Yet Another Local Search Method for Constraint Solving

Philippe Codognet[1] and Daniel Diaz[2]

[1] University of Paris 6, LIP6, 8 rue du Capitaine Scott, 75 015 Paris, FRANCE
`Philippe.Codognetlip6.fr`
[2] University of Paris I, CRI, Bureau C1407, 75 634 Paris Cedex 13, FRANCE
`Daniel.Diaz@inria.fr`

Abstract. We here propose a generic, domain-independent local search method called adaptive search for solving Constraint Satisfaction Problems (CSP). We design a new heuristics that takes advantage of the structure of the problem in terms of constraints and variables and can guide the search more precisely than a global cost function to optimize (such as for instance the number of violated constraints). We also use an adaptive memory in the spirit of Tabu Search in order to prevent stagnation in local minima and loops. This method is generic, can apply to a large class of constraints (e.g. linear and non-linear arithmetic constraints, symbolic constraints, etc) and naturally copes with over-constrained problems. Preliminary results on some classical CSP problems show very encouraging performances.

1 Introduction

Heuristic (i.e. non-complete) methods have been used in Combinatorial Optimization for finding optimal or near-optimal solutions since a few decades, originating with the pioneering work of Lin on the Traveling Salesman Problem [10]. In the last few years, the interest for the family of Local Search methods for solving large combinatorial problems has been revived, and they have attracted much attention from both the Operations Research and the Artificial Intelligence communities, see for instance the collected papers in [1] and [20], the textbook [12] for a general introduction, or (for the French speaking reader) [8] for a good survey. Although local search techniques have been associated with basic hill-climbing or greedy algorithms, this term now encompasses a larger class of more complex methods, the most well-known instances being simulated annealing, Tabu search and genetic algorithms, usually referred as "meta-heuristics". They work by iterative improvement over an initial state and are thus anytime algorithms well-suited to reactive environments. Consider an optimization problem with cost function which makes it possible to evaluate the quality of a given configuration (assignment of variables to current values) and a transition function that defines for each configuration a set of "neighbors". The basic algorithm consists in starting from a random configuration, explore the neighborhood, select an adequate neighbor and then move to the best candidate. This process

will continue until some satisfactory solution is found. To avoid being trapped in local optima, adequate mechanisms should be introduced, such as the adaptive memory of Tabu search, the cooling schedule of simulated annealing or similar stochastic mechanisms. Very good results have been achieved by dedicated and finely tuned local search methods for many problems such as the Traveling Salesman Problem, scheduling, vehicle routing, cutting stock, etc. Indeed, such techniques are now the most promising approaches for dealing with very large search spaces, when the problem is too big to be solved by complete methods such as constraint solving techniques.

In the last years, the application of local search techniques for constraint solving started to raise some interest in the CSP community. Localizer [13,14] proposed a general language to state different kinds of local search heuristics and applied it to both OR and CSP problems, and [18] integrated a constraint solving component into a local search method for using constraint propagation in order to reduce the size of the neighborhoods. GENET [4] was based on the Min-Conflict [16] heuristics, while [17] proposed a Tabu-based local search method as a general problem solver but this approach required a binary encoding of constraints and was limited to linear inequalities. Very recently, [7] developed another Tabu-based local search method for constraint solving. This method, developed independently of our adaptive search approach, also used so-called "penalties" on constraints that are similar to the notion of "constraint errors" that will be described later. It is worth noticing that the first use of such a concept is to be found in [2].

We propose a new heuristic method called Adaptive Search for solving Constraint Satisfaction Problem. Our method can be seen as belonging to the GSAT [22], Walksat [23] and Wsat(OIP) [26] family of local search methods. But the key idea of our approach is to take into account the structure of the problem given by the CSP description, and to use in particular variable-based information to design general meta-heuristics. This makes it possible to naturally cope with heterogeneous problem descriptions, closer to real-life application that pure regular academic problems.

Preliminary results on classical CSP benchmarks such as the simple "N-queens" problem or the much harder "magic square" or "all-intervals" problems show that the adaptive search method performs very well w.r.t. traditional constraint solving systems.

2   Adaptive Search

The input of the method is a problem in CSP format, that is, a set of variables with their (finite) domains of possible values and a set of constraints over these variables. A constraint is simply a logical relation between several unknowns, these unknowns being variables that should take values in some specific domain of interest. A constraint thus restricts the degrees of freedom (possible values) the unknowns can take; it represents some partial information relating the objects of interest. Constraint Solving and Programming has proved to be very

successful for Problem Solving and Combinatorial Optimization applications, by combining the declarativity of a high-level language with the efficiency of specialized algorithms for constraint solving, borrowing sometimes techniques from Operations Research and Numerical Analysis [21]. Several efficient constraint solving systems for finite domain constraints now exists, such as ILOG Solver [19] on the commercial side or clp(FD)[3] and GNU-Prolog [5] on the academic/freeware side. Although we will completely depart in adaptive search from the classical constraint solving techniques (i.e. Arc-Consistency and its extensions), we will take advantage of the formulation of a problem as a CSP. Such representation indeed makes it possible to structure the problem in terms of variables and constraints and to analyze the current configuration (assignment of variables to values in their domains) more carefully than a global cost function to be optimized, e.g. the number of constraints that are not satisfied. Accurate information can be collected by inspecting constraints (that typically involve only a subset of all the problem variables) and combining this information on variables (that typically appear in only a subset of all the problem constraints).

Our method is not limited to any specific type of constraint, e.g. linear constraints as classical linear programming or [26]. However we need, for each constraint, an *error function* that will give an indication on how much the constraint is violated. Consider a n-ary constraint $C(X_1, \ldots, X_n)$ and domains $D_1, \ldots, D_n$ for variables $\{X_1, \ldots, X_n\}$. An error function f_c associated to the constraint C is simply a real-valued function from $D_1 \times \ldots \times D_n$ such that $f_C(X_1, \ldots, X_n)$ has value zero if $C(X_1, \ldots, X_n)$ is satisfied. We do not impose the value of f_C to be different from zero when the constraint is not satisfied, and this will indeed not be the case in some of the examples we will describe below. The error function will in fact be used as a heuristic value to represent the "degree of satisfaction" of a constraint and thus to check how much a constraint is violated by a given tuple. For instance the error function associated to an arithmetic constraint $|X - Y| \leq C$ will be $max(0, |X - Y| - C)$. Adaptive search relies on iterative repair, based on variables and constraint errors information, seeking to reduce the error on the worse variable so far. The basic idea is to compute the error function of each constraint, then combine for each variable the errors of all constraints in which it appears, therefore projecting constraint errors on involved variables. Finally, the variable with the maximal error will be chosen as a "culprit" and thus its value will be modified. In this second step we use the well-known min-conflict heuristics [16] and select the value in the variable domain that has the best error immediate value, that is, the value for which the total error in the next configuration is minimal. This is similar to the steepest ascent heuristics for traditionnal hillclimbing.

In order to prevent being trapped in local minima, the adaptive search method also includes an adaptive memory as in Tabu Search : each variable leading to a local minimum is marked and cannot be chosen for the few next iterations. A local minimum is a configuration for which none of the neighbor improve the current configuration. This corresponds in adaptive search to a variable whose current value is better than all alternative values in its domain. It is worth

noticing that conversely to most Tabu-based methods (e.g. [6] or [7] for a CSP-oriented framework) we mark variables and not couples $< variable, value >$, and that we do not systematically mark variables when chosen in the current iteration but only when they lead to a local minimum. Observe however that, as we use the min-conflict heuristics, the method will never choose the same variable twice in a row.

It is worth noticing that the adaptive search method is thus a generic framework parametrized by three components :

- A family of error functions for constraints (one for each type of constraint)
- An operation to combine for a variable the errors of all constraints in which it appears
- A cost function for a evaluating configurations

In general the last component can be derived from the first two one. Also, we could require the combination operation to be associative and commutative.

3 General Algorithm

Let us first detail the basic loop of the adaptive search algorithm, and then present some extra control parameters to tune the search process.

Input :
　　Problem given in CSP form :
　　　- a set of variables $V = \{V_1, V_2, \ldots, V_n\}$ with associated domains of values
　　　- a set of constraints $C = \{C_1, C_2, \ldots, C_k\}$ with associated error functions
　　　- a combination function to project constraint errors on variables
　　　- a (positive) cost function to minimize
Output :
　　a sequence of moves (modification of the value of one of the variables) that will lead to a solution of the CSP (configuration where all constraints are satisfied) if the CSP is satisfied or to a partial solution of minimal cost otherwise.
Algorithm :
　　Start from a random assignment of variables in V
　　Repeat
　　　　1. Compute errors of all constraints in C and combine errors on each variable by considering for a given variable only the constraints on which it appears.
　　　　2. select the variable X (not marked as Tabu) with highest error and evaluate costs of possible moves from X
　　　　3. if no improving move exists
　　　　　　then mark X tabu for a given number of iterations
　　　　　　else select the best move (min-conflict) and change the value of X accordingly
　　until a solution is found or a maximal number of iterations is reached

Some extra parameters can be introduced in the above framework in order to control the search, in particular the handling of (partial) restarts. One first has to precise, for a given problem, the *Tabu tenure* of each variable, that is, the number of iteration a variable should not be modified once it is marked due to local minima. Thus, in order to avoid being trapped with a large number of Tabu variables and therefore no possible diversification, we decide to randomly reset a certain amount of variables when a given number of variables are Tabu at the same time. We thereafter introduce two other parameters : the *reset limit*, i.e. the number of simultaneous Tabu variables to reach in order to randomly reset a certain ratio of variables (*reset percentage*). Finally, as in all local search methods, we parametrize the algorithm with a maximal number of iterations (*max iterations*). This could be used to perform early restart, as advocated by [23]. Such a loop will be executed at most *max restart* times before the algorithm stops.

This method, although very simple, is nevertheless very efficient to solve complex combinatorial problems such as classical CSPs, as we will see in the next section. It is also worth noticing that this method has several sources of stochasticity. First, in the core algorithm, both in the selection of the variable and in the selection of the value for breaking ties between equivalent choices (e.g. choosing between two variables that have the same value for the combination of their respective constraint errors). Second, in the extra control parameters that have just been introduced, to be tuned by the user for each application. For instance if the reset limit (number of simultaneous tabu variables) is very low, the algorithm with restart very often, enhancing thus the stochastic aspects of the method; but on the other hand if the reset limit is too high, the method might show some trashing behavior and have difficulties in escaping local minima. Last but not least, when performing a restart, the algorithm will randomly modify the values of a given percentage of randomly chosen variables (the reset percentage). Thus a reset percentage of 100 % will amount to restart each time from scratch.

4   Examples

Let us now detail how the adaptive search method performs on some classical CSP examples. We have tried to choose benchmarks on which other constraint solving methods have been applied in order to obtain comparison data, but is it worth noticing that all these benchmarks have satisfiable instances. For each benchmark we give a brief description of the problem and its modeling in the adaptive search approach. Then, we present performance data averaged on 10 executions, including:

- − instance number (i.e. problem size)
- − average, best and worst CPU time
- − total number of iterations (within a single run, on average)
- − number of local minima reached (within a single run, on average)
- − number of performed swaps (within a single run, on average)

– number of resets (within a single run, on average)

Then we compare those performance results (essentially the execution time) with other methods among the most well-known constraint solving techniques: constraint programming systems [5,19], general local search system [13,14], Ant-Colony Optimization [24]. We have thus favored academic benchmarks over randomly generated problems in order to compare to literature data.

Obviously, this comparison is preliminary and not complete but it should give the reader a rough idea of the potential of the adaptive search approach. We intend to make a more exhaustive comparison in the near future.

4.1 Magic to the square

The magic square puzzle consists in placing on a NxN square all the numbers in $\{1, 2, \ldots, N^2\}$ such as the sum of the numbers in all rows, columns and the two diagonal are the same. It can therefore be modeled in CSP by considering N^2 variables with initial domains $\{1, 2, \ldots, N^2\}$ together with linear equation constraints and a global all_different constraint stating that all variables should have a different value. The constant value that should be the sum of all rows, columns and the two diagonals can be easily computed to be $N(N^2 + 1)/2$.

The instance of adaptive search for this problem is defined as follows. The error function of an equation $X_1 + X_2 + \ldots + X_k = b$ is defined as the value of $X_1 + X_2 + \ldots + X_k - b$. The combination operation is the absolute value of the sum of errors (and not the sum of the absolute values, which would be less informative : errors with the same sign should add up as they lead to compatible modifications of the variable, but not errors of opposite signs). The overall cost function is the addition of absolute values of the errors of all constraints The method will start by a random assignment of all N^2 numbers in $\{1, 2, \ldots, N^2\}$ on the cells of the NxN square and consider as possible moves all swaps between two values.

The method can be best described by the following example which depicts information computed on a 4x4 square:

Values and Constraint errors						Projections on variables					Costs of next configurations			
			-8											
11	7	8	15	7		8	2	1	8		39	54	51	33
16	2	4	12	0		4	8	16	9		53	67	61	41
10	6	5	3	-10		6	23	21	1		45	57	57	66
1	14	9	13	3		1	2	5	9		77	43	48	41
4	-5	-8	9	-3										

The table on the left shows the configuration of the magic square at some iteration (each variable corresponds to a cell of the magic square). Numbers on the right of rows and diagonals, and below lines, denote the errors of the corresponding constraints. The total cost is then computed as the sum of the absolutes

values of those constraints errors and is equal to 57. The table immediately on the right shows the combination of constraint errors on each variable. The cell (3,2) with value **6** (in bold font on the left table) has maximal error (**23**) and is thus selected for swapping. We should now score all possible swaps with other numbers in the square; this is depicted in the table on the right, containing the cost value of the overall configuration for each swap. The cell (1,4) with value *15* (in italic) gives the best next configuration (with cost *33*) and is thus selected to perform a move. The cost of the next configuration will therefore be 33.

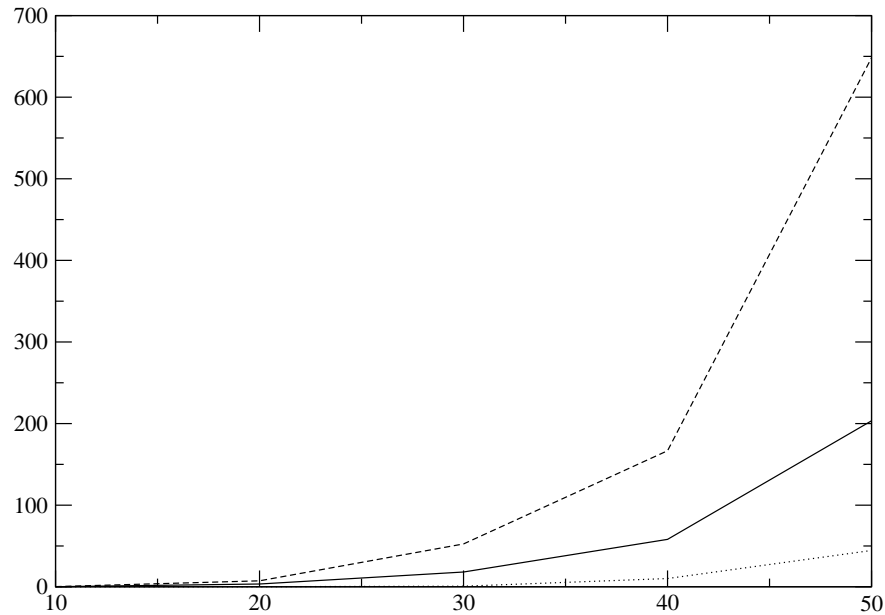| problem instance	time (sec) avg of 10	time (sec) best	time (sec) worst	nb iterations	nb local minima	nb swaps	nb resets
10x10	0.13	0.03	0.21	6219	2354	3864	1218
20x20	3.41	0.10	7.35	47357	21160	26196	11328
30x30	18.09	0.67	52.51	116917	54919	61998	29601
40x40	58.07	10.13	166.74	216477	102642	113835	56032
50x50	203.42	44.51	648.25	487749	233040	254708	128625

