# Implementing Local Search with Membrane Computing

Miguel A. Gutiérrez-Naranjo, Mario J. Pérez-Jiménez

Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla. Avda. Reina Mercedes s/n, 41012, Sevilla, Spain
magutier@us.es, marper@us.es

**Summary.** Local search is currently one of the most used methods for finding solution in real-life problems. In this paper we 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.

## 1 Introduction

Searching is on the basis of many processes in Artificial Intelligence. The key point is that many real-life problems can be settled as a *space of states*: a *state* is the description of the world in 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 directed graph where the nodes are the states and the edges are the actions is considered. Given a starting 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, where the nodes are the states and the arcs are the transitions, classical search has focused on the order in which the nodes of the tree 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 [4], a first study on depth-first search on the framework of

Membrane Computing was presented. In this paper we go on with the study of searching methods in Membrane Computing by exploring local search.

The paper is organized as follows: First we recall some basic definitions on local search. Then we remember the problem used as a case study: the $N$-queens problem, previously studied in the framework of Membrane Computing in [3]. Next we provide some guidelines of the implementation of local search our case study and 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 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 explore systematically 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 a very little memory (usually constant) is used.
- These algorithms can often find a reasonable solution in 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 to move to one of its neighbors. This movement is not made randomly. In order to decide where to move, in local search a *measure of goodness* is introduced. In this way, the movement is performed towards the best neighbor. It is usual to represent 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. Different situations as *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[1].

[1] Further information can be obtained in [7].

In this paper we will only consider its basic algorithm: 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 back to 1.
        1.2.2.- If it does not exist, we go back to 0
```

## 3 The *N*-queens Problem

Along 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 on putting eight queens on an $8 \times 8$ 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.

In [3], a first solution to the $N$-queens problem in 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 $N$-queens 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 [3] uses a modified solution for SAT from [6]. Some experiments were presented by running the P systems with an updated version of the P-lingua simulator [2]. 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 [3], 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. It took 7 seconds to reach the halting configuration. It is the 117-th configuration and in this configuration one object `No` appears in the environment. As expected, this means that we cannot place three queens on a $3{\times}3$ chessboard satisfying the restriction of the problem.

In the 4-queens problem, we try to place four queens on a $4{\times}4$ 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`

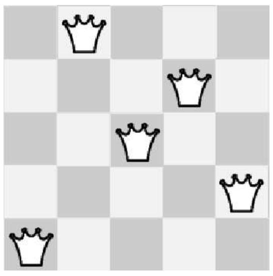

**Fig. 1.** Five queens on a board

appears in the environment. This configuration has two elementary membranes encoding the two solutions of the problem (see [3] for details).

In [4], a study of depth-first search in Membrane Computing was presented. The case study was also the $N$-queens problem. An *ad hoc* CLIPS program was written based on 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 $4 \times 4$ board and 15,944 seconds for a $20 \times 20$ board.

## 4 A P system Family for Local Search

In this section we give an sketch of the design of a P system family which solves the $N$-queens problem by using local search, $\mathbf{\Pi} = \{\Pi(N)\}_{N \in \mathbb{N}}$. Each P systems $\Pi(N)$ 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.

One *state* is represented by a $N \times N$ chess board where $N$ queens have been placed. In order to limit the number of possible states, we will 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 if a board is a solution to the problem or not.

These boards can be easily represented with P systems. For $\Pi(N)$, we consider a membrane structure which contains $N$ elementary membranes and $N$ objects $y_i$, $i \in \{1, \dots, N\}$ in the skin. By using rules of type $y_i\,[\,]_j \to [y_i]_j$ the objects $y_i$ are non-deterministically sent into the membranes and the object $y_i$ inside a membrane with label $j$ will be 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*

[8]: *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 from the 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 of 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 in $(k, s)$, 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 with these definitions, the local search algorithm for the $N$-queens problem can be settled as follows:

