GRASP-Evolution for Constraint Satisfaction Problems

Manuel Cebrián
Departamento de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Madrid, Spain
manuel.cebrian@uam.es

Iván Dotú
Departamento de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Madrid, Spain
ivan.dotu@uam.es

ABSTRACT

There are several evolutionary approaches for solving random binary Constraint Satisfaction Problems (CSPs). In most of these strategies we find a complex use of information regarding the problem at hand. Here we present a hybrid Evolutionary Algorithm that outperforms previous approaches in terms of effectiveness and compares well in terms of efficiency. Our algorithm is conceptual and simple, featuring a GRASP-like (GRASP stands for Greedy Randomized Adaptive Search Procedure) mechanism for genotype-to-phenotype mapping, and without considering any specific knowledge of the problem. Therefore, we provide a simple algorithm that harnesses generality while boosting performance.

Categories and Subject Descriptors

I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic methods; G.2.1 [**Combinatorics**]: Combinatorial algorithms

General Terms

Algorithms, Performance, Experimentation

Keywords

Evolutionary Combinatorial Optimization, Random Binary CSPs, Constraint Handling, Heuristics, Hybridization

1. INTRODUCTION

Random binary CSPs is a widely used benchmark within the constraint programming community as an efficiency test for several algorithms and solvers. However, we can also find a wide spectrum of evolutionary approaches for solving random binary CSPs. In [1] we find a comprehensive comparison of those methods, such as SAW [2], Glass-box [3], MID, CCS, among others.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'06, July 8–12, 2006, Seattle, Washington, USA.
Copyright 2006 ACM 1-59593-186-4/06/0007 ...$5.00.

This paper presents the hybrid evolutionary algorithm GA-GRASP$_{Vo}$ for solving random binary CSPs. The algorithm is conceptual, simple and uses a key modeling based on the ideas in [4]. GA-GRASP$_{Vo}$ specifically applies the idea of a GRASP-like mechanism to perform genotype-to-phenotype mapping for solving random binary CSPs.

The main difference between our algorithm and that of [4], in terms of introducing a GRASP-like genotype-to-phenotype mapping, is that the genotypes represent two completely different aspects. In [4], the genotype represents values to assign to variables, while in our algorithm the genotype represents the order in which the variables will be tentatively assigned. This is a novel approach to solve CSPs, and it will be explained in more detail in the next sections.

We provide a comparison with two of the most successful state-of-the-art evolutionary algorithms as shown in [1]. Our simple algorithm outperforms the best approach in terms of effectiveness (measured by success rate, mean error at termination and average champion error) while outdoing it also in terms of efficiency (measured by the average number of evaluations to find a solution). Furthermore, it compares with the best approach in terms of efficiency, while outperforming it in terms of effectiveness.

The main contributions of this research are:

- A novel representation which focuses on finding an optimal variable ordering, and that borrows ideas from [4] for a GRASP-like genotype-to-phenotype mapping.
- A general evolutionary algorithm which can be easily suited to solve any kind of CSP problem without considerable implementation effort.
- Outstanding results that outperform and compare with the best evolutionary algorithms which usually involve complex heuristics and fitness adjustment functions.
- Showing that a simple algorithm can yield outstanding results if an appropriate modeling is chosen; therefore, stating the importance of representation in evolutionary strategies.

The rest of the paper is organized as follows: first we briefly introduce constraint satisfaction problems for Evolutionary Algorithms (EAs). We then introduce the GRASP framework and present our hybrid algorithm. The following sections are devoted to the experimental comparison with other methods: in section 5 we define our test-suite problem (random binary CSPs), in section 6 we introduce the methods against which our algorithm will be compared, section

```
procedure GRASP(maxIt,seed)
  1. Read_Input()
  2. for k=1,..., maxIt do
  3.     Solution ← Greedy_Randomized_Construction(seed);
  4.     Solution ← Local_Search(Solution);
  5.     Update_Solution(Solution);
  6. end;
  7. return Best_Solution;
  end GRASP
```

Figure 1: The GRASP pseudocode

```
procedure Greedy_Randomized_Construction(seed)
  1. Solution ← ∅
  2. Evaluate the incremental costs of candidate elements
  3. While Solution is not complete do
  4.     Build the restricted candidate list RCL
  5.     Select element s from RCL at random
  6.     Solution ← Solution ∪ {s};
  7.     Reevaluate the incremental costs;
  8. end;
  9. return Solution;
  end Greedy_Randomized_Construction
```

Figure 2: The Greedy Randomized Construction pseudocode

7 describes the measures we use for the comparison, and section 8 shows the experimental results and the comparison itself. The paper ends with some conclusions and future work.

2. CONSTRAINT SATISFACTION PROBLEMS AND EAs

In a *constraint satisfaction problem* (CSP) we are given a set of variables, where each variable has a domain of values, and a set of constraints acting between variables. The problem consists of finding an assignment of values to variables in such a way that the restrictions imposed by the constraints are satisfied.

