A Recursive Split, Solve and Join Strategy for Solving Constraint Satisfaction Problems

José Carlos Ortiz-Bayliss, Dulce Jaqueline Magaña-Lozano, Hugo Terashima-Marín and
Santiago Enrique Conant-Pablos
National School of Engineering and Sciences
Tecnológico de Monterrey
Monterrey, Mexico
{jcobayliss, terashima, sconant}@itesm.mx, dulce_magloz@yahoo.com.mx

***Abstract*—Constraint satisfaction is a recurrent problem found in both industrial and academic environments. The importance of this particular problem relies on the fact that many other problem domains can be represented as constraint satisfaction problems. The instances of this problem are usually difficult to solve due to their combinatorial nature, as they often require an exponential time in the number of variables. Various solving strategies have been proposed to tackle this problem in the past, being the two most important trends local search and backtracking-based methods. In this paper we propose a novel solving strategy that partitions constraint satisfaction instances into smaller ones that can be independently solved and later, uses their solutions to solve the original instance. This process is performed in a recursive fashion, including both a local search-based and a backtracking-based solver. Tests of the proposed approach on a set of benchmark instances taken from public repositories obtained encouraging results with respect to both local search and backtracking-based solvers applied in isolation.**

***Keywords*-Constraint satisfaction, Graph partitioning, Local search, Backtracking**

I. Introduction

The Constraint Satisfaction Problem (CSP) is among the most representative combinatorial problems in artificial intelligence and operations research because of its many practical applications [1], [2], [3]. A CSP instance is defined by a set of variables, each one with a set of available values, and a set of constraints among the variables. According to its complexity, CSP belong to the NP-Complete class of problems [4]. The solution space, and as a consequence, the time for finding a solution to a CSP instance grows exponentially with the number of variables in the instance. More specifically, in a CSP with n variables and m available values per variable, the solution space is given by m^n. The latter makes an exhaustive enumeration of all the potential solutions an infeasible strategy for most instances.

In general, CSPs are solved by using local search, backtracking-based methods or a combination of both. Local search methods produce and evaluate potential solutions by assigning values to all the variables at the same time, while backtracking-based algorithms systematically explore the solution space and iteratively construct a solution to the problem by assigning a value to one variable at a time.

In this paper we introduce the recursive split, solve and join (RSS&J) algorithm as an alternative to traditional local search and backtracking-based methods. The RSS&J algorithm is able to deal with both small and large CSPs from different classes. The method recursively partitions a CSP instance into smaller subinstances that are independently solved and later, the solutions found for those subinstances are joined and used for a last stage where a solution to the original instance is produced. Although other similar divide and conquer approaches for solving CSP have been proposed before, the model described in this investigation explores an interesting and novel way of splitting the instances and reusing their solutions to produce a solution to the original instance.

This paper is organized as follows. Section II presents the background and related works to this investigation. Section III introduces the solving strategy proposed in this work. Section IV describes the experiments conducted and the analysis of the results. Finally, Sect. V presents the conclusion and future work.

II. Background and Related Work

Graph partitioning deals with the task of dividing a given graph into smaller regions that have roughly the same number of nodes in such a way that the sum of edges between different regions is minimized [5], [6], [7]. Because binary CSPs can be represented as a constraint graph, they can be decomposed into smaller subproblems and then be solved in a semi-independent fashion.

Liu and Sycara described a methodology called constraint partition and coordinated reaction (C&CR) that partitions a CSP instance into different subinstances each of which concerns to a particular type of constraint [8]. An agent is assigned to each subinstance with the task to solve it by using a local search approach. Because one variable may be involved in more than one type of constraint, the algorithm requires to coordinate the local solvers on how to instantiate and revise the instantiation of the variables in order to satisfy its own constraints. The agents activate by turns, changing the value of a variable in conflict among the constraints within its own subinstance. When all constraints assigned to one agent are satisfied, the agent makes no further changes. A solution is found once all the agents have satisfied the constraints in their

respective subproblems.

Berlandier et al. [9] proposed a partition solving strategy that incorporates the concept of interface variables and interface constraints –variables that appear in more than one subinstance and the constraints between those variables. Their approach partitions a CSP based on its constraint graph into k subinstances of a similar complexity by using a partition algorithm proposed by Kerninghan and Lin [5]. Each of these subinstances is solved in parallel by an independent processor.

