Local Search with P Systems:
A Case Study

Miguel A. Gutiérrez-Naranjo, University of Sevilla, Spain

Mario J. Pérez-Jiménez, University of Sevilla, Spain

ABSTRACT

Local search is currently one of the most used methods for finding solutions in real-life problems. It is usually considered when the research is interested in the final solution of the problem instead of the how the solution is reached. In this paper, the authors present an implementation of local search with Membrane Computing techniques applied to the N-queens problem as a case study. A CLIPS program inspired in the Membrane Computing design has been implemented and several experiments have been performed. The obtained results show better average times than those obtained with other Membrane Computing implementations that solve the N-queens problem.

Keywords: *CLIPS, Local Search, Membrane Computing, N-Queens, P Systems, Problem*

1. INTRODUCTION

Searching is the basis of many processes in Artificial Intelligence. The key point is that many real-life problems can be stated as a *space of states*: a *state* is the description of the world at a given instant (expressed in some language) and two states are linked by a *transition* if the second state can be reached from the previous one by applying one *elementary operation*. By using these concepts, a searching tree where the nodes are the states, the root is the starting state and the edges are the actions considered. Given an initial state, a sequence of transitions to one of the final states is searched.

By using this abstraction, searching methods have been deeply studied by themselves, forgetting the real-world problem which they fit. The studies consider aspects as the completeness (if the searching method is capable of finding a solution if it exists), complexity in time and space, and optimality (if the found solution is *optimal* in some sense). By considering the searching tree, classical search has focused on the order in which the nodes should be explored. In this classical search two approaches are possible: the former is *blind search*, where the search is guided only by the topology of the tree and no information is available from the states; the latter is called *informed search* and some information about the features of the nodes is used to define a *heuristics* to decide the next node to explore.

Searching problems have been previously studied in the framework of Membrane Computing. In Gutiérrez-Naranjo and Pérez-Jiménez

DOI: 10.4018/jncr.2011040104

(2010), a first study on depth-first search in the framework of Membrane Computing was presented. In this paper we go further with the study of searching methods in Membrane Computing by exploring local search. We consider the N-queens problem as a case study and we present a family of P systems which solves it by implementing local search ideas.

The paper is organized as follows: First we recall some basic definitions related to local search. Then, we recall the problem used as a case study: the *N*-queens problem, previously studied in the framework of Membrane Computing in Gutiérrez-Naranjo, Martínez-del-Amor, Pérez-Hurtado, and Pérez-Jiménez (2009). Next, we provide some guidelines of the implementation of local search in our case study and show some experimental results. The paper finishes with some final remarks.

2. LOCAL SEARCH

Classical search algorithms explore the space of states systematically. This exploration is made by keeping one or more paths in memory and by recording the alternatives in each choice point. When a final state is found, the path, that is, the sequence of *transitions*, is considered as the solution of the problem. Nonetheless, in many problems, we are only interested in the found state, not properly in the path of transitions. For example, in job-shop scheduling, vehicle routing or telecommunications network optimization, we are only interested in the final state (a concrete disposition of the objects in the world), not in the way in which this state is achieved.

If the sequence of elementary transitions is not important, a good alternative to classical searching algorithms is *local search*. This type of search operates using a single state and its set of neighbors. It is not necessary to keep in memory how the current state has been obtained.

Since these algorithms do not systematically explore the states, they do not guarantee that a final state can be found, i.e., they are not complete. Nonetheless, they have two advantages that make them interesting in many situations:

- Only a little piece of information is stored, so very little memory (usually constant) is used.
- These algorithms can often find a reasonable solution in an extremely large space of states where classical algorithms are unsuitable.

The basic strategy in local search is considering a current state and, if it is not a final one, then it moves to one of its neighbors. This movement is not made randomly. In order to decide where to move, a *measure of goodness* is introduced in local search. In this way, the movement is performed towards the best neighbor or, at least, a neighbor who improves the current measure of goodness. It is usual to visualize the *goodness* of a state as its height in some geometrical space. In this way, we can consider a landscape of states and the target of the searching method is to arrive to the *global maximum*. This metaphor is useful to understand some of the drawbacks of this method: *flat regions*, where the neighbors are as good as the current state, or *local maximum* where the neighbors are worse than the current state, but it is not a *global maximum*. A deep study of local search is out of the scope of this paper. Further information can be found in Russell and Norvig (2002).

