Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 196 (2022) 159–166

Procedia Computer Science

www.elsevier.com/locate/procedia

CENTERIS - International Conference on ENTERprise Information Systems / ProjMAN - International Conference on Project MANagement / HCist - International Conference on Health and Social Care Information Systems and Technologies 2021

Asynchronous Backtracking variant applied to a rolls replacement process within a wire rod mill

Ana María Valdeón Junquera, Vicente Rodríguez Montequín*, Joaquín Manuel Villanueva Balsera, Javier García González

University of Oviedo, C/Idependencia 13,33004 Oviedo, Spain

Abstract

There is a wide number of application problems included in multi-agent systems and distributed artificial intelligence which can be described as the issue of finding a consistent combination satisfying all the existing inter-agent constraints, that is, they are distributed constraint satisfaction problems (DisCSPs). In this paper, a new technique for solving such kind of issues when they can be modelled as a succession of interrelated elements with constraints among an agent and its immediately following, having one of the agents the ability to notify either that the problem has been solved or that there are not any possible solutions, is presented. It has been developed using Asynchronous Backtracking, one of the most known algorithms for solving DisCSPs as a basis. To validate its applicability, it has been tested in a wire rod rolling mill for rolls replacement decision making process. It was implemented using the PADE multi-agent framework. The results demonstrate the method is valid to solve this kind of problems usually solved with recursive algorithms such as backtracking.

Keywords: Asynchronous Backtracking; Wire rod mill; DisCSP.

* Corresponding author. Tel.: +34-985104272; fax: +34-985104256.
E-mail address: montequi@uniovi.es

10.1016/j.procs.2021.12.001

1. Introduction

Distributed Artificial Intelligence (DAI) is a part of artificial intelligence which is concerned with interaction. It is composed by different combinatorial problems where numerous agents need to cooperate to ensure a consistent combination of actions. These problems are of a very varied nature from distributed scheduling [1] or planning [2] to distributed resource allocation [3] or distributed entities coordination [4,5]. In these problems agents have to achieve a suitable combination in a distributed way without centralization.

Asynchronous Backtracking (ABT) [6–8] is one of the most known algorithms for solving Constraint Satisfaction Problems in a distributed way [9]. It was introduced by Makoto Yokoo and it is an asynchronous algorithm in which all the agents execute concurrently and autonomously. ABT assumes a single static total priority order on the agents. Given an agent, those that appear before it in the order are higher priority and the ones after it are lower priority. Agents do not wait for the decisions of other agents, they perform assignments and send messages to lower priority neighbours informing them. The aim of each agent is finding an assignment that matches both its constraints and the ones from higher priority neighbours.

In the last years, the interest in solving distributed problems such as constraint satisfaction ones with the agents paradigm has increased in the AI community. This interest stems from naturally distributed constraint problems, for which gathering the full knowledge within a single agent is not suitable neither solving them with centralized algorithms. The reasons are diverse: collecting all data into an only agent could be challenging or perhaps not recommendable because of security or privacy considerations. However, no matter the problem is solved in a distributed way, it is possible that its nature possibilities that it can be modelled as a succession of interrelated elements existing constraints among an agent and its immediately following and in a way that requires that one of the agents has the ability to notify either that the problem has been solved or that there are not any possible solutions. In this paper, a specialisation of the ABT algorithm is proposed in which this ability has been introduced and its performance has been illustrated applying it to a real industrial problem, being able to be applied to other cases where the configuration is similar.

2. Distributed Constraint Satisfaction

A DisCSP (Distributed Constraint Satisfaction Problem) is a Constraint Programming problem, which is distributed in a MAS system among a set of autonomous agents, each owning its particular variables, which are related to those in other agents by its own constraints. To solve such kind of problems, most of the algorithms proposed in the literature, rely on message exchanging in order to find a solution that satisfies every constraint of each agent. The values assigned to each variable by the agent community must meet the constraints. To reach this goal, each agent assigns values to its variables which fulfil its own constraints and exchanges messages with the rest of the agents to satisfy those which involve other agents and revises the decisions with the feedback provided.