Although not directly related to CSPs, Qian et al. [10] proposed a two-step graph partitioning algorithm that discovers clusters in constrained graphs. The algorithm represents the graph as a sequence of nodes where nodes belonging to the same cluster are put together and then, makes the partition based on such sequence.

Pedamallu et al. [11], [12] described a cooperative solution methodology for solving the Feasible Sequential Quadratic Problem (FSQP). The model proposed by Pedamallu et al. uses a local search method based on interval partitioning to divide the domains and identify feasible solutions, resulting in a complete method that is able to find all the solutions to FSQP instances.

More recently, Magaña-Lozano et al. [13] explored two different heuristics to partition the Course Timetabling Problem (CTT) into smaller subinstances that could be independently solved. Once the subinstances are solved, the authors also studied the effect of two heuristics for joining the solutions to the individual subproblems.

III. The Recursive Split, Solve and Join Algorithm for Solving CSPs

The idea behind the RSS&J algorithm starts with the very same premise that has inspired other partitioning ideas on CSP: if the solution space of a given instance is given by m^n, where n is the number of variables and m the number of available values per variable, one strategy to better deal with the exponential growth of the solution space could be to split the instance into smaller subinstances that could be independently solved in a fast way and later, join their individual solutions to solve the original instance. From a general perspective, the RSS&J algorithm consists of three important stages:

1) **Split**. The original instance is split into two independent subinstances.
2) **Solve**. Each subinstance is independently solved. If the instance is already 'easy' to solve, it is solved as it is, otherwise the instance is solved by using a recursive call to the algorithm.
3) **Join**. The solutions from the two individual subinstances (in case they exist) are joined. The algorithm solves the original instance using the joined solution as starting point.

We will discuss in detail each one of these stages in the following lines.

A. Splitting the Instances

As we already mentioned in Sect. II, various strategies have been proposed in the past for partitioning CSP instances. Most of the existing strategies try to break the instances down into subinstances with similar complexities, but in our investigation this is not the case. To split the instances we use the degree of the variables. The degree of a variable is the number of edges connecting such variable. The variables in the CSP instance are sorted according to the degree (the variables with the smallest degrees are located at the first positions of the list and ties are randomly broken). Given a CSP instance p_0 that contains n variables, the splitting process will produce two subinstances (let us call them p_L and p_R, for left and right, respectively). p_L will contain the first αn variables in the ordered list while the remaining $(1-\alpha)n$ variables will be assigned to p_R. The splitting process guarantees that the produced subinstances have no variables in common. For this reason, all the constraints that connect variables from distinct subinstances are removed during the splitting process. Independently of the value of α (that determines the size of the resulting subinstances), subinstance p_L is expected to have a lower constraint density than subinstance p_R. In practice, we found that this splitting strategy makes p_L trivially solvable, as the number of constraints within this subinstance is almost always close to zero. On the other hand, p_R will always contain fewer constraints than the original instance but the highly constrained variables remain highly constrained after the partition process.

The RSS&J algorithm always splits instances into two subinstances per call. To split the problem into more than two subinstances the algorithm is invoked in a recursive way. Because the value of α is kept unchanged during the splitting process, with each recursive call the proportion of variables assigned to subsequent p_L and p_R remains the same but the actual number of variables with small degrees, decreases. Figure 1 depicts the result of the splitting process on the original instance and the subinstances produced at each recursive call. It is important to mention that the splitting process leaves the domains of the variables and the conflicts within the constraints unaltered.

When the RSS&J algorithm receives the original CSP instance p_0 with n variables, it splits it into two subinstances, p_{L_1} and p_{R_1}; each one containing αn and $(1-\alpha)n$ variables, respectively. Then, p_{R_1} is sent to the RSS&J algorithm to be split into two more subinstances. After the second call to the algorithm, three instances exist: p_{L_1}, p_{L_2} and p_{R_2}; each with αn, $\alpha(1-\alpha)n$ and $(1-\alpha)^2 n$ variables, respectively. The process is repeated until the maximum number of recursive calls, k, is reached. At that time, $k+1$ subinstances have been produced: p_{L_1}, p_{L_2}, p_{L_3}, ..., p_{L_k} and p_{R_k}; each one with αn, $\alpha(1-\alpha)n$, $\alpha(1-\alpha)^2 n$, ..., $\alpha(1-\alpha)^{k-1} n$ and $(1-\alpha)^k n$ variables, respectively.