In this paper we will only consider the basic algorithm of local search: Given a *set of states*, *a movement operator* and *a measure to compare states*

0. We start with a state randomly chosen.
1. We check if the current state is a final one.
1.1. If so, we finish. The system outputs the current state.
1.2. If not, we look for a movement which reaches a *better* state.
1.2.1. If it exists, we randomly choose one of the possible movements.
The reached state becomes the *current state* and we go back to 1.
1.2.2. If it does not exist, we go back to 0.

3. THE N-QUEENS PROBLEM

Through this paper we will consider the *N*-queens problem as a case study. It is a generalization of a classic puzzle known as the 8-queens puzzle. The original one is attributed to the chess player Max Bezzel and it consists of putting eight queens on an 8×8 chessboard in such way that none of them is able to capture any other using the standard movement of the queens in chess, i.e., only one queen can be placed on each row, column and diagonal line (Hoffman, Loessi, & Moore, 1969; Bernhardsson, 1991).

In Gutiérrez-Naranjo et al. (2009), a first solution to the *N*-queens problem in the framework of Membrane Computing was shown. For that aim, a family of deterministic P systems with active membranes was presented. In this family, the *N*-th element of the family solves the *N*-queens problem and the last configuration encodes *all* the solutions of the problem.

In order to solve the problem, a truth assignment that satisfies a formula in conjunctive normal form (CNF) is searched. This problem is exactly SAT, so the solution presented in Gutiérrez-Naranjo et al. (2009) uses a modified solution for SAT from Pérez-Jiménez, Romero-Jiménez, and Sancho-Caparrini (2003). Some experiments were presented by running the P systems with an updated version of the P-lingua simulator (García-Quismondo, Gutiérrez-Escudero, Pérez-Hurtado, Pérez-Jiménez, & Riscos-Núñez, 2009). The experiments were performed on a system with an Intel Core2 Quad CPU (a single processor with 4 cores at 2,83Ghz), 8GB of RAM and using a C++ simulator under the operating system Ubuntu Server 8.04. According to the representation in Gutiérrez-Naranjo et al. (2009), the 3-queens problem is expressed by a formula in CNF with 9 variables and 31 clauses. The *input multiset* has 65 elements and the P system has 3185 rules. Along the computation, 2^9=512 elementary membranes need to be considered in parallel. Since the simulation was carried out on a uniprocessor system, these membranes were evaluated sequentially. The 117-th configuration was a halting one. It took 7 seconds to reach it and it has an object No in the environment. As expected, this means that three queens cannot be placed on a 3×3 chessboard satisfying the restrictions. In the 4-queens problem, four non-attacking queens are searched on a 4×4 chessboard. According to the representation, the problem can be expressed by a formula in CNF with 16 variables and 80 clauses. Along the computation, 2^{16}=65536 elementary membranes were considered in the same configuration and the P system has 13622 rules. The simulation takes 20583 seconds (> 5 hours) to reach the halting configuration. It is the 256-th configuration and in this configuration one object Yes appears in the environment. This configuration has two elementary membranes encoding the two solutions of the problem (Gutiérrez-Naranjo et al., 2009).

In Gutiérrez-Naranjo and Pérez-Jiménez (2010), a study of depth-first search in the framework of Membrane Computing was presented. The case study was also the *N*-queens problem. An *ad hoc* CLIPS program was written based on a Membrane Computing design. Some experiments were performed on a system with an Intel Pentium Dual CPU E2200 at 2,20 GHz, 3GB of RAM and using CLIPS V6.241 under the operating system Windows Vista. Finding one solution took 0,062 seconds for a 4×4 board and 15,944 seconds for a 20×20 board.

