Monte Carlo Methods Appl. **18** (2012), 147–160
DOI 10.1515/mcma-2012-0004

A restarted estimation of distribution algorithm for solving sudoku puzzles

Sylvain Maire and Cyril Prissette

Abstract. In this paper, we describe a stochastic algorithm to solve sudoku puzzles. Our method consists in computing probabilities for each symbol of each cell updated at each step of the algorithm using estimation of distributions algorithms (EDA). This update is done using the empirical estimators of these probabilities for a fraction of the best puzzles according to a cost function. We develop also some partial restart techniques in the RESEDA algorithm to obtain a convergence for the most difficult puzzles. Our algorithm is tested numerically on puzzles with various levels of difficulty starting from very easy ones to very hard ones including the famous puzzle AI Escargot. The CPU times vary from few hundreds of a second for the easy ones to about one minute for the most difficult one.

Keywords. Stochastic algorithms, estimation of distribution, sudoku puzzles, restart techniques.

2010 Mathematics Subject Classification. 90C59, 65C05.

1 Introduction

Sudoku is a very popular logic-based puzzle game which appeared in Europe in the early 2000s. Even though there are many variants of sudoku puzzles, its main version consists of the 81 cells of a 9×9 grid. Each cell should be filled with symbols that are usually the numbers from one to nine. The main grid is additionally divided in nine 3×3 blocks. Some of the cells are pre-filled by the puzzle builder such that there is one and only one puzzle that meets the following constraints: the numbers one to nine must appear only once in each row, each column and each 3×3 block. A typical easy puzzle is given in Figure 1.

While human people try to solve sudoku puzzles logically using a certain number of basic or more sophisticated tricks, most algorithms available do not use logic but optimization tools. A wide range of deterministic algorithms are used based on backtracking [16], brutal force search or constraint programming [2]. The most recent works also include a Sinkhorn balancing algorithm [13] and an algorithm based on the connections with sparse solution of underdetermined linear systems [1].

		8	6	7		4		2
4		2	3				9	
3	7			2				
			4				5	
8	9		2	5	1		6	7
	1				3			
				9			7	3
	3				5	2		1
7		5		3	2	9		

Figure 1. Easy puzzle.

Stochastic algorithms have also been developed based on simulated annealing [10] or on different metaheuristic techniques [14] like cultural algorithms [15], repulsive particle swarm optimization [9] or quantum simulated annealing [4]. The method we propose is also a stochastic algorithm but more based on adaptive Monte Carlo strategies, that is, sequential Monte Carlo methods. These methods were introduced by John Halton in the early 1960s [6] to solve in particular linear systems. They have been used successfully more recently to solve partial differential equations like transport [3] or diffusion equations [5] but also to compute approximations on orthonormal bases [12]. In the case of stochastic optimization, these algorithms are called estimation of distribution algorithms (EDA) [8, 11]. The idea is to compute adaptively probabilities for each number of each cell updated after each step of the algorithm. Empirical estimators of the probabilities of a fraction of the best puzzles according to the cost function are used to make these updates. For the most difficult puzzles, we introduce a new method called RESEDA (for restarted EDA) based on partial restarts of the algorithm while keeping parts of the puzzles that were not hundred percent correct.

Our paper is organized as follows. In Section 2, we describe the basic version of our algorithm and its main parameters. We give first the cost function that we use to evaluate the quality of a grid and how we initialize the probabilities in the different cells. Then, we discuss the updating of these probabilities from one step to another by making a linear combination of the empirical estimators of the probabilities of a fraction of the best puzzles at step $n + 1$ and of the current probabilities at step n. Finally, we describe the stopping criterion of the algorithm and the restart procedures in case of failure.

In Section 3, we test our algorithm on easy or medium examples to analyze its performances and to make the most efficient choices for its different parameters. We especially focus on the balance between the probability of success and the number of steps until convergence. It can be a lot more efficient to restart one or few times a cheap algorithm that does not always converge than a very consuming one that always does.

Section 4 is devoted to restart techniques. We observe that the usual full restart technique of the algorithm is not sufficient to make the algorithm efficient. We develop partial restart techniques where some parts of the solution obtained after the first try are considered as fixed to start the algorithm again. This idea is really useful when the chosen part is a random block, row or column.