```
0.- We start with a state randomly chosen
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 a movement which reaches a state which decrease the
number of collisions.
        1.2.1.- If it exists, we randomly choose one of the possible movements.
                The reached state becomes the current state and we back to 1.
        1.2.2.- If it does not exist, we go back to 0
```

At this point, three basic question 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 a board $N \times N$ is made by using $N$ elementary membranes where the objects $y_1, \dots, y_N$ are placed. These $N$ elementary membranes, with labels $1, \dots, N$ are not the unique elementary membranes in the membrane structure. As pointed above, in an $N \times N$ board, there are $2N-1$ ascendant and $2N-1$ descendant diagonals. We will denote the ascendant diagonals as $u_{-N+1}, \dots, u_{N-1}$, 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 $d_2, \dots, d_{N+N}$ 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 $1, \dots, N$ for encoding the board, we will also place $4N - 2$ elementary membranes in the structure, with labels $u_{-N+1}, \ldots, u_{N-1}, d_2, \ldots, d_{N+N}$. 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 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 of $i$ of $z_i$ denotes how may 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 the number of queens in each diagonal. In our design, this is not necessary. As we will see, we only need no 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 the number of queens.

Technically, after 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 on 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 queens 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 with 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 if a diagonal is ready to receive queens (0 queens) or it needs to send queens to another diagonal, 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.

Firstly, let us consider two queens placed on the squares $(i, j)$ and $(k, s)$ on 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 in $(i, s)$ and $(k, j)$ improves the board.

In order to answer this question we will consider:

- $u_{i-j}(0, 0, UC_{i-j}, UD_{i-j})$: The ascendant diagonal $u_{i-j}$ has at least 2 queens, so the two first coordinates are 0.
- $d_{i+j}(0, DB_{i+j}, DC_{i+j}, DD_{i+j})$ and $d_{k+s}(0, DB_{k+s}, DC_{k+s}, DD_{k+s})$: The descendent diagonals $d_{i+j}$ and $d_{k+s}$ have at least 1 queen, so the we first coordinate is 0.

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

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

Analogously, in order to compute the augmentation in the number of collisions produced by the placement of two queens in the squares $(i, s)$, $(k, j)$ we will consider the objects $d_{i+s}(DA_{i+s}, DB_{i+s}, DC_{i+s}, DD_{i+s})$, $u_{i-s}(UA_{i-s}, UB_{i-s}, UC_{i-s}, UD_{i-s})$ 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 is we have an improvement, 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 analogous.

If the queens do not share 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*

$$\begin{array}{ll}
u_{i-j}(UA_{i-j}, UB_{i-j}, UC_{i-j}, UD_{i-j}) & d_{i+j}(DA_{i+j}, DB_{i+j}, DC_{i+j}, DD_{i+j}) \\
u_{k-s}(UA_{k-s}, UB_{k-s}, UC_{k-s}, UD_{k-s}) & d_{k+s}(DA_{k+s}, DB_{k+s}, DC_{k+s}, DD_{k+s}) \\
u_{i-s}(UA_{i-s}, UB_{i-s}, UC_{i-s}, UD_{i-s}) & d_{i+s}(DA_{i+s}, DB_{i+s}, DC_{i+s}, DD_{i+s}) \\
u_{k-s}(UA_{k-s}, UB_{k-s}, UC_{k-s}, UD_{k-s}) & d_{k+j}(DA_{k+j}, DB_{k+j}, DC_{k+j}, DD_{k+j})
\end{array}$$

*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 choose one of them in a non-deterministic way. The application of the rule introduces an object $change_{ijks}$ 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 full of technical details. 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 (produces a configuration equivalent to the initial one) if no more improvements can be achieved from the current configuration.

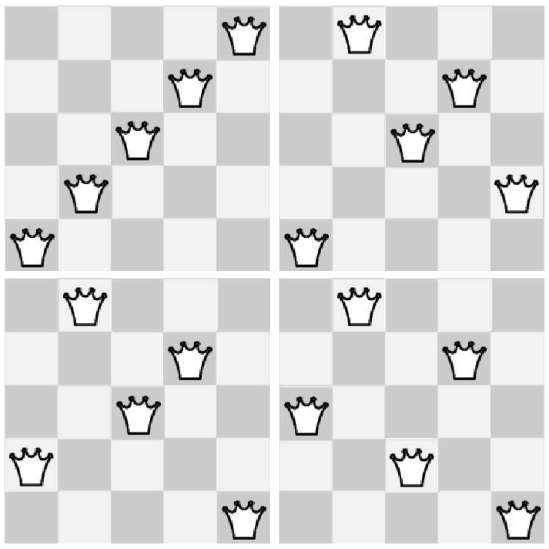

**Fig. 2.** Starting from a configuration $C_0$ with 4 collisions (up-left) we can reach $C_1$ with 3 collisions (up-right) and then $C_2$ with 2 collisions (bottom-left) and finally $C_3$ with 0 collisions (bottom-right), which is a solution to the 5-queens problem.

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 (up-left). The collisions in this diagonal (and in the whole board) is 4. By changing the queens from the columns 2 and 5, we obtain the board shown in Figure 1 with 3 collisions (up-right in Figure 2). In the next step, the queens form 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 the queens is the columns 1 and 3, we get a board with no collisions that represent a solution to the 5-queens problem (bottom-right).

## 5 Experimental Results

An *ad hoc* CLIPS program was written *inspired* on this 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.

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 of queens $N$ 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. The following table shows the result of the experiments.

| Number of queens | Average of steps | Average 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 |

Notice, for example, that in the solution presented in [4], the solution for 20 queens was obtained after 15,944 seconds. The average time obtained with this approach is 0.133275 seconds.

## 6 Final Remarks

Due to the high computational cost of classical methods, local search has became an alternative for searching solution to real-life hard problems [1, 5].

In this paper we present a first approach to the problem of local search by using Membrane Computing and we have applied to the N-queens problem as a case study. As 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 obtained ones with an one-processor computer.

**Acknowledgements**

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

1. Bijarbooneh, F.H., Flener, P., Pearson, J.: Dynamic demand-capacity balancing for air traffic management using constraint-based local search: First results. In: Deville, Y., Solnon, C. (eds.) LