Parallel Randomized Heuristics For The Set Covering Problem

MARIA STELLA FIORENZO CATALANO
Transportation and Traffic Engineering Section
Delft University of Technology
P.O. Box 5048 - 2600 GA Delft
THE NETHERLANDS
S.Catalano@CiTG.TUDelft.NL

FEDERICO MALUCELLI
DEI - Politecnico di Milano
Piazza L. da Vinci 32 - 20133 Milano
ITALY
malucell@elet.polimi.it http://www.elet.polimi.it/people/malucell

Abstract: - We propose a general scheme to derive heuristics for the Set Covering Problem. The scheme is iterative and embeds constructive heuristics within a randomized procedure. A first group of heuristics is obtained by randomizing the choices made at each step when the solution is constructed in a way similar to that of the so called "Ant System"; a second group of more efficient heuristics is obtained by introducing a random perturbation of the costs of the problem instance. Some computational results are presented. Different parallel implementations are discussed and some performance measures reported.

Keywords: - heuristic algorithms, randomized methods, set covering, parallel algorithms

1. Introduction

The Set Covering Problem plays an important role in combinatorial optimization. Many important applications can be formulated in terms of Set Covering, as for example crew scheduling, vehicle routing, facility location, assembly line balancing, information retrieval [10].
Given a set $I = \{1, \ldots, m\}$ and a collection of proper subsets of I, $F=\{I_1, \ldots, I_n\}$, a *cover* $S \subseteq F$ of I is such that each element of I belongs at least to one of the subsets in S. If we associate a cost c_j with each subset $I_j \in F$, and we define the cost of a cover as the sum of the cost of its components, the Set Covering Problem consists in finding minimum cost cover.
Due to the difficulty of determining the optimal solution (the problem is NP-hard), and due to the great scale of real life problems, a many heuristic algorithms have been devised. Though it has been proved that the existence of a polynomial algorithm approximating the optimal solution within 1/4 log m, implies $P=NP$ [20], in practice most of the literature heuristics very efficiently yield near optimal solutions [9].
In the present paper we propose a class of randomized heuristic algorithms. The scheme of the algorithms is iterative. At each iteration a feasible solution is constructed; this is done by exploiting any constructive heuristic of the literature. The construction of solutions considers not only the usual goodness criteria, but also the "story" of previously produced solutions, following the ideas introduced in the so called "Ant System" [12]. We will study the combination of this general scheme with some of the most efficient classical heuristics. The class of Randomized algorithms (with or without memory) proved to give good results when applied to the set covering in a large collection of test problems, as reported in [18].
In Section 2 we introduce the basic notation and we review some of the most efficient constructive heuristic algorithms to be used within the new heuristics. Section 3 illustrates the randomized heuristic scheme as well as some of the algorithms which can be obtained. Section 4 contains some computational results. The randomized heuristic scheme is easy to parallelize. In Section 5 we discuss some parallel implementations of the algorithms and we present the performance evaluation of selected heuristics.

2. Problem formulation and some classical heuristic algorithms

The Set Covering Problem can be formulated in terms of integer linear programming. A 0-1 $m \times n$ matrix A is used to represent the collection of subsets F where coefficients a_{ij} are equal to 1 if and only if $i \in I_j$, and 0 otherwise. Decision variables x_j, $j = 1, \ldots, n$, are used to select the subsets in the cover, that is x_j will assume value 1 if and only if subset is in the cover S and 0 otherwise. Denoting by c_j the cost of including subset j in the solution, the problem is:

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} c_j x_j \\ & A x \geq e, \\ & x_j \in \{0,1\}, \qquad j=1,\ldots,n, \end{aligned} \tag{1}$$

where e denotes the unitary vector. Further on, we will denote the cost of a solution S by $Z(S) = \sum_{j \in S} c_j$. The most efficient exact algorithms are based on Branch&Bound [3, 6] and Branch&Cut techniques [1].

2.1 Constructive heuristic algorithms

Now we present a short survey of some of the most efficient constructive heuristics of the literature. These algorithms will be used within the general scheme of the randomized heuristic (see Section 3). For a more detailed review see [14], and [10].

2.1.1 Greedy algorithms

Greedy algorithms are among the simplest and most efficient in terms of computational time. The basic idea is to build a solution starting from $S = \emptyset$ and including in S, at each iteration, the “best” of the remaining subsets. There are several criteria to determine the best subset; these criteria characterize the different greedy algorithms. The general scheme of greedy algorithm is the following:

```
procedure greedy;
begin
    S:=Ø;
    while S does not cover I do
    begin
        select and remove I_j from F;
        S:=S∪{I_j};
        reduce(F)
    end
end.
```

Procedure **reduce**(F) eliminates from F those subsets dominated by subsets already selected in the solution. The selection of the subset can be made according to the criterion proposed by Chvátal [11]:

$$I_j = \text{argmax } \left\{ \frac{|I_k|}{c_k} : I_k \in F \right\}. \tag{2}$$

Other criteria ascribe different values to the numerator or to the denominator of (2), giving more strength to the cardinality of the subset or to its cost. For a detailed discussion see [2].

The greedy scheme is used as a base for the Greedy Randomized Adaptive Search Procedure (GRASP) [13]. The GRASP executes several times the greedy; but at each iteration of the greedy, instead of selecting deterministically the best subset, it picks up randomly one subset from the more promising ones determined by use of the selection criterion (2).

2.1.2 Beasley's algorithm

The heuristic algorithm proposed by Beasley [4] usually provides solutions whose value is very close to the optimum. It is a kind of Primal-Dual algorithm. The idea is to consider the Lagrangean dual of problem (1), where constraints $Ax \geq e$ are relaxed. The problem is

$$\max \{\varphi(u) : u \geq 0\},$$

and the Lagrangean function is

$$\min \left\{ \sum_{j=1}^{n} \left(c_j - \sum_{i=1}^{m} a_{ij} u_i\right) x_j : x_j \in \{0,1\}, j=1,\ldots,n \right\}.$$

The Lagrangean Dual is approached with a very simple and standard subgradient method where, at each iteration, a tentative value for multipliers u_i is fixed and the Lagrangean function $\varphi(u)$ is computed; then the value of u is updated and the procedure iterates until some stopping conditions hold. At each iteration

of the subgradient algorithm, a dual solution u is at hand; dual variables u are used to compute the *reduced cost* associated with every subset of F. The reduced cost of subset j is given by the following expression:

$$c_j - \sum_{i=1}^{m} a_{ij} u_i.$$

A feasible cover is obtained by selecting all subsets with non positive reduced cost, and, if there are still uncovered elements of I, the solution is completed by selecting the remaining subsets in a greedy fashion. At the end, the algorithm makes the solution minimal, that is it eliminates from the solution possible redundant subsets. Other Primal-Dual algorithms are studied theoretically in [7, 8], where a worst case bound to the solution value is provided.

3. The randomized heuristic scheme

The general randomized heuristic scheme that we propose is inspired by the Ant System ([12]), which is an adaptive metaheuristic that has been applied successfully to several combinatorial optimization problems. The proposed scheme is iterative: at each iteration a (possibly new) solution is constructed. The solutions are constructed according to a *probability law*. This probability law considers (i) the usual *attractiveness* factors (as for example function (2) used in the greedy algorithm or in the GRASP), in conjunction with (ii) a *memory factor*. The memory factor (called *trace*) takes into account the quality of the solutions already produced, in such a way that the best encountered solutions increase the probability that their elements are selected during the successive iterations. Note that this memory factor is not present in the GRASP, or in other randomized algorithms, but it is a feature typical to Ant Systems.

Let us discuss in greater detail the memory factor mechanism. Each subset of F is characterized by a *trace value*. This value is increased each time that the subset is included in a solution, and the trace increment is proportional to the quality of the solution. In the algorithm, at each iteration, the trace is updated by a set H of independent "agents", each of which constructs its own solution according to the given probability law.

The trace at the end of iteration i is given by the sum of the trace present at iteration i-1 multiplied by a *persistency coefficient* $\rho \in [0,1]$ and the traces produced by the agents in H during the i-th iteration. High values of persistency ρ give a long term memory effect. Formally the trace $\tau_j(i)$ associated with subset I_j at the end of the i-th iteration is given by:

$$\tau_j(i) = \rho \tau_j(i-1) + \Delta\tau_j(i), \qquad (3)$$

where $\Delta\tau_j(i)$ is the sum of the traces $\Delta\tau_j^h(i)$ related to subset I_j produced by each agent $h \in H$ during the i-th iteration, that is:

$$\Delta\tau_j(i) = \sum_{h \in H} \Delta\tau_j^h(i).$$

Let us recall that the trace has to take into account the quality of the produced solution (i.e. its cost). A possible expression of the trace is:

$$\Delta\tau_j^h(i) = \begin{cases} 1/Z^h(i) & \text{if } j \in S \text{ produced by agent } h \text{ during the } i\text{-th iteration,} \\ 0 & \text{otherwise,} \end{cases} \qquad (4)$$

where $Z^h(i)$ is the objective function value of the solution S produced by agent h during the i-th iteration. If the value of a lower bound Z_{lb} of the optimal solution is available, an alternative expression of the trace is:

$$\Delta\tau_j^h(i) = \begin{cases} 1/(Z^h(i) - Z_{lb}) & \text{if } j \in S \text{ produced by agent } h \text{ during the } i\text{-th iteration} \\ 0 & \text{otherwise} \end{cases} \qquad (5)$$

We can describe two different algorithmic schemes. In the first, we "randomize" the heuristic which constructs the solution at each iteration, while in the second we "randomize" the problem instance by suitably perturbing the costs.

3.1 Heuristics obtained by randomizing the greedy

The function **generate_solution**(τ,I,F,c) produces a feasible solution taking into account the costs as well as the trace associated with the subsets. In summary, the scheme of the algorithm is the following:

```
procedure randomized;
begin
    for each I_j∈F do
        τ_j(0):=0;                                              {trace initialization}
    for i=1,…,niter do
    begin
        for each h∈H do
        begin
            S:=generate_solution(τ,I,F,c);
            if Z(S) < Z(best_solution) then best_solution:=S;
            for each I_j∈F do compute_trace (Δτ_j^h(i),S)       {using (4) or (5)}
        end;
        update_trace(τ_j(i))                                    {using (3)}
    end;
    return(best_solution)
end.
```

This general scheme can be used to devise different heuristics depending on the implementation of the function **generate_solution**(τ,I,F,c). Obviously, as the function is called several times, in order to generate the solutions, efficient heuristics are preferred. To this end, we will study "randomized" versions of the greedy where decisions, instead of being deterministic, are taken according to a probability law which considers both attractiveness and memory factors. Other constructive heuristics can equally be adapted to this general scheme as for example clustering heuristics [19]. The results of the randoomized clustering are reported in [16].

3.1.1 Randomizing the greedy

Let us consider the randomization of the greedy algorithm. At each iteration of the greedy the subset entering the solution is selected according to a probability law. The probability of selecting subset I_j at iteration i, is given by:

$$p_j(i) = \frac{[\tau_j(i\text{-}1)]^{\alpha}[\phi_j]^{\beta}}{\sum_j[\tau_j(i\text{-}1)]^{\alpha}[\phi_j]^{\beta}}, \qquad (6)$$

where ϕ_j denotes the attractiveness factor of subset j, and the summation in the denominator is over all the subset not yet included in the partial solution. An example of this factor is:

$$\phi_j = \frac{|I_j|}{c_j}, \qquad j=1,\ldots,n. \qquad (7)$$

The denominator of (6) normalizes the probability values in the interval [0,1]. Parameters α and β are useful to control the influence of memory and attractiveness. High values of α induce the algorithm to rapidly converge to a status in which the same solution is always generated. On the other hand, high values of β make the behavior of the algorithm extremely random. For a discussion on the best values of α and β see [12]. This algorithm will be called Randomized Greedy.

Note that the computational complexity of the randomized versions of greedy algorithm is unchanged with respect to the deterministic version, if we exclude the overhead due to the computation of the traces and the probabilities.

3.1.2 Randomizing Beasley's algorithm

The algorithms obtained by applying the randomized scheme with the greedy will be labelled *randomized-greedy*. This algorithm can be applied in conjunction with Beasley's heuristic. The idea is the following: at each iteration of Beasley's algorithm we have the reduced costs of the subsets in F. All the subsets which have reduced cost less than or equal to $-\varepsilon$ ($\varepsilon>0$) are put into the solution; the solution is completed by applying randomized-greedy to the remaining subsets using the reduced costs instead of the original costs. Obviously, this algorithm is much more time consuming than Beasley's algorithm, as at each iteration several solutions are constructed. We will call these algorithms *B-randomized-greedy*.

3.2 Heuristics obtained by perturbing the costs of the problem

The class of algorithms that we propose in this section, instead of randomizing the generation of solutions, introduces a random perturbation into the cost of the problem instance under consideration. The perturbation of the cost takes place by the same mechanism of the trace described in Section 3.1. We have a trace associated with each subset. The traces are computed and updated according to (3), (4) or (5).

Once the traces have been updated, the costs of the problem are modified. At iteration i, the probability of perturbing the cost c_j of subset I_j is given by:

$$p_j(i) = (1 - e^{-[\tau_j(i-1)]^{\alpha}[\phi_j]^{\beta}}), \qquad (8)$$

where ϕ_j is the usual attractiveness factor (7), and α and β are suitable parameters used to control the influence of memory and attractiveness. The new cost value c_j is given by a random number uniformly generated in the interval $[c_j - \delta, c_j]$. This implies that, the larger the trace of a subset is, the higher the probability of decreasing its cost.
The scheme of the heuristic is the following:

```
procedure perturb;
begin
    for each Ij∈F do
        τj(0):=τ0;                                          {trace initialization}
    for i=1,…,niter do
    begin
        for each h∈H do
        begin
            perturb_costs(c, τ);
            S:=heuristic(I, F, c);
            if Z(S) < Z(best_solution) then best_solution:=S;
            for each Ij∈F do compute_trace (Δτj^h(i),S)      {using (4) or (5)}
        end;
        update_trace(τj(i));                                {using (3)}
    end;
    return(best_solution)
end.
```

The function **heuristic**(I, F, c) computes a solution of the perturbed problem. To this aim, we can use any reasonable heuristic. The procedure **perturb_costs**(c, τ) modifies the problem instance by decreasing the costs according to probabilities (11). We will denote by *perturbed-greedy* and *perturbed-Beasley* the heuristic obtained by applying the above scheme to greedy and Beasley's algorithms.

4. Computational results

The algorithms have been tested on the set covering instances taken from OR-Library [5]. Before applying the algorithms, the problems have been reduced according to the reduction rules proposed in [3, 17]. For the considered test problems the reduction requires relatively little time; on the other hand, the numerical results of the algorithms applied to the reduced problems are comparable with those of the unreduced ones, while the computation time is much smaller. These results are reported in [15]. The considered test problems are of small-medium size: the number of elements m ranges from 300 to 400, while the number of subsets n ranges from 400 to 650. The density of non zero elements of matrix A is 2-5%. All algorithms have been implemented in C language and run on a Digital Alpha 200-4/166.
In Table 1 we present a summary of the results obtained by applying the heuristics of the literature described in Section 2. Column **opt** contains the optimal solution values, column **G** the results of the greedy implemented with rule (2), column **B** the results of Beasley's heuristic, and column **GRASP** the results of the GRASP. The GRASP iterates 320 times in order to compare it with the perturbed greedy and the randomized greedy whcih generate te same number of solutions. The results of the Beasley's heuristic may sometimes be different from those reported in [4] due to different settings of the subgradient algorithm used to solve the Lagrangean dual.
In Table 2 we report the results of the randomized scheme where the randomization of the greedy, and the B-randomized-greedy algorithms are used (**RG** and **RB)**, moreover we report the results of the randomized scheme where the greedy and Beasley's algorithm are applied to the problem with perturbed costs (**PG** and **PB**) The last column reports the results obtained letting the GRASP run the same amount of time as PB (column **GRASP***). The randomized scheme is implemented with 32 agents and 10 iterations; parameters α, β and ρ have been set to 1, 4 and 0.5, respectively and the initial trace $\tau_{ij}(0) = 0.01$. These values have been chosen according to the indications given in [12] and after some preliminary tests. To generate the trace we used (4) for RG, while we used (5) for and RB, since in this case we always have a lower bound.