We can also define a CSP as a triplet $< X, D, C >$, where $X = \{x_1, \ldots, x_n\}$ is the set of variables, $D = \{D_1, \ldots, D_n\}$ is the set of nonempty domains for each variable x_i, and $C = \{C_1, \ldots, C_m\}$ is the set of constraints. Each constraint is defined over some subset of the original set of variables $\{x_1, \ldots, x_n\}$ and specifies the allowed combinations of these variable values. Thus, solving the CSP is equivalent to finding a complete assignment for the variables in X with values from their respective domain set D, such that no constraint $C_i \in C$ is violated.

The evolutionary framework presents the issue of constraint handling: constraints can either be handled directly or indirectly [5].

- **Indirect handling** involves transforming the constraint into an optimization objective which the EA will pursue; while,
- **Direct handling** leaves the constraint as it is, and enforces it somehow during the execution of the algorithm.

Direct handling is not oriented for EA due to the lack of an optimization function in the CSP, which would result in no guidance towards the objective. Thus, indirect handling is the best suited approach for EA, although a mixed strategy where some constraints are enforced and some are transformed into an optimization criteria is suited as well.

3. GREEDY RANDOMIZED ADAPTIVE PROCEDURES

The GRASP (Greedy Randomized Adaptive Search Procedure) metaheuristic can be viewed as an iterative process, each iteration consisting of two phases: construction and local search [6]. The construction phase builds a solution whose neighborhood is investigated by the local search procedure. During the whole process, the best solution is updated and returned at the end of a certain number of iterations. Figure 1 illustrates the basic GRASP procedure.

Any local search algorithm can be incorporated to improve a solution: tabu search and simulated annealing [7, 8], large neighborhoods [9] or variable neighborhood search [10]. However, we are interested in the greedy construction phase, where a tentative solution is built in a greedy fashion.

Randomly generated solutions are usually of a poor quality, while greedy generated solutions tend to be attracted by local optima, due to the less amount of variability. A *greedy randomized heuristic* [11] adds variability to the greedy algorithm. A certain greedy function yields a ranked candidate list, which is called *restricted candidate list* (RCL). An element from that list is randomly selected and added to the solution.

The procedure to construct the *greedy randomized* solution is depicted in Figure 2. A key step in this pseudocode is the selection of an attribute from the RCL. This can be performed using a qualitative or quantitative criterion. In the former, the element is selected among the k best elements; while in the latter, the element is selected among the elements with a quality $\alpha\%$ of the greedy value. Note that $k = 1$ or $\alpha = 100$ yields a pure greedy selection.

Reactive GRASP

As can be seen in the procedure described below, the selection of the k parameter is problematic. The use of a fixed value for this parameter could hinder high quality solutions [12]. A learning-based strategy named *reactive* GRASP was introduced in [13], selecting a different value in each iteration from a finite set of values. The selection of a certain value in a given iteration can be chosen on the basis of the goodness of the best solution generated by this parameter. A possibility is to maintain a vector of parameter values to use in each iteration, where a position p_i denotes the value of the parameter that serves to choose the $i-th$ candidate. From now on we will refer to this vector as *GRASP parameters vector*.

For example, a certain position of the GRASP parameters vector $p_i = 3$ makes us choose a random candidate among the four best candidates, for the $i-th$ decision, in the RCL list (rrom now on we will consider that the first value in the RCL is in position 0 and the last one $n-1$, where n would be the length of the RCL).

4. THE HYBRID EVOLUTIONARY ALGORITHM

We now turn to the hybrid evolutionary algorithm for solving CSP problems. The algorithm maintains a population of GRASP parameters and performs a number of iterations until a solution is found. In each iteration it selects two individuals of the population and, with some probabilities, crosses and/or mutates them. The next population will be obtained in an elitist fashion.

Following the framework presented in [1], our algorithm consists of a generational evolutionary model with an elitist selection of the new generations, a one-point crossover recombination operator, and a mutation operator which selects, for each s_i (each single GRASP parameter in the vector), a uniformly random new value (subject to a given probability); the parent selection is performed in a binary tournament fashion and the constraint handling is purely indirect by using GRASP parameters. It presents neither fitness adjustments nor use of heuristics. Each of these characteristics is now reviewed in more detail.

Evolutionary model

The algorithm consists of a generational strategy where new populations are selected in an elitist fashion, which means that in each new generation, the population is calculated by maintaining the best individuals among the previous population and the offspring.

Fitness function

In order to calculate the fitness of a certain individual, we will take into account how many variables are in conflict with the rest. Thus, the fitness function would be as follows:

$$f(s) = \sum_{j=1}^{n} \upsilon(s, C_j), \text{where,}$$

$$\upsilon(s, C_j) = \begin{cases} 1, & \text{if } s \text{ violates at least one } c \in C_j; \\ 0, & \text{otherwise.} \end{cases}$$

Where s is a complete assignment of values to variables , C_j is the set of constraints containing variable v_j and n is the number of variables.

