrainedness κ[5]. For both algorithms, we give the value of each parameter used: tabu tenure k for TS and the initial temperature t_0 and initial step length l_0 for SA. We give then the minimum, average and maximum value of the cost function with in parenthesis the number of times a value is reached (over 10 runs). The last three columns indicate the difference between the cost function of TS and SA, in minimum, average and maximum.

From the data of Table 1, we observe that the minimal cost found by TS is better (smaller) than that of SA for 10 instances out of 15 with a difference of 1 to 4 for costs of 0 to 25. For

[5] These networks are all near the critical value 1 or greater than 1, implying that they are either critically constrained or over-constrained.

4 instances out of 5 where SA finds the same minimal costs as TS does, TS finds them more often. In terms of the value of average costs, TS gives always better results with a difference of -0.2 to -4.7. Finally the maximal cost of TS is always smaller (better) than that of SA except one instance with a difference of 1 to 5.

Figure 3 gives a more global picture of the difference of performance of the two algorithms for an instance of the class *100.15.20.30*. X-axis gives the number of moves from 0 to 200,000 with a step of 20,000 and Y-axis the best cost found by each algorithm at a given number of moves. We observe that for a same number of moves, TS finds always better solutions than SA. We observe also that to reach a given cost value, SA needs higher number of moves than TS. These comments are valid for all the tested instances.

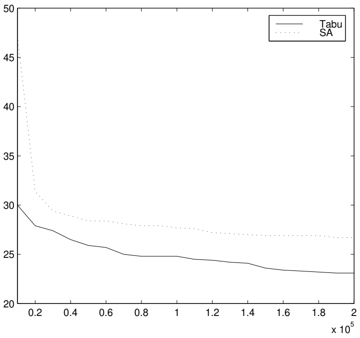

Figure 3 : *Comparisons of cost values found by SA and TS*

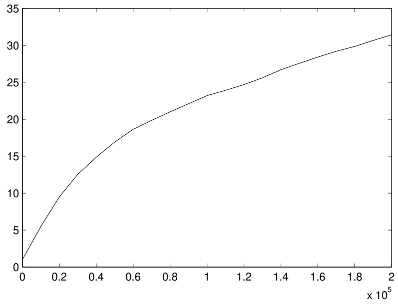

Figure 4 : *Evolution of the ratio "iterations/moves" for SA during the search*

Table 2 presents the CPU times (averaged over 10 runs) required by RS and TS to carry out 200,000 moves. Times are indicated in seconds and obtained with a SPARC 5 station (32Mb RAM and 75MHz)[6].

Problem	Mean running time		SA/TS
	TS	SA	
100.10.15.25	208	263	1.3
100.10.20.25	361	691	1.9
100.15.10.45	585	1546	2.6
100.15.20.30	844	2440	2.9
100.15.30.20	1020	3952	3.9
200.18.20.13	1668	2039	1.2
200.20.12.22	1717	3256	1.9
200.20.18.14	1697	2182	1.3
200.20.20.15	1904	6538	3.4
200.20.26.11	2084	3163	1.5
300.20.10.18	3013	6571	2.2
300.25.18.10	3522	7689	2.2
300.28.12.16	7339	15981	2.2
300.30.16.12	9074	13224	1.5
300.30.20.10	5682	10735	1.9

Table 2 : *Average running times of SA and TS for carrying out 200,000 moves*

From the table, we observe that the running times of TS are much smaller than those of SA to carry out 200,000 *moves*, even if intuitively a TS iteration is more expensive than a SA iteration. Roughly speaking, TS needs only half the time of SA to realize the same number of moves. The main reason is the following. TS and SA require almost the same effort to carry

[6] Both algorithms are programmed in C++.

out an iteration (c.f. Section 4.2 and 4.3). At the same time, a TS iteration leads always to a move while this is not the case for SA. Indeed, not only a SA iteration doesn't lead to a move, the number of iterations needed for SA to make a move increases rapidly when the search progresses. Figure 4 shows the typical evolution of the ratio iterations/moves of SA during the first 200,000 moves. The figure is obtained for an instance of the class *200.20.20.15*.

We observe that the ratio "iterations/moves" increases sharply following a logarithmic function. For example, before 10,000 moves (x-axis), a move needs at most 5 iterations (y-axis). However, this number reaches 20 at 70,000 moves and 31 at 190,000. In other words, in order for SA to catch up TS for a move, SA must be at least several (> 5) times faster for each iteration. However this seems very difficult since the time complexicity of a TS iteration is already quite low thanks to the move-value matrix δ (Section 4.3).