Section 5 gives the mean performances of the algorithm on many puzzles with five different levels of difficulty taken from an open source code. We also solve the AI Escargot puzzle [7] which is considered as one of the most difficult puzzles available. Some additional restart tricks are necessary for its numerical resolution.

2 Description of the optimization algorithm

2.1 Initialization and cost function

The principle of our method is to compute a probability for each symbol of each cell at every step of the algorithm. The cells of the puzzle are denoted by $a_{i,j}$ with $1 \le i, j \le 9$ like in matrix notations and we denote by $p^{(n)}_{i,j,k}$ the probability $P^{(n)}(a_{i,j} = k)$ for the cell $a_{i,j}$ to contain the symbol k at step n of the algorithm. For each puzzle a set $G \subset [1,9]^2 \times [1,9]$ constituted of M cells and symbols is given. This set is assumed to be rich enough so that the puzzle has a unique solution. We set $P^{(0)}(a_{i,j} = k) = 1$ for all $(i, j, k) \in G$. The initialization of the other probabilities for the $81 - M$ cells and symbols belonging to G^c is uniform on the set of possible points. More precisely, for each cell $a_{i,j}$, we define the set $V_{i,j}$ of all the symbols corresponding to the points $\in G$ that have a column, a row or a 3×3 block in common with the cell $a_{i,j}$. Then we set

$$P^{(0)}(a_{i,j} = k) = \begin{cases} 0 & \text{if } k \in V_{i,j}, \\ \frac{1}{9-\mathrm{card}(V_{i,j})} & \text{otherwise.} \end{cases}$$

The optimization algorithm involves samples from the $p^{(n)}_{i,j,k}$ and we need to define a cost function to measure the quality of a puzzle and to stop the algorithm when a sample vanishes this cost function. Many cost functions are possible, the constraint is that the cost function is zero when all the rows, all the columns and all the 3×3 blocks contains all the 9 symbols. Our cost function is the sum over all rows, all columns and all blocks of the missing symbols in the rows, columns and blocks. For example, if a row contains the symbols $(2,2,3,3,3,5,6,7,8)$ its contribution to the cost function is 3 as the numbers 1, 4 and 9 are missing.

2.2 Optimization algorithm

Once the probability distribution $p_{i,j,k}^{(n)}$ is built, we draw Q samples from it and keep a small number Q_1 of the best ones according to the cost function. These best samples can be obtained by sorting the Q samples and keep the Q_1 with lowest cost function using insertion for instance. Another possibility is to take the sample with lowest cost function among $\frac{Q}{Q_1}$ samples of size Q_1. The second method has a smaller complexity but may omit very good samples.

Then for each cell $a_{i,j}$, we compute the empirical probability distribution $r_{i,j,k}^{(n+1)}$ of the k symbols among the Q_1 samples. It remains to update the new probability distribution $p_{i,j,k}^{(n+1)}$ by letting

$$p_{i,j,k}^{(n+1)} = \alpha p_{i,j,k}^{(n)} + (1-\alpha) r_{i,j,k}^{(n+1)},$$

where $0 \leq \alpha \leq 1$. The smoothing parameter α represents the fraction of the probability distribution which is kept from an iteration to the other. It is very similar to the cooling parameter which appears in simulated annealing. The larger this fraction is, the slower the convergence but the more robust the algorithm is. The number Q_1 of the best samples kept among Q samples plays the same role in the convergence. If Q_1 is large, the algorithm is more robust but the convergence is slower.

2.3 Stop and restart

The algorithm obviously stops when one of the samples solves the puzzle but we also need to stop it when the probability to obtain the solution becomes too small. Our stopping test is based on

$$\rho(n) = \min_{1\leq i,j\leq 9} \left(\max_{1\leq k\leq 9} p_{i,j,k}^{(n)} \right).$$

We stop the algorithm when $\rho(n) > \beta$ where β is a parameter close to one. Indeed, if $\rho(n) > \beta$ the algorithm is around a local minimum of the cost function which is not zero and can only escape from it with a very small probability. In many optimization problems (solved or not by stochastic algorithms) like for example the traveling salesman problem, one is not necessarily looking for the optimal solution but only to a good one according to a given criterion. The main difficulty of our optimization problem is that people are not interested in a puzzle with very few errors but only with the one with none. Moreover, the cost function often needs to cross a gap of 3 or 4 to reach the exact solution.

