Comparing the Performance of Evolutionary Algorithms for Permutation Constraint Satisfaction

Luis de-Marcos, Antonio García, Eva García, José-Amelio Medina, Salvador Otón
Computer Science Department. University of Alcalá
Ed. Politécnico. Alcalá de Henares. Madrid. Spain
{luis.demarcos; a.garciac; eva.garcial; josea.medina; salvador.oton}@uah.es

ABSTRACT
This paper presents a systematic comparison of canonical versions of two evolutionary algorithms, namely Particle Swarm Optimization (PSO) and Genetic Algorithm (GA), for permutation constraint satisfaction (permut-CSP). Permut-CSP is first characterized and a test case is designed. Agents are then presented, tuned and compared. They are also compared with two classic methods (A* and hill climbing). Results show that PSO statistically outperforms all other agents, suggesting that canonical implementations of this technique return the best trade-off between performance and development cost for our test case.

Categories and Subject Descriptors
I.2.8 [Artificial Intelligence]:Problem Solving, Control Methods, and Search – heuristic methods.

General Terms
Algorithms, Performance, Experimentation.

Keywords
constraint satisfaction, CSP, PSO, genetic algorithm

1. PROBLEM STATEMENT
A constraint satisfaction problem (CSP) is a problem composed of a set of variables that must be given a value and set of constraints that limit the values that those variables can take. Thus the aim of a CSP-problem solver is to find an assignment for all the variables satisfying every constraint [1]. If all the solutions of a CSP are permutations of a tuple, then the CSP it is said to be a permut-CSP. A permut-CSP is Loosely Constrained (LC-permut-CSP) when the set of constraints does not reduce extremely the number of feasible solutions. That usually happens because the density of constraints is low resulting in problems easier to solve than harder instances which have very few, if any, feasible solutions. LC-permut-CSPs also have many applications in real world problems in a wide variety of domains. LC-permut-CSPs are then important for many practitioners and engineers.

Following Tsang [1], we will define a CSP as a triple *(X,D,C)* where $X = \{x_0, x_1, \dots, x_{n-1}\}$ is finite set of variables, D is a function that maps each variable to its corresponding domain *D(X)*, and $C_{ij} \subset D_i \times D_j$ is a set of constraints for each pair of values *(i, j)* with $0 \leq i < j < n$. To solve the CSP is to assign all variables x_i in X a value from its domain D, such that all constraints are satisfied. A constraint is satisfied when $(x_i, x_j) \in C_{i,j}$, and (x_i, x_j) it is said to be a valid assignment. If $(x_i, x_j) \notin C_{i,j}$ then the assignment (x_i, x_j) violates the constraint.

If all solutions from a CSP are permutations of a given tuple then it is said that the problem is a permutation CSP or PermutCSP. A PermutCSP is defined by a quadruple *(X,D,C,P)* where *(X,D,C)* is a CSP and $P=<v_0, v_1, \dots, v_{n-1}>$ is a tuple of $|X|=n$ values. A solution S of a PermutCSP must be a solution of *(X,D,C)* and a complete permutation of P. LC-CSPs may be characterized as CSPs or PermutCSPs in this way.

As for the fitness function a standard penalty function will be used. This is a common choice when the domain of the problem does not provide any objective function.

$$f(X) = \sum_{0 \leq i < j < n} \mathrm{V_{i,j}}(\mathrm{x_i}, \mathrm{x_j}) \quad (1)$$

where $\mathrm{V_{i,j}}: \mathrm{D_i} \times \mathrm{D_j} \rightarrow \{0,1\}$ is the violation function

$$\mathrm{V_{i,j}}(\mathrm{x_i}, \mathrm{x_j}) = \begin{cases} 0 \text{ if } (x_i, x_j) \in C_{i,j} \\ 1 \text{ otherwise} \end{cases} \quad (2)$$

One hundred random permut-CSPs are generated to input the algorithms that are subsequently tested. 24 variables are used and 20 to 40 binary constraints are randomly created. Please note that each constraint involves two variables exactly. They are binary constraints and thus we will be dealing with randomly generated binary permut-CSPs. A class of randomly generated binary CSPs is characterized by the 4-tuple $\{n, m, p_1, p_2\}$ [2]. m is the number of variables and n is the number of values in each variable domain. p_1 is the constraint density. It is the portion of the $n \cdot (n-1)/2$ constraints in the graph. Considering that the average test case has 35 constraints p_1 will be .127. p_2 is a measure of the tightness of constraints. It determines the number of incompatible pairs of values for each constraint.

The problem may seem under-constrained and then it would result easy to solve for any solver. To further justify our decisions we will turn now to studies on phase transition. Transitions have an easy-hard-easy structure and the region in which changes occur is called the mushy region [3]. It determines the area where most of the hard instants exist and therefore it may be thought as the most promising area to test problem-solvers. The mushy region is wider when smaller values of p_1 are employed and it is narrower when the constraint density is higher [4]. That means that with a lower value of p_1 it is more likely to create hard instances when a random problem generator is employed. Prosser [4] further reports that, in his experiments, when p_1=.1 there is a higher variability in search effort, even well before the phase transition. We find this to be a very desirable characteristic to test the performance of problem solvers since harder instances will be present among a fair amount of easy instances. As the density of the constraint graph increases search effort for the hardest instances increases too but variability diminishes as the mushy region narrows and

GECCO'11, July 12–16, 2011, Dublin, Ireland.
ACM 978-1-4503-0690-4/11/07.

consequently hard instances are more difficult to find. A similar behavior is observed as n increases. The mushy region narrows (search effort also increases exponentially). We then need to find a balance between computational cost and solvability. As we previously explained we set n=24 to be able to run a sufficient number of tests to have statistical significance. As for the domain size (m) it is set as m=n because we have a permut-CSP in which all solutions are complete permutations of a given tuple. Prosser study also asserts that the width of the mushy region does not change significantly as domain size varies, so this setting does not have any significant influence on the final results.

