International Journal of Computational Intelligence and Applications
Vol. 15, No. 3 (2016) 1650017 (22 pages)
© World Scientific Publishing Europe Ltd.
DOI: 10.1142/S1469026816500176

A New Parallel GA-Based Method for Constraint Satisfaction Problems

Reza Abbasian and Malek Mouhoub*

Department of Computer Science, University of Regina
Regina S4S 0A2, Canada
**mouhoubm@uregina.ca*

Received 2 December 2015
Revised 24 May 2016
Published 7 September 2016

Despite some success of Genetic Algorithms (GAs) when tackling Constraint Satisfaction Problems (CSPs), they generally suffer from poor crossover operators. In order to overcome this limitation in practice, we propose a novel crossover specifically designed for solving CSPs including Temporal CSPs (TCSPs). Together with a variable ordering heuristic and an integration into a parallel architecture, this proposed crossover enables the solving of large and hard problem instances as demonstrated by the experimental tests conducted on randomly generated CSPs and TCSPs based on the model RB. We will indeed demonstrate, through these tests, that our proposed method is superior to the known GA-based techniques for CSPs. In addition, we will show that we are able to compete with the efficient MAC-based Abscon 109 solver for random problem instances as well as those instances taken from Lecoutre's CSP library. Finally, we conducted additional tests on very large consistent and over constrained CSPs and TCSPs instances in order to show the ability of our method to deal with constraint problems in real time. This corresponds to solving the CSP or the TCSP by giving a solution with a quality (number of solved constraints) depending on the time allocated for computation.

Keywords: Parallel genetic algorithms (PGA); constraint satisfaction; evolutionary techniques.

1. Introduction

A Constraint Satisfaction Problem (CSP) consists of a finite set of variables with finite domains, and a finite set of constraints restricting the possible combinations of variable values.[1] A solution tuple to a CSP is a set of assigned values to variables that satisfy all the constraints. A binary CSP is a CSP where each constraint involves at most two variables. A binary CSP is often represented by a graph where vertices correspond to variables while edges represent the constraints between these variables. In this paper, we focus on the case of binary CSPs (and TCSPs) using the graph representation. Note that our work can be easily generalized to nonbinary CSPs. We can in this case adapt our proposed techniques to handle nonbinary constraints or simply convert nonbinary CSPs into binary ones using dual encoding or hidden variables encoding methods.[2]

Many real-life applications under constraints can be efficiently represented and solved with CSPs.[3–7] In some of these applications such as scheduling and planning, the constraints correspond to numeric and symbolic temporal information. This has motivated us to develop a CSP-based model, TemPro, for managing CSPs involving numeric and symbolic temporal constraints.[8,9] More formally, TemPro translates an application involving temporal information into a binary CSP where variables are temporal events defined on domains of numeric intervals and binary constraints between variables correspond to disjunctions of Allen primitives.[10] This latter is called a Temporal CSP (TCSP).[a] A CSP is known to be an NP-complete problem in general,[b] a backtrack search algorithm of exponential time cost is needed to find a complete solution. In order to overcome this difficulty in practice, systematic solving methods based on constraint propagation techniques have been proposed in the literature.[1] The goal here is to reduce the size of the search space before and during the backtrack search. While these proposed techniques have a lot of merits when tackling small and medium size problems, their combination with the backtracking algorithm suffers from the exponential time cost of this latter especially for large size problems. An alternative is to use approximation methods such as Genetic Algorithms (GAs). Despite some success of GAs for solving CSPs,[12–15] they generally suffer from poor crossover operators in solving constraint problems. The main reason for such a phenomenon is that in CSPs, changing the value of a variable can have direct effects on other variables that are in constraint relation with the changing variable and indirect effect on other variables. As a result, performing a random crossover can often reduce the quality of the solution.

In this paper, we propose a novel crossover, that we call Parental Success Crossover (PSC), specially designed for solving CSPs using GAs. PSC is integrated into a parallel architecture where our Genetic Modification (GM) function injects at the beginning of each cycle good individuals generated based on information (constraints violations) gathered at previous cycles. More precisely, at the end of each cycle GM gathers the information about constraint violations and order the variables accordingly before generating good individuals through backtracking with constraint propagation.

In order to assess the performance of our proposed crossover over the basic one-point crossover, Asexual Crossover for CSP (ASXC), Multi-Parent Crossover (MPC) as well as well-known heuristic-based GAs for CSPs,[12,13] we conducted several experiments on CSP and TCSP instances randomly generated using the model RB.[16] This model is a revision of the standard Model B,[17] has exact phase transition and the ability to generate asymptotically hard instances. The test results

[a] Note that this name and the corresponding acronym was used in Ref. 11. The TCSP, as defined by Dechter *et al.*, is a quantitative temporal network used to represent only numeric temporal information. Nodes represent time points while arcs are labeled by a set of disjoint intervals denoting a disjunction of bounded differences between each pair of time points.

[b] There are special cases where CSPs are solved in polynomial time, for instance, the case where the related constraint network is a tree.[1]

clearly show that our proposed crossover outperforms the considered GA methods on all problem instances in terms of success rate and time needed to reach the solution. In addition, we evaluated the performance of an integration of our crossover, together with a variable ordering heuristic,[18] within our Hierarchical Parallel GA (HPGA) architecture we have proposed for solving graph coloring problems (GCP).[19] The results are very appealing especially when dealing with hard problem instances. Moreover, our proposed method is able to compete with the efficient MAC-based Abscon solver for CSPs[20] as demonstrated by the comparative experiments conducted on random CSPs with different sizes as well as those instances taken from Lecoutre's library.[21] Finally, our GA-based solving method incorporates many greedy principles and has the ability to solve a CSP by giving a solution with a quality (number of solved constraints) depending on the time allocated for computation. This is clearly demonstrated through several tests we conducted both on consistent and inconsistent (over constrained) random CSPs and TCSPs. This "anytime behavior" is very relevant in practice especially for real-time applications where a solution needs to be returned within a given deadline. The user can, for instance, get a solution with a given quality at a particular time point or let the program run for another amount of time, if this can be afforded, to get a better quality.

The rest of the paper is structured as follows. The next section summarizes the different CSP solving techniques. Our proposed GA-based method and its integration into the HPGA architecture is then covered in Sec. 3. Section 4 reports the results of comparative experiments we conducted on randomly generated CSPs and TCSPs. Finally, concluding remarks and future directions are listed in Sec. 5.

2. CSP Solving Techniques

2.1. *Backtrack search and constraint propagation*

The backtrack search method is a depth-first search technique that extends a partial solution to a complete and consistent one. More precisely, at each step one variable will be assigned a value from its domain and the partial solution gets checked for consistency. This standard method is not efficient due to thrashing; that is the search repeatedly failing due to the same reason which could be identified earlier in the search. In order to overcome this difficulty in practice, local consistency techniques have been proposed.[1] The aim of these constraint propagation techniques is to enforce the consistency on a subset of CSP variables before and during the backtrack search. One of the most known forms of local consistency is called Arc Consistency (AC).[22] More formally and in the case of binary CSPs, for each pair of variables (x_1, x_2) sharing a constraint, AC removes from the domain of x_1 any value that is inconsistent with all the values of x_2 domain. When used before the search, the goal of AC is to reduce the size of the search space before Backtracking takes place. When used during the search, AC helps to detect later failure earlier following a lookahead technique such as Forward Checking (FC) or Maintaining Arc Consistency

(MAC).[23] Each time we assign a value to a variable, FC enforces AC on this latter variable and all future active variables (variables not assigned yet). MAC extends this constraint propagation technique to the subset containing all the nonassigned variables. Abscon 109[20] the solving platform we used for our comparative experimental tests in Sec. 4 uses a variant of MAC procedure.