A time limit is imposed to the algorithm to control its execution an avoid extremely long runs. When the running time exceeds the time limit provided, the system stops the execution and updates the exit code. There are three possible exit cases: (1) the instance was solved, (0) the solver exceeded the time limit and was forced to stop, and (-1) the system proved that the instance is unsatisfiable.

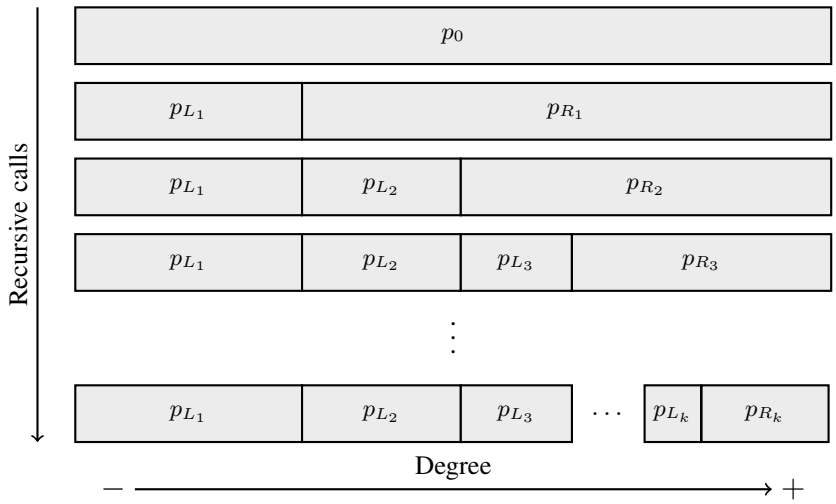

Fig. 1. The effect of the recursive calls during the splitting process.

B. Solving the Independent Subinstances

Once the subinstances have been produced, the algorithm solves them independently by using a backtracking-based solver. The solver is a constructive one, it iteratively constructs a solution by assigning a value to one variable at the time. From this point on we will refer to this solver as CONSTRUCTIVE. Because backtracking-based solvers are complete ones, CONSTRUCTIVE guarantees to find a solution if it exists and enough time is given to the solver. CONSTRUCTIVE selects the next variable to instantiate based on dynamic ordering heuristic. Although many of these heuristics exist (see for example: [14] and [15]), in this investigation we focused on WDEG [16] due to its capacity to 'learn' from failures as the exploration takes place to improve further decisions. WDEG assigns first the variable with the largest weighted degree. WDEG assigns a counter to each constraint, and every time such constraint is unsatisfied, the corresponding counter is increased by one. This is, the variable with the largest sum of weights over the constraints it participates in. In the case of ties, these are randomly broken. In addition to WDEG, the values of the selected variable are tried according to the min-conflicts heuristic (MINC) [17]. Once WDEG selects the variable to be assigned, the first value to be tried is the one most likely to success, the one that participates in the fewest conflicts (forbidden pairs of values between two variables). In case of ties, the minimum value is always preferred.

In contrast to other approaches, the subinstances produced by the RSS&J algorithm during the splitting process cannot be solved in parallel because of the recursive nature of the algorithm. More importantly, we found that because of the splitting process, subinstances p_{L_i} are almost surely trivial to solve. For this reason, the splitting strategy only produces one subinstance that requires computational resources to be solved: p_{R_k}, the right subinstance obtained from the deeper recursive call. Using parallelism to solve trivial instances would not be beneficial for the algorithm.

C. Joining the Solutions

Because the process of splitting the instance is forced to omit constraints between variables from the independent subinstances, joining the two solutions has a cost, as it has to satisfy the recently restored constraints that were discarded during the splitting process. In order to reuse the solutions found for the individual subinstances, we opted for a local search-based solver for the task of producing the solution to the original CSP instance. Because this solver uses an initial solution that is iteratively perturbed until a solution is found or the system is forced to stop, we will refer to it as PERTURBATIVE. PERTURBATIVE is based on the min-conflicts local search based solver for CSPs [17]. This solver produces a list of the variables in conflict and randomly chooses one of them to be assigned a different value. This process is repeated iteratively to gradually reduce the number of conflicts that exist between the variables, and eventually obtain a feasible solution. In order to improve the local search process we have also included a tabu list to avoid moving to recently visited states [18].