A DisCSP can be defined by a tuple (χ, D, C, A, φ), which are defined formally as follows:

$\chi = \{x_1, ..., x_n\}$, stands for the set of variables of the problem.
$D = \{D_1, ..., D_n\}$, stands for the finite domains of each of the variables.
$C = \{c_{12}, ..., c_{ij}\}$ where variables x_i and x_j have constraint c_{ij}.
$A = \{1, ..., p\}$, stands for a set of agents p.
$\phi = x \rightarrow A$ is a function that maps each variable to its agent.

Many distributed algorithms have been designed for solving DisCSPs, among them, the Asynchronous Backtracking (ABT) is the most remarkable.

3. Asynchronous Backtracking

Asynchronous Backtracking, introduced by Makoto Yokoo, is an asynchronous algorithm that consists on a set of agents which execute concurrently and autonomously. The agents of this set are in a single static total priority order, that is, for each agent the ones before it have a higher priority while those after it are lower priority. Agents act without

waiting for the other ones by performing assignments and sending information messages to lower priority neighbours on their local knowledge, the agent view. Each agent aims to find an assignment that does not violate neither its constraints or the ones from higher priority neighbours. If an agent detects an inconsistency, it determines the set of assignments which are in conflict (a no-good) and sends a message to the lowest priority agent in the conflict set so that it changes its current assignment.

The cycle of the algorithm is as follows:

1. At the beginning, each agent sets a value for its variable and informs the ones in its lower agent list of the value assigned.
2. When every agent has had its turn, the information messages arrive, and the agents process them.
3. The cycle continues until there are no messages left.

The exchanged messages are of different types:

- *Ok-message*: When an agent changes the value of a variable, it sends an *ok-message* to every lower priority agent connected to it with the new value assigned. The agents that receive an *ok-message*, add the new value to its agent view and check if the constraints are met.
- *Nogood-message*: An agent sends this type of message when there is no possible value from its domain that meets the constraints so the agent has to backtrack. In this case, it sends a *nogood-message* to the lowest priority agent from its list of higher priority agents and removes the agent and the values of its variables from the agent view. The *nogood* contains all the *nogoods* in its *nogood* list. The agent then backtracks and searches a value from its domain that meets the restrictions concerning the values in its agent view, now that a constraint has been removed. If there is no possible value, it backtracks again. When an agent receives a *nogood-message*, it checks if the message is consistent with its agent view. If it is, the *nogood* is added to its *nogood* list and tries to find another value that meets the constraints.
- *AddLink-message*: If an agent receives a *nogood* from an agent that has lower priority, it can contain values of agents that are not linked in a direct way to the agent that received the *nogood*. If the priority of these agents is higher than the agent that receives the *nogood* and this is consistent with its own agent view, it sends an *AddLink-message* to the higher priority agent and it is added to the higher agents list. When an agent receives an *AddLink-message*, it adds the agent to its list of lower agents and sends it an *ok-message*.

There is a set of independent algorithms related to ABT that were proposed in the past and can be embedded in a wider framework, given the large points they share and the slight differences among them. These differences are related to the communication links in the network of agents, that can be of three different types:

1. Permanent links: Every link added is maintained until the execution ends so two agents remain connected once a link is added.
2. Temporary links: Links added are maintained until a certain condition is met, when the link is removed letting the agents disconnected.
3. No links: The links remain the same during all the execution as there are not any new added.

These three options allow to divide the algorithms in the following categories:

- ABT_{all} [10]. The links are permanent and they are added during a preprocessing phase.
- ABT. This is the approach taken by the original ABT algorithm. Links are permanent and added dynamically when a *nogood-message* received contains non connected higher priority agents.
- ABT_{temp} [11]. The links between agents are added dynamically, as ABT, but they are temporary and only remain until a fixed number of messages is exchanged through them.
- ABT_{not} or DisDB [12]. There are not new links added so to achieve completeness, it removes obsolete information in finite time, forgetting all *nogoods* that probably could become obsolete when backtracking.

4. Proposed algorithm

To illustrate how the model works, it has been applied to a real-world problem, the replacement of the rolls of a wire rod rolling mill sections, the Reducing Sizing Mill (RSM). Wire rod consists on a hot rolled round section which is transformed into wire by means of cold rolling processes or wiredrawing. It is used as a raw material to produce tires, construction materials or screws among others. RSM is one of the several sections the rolling process has. It consists of four stands located at the finishing area which are in charge of providing the final quality to the product.

