Applying Scheduling by Edge Reversal to Constraint Partitioning

Marluce Rodrigues Pereira[1] Patrícia Kayser Vargas[1 2]
Felipe M. G. França[1] Maria Clicia Stelling de Castro[1 3] Inês de Castro Dutra[1]

[1]COPPE - Engenharia de Sistemas e Computação – Universidade Federal do Rio de Janeiro – Rio de Janeiro, RJ, Brasil
[2]Curso de Ciência da Computação – Centro Universitário La Salle Canoas, RS, Brasil
[3]Instituto de Matemática e Estatística – Universidade Estadual do Rio de Janeiro – Rio de Janeiro, RJ, Brasil
{marluce,kayser,felipe,clicia,ines}@cos.ufrj.br

Abstract

Scheduling by Edge Reversal (SER) is a fully distributed scheduling mechanism based on the manipulation of acyclic orientations of a graph. This work uses SER to perform constraint partitioning of Constraint Satisfaction Problems (CSP). In order to apply the SER mechanism, the graph representing the constraints must receive an acyclic orientation. Since obtaining an optimal acyclic orientation is an NP-hard problem, this work studies three non-deterministic strategies known in the literature: Alg-Neigh, Alg-Edges, and Alg-Colour. We implemented the three algorithms and the SER scheduling mechanism, applying them to the CSP constraint networks generated from 3 applications. Our results show that SER has a great potential to perform a good partitioning of the constraint graphs.

1. Introduction

This work proposes the utilisation of the technique known as *Scheduling by Edge Reversal* (SER) [15, 7, 4, 5] to perform partitioning of Constraint Satisfaction Problems (CSP). These applications have characteristics that can benefit from local computations on subsets of constraints to reduce time and space complexity.

There are several methods in the literature to partition constraints [8, 11, 17, 18]. Their major objective is to speedup the execution in one processor by separating independent blocks of constraints, or to speedup execution through actual parallelisation, where each block of constraints can be computed independently by a different processor.

Most methods used in the literature to perform constraint partitioning either have a very high complexity or produce solutions far from optimal. Some of them perform manual partitioning. In our work we show that the use of SER can overcome these problems and produce automatically near-optimal solutions on Parallel Constraint Programming environments.

SER is a fully distributed scheduling mechanism based on the manipulation of acyclic orientations of a connected undirected graph G. Each node represents a process and two nodes are connected by an edge if and only if the corresponding processes share a resource. A resource is therefore represented in G by a clique, that is, a completely connected subgraph, since all processes that share the same resource will be connected. Note that a process can access an arbitrary number of resources.

At any time, a node is either idle or operating. An idle node waits for a resource. A node in operating state uses one or more of the resources it shares with other nodes. Thus G represents a system called *neighborhood constrained*, i.e, a system in which processes are constrained by their neighborhoods to operate.

In the SER synchronous distributed algorithm, the first step is to get an acyclic orientation of the graph G. Let the sinks in this orientation be the nodes that have no outgoing edge and that are in operating state. Since every acyclic orientation has at least one sink, there is no deadlock or starvation. At the next step, the edges incident to the sinks are reversed modifying this orientation and resulting in a new acyclic orientation, as well a new set of sinks. So, the scheduling can be regarded as the evolution in time of acyclic orientations of G and the schedule as an infinite sequence of orientations.

In a greedy schedule, a new acyclic orientation is obtained from the previous one by reversing all sinks. Under this assumption, orientations repeat themselves periodically from a certain time. Thus, period is the sequence of distinct orientations α_0,..., α_{p-1} such that $\alpha_k = g(\alpha_{(k-1)\ mod\ p})$, where $g(\alpha_{(k-1)\ mod\ p})$ is the orientation obtained from $\alpha_{(k-1)\ mod\ p}$ under greedy operation and p is the period's length. All nodes operate the same number m of times in a period. The concurrency of a period is defined by the coefficient m/p [7].

Several works used SER to solve scheduling problems in different contexts [12, 9, 13, 1, 6]. Most of these problems do not require a special initial acyclic orientation, because the graphs that represent the applications have an implicit orientation, i. e., they are directed graphs.

For undirected graphs, one simple way to obtain an acyclic orientation is to assign distinct, totally ordered identifications to all nodes and then orient the edges according to this total order. However, unless it can be assumed that nodes start out with these identifications, it may be necessary to resort to probabilistic techniques to make sure that they are distinct [7]. Therefore the concurrency provided by a deterministic acyclic orientation could be worse than a random acyclic orientation. In fact, this was the case for our applications. In order to apply SER we used three non deterministic algorithms to impose an initial acyclic orientation.

While most works in the literature use SER during execution to determine when an element must operate, our work uses the SER mechanism at compile time to decide how to perform constraint partitioning. The operational control is performed by the parallel CSP system.

Our results show that SER has a great potential to perform a good partitioning of constraint graphs.

The paper is organised as follows. Section 2 describes our representation of the constraint network in a graph. In Section 3 we describe the three acyclic orientation algorithms used in this work. Section 4 presents our applications describing their characteristics. Implementation details are presented in Section 5. The analysis of the impact of the acyclic orientation algorithms and of the SER mechanism are presented in Section 6. Finally, in Section 7, we conclude our work and present perspectives of future work.

2. Distributed constraints using Scheduling by Edge Reversal

Finite domain Constraint Satisfaction Problems (CSP) usually describe NP-complete search problems. Arc-consistency (AC) algorithms are an approach used to working locally on constraints and their related variables, pruning the search space in an efficient way [16]. Generally, arc-consistency algorithms work with constraint graphs where each node represents a variable and edges correspond to constraints relating two or more variables. Depending on the nature of the CSP being solved, these graphs will have different topologies, that can benefit from a distributed local execution. In fact, some works in the literature use some techniques to partition the constraint network by taking advantage of the degree of each node in the graph.

If there are some groups of nodes in the graph that are strongly connected, all these subgraphs can operate in a somewhat independent way from each other, allowing for a significant improvement in execution time. This happens because each subgraph computation does not need to propagate its values to the other subgraphs. Partitioning the constraint graph has another advantage: allows execution of subgraphs in parallel, in the presence of a multiprocessor or multicomputer machine. The identification of such independent subgraphs is an NP-hard problem. Therefore, most work in the literature shows either approximations to the practical problem [18, 17] or theoretical results [8, 11].

In a previous work [18, 19] we showed that a good partitioning of constraints can produce very good performance in both sequential and parallel implementations. We performed several experiments in both a shared memory and a distributed memory architecture, using different constraint partitioning methods, manually set.

In this work, we use an automatic static scheduling method, based on the SER algorithm, starting from three different kinds of acyclic orientation.

The first step in our modelling is the mapping of CSP problems to a graph representation. Any CSP problem can be represented using constraints that relate one or more variables. Each variable has an initial domain set by the user or by the CSP solver. The goal is to get a valuation for each variable, i.e., each variable will be assigned one single value that does not violate any of the constraints. There are different ways to build a graph for a CSP problem. The most popular way of building it is to represent the variables as nodes and the constraints between the variables as the edges. Because in the SER mechanism, each edge corresponds to a shared resource, we chose to build the graph in an alternative way, where each constraint is represented as a node, and edges as dependencies between constraints. So, if two nodes share a variable, there will be an edge connecting them.

Our second step is to build an undirected complementary graph from the original graph. Sinks in the same state, represent a clique in the original graph. Thus these sinks represent constraints that share one or more variables and must be kept in the same group. This approach of using the complementary graph was also used in França *et al* [14].

Once we have the graph, we need to impose an initial acyclic orientation on it. This is another important step, because the amount of concurrency achieved is highly dependent upon this orientation of G. However, finding the optimal amount of concurrency of G is an intractable problem. So, there are proposals to get a random acyclic orientation as will be discussed in Section 3.

After obtaining the initial acyclic orientation, we then can apply the SER algorithm, by reversing edges, until a period is found. Only the orientations belonging to the period are considered as a result, because all nodes operate the same number of times. The final step is the process mapping. The sink nodes in the same orientation are associated to the same process.

Next section discusses the different algorithms to obtain an acyclic orientation, used in this work.

3. Acyclic orientation algorithms

Determining an initial acyclic orientation of the graph that minimises dependencies among groups is NP-hard [7, 10]. However, there are several algorithms that can get a non-optimal initial orientation. In our experiments we studied the qualitative impact of three different algorithms to get non-optimal acyclic orientations: Alg-Neigh, Alg-Colour, and Alg-Edges [2, 3].

Alg-Neigh is an extension of Calabrese and França algorithm [9]. Alg-Neigh uses a dice with f faces to generate random values associated to each node. Initially, all nodes obtain a random value between 0 and f-1. Each node compares its value with all its neighbour's values. If it has the greatest value, all edges are oriented to itself and it becomes inactive, i.e, a deterministic node. All nodes that do not have all edges oriented continue active. These active nodes are probabilistic and they will participate in the next step. The described procedure repeats until all nodes become deterministic.

Alg-Colour is an extension of Alg-Neigh. It has two phases: colouring and orientation. In the first phase, it uses a dice with f faces to generate random values associated to each node. Initially, all nodes obtain a random value between 0 and f-1. Each node compares its value with all its neighbour's values. If it has the greatest value, it sets a colour represented by a non negative integer and it becomes a deterministic node. This colour must be the minimum value different from its neighbour's colours. All nodes that do not have a colour yet assigned continue active, and will participate in the next step. This phase continue until all nodes become deterministic. In the second phase each edge is oriented from the node with the greater colour to the node with the smaller colour.

Alg-Edges is an evolution of Alg-Neigh and Alg-Colour. This algorithm also uses a dice with f faces to generate random values associated to each node. Initially, all nodes obtain a random value between 0 and f-1. Each edge is oriented from the node with the greatest value to the node with the smallest value. When two neighbours draw the same value, they must get a new random value to orient this edge. The algorithm proceeds until all edges become oriented.

The three algorithms differ in their complexity to converge and quality of results. Alg-Colour and Alg-Neigh present a similar convergency ($O(n)$ [2]). However, Alg-Colour has a better quality orientation with respect to the application of the SER algorithm: it produces a short path and consequently the shortest period. Alg-Edges will cause SER to produce the longest period and thus is not suitable to our applications.

4. Applications

We used three applications to study the impact of both the acyclic orientation algorithms and the SER partitioning. These applications were studied before, but using other techniques to partition the constraints [20, 21, 18].

The first benchmark, *Arithmetic*, is a synthetic benchmark. It defines x blocks of arithmetic relations, $\{B_1, \ldots, B_x\}$, where each block contains y equations and inequations relating z variables. Blocks B_i and B_{i+1} are connected by an additional equation between a pair of variables, one from B_i and the other one from B_{i+1}. Coefficients were randomly generated. The goal is to find an integer solution vector. This kind of constraint programming is very much used for decomposition of large optimisation problems. Table 1 describes the main characteristics of the instances of the *Arithmetic* problem we used. The connectivity represents the percentage of connection between the nodes. Connectivity is the number of graph edges e divided by the number of edges in a corresponding fully connected graph with n nodes (connectivity = $\frac{e}{[n*(n-1)]/2}$). Note that all our results present connectivity related to the complementary graph.

Our second benchmark, *N-Queens*, consists of placing N queens in an NxN chessboard in such a way that no queen attacks each other in the same row, column and diagonal. The constraints are all inequations. Table 2 shows the number of constraints, number of edges, and the connectivity of each graph for each of the instances. We can observe that the number of constraints and edges tend to grow exponentially for this application.

Our third benchmark, the *Parametrisable Binary Constraint Satisfaction Problem* (PBCSP), is a synthetic benchmark. Instances of this problem are randomly generated given four parameters: number of variables (nv), the size of the initial domains (ds), density, and tightness. Density and tightness are defined as follows: $\frac{nc}{nv-1}$ and $1 - \frac{np}{ds^2}$, respectively, where nc is the number of constraints involving one variable (it is the same for all of them), and np is the number of pairs that satisfies the constraints. In our ex-

App.	Blocks	Equations / Inequations	Variables per block	Total of variables	Constraints	Edges	Connectivity
eq6	2	2	2	6	11	39	0.71
eq38	8	7	3	38	101	4,672	0.93
eq126	16	15	6	126	381	69,180	0.96
eq254	32	31	6	254	1,277	793,708	0.97
eq446	32	31	12	446	1,469	1,051,276	0.97

Table 1. *Arithmetic* Application Characteristics

N-Queens	Constraints	Edges	Connectivity
4	22	69	0.30
8	92	2,422	0.58
16	376	54,300	0.77
24	852	305,394	0.84
32	1,520	1,016,056	0.88
40	2,380	2,557,230	0.90
64	6,112	17,532,144	0.94

Table 2. *N-Queens* Application Characteristics

periments, the domain size (20) and tightness (0.85) are the same to all experiments. We used different sets of pairs of variable and density. Table 3 presents the number of constraints and edges, and the connectivity of each of these sets.