2.2. *Metaheuristics*

The goal of the constraint propagation techniques we presented in the previous section is to reduce the size of the search space before and during the backtrack search. While these techniques have lots of merits when tackling small and medium size problems, their combination with the backtracking algorithm suffers from the exponential time cost of this latter especially for large size problems. An alternative, in practice, is to use approximation methods. While these randomized algorithms do not always guarantee to find a consistent solution, they are in general faster than backtrack search. Following a greedy approach, in most of these metaheuristic methods such as local search and evolutionary algorithms, the search starts with one or more complete assignments and then changes are made to make the assignments satisfy more and more constraints until reaching the solution. The problem with these algorithms is that because of their randomness, they cannot be used to prove the inconsistency of a CSP. In cases where the problem does not have a solution, these algorithms would run forever without success. In practice, a time limit should be set after which the algorithm returns no solution. To use these algorithms, we need to define a representation of the potential solution and a fitness function to measure the quality of a solution. In CSPs, the representation is an array of variables and the value in each index of the array represents the value of the corresponding variable. The fitness function measures the number of unsatisfied constraints or if constraints have weights, the sum of the weights of the unsatisfied constraints.

2.3. *Variable ordering heuristics*

The ordering of the variables in a backtrack search has a tremendous effect on the size of the search space. Following a given criterion, a heuristic is used to choose what variable to assign next at each step of the search. The well-known first-fail principle is a criterion that has been suggested for evaluating different heuristics. This principle consists of picking, at each time, the most constrained variable and supposes that the best search order is the one that minimizes the length or the depth of each branch. Variable ordering heuristics can be defined into two categories: Static Variable Ordering (SVO) and Dynamic Variable Ordering (DVO). SVO heuristics use the initial structure of the constraint network and maintain the same variable ordering during the search to decide the next variable to assign a value to. Smallest Domain First (SDF), is a SVO in which variables are sorted based on their domain size so that variables with smaller domain are checked first. The reasoning behind this is that with all other factors being equal, the variable with the least number of values would have less sub trees rooted at those values. Another method for SVO is maximum degree (deg)[24] which chooses the variable that has the maximum degree in the constraint graph. DVO use the information about the current state of the search to decide on the next variable to assign. A very well-known heuristic of this type, known as dom,[25] selects the next variable that has the least remaining values in its domain and thus constrains the remainder of the search space the most. The dynamic version of deg, called ddeg, chooses the variables that are involved in the least amount of constraints with unassigned variables. dom/ddeg[26] is derived from combining these two heuristics. More precisely, it selects a variable that has the minimum ratio of the current domain to the current dynamic degree. Impact-based heuristics[27] focus on the search tree size as a criterion for variable ordering. They measure the importance of each variable in reducing the size of the search space by considering the changes its value assignments can make to the size of the search tree. Conflict driven variable ordering[28] is a technique that uses MAC to gather the information about constraints violations. More precisely, each constraint has a weight that is incremented every time the constraint causes a domain wipe out during the constraint propagation phase. In the wdeg technique, the variable with the largest weight is then selected. Here the variable weight is the sum of the weights of the constraints that the variable is involved in. The combination of wdeg and dom, called dom/wdeg, prefers the variable that has the least ratio of current domain size to the current weighted degree. The obvious drawback to the wdeg technique is that for the first few choices which are also the most important ones, the search does not have enough information to choose the best variables, and that can tremendously affect the size of the search space. To address this issue, Weighted Information gathering (WNDI) and RANDom Information gathering (RNDI) have been proposed in Ref. 29 using the dom/wdeg heuristic. However, they do a number of search restarts to gather information from different parts of the search space before starting the main search process. Having this information makes it possible to make better choices for first variables. In RNDI, a variable is selected randomly at each variable selection point during the search for the first $R - 1$ runs. On the final restart, dom/wdeg is used in the normal way. However, the weights are not initialized to zero but they are set to the weights that the search learnt from previous random probes. This would enable the search to make better early decisions. In WNDI the search updates the weights consequently in all runs and uses the information gathered from each run at the start of the next run. Experiments in Ref. 30 show that RNDI can perform better as it learns from more diverse parts of the search space and therefore can give a better approximation of the areas of global contention. In Ref. 18 two SVO hybrid methods combining systematic and heuristic techniques are proposed. The first one is an iterative algorithm based on Hill Climbing (HC) while the second is a constructive approach based on Ant Colony Optimization (ACO).[31] Both methods tackle the CSP by first using HC or ACO to gather information about the search space during the search. Then, the information gained is used to sort the variables before the backtrack search. Abscon 109 uses dom/wdeg as a variable ordering heuristic.

3. Proposed GA-Based Method for CSPs and TCSPs

3.1. *Background: Parallel genetic algorithms*

In GAs, there is a population of potential solutions called individuals. The GA performs different genetic operations on the population, until the given stopping criteria are met. The Parallel Genetic Algorithm (PGA) is a parallel architecture allowing several GAs to run in parallel.

There are mainly three different types of PGAs.[32] First, the Master-Slave PGA (MSPGA) in which, there is only one single population divided into fractions. Each fraction is assigned to one slave process on which genetic operations are performed.[33] Second, the Multi-Population PGA which contains a number of subpopulations, which can occasionally exchange individuals. The exchange of individuals is called migration. Migration is controlled using several parameters. Multi-population PGAs are also known as Island PGAs (IPGAs), since they resemble the "island model" in population genetics that considers relatively isolated demes. Finally, the Fine-Grained PGA which consists of only one single population, that is designed to run on closely linked massively parallel processing systems. In this paper we use a Master-Slave architecture for designing the PGA, called HPGA, that we have proposed in Ref. 19 for solving GCP. Section 3.8 provides a detailed description of HPGA and its related procedure.

3.2. *Individual representation in CSPs*

In GAs each possible solution to the given problem is represented using an encoding known as the chromosome (or individual). Each chromosome contains a number of genes. To represent a potential solution to a CSP in GAs, we normally use an array structure. Each index of the array is considered as a gene that corresponds to a variable in the CSP. The data that each gene carries is from the domain value of the variable it represents. Moreover, each individual has a fitness corresponding to the total number of constraint violations. An individual with a fitness equal to zero is a solution to the problem. For instance, Fig. 1 illustrates the GCP modeled as a CSP and its representation using GAs. The GCP consists of coloring the graph vertices such that no two adjacent nodes (nodes linked by an edge) have the same color.