4. A P SYSTEM FAMILY FOR LOCAL SEARCH

In this section we give a sketch of the design of a P system family which solves the *N*-queens problem by using local search, $\prod=\{\prod(N)\}_{N\in N}$. Each P system $\Pi(N)$ solves the *N*-queens problem in a non-deterministic way, according to the searching method. The membrane structure does not change along the computation and we use electrical charges on the membranes as in the model of active membranes[1].

One *state* is represented by an $N\times N$ chess board where N queens have been placed. In order to limit the number of possible states, we consider an important restriction: we consider

that there is only one queen in each column and in each row. By using this restriction, we only need to check the diagonals in order to know whether a board is a solution to the problem or not.

These boards can be easily represented in Membrane Computing. For the P system $\Pi(N)$, we consider a membrane structure which contains N elementary membranes labelled with $1,\ldots,N$ and N objects y_i $i\in\{1,\ldots,N\}$ in the skin. By using rules of type $y_i[\ ]j\rightarrow[y_i]_j$ the objects y_i are non-deterministically sent into the membranes and the object y_i inside a membrane with label j is interpreted as a queen placed on the row i of the column j. For example, the partial configuration $[[y_1]_1\ [y_5]_2\ [y_3]_3\ [y_4]_4\ [y_2]_5$ is a membrane representation of the board in Figure 1.

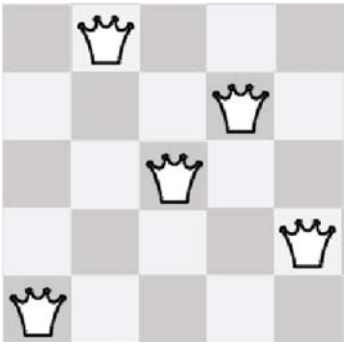
In order to know if one state is better than another, we need to consider a *measure*. The natural measure is to associate to any board the number of *collisions* (Sosic & Gu, 1994): *The number of collisions on a diagonal line is one less than the number of queens on the line, if the line is not empty, and zero if the line is empty. The sum of collisions on all diagonal lines is the total number of collisions between queens.* For example, if we denote by d_p the descendant diagonal for squares (i,j) where $i+j=p$ and by u_q the ascendant diagonal for squares (i,j) where $i-j=q$, then the board shown in Figure 1 has 3 collisions: 2 in u_0 and 1 in d_7. This basic definition of *collisions of a state* can be refined in a Membrane Computing algorithm. As we will see below, in order to compare two boards, it is not important the exact amount of collisions when they are greater than 3.

Other key definitions in the algorithm are the concepts of *neighbor* and *movement*. In this paper, a movement is the interchange of columns of two queens by keeping the rows. In other words, if we have one queen at (i,j) and another at (k,s), after the movement these queens are placed at (i,s) and (k,j). It is trivial to check that, for each movement, if the original board does not have two queens on the same column and row, then the final one does not have it. The definition of neighbor depends on the definition of movement: the state s_2 is a neighbor of state s_1 if it can be reached from s_1 with one movement.

According to these definitions, the local search algorithm for the N-queens problem can be written as follows:

0. We start with a randomly chosen state;
1.- We check if the number of collisions of the current state is zero;
1.1. If so, we finish. The halting configuration codifies the solution board.
1.2. If not, we look for movements which reach a state with a lower number of collisions.
1.2.1. If they exist, we randomly choose one of the possible movements.
The reached state becomes the *current state* and we go back to 1.
1.2.2. If they do not exist, we go back to 0.

Figure 1. Five queens on a board. We consider the origin of coordinates at bottom left



At this point, three basic questions arise from the design of Membrane Computing: (1) how the number of collisions of a board is computed? (2) how a *better* state is searched? and (3) how a movement is performed?