Vars.	Density	Constraints	Edges	Connectivity
	0.35	233	23,452	0.868
	0.55	359	55,158	0.858
25	0.65	409	71,436	0.856
	0.75	483	99,382	0.854
	0.35	914	386,929	0.927
	0.55	1,368	864,861	0.925
50	0.65	1,620	1,211,985	0.924
	0.75	1,864	1,603,765	0.924
	0.35	3,538	6,018,790	0.962
	0.55	5,530	14,695,230	0.961
100	0.65	6,532	20,499,574	0.961
	0.75	7,538	27,296,710	0.961

Table 3. *PBCSP* Application Characteristics

5. Implementation

Our implementation receives as input a file generated by a special Prolog program. This file contains the constraint network necessary to execute the constraint satisfaction application. The constraint network consists of a list of constraints, that relates several variables. The partitioning generated by the SER algorithm is presented to the user as a Prolog program, that consists of a database of Prolog facts. This format is suitable to be used as input to a Constraint Logic Programming Solver.

Our program has three main steps: (1)input data reading and graph generation, (2) graph orientation, and (3) partitioning using SER and writing output data.

In the first step, the input file is read, and: (a) each constraint is represented as a node of a vector element; (b) each node receives a sequential integer identification; (c) each node stores the variables expressed in the constraint.

An nxn matrix is created to represent the complementary constraint graph. It uses the data structure built from the input data with n being the number of constraints.

Then, the graph is oriented using one of the algorithms described in Section 3. The user can choose the orientation algorithm to be used.

After getting the initial orientation, the SER algorithm is executed as follows:

```
while a period is not found
 sinks are detected checking the matrix rows;
 sinks are stored in a history structure;
 sinks' incoming edges are reversed;
end-while
history is presented as the final result,
excluding the initial state that does not
belong to the period
```

The history obtained is presented to user as a set of Prolog facts into an output file.

Following, we present the results obtained executing the three applications using the described implementation.

6. Experimental data and analysis

Our experimental platform is a machine with a Pentium IV 1.8 GHz with 1 GByte of memory and the operating system Red Hat Linux 7.2. Each experiment was executed 20 times. The three orientation algorithms and the SER algorithm were executed for each application. Following we show the results obtained to *Arithmetic*, *N-Queens*, and *PBCSP*.

Arithmetic Table 4 shows the results of applying SER to the three orientations produced by Alg-Edges, Alg-Neigh and Alg-Colour, for the 5 instances of the application. For each algorithm, first column represents the average period size and the second column presents the standard deviation. We can observe that for Alg-Colour the standard deviation was 0.00 for all instances. Note that this result does not mean a deterministic partitioning. Each execution produced a different partitioning due to the random orientation.

Because it would be cumbersome to graphically show larger instances, we concentrate our detailed study on the

Instances	Orientation Algorithm					
	Alg-Edges		Alg-Neigh		Alg-Colour	
eq6	8.25	(1.04)	7.80	(1.29)	6.00	(0.00)
eq38	93.95	(2.96)	90.30	(3.20)	38.00	(0.00)
eq126	364.85	(3.69)	355.70	(4.27)	126.00	(0.00)
eq254	1,242.65	(5.23)	1,225.05	(6.23)	254.00	(0.00)
eq446	1,433.20	(5.70)	1,408.35	(6.94)	446.00	(0.00)

Table 4. *Arithmetic*: avg. size of period and std. deviation

smaller instance called eq6. The corresponding Constraint Logic Program associated to the instance eq6 is shown in Figure 1. This figure shows the equations of an instance of the *Arithmetic* application that consists of 2 blocks of equations with 2 equations per block, relating 2 variables. The total number of variables is 6, where we have 2 variables per block plus 2 variables that are the links between the blocks.

```
      main(5) :-
(0,1)    domain([Aa,Ab], 1, 8),          % 2 8
(2)      0+ 11*Aa + 13*Ab #= 126,
(3)      0+ 10*Aa + 14*Ab + VAb #= 139,
(4,5)    domain([VAb, VBa], 1, 8),       % 7 2
(6)      VAb + VBa #= 9,
(7,8)    domain([Ba,Bb], 1, 8),          % 3 4
(9)      0+  3*Ba + 12*Bb + VBa #= 59,
(10)     0+  4*Ba +  7*Bb #= 40.
```

Figure 1. *Arithmetic* application with 6 variables

The numbers after the percent symbol show the solutions for each pair of variables. Numbers that appear between parenthesis before the lines are the constraint identifiers. According to this program, one good allocation could be to place variables `Aa` and `Ab` in one process, and variables `Ba` and `Bb` in another process. Variables `VAb` and `VBa` could be allocated to either process. In other words, constraints (2) and (3) will be clustered in a group and (9) and (10) in another group, because this clustering will minimise the communication between the processes.