Note that, even though we name it *fitness function*, we want to minimize its value, since this is equivalent to reducing the number of violated variables in the problem. Indeed, if $f(s) = 0$ then the assignment s produces no violations, and, therefore, it is a solution.

Crossover

The hybrid EA uses a one-point crossover for crossing two individuals σ_1 and σ_2. It selects a random number r in $1..n$ and the child is obtained by selecting the first r genes from σ_1 and the remaining $n - r$ genes from σ_2.

Mutation

The mutation here is achieved by, with a given probability, randomly selecting a new value for each single GRASP parameter in the vector that defines the individual.

Parent selection

In each iteration the algorithm selects two individuals in the population (the parents). This is performed in a binary tournament fashion: randomly selecting two individuals from the population and choosing the best of them; the same is carried out for the second parent.

Representation

This is the most important feature in our algorithm, in fact, the rest of the characteristics are common in simple evolutionary schemes. However, the representation of the CSP is a key factor in the algorithm efficiency. In order to define our implementation, we must introduce some basic concepts.

GRASP parameters vector

Our population is then, a set of GRASP parameter vectors. In [4] a Hybrid GRASP - Evolutionary algorithm for finding Golomb rulers is introduced. Our representation makes the same use of the GRASP features as in the mentioned algorithm. In the same manner, the value of each parameter defines the exact candidate to select, instead of a range for a random selection as defined in the GRASP section.

The value of each parameter reflects the decision to take in this step, forcing us to make the decision ranked in the position indicated by the value. Decisions are ranked according to some quality criteria, thus, a parameter value 0 will involve making the "best" decision. A vector with all parameters set to 0 corresponds to a plain greedy strategy.

Parameters concordance

Solving a CSP usually implies assigning values to variables iteratively until either a consistent solution has been reached, or the problem has been proved to be unsolvable. Every time a variable is *instantiated* with a value, a consistency test is performed to ensure that the rest of the variables will have consistent values to be assigned to. If this test fails, the procedure will backtrack to the previous decision (to the last consistent variable instantiation) and try to assign a different value to the current variable. If the test is positive the procedure will choose a new variable to instantiate.

However, there is a crucial element on the efficiency of this solving procedure: the order in which the variables are chosen for instantiation. This is called the *variable ordering heuristic*. According to [14], the ordering heuristic for assigning variables is a key factor in quickly finding a feasible solution. Based on that, we will assume that it is possible to assign the variables in a certain order such that we will be able to find a solution assigning values that do not generate conflicts. This can seem a fuzzy assumption, and perhaps an optimistic statement; however, it is true. If you could know beforehand the search tree of a solution, you could reproduce it by assigning a value that yields no violations (the value that appears in the solution) to every variable in the order that the search tree indicates. This will produce a valid solution.

Let us exemplify this with a toy CSP. Imagine we have three variables x, y and z. The common domain of the variables is $D = 0, 1, 2$. The variables are subject to two constraints:

$$c_0 :: x + y = 2$$
$$c_1 :: z + y = 1$$

Now we are going to try to solve the problem (finding a consistent assignment of values to variables) by instantiating the variables in lexicographic order. Very briefly, we will assign $x = 0$, then $y = 2$ to be consistent with c_0 and we

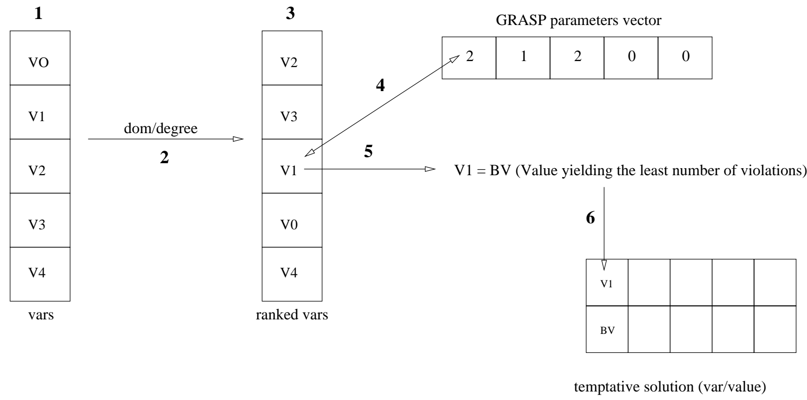

Figure 3: Assigning the first variable using the GRASP parameters vector in 6 steps: Step 1 shows the variables available to select. Step 2 applies the dom/degree heuristic to these variables. Step 3 shows the resultant RCL list. Step 4 selects the candidate variable that the GRASP parameters vector indicates. In Step 5 this variable is instantiated with the best value possible and the last step reflects this selection and instantiation in the first position of a vector that represents an actual tentative solution of the problem.

would not be able to assign a value to z that satisfies c_1. However, if we would have ordered the variables y, x, z, we would have assigned $y = 0$, then $x = 2$ to satisfy c_0, and finally $z = 1$ satisfying c_1. Note that the order in which we assign the values is also important. In our approach we assume a static lexicographic ordering, and we will always assign the first value that yields no violations.