Problem	opt.	G	B	GRASP
A.1	253	271	256	259
A.2	252	276	256	256
A.3	232	263	234	240
A.4	234	253	235	241
A.5	236	251	238	258
B.1	69	79	70	70
B.2	76	89	78	76
B.3	80	87	81	80
B.4	79	89	79	81
B.5	72	73	72	72
C.1	227	242	232	235
C.2	219	240	224	222
C.3	243	278	249	249
C.4	219	252	224	228
C.5	215	243	216	218

Table 1: value of the solutions obtained by the basic algorithms

The randomized scheme is implemented with 32 agents and 10 iterations; parameters α, β and ρ have been set to 1.3, 3 and 0.5, respectively and the initial trace $\tau_{ij}(0)$ =0.01, as in the previous case.

Problem	opt.	RG	RB	PG	PB	GRASP*
A.1	253	257	254	263	254	257
A.2	252	256	252	269	256	256
A.3	232	237	233	243	232	235
A.4	234	239	234	241	234	240
A.5	236	238	236	242	236	238
B.1	69	69	69	72	69	70
B.2	76	76	76	82	76	76
B.3	80	81	80	84	81	80
B.4	79	79	79	84	79	80
B.5	72	72	72	73	72	72
C.1	227	233	227	240	229	234
C.2	219	225	219	232	220	221
C.3	243	249	243	263	245	247
C.4	219	229	219	233	219	226
C.5	215	218	216	222	215	218

Table 2: value of the solutions obtained by the randomized and perturbed algorithms

Table 3 reports the computational time in seconds of some heuristics applied to problem A.1, which is one of the problems that requires more time.

G	B	GRASP	RG	RB	PG	PB
<0.01	1.40	2.40	13.72	807.88	1.42	245.15

Table 3: computational times of some heuristics applied to A.1

The best solution values are provided by RB and PB. These algorithms very often attain the optimum, and in general are very near to it. However, it should be noted that these two algorithms generate a larger number of solutions with respect to the other heuristics. Looking at the computational efficiency, we can note that perturbed-greedy is ten time faster than RG. This is due to the fact that while the randomized version has to compute the probabilities at each single step of greedy, the perturbed version computes the

probabilities only at the beginning of the greedy, when the costs are modified. Note also that PG and GRASP generate the same number of solutions, but PG is faster.

5. Parallel implementation of randomized heuristic scheme

The randomized scheme that we hve proposed has a natural and straightforward parallel implementation. In fact, we can assign the duty of every agent to a different processor. Here we discuss the parallel implementation of the randomized scheme. We report also some results on the performance of the most representative heuristics among those presented in the previous sections, obtained on a CRAY T3D.

5.1 A synchronous implementation

Let us present a synchronous version of the algorithms implemented on a distributed computing system without shared memory. We have a set of agents running in parallel. At iteration *i* of the randomized scheme, each agent $h \in H$ executes the following operations:

- generate the solution;
- compute the trace $\Delta\tau_j^h(i)$ using (4) or (5);
- send $\Delta\tau_j^h(i)$, $j=1,\ldots,n$, to all other agents in H;
- receive $\Delta\tau_j^k(i)$ from the other agents $k \in H$;
- update the trace using (3).

At the end of the generation of the solution, each agent broadcasts to all others the value of the trace related to the solution it has obtained, and receives from all the other agents the value of their trace so that it can compute the new trace value according to (3). This is the synchronization phase of the algorithm. Note that the information about the trace is contained in each processor, and after the synchronization phase is complete this information is consistent.