Table 1. magic square results

Fig. 1. magic square graph

Table 1 details the performances of this algorithm on bigger instances. For a problem of size NxN the following settings are used: Tabu tenure is equal to N-1 and 10 % of the variables are reset when $N^2/6$ variables are Tabu. The programs were executed on a PentiumIII 733 MHz with 192 Mb of memory running Linux RedHat 7.0.

Figure 1 depicts how CPU times evolve w.r.t. problem size: the dotted line represents the best execution time, the dashed line the worst one and the solid line the average one.

size	Localizer	Adaptive
16x16	50.82	1.14
20x20	313.2	3.4
30x30	1969	18.09
40x40	8553	58.07
50x50	23158	203.4

Table 2. Comparison with Localizer

Constraint programming systems such as GNU-Prolog or ILOG Solver perform poorly on this benchmark and cannot solve instances greater than 10x10. We can nevertheless compare with another local search method: this benchmark has been attacked by the Localizer system with a Tabu-like meta-heuristics. Localizer [13,14,15] is a general framework and language for expressing local search strategies which are then compiled into C++ code. Table 2 compares the CPU times for both systems (in seconds). Timings for Localizer come from [15] and have been measured on a PentiumIII-800 and thus on a machine similar to ours (PentiumIII-733), but it is worth noticing however that the method used in Localizer consists in exploring at each iteration step the whole single-value exchange neighborhood (of size n^2). Our results compare very favorably with those obtained with the Localizer system, as the adaptive search is two orders of magnitude faster. Moreover its performances could certainly be improved by careful tuning of various parameters (global cost function, Tabu tenure, reset level and percentage of reset variables, ...) in order to make the method truly adaptive indeed...

4.2 God saves the queens

This puzzle consists in placing N queens on a NxN chessboard so that no two queens attach each other. It can be modeled by N variables (that is, one for each queen) with domains $\{1, 2, \ldots, N\}$ (that is, considering that each queen should be placed on a different row) and $3 \times N(N-1)/2$ disequation constraints stating that no pair of queens can ever be on the same column, up-diagonal or down-diagonal :

$$\forall (i,j) \in \{1, 2, \ldots, N\}^2, \ s.t. \ i \neq j : Q_i \neq Q_j, \ Q_i + i \neq Q_j + j, \ Q_i - i \neq Q_j - j$$

Observe that this problem can also be encoded with three all_different global constraints.

We can define the error function for disequation as follows, in the most simple way : 0 if the constraint is satisfied and 1 if the constraint is violated. The combination operation on variables is simply the addition, and the overall cost function is the sum of the costs of all constraints.

problem instance	time (sec) avg of 10	time (sec) best	time (sec) worst	nb iterations	nb local minima	nb swaps	nb resets
100	0.00	0.00	0.01	30	0	29	0
200	0.01	0.00	0.01	50	0	50	0
500	0.04	0.04	0.05	114	0	114	0
1000	0.14	0.13	0.15	211	0	211	0
2000	0.53	0.50	0.54	402	0	402	0
3000	1.16	1.13	1.19	592	0	592	0
4000	2.05	2.01	2.08	785	0	785	0
5000	3.24	3.19	3.28	968	0	968	0
7000	6.81	6.74	6.98	1356	0	1356	0
10000	13.96	13.81	14.38	1913	0	1913	0
20000	82.47	81.40	83.81	3796	0	3796	0
30000	220.08	218.18	221.14	5670	0	5670	0
40000	441.93	439.54	444.84	7571	0	7571	0
100000	3369.89	3356.75	3395.45	18846	0	18846	0

Table 3. N-Queens results

Table 3 details the performances of this algorithm on large instances. For a problem of size NxN the following settings are used: Tabu tenure is equal to 2 and 10 % of the variables are reset when $N/5$ variables are Tabu. The programs were executed on a PentiumIII 733 MHz with 192 Mb of memory running Linux RedHat 7.0.

Figure 2 depicts how CPU times evolve w.r.t. problem size: the dotted line represents the best execution time, the dashed line the worst one and the solid line the average one.