Therefore, we are going to transform the problem of finding values for the variables (approach followed in [4]) into finding an optimal ordering for the variables that will yield a feasible solution. Our vector of GRASP parameters will allow us to choose, among the ranked variables, which one we want to instantiate next. The variables will be dynamically ranked using the dom/degree ordering heuristic [14] (quality criteria), which gives more weight to variables with few available values in its domain, and that take place in a greater amount of constraints (Note that this heuristic will yield the ordering introduced in the example above where we were able to immediately find a solution). The values that the parameters can take, will fall within the range $[0, n - pos_i]$, where $pos_i \in 1..n$ is the position of the given parameter within the vector. In this case, the last parameter will always be 0, since there is just one variable left to assign. Once we have selected a variable we will instantiate it with the best value possible (the value that yields the least amount of constraint violations). In Figure 3 this process is explained in 6 steps.

It is worth mentioning that, opposite to [4], we allow non feasible instantiations. This follows immediately from the fact that we are considering a feasibility problem, instead of an optimization problem. In the latter we are searching for the best feasible solution, hence, we can restrict the search to feasible solutions; while in the former we are searching for the best unfeasible solution, which corresponds to a feasible solution (the one with less constraint violations).

4.1 The Hybrid Algorithm

We are now ready to present the hybrid EA GA-GRASP$_{Vo}$ which is depicted in Figure 4. Lines 2-4 perform the initializations. In particular, the population is randomly generated in lines 2-3 and the generation counter g is initialized in line 4.

1. GA-GRASP$_{Vo}$(csp)
2.   **forall** $i \in 1..populationSize$
3. $\Sigma \leftarrow \Sigma \cup \{$RANDOMCONFIGURATION$(csp.n)\}$;
4. $g \leftarrow 0$;
5.   **while** $g \leq maxGen$ & $v(\Sigma) > 0$ **do**
6. $i \leftarrow 0$;
7. $\Sigma^+ \leftarrow \emptyset$;
8.   **while** $i \leq populationSize$ **do**
9.   **select** $(\sigma_1, \sigma_2) \in \Sigma$;
10.   **with probability** P_c
11. $\sigma^* \leftarrow$ crossover(σ_1, σ_2);
12.   **if** $v(\sigma^*) == 0$
13.   return σ^*;
14.   **with probability** P_m
15. $\sigma^* \leftarrow$ mutate(σ^*);
16.   **if** $v(\sigma^*) == 0$
17.   return σ^*;
18. $\Sigma^+ \leftarrow \Sigma^+ \cup \{\sigma^*\}$;
19. $i \leftarrow i++$;
20. $\Sigma \leftarrow$ **select**(Σ^+, Σ);
21. $g \leftarrow g + 1$;

Figure 4: Algorithm GA-GRASP$_{Vo}$ **for CSP problems.**

The core of the algorithm is in lines 5-21. They generate new generations of individuals for a number of iterations or until a solution is found. The new generation is initialized in line 7, while lines 8-19 create the new generation. The new individuals are generated by selecting the parents in line 9, applying a crossover with probability P_c (lines 10-11), and applying a mutation with probability P_m (lines 14-15). The new individuals are added to the new population in line 18. The current population is selected among the previous and the new population in line 20. Note that after crossover and mutation we need to calculate the cost of the individual in order to detect solutions and/or keep track of the cost in order to properly select parents and next population.

p	E(solutions)
0.24	1707299.07
0.25	258652.614
0.26	38984.6092
0.27	5600.99655
0.28	838.870129
0.29	125.400589
0.30	19.6420135
0.31	2.79148238
0.32	0.42173145
0.33	0.06618763

Table 1: The Smith's conjecture prediction of the number of solutions as a function of p.

5. OUR BENCHMARK: RANDOM BINARY CSPS

In this paper we consider random binary constraint satisfaction problems, since their properties in terms of difficulty to be solved are well-understood and hence such problems have been used for testing the performance of algorithms for solving binary CSPs. In [15] it was shown that any CSP can be equivalently transformed to a binary CSP, thus without a loss of generality.

Various problem instance generators have been developed for the class of binary CSPs, based on several theoretical models. All of these models are parameterized by n, m, D, and k, where n is the number of variables, m is the number of constraints, D is the number of values in each domain and k is the arity of each constraint. In a binary constraint network, the value of k is fixed to 2.

There are four traditional models, called A, B, C and D developed from a general framework presented in [16] and [17], all of which are unsolvable with high probability. In our work we use the E model proposed by Achioptlas *et al.* [18], which has the advantage that it generates solvable benchmarks. This model is usually specified as $E(n, p, D, k)$ with p defined as $p = m[\binom{n}{k}D^k]^{-1}$ and works by choosing uniformly, independently and with repetitions, conflicts between two values of two different variables. We are aware of the existence of another good random binary CSPs generator, Model F [19], and we plan to use it as a benchmark generator in future experiments.