4.1. Computing Collisions

The representation of an $N \times N$ board is made by using N elementary membranes where the objects $y_1, ..., y_N$ are placed. These N elementary membranes, with labels $1,\ldots,N$, are not the unique elementary membranes in the membrane structure. There are $2N-1$ ascendant and $2N-1$ descendant diagonals in an $N \times N$ board. As pointed above, we denote the ascendant diagonals as $u_{-N+1}, ..., u_{N-1}$ where the index p in u_p denotes that the diagonal corresponds to the squares (i,j) with $i-j=p$. Analogously, the descendant diagonals are denoted by $d_2, ..., d_{N+N}$ where the index q in d_q denotes that the diagonal corresponds to the squares (i,j) with $i+j=q$.

Besides the N elementary membranes with labels $1,\ldots,N$ for encoding the board, we also place $4N-2$ elementary membranes in the structure, with labels $u_{-N+1}, ..., u_{N-1}, d_2, ..., d_{N+N.}$ These membranes will be used to compute the collisions.

Bearing in mind the current board, encoded by membranes with an object $[y_i]_j$, we can use rules of type $[y_i]_j \rightarrow d_{i+j}\, u_{i-j}$. These rules are triggered in parallel and they produce as many objects d_q (resp. u_p) as queens which are placed on the diagonal d_q (resp. u_p).

Objects d_q and u_p are sequentially sent into the elementary membranes labelled by d_q and u_p. In a first approach, one can consider a counter z_i which evolves to z_{i+1} inside each elementary membrane when an object d_q or u_p is sent in. By using this strategy, the index i of z_i denotes how many objects have crossed the membrane, or in other words, how many queens are placed on the corresponding diagonal.

This strategy has an important drawback. In the worst case, if all the queens are placed on the same diagonal, at least N steps are necessary in order to count them. In our design, this is not necessary. As we will see, we only need to know if the number of queens in each diagonal is 0, 1, 2, or more than 2. Due to the parallelism of the P systems, this can be checked in a constant number of steps regardless of the number of queens.

Technically, after using a complex set of rules where the electrical charges are used to control the flow of objects, each membrane d_q sends to the skin a complex object of type $d_q(DA_q, DB_q, DC_q, DD_q)$ where DA_q, DB_q, DC_q, $DD_q \in \{0,1\}$ codify the number of queens on the diagonal (for u_p the development is analogous, with the notation $u_p(UA_p, UB_p, UC_p, UD_p)$). We consider four possibilities:

- $d_q(1,0,0,0)$. The 1 in the first coordinate denotes that there is no queen placed on the diagonal and the diagonal is ready to receive one queen after a movement.
- $d_q(0,1,0,0)$. The 1 in the second coordinate denotes that there is one queen placed on the diagonal. This diagonal does not contain collisions but it should not receive more queens.
- $d_q(0,0,1,0)$. The 1 in the third coordinate denotes that there are two queens placed on the diagonal. This diagonal has one collision which can be solved by a unique appropriate movement.
- $d_q(0,0,0,1)$. The 1 in the fourth diagonal denotes that there are more than two queens placed on the diagonal. This diagonal has several collisions and it will have at least one collision even if one movement is performed.

Bearing in mind that a diagonal is ready to receive queens (0 queens) or it needs to send queens to another diagonal (2 or more queens), we can prevent if a movement produces an improvement in the whole number of collisions before performing the movement. We do not need to perform the movement and then to count the number of collisions in order to know if the movement decreases the number of collisions. In order to do that, we distinguish if the pair of queens to be moved are placed on the same diagonal or not.

Firstly, let us consider two queens placed in the squares (i,j) and (k,s) of the same ascendant diagonal, i.e., $i-j=k-s$. We wonder if the movement of interchanging the columns of two queens by keeping the rows will improve the total number of collisions. In other words, we wonder if removing the queens from (i,j) and (k,s) putting them at (i,s) and (k,j) improves the board.

In order to answer this question we consider the following objects:

- $u_{i-j}(0,0,UC_{i-j},UD_{i-j})$: The ascendant diagonal u_{i-j} has at least 2 queens, so the first two coordinates are 0. The parameters UC_{i-j}, UD_{i-j} can be 0 or 1, but exactly one of them is 1.
- $d_{i+j}(0,DB_{i+j},DC_{i+j},DD_{i+j})$ and $d_{k+s}(0,DB_{k+s},DC_{k+s},DD_{k+s})$: The descendent diagonals and d_{k+s} have at least 1 queen, so the first coordinate is 0. The remaining ones can be 0 or 1, but exactly one of them is 1.