The rolls assembled on those stands work in pairs to provide the required cross-section profile to the raw material as it goes through. The quantity of rolls in the factory is very high though limited, usually greater than 4,000. The rolls are not equal among them as they have its particularities such as the geometry, dimensions and material. One of these particularities is the number of tracks they have, which are grinded and shaped depending on the position of the roll within the mill and the product that wants to be made. Each track is used independently for a specific assembly and once worn, the roll is disassembled, being the track marked. The roll, as long as it has unused tracks, it is ready to be assembled again. New rolls and those that have all of its tracks used need to have their tracks prepared in order to be able to be assembled again.

Whenever the process manager searches for a replacement of the rolls, as they work in pairs, it has to select a pair of them for each of the stands. The rolls must have an available track that meets the geometry and shape of the stand. The manager choice is based on several important factors such as the state of the roll or the number of rolled tons it has accumulated. The life cycle of the rolls is traceable as all the data related to the stock of rolls is stored in a database. There are several constraints that the rolls must meet, for example, the rolls are only valid for one of the sections of the rolling mill and the diameters of the rolls assembled on different rolling stands must accomplish dimension restrictions.

The task of selecting the suitable set of rolls for a specific set of orders is particularly complex as there is a very high number of assembled rolls, greater than a hundred, and the requirements vary among the different sections of the rolling mill, the kind of product to be rolled and the track where they are going to be assembled. Rolls also must meet diameter restrictions among rolls assembled in the same and successive stands.

Suppose a group of agents $A=(a_c,A_i)$, where A_i is the set of agents which exchange the *OK* and *NOGOOD* messages, the stands, defined by $A_i=\{A_1,A_2,\ldots,A_n\}$ and a_c is the controller agent to whom the agent A_n informs that a solution has been reached. The domain of each of the agents is composed by the rolls which meet the shape and geometry requirements $R_i=\{R_1, R_2,\ldots,R_n\}$. Each agent A_i has to select from the domain depicted in Fig. 1 one of those codes to be assembled on the stand it represents, as seen in Fig. 2, fulfilling the constraint which is supposed to be a tolerance of 5mm (this is a fictitious tolerance just to illustrate the problem in a simpler way) of difference on the diameters among consecutive rolls and the decision of the selection of a roll is taken based on the agent view, which is composed by the rolls assembled on the previous stands whose *OK* messages have reached the current agent.

In the proposed algorithm the agents have an initial predefined order, depending each one directly on the previous one in the defined order, that is, there are constraints between an agent A_i and the agents A_{i-1} and A_{i+1}. If two agents are consecutive, then some values are valid for both of them, as it is seen in Fig. 2, but only one of them can use a value simultaneously, that is why in the Fig. 2 (1) the agent A_2 cannot use the value R_2, as that roll code is already assembled on stand A_1. In this particular method, as the order is defined at the beginning in this way, with existing constraints only among consecutive agents, the *NOGOODS* involve only the previous agent so there is no need of using *ADDLINK* messages. An agent communicates with the following one sending *OK* messages in order that this one checks that the constraints are not violated and that the values are not repeated, that is, each agent sends *OK* messages to its lower priority neighbor (A_i to A_{i+1}). *OK* messages are also sent from an agent A_i to an agent A_{i+1} when the agent A_i receives an *OK* message from the A_{i-1} agent that meets the constraints so that every agent is informed of the values used by the previous ones. When an agent A_i receives an *OK* message and there are no constraints violations, the values are stored in its agent view. Given this procedure, the last agent, the agent A_4 in the Fig. 2, knows the values used on each of the previous ones, as its agent view is complete, when a solution has been reached and is able to communicate it to a controller agent.

A *NOGOOD* message is sent to an agent and the value is removed when the *OK* message received from this agent violates the diameter constraints just like in the original ABT algorithm or if the code of the roll selected is already used by the agent and there is no available value that meets the constraints for that agent. If an agent has no value that satisfies the restrictions at the beginning of the execution or if it is the first agent and has to backtrack, having no agent to do so, the controller agent receives a message informing that there is no solution.

In Fig. 2 the agent A_1 sends an *OK* message (1) to the agent A_2 informing that it has selected the roll code R_2, the agent A_2 verifies that there are no constraint violations when selecting R_3 and stores those values in its agent view. As the restrictions are met, there is no need to send a *NOGOOD* message.