Our test suit consists of 250 solvable problem instances available on the Web [20] and used also as a benchmark in [1]; they are generated using the model $E(20, p, 20, 2)$ with 25 solvable instances for each value of p in {0.24, 0.25, 0.26, 0.27, 0.28, 0.29, 0.30, 0.31, 0.32, 0.33}. By using the conjecture of Smith [17] we show that the range for p in model E actually runs through the mushy region. The term mushy region is used to indicate the region where the probability that a problem is soluble changes from almost zero to almost one. Within the mushy region, problems are in general difficult to solve or to prove unsolvable. In Table 1 it can be seen that the predicted number of solutions drops bellow one when moving from $p = 0.31$ to $p = 0.32$, precisely what defines the mushy region.

6. RELATED WORK

There are several evolutionary algorithms focused on solving random binary CSPs [1]. Most of them use knowledge of the problem, either to develop heuristics or to implement a fitness adjustment technique. The nice feature of our algorithm is that no knowledge about the problem is taken advantage of, hence, harnessing generality without a loss in efficiency or effectiveness.

In this section we are going to briefly introduce the two most successful approaches according to [1], which will be later used to compare against our algorithm.

SAW

The basic idea behind the SAW (Stepwise Adaptation of Weights) algorithm lies in the way that the fitness function is evaluated. Each k evaluations[1] the variables causing the constraint violations in the best individual of the current population are given a high weight (penalty), because they are considered to be harder than the others. These weighted-up variables will have a greater impact in the fitness of the following evaluations. A comprehensive study of different parameters and genetic operators of SAW can be found in [2].

Glass-box

Glass-box works by decomposing complex constraints in two steps: elimination of functional constraints and decomposition of the CSP into primitive constraints, usually of the form $\alpha\cdot p_i - \beta\cdot p_j \neq \gamma$ where p_i and p_j are the values of variables v_i and v_j. A common repair rule used is the following

$$\text{if} \quad \alpha\cdot p_i - \beta\cdot p_j = \gamma \quad \text{then change} \quad v_i \quad \text{or} \quad v_j \qquad (1)$$

Repairing a violated constraint can result in the production of new violated constraints, thus at the end of the repairing process, the chromosome will not in general be a solution. An extensive work on this constraint processing technique is presented in [3] and [21].

7. MEASURES OF EFFECTIVENESS AND EFFICIENCY

Genetic algorithms are random algorithms, therefore the behavior of the optimization in a problem instance varies from execution to execution. In order to obtain a more accurate idea of the performance of an algorithm in a concrete binary CSP instance, we are going to run it 10 times for each problem instance, thus having 250 executions for each p value (10 executions for each 25 problem instances belonging to a concrete p value). The set of executions for each p value is denoted by S_p.

An execution is finished when the genetic algorithm finds the solution or when a given number of evaluations is reached. An evaluation is the calculation of the fitness of an individual (in this case the number of variables violating a constraint).Thus, we define θ as the maximum number of evaluations for each execution. Now, we are ready to define some effectiveness and efficiency measures as a function of p.

Effectiveness

Effectiveness is measured by the success rate (SR), the mean error at termination (ME) and the average champion error (ACE).

[1] In [1] the period k is set to 25 evaluations.

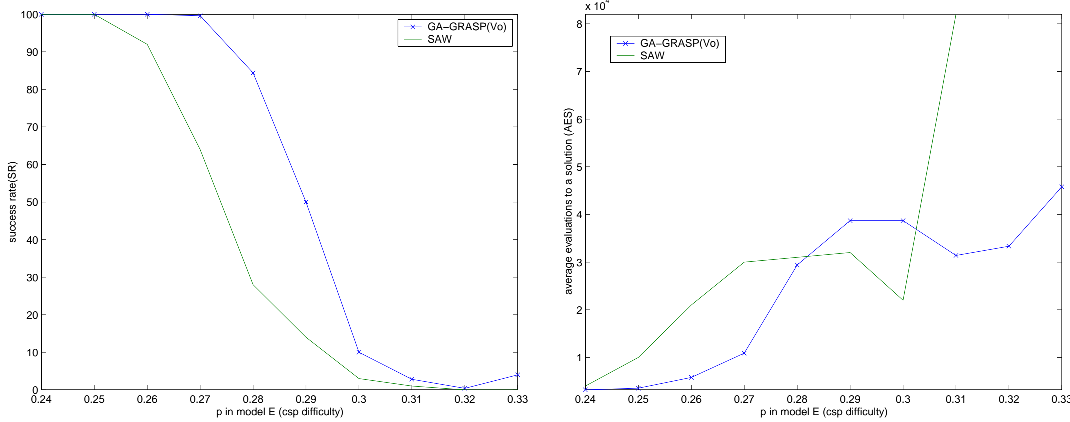

Figure 5: SR and AES measures from the GA-GRASP$_{Vo}$ and SAW algorithms.