It is easy to check that the reduction in the whole amount of collisions produced by the removal of the queens from the squares (i,j) and (k,s) is

$$(2\ UC_{i-j}) + (3\ UD_{i-j}) + DB_{i+j} + (2\ DC_{i-j}) + (3\ DD_{i+j}) + DB_{k+s} + (2\ DC_{k+s}) + (3\ DD_{k+s}) - 3$$

Analogously, in order to compute the change in the number of collisions produced by the placement of two queens in the squares (i,s), (k,j) we consider $d_{i+s}(DA_{i+s},DB_{i+s},DC_{i+s},DD_{i+s})$, $u_{i-s}(UA_{i-s},UB_{i-s},Uc_{i-s},UD_{i-s})$ and $u_{k-j}(UA_{k-j},UB_{k-j},UC_{k-j},UD_{k-j})$.

By using this notation, it is easy to check that the augmentation in the number of collisions is $4 - (DA_{i+s} + UA_{i-s} + UA_{k-j}$.

The movement represents an improvement in the general situation of the board if the reduction in the number of collisions is greater than the augmentation. This can be easily expressed with a simple formula depending on the parameters.

This is the key point in our Membrane Computing algorithm, since we do not need to perform the movement and then to check if we have an improvement, but we can evaluate it *a priori*, by exploring the objects placed in the skin. Obviously, if the squares share a descendant diagonal, the situation is symmetric.

If the queens do not share a diagonal, the study is analogous, but the obtained formula by considering that we get a *feasible movement* if the reduction is greater than the augmentation is slightly different.

From a technical point of view, we consider a finite set of rules with the following interpretation: *If the corresponding set of objects*

$u_{i-j}(UA_{i-j},UB_{i-j},UC_{i-j},UD_{i-j})$
$d_{i+j}(DA_{i+j},DB_{i+j},DC_{i+j},DD_{i+j})$
$u_{k-s}(UA_{k-s},UB_{k-s},UC_{k-s},UD_{k-s})$
$d_{k+s}(DA_{k+s},DB_{k+s},DC_{k+s},DD_{k+s})$
$u_{i-s}(UA_{i-s},UB_{i-s},UC_{i-s},UD_{i-s})$
$d_{i+s}(DA_{i+s},DB_{i+s},DC_{i+s},DD_{i+s})$
$u_{k-j}(UA_{k-j},UB_{k-j},UC_{k-j},UD_{k-j})$
$d_{k+j}(DA_{k+j},DB_{k+j},DC_{k+j},DD_{k+j})$

is placed in the skin, then the movement of queen from (i,j) and (k,s) to (i,s) and (k,j) improves the number of collisions.

In the general case, there will be many possible applications of rules of this type. The P system chooses one of them in a non-deterministic way. The application of the rule introduces an object $change_{iks}$ in the skin. After a complex set of rules, this object produces a new configuration and the cycle starts again.

The design of the P system depends on N, the number of queens, and it is rather complex from a technical point of view. It uses cooperation, inhibitors and electrical charges in order to control the flow of objects. In particular, a set of rules halts the P system if a board with zero collisions is reached and another set of rules re-starts the P system (i.e., it produces a configuration equivalent to the initial one) if no more improvements can be achieved from the current configuration.

Figure 2 shows a solution found with the corresponding P system for the 5-queens problem. We start with a board with all the queens in the main ascendant diagonal (upper left). The

number of collisions in this diagonal (and in the whole board) is 4. By changing queens from the columns 2 and 5, we obtain the board shown in Figure 1 with 3 collisions (upper right in Figure 2). In the next step, the queens from columns 1 and 5 are changed, and we get a board with 2 collisions, produced because the two main diagonals have two queens each (bottom left). Finally, by changing queens is the columns 1 and 3, we get a board with no collisions that represents a solution to the 5-queens problem (bottom right).