To produce a solution, the solutions obtained for the subinstances are joined in the inverse order that produced them. By following this idea, $p_{R_{k-1}}$ is solved by running PERTURBATIVE on the solution resulting from joining the individual solutions to p_{L_k} and p_{R_k}. Later, the solution to $p_{R_{k-2}}$ is produced by running PERTURBATIVE on the solution resulting from joining the solutions to $p_{L_{k-1}}$ and the recently found solution to $p_{R_{k-1}}$. The process is repeated until a solution to p_0 is obtained or the maximum running time is exceeded.

Now that all the elements of the RSS&J algorithm have been described, Algorithm 1 presents the pseudo code of the algorithm. The first call to `solveSS&J` takes as arguments the original CSP instance to solve, the desired value of α (the proportion of instances that are assigned to the left subinstances), 0 (the initial call to the algorithm) and the

desired number of recursive calls.

Algorithm 1 RSS&J Algorithm for solving CSPs.

```
1:  procedure SOLVERSS&J(p, α, i, k)
2:      if i == k then
3:          return SOLVECONSTRUCTIVE(p)
4:      else
5:          [p_L, p_R] ← SPLIT(p, α)
6:          s_L ← SOLVECONSTRUCTIVE(p_L)
7:          if s_L == [ ] then
8:              return [ ]
9:          end if
10:         s_R ← SOLVERSS&J(p_R, α, i + 1, k)
11:         if s_R == [ ] then
12:             return [ ]
13:         end if
14:         s ← JOIN(s_L, s_R)
15:         return SOLVEPERTURBATIVE(p, s)
16:     end if
17: end procedure
```

As the reader may have already observed, the performance of the RSS&J algorithm is affected by two parameters: the proportion of variables that are included in the left subinstance, α; and the number of recursive calls the algorithm will execute to produce the subinstances, k. For this investigation, the values of these parameters were chosen based on our experience with preliminary experiments. We observed that, in the case of α, good values are unlikely to exceed 20%. For deciding the number of iterations the number is usually small, at most 5. We are aware that automatically finding the most adequate values for these parameters is an important consideration to be addressed in the future.

IV. Experiments

We have used six sets of CSP instances taken from the public repository at http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html. The instances cover a wide range of instances as quasi-random, structured and real-world instances are present in the sets. Also, there are both satisfiable and unsatisfiable instances in the sets. All the instances in this investigation are binary constraints defined in extension and coded in XCSP 2.1 format.

Before using the RSS&J solver we conducted some preliminary experiments to define the most suitable values of α and the number of recursive calls. Adjusting these parameters takes a few minutes, as it is only necessary to run some values on one of the instances in each set and then, the same values are used for the rest of the instances.

Each instance analyzed in this investigation was solved five times and the median running time (in milliseconds) of those five runs was reported. To maintain the running time of the experiments between a reasonable time, the solvers are imposed a limit of 30 seconds per run on each instance. We consider this time limit fair as it serves to show which methods are currently suitable for some instances and which are not. In all cases, the performance of CONSTRUCTIVE, PERTURBATIVE and RSS&J are presented. The idea of the experiments is to explore the performance of the RSS&J algorithm when compared against the pure backtracking-based and local search solvers applied in isolation. The configuration of CONSTRUCTIVE and PERTURBATIVE is exactly the same used by RSS&J during the solving and joining stages (lines 6 and 15 in Algorithm 1).

To properly compare the results, we have included statistical tests on the means of the methods described for some particular sets of instances. In all cases, a paired t-test is used. The null hypothesis in the tests states that the means of the methods are equal, while the alternative hypothesis states the opposite.

A. Performance on Quasi-Random Instances

For this experiment we used two sets of quasi-random instances. The set COMPOSED25[1] is formed by 10 satisfiable instances with a main under-constrained fragment and some added constraints to smaller fragments. The instances in COMPOSED25 contain 105 variables with uniform domains of 10 values. The second set, GEOM[2] contains 100 kind-of-random instances in which a constraint between two variables exists only if the random generator attempts to create the constraint and the distance between the two variables involved is less than $\sqrt{2}$. The GEOM generation model assumes that random coordinates are assigned to the variables in order to estimate the distance. GEOM contains 100 instances with 50 variables each and a uniform domain size of 20. GEOM contains both satisfiable and unsatisfiable instances.