In our approach, when we represent the application as a graph, each node is a constraint and the edges are the common variables between the nodes.

As discussed in Section 2, we must generate an acyclic orientation to the complementary graph before executing the SER algorithm, in order to obtain a group allocation to the constraints. Each algorithm generated a different acyclic orientation, so they generated a different group allocation by the SER mechanism.

Figure 2 shows the execution of the SER algorithm over each generated orientation: Alg-Edges (a), Alg-Neigh (b), and Alg-Colour (c). In the graph, each node represents a constraint of the program shown in Figure 1. In this figure,

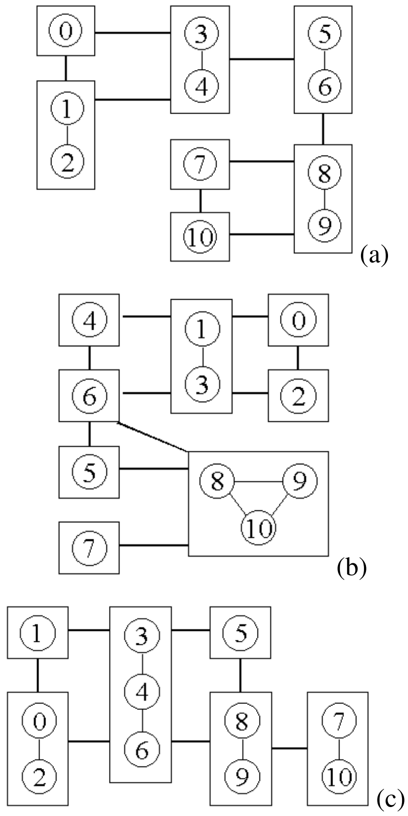

Figure 2. SER partitioning: graphs of *Arithmetic* oriented using each of the 3 algorithms

we have 11 nodes, 5 of which (2, 3, 6, 9, and 10) represent constraints relating 2 or more variables, and 6 that represent the constraints associated to the domain of each variable. We can observe that Alg-Colour clustered the nodes in a small number of groups, confirming results in the literature [2]. This clustering avoids too much communication among processors in parallel environments, and is convenient when we have a scarce number of resources.

The three partitionings generated by the algorithms are not optimal. However, they are very good approximations of a good partitioning for the program shown in Figure 1. One such good partitioning could be something like the one shown in Figure 3.

Because the SER algorithm works with an unlimited number of processes, periods can be very long. Therefore, we may need to rearrange the groups using an algorithm that minimises the number of edges among groups. SER can then be applied again to the new graph. Or one can use another algorithm that will have small complexity because of the smaller size of the newer graph.

For all other instances of *Arithmetic*, the smaller periods (or groups) were found with the Alg-Colour algorithm. It is

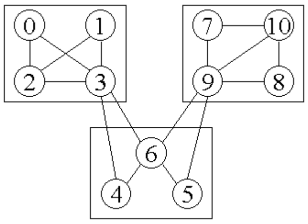

Figure 3. A good partition to the *Arithmetic* app. (eq6)

interesting to note that for this application, Alg-Colour always partitions the instances according to their number of variables.

N-Queens Figure 4 shows a Constraint Logic Program that describes the general N-queens problem.

```
main([N,L]) :-                   no_threat(X,Y,I) :-
  length(L,N),                     X #\= Y,
  domain(L,1,N),                   X + I #\= Y,
  constrain_all(L).                X #\= Y + I.
constrain_all([]).
constrain_all([X|Xs]) :-
  constrain_between(X,Xs,1),
  constrain_all(Xs).
constrain_between(_X,[],_N).
constrain_between(X,[Y|Ys],N) :-
  no_threat(X,Y,N),
  N1 is N + 1,
  constrain_between(X,Ys,N1).
```

Figure 4. *N-Queens* application with 4 variables

This program has 4 predicates. The `main/1` predicate starts to find a solution to the problem, according to the board size `N` given by the user. `L` will contain the solution. The predicates `constrain_all/1` and `constrain_between/2` do the search for possible queens locations, considering the constraints represented by the `no_threat/3` predicate. The first constraint avoids placement of queens in the same column. The second and third constraints avoid placement of queens in the same main or secondary diagonal. The smaller instance of this problem that has a solution is a board of size 4x4. This instance generates a constraint network with 22 constraints: 4 are related to the domains of the variables and the remaining 18 are related to the binary constraints, shown in the `no_threat/3` predicate. The variables are related in such a way that the constraint graph has a very high degree of connectivity, which makes it very difficult to do a manual partitioning.