Our algorithm may fail to converge in one shot, so it is necessary to start it again using possibly the approximate solution obtained after this first shot. The most simple approach consists in a full restart of the algorithm starting again with the original grid. We will see in some of the numerical examples that this trick is not efficient enough for very difficult puzzles. For such puzzles, our algorithm is attracted almost every time towards a local minimum which is not the solution. Nevertheless, a large proportion of the cells have the right symbol and it is worth using this information to restart the algorithm. The partial restart algorithm consists in restarting the algorithm with additional fixed symbols or more generally with symbols with an increased probability.

3 Basic numerical examples without restarting

3.1 Easy example

Our first example (see Figure 1) is an easy puzzle taken from `www.e-sudoku.fr` (puzzle 114 949).

Our algorithm is applied without restart procedures using $Q_1 = 10$ puzzles from a sample of $Q = 100$ puzzles to update the empirical probabilities. The algorithm stops when $\rho(n) = 0.6$. These parameters will be used in all following numerical tests. We compare three algorithms based on different strategies. The first one is based on the additional constraints described in Section 2.1.

The second one gets rid of these constraints but tries to pick permutations in each of the nine 3×3 blocks whenever it is possible instead of picking each case individually. To do this, we have to conciliate the probabilities of each symbol of the individual cells while trying to pick a permutation. We do not know if this can be done optimally. Our technique is to first pick a symbol in the cell having the highest probability. Once this symbol is picked, it is removed from the possible symbols of the remaining cells of the block and the probabilities in these cells are recomputed knowing that the probability of this symbol is now zero. This method is iterated until the last cell is filled. Note that it may happen that it is impossible to pick a permutation especially for the final steps of the process. In this case, we just pick the remaining symbols uniformly at random.

The third strategy is the combination of the first two: trying to pick permutations in blocks while taking into account the constraints given by the fixed cells. This may increase the number of blocks that are not permutations but these ones are likely to be discarded by the selection procedure. If we go back to the properties

	permutations		pre-computation		both optimizations	
α	success	rounds	success	rounds	success	rounds
0.99	98	574	100	522	100	188
0.9	82	67	96	60	100	25
0.7	42	25	71	22	100	9
0.5	11	17	35	14	100	6

Table 1. Success rates with the different optimizations.

of EDA [11], the strategy of filling the cells independently is univariate while the two other ones are multivariate strategies on the nine cells of a block.

Table 2 represents the number of successful algorithms among one hundred and the number of iteration steps in case of convergence for different values of α for each of the strategies described above.

These preliminary results show that the algorithm works well in all its versions and it is better to run our algorithm with the two optimization tools simultaneously. Indeed these tools reduce the search space and avoid many impossible puzzles. As a consequence they will be used in all following numerical examples. We can also remark that increasing α leads to a more successful algorithm but may increase significantly the number of iterations until convergence. Looking at the results we obtain on this easy puzzle, it seems to be a lot more efficient in terms of computational times to take a smaller value of α coupled eventually with a full restart procedure. We shall investigate on other examples if this still holds and choose a value of α close to optimality in terms of complexity. Finally, we can give the performances of the algorithm. If we choose both optimizations and $\alpha = 0.7$, the mean number of puzzles tested is only 900 and the mean CPU times are 0.03 seconds. Our program is written in C++ and executed on a desktop PC, with an Intel Core 2 Duo 2.66 GHz processor running under GNU/Linux.

3.2 Medium difficulty example

The second puzzle (Figure 2) is taken from [1] and has a medium level of difficulty. Our algorithm can still solve quite easily this puzzle but the probability of success decreases significantly when α decreases as illustrated in Table 2 where α is taken between 0.1 and 0.99.

As the algorithm does not always solves the puzzle, it is necessary to restart the algorithm one or a few times until convergence. It is interesting to find a balance between the probability of success and the mean number of iterations until

	2	4			7			
6								
		3	6	8		4	1	5
4	3	1			5			
5							3	2
7	9						6	
2		9	7	1		8		
	4			9	3			
3	1				4	7	5	

Figure 2. Medium puzzle.

α	success	iterations
0.99	99	277
0.9	81	37
0.7	61	14
0.5	43	8
0.3	26	7
0.1	12	7

Table 2. Success rates.

convergence or a restart. We have tested different puzzles with different levels of difficulty for which the value $\alpha = 0.7$ has appeared as a good balance. This value will be used in all of the following tests.