Surprisingly the behavior of the adaptive search is almost linear and the variance between different executions is quasi inexistent. Let us now compare with a constraint programming system (ILOG solver) and an ant colony optimization method (Ant-P solver), both timings (in seconds) are taken from [24] and divided by a factor 7 corresponding to the SPECint 95 ratio between the processors. Timings for Localizer come again from [15] and have been measured on a PentiumIII-800 and thus on a machine slighty more performant than ours. Table 4 clearly show that adaptive search is much more performant on this benchmark, which might not be very representative of real-life applications but is a not-to-be-missed CSP favorite...
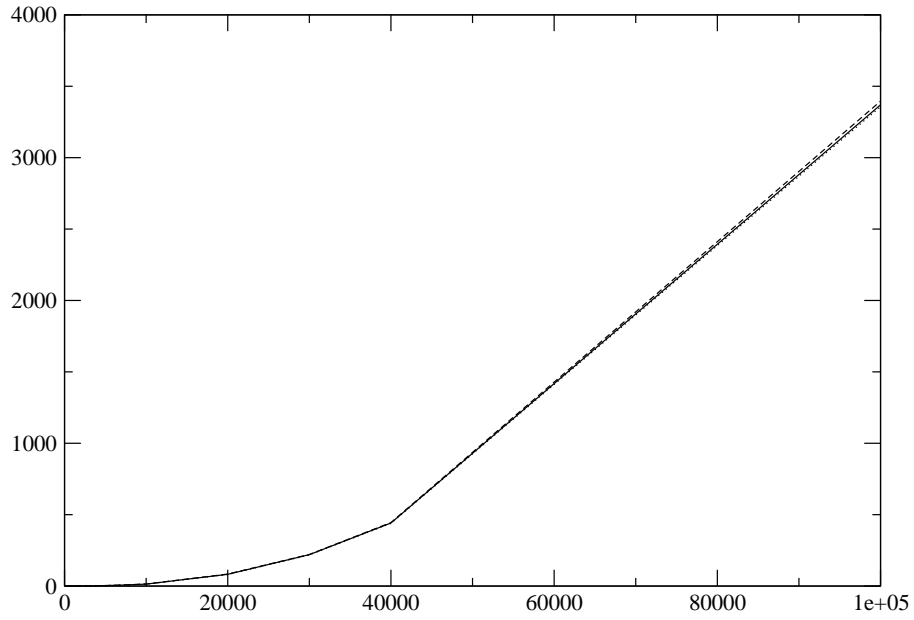
Fig. 2. N-Queens graph

size	ILOG	Ant-P	Localizer	Adaptive
50	0.09	0.39		0.00
100	0.07	2.58		0.00
150	79.3	20.6		0.00
200	36.6	40.37		0.01
256	*	*	0.16	0.01
512	*	*	0.43	0.04
1024	*	*	1.39	0.15
2048	*	*	5.04	0.54
4096	*	*	18.58	2.14
8192	*	*	68.58	9.01
16384	*	*	260.8	46.29
32768	*	*	1001	270.2
65536	*	*	10096	1320

Table 4. Comparison with ILOG Solver, Ant-P and Localizer

4.3 All-Intervals Series

Although looking like a pure combinatorial search problem, this benchmark is in fact a well-known exercise in music composition [25]. The idea is to compose a sequence of N notes such that all notes are different and tonal intervals between consecutive notes are also distinct. This problem can be modeled as a permutation of the N first integers such that the absolute difference between two consecutive pairs of numbers are all different.

This problem is modeled by considering N variables $\{V_1, \ldots, V_N\}$ that represent the notes, whose values will represent a permutation of $\{0, \ldots, N-1\}$. There is only one constraint to encode stating that absolute values between each pair of consecutive variables are all different. Possible moves from one configuration consist in all possible swaps between the values of two variables. As all variables appear symmetrically in this constraint there is no need to project errors on each variable (all variable errors would be equal) and we just have to compute the total cost for each configuration. One way to do this is to first compute the distance between 2 consecutive variables:

$$D_i = |V_{i+1} - V_i| \text{ for } i \in [1, n-1]$$

Then one has to define the number of occurrence of each distance value:

$$Occ_j = \Sigma_{i=1}^{N-1}(\textbf{if } D_i = j \textbf{ then } 1 \textbf{ else } 0)$$

Obviously, the all_different constraint on the distance values is satisfied iff for all $j \in [1, n-1]$, $Occ_j = 1$. It is thus interesting to focus on the values j such that $Occ_j = 0$ representing the "missing values" for the distances. We will moreover consider that it is harder to place bigger distances and thus introduce a bias in the total cost as follows:

$$cost = \Sigma_{j=1}^{n-1}(\textbf{if } Occ_j = 0 \textbf{ then } j \textbf{ else } 0)$$

Obviously a solution is found when $cost = 0$.

Table 5 details the performances of this algorithm on several instances. For a problem of size N the following settings are used: Tabu tenure is equal to $N/10$ and 10 % of the variables are reset when 1 variable is Tabu. The programs were executed on a PentiumIII 733 MHz with 192 Mb of memory running Linux RedHat 7.0.

Figure 3 depicts how CPU times evolves w.r.t. problem size: the dotted line represents the best execution time, the dashed line the worst one and the solid line the average one.