To finish this subsection, we show in Table 3 comparative results of 6 instances taken randomly from a same class (the class *100.15.10.45*, $\kappa = 1.64$). The last number of each instance indicates the random seed used to generate the given instance. We observe once agian that TS finds better solutions for the same number of moves. Further experiments (data not reported here) confirm also that TS needs fewer moves than SA to produce solutions of the same quality.

Problem	Tabu				SA					Tabu-SA	
	k	cost			t_0	l_0	cost				
		min.	ave.	max.			min.	ave.	max.	min.	ave.
100.15.10.45.0	25	11(4)	11,7	13(1)	2	2000	12(1)	13,3	14(4)	-1	-1,6
100.15.10.45.1	25	10(2)	11.3	13(1)	2	2000	12(1)	13.3	15(1)	-2	-2
100.15.10.45.2	25	10(2)	11.2	12(4)	2	2000	11(1)	12.9	15(1)	-1	-1.7
100.15.10.45.3	25	8(1)	9.9	11(2)	2	2000	10(4)	11.2	13(1)	-2	-1.3
100.15.10.45.4	25	8(1)	10	11(3)	2	2000	8(1)	10	14(1)	0	0
100.15.10.45.5	25	9(4)	9.8	11(2)	2	2000	9(3)	10.4	13(1)	0	-1.6

Table 3 : *Comparative results of SA and TS for different instances of a same class*

5.4 Discussions

Influences of parameters

It is well known that the parameters play a important role for SA and TS. This point is confirmed in this study. Figures 5, 6 and 7 show the influences of parameters on the quality of solutions for the SA and TS algorithms.

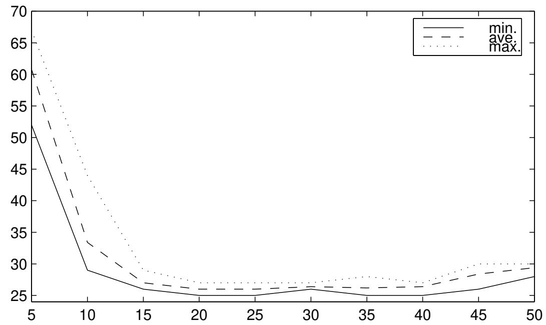

Figure 5 : *Influence of tabu tenure*

Figure 5 shows the variations of the results (minimum, average, maximum) of the cost function (y-axis) for an instance *100.15.20.30* when the tabu tenure varies (x-axis). We observe that a size too small (<15) or too big (>45) leads to bad results. In the contrary, values between 15 and 45 are more interesting. In particular, the value 20 gives the best cost 25 for this instance.

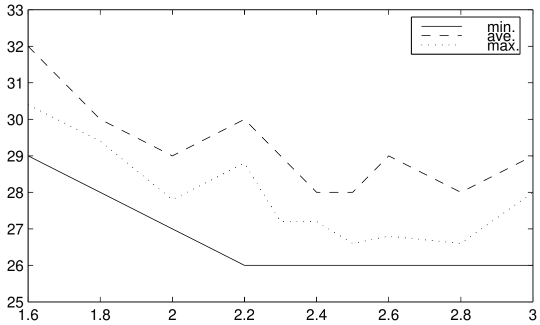

Figure 6 : *Influence of the initial temperature of SA on the cost function*

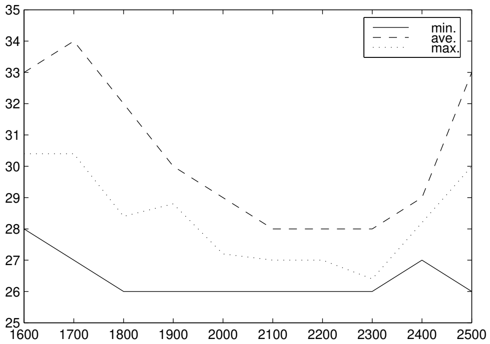

Figure 7 : *Influence of the initial step length of SA on the cost function*

In a similar way, Figures 6 and 7 give the same results when the initial temperature and step length of the SA algorithm are changed. Like for TS, we observe always an interesting interval whose values give good results. At the same time, we observe that there are more than one values giving the same minimum (26). This implies that SA is less sensitive to its parameters to obtain its results.

This remark seems valid across different instances of different classes (see Table 1). Indded, from the data of Table 1, we observe also that the tabu tenure k varies notably according to instances, ranging from 15 to 40. the averaged tabu tenure seems to increase with the size of instances (24 for the instances of 100 variables, 28.75 for the 200 variables and 32 for the 300 variables). A possible explanation may be the following. Tabu list is designed to avoid cycles. However, it is possible that larger instances have larger cycles. In order to avoid such cycles, the tabu tenure should be increased. Concerning the parameters of SA, they seem more stable across all the instances tested.