4 Solving sudoku puzzles with partial restart

4.1 Behavior of the algorithm with harder puzzles

We now take a harder example from [1]. This puzzle, shown in Figure 3, is a lot more difficult to solve with our algorithm. In fact, we only succeed in solving it 3 times out of a hundred.

	1		7		8	9		
3	8							
		9			5	6		
	9			7				
	3	1					2	
			4	5			8	
	5			6	2	4	9	
6	7	3		4	9		5	1
	4							3

Figure 3. Hard puzzle.

	A	B	A	B	A	A	B	
A	A		A	A	A			
B	A	A	A	B	A	A		A
A	A		A	A				
	A	A	A	A	A		A	
A	A	A	A	A			A	A
A	A	A	A	A	A	A	A	A
A	A	A	A	A	A	A	A	A
A	A	A	A	A	A	A	A	A

Figure 4. Hard puzzle: level of correctness of each cell.

Even though the probability of solving the puzzle in one shot is quite small, it is very interesting to have a precise look at the puzzles obtained even if they are not completely solved. Indeed, to distinguish two puzzles only by the fact that they are true or wrong may hide that the wrong puzzles are partially correct. The grid shown in Figure 4 has been built using one hundred tries of the algorithm and indicates the level of correctness of each of the cells of the puzzles. For each cell of the puzzle, we give the grade A if it is always correct and the grade B if it is not always correct but at least ninety times out of a hundred. An empty cell indicates that the right value has been found less than ninety times out of a hundred.

We can see that many rows, blocks or columns are always or very often correct using our algorithm. For instance the three bottom blocks (and consequently the three bottom rows!) as well as column two and four are always correct. Some other regions like column five or line three have also a large probability to be correct. In the next subsection, we shall use these nice properties to develop partial restart techniques.

4.2 Restarted estimation of distribution algorithm

The difficulty of a puzzle mainly depends on the number of the initial fixed cells. Adding only few more fixed cells can transform a very hard puzzle into an easy one. Our algorithm could for instance take advantage of few basic deterministic guesses from human sudoku experts but we intend for the moment to keep it fully stochastic. Nevertheless we can try to make these guesses from our algorithm using the conclusions of the previous subsection. We have seen that many individual cells and also complete rows, columns or blocks are correct. The partial restart estimation of distribution algorithm (RESEDA) consists in adding to the initial fixed cells columns, rows or blocks obtained after a first resolution of the algorithm while discarding obviously any of them which is not a permutation. We have tried to fix cells at random among the ones obtained after the first resolution of the algorithm but this was not really efficient. Indeed, this increases a lot the probability that one of the new fixed cells is wrong compared to fixing all the cells of a row, block or column. Similarly, we have chosen to add only one zone at a time because the probability that two zones are simultaneously correct is small.

There are 27 ways to complete the initial puzzle if we add one zone at a time. We could run any of these 27 configurations but we have observed in practice that if the algorithm fails to converge for one of the type of zones it is unlikely to converge for the others. In this case, it is better to make a full restart which avoids 24 more resolutions from initial configurations that are likely to be wrong.

Table 3 describes this behavior for the previous hard puzzle. It gives the number of successes among 1000 attempts of resolution and the average total execution time in seconds. Four methods are compared: without restart, when we restart with one fixed zone at a time out of 3, 9 or 27 zones which are, respectively, the 3×3 blocks on the first diagonal, all the 3×3 blocks and all the columns, rows and 3×3 blocks. The estimation of mean time of success (ETS) of the algorithm is also given.

As one can see in this example, the best trade-off between execution time and success rate is a partial restart with one block at a time, out of the 3 blocks on the first diagonal. Moreover, this example shows that the algorithm is faster with a

method	number of successes	time	ETS
no partial restart	89	0.198	2.225
restart with 1 zone out of 3	474	0.614	1.295
restart with 1 zone out of 9	619	1.113	1.798
restart with 1 zone out of 27	906	1.694	1.869

Table 3. Comparison of restart techniques on the hard puzzle.

partially completed puzzle. This is a common and quite obvious behavior: adding clues makes the puzzle easier and so the resolution attempt needs less time.

Our partial restart technique will be based only on resolutions on puzzles completed by one of the 3 blocks obtained after a first resolution of the basic algorithm. If the algorithm does not converge after trying each of these 3 blocks, then we do a full restart of the algorithm.