Our objective is to focus on algorithms that offer a reasonable performance and that can be developed and tested using also reasonable resources. Two fairly recent evolutionary optimization methods have shown a good balance between performance and complexity in a wide variety of problems in different domains. Genetic algorithms were introduced in the mid-70s [5] and they simulate natural selection processes in order to find solutions to problems. Particle swarm optimization (PSO) was introduced in the mid-90s [6, 7] as a new problem solver based on the foraging behavior observed in social insects like bees. Our aim is to systematically test and compare these evolutionary techniques to solve CSPs focusing, simultaneously, on the trade-off between their efficiency, and their development (and tuning) requirements.

A canonical permut-PSO agent is designed to solve permut-CSP. Original PSO is intended to work on continuous spaces. We use the version that is designed to deal with permutation problems introduced in [8]. A fully informed swarm is preferred based on empirical evidence too [9]. A permut-GA agent with order recombination, swap mutation and generational replacement with elitism is also implemented in order to test its performance for solving LC-CSPs.

2. EXPERIMENTATION

PSO and GA permut-CSP agents are subsequently tuned in order to find the best configuration. Each configuration is inputted with the 100 different test cases previously described and data of each execution are collected for statistical analysis. Data transformation is required because gathered data is not normally distributed but rather it seems to follow a lognormal distribution. This can be explained partially by the long tail (higher values in the number of calls to the fitness function) which may be representative of the most difficult instances of the random problems. They give information about the behavior of the problem solvers in the worst case. A logarithmic transformation (base-10) is then applied and normality tests are also performed. Kolmogorov-Smirnov tests are used to prove normality. For the PSO agent four configurations are tested. ANOVA tests are then performed to determine the best one. As for the GA agent four different parameters require tuning and 16 different configurations are tested covering a wide range of values. A General Linear Model is used to determine which parameters influence the final performance, which are further analyzed using ANOVA to determine the best possible values.

Both approaches are finally compared to test their relative performance to solve LC-permut-CSPs. Basic versions of two classic algorithms, random-restart hill climbing (HC) and A*, are also implemented to compare them with evolutionary approaches. Best configurations found for each evolutionary algorithm is used for the comparative analysis. For the PSO the canonical version is used. As for the GA, the optimal parameter settings determined by previous experimentation are employed (μ=20, k=2μ/3, p=.1).

An initial overview of descriptive statistics of both algorithms seems to shows that PSO approach outperforms the GA. This is confirmed with an ANOVA test (F=38.28, p=.000, R^2=16.2). Both PSO and GA also outperform HC and A* algorithms. Confidence intervals are presented in figure 1. We can then conclude that a canonical version of the PSO is a better option to solve random LC-permut-CSPs. PSO also has less parameters and tuning is actually not required if we follow the recommendations available on literature. On the other side, GA has several parameters in which decisions need to be made. Considering that the kind of experimental random LC-CSPs that we have employed here bears important resemblances with many real world optimization problems, this study suggest that such problems should firstly be approached using a PSO algorithm rather than a GA. This would result in better performance in terms of efficiency as well as in terms of development effort.

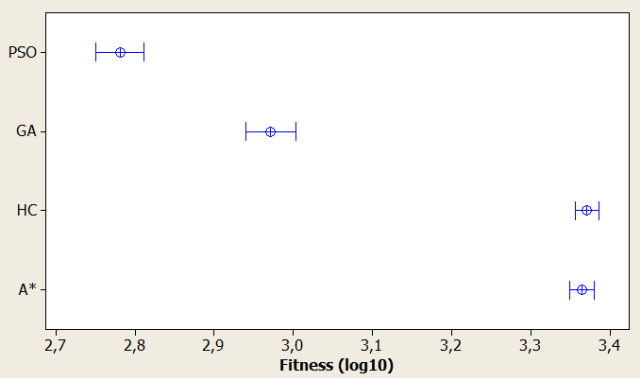

Figure 1. Confidence intervals of fitness for each algorithm (CI=95% of the mean).

3. REFERENCES

[1] Tsang, E. *Foundations of Constraint Satisfaction*. Academic Press, London, 1993.

[2] Solnon, C. Ants Can Solve Constraint Satisfaction Problems. *IEEE Transactions on Evolutionary Computation*, 6, 4 2002), 347-356.

[3] Hogg, T., Huberman, B. A. and Williams, C. P. Phase transitions and the search problem. *Artificial Intelligence*, 811996), 1-15.

[4] Prosser, P. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 811996), 81-109.

[5] Holland, J. H. *Adaptation In Natural and Artificial Systems*. The University of Michigan Press, Michigan (USA), 1975.

[6] Eberhart, R. and Kennedy, J. *A new optimizer using particle swarm theory*. City, 1995.

[7] Kennedy, J. and Eberhart, R. *Particle swarm optimization*. City, 1995.

[8] Hu, X., Eberhart, R. C. and Shi, Y. *Swarm intelligence for permutation optimization: a case study of n-queens problem*. IEEE Press, City, 2003.

[9] Mendes, R., Kennedy, J. and Neves, J. *The fully informed particle swarm: simpler, maybe better.* Evolutionary Computation, IEEE Transactions on*, 8, 3 2004), 204-210.*