This implementation can be used for all heuristics, except RB. In fact, RB is more suitably implemented by a *master-slave* scheme, as it requires the computation of the dual solution which is global. The master executes iteratively the following tasks:

- compute the solution of the Lagrangean function and obtain the reduced costs;
- send the reduced costs to all the slaves (agents);
- receive the solution from all the agents $h \in H$;
- compute the best among the received solutions and update the multipliers u according to the subgradient method.

Each slave $h \in H$ executes the following operations:

- receive the reduced costs from the master;
- reduce the problem by putting in the solution the subsets with reduced cost $\leq -\varepsilon$;
- repeat K times:
  - generate the solution for the reduced problem;
  - compute the trace $\Delta\tau_j^h(i)$ using (4) or (5);
  - send $\Delta\tau_j^h(i)$, $j=1,\ldots,n$, to all the other agents in H;
  - receive $\Delta\tau_j^k(i)$ from the other agents $k \in H$;
  - update the trace using (3);
- send the value of the solution to the master.

5.2 An asynchronous implementation

We identified two main problems derived from the proposed parallel implementations. The first one is related to the synchronization phase. The presence of many relatively large messages circulating at the same time in the communication network may generate congestion in the processing system. The second one concerns the fact that the computation of the agents may take different periods of time, thus leaving some processors idle waiting for the synchronization. To overcome these two problems, a completely asynchronous algorithm is proposed.

The asynchronous algorithm is organized in a master-slave framework. The master has the following duties:

- receive $\Delta\tau_j^h$, j=1,…,n, from agents;
- update the trace $\tau_j := \rho\tau_j + \Delta\tau_j^h$, j=1,…,n;
- send the trace τ_j, j=1,…,n, to the agents that ask for it.

The slaves execute the following operations:

- ask for the trace τ_j, j=1,…,n, from the master;
- generate the solution;
- compute the trace $\Delta\tau_j^h$ using (6) or (7);
- send $\Delta\tau_j^h$, j=1,…,n, to the master.

Note that in this scheme, the agents are not obliged to execute the same number of iterations, and the trace is not updated as in the sequential algorithm. This may produce different solutions with respect to the sequential case.

5.3 Performance evaluation

As the trace is, actually, a global information, the algorithms could be equivalently implemented on a parallel computer with shared memory. Thus instead of broadcasting the trace information during each synchronization phase, we can exploit the Shared Memory Access library of CRAY MPP to update the trace τ and make it visible to all agents. The Shared Memory Access library efficiently emulates a shared memory environment in a distributed parallel computer such as CRAY T3D. From the preliminary experiments [15], parallel implementation utilizing the Shared Memory Access library resulted slightly more efficient than the implementation using the inter processor communication routines.
All the algorithms have been implemented in C language. Here we report some computational results of the Shared Memory version of RG, PG and PB, and of the master slave implementation of RB. Table 4 reports the computational times in seconds required by the execution of these algorithms on a CRAY T3D when the number of used processors is increased up to 64. For each execution the computation load is the same, that is, the constructive heuristic is called 320 times, and the number of agents is equal to the number of processors. The main purpose of these computational experiments is not the evaluation of the efficacy of the proposed algorithms, which has been already discussed in the previous section, but to verify scalability; that is, how the computing time decreases when the number of processors is increased. For this reason we do not report the numerical results of the solutions found, even though in some cases of the asynchronous implementation they may differ from those obtained with the sequential algorithm. The times have been obtained for test problem A.1.

n. proc.	PB	PG	RB	RG
1	97.7	4.8	411.3	12.4
2	59.1	2.5	489.3	6.3
4	30.5	1.3	354.8	3.2
8	16.2	0.7	171.6	1.7
16	8.8	0.4	99.58	1.0
32	5.1	0.2	78.18	0.7
64	3.1	0.1	64.43	0.7

Table 4: computing times (in sec.)

Note that PB, and RB generate more than 320 solutions since Beasley's heuristic is iterative: RB 320 solutions are generated at each iteration of the subgradient method, while in PB, the subgradient method is called 320 times. This explains the fact that the times of PB and RB are greater than in the other cases. In Figure 1 we compare the speed-up of the four algorithms, when the number of processors is increased.