Let us now compare with a constraint programming system (ILOG solver) and an ant colony optimization method (Ant-P solver), both timings are taken from [24] and divided by a factor 7 corresponding to the SPECint 95 ratio between the processors. ILOG Solver might take advantage of global constraints to model this problem, but nevertheless perform poorly and can only find (without any backtracking) the trivial solution :

$$< 0, N-1, 1, N-2, 2, N-3, \ldots >$$

problem instance	time (sec) avg of 10	time (sec) best	time (sec) worst	nb iterations	nb local minima	nb swaps	nb resets
10	0.00	0.00	0.00	14	6	8	3
12	0.00	0.00	0.01	46	20	25	20
14	0.00	0.00	0.01	85	38	46	38
16	0.01	0.00	0.02	191	88	103	88
18	0.04	0.01	0.08	684	316	367	316
20	0.06	0.00	0.17	721	318	403	320
22	0.16	0.04	0.36	1519	527	992	791
24	0.70	0.10	2.42	5278	1816	3461	2724
26	3.52	0.36	9.26	21530	7335	14194	11003
28	10.61	1.38	25.00	53065	18004	35061	27006
30	63.52	9.49	174.79	268041	89518	178523	134308

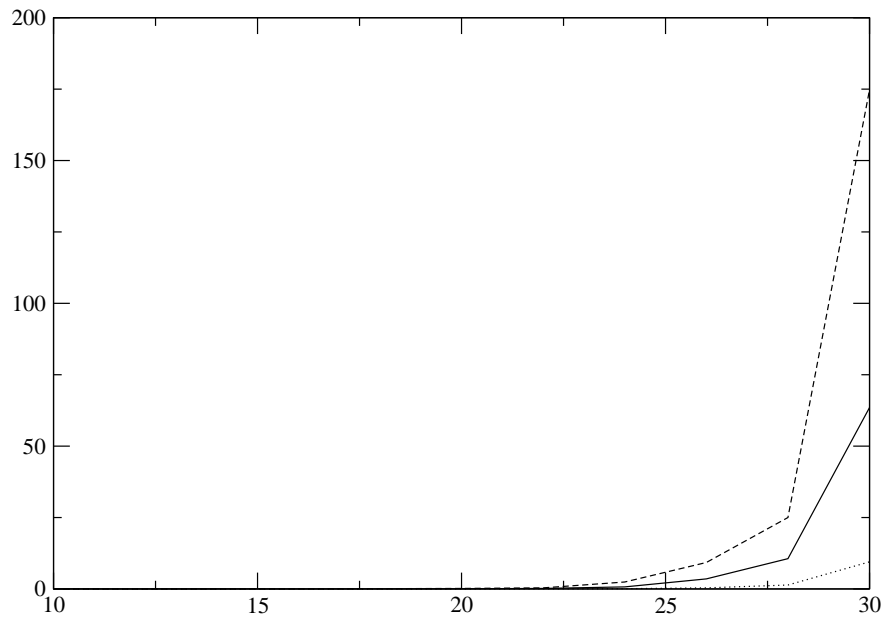**Table 5.** All-intervals series result

Fig. 3. All-intervals series graph

For instances greater than 16, no other solution can be found in reasonable time: [24] reported that the execution times where greater than a full hour of CPU time (this is depicted by a * symbol in our table).

size	ILOG	Ant-P	Adaptive
14	4.18	0.07	0.00
16	126.05	0.28	0.01
18	*	0.52	0.04
20	*	1.48	0.06
22	*	2.94	0.16
24	*	9.28	0.70

Table 6. Comparison with ILOG Solver and Ant-P

Adaptive search is therefore more than an order of magnitude faster than Ant-Colony Optimization on this problem (see table 6, where timings are given in seconds).

4.4 Number Partitioning

This problem consists in finding a partition of numbers $\{1, \ldots, N\}$ into two groups A and B such that:

- A and B have the same cardinality
- sum of numbers in A = sum of numbers in B
- sum of squares of numbers in A = sum of squares of numbers in B

This problem admits a solution iff N is a multiple of 8 and is modeled as follows. Each configuration consists in the partition of the values $V_i \in \{1, \ldots, N\}$ in two subsets of equal size. There are two constraints :

$$\Sigma_{i=1}^{n} V_i \;=\; N(N+1)/2$$

$$\Sigma_{i=1}^{n} V_i^2 \;=\; N(N+1)(2N+1)/6$$

The possible moves from one configuration consist in all possible swaps exchanging one value in the first subset with another one in the second subset. The errors on the equality constraints are computed as previously in the magic square problem. In this problem again, as in the previous all-intervals example, all variables play the same role and there is no need to project errors on variables. The total cost of a configuration can be obtained as the sum of the absolute values of all constraint errors. Obviously again, a solution is found when the total cost is equal to zero.