We tested the three solvers on these sets and we observed that, in the case of COMPOSED25, there is no point in splitting the instances at all, as the performance of CONSTRUCTIVE is exceptional. CONSTRUCTIVE required, in average, around 3 seconds per instance while RSS&J took around 8 seconds per instance. PERTURBATIVE was the worst solver in this case, requiring, in average, around 20 seconds per instance. The results obtained for GEOM, on the other hand, show some improvement of RSS&J over CONSTRUCTIVE (Fig. 2). Although the set was solved faster by using RSS&J with respect to any of the other two solvers, the statistical evidence suggests that the means of CONSTRUCTIVE and RSS&J are equal (p-value = 0.3445). Both CONSTRUCTIVE and RSS&J are statistically different from PERTURBATIVE for GEOM (p-values equal to 7.231e-13 and 2.2e-16, respectively).

B. Performance on Patterned Instances

We analyzed two sets of patterned instances: QCP10[3] and QCP15[4]. These sets contain Quasi-group Completion Problem (QCP) instances, where some of them are unsatisfiable. In QCP, the task is to determine whether the remaining entries of the partial latin square can be filled in such a way that we

[1] http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/composed-25-10-20.tgz

[2] http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/geom.tgz

[3] http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/QCP-10.tgz

[4] http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/QCP-15.tgz

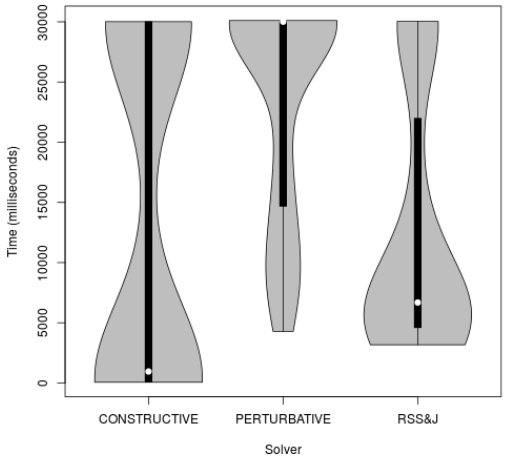

Fig. 2. Performance of CONSTRUCTIVE, PERTURBATIVE and RSS&J on GEOM.

obtain a complete latin square. The sets QCP10 and QCP15 contain instances with 100 and 225 variables, respectively; and domains of different sizes.

For these sets, RSS&J was clearly the best solver. In both sets the performance of CONSTRUCTIVE was rather poor. It was never able to finish the search before the time limit was reached. PERTURBATIVE was only competitive for QCP10 (Fig. 3), but when we moved to the larger instances contained in QCP15, its performance decreased significantly, being unable to finish the search within the time limit for all the instances. In both sets, RSS&J was able to solve all of the instances and prove which ones of them were unsatisfiable. In average, RSS&J required around $1/3$ of second per instance for QCP10 and around 8.2 seconds for the instances in QCP15.

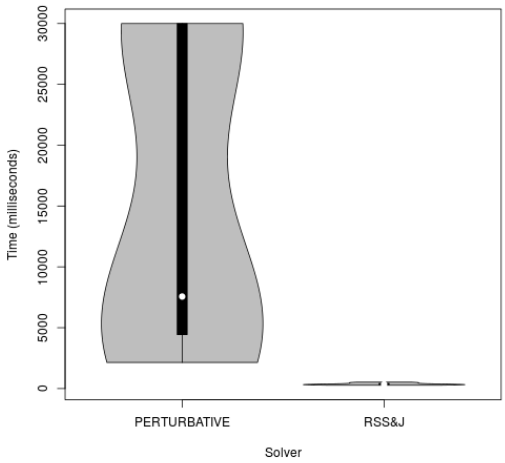

Fig. 3. Performance of PERTURBATIVE and RSS&J on QCP10 (CONSTRUCTIVE is left out of the comparison for clarity).

C. Performance on Real-World Instances

We also tested our approach on two sets of real instances. The set DRIVER[5] contains seven satisfiable real-world instances. The number of variables varies among the instances, ranging from 71 to 605. The values in the domains of the variables is also different among the instances.