3.3. *Individual representation in TCSPs*

TemPro[8] transforms a temporal problem under qualitative and quantitative constraints into a binary CSP where constraints are disjunctions of Allen primitives (see Fig. 2 for the definition of the Allen primitives) and variables, representing temporal events, are defined on domains of time intervals. Each event domain (called also temporal window) contains the Set of Possible Occurrences (SOPO) of numeric intervals the corresponding event can take. The SOPO is the numeric constraint of the event. It is expressed by the fourfold: [earliest start, latest end, duration, step] where: earliest start is the earliest start time of the event, latest end is the latest end

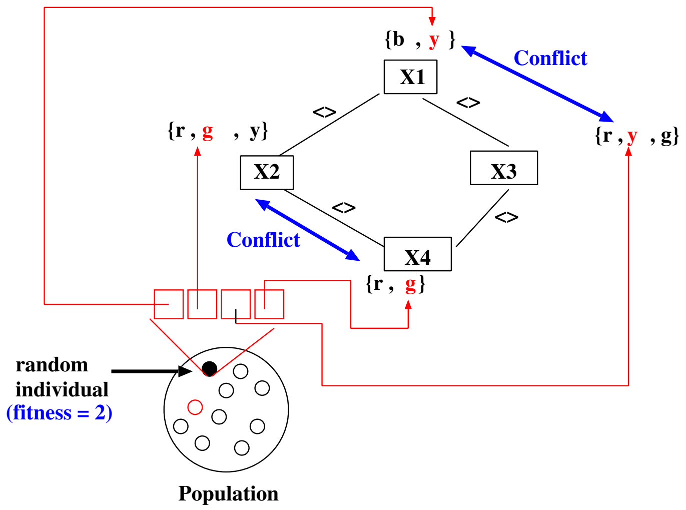
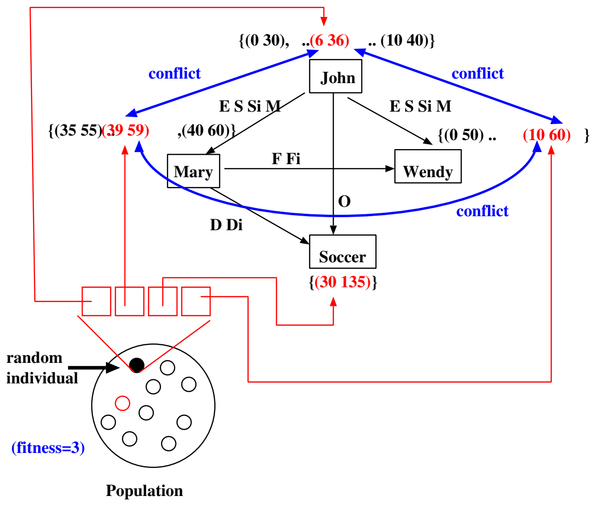


Fig. 1. GA representation of the GCP.

time of the event, duration is the duration of the event and step is the discretization step corresponding to the number of time units between the start time of two adjacent intervals. In order to better understand TemPro and its related components, let us consider the following temporal constraint problem.

Example 1.

(1) *John, Mary and Wendy separately rode to the soccer game.*
(2) *It takes John 30 min, Mary 20 min and Wendy 50 min to get to the soccer game.*



Relation	Symbol	Inverse	Meaning
X precedes Y	P	P-	X Y
X equals Y	E	E	X Y
X meets Y	M	M-	X Y
X overlaps Y	O	O-	X Y
X during Y	D	D-	X Y
X starts Y	S	S-	X Y
X finishes Y	F	F-	Y X

Fig. 2. GA representation of the TCSP in Example 1.

(3) *John either started or arrived just as Mary started.*
(4) *John left home between 7:00 and 7:10.*
(5) *Mary arrived at game between 7:55 and 8:00.*
(6) *Wendy left home between 7:00 and 7:10.*
(7) *John's trip overlapped the soccer game.*
(8) *Mary's trip took place during the game or else the game took place during her trip.*
(9) *The soccer game starts at 7:30 and lasts 105 min.*
(10) *John either started or arrived just as Wendy started.*
(11) *Mary and Wendy arrived together but started at different times.*

Figure 2 shows a graph corresponding to the TCSP of Example 1. There are four main events: John, Mary and Wendy are going to the soccer game respectively and the soccer game itself. Some numeric constraints specify the duration of the different events, e.g., "*20 min is the duration of Mary's event*". Other numeric constraints describe the temporal windows in which the different events occur. For instance the earliest start time of John's event is 7:00 and the latest start is 7:10. Given these two information and the fact that the duration of John's trip is 30 min we can produce the domain of John's event which will correspond to the following set of possible intervals (given that the discretization step is 1 min): $\{(0\ 30), (1\ 31), \ldots, (10\ 40)\}$.

Finally, symbolic constraints state the relative positions between events. For instance, the relation between John's and Wendy's events, "*John either started or arrived just as Wendy started*", can be represented with the following disjunction of Allen primitives: John $(E \vee S \vee S^{\smile} \vee M)$[c] Wendy.

An individual corresponds to a given temporal scenario and represents a complete assignment of intervals to all the events of the problem. For each individual, the fitness function is computed by counting the number of conflicts (constraint violations) due to the complete assignment. An individual with a fitness equal to zero is a solution to the TCSP (since all the temporal constraints are satisfied). For example, the fitness of the individual shown in Fig. 2 is equal to 3.

3.4. *Crossovers for GAs*

Several crossovers have been proposed in the literature. The simplest one is the One Point Crossover (OPC) that works as follows. First, we randomly choose a crossing point. All the genomes (individuals) from the first parent that are before the crossing point will be included in the offspring. In addition, all the genomes in the second parent that are after the crossing point, will be included as well in the offspring. The ASXC is proposed by Eiben *et al.*[13,14] The idea here is as follows. To produce the offspring we use an asexual crossover that selects, from the chromosome, a group of

[c]Note that the inverse of a given Allen primitive, S for instance, can also be denoted as Si.

variables having the largest number of violated constraints and change their respective value such that the number of violations (or conflicts) are minimized. The group size should be determined at the beginning, but was suggested to be 1/4 of the individual size.[13,14] As per the description in Refs. 13 and 14, the MPC for CSPs operates by scanning the genes of the parents consecutively. It then chooses a value that has occurred the most for that gene (CSP variable) amongst the parents and assigns it to the corresponding gene of the offspring. The value selection can also be performed randomly or based on the fitness of the parents.

3.5. *Proposed PSC*

Our goal here is to develop a new crossover that minimizes the number of constraints violations as much as possible. Unlike the well known methods described in the previous Section, our proposed technique relies on the past crossover history of the different individuals, in addition to their respective constraint violations. The intuition here is that learning from both parents past history in terms of constraints violations can be very useful when deciding on how to cross them. In this regard, we propose the following crossing technique. When choosing two parents p_1 and p_2 for a crossover, the offspring is produced as follows. For each gene (corresponding to an assignment of a given value to a variable) in parent p_1 (respectively p_2) we compute the number of conflicts due to this assignment. We then select the gene with the minimum number of conflicts. If both genes (from p_1 and p_2) have the same number of conflicts then we rely on past success history of p_1 and p_2 and choose the gene from the parent with a better "parental success ratio". This parental success ratio is computed as follows. For each parent p_i in the population we maintain two numbers: the number of times p_i has participated in a crossover and the number of times p_i produced a fitter offspring. The parental success ratio is the ratio of these two numbers.

3.6. *Reproduction and mutation*

Reproduction is performed amongst a number of fittest individuals in the population. To generate new offsprings using our proposed crossover, we randomly chose two individuals among the fittest ones in the population as the parents. We then pass them to the PSC. Also, every I iterations, we pick the parents totally random as a means to preserve the diversity in the population.

The goal of the mutation operator is to reduce the number of conflicts. Therefore, we perform the mutation as follows. Given a constraint graph representing a CSP, N_{mutation} random vertices (corresponding to the CSP variables) of the individual are selected and the numbers of conflicts between the chosen vertices and their adjacent vertices are minimized. This can be done by picking, for each variable X, a value that minimizes the number if conflicts X has with its neighbors. This can of course lead to a local minimum. In order to preserve the diversity of the population and to avoid being trapped in this local minimum, from time to time we also use a random value change for N_{mutation} vertices.