We define S_p^+ as the executions of S_p that found a solution before θ evaluations, and $S_p^- = S_p - S_p^+$. The SR over p is the percentage of runs that find a solution in no more than θ evaluations.

$$SR(p) = 100\frac{|S_p^+|}{|S_p|}$$

The error at termination (ET) is defined for a single run as the number of constraints violated by the best candidate solution in the population when the execution reaches θ evaluations. If an execution finishes before θ evaluations, then its ET is considered 0, thus the mean error at termination (ME) is defined as

$$ME(p) = \frac{1}{|S_p|}\sum_{s\in S_p^-} EAT(s)$$

We use another effectiveness measure that focuses on the convergence speed of the algorithm. We define the champion error (CE) as the number of constraints violated by the best individual found up to a given time (measured in evaluations) during a run, thus the average champion error is defined as

$$ACE(p,t) = \frac{1}{|S_p|}\sum_{s\in S_p} CE(s,t)$$

If s has finished before t evaluations, then $CE(s,t) = 0$.

Efficiency

In our experiments, we use the average number of evaluations to find a solution (AES) in order to measure efficiency. The AES is the average number of evaluations to find a solution (ES) over the successful runs S_p^+.

$$AES(p) = \frac{1}{|S_p^+|}\sum_{s\in S_p^+} ES(s)$$

It is important to note that if $S_p^+ \ll S_p^-$ then AES is statistically unreliable.

Another interesting efficiency measurement is the average conflict checks needed to find a solution, used in [1]. Unfortunately, it is not possible to find a correspondence between the typical conflict check and the way in which our algorithm computes constraint violations.

8. EXPERIMENTAL RESULTS

We have chosen the SR, ME, ACE and AES measures in order to compare the measures obtained from our algorithm GA-GRASP$_{Vo}$ with same measures of the best ones of the algorithms analyzed in [1]. The test suite (the 250 instances generated from model E available in [20]),the limit of evaluations ($\theta = 100000$)[2] and our mutation probability parameter (set to 0.3 in all the experiments) are also the same.

In the following two subsections we compare our algorithm with the winners of effectiveness and efficiency of the analysis performed in [1].

8.1 Comparing against the most effective algorithm: SAW

The most important measure in evaluating effectiveness is the success rate, because the main goal of an algorithm for solving CSPs is to obtain a solution. In [1] it is shown that the overall winner regarding success rates is the SAW algorithm.

In Figure 5 we give a comparison of the SR measures between the GA-GRASP$_{Vo}$ and SAW algorithms. SAW is outperformed by GA-GRASP$_{Vo}$ in all p values. Moreover, if we consider the global success rate for all p's, we obtain an overall SR of 55% for GA-GRASP$_{Vo}$ and 44% for SAW, which implies more than a 10% of successful executions.

Unfortunately, due to the way in which the fitness function is computed in SAW, its ME and ACE measures cannot be compared to those from GA-GRASP$_{Vo}$. This is explained because the fitness of the SAW algorithm is not the number of constraint violations, but a weights-scaled function.

We include an efficiency comparison in Figure 5, the average number of evaluations to a solution. In all but two values of p the GA-GRASP$_{Vo}$ requires less evaluations than SAW. Curiously, from $p = 0.24$ to $p = 0.30$ the two algorithms seem to converge, but from that point onwards the

[2]This limit is achieved by a population size of 1000 individuals, a maximum generations limit of 50 and two evaluations for each individual and generation: after crossover and after mutation. Similar results can be achieved with a smaller population size and with a larger amount of generations by means of a simple restarting policy. Thus, diversity is very important for the performance of the algorithm in this benchmark.