5 More numerical tests

5.1 Test on different levels of difficulty

We shall now give in Table 4 more numerical examples on puzzles with five levels of difficulty: very easy, easy, medium, hard and fiendish taken from the open source code sudoku 1.0.1-3 by Michael Kennett. For each level of difficulty, 1000 puzzles are randomly generated. The measured times are the execution time expressed in seconds until the algorithm stops in case of convergence or in case of failure.

For each set of 1000 puzzles, we count the number of successfully solved puzzles and the average execution time fixing successively one zone among the 27 zones after the restart. Then we do the same algorithm with only the nine 3×3 blocks and finally among the three 3×3 blocks of the first diagonal.

The success rate is high when we fix exhaustively the 27 zones, but one can see that using less zones insures a good success rate and decreases significantly the execution time. This can be explained quite easily: at the end of the first step, the algorithm gives many successfully solved zones, even if we don't know which zones are successfully solved. This is the key idea of the partial restart.

However, sometimes at the end of the first step, the algorithm finds a local minimum which is very different from the solution. In this case, restarting is a loss of time if no fixed zone is correct. Thus, limiting the number of tested fixed blocks

	27 zones		9 zones		3 zones	
	success	time	success	time	success	time
very easy	1000	0.046	1000	0.049	1000	0.047
easy	1000	0.113	994	0.129	982	0.104
medium	959	1.015	884	0.718	659	0.502
hard	811	2.394	563	1.451	365	0.726
fiendish	605	3.458	336	1.831	221	0.824

Table 4. Simulations on puzzles with different difficulties.

for a restart is a good choice because it keeps a good success rate if the first step has found an interesting partial solution, and limits the loss of time if this first step gives a poor partial solution.

So, we recommend to make a partial restart 3 times, fixing one of the 3 blocks on the first diagonal for each restart. If no solution is found, then we recommend to make a full restart. The mean time of resolution using this strategy is the mean time of the algorithm divided by the probability of success. This mean time goes from 0.047 seconds for the very easy puzzles to $\frac{0.824}{0.221} \approx 3.71$ seconds for the fiendish ones.

5.2 Very hard puzzle: AI Escargot

Until now, the puzzles were either quite easy puzzles built to be solved by human players, or randomly generated puzzles with a level of difficulty measure with some criteria, such as the length of a deduction chain needed to solve them with a deterministic algorithm.

In order to illustrate the behavior of the RESEDA algorithm, we now use it to solve a puzzle of another kind: a puzzle built to be the most complex to solve. The puzzle we study is known as AI Escargot and has been created by Arto Inkala who introduces it as the most difficult puzzle in the world. This puzzle which has a unique solution is given in Figure 5.

Once more, our algorithm is executed 1000 times and we measure in Table 5 the success rate and the average execution time, in the 3 configurations: partial restart with one zone out of 3, 9 or 27 zones.

As one can see, this is a hard puzzle to solve for the algorithm RESEDA. The success rate is not zero if we make a partial restart with 9 or 27 zones, but it is still low. There are many reasons to explain this behavior. First, the puzzle has

1					7		9	
	3			2				8
		9	6			5		
		5	3			9		
	1			8				2
6					4			
3							1	
	4							7
		7				3		

Figure 5. AI Escargot puzzle.

number of tested zones	number of successes	CPU time
3	0	1.251
9	1	2.857
27	9	7.045

Table 5. Simulations with 1000 resolutions of AI Escargot.

few clues: only 23 numbers are given. This can be a hindrance to the distribution evaluation at the beginning of the algorithm and make it converge towards a local minimum which is very different from the solution. Moreover, even if a few zones are less hard to solve, we have no evidence that they are the 3×3 blocks of the first diagonal. So for this kind of puzzle, we recommend to use exhaustively the 27 zones for the partial restarts.

The execution time (with success or not) is roughly twice the time needed to solve a fiendish puzzle but the probability of success is only $\frac{9}{1000}$. This leads to a mean time of success of about 778 seconds which is quite long. In order to reduce this CPU times, we propose to couple RESEDA with brutal force techniques. The idea is to run the algorithm considering as fixed each of the admissible permutations of a given 3×3 block. If we do this for the upper left corner block, there are only 24 permutations meeting the basic constraints. One of these 24 permutations is the right permutation, and it adds 6 clues to the puzzle. When it is correctly guessed, the success rate grows to $\frac{580}{1000}$ with a CPU time of 1.517 seconds, which leads to a mean time of success of $\frac{1000 \cdot 24 \cdot 1.517}{580} \approx 63$ seconds. Using this new idea, the CPU times are divided by ten.