3.7. *The GM operator*

The GM operator runs concurrently with the PGA to generate good individuals outside the scope of the GA. The PGA then incorporates these newly generated, near optimal individuals to its population to give them a chance to participate in reproduction. Whenever the GM produces a population of individuals, the PGA keeps them until the next reproduction. Then, just before the reproduction, the PGA distributes them between the subpopulations. The GM uses a variable ordering for individual generation. Whenever the GM needs to create a new individual, it starts from the first variable in the ordering and generates a random value for each variable in turn. Following a look ahead principle, when a variable is assigned a value, the GM propagates this change by removing the values that would result in conflicts from the domain of its neighbors. This way, it is guaranteed that at each time, the chosen value for a variable will not cause a conflict. However, at the end of initializing variables, we might end up with some variables that have empty domains. In this case, the GM randomly chooses a value for them. The GM generates P_{GM} individuals and signals the PGA's master process. The master process will then distribute the generated individuals amongst the subpopulation for the next reproduction.

Algorithm 1 MSPGA Integrated with HC

```
 1: generationNumber= 0
 2: In Parallel: generate a random population of size $P$.
 3: Calculate the fitness of each individual.
 4: if a consistent solution is found or generationNumber = maxValue then
 5:     signal the CP and wait for a task from the CP.
 6: else, go to the next step.
 7: end if
 8: if (generationNumber mod $k = 0$) then
 9:     Pick the best solution found by slaves.
10:     Calculate new weights for constraints (based on HC).
11:     Create a new ordering based on new weights.
12:     Pass the new ordering to GM.
13: end if
14: Before entering the reproduction, check if the GM process has created a modified
    population. If so, distribute them amongst sub populations.
15: In Parallel: perform reproduction, mutation, and fitness calculation.
16: Increment generationNumber by 1. Go to step 3.
```

We used the heuristic proposed in Ref. 18 for variable ordering. This heuristic is based on HC for weighing constraints and works as a SVO algorithm as it runs prior to the actual backtrack search algorithm. More precisely, HC is run for a given number of cycles, during which, the constraints gain weight. After this information gathering phase, each variable gets a weighted degree, which is the sum of the weights of the constraints that the variable is involved in. Variables are then sorted based on their weights and those with larger weight get more priority in the ordering. We converted this heuristic into a DVO one by adapting it into each of the PGAs' master process. The idea is that HC will operate as part of each PGA and continues to run through the whole runtime of the PGAs. At the end of each k generations, the master process picks the best solution found by the slaves and calculates new weights for the constraints. It then creates an ordering based on the weights and passes it to the GM. The GM would then use this new ordering to create new individuals. The integration of our DVO into PGA's master process is described in the next section (see Algorithm 1).

3.8. *Hierarchical PGA (HPGA)*

The known advantage of PGAs is their ability to evolve in diverse directions simultaneously.[32–34]

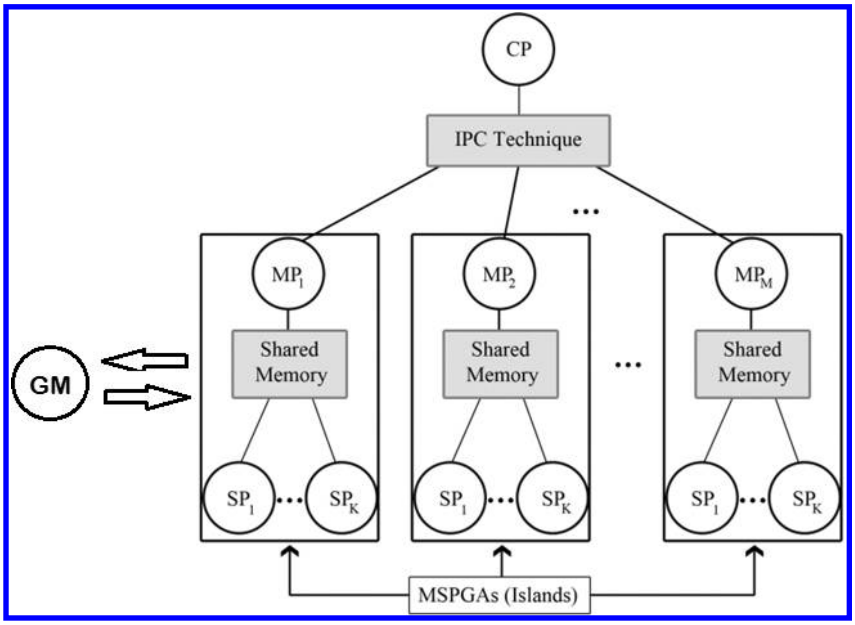


Fig. 3. Architecture of the HPGA.

It has been shown that PGAs speed up the search process when tackling hard problems.[35,36] Our proposed GA-based method is integrated into a parallel architecture, called HPGA, we have proposed for GCP.[19]

Figure 3[19] illustrates the HPGA architecture where M islands (that we call IPGAs) are coordinated through a Coordinator Process (CP) and executed in parallel looking for a solution to a given CSP instance. These IPGAs communicate using a given Inter-Process Communication (IPC) technique. Each IPGA is a MSPGA involving a set of K GAs running in parallel and communicating through a shared memory. The MP_i and SP_i stand respectively for Master and Slave processes. Algorithm 1 illustrates the MSPGA procedure.

4. Experimentation

4.1. *Experimentation environment*

All the algorithms are implemented in Java programming language. We used a machine with 2.5 GHz Core 2 Duo CPU, 4 GB of RAM running JDK 1.6. The mutation is implemented as described in Sec. 3.6. The number of variables to change (N_{mutation}) is also determined randomly from [2, individualLength/10] where individualLength is the length of each individual. For all the tests reported in this section, each problem instance is solved 20 times by the given method and the average running time needed to return the solution is computed together with the standard deviation. Table 1 lists the operator configuration used for PSC and OPC.

4.2. *Problem instances*

Following the model RB,[16] we generate each CSP instance in two steps as shown below and using the parameters n, p, α and r where:

– n is the number of variables,
– p $(0 < p < 1)$ is the constraint tightness which can be measured, as shown in Ref. 37, as the fraction of all possible pairs of values from the domain of two variables that are not allowed by the constraint,
– and r and α $(0 < \alpha < 1)$ are two positive constants (respectively set to 0.5 and 0.8).

(1) Select with repetition $rn \ln n$ random constraints. Each random constraint is formed by selecting without repetition 2 of n variables.

Table 1. PSC and OPC configurations.

Operator	PSC	OPC
Selection	Truncation	Truncation
Mutation	Random	Random
Crossover	Parental Success	One Point
Replacement	Worst Individuals	Worst Individuals

(2) For each constraint we uniformly select without repetition pd^2 incompatible pairs of values from the domains of the pair of variables involved by the constraint. $d = n^\alpha$ is the domain size of each variable.

4.3. *Evaluation of the performance of PSC*

First, we compared our proposed PSC against the OPC, the Asexual Crossover (ASXC) and the MPC proposed in Ref. 13. The population size is fixed here to 2000 and the mutation chance to 0.2. Tables 2 and 3 show the results of running these methods respectively without and with mutation. Consistent CSP instances are randomly generated with 100 variables and tightness p-values ranging from 0.05 to 0.6. For each test we report the running time (in seconds) together with the Success

Table 2. Comparing different crossovers in a sequential GA (no mutation).

	ASXC	MPC		OPC		PSC	
T	SR,BF	SR,BF	Time	SR,BF	Time	SR,BF	Time
0.05	0.2	100%,0	2.98	100%,0	3.32	100%,0	1.97
0.1	0.8	21%,0	55.72	100%,0	13.18	100%,0	2.18
0.15	0.16	0.4	—	86%,0	21.72	100%,0	2.75
0.2	0.24	0.14	—	23%,0	54.9	100%,0	5.78
0.25	0.38	0.19	—	0.5	—	100%,0	7.5
0.3	0.43	0.27	—	0.8	—	100%,0	7.98
0.35	0.54	0.37	—	0.16	—	100%,0	11.84
0.4	0.67	0.45	—	0.23	—	58%,0	46.77
0.45	0.79	0.58	—	0.29	—	0.5	—
0.5	0.91	0.71	—	0.33	—	0.14	—
0.55	0.103	0.79	—	0.47	—	0.17	—
0.6	0.114	0.86	—	0.65	—	0.24	—

Table 3. Comparing different crossovers in a sequential GA.