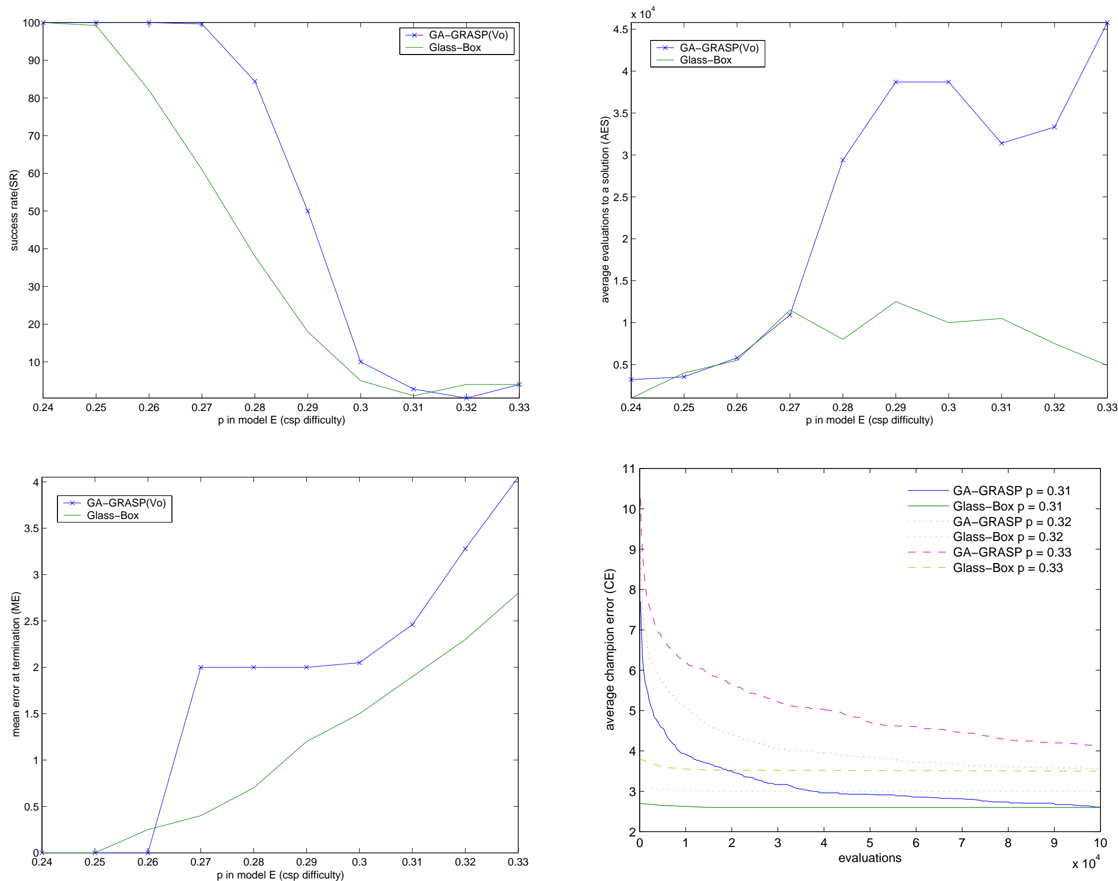

Figure 6: Efficacy and efficiency measures from the GA-GRASP$_{Vo}$ and Glass-Box algorithms.

AES for SAW growths exponentially while having only a slight increase for GA-GRASP$_{Vo}$.

8.2 Comparing against the most efficient algorithm: Glass-box

In [1] it is shown that the winner regarding Efficiency, measured by the number of fitness evaluations, is the Glass-box genetic algorithm.

In Figure 6 we give a comparison of the efficiency measure (AES) between the GA-GRASP$_{Vo}$ and Glass-box algorithm. In the easy region (0.24 to 0.27) GA-GRASP$_{Vo}$ needs a comparable number of evaluations, but is surpassed in terms of efficiency by Glass-Box in the mushy region. The average number of solutions over all p's is 24077 for GA-GRASP$_{Vo}$ and 7889 for Glass-box, being the last a 32% more efficient.

In efficacy terms, the two algorithms are more balanced. GA-GRASP$_{Vo}$ outdoes Glass-box in the whole easy region and in the beginning of the mushy region (0.24 to 0.31), with an equilibrium in the rest of the values. In overall terms the Glass-Box successfully finishes a 40% of the executions, a 15% less than GA-GRASP$_{Vo}$.

Observing the ME and ACE of Figure 6 it can be see that the quality of the partial solutions during the execution is slightly better for Glass-box, specially in the mushy region, where it has 1 less violated constraint on average at the end of the run.

In all efficiency measures we have used the evaluation as the unit of computational effort. In order to obtain a more realistic picture about the efficiency of the algorithms, we have computed an average of the CPU time consumed by an evaluation: 0.0062 ± 0.0028 seconds on a Pentium IV at 2.8 GHz with 512 MBytes of RAM.

9. CONCLUSION AND FUTURE WORK

In this paper we have presented a hybrid evolutionary algorithm for solving random binary CSPs, which yields outstanding results, as it outperforms the best previous approach in terms of effectiveness, and compares with the best strategy in terms of efficiency.

Our hybrid algorithm incorporates features of GRASP, in a similar way as in [4], where a GRASP-like mechanism is applied to genotype-to-phenotype mapping. However, our approach features a novel representation which focus on finding a variable ordering instead of a value to variable assignment.

The rest of the algorithm is conceptual and simple, making no use of information regarding the problem, which harnesses generality. It also demonstrates that modeling (or representation) is a key factor in evolutionary strategies.

Moreover, we believe there is a large space for improvement. Learning techniques and restart policies should be introduced and tested. We are also studying hybridizations with local search techniques that are already yielding very promising results.