Table 7 details the performances of this algorithm on several instances. For a problem of size N the following settings are used: Tabu tenure is equal to 2

problem instance	time (sec) avg of 10	time (sec) best	time (sec) worst	nb iterations	nb local minima	nb swaps	nb resets
80	0.01	0.00	0.02	169	108	61	123
120	0.02	0.01	0.04	194	118	76	177
200	0.11	0.06	0.19	383	216	166	433
512	1.13	0.07	3.26	721	348	372	1918
600	1.86	0.02	8.72	870	414	456	2484
720	4.46	0.54	21.12	1464	680	784	5101
800	6.41	0.26	12.55	1717	798	919	6385
1000	8.01	2.44	17.25	1400	630	769	6306

Table 7. number partitioning results

and 2 % of the variables are reset when one variable is Tabu. The programs were executed on a PentiumIII 733 MHz with 192 Mb of memory running Linux RedHat 7.0. Figure 4 depicts how CPU times evolve w.r.t. problem size: the dotted line represents the best execution time, the dashed line the worst one and the solid line the average one. Constraint Programming systems such as GNU Prolog cannot solve this problem for instances larger than 128.

4.5 The Alpha cipher

This problem has been posted on the newsgroup rec.puzzles a few years ago, it consists in solving a system of 20 simultaneous equations over the integers as follows. The numbers $\{1, \ldots, 26\}$ have to be assigned to the letters of the alphabet. The numbers beside each word are the total of the values assigned to the letters in the word. e.g for LYRE L,Y,R,E might equal 5,9,20 and 13 respectively or any other combination that add up to 47. The problem consists in finding the value of each letter satisfying the following equations:

$$
\begin{array}{llll}
\text{BALLET} = 45 & \text{GLEE} = 66 & \text{POLKA} = 59 & \text{SONG} = 61 \\
\text{CELLO} = 43 & \text{JAZZ} = 58 & \text{QUARTET} = 50 & \text{SOPRANO} = 82 \\
\text{CONCERT} = 74 & \text{LYRE} = 47 & \text{SAXOPHONE} = 134 & \text{THEME} = 72 \\
\text{FLUTE} = 30 & \text{OBOE} = 53 & \text{SCALE} = 51 & \text{VIOLIN} = 100 \\
\text{FUGUE} = 50 & \text{OPERA} = 65 & \text{SOLO} = 37 & \text{WALTZ} = 34
\end{array}
$$

This is obviously modeled by a set of 20 linear equations on 26 variables. The errors on the linear constraints are computed as previously in the magic square example. The projection on variables is the absolute value of the sum of each constraint error multiplied by the coefficient of the variable in that (linear) constraint. The total cost is, as usual, the sum of the absolute values of constraint errors.

Local search is certainly not the best way to solve such a (linear) problem. Nevertheless it could be interesting to see the performances of adaptive search on such a benchmark in order to observe the versatility of this method. Table 8 details the performances of this algorithm. The following settings are used: Tabu
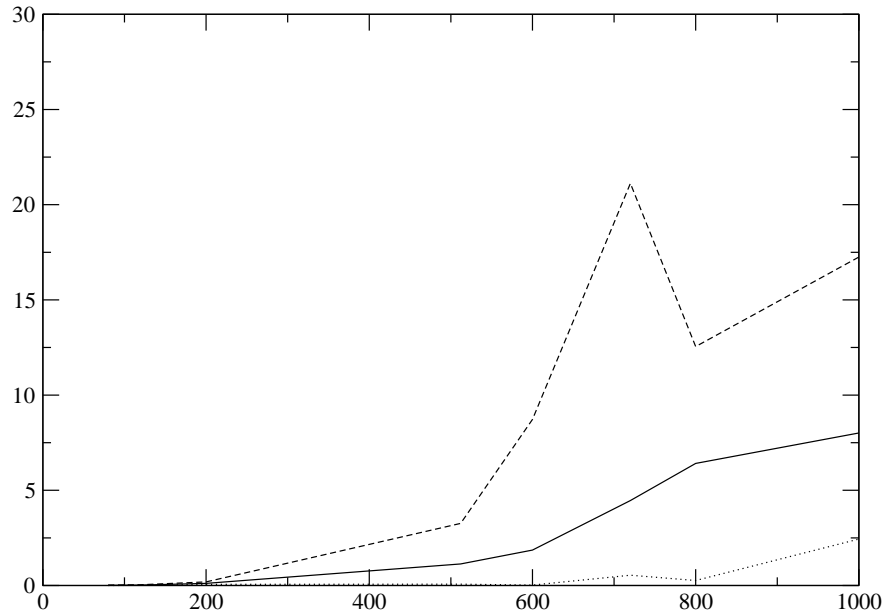
Fig. 4. number partitioning graph

tenure is equal to 1 and 5 % of the variables are reset when 6 variables are Tabu. The programs were executed on a PentiumIII 733 MHz with 192 Mb of memory running Linux RedHat 7.0.

problem instance	time (sec) avg of 10	time (sec) best	time (sec) worst	nb iterations	nb local minima	nb swaps	nb resets
alpha-26	0.08	0.03	0.18	5419	3648	1770	751

Table 8. alpha cipher result

Constraint Programming systems such as GNU Prolog can solve this problem in 0.25 seconds with standard labeling and in 0.01 seconds with the first-fail labeling heuristics. Surprisingly, adaptive search is not so bad on this example, which is clearly out of the scope of its main application domain.