	MPC		OPC		PSC	
T	SR,BF	Time	SR,BF	Time	SR,BF	Time
0.05	100%,0	2.448	100%,0	2.57	100%,0	1.88
0.1	100%,0	57.13	100%,0	11.46	100%,0	2.11
0.15	0.4	—	100%,0	16.43	100%,0	3.18
0.2	0.9	—	27%,0	51.03	100%,0	4.1
0.25	0.17	—	0.3	—	100%,0	6.92
0.3	0.25	—	0.5	—	100%,0	7.26
0.35	0.28	—	0.10	—	100%,0	13.86
0.4	0.37	—	0.12	—	62%,0	44.56
0.45	0.45	—	0.27	—	0.5	—
0.5	0.53	—	0.30	—	0.9	—
0.55	0.69	—	0.42	—	0.15	—
0.6	0.78	—	0.62	—	0.17	—

Rate, SR (for reaching a complete solution) and the best fitness, BF (number of violated constraints, in the case where a complete solution is not obtained). When the timeout (60 s) is reached for a particular case, "–" is displayed. Note that for the ASXC method, the time is not reported in Table 3 as this method fails to solve all the problem instances in the allocated runtime limit in the case where mutation is used. From the two tables it is obvious to see that our PSC-based GA method is the only one that is successful for solving under constrained and middle constrained problems (when the tightness is less than 0.4). Moreover, for hard instances (tightness between 0.4 and 0.6) our method returns better quality solutions (BF values) than all the other techniques.

Finally, we compared our proposed PSC to the best heuristic-based GA, called Sawing GA, for solving a particular case of CSPs called TCSPs.[12] The TCSPs instances have 80 variables each and are generated as described in Sec. 4.2. The Sawing GA[38] is one of the adaptive fitness algorithms where the search is guided by changing the way the fitness is computed so that individuals are evaluated based on some special characteristics they may have. The Sawing GA is based on the Sawing mechanism where the idea is that constraints that are not satisfied or variables causing constraint violations after a certain number of steps must be hard, thus must be given a high weight (penalty). The results of this comparison are listed in Table 4. Our proposed algorithm obviously outperforms OPC and Sawing GA both in terms of running time and SR.

4.4. *Evaluation of the performance of GM within the HPGA*

The parallelism provides the ability to investigate different regions of the search space simultaneously. In order to assess the performance of the integration of our PSC within the HPGA, we conducted more tests on consistent CSPs as follows.

As we mentioned in Secs. 3.1 and 3.8, our PGA is implemented using the Master-Slave architecture. Within the HPGA, two islands (IPGAs) are used with the number of slaves for each fixed to 10 and the population size per slave equal to 200.

Table 5 reports the results of the tests we conducted on the same instances used for the sequential methods. We evaluate the performance of two methods: our PSC within the HPGA (HPGA+PSC) and the PSC with the proposed GM within the HPGA (HPGA+GM+PSC). It is clear from Table 5 that the parallelization of our method allows us to successfully solve all the instances up to 0.5 tightness value

Table 4. PSC, OSC and sawing GA for consistent TCSPs.

	PSC			OPC			Sawing GA		
T	Success rate	Time (s)	BF	Success rate	Time (s)	BF	Success rate	Time (s)	BF
0.24	100%	0.959	0	100%	2.817	0	20%	50	0
0.16	100%	0.213	0	100%	0.595	0	90%	18	0
0.05	100%	0.118	0	100%	0.310	0	100%	3	0

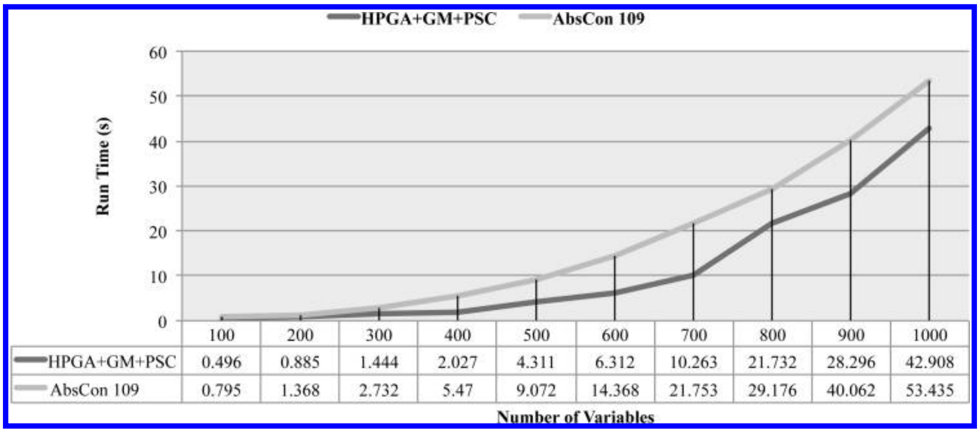
Table 5. Comparing different crossovers in a parallel GA.

HPGA+PSC		HPGA+GM+PSC	
(R,BF)	Time (s)	(SR,BF)	Time (s)
100%,0	0.31	100%,0	0.24
100%,0	0.42	100%,0	0.26
100%,0	0.77	100%,0	0.40
100%,0	0.97	100%,0	0.43
100%,0	1.09	100%,0	0.47
100%,0	1.14	100%,0	0.49
100%,0	1.28	100%,0	0.50
100%,0	1.54	100%,0	0.51
100%,0	1.72	100%,0	0.52
100%,0	2.03	100%,0	0.56
73%,0	37.46	100%,0	1.72
0.3	—	79%,0	31.29

without the GM and up to 0.55 when GM is used. In addition, with GM we can even solve the hardest instances (tightness of 0.6) with a high SR.

Figure 4 reports the results comparing our HPGA+GM+PSC method and Abscon 109[20] for solving randomly generated consistent CSPs with tightness equal to 0.35 and the number or variables varying from 100 to 1000.

In the case of HPGA+GM+PSC, we have used the following parameters tuned to their best values. For instances with less than 600 variables, the number of islands is 4 with 5 slaves per island and 100 individuals per slave. For instances with 600 and more variables, the number of islands is 8 with 5 slaves per island and 100 individuals per slave.



	100	200	300	400	500	600	700	800	900	1000
HPGA+GM+PSC	0.496	0.885	1.444	2.027	4.311	6.312	10.263	21.732	28.296	42.908
AbsCon 109	0.795	1.368	2.732	5.47	9.072	14.368	21.753	29.176	40.062	53.435

Fig. 4. Running time comparison of HPGA+GM versus AbsCon 109 on randomly generated instances.

From Fig. 4 it is clearly shown that our method is better than Abscon 109 for all problem instances in terms of running time needed to return a complete solution.

In addition to randomly generated instances, we conducted further comparative experiments on the following CSP instances from Lecoutre's library.[21]

- **Real-world instances:** `driverlogw` instances. This is a logistic planning problem where the goal is to move a subset of drivers, trucks and packages to given locations.
- **Instances with regular pattern:** `e0ddr1` instances corresponding to the Job-Shop problem.
- **Nonrandomly generated instances from the academia:** `queens` instances corresponding to the N-Queens problem.