Finally, we are interested in using real life CSP bench-

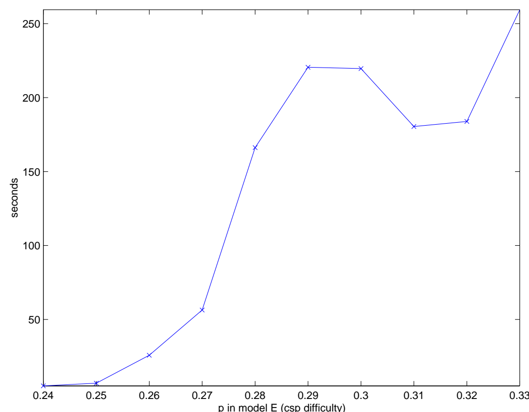

Figure 7: Average time to solution of the GA-GRASP$_{Vo}$ **for several values of** p**.**

marks in order to compare results with constraint programming techniques and other evolutionary approaches available. Binary CSPs are not very common in real life applications and even though any CSP can be transformed into a binary CSP in polynomial time [15], we plan to generalize our solver to deal with n-ary CSPs.

Acknowledgments

This work was partially supported by grant TSI 2005-08255-C07-06 of the Spanish Ministry of Education and Science. We would like to thank Carlos Cotta and Antonio Fernández for several useful discussions on their publication [4]. Finally, we would also like to thank the reviewers for many useful comments.

10. REFERENCES

[1] B. G. W. Craenen, A. E. Eiben and J. I. van Hemert. Comparing Evolutionary Algorithms on Binary Constraint Satisfaction Problems. *IEEE Transactions on Evolutionary Computation*, 7(5): 424-445, October 2003.

[2] B. G. W. Craenen and A. E. Eiben. Stepwise Adaption of Weights with Refinement and Decay on Constraint Satisfaction Problems. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, pages 291-298, July 2001.

[3] E. Marchiori. Combining constraint processing and genetic algorithms for constraint satisfaction problems. In *Proceedings of the 7th International Conference on Genetic Algorithms*, pages 330-337, 1997.

[4] C. Cotta and A. Fernández. A Hybrid GRASP-Evolutionary Algorithm Approach to Golomb Ruler Search. *Parallel Problem Solving From Nature VIII*, number 3242, pages 481-490, 2004.

[5] A. E. Eiben. Evolutionary algorithms and constraint satisfaction: Definitions, survey, methodology, and research directions *Theoretical Aspects of Evolutionary Computation*, pages 13-58, 2001.

[6] M. G. C. Resende and C. C. Ribeiro. Greedy randomized adaptive search procedures. *Handbook of Metaheuristics*, pages 219-249, 2003.

[7] H. Delmaire, J.A. Díaz, E. Fernández, and M. Ortega. Reactive GRASP and Tabu Search based heuristics for the single source capacitated plant location problem. *INFORMS Journal on Computing*, 37:194-225, 1999.

[8] X. Liu, P. M. Pardalos, S. Rajasekaran and M. G. C. Resende. A GRASP for frequency assignment in mobile radio networks. *Mobile networks and computing*, 52:195-201, 2000.

[9] R. K. Ahuja, J. B. Orlin, and D. Sharma. New neighborhood search structures the capacitated minimum spanning tree problem. Technical Report, Florida, 1998.

[10] P. Hansen and M. Mladenović. An introduction to variable neighborhood search. *Meta-heuristics, Advances and trends in local search paradigms for optimizations*, pages 433-458, 1998.

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

[12] M. Prais and C. Ribeiro. Parameter version in GRASP procedures. *Investigación Operativa*, 9:1-20, 2000.

[13] M. Prais and C. Ribeiro. Reactive GRASP: an application to a matrix decomposition problem in TDMA traffic assignment. *INFORMS Journal on Computing*, 12:164-176, 2000.

[14] I. Gent, E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proc. 2nd International Conference on Principles and Practice of Constraint Programming (CP-96)*, pages 179-193, 1996.

[15] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proc. 9th European Conf. Artificial Intelligence ECAI'90*, pages 550-556, 1990.

[16] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *J. Artif. Intell.*, 81:81-109, 1996.

[17] B. Smith and M. Dyer. Locating the phase transition in binary constraint satisfaction problems. *J. Artif. Intell.*, 81(1-2):155-181, 1996.

[18] D. Achlioptas, L. Kirousis, E. Kranakis, D. Krizanc, M. Molloy and Y. Stamatiou. Random constraint satisfaction: A more accurate picture. *Constraints*, 4(6):329-344, 2001.

[19] B. G. W. Craenen. JavaCsp: a random binary constraint satisfaction problem instance generator in Java. http://www.xs4all.nl/~bcraenen/JavaCsp/download.html

[20] CSP Problem Instances Using Model E (2002). http://www.cs.vu.nl/~bcraenen/resources/csps_modelE_v20_d20.tar.gz

[21] P. van Hentenryck, V. Saraswat and Y. Deville. Constraint processing in cc(FD). *Constraint Programming: Basics and Trends*, 1995.