The *speed-up* for a given number of processors (k) is given by the division of computing time of the sequential algorithm by the computing time of the parallel algorithm using k processors.The speed-up of the class of perturbed algorithms is higher than for the class of randomized ones, and has values which are quite interesting. For RB the synchronous implementation seems to be particularly unsuitable; in this case, in fact, most of the time is spent in waiting for the synchronization. This is due to the fact that the time required by each agent is highly dependent on the input data, and the number of iterations for each agent

is extremely variable. Thus, for this algorithm, a less tightly coupled implementation would be more suitable.

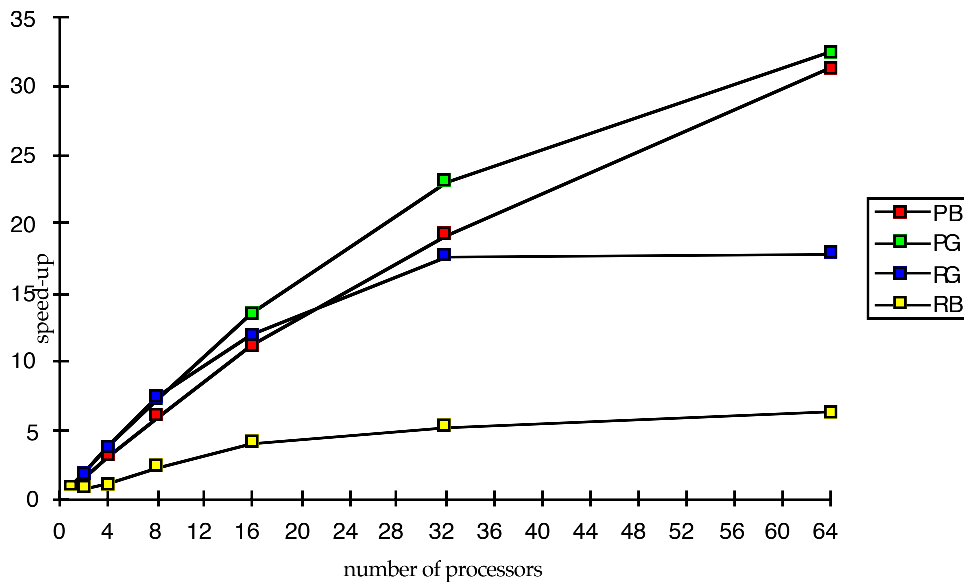

Figure 1: speed-ups

The speed-up of the class of perturbed algorithms is higher than for the class of randomized ones, and has values which are quite interesting. For RB the synchronous implementation seems to be particularly unsuitable; in this case, in fact, most of the time is spent in waiting for the synchronization. This is due to the fact that the time required by each agent is highly dependent on the input data, and the number of iterations for each agent is extremely variable. Thus, for this algorithm, a less tightly coupled implementation would be more suitable.

We implemented also the asynchronous version of the PB, which is the more interesting algorithm in the class of those proposed in this paper, and the one that could most benefit from possible improvement. The problem is perturbed 320 times. The interprocessor communications use the PVM routines. In Table 6 we report the computiation times, when the number of processors is increased: beginning with a configuration of one master and three slaves. The times have been obtained for test problem A.1.

n. proc.	PB
4	24
8	13
16	9
32	5
64	3

Table 6: computing times (in sec.)

PB achieves a good scalability also in the asynchronous implementation. It can be pointed out that it also has abetter performance with respect to the synchronous implementation when a small number of processors is used.

6. Conclusions and future work

In this paper we presented two classes of randomized heuristics for the set covering problem. The method, illustrated in detail in [16], iteratively applies either randomized versions of known constructive heuristics to the problem at hand, or the deterministic versions of the heuristic to random instances obtained by suitably perturbing the costs of the original problem. The randomization is in some way "adaptive" as it considers a memory factor. The computational results are encouraging: the solutions provided are often optimal or very close to the optimum. The heuristics derived from the proposed scheme can be easily parallelized. We have also presented a simple synchronous version of the algorithms, an asynchronous version of the most promising algorithm, and some performance measures. Due to the general features of the proposed algorithmic scheme, all the proposed approaches and the parallel implementations can be extended to other combinatorial optimization problems for which efficient constructive algorithms exist.