When the agent A_2 selects its values, it sends an *OK* message (2) to the agent A_3 which after checking the constraints it determines that the restrictions are violated, the difference among diameters is greater than 5mm and there are no

available roll codes which fulfil the restrictions. Provided this situation and as it is seen in Fig. 2, it sends a *NOGOOD* message (3) and its agent view does not change.

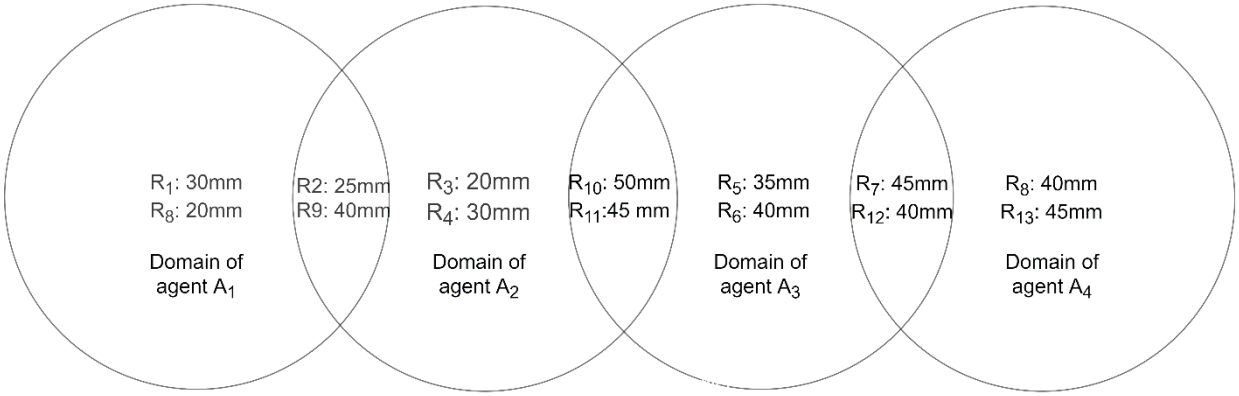
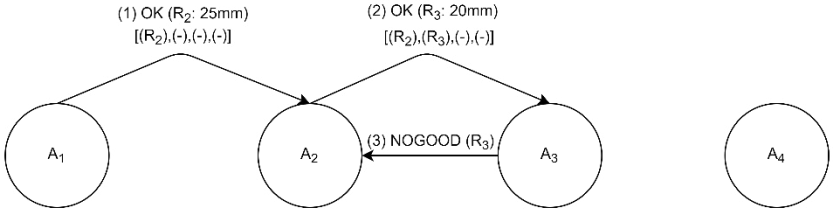


Fig. 1. Domain of the agents

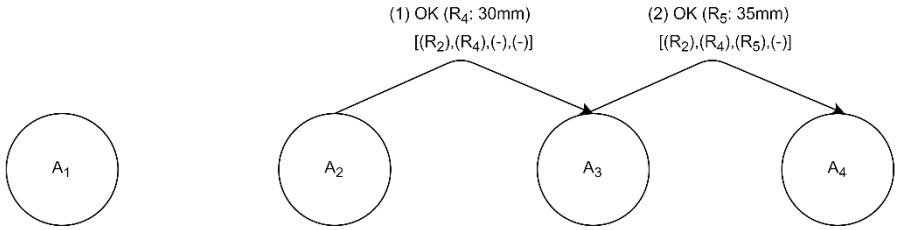


Initial situation	Value	(R_2: 25mm)	(R_2: 25mm)	(R_5: 35mm)	(R_8: 40mm)
	Agent view	[R_2,(-),(-),(-)]	[(-),(R_2),(-),(-)]	[(-),(-),(R_5),(-)]	[(-),(-),(-),(R_8)]
(1)	Value	(R_2: 25mm)	~~(R2: 25mm)~~ (R_3: 20mm)	(R_5: 35mm)	(R_8: 40mm)
	Agent view	[R_2,(-),(-),(-)]	[(R_2),(R_3),(-),(-)]	[(-),(-),(R_5),(-)]	[(-),(-),(-),(R_8)]
(2)	Value	(R_2: 25mm)	(R_3: 25mm)	(R_5: 35mm)	(R_8: 40mm)
	Agent view	[R_2,(-),(-),(-)]	[(R_2),(R_3),(-),(-)]	[(R_2),~~(R_3)~~,(R_5),(-)]	[(-),(-),(-),(R_8)]
(3)	Value	(R_2: 25mm)	~~(R_3: 25mm)~~ (R_4: 30mm)	(R_5: 35mm)	(R_8: 40mm)
	Agent view	[R_2,(-),(-),(-)]	[(R_2),(R_4),(-),(-)]	[(R_2),(-),(R_5),(-)]	[(-),(-),(-),(R_8)]

Fig. 2. Initialization of the agents and messages sent

After receiving that *NOGOOD* the agent A_2 selects other value and checks if these are valid by sending a new *OK* message Fig. 3 (1) to the agent A_3, which after checking that the constraints are met stores it in its agent view.