Figure 5 shows the SER partitionings generated using the three orientation algorithms. Edges between constraints in different groups were omitted in this figure for readability purpose. Only one connection between related groups is presented. Once more, Alg-Colour (Figure 5(c)) generated an acyclic orientation that produced the best partitioning. The obtained partition has only 4 groups.

Notice that although Alg-Neigh (Figure 5(b)) and Alg-Edges (Figure 5(a)) produced an orientation that generated a greater number of groups, these can be re-partitioned to form a smaller number of groups. This re-partitioning can also be done for the Alg-Colour algorithm for greater instances of this problem. This is important whenever we have less processors than number of groups.

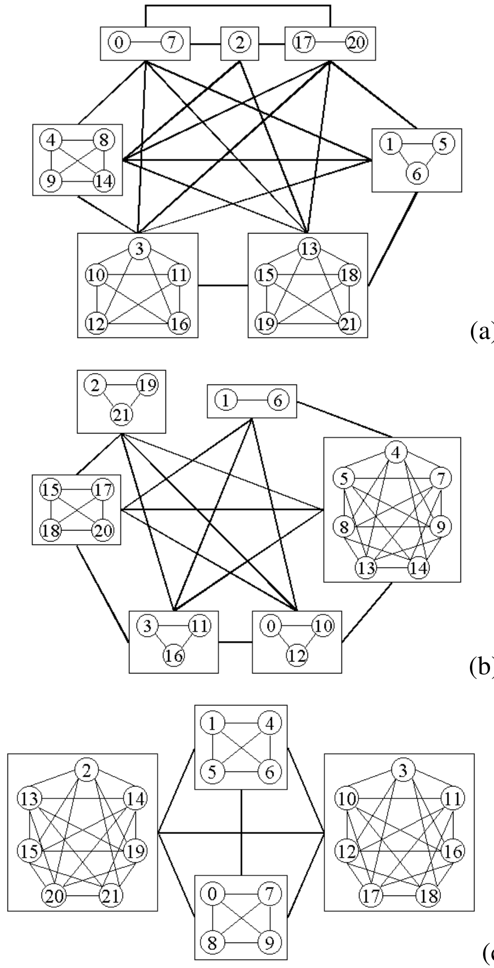

Figure 5. SER partitioning: graphs oriented using each of the 3 algorithms

Table 5 shows the average and the standard deviation for the period size found by SER using the orientations produced by Alg-Edges, Alg-Neigh and Alg-Colour for all instances of *N-Queens*.

	Orientation Algorithm					
	Alg-Edges		Alg-Neigh		Alg-Colour	
q4	8.9	(1.2)	7.5	(0.9)	4.9	(0.3)
q8	58.6	(4.1)	49.7	(2.1)	9.3	(0.8)
q16	294.2	(7.3)	268.2	(7.6)	17.8	(0.8)
q24	725.4	(8.4)	668.7	(8.5)	26.9	(1.2)
q32	1,347.6	(11.6)	1,277.2	(12.7)	34.8	(1.5)
q40	2,161.8	(11.5)	2,064.9	(14.5)	42.4	(1.4)
q64	5,755.6	(21.5)	5,582.2	(18.5)	67.9	(1.7)

Table 5. *N-Queens*: avg. size of period and std. deviation

We found out again that the Alg-Colour algorithm produced the smallest period (or number of groups). It presents period size with almost the same value as the number of variables of each instance. For our applications, we also noticed that Alg-Colour is the algorithm that gives more stable solutions with a very small variance in period size among the executions.

PBCSP Table 6 shows the average and the standard deviation for the period size found by SER using the orientations produced by Alg-Edges, Alg-Neigh and Alg-Colour.