5. EXPERIMENTAL RESULTS

An *ad hoc* CLIPS[2] program was written *inspired* on this Membrane Computing design[3]. Some experiments were performed on a system with an Intel Pentium Dual CPU E2200 at 2,20 GHz, 3GB of RAM and using CLIPS V6.241 under the operating system Windows Vista.

Due to the random choosing of the initial configuration and the non-determinism of the P system for choosing the movement, 20 experiments have been performed for each number N of queens for $N \in \{10,20,\ldots,200\}$ in order to get an informative parameter. We have considered the average of these 20 experiments on the number of P system steps and the number of seconds. Table 1 shows the result of the experiments.

Notice, for example, that in the solution presented in Gutiérrez-Naranjo and Pérez-Jiménez (2010), the solution for 20 queens was obtained after 15,944 seconds. The average time obtained with this approach is 0.133275 seconds.

Figure 2. Starting from a configuration C_0 with 4 collisions (up-left) we can reach C_1 with 3 collisions (up-right); then C_2 with 2 collisions (bottom-left) and finally C_3 with 0 collisions (bottom-right), which is a solution to the 5-queens problem

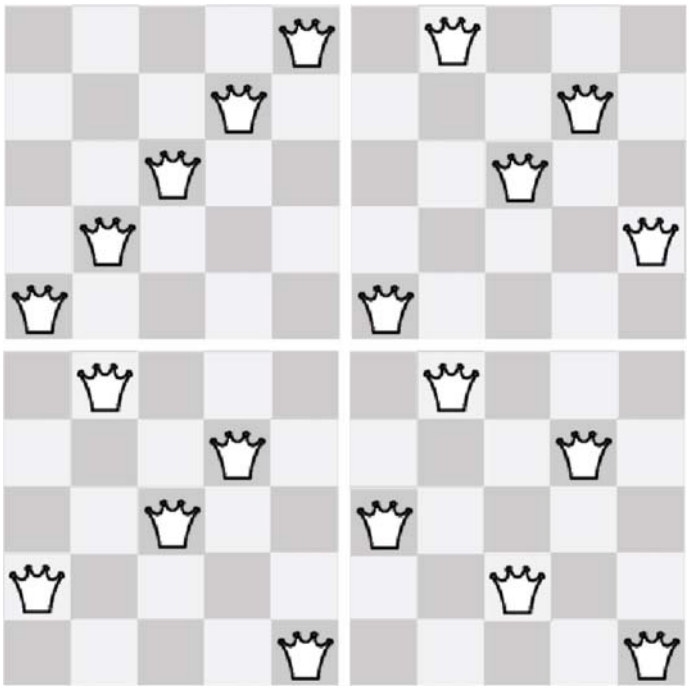


Table 1. Experimental results

Number of queens	Average number of steps	Average number of secs.
10	141.35	0.0171549
20	166.25	0.133275
30	270.9	0.717275
40	272.7	1.71325
50	382.4	4.75144
60	453.85	9.65071
70	495.45	16.9358
80	637.6	33.1815
90	625	47.3944
100	757.6	80.6878
110	745.75	113.635
120	841.75	157.937
130	891.25	216.141
140	983.7	311.71
150	979.75	381.414
160	1093	541.022
170	1145.5	683.763
180	1206.25	872.504
190	1272.256	1089.13
200	1365.25	1423.89

6. CONCLUSION

Due to the high computational cost of classical methods, local search has become an alternative for searching solution to real-life hard problems (Hoos & Stützle, 2004; Bijarbooneh, Flener, & Pearson, 2009).

In this paper we present a first approach to the problem of local search by using Membrane Computing and we have applied it to the N-queens problem as a case study. As a future work, several possibilities arise: One of them is to improve the design from a P system point of view, maybe considering new ingredients; a second one is to consider new case studies closer to real-life problems; a third one is to implement the design in parallel architectures and compare the results with the ones obtained with a one-processor computer.