The agent A_3 sends an *OK* message Fig. 3 (3) to the agent A_4 to inform that the state of the previous agents has changed and that this agent knows the values used on the previous stands and its own value and stores them in its agent view. In case the *OK* message was only to inform that the state of the previous agents had changed, as there are no restrictions among not consecutive agents that have to be met, there would be no possibility of a *NOGOOD* message sent.



Initial situation	Value	(R_2: 25mm)	(R_4: 30mm)	(R_5: 35mm)	(R_8: 40mm)
	Agent view	[R_2,(-),(-),(-)]	[(R_2),(R_4),(-),(-)]	[(R_2),(-),(R_5),(-)]	[(-),(-),(-),(R_8)]
(1)	Value	(R_2: 25mm)	(R_4: 30mm)	(R_5: 35mm)	(R_8: 40mm)
	Agent view	[R_2,(-),(-),(-)]	[(R_2),(R_4),(-),(-)]	[(R_2),(R_4),(R_5),(-)]	[(-),(-),(-),(R_8)]
(2)	Value	(R_2: 25mm)	(R_4: 30mm)	(R_5: 35mm)	(R_8: 40mm)
	Agent view	[R_2,(-),(-),(-)]	[(R_2),(R_4),(-),(-)]	[(R_2),(R_4),(R_5),(-)]	[(R_2),(R_4),(R_5),(R_8)]

Fig. 3. Selection of a new value after *NOGOOD* and *OK* message to inform of changes in previous agents

The proposed algorithm described previously is summarized below with the aid of the pseudocode:

ON_START
if *an agent has available values*
  then *the agent selects an initial value and sends OK to its lower priority neighbour*
  else *the agent sends a message to the controller agent to terminate*

CHECKVIEW
if *the restrictions are met and new information has been received from previous agents*
  then *send OK with that information to its lower priority neighbour*
  else if *there are other available values that meet the constraints*
    then *select one*
      send OK to its lower priority neighbour
    else
      backtrack
      if *new information has been received from previous agents*
        then *send OK to its lower priority neighbour with that information*
if *it is the last agent and the agent view is complete*
  then *send a message to the controller agent*

BACKTRACK
if *it has higher priority neighbour*
  then *add higher priority neighbour to nogoods*

__if__ there are no nogoods
__then__ there is no solution (it is the first agent), send a message to the controller agent to terminate
__else__ send NOGOOD to the higher priority neighbour
delete element from agent view and from nogoods
checkview

HANDLE_OK_MESSAGE
add element to agent view
check if new information has been received from previous agents
checkview

HANDLE_NOGOOD_MESSAGE
__if__ the problem is that the diameter violates the constraints
__then__ all de codes of the rolls with that diameter are not selectable
__else__ the problem is that the code of the roll is already used so it is not selectable
checkview

5. Validation and conclusions

A small set of rolls, around 800, has been simulated including the different types of rolls that are in the rolling mill taking into account its geometry and the number of positions, each one having its particular shape. In this way, the situation of the warehouse has been simulated. As well, in order to test the algorithm, specific jobs scheduled to be done have been provided to the algorithm having this one to find a suitable combination of rolls to assemble. These jobs involve different number of stands, each having its particular constraints. The algorithm has been developed in Python using the framework PADE [13] and has been tested in a Windows 10 desktop computer powered by an Intel Core i5-8500 processor of 3.00 GHz with 8.00 GB of DDR4 RAM taking about 15 seconds for each simulation performed. The results are shown in Fig. 4 and as it is seen, the number of *OK* messages increases exponentially with the number of stands, however it does not happen something similar with *NOGOOD* messages, the ones with would imply a considerable delay, whose behavior is linear. Though given this behavior of the *OK* messages the algorithm is not suitable if the number of agents is very high, when the number of agents is reduced, the ability of each agent of selecting its values and conclude its feasibility just sending a message to its lower priority neighbor, having a *NOGOOD* as an answer if it is not valid, turn it into a better approach than a recursive technique such as backtracking.