Influences of neighborhoods

The neighborhood is another critical element for the performance of any local search algorithm. In this study, the neighborhood (noted N_2) is defined over a subset of particular variables, that is, the set of variables involved in some unsatisfied constraints. Another natural neighborhood (noted N_1) consists in including all the variables of the instance, that is, s and s' are neighbors if s' can be obtained by changing the current value of a single (whatever) variable in s. It is clear that N_2 is a subset of N_1 and contains in general much fewer neighbors than N_1.

Experiments have been carried out to compare the performance of these two neighborhoods. Figures 8 and 9 show the differences in terms of solution quality for both the SA and TS algorithms. For example, with the same test condition, the neighborhood N_2 allows SA to get a minimal cost of 14 while N_1 leads to a cost of 15. The influence of neighborhood seems even more important for TS since there is a difference of 2 in terms of minimal cots found with N_1 and N_2.

It is interesting to see that the smaller neighborhood N_2 does better than the large one N_1. One intuitive reason is that N_2 allows an algorithm to concentrate on promising moves and to avoid a large number of moves which don't lead to any improvement of the cost function.

More studies may be needed to give a more satisfactory explanation. For example, a study about the topology of local optima of the two neighborhoods may be highly helpful.

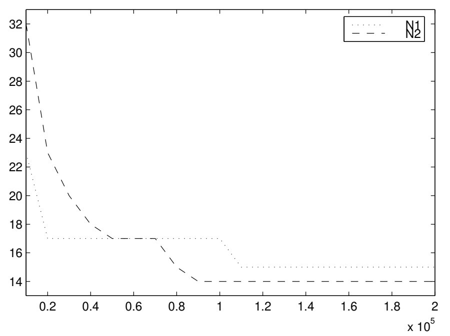

Figure 8 : *Performance comparison of two neighborhoods for SA*

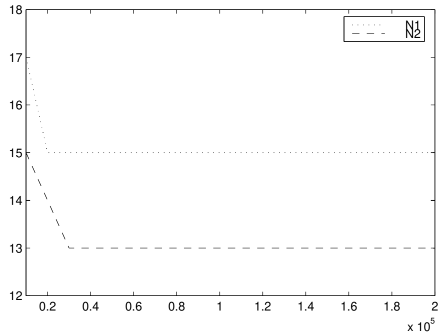

Figure 9 : *Performance comparison of two neighborhoods for TS*

Finally, note that other neighborhoods are equally possible. For example, a dynamic mixing of N_1 and N_2 may be interesting and remains to be investigated.

Configuration evaluation

For both algorithms, the evaluation of configurations is the most time-consuming step. This step is ever crucial for TS since TS looks at many neighboring configurations at each iteration. Efficient data structures are therefore indispensable to achieve a good performance. The δ matrix for move values used by TS is one example. Thanks to this data structure, TS can carry out an iteration with a time complexicity similar to that of SA. Without this structure, the running time of TS will become much more important and its performance may be compromised. Similarly, the partial and local evaluation technique used by SA is also indispensable for its performance.

6 Conclusions

In this study, two local search algorithms based on simulated annealing and tabu search have been investigated for solving constraint problems, in particular, maximal constraint satisfaction problems. Essential components and techniques used in these algorithms are presented in details so that the work can be reproduced. Extensive experiments have been carried out to compare empirically the performance of these algorithms for solving random MCSP instances of large size. Results have showed that:

- TS gives better solutions than SA for a fixed number of moves;
- TS needs fewer moves to get solutions of the same quality as SA;
- TS is faster in terms of running time to carry out a same number of moves.

These results are consistent with previous comparative studies on special classes of constraint problems such as graph coloring [1, 9, 13] and Max-SAT [6].

This study confirms that an appropriate neighborhood and good data structures are indispensable for good performance of an algorithm. It confirms also the importance and difficulty of an appropriate tuning of the parameters.

Finally it remains to know how and to which extent these results can be generalized to other constraint networks (for instance, highly over-constrainted networks, *i.e.* $\kappa >> 1$) and other constraint problems.

Acknowledgment: We would like to thank J.C. Regin and C. Bessière for their assistance for the random generator. We thank also the reviewers for their comments. This study is partially supported by the French National Agency for Innovation and Research (ANVAR).

References

[1] M. Chams, A. Hertz and D. de Werra, "Some experiments with simulated annealing for coloring graphs, European Journal of Operational Research 32, 260-266, 1987.