ACKNOWLEDGMENTS

The authors acknowledge the support of the projects TIN-2009-13192 of the Ministerio de Ciencia e Innovación of Spain and the support of the Project of Excellence of the Junta de Andalucía, grant P08-TIC-04200.

REFERENCES

Bernhardsson, B. (1991). Explicit solutions to the N-queens problem for all. *ACM SIGART Bulletin*, *2*(2), 7. doi:10.1145/122319.122322

Bijarbooneh, F. H., Flener, P., & Pearson, J. (2009). Dynamic demand-capacity balancing for air traffic management using constraint-based local search: First results. In *Proceedings of the 6th International Workshop on Local Search Techniques in Constraint Satisfaction* (Vol. 5, pp. 27-40).

García-Quismondo, M., Gutiérrez-Escudero, R., Pérez-Hurtado, I., Pérez-Jiménez, M. J., & Riscos-Núñez, A. (2009). An overview of P-lingua 2.0. In G. Păun, M. J. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, & A. Salomaa (Eds.), *Proceedings of the 10th International Workshop on Membrane Computing* (LNCS 5957, pp. 264-288).

Gutiérrez-Naranjo, M. A., Martínez-del-Amor, M. A., Pérez-Hurtado, I., & Pérez-Jiménez, M. J. (2009). Solving the N-queens puzzle with P systems. In *Proceedings of the Seventh Brainstorming Week on Membrane Computing*, Seville, Spain (Vol. 1, pp. 199-210).

Gutiérrez-Naranjo, M. A., & Pérez-Jiménez, M. J. (2010). Depth-first search with P systems. In M. Gheorghe, T. Hinze, G. Păun, G. Rozenberg, & A. Salomaa (Eds.), *Proceedings of the Eleventh International Conference on Membrane Computing* (LNCS 6501, pp. 257-264).

Hoffman, E., Loessi, J., & Moore, R. (1969). Constructions for the solution of the N queens problem. *National Mathematics Magazine*, *42*, 66–72.

Hoos, H. H., & Stützle, T. (2004). *Stochastic local search: Foundations & applications* (1st ed.). San Francisco, CA: Morgan Kaufmann.

Păun, G. (2002). *Membrane computing: An introduction*. Berlin, Germany: Springer-Verlag.

Păun, G., Rozenberg, G., & Salomaa, A. (Eds.). (2010). *The Oxford handbook of membrane computing*. Oxford, UK: Oxford University Press.

Pérez-Jiménez, M. J., Romero-Jiménez, Á., & Sancho-Caparrini, F. (2003). Complexity classes in models of cellular computing with membranes. *Natural Computing*, *2*(3), 265–285. doi:10.1023/A:1025449224520

Russell, S. J., & Norvig, P. (2002). *Artificial intelligence: A modern approach* (2nd ed.). Upper Saddle River, NJ: Prentice Hall.

Sosic, R., & Gu, J. (1994). Efficient local search with conflict minimization: A case study of the N-queens problem. *IEEE Transactions on Knowledge and Data Engineering*, *6*(5), 661–668. doi:10.1109/69.317698

ENDNOTES

1. We assume that the reader is familiar with the concepts of Membrane Computing. We refer to Păun (2002) for basic information in this area and to Păun, Rozenberg, and Salomaa (2010) for a comprehensive presentation and the web site http://ppage.psystems.eu for up-to-date information.
2. CLIPS is an expert system tool originally developed by the Software Technology Branch (STB), NASA/Lyndon B. Johnson Space Center, see http://clipsrules.sourceforge.net/.
3. Available from the authors.

Miguel A. Gutiérrez-Naranjo obtained his PhD in Mathematics in 2002. Currently, he is a professor in the Computer Science and Artificial Intelligence Department at the University of Seville, Spain. He is also a member of the Research Group on Natural Computing of the University of Seville. His research interest includes topics related to Artificial Intelligence and Natural Computing. He has coauthored more than 30 scientific papers in these areas.