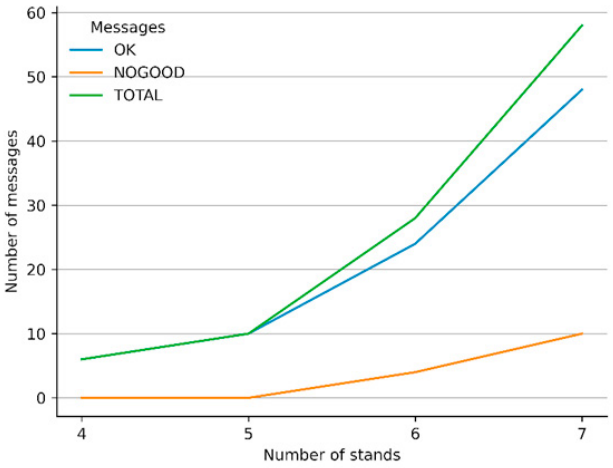


Fig. 4. Number of messages exchanged given the number of agents (stands)

Acknowledgements

This research was funded by AGENCIA ESTATAL DE INVESTIGACION (Spain), grant number MCIU-19-PCI2019-103443. The authors acknowledge CHIST-ERA SOON (Social Network of Machines) project for supporting this work.

References

[1] MAHESWARAN R, Tambe M, Bowring E, Pearce J, VARAKANTHAM P. Taking DCOP to the Real World: Efficient Complete Solutions for Distributed Event Scheduling. AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems: New York, USA, July 19-23, 2004 2004:310–7. https://doi.org/10.1109/AAMAS.2004.257.

[2] Bowring E, Tambe M, Yokoo M. Multiply-Constrained DCOP for Distributed Planning and Scheduling n.d.:8.

[3] Petcu A, Faltings B. A Value Ordering Heuristic for Local Search in Distributed Resource Allocation. In: Faltings BV, Petcu A, Fages F, Rossi F, editors. Recent Advances in Constraints, Berlin, Heidelberg: Springer; 2005, p. 86–97. https://doi.org/10.1007/11402763_7.

[4] Junges R, Bazzan ALC. Evaluating the performance of DCOP algorithms in a real world, dynamic problem. Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 2, Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems; 2008, p. 599–606.

[5] Ottens B, Faltings B. Coordination agent plans trough distributed constraint optimization. Proceedings of the Multi Agent Planning Workshop MASPLAN08, 2008.

[6] Yokoo M, Durfee EH, Ishida T, Kuwabara K. The distributed constraint satisfaction problem: formalization and algorithms. IEEE Transactions on Knowledge and Data Engineering 1998;10:673–85. https://doi.org/10.1109/69.729707.

[7] Modi PJ, Shen W-M, Tambe M, Yokoo M. Adopt: asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence 2005;161:149–80. https://doi.org/10.1016/j.artint.2004.09.003.

[8] Yokoo M. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In: Montanari U, Rossi F, editors. Principles and Practice of Constraint Programming — CP '95, Berlin, Heidelberg: Springer; 1995, p. 88–102. https://doi.org/10.1007/3-540-60299-2_6.

[9] Weiss G. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. MIT Press; 1999.

[10] Dechter R, Pearl J. Network-based heuristics for constraint-satisfaction problems. Search in artificial intelligence, Springer; 1988, p. 370–425.

[11] Silaghi M-C, Sam-Haroud D, Faltings B. Polynomial space and complete multiply asynchronous search with abstractions. Proceedings of the IJCAI'2001 Workshop on Distributed Constraint Reasoning, IJCAI; 2001, p. 17–32.

[12] Bessière C, Maestre A, Brito I, Meseguer P. Asynchronous backtracking without adding links: a new member in the ABT family. Artificial Intelligence 2005;161:7–24.

[13] Melo LS, Sampaio RF, Leão RPS, Barroso GC, Bezerra JR. Python-based multi-agent platform for application on power grids. International Transactions on Electrical Energy Systems 2019;29:e12012. https://doi.org/10.1002/2050-7038.12012.