6 Conclusion

We have introduced and tested a new stochastic algorithm RESEDA for solving sudoku puzzles. The basic versions enable us to solve easy or medium puzzles in one or a few shots. To solve the most difficult ones, partial restart techniques were necessary to obtain good convergence properties. The CPU times vary from few hundreds of a second to about one minute for the AI Escargot puzzle. The algorithm seems to be faster than the other stochastic algorithms developed in [10, 14]. Finding ways for decreasing the difficulty of the puzzle is a key idea for further improvements of RESEDA. We have already done a basic attempt based on a combination with brutal force but it should be certainly more efficient to combine RESEDA with other resolution algorithms especially for 4×4 or 5×5 puzzles. One could for instance find some easy clues using a deterministic method and then use our algorithm with these supplementary clues.

Bibliography

[1] P. Babu, K. Pelckmans, P. Stoica and J. Li, Linear systems, sparse solutions, and sudoku, *IEEE Signal Process. Lett.* **17** (2010), 40–42.

[2] A. Bartlett and A. Langville, An integer programming model for the sudoku problem, *J. Online Math. Appl.* **8** (2008), article ID 1798.

[3] T. Booth, Exponential convergence on a continuous Monte Carlo transport problem, *Nucl. Sci. Eng.* **127** (1997), 338–345.

[4] A. Das and B. K. Chakrabati (Eds.), *Quantum Annealing and Related Optimization Methods*, Lecture Notes in Physics 679, Springer-Verlag, Heidelberg, 2005.

[5] E. Gobet and S. Maire, A spectral Monte Carlo method for the Poisson equation, *Monte Carlo Methods Appl.* **10** (2004), 275–285.

[6] J. Halton, Sequential Monte Carlo, *Proc. Camb. Phil. Soc.* **58** (1962), 57–78.

[7] A. Inkala, *AI Escargot – The Most Difficult Sudoku Puzzle*, Bertrams Print on Demand, 2007.

[8] P. Larranaga and J. A. Lozano (Eds.), *Estimation of Distribution Algorithms, A New Tool for Evolutionary Computation*, Kluwer Academic Publishers, 2002.

[9] K. H. Lee, S. W. Baek and K. W. Kim, Inverse radiation analysis using repulsive particle swarm optimization, *Int. J. Heat Mass Transfer* **51** (2008), 2772–2783.

[10] R. Lewis, Metaheuristic can solve sudoku puzzles, *J. Heuristics* **13** (2007), 387–401.

[11] J. A. Lozano, P. Larranaga, I. Inaki and E. Bengoetxea (Eds.), *Towards a New Evolutionary Computation*, Advances on Estimation of Distribution Algorithms, Springer, 2006.

[12] S. Maire, Reducing variance using iterated control variates, *J. Stat. Comput. Simul.* **73** (2003), 1–29.

[13] T. K. Moon, J. H. Gunther and J. Kupin, Sinkhorn solves Sudoku, *IEEE Trans. Inform. Theory* **55** (2009), 1741–1746.

[14] M. Perez and T. Marwala, Stochastic optimization approaches for solving sudoku, preprint (2008), `http://arxiv.org/abs/0805.0697`.

[15] R. G. Reynolds, An introduction to cultural algorithms, in: *Proceedings of the 3rd Annual Conference on Evolutionary Programming*, World Scientific Publishing (1994), 131–139.

[16] G. Santos-Garcia and M. Palomino, Solving sudoku puzzles with rewriting rules, *Electron. Notes Theor. Comput. Sci.* **17** (2007), 79–83.

Received July 20, 2011; accepted February 16, 2012.

Author information

Sylvain Maire, Laboratoire des Sciences de l'Information et des Systemes (LSIS), UMR6168, ISITV, Universite de Toulon et du Var, Avenue G. Pompidou,
BP 56, 83262 La Valette du Var cedex, France.
E-mail: `maire@univ-tln.fr`

Cyril Prissette, Laboratoire des Sciences de l'Information et des Systemes (LSIS), UMR6168, ISITV, Universite de Toulon et du Var, Avenue G. Pompidou,
BP 56, 83262 La Valette du Var cedex, France.
E-mail: `prissette@univ-tln.fr`