The results of these experiments are reported in Fig. 5 and again show the superiority of our solver over AbsCon 109. These comparative results have been validated with the statistical analysis we conducted. In this regard, Table 6 lists the details of the results reported in Figs. 4 and 5 including the standard deviation.

Our GA-based solving method incorporates many greedy principles and has the ability to solve a CSP by giving a solution with a quality (number of solved constraints) depending on the time allocated for computation. This can be shown using the results of tests that we have performed with our HPGA+GM+PSC on a random consistent CSP instances of size 1000 and with tightness equal to 0.35. The "anytime curve" reporting the results is presented in Fig. 6. This curve is based on the number of constraint violations found after each period of time. More precisely, the y-axis corresponds to the number of constraints violations (fitness) while the x-axis shows the different time points in seconds. A complete solution (corresponding to 0 constraint violations) is obtained at 48.96 s.

Finally, Fig. 7 shows the "anytime curve" corresponding to tests conducted on a random inconsistent CSP of size 1000. As we can see from the figure, it took 315 seconds for the parallel method to reach a quality of the solution equal to 200 constraint violations.

The anytime behavior reported in Figs. 6 and 7 is very relevant in practice especially for real-time applications where a solution needs to be returned within a given deadline. The user can, for instance, get a solution with a given quality at a particular time point or let the program run for another amount of time, if this can be afforded, to get a better quality.

Similar tests have been conducted on TCSP instances and the results are reported respectively in Fig. 8 (for consistent TCSPs) and Fig. 9 (for over constrained TCSPs). Consistent TCSPs have tightness set to 0.35 while inconsistent TCSPs have tightness equal to 0.65.

The number of variables is ranging from 100 to 1000 (values on the x-axis of the figure). Note that with the tightness fixed to 0.35 these are hard to solve problems

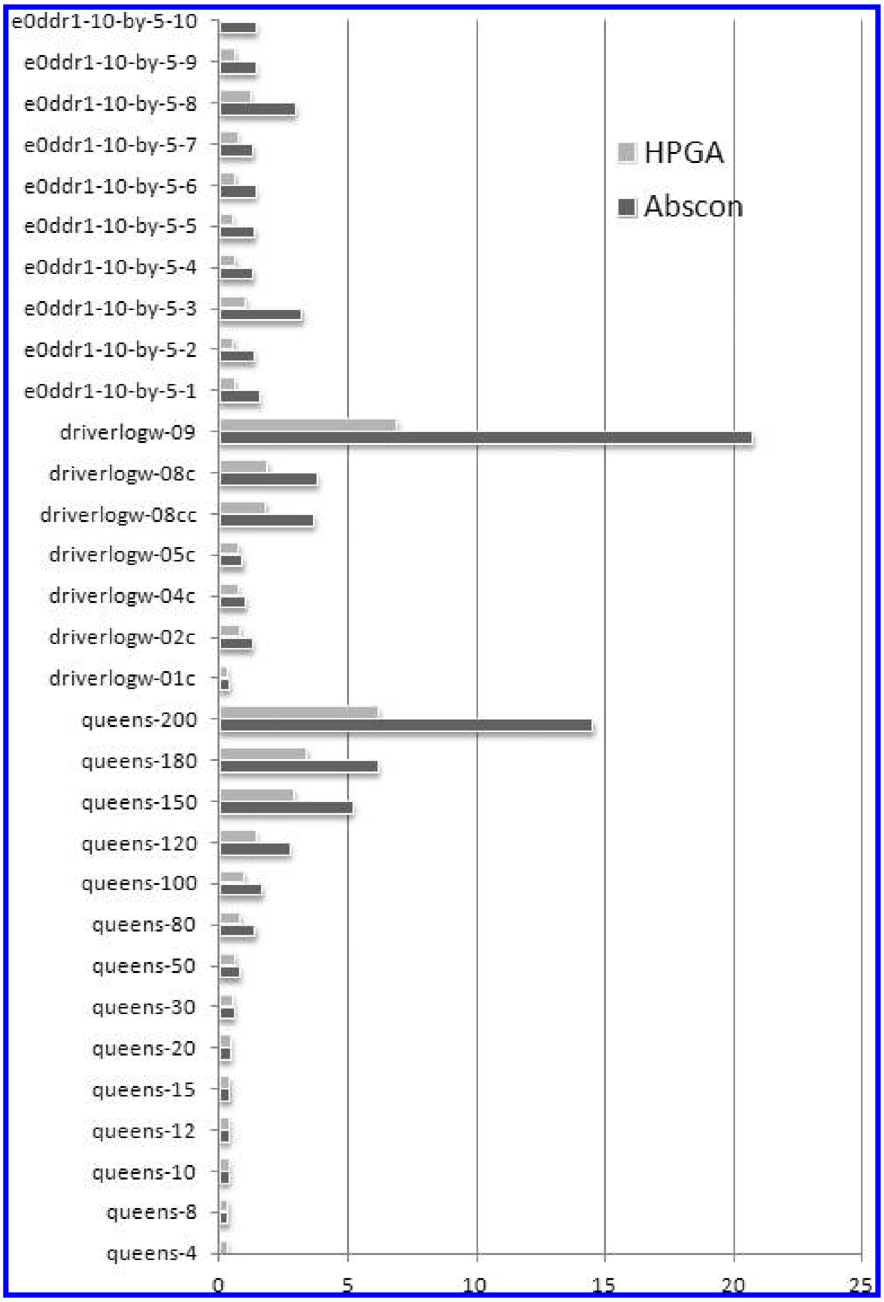


Fig. 5. Running time comparison of HPGA+GM versus AbsCon 109 on instances from Lecoutre's CSP library.

Table 6. Running time comparison of HPGA+GM versus AbsCon 109 with standard deviation.

Problem instance	Abscon	HPGA (avg)	HGPA (σ)
queens-4	0.30	0.30	0.00
queens-8	0.33	0.34	0.00
queens-10	0.36	0.35	0.00
queens-12	0.37	0.37	0.00
queens-15	0.41	0.43	0.00
queens-20	0.43	0.42	0.00
queens-30	0.56	0.51	0.00
queens-50	0.78	0.63	0.00
queens-80	1.38	0.88	0.00
queens-100	1.64	0.95	0.04
queens-120	2.73	1.40	0.05
queens-150	5.18	2.90	0.10
queens-180	6.16	3.40	0.10
queens-200	14.51	6.18	0.34
driverlogw-01c	0.37	0.33	0.00
driverlogw-02c	1.26	0.86	0.02
driverlogw-04c	1.01	0.77	0.02
driverlogw-05c	0.90	0.75	0.02
driverlogw-08cc	3.66	1.84	0.10
driverlogw-08c	3.80	1.88	0.32
driverlogw-09	20.71	6.95	0.56
e0ddr1-10-by-5-1	1.54	0.66	0.05
e0ddr1-10-by-5-2	1.34	0.55	0.05
e0ddr1-10-by-5-3	3.21	1.01	0.23
e0ddr1-10-by-5-4	1.30	0.63	0.02
e0ddr1-10-by-5-5	1.36	0.52	0.03
e0ddr1-10-by-5-6	1.40	0.61	0.05
e0ddr1-10-by-5-7	1.30	0.73	0.05
e0ddr1-10-by-5-8	2.95	1.22	0.08
e0ddr1-10-by-5-9	1.40	0.63	0.00
e0ddr1-10-by-5-10	1.41	0.74	0.02
RB-100	0.80	0.50	0.00
RB-200	1.37	0.89	0.02
RB-300	2.73	1.44	0.38
RB-400	5.47	2.03	0.63
RB-500	9.07	4.31	0.61
RB-600	14.37	6.31	0.77
RB-700	21.75	10.26	1.12
RB-800	29.18	21.73	0.92
RB-900	40.06	28.30	1.35
RB-1000	53.44	42.91	1.33

especially for large size variables. Thanks to the PSC and parallelism, the algorithm is able to solve completely all the instances in a reasonable time. For consistent TCSPs with 1000 variables for example, a complete solution is returned in less than a minute which is very impressive.