		Orientation Algorithm		
Vars.	Density	Alg-Aresta	Alg-Viz	Alg-Color
25	0.35	202.60 (4.61)	194.00 (3.97)	25.80 (0.81)
	0.55	309.75 (6.14)	293.15 (5.76)	26.25 (1.13)
	0.65	354.10 (5.19)	333.35 (7.64)	26.55 (1.28)
	0.75	416.70 (5.17)	390.95 (5.47)	27.05 (1.36)
50	0.35	847.45 (7.05)	819.05 (9.35)	51.85 (1.31)
	0.55	1,266.30 (8.62)	1,225.90 (12.05)	51.70 (1.23)
	0.65	1,498.75 (6.91)	1,449.40 (9.26)	52.25 (1.30)
	0.75	1,726.10 (8.87)	1,670.00 (7.60)	52.55 (1.40)
100	0.35	3,406.11 (11.01)	3,350.25 (15.71)	101.40 (0.92)
	0.55	5,318.75 (15.09)	5,218.60 (6.97)	102.80 (1.17)
	0.65	6,280.75 (12.21)	6,172.00 (31.34)	103.20 (1.17)
	0.75	7,247.20 (14.25)	7,121.50 (12.82)	102.80 (0.98)

Table 6. *PBCSP*: avg. size of period and std. deviation

We can observe that once more Alg-Colour is the algorithm that produces the smallest period size with almost the same value as the number of variables of each instance.

It is important to notice that, except for the *Arithmetic* application, all constraint graphs have a strongly connected graph that make it difficult to use any kind of manual or deterministic algorithm to obtain a good partitioning. Therefore, we consider that SER was successful in producing good quality graph partitioning that can optimise the sequential and/or parallel execution of Constraint Satisfaction Problems.

7. Conclusions and future works

In this work we used a technique known as *Scheduling by Edge Reversal* (SER) to perform the partitioning of constraints to be used in a Parallel Constraint Programming environment. Constraint Satisfaction Problems (CSP) have characteristics that can benefit from local computations on subsets of constraints to reduce time and space complexity. Since SER works with graphs where the edges represent shared resources, we mapped the CSP problems to graphs where each edge represents shared variables between two constraints.

The results produced by SER are sensitive to an initial orientation of the graph. We then implemented three orientation algorithms to obtain this initial orientation: Alg-Edges, Alg-Neigh and Alg-Colour.

Alg-Edges is the algorithm that generates orientations in less time, but with period size greater than the others, as was shown in Section 6. Alg-Neigh generates orientations in less time than Alg-Edges with smaller periods. Alg-Colour is the algorithm that generates periods with smaller size and better concurrency.

In a previous work [18], we used two kinds of constraint partitioning to study their impact in the performance of a parallel constraint logic programming solver. This was a non-trivial task for all applications, because some of the graphs were strongly connected. This task became even more difficult as the problem size increased. The advantage of using an algorithm such as SER is that constraint partitioning is performed with good quality for any application independent of their sizes or graph characteristics, automatically.

One disadvantage of using SER is that because it works with an unlimited number of processes, periods can be very long. Therefore, we may need to rearrange the groups using an algorithm that minimises the number of edges among groups. SER can be applied again to the new graph or deterministic algorithms that minimise dependencies between nodes can be used with a small complexity because of the smaller size of the newer graph.

Overall, we consider that SER was successful in producing good quality graph partitioning that can optimise the sequential and/or parallel execution of Constraint Satisfaction Problems.

As future work, we intend to use the output of our implementation to confirm that the Constraint Satisfaction graphs partitioned by SER produce a good speedup either in a sequential or in a parallel constraint satisfaction solver. We also would like to investigate the impact of acyclic orientation algorithms that support weights associated to each node. This is important in the context of CSP problems, because each connection between two nodes can represent more than one variable being shared. In our current implementation, each edge in the graph supports only one kind of sharing without taking into account the number of shared variables.

Acknowledgements

This work is partially supported by the Brazilian Research Council CNPq, and by Capes.

References

[1] V. C. Alves, F. M. G. França, and E. P. Granja. A BIST scheme for asynchronous logic. In *7th. Asian Test Symposium*, pages 27–32, Singapore, Dec 02–04 1998. IEEE.

[2] G. M. Arantes Jr. *Orientações Acíclicas em Sistemas Distribuídos Anônimos e suas Aplicações no Compartilhamento de Recursos*. Dissertação de mestrado, COPPE/Sistemas – UFRJ, Rio de Janeiro, RJ, Brazil, 1999.

[3] G. M. Arantes Jr, F. M. G. França, and C. A. Martinhon. Algoritmos randômicos para a geração de orientações acíclicas em sistemas distribuídos. In *Anais do Simpósio Brasileiro de Pesquisa Operacional (XXXIV SBPO)*, RJ, Brazil, Nov 8-11 2002.