5   Conclusion and Perspectives

We have presented a new heuristic method called adaptive search for solving Constraint Satisfaction Problems by local search. This method is generic, domain-independent, and uses the structure of the problem in terms of constraints and

variables to guide the search. It can apply to a large class of constraints (e.g. linear and non-linear arithmetic constraints, symbolic constraints, etc) and naturally copes with over-constrained problems. Preliminary results on some classical CSP problems show very encouraging results, about one or two orders of magnitude faster than competing methods on large benchmarks. Nevertheless, further testing is obviously needed to assess these results.

It is also worth noticing that the current method does not perform any planning, as it only computes the move for the next time step out of all possible current moves. It only performs a move if it immediately improves the overall cost of the configuration, or it performs a random move to escape a local minimum. A simple extension would be to allow some limited planning capability by considering not only the immediate neighbors (i.e. nodes at distance 1) but all configurations on paths up to some predefined distance (e.g. all nodes within at distance less than or equal to some k), and then choose to move to the neighbor in the direction of the most promising node, in the spirit of variable-depth search [11] or limited discrepancy search [9]. We plan to include such an extension in our model and evaluate its impact. Further work is needed to assess the method, and we plan to develop a more complete performance evaluation, in particular concerning the robustness of the method, and to better investigate the influence of stochastic aspects and parameter tuning of the method. Future work will include the development of dynamic, self-tuning algorithms.

References

1. E. Aarts and J. Lenstra (Eds). *Local Search in Combinatorial Optimization*. Wiley, 1997.
2. A. Borning, B. Freeman-Benson and M. Wilson. Constraint Hierarchies. *Lisp and Symbolic Computation*, vol. 5 no. 3, 1992, pp 223-270.
3. P. Codognet and D. Diaz. Compiling Constraint in `clp(FD)`. *Journal of Logic Programming*, Vol. 27, No. 3, June 1996.
4. A. Davenport, E. Tsang, Z. Kangmin and C. Wang. GENET : a connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *proc. AAAI 94*, AAAI Press, 1994.
5. D. Diaz and P. Codognet. The implementation of GNU Prolog. In *proc. SAC'OO, 15th ACM Symposium on Applied Computing*. Como, Italy, ACM Press 2000.
6. F. Glover and M. Laguna. *Tabu Search*, Kluwer Academic Publishers, 1997.
7. P. Galinier and J-K. Hao. A General Approach for Constraint Solving by Local Search. *draft*, 2001.
8. J-K. Hao, P. Galinier and M. Habib. Metaheuristiques pour l'optimisation combinatoire et l'affectation sous contraintes. *Revue d'Intelligence Artificielle*, vol.2 no. 13, 1999, pp 283-324.
9. W. Harvey and M. Ginsberg. Limited Discrepancy Search. In *proc. IJCAI'95, 14th International Joint Conference on Artificial Intelligence*, Montreal, Canada, 1995.
10. S. Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, vol. 44 (1965), pp 2245-2269.
11. S. Lin and B. Kerninghan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, vol. 21 (1973), pp 498-516.

12. Z. Michalewicz and D. Fogel. *How to solve it: Modern Heuristics*, Springer Verlag 2000.

13. L. Michel and P. Van Hentenryck. Localizer : a modeling language for local search. In *proc. CP'97, 3rd International Conference on Principles and Practice of Constraint Programming*, Linz, Austria, Springer Verlag 1997.

14. L. Michel and P. Van Hentenryck. Localizer. *Constraints*, vol. 5 no. 1&2, 2000.

15. L. Michel and P. Van Hentenryck. Localizer++ : an open library for local search. Research Report, Brown University 2001.

16. S. Minton, M. Johnston, A. Philips and P. Laird. Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, vol. 58, 1992, pp 161-205.

17. K. Nonobe and T. Ibaraki. A tabu search approach to the constraint satisfaction problem as a general problem solver. *European Journal of Operational Research*, vol. 106, 1998, pp 599-623.

18. G. Pesant and M. Gendreau. A Constraint Programming Framework for Local Search Methods. *Journal of Heuristics*, vol. 5 no. 3, 1999, pp 255-279.

19. J-F. Puget. A C++ implementation of CLP. In *proc. SPICIS'94*, Singapore, 1994.

20. V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, G. D. Smith. *Modern Heuristic Search Methods*. Wiley, 1996.

21. V. Saraswat, P. Van Hentenryck et al. Constraint Programming, *ACM Computing Surveys*, vol. 28 no. 4, December 1996.

22. B. Selman, H. Levesque and D. Mitchell. A new method for solving hard satisfiability problems. In *proc. AAAI'92*, AAAI Press 1992.

23. B. Selman, H. Kautz and B. Cohen. Noise strategies for improving local search. In *proc. AAAI'94*, AAAI Press 1994.

24. C. Solnon. Solving permutation problems by ant colony optimization. In *proc. ECAI'2000*, Berlin, Germany, Wiley, 2000.

25. C. Truchet, C. Agon and G. Assayag. Recherche Adaptative et Contraintes Musicales. In *proc. JFPLC2001, Journes Francophones de Programmation Logique et Programmation par Contraintes*, P. Codognet (Ed.), Hermes, 2001.

26. J. P. Walser. *Integer Optimization by Local Search : A Domain-Independent Approach*, LNAI 1637, Springer Verlag 1999.