Although the set DRIVER was solved slightly faster when CONSTRUCTIVE was used, the statistical evidence suggested that CONSTRUCTIVE and RSS&J were similar in performance (p-value = 0.2831). Also, both CONSTRUCTIVE and RSS&J were statistically different from PERTURBATIVE (p-values equal to 0.006864 and 0.006979, respectively). In this case, there was no benefit from using a partitioning strategy like the one described in this investigation with respect to the performance of CONSTRUCTIVE.

The second real set is called SCENS[6]. SCENS contains 11 Radio Link Frequency Assignment Problem (RLFAP) instances. In RLFAP the task consists of assigning frequencies to a number of radio links in such a way that it satisfies a large number of constraints and use as few distinct frequencies as possible. SCENS contains both satisfiable and unsatisfiable instances of distinct number of variables, ranging from 200 to 916 according to the instance.

The results depicted in Fig. 4 show that RSS&J was a competitive solver for the instances in SCENS. Because PERTURBATIVE was not able to produce a solution for any of the satisfiable instances in the set, it is removed from the figure for clarity. While CONSTRUCTIVE required, in average, 3.4 seconds to solve each instance in the set, RSS&J required only 0.86 seconds per instance. Although the difference in the running time between CONSTRUCTIVE and RSS&J is significant in practice, the statistical evidence suggests that RSS&J and CONSTRUCTIVE were not different (p-value = 0.001629).

D. Discussion

We have tested RSS&J on instances with diverse characteristics. Both CONSTRUCTIVE and PERTURBATIVE were also tested in isolation on the same instances. RSS&J obtained promising results, specially on unsatisfiable instances where it is able to identify small unsatisfiable subsets of variables at the first stages of the search, reducing the time to produce a failure. Because RSS&J internally uses a local search solver during the join stage, RSS&J is an incomplete solver and cannot guarantee that it will find a solution, even with large amounts of time. But, the fact that it contains also a backtracking-based solver that operates during the solve stage makes it possible to identify, in some cases, when the instances are unsatisfiable.

We also learned that not all the instances are suitable for a partitioning approach. There are cases where the instances can be easily solved with a traditional method without the need of partitioning the instance. One observation is that the fact that a CSP instance is suitable for partitioning depends not only

[5]http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/driver.tgz

[6]http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/rlfapScens.tgz

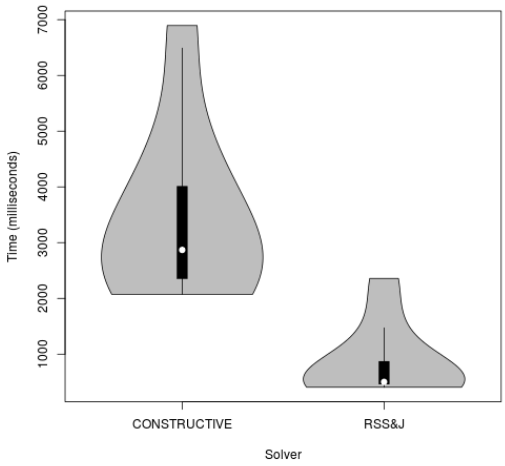

Fig. 4. Performance of CONSTRUCTIVE and RSS&J on SCENS (PERTURBATIVE is left out of the comparison for clarity).

on its size, but also the way constraints are distributed among the instance.

An important observation is that the reported times for the experiments consider the whole running time of any of the algorithms. In the case of CONSTRUCTIVE and PERTURBATIVE, there is no additional time to consider, as all the time they used is devoted to solve the instances. In the case of RSS&J, the solver needs to use time to actually produce the subinstances. We found that this is the most time consuming task of our algorithm, and the one that requires to be optimized in the future. Regarding the actual time for solving the instances, the task of joining the solutions of the subinstances and solving the instance that originated those subinstances takes most of the time the RSS&J algorithm uses. The backtracking-based solver is always fast, taking no more than 1 second to solve the largest p_{R_k} subinstances in all the experiments conducted in this investigation.

V. Conclusion

We have proposed a new partitioning algorithm for solving CSPs. The algorithm proposed, called RSS&J, recursively partitions a CSP instance until it is easy to solve and then, it goes through a joining process where the solutions of the subinstances are combined into a solution to the original instance. The algorithm was tested on various sets of instances achieving promising results.