[4] V. C. Barbosa. *An Introduction to Distributed Algoritms*. The MIT Press, London, England, 1996.

[5] V. C. Barbosa. *An Atlas of Edge-Reversal Dynamics*. Chapman and Hall/CRC, 2000.

[6] V. C. Barbosa. The combinatorics of resource sharing. In R. Corrêa, I. Dutra, M. Fiallos, and F. Gomes, editors, *Models for Parallel and Distributed Computation: Theory, Algorithmic Techniques and Applications*. Kluwer Academic Publishers, 2002.

[7] V. C. Barbosa and E. Gafni. Concurrency in heavily loaded neighborhood-constrained systems. *ACM Transactions on Programming Languages and Systems*, 11(4):562–584, Oct 1989.

[8] C. Bliek, B. Neveu, and G. Trombettoni. Using graph decomposition for solving continuous csps. In M. J. Maher and J.-F. Puget, editors, *Principles and Practice of Constraint Programming, 4th Intl. Conf., Pisa, Italy*, volume 1520 of *Lecture Notes in Computer Science*, page 102. Springer, Oct 26-30 1998.

[9] A. Calabrese and F. M. G. França. Randomized distributed primer for updating control of anonymous ANNs. In *Proc. of the Intl. Conf. on Artificial Neural Networks (ICANN)*, Sorrento, Italy, May 26-29 1994.

[10] A. Calabrese and F. M. G. França. Distributed computing on neighbourhood constrained systems. In *Proc. of the 3rd Intl Conf. On Principles Of Distributed Systems*, pages 219–234, Hanoi, Vietnam, October 1999.

[11] M. Ferris and O. Mangasarian. Parallel constraint distribution. *SIAM Journal on Optimization*, 1:487–500, 1991.

[12] F. M. G. França, V. C. Alves, and E. P. Granja. Edge reversal-based asynchronous timing synthesis. In *Proc. of the IEEE Intl. Symp. on Circuits and Systems*, volume II, pages 45–48, Monterey, CA, USA, May-June 1998.

[13] F. M. G. França and L. Faria. Optimal mapping of neighbourhood-constrained systems. In *Parallel Algorithms for Irregularly Structured Problems, 2nd Intl. Workshop, IRREGULAR '95*, volume 980 of *Lecture Notes in Computer Science*, pages 165–170, Lyon, France, September 4-6 1995. Springer.

[14] F. M. G. França, J. A. M. Filho, and G. A. L. Paillard. Uma proposta de um escalonador para gamma. In *Anais do II Workshop em Sistemas Computacionais de Alto Desempenho (em conjunto com o SBAC-PAD'2001)*, pages 47–54, Pirenópolis, GO, setembro 2001.

[15] E. M. Gafni and D. P. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Transactions on Communications*, 1(29):11–18, January 1981.

[16] K. Marriot and P. J. Stuckey. *Programming with constraints: An Introduction*. MIT Press, 1998.

[17] T. Müller. Solving set partitioning problems with constraint programming. In *Proceedings of the 6th Intl. Conf. on the Practical Application of Prolog and the 4th Intl. Conf. on the Practical Applications of Constraint Technology (PAPPACT)*, pages 313–332, London, UK, Mar 1998. The Practical Application Company Ltd.

[18] M. R. Pereira, I. C. Dutra, and M. C. S. Castro. Arc-consistency algorithms on a software dsm platform. *Colloquium on Implementation of Cronstraint and Logic Programming Systems - CICLOPS 2001*, (NMSU-CSTR-003/2001):103–117, December 2001.

[19] M. R. Pereira, I. C. Dutra, and M. C. S. Castro. Parallelisation of arc-consistency algorithms. In *Jornadas Chilenas de Computación, 2002. VI Workshop on Distributed Systems and Parallelism*, Copiapó-Chile, Nov. 2002. Sociedad Chilena de Ciencia de la Computación.

[20] A. Ruiz-Andino, L. Araujo, and J. Ruz. Parallel solver for finite domain constraints. Technical Report SIP 71/98, Universidade Complutense de Madri, 1998.

[21] A. Ruiz-Andino, L. Araujo, F. Sáenz, and J. Ruz. Parallel execution models for constraint programming over finite domains. In G. Nadathur, editor, *Principles and Practice of Declarative Programming, Intl. Conf. PPDP, Paris, France*, volume 1702 of *Lecture Notes in Computer Science*, pages 134–151. Springer, Sep 29–Oct 1 1999.