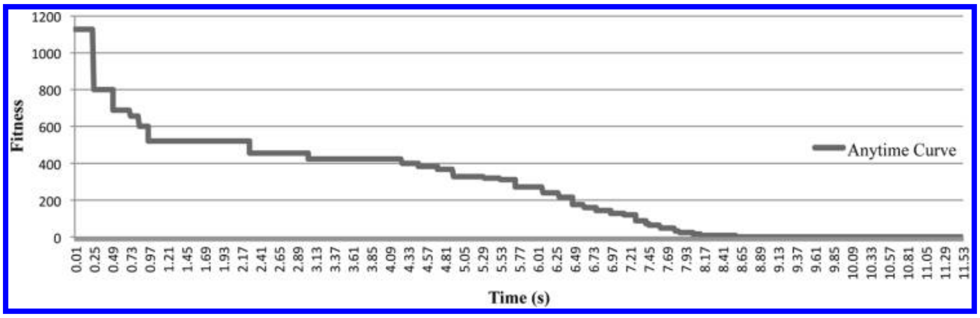


Fig. 6. Anytime curve for consistent CSPs with 1000 variables.

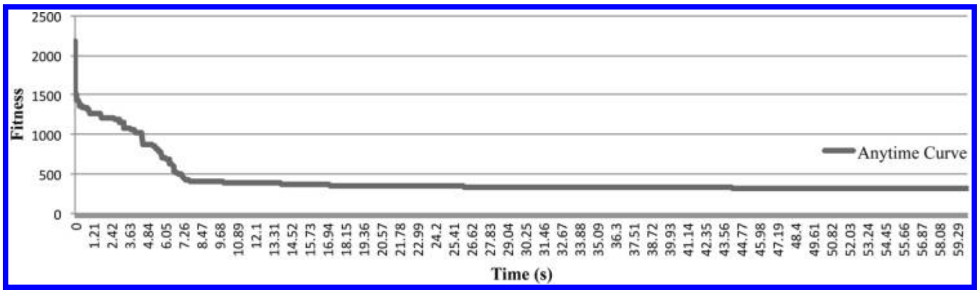


Fig. 7. Anytime curve for inconsistent CSPs with 1000 variables.

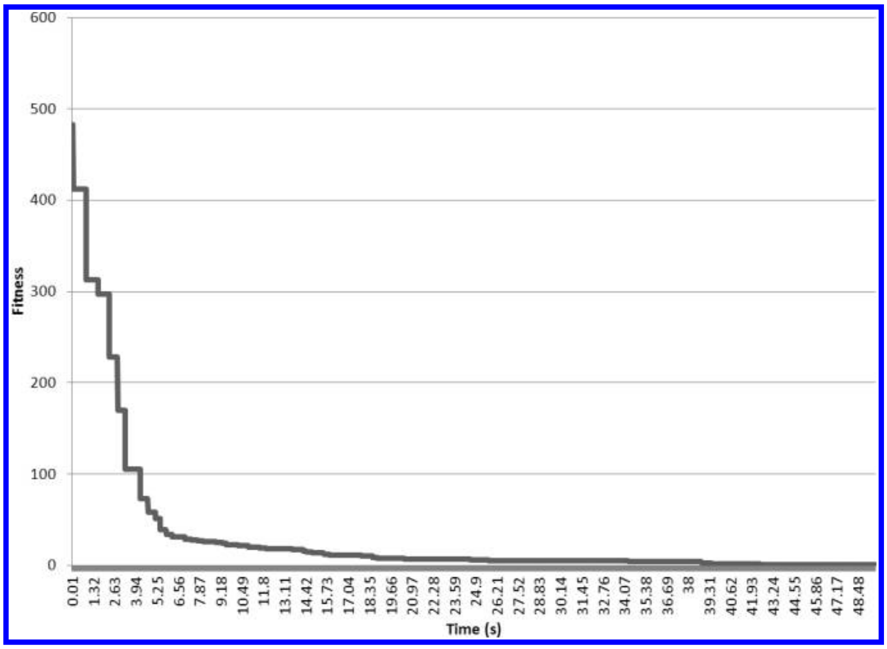


Fig. 8. Anytime curve for a consistent TCSP with 1000 variables.

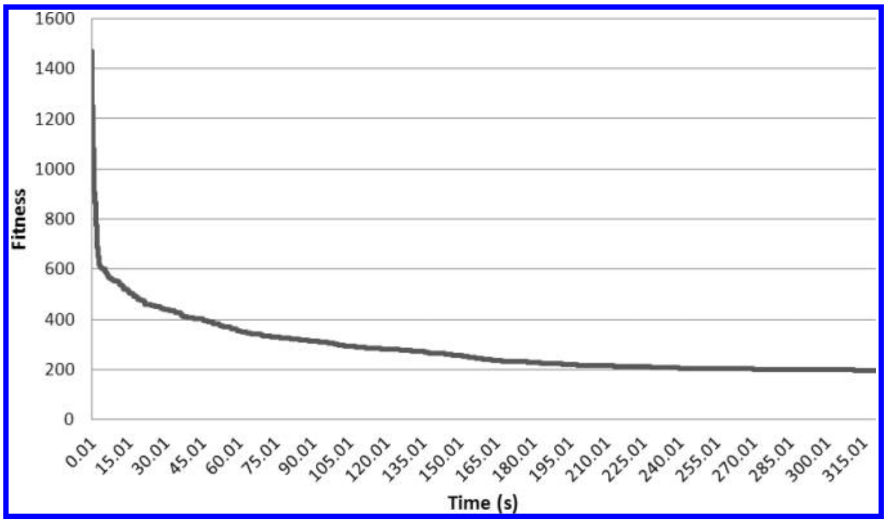


Fig. 9. Anytime curve for an inconsistent TCSP with 1000 variables.

5. Conclusion and Future Work

In order to overcome the difficulty when solving CSPs, we have proposed in this paper a novel crossover operator namely the PSC and its integration within a Hierarchical PGA (HPGA). Through different experiments on randomly generated CSPs as well as TCSP instances, we demonstrated that our proposed crossover is superior than the well-known crossovers, both in terms of time efficiency as well as the quality of the returned solution. In addition, a comparison of our HPGA to the well-known Abscon solver on random as well as real-world instances, show the superiority of our method in terms of running time.

These promising results motivated us to follow this work further into exploring variants of CSPs such as dynamic CSPs,[8,39] probablistic CSPs[40] as well as CSPs under preferences and change.[41,42] These latter problems are optimization problems where the goal is to come up with a solution maximizing some objective functions. In this case, the use of weighted CSPs[43] will be the option we are going to explore.

We also intend to extend our solving system to nonbinary CSPs. This can be done by converting the nonbinary CSP into a binary one and then use our solving approach on the latter. Nonbinary CSPs can be converted into binary ones using dual encoding or hidden variables encoding methods.[2]

References

1. R. Dechter, *Constraint Processing* (Morgan Kaufmann, 2003).
2. F. Bacchus and P. van Beek, On the conversion between non-binary and binary constraint satisfaction problems, *AAAI/IAAI* (1998), pp. 310–318.
3. A. Ben Hassine, T. Ho and T. Ito, Meetings scheduling solver enhancement with local consistency reinforcement, *Appl. Intell.* **24**(2) (2006) 143–154.