[2] P. Cheeseman, B. Kanefsky and W.M. Taylor, "Where the really hard problems are", Proc. of the 12th IJCAI'90, pp163-169, 1991.

[3] C. Fleurent and J.A. Ferland, "Genetic and hybrid algorithms for graph coloring", G. Laporte, I. H. Osman, et P. L. Hammer (Eds.), Special Issue of Annals of Operations Research, "Metaheuristics in Combinatorial Optimization", 1996.

[4] E.C. Freuder and R.J. Wallace, "Partial constraint satisfaction", Artificial Intelligence, Vol.58(1-3) pp21-70, 1992.

[5] F. Glover and M. Laguna, "Tabu Search", dans C. R. Reeves (Ed.), Modern heuristics for combinatorial problems, Blackwell Scientific Publishing, Oxford, 1993.

[6] P. Hensen and B. Jaumard, "Algorithms for the maximum satisfiability problem", Computing Vol.44, pp279-303, 1990.

[7] R.M. Haralick and G.L. Elliot, "Increasing tree search efficiency for constraint satisfaction problems", Artificial Intelligence, Vol. 14, pp263-313, 1980.

[8] J.K. Hao and R. Dorne, "Empirical studies of heuristic local search for constraint solving", Proc. of CP-96, LNCS 1118, pp194-208, Cambridge, MA, USA, 1996.

[9] A. Hertz and D. de Werra, "Using Tabu search techniques for graph coloring". Computing Vol.39, pp345-351, 1987.

[10] B.A. Huberman, T. Hogg and C.P. Williams, Artificial Intelligence, Special Issue on Phase Transition and Complexity. Vol. 82(1-2), 1996.

[11] P. Galinier and J.K. Hao, "Tabu search for maximal constraint satisfaction problems", Proc. of CP-97, LNCS 1330, pp196-208, Schloss Hagenberg, Austria, Oct-Nov, 1997.

[12] I.P. Gent, E. MacIntyre, P.Prosser, and T. Walsh, "Scaling effects in the CSP phase transition", Proc. of CP95, pp70-87, 1995.

[13] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon, Optimization by simulated annealing: an experimental evaluation; Part II, graph coloring and number partitioning. Operations Research Vol. 39(3), pp378-406, 1991.

[14] S. Kirkpatrick, C.D. Gelatt Jr. and M.P. Vecchi, "Optimization by simulated annealing", Science No.220, pp671-680, 1983.

[15] G. Laporte and I.H. Osman, "Metaheuristics in combinatorial optimization", Annals of Operations Research, 63, J.C. Baltzer Science Publishers, Basel, Switzerland, 1996.

[16] J. Larrosa and P. Meseguer, "Optimization-based heuristics for maximal constraint satisfaction", Proc. of CP-95, pp190-194, Cassis, France, 1995.

[17] A.K. Mackworth, "Constraint satisfaction", in S.C. Shapiro (Ed.) Encyclopedia on Artificial Intelligence, John Wiley & Sons, NY, 1987.

[18] S. Minton, M.D. Johnston and P. Laird, "Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems", Artificial Intelligence, Vol.58(1-3), pp161-206, 1992.

[19] P. Morris, "The Breakout method for escaping from local minima", Proc. of AAAI-93, pp40-45, 1993.

[20] N.B. Nadel, "Constraint satisfaction algorithms", Computional Intelligence, Vol 5, pp188-224, 1989.

[21] C.H. Papadimitriou and K. Steiglitz, "Combinatorial optimization - algorithms and complexity", Prentice Hall, 1982.

[22] B. Selman, H.J. Levesque and M. Mitchell, "A new method for solving hard satisfiability problems", Proc. of AAAI-92, pp.440-446, San Jose, CA, 1992.

[23] B. Selman and H.Kautz, "Domain-independent extensions to GSAT: solving large structured satisfiability problems", Proc. of IJCAI-93, Chambery, France, 1993.

[24] B.M. Smith, "Phase transition and the mushy region in constraint satisfaction problems", Proc. of ECAI-94, pp100-104, 1994.

[25] E. Tsang, "Foundations of constraint satisfaction", Academic Press, 1993.

[26] R.J. Wallace, "Enhancements of branch and bound methods for the maximal constraint satisfaction problem", Proc. of AAAI-96, pp188-196, Portland, Oregon, USA, 1996.

[27] N. Yugami, Y. Ohta and H. Hara, "Improving repair-based constraint satisfaction methods by value propagation", Proc. of AAAI-94, pp344-349, Seattle, WA, 1994.