In this investigation we only used WDEG as variable ordering heuristic for the backtracking-based solver. We are aware that the performance of the heuristics for variable ordering is sensitive to the instances under exploration and then, it may be the case that for some of the instances used in this investigation, WDEG may not be the best choice. We plan to incorporate other heuristics to the constructive solver and evaluate whether the performance of RSS&J is affected or not. We also know that more experimentation is needed to validate our results on new heuristics, as the results presented are only valid when the variables and the values are ordered with WDEG and MINC, respectively. A more complete analysis on the impact of the ordering heuristics used as part of the RSS&J algorithm is expected as part of the future work.

Choosing the right values of k and α is a critical part of the algorithm and should be done in an automatic fashion. We consider that the most important aspect of the future work is to incorporate a method to dynamically define the adequate number of recursive calls (k) and the most suitable value of α based to the instance features. In this way, these parameters would be automatically adjusted for each instance. We consider this might improve the performance of the RSS&J algorithm.

Acknowledgments

This research was supported in part by ITESM Research Group with Strategic Focus in Intelligent Systems and CONACyT Basic Science Project under grant 241461.

References

[1] P. Hell and J. Nesetril, "Colouring, constraint satisfaction, and complexity," *Computer Science Review*, vol. 2, no. 3, pp. 143–163, 2008.

[2] J. A. Berlier and J. M. McCollum, "A constraint satisfaction algorithm for microcontroller selection and pin assignment," in *Proceedings of the IEEE SoutheastCon 2010*, 2010, pp. 348–351.

[3] R. Barták and M. A. Salido, "Constraint satisfaction for planning and scheduling problems," *Constraints*, vol. 16, no. 3, pp. 223–227, 2011.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[5] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[6] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '95. New York, NY, USA: ACM, 1995.

[7] G. Karypis and V. Kumar, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *J. Parallel Distrib. Comput.*, vol. 48, no. 1, pp. 71–95, Jan. 1998.

[8] J. Liu and K. P. Sycara, "Distributed constraint satisfaction through constraint partition and coordinated reaction," in *proceedings of the 12th International Workshop on Distributed AI*, 1993.

[9] P. Berlandier and B. Neveu, "Problem partition and solvers coordination in distributed constraint satisfaction," in *Parallel Processing for Artificial Intelligence 3*, ser. Machine Intelligence and Pattern Recognition, H. K. James Geller and C. B. Suttner, Eds. North-Holland, 1997, vol. 20, pp. 165 – 178.

[10] Y. Qian, K. Zhang, and W. Lai, "Constraint-based graph clustering through node sequencing and partitioning," in *Advances in Knowledge Discovery and Data Mining*, ser. Lecture Notes in Computer Science, H. Dai, R. Srikant, and C. Zhang, Eds. Springer Berlin Heidelberg, 2004, vol. 3056, pp. 41–51.

[11] C. S. Pedamallu, L. Ozdamar, and M. Ceberio, "Efficient interval partitioning-local search collaboration for constraint satisfaction," *Comput. Oper. Res.*, vol. 35, no. 5, pp. 1412–1435, 2008.

[12] C. S. Pedamallu, A. Kumar, T. Csendes, and J. Posfai, "An interval partitioning algorithm for constraint satisfaction problems," *Int. J. of Modelling, Identification and Control*, vol. 14, no. 1/2, pp. 133–140, 2011.

[13] D. J. Magaña Lozano, S. E. Conant-Pablos, and H. Terashima-Marín, "Exploring the solution of course timetabling problems through heuristic segmentation," in *Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science, I. Batyrshin and M. González-Mendoza, Eds. Springer Berlin Heidelberg, 2013, vol. 7629, pp. 347–358.

[14] A. Arbelaez, Y. Hamadi, and M. Sebag, "Online heuristic selection in constraint programming," in *Symposiumon Combinatorial Search (SoCS)*, 2009.

[15] I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh, "An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem," in *Proceedings of CP-96*. Springer, 1996, pp. 179–193.

[16] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, "Boosting systematic search by weighting constraints," in *European Conference on Artificial Intelligence (ECAI'04)*, 2004, pp. 146–150.

[17] S. Minton, M. D. Johnston, A. Phillips, and P. Laird, "Minimizing conflicts: A heuristic repair method for CSP and scheduling problems," *Artificial Intelligence*, vol. 58, pp. 161–205, 1992.

[18] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Comput. Oper. Res.*, vol. 13, no. 5, pp. 533–549, 1986.