4. P. W. Yaner and A. K. Goel, Visual analogy: Viewing analogical retrieval and mapping as constraint satisfaction problems, *Appl. Intell.* **25** (2006) 91–105.
5. A. Gnay and P. Yolum, Constraint satisfaction as a tool for modeling and checking feasibility of multiagent commitments, *Appl. Intell.* **39**(3) (2013) 1–21.
6. E. Santos, S. DeLoach and M. Cox, Achieving dynamic, multi-commander, multi-mission planning and execution, *Appl. Intell.* **25**(3) (2006) 335–357.
7. M. Alfonso and F. Barber, A mixed closure-csp method for solving scheduling problems, *Appl. Intell.* **21**(2) (2004) 173–193.
8. M. Mouhoub, Systematic versus nonsystematic techniques for solving temporal constraints in a dynamic environment, *AI Commun.* **17**(4) (2004) 201–211.
9. M. Mouhoub and A. Sukpan, Conditional and composite temporal CSPs, *Appl. Intell.* **36** (1) (2012) 90–107.
10. J. F. Allen, Maintaining knowledge about temporal intervals, *Commun. ACM* **26**(11) (1983) 832–843.
11. R. Dechter, I. Meiri and J. Pearl, Temporal constraint networks, *Artif. Intell.* **49** (1991) 61–95.
12. B. J. Jashmi and M. Mouhoub, Solving temporal constraint satisfaction problems with heuristic based evolutionary algorithms, in *Proc. 2008 20th IEEE Int. Conf. Tools with Artificial Intelligence — Volume 02* (IEEE Computer Society, Washington, DC, USA, 2008), pp. 525–529.
13. A. E. Eiben and J. K. Van Der Hauw, Adaptive penalties for evolutionary graph coloring, *Artificial Evolution*, Lecture Notes in Computer Science, Springer, Vol. 1363 (1998), pp. 95–106.
14. J. van der Hauw, Evaluating and improving steady state evolutionary algorithms on constraint satisfaction problems, Master's thesis, Leiden University (1996).
15. M. Cebrian and I. Dotu, Grasp-evolution for constraint satisfaction problems, in *Proc. 2006 Genetic and Evolutionary Computation Conf. (GECCO)*, Seattle, USA (2006), pp. 531–538.
16. K. Xu and W. Li, Exact phase transitions in random constraint satisfaction problems, *J. Artif. Intell. Res.* **12** (2000) 93–103.
17. B. Smith and M. Dyer, Locating the phase transition in binary constraint satisfaction problems, *Artif. Intell.* **81** (1996) 155–181.
18. M. Mouhoub and B. J. Jashmi, Heuristic techniques for variable and value ordering in CSPs, in *GECCO,* eds. N. Krasnogor and P. L. Lanzi (ACM, 2011), pp. 457–464.
19. R. Abbasian and M. Mouhoub, A hierarchical parallel genetic approach for the graph coloring problem, *Appl. Intell.* **39**(3) (2013) 510–528.
20. C. Lecoutre and S. Tabary, Abscon 109: A generic CSP solver, *2nd Int. Constraint Solver Competition, held with CP'06 (CSC'06)* (2008), pp. 55–63.
21. C. Lecoutre, https://www.cril.univ-artois.fr/∼lecoutre/benchmarks.html, March 21, 2016.
22. A. K. Mackworth, Consistency in networks of relations, *Artif. Intell.* **8** (1977) 99–118.
23. R. M. Haralick and G. L. Elliott, Increasing tree search efficiency for constraint satisfaction problems, *Artif. Intell.* **14** (1980) 263–313.
24. R. Dechter and I. Meiri, Experimental evaluation of preprocessing techniques in constraint satisfaction problems, in *Proc. Eleventh Int. Joint Conf. Artificial Intelligence*, Vol. 1 (1989), pp. 271–277.
25. R. M. Haralick and G. L. Elliott, Increasing tree search efficiency for constraint satisfaction problems, *Artif. Intell.* **14**(3) (1980) 263–313.
26. B. M. Smith and S. A. Grant, Trying harder to fail first, Research Report Series-University of Leeds School of Computer Studies LU SCS RR (1997).

27. P. Refalo, Impact-based search strategies for constraint programming, in *Proc. Int. Conf. Principles and Practice of Constraint Programming (CP 2004)* (2004), pp. 557–571.
28. F. Boussemart, F. Hemery, C. Lecoutre and L. Sais, Boosting systematic search by weighting constraints, in *Proc. 16th European Conf. Artificial Intelligence (ECAI 2004)*, Valencia, Spain (IOS Press, 2004), pp. 146–150.
29. D. Grimes and R. J. Wallace, Learning to identify global bottlenecks in constraint satisfaction search, *20th Int. FLAIRS Conf.* (2007), pp. 592–597, Key West.
30. T. Balafoutis and K. Stergiou, Experimental evaluation of modern variable selection strategies in constraint satisfaction problems, in *Proc. 15th RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, Online at CEUR Workshop Proc.*, ISSN 1613-0073, pp. 1–15, Udine, Vol. 451 (2008).
31. M. Dorigo and T. Stutzle, *Ant Colony Optimization* (The MIT Press, 2004).
32. E. Cantu-Paz, *Efficient and Accurate Parallel Genetic Algorithms* (Kluwer Academic Publishers, 2000).
33. D. Lim, Y.-S. Ong, Y. Jin, B. Sendhoff and B.-S. Lee, Efficient hierarchical parallel genetic algorithms using grid computing, *Future Gener. Comput. Syst.* **23**(4) (2007) 658–670.
34. E. Alba and J. Troya, Influence of the migration policy in parallel distributed gas with structured and panmictic populations, *Appl. Intell.* **12**(3) (2000) 163–181.
35. S. A. Giuseppe, D. Megherbi and G. Isern, Implementation of a parallel genetic algorithm on a cluster of workstations: Traveling salesman problem, a case study, *Future Gener. Comput. Syst.* **17**(4) (2001) 477–488.
36. Z. Liu, A. Liu, C. Wang and Z. Niu, Evolving neural network using real coded genetic algorithm (GA) for multispectral image classification, *Future Gener. Comput. Syst.* **20**(7) (2004) 1119–1129.
37. D. Sabin and E. C. Freuder, Contradicting conventional wisdom in constraint satisfaction, in *Proc. Eleventh European Conf. Artificial Intelligence* (John Wiley and Sons, Amsterdam, The Netherlands, 1994), pp. 125–129.
38. B. Craenen and A. E. Eiben, Comparing evolutionary algorithms on binary constraint satisfaction problems, *IEEE Trans. Evol. Comput.* **7**(5) (2003) 424–444.
39. M. Mouhoub, Dynamic path consistency for interval-based temporal reasoning, *21st Int. Conf. Artificial Intelligence and Applications (AIA'2003)* (ACTA Press, 2003), pp. 393–398.
40. M. Mouhoub and J. Liu, Managing uncertain temporal relations using a probabilistic interval algebra, in *Proc. IEEE Int. Conf. Systems, Man and Cybernetics, Singapore*, 12–15 October 2008 (IEEE, 2008), pp. 3399–3404.
41. M. Mouhoub and A. Sukpan, Managing dynamic CSPs with preferences, *Appl. Intell.* **37** (3) (2012) 446–462.
42. M. Mouhoub and A. Sukpan, Managing temporal constraints with preferences, *Spatial Cogn. Comput.* **8**(1–2) (2008) 131–149.
43. J. E. Gallardo, C. Cotta and A. J. Fernandez, Solving weighted constraint satisfaction problems with memetic/exact hybrid algorithms, *J. Artif. Intell. Res.* **35** (2009) 535–555.