A similar approach has been applied in the field of flexible passenger transportation in a urban environment [21].

References:

[1] Balas E., S. Ceria, G. Cornuéjols, Mixed 0-1 programming by lift-and-project in a branch-and-cut framework, *Management Sciences*, Vol. 42, No. 9, 1996, pp. 1229-1246.

[2] Balas E., Ho A., Set covering algorithm using cutting planes, heuristics and subgradient optimization: a computational study. *Mathematical Programming*, Vol. 12, 1980, pp. 37-60.

[3] Beasley J.E., An algorithm for set covering problems. *European Journal of Operational Research*, Vol. 31, 1987, pp. 85-93.

[4] Beasley J.E., A lagrangian heuristic for set covering problems. *Naval Research Logistics*, Vol. 37, 1990, pp. 151-164.

[5] Beasley J.E., OR-Library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, Vol. 41, 1990, pp. 1069-1072.

[6] Beasley J.E., Jørnsten K. (1992) Enhancing an algorithm for set covering problems. *European Journal of Operational Research*, Vol. 58, 1992, pp. 293-300.

[7] Bertsimas D., Teo C.-P., From valid inequalities to heuristic: a unified view of primal-dual approximation algorithms in covering problems, *Working paper* OR 294-94, MIT, 1994.

[8] Bertsimas D., Vohra R., Linear programming relaxations, approximation algorithms and randomization; a unified view of covering problems, *Working paper*, MIT, 1994.

[9] Caprara A., M. Fischetti, P. Toth (1995), A Heuristic Method for the Set Covering Problem, Proc. of the Fifth IPCO Conference, Lecture Notes on Computer Science Vol 1084, pp. 72-84.

[10] Ceria S., P. Nobili, A. Sassano, Set Covering Problem, in Annotated bibliographies in Combinatorial Optimization, Dell'Amico M., F. Maffioli, S. Martello eds, John Wiley & Sons - Chichester, 1997, pp. 415-428.

[11] Chvatal V., A greedy heuristic for the set covering problem. *Mathematics of Operations Research*, Vol. 4, No. 3, (1979), pp. 233-235.

[12] Dorigo M., Maniezzo V., Colorni A., The Ant System: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 25 No. 12, 1996, pp. 29-41.

[13] Feo T.A., Resende M.G.C., A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, Vol. 8, 1989, pp. 67-71.

[14] Fiorenzo Catalano M.S. (1995) Euristiche per il problema di copertura. *Tesi di Laurea* Dipartimento di Informatica - Università di Pisa.

[15] Fiorenzo Catalano M.S., Malucelli F. (1995) Stochastic heuristics for the set covering. Pesented at Giornate di Lavoro AIRO ' 95 Ancona - Settembre 1995.

[16] Fiorenzo Catalano M.S., Malucelli F., Randomized heuristic schemes for the set covering problem, DEI - Politecnico di Milano, working paper 2000. (http://www.elet.polimi.it/Users/DEI/Sections/Automation/Federico.Malucelli/papers/stella.pdf).

[17] Garfinkel R. S., Nemhauser G. L., The set-partitioning problem: the set covering with equality constraints. *Operations Research*, Vol. 17, 1969, pp. 848-856.

[18] Grossman, T., A. Wool, Computational experience with approximation algorithms for the set covering problem. European Journal of Operational Research, Vol. 101, No. 1, 1997, pp.81-92.

[19] Kwatera R.K., Simeone B., Clustering heuristics for set covering. *Annals of Operations Research*, Vol. 43, 1993, pp. 295-308.

[20] Lund C., Yannakakis M., On the hardness of approximating minimization problems, *33rd IEEE Symposium on Foundations of Computer Science*, 1992.

[21] Malucelli F., M. Nonato and S. Pallottino, *Demand Adaptive Systems: some proposals on flexible transit* in Operational Research in Industry London, McMillan Press, 1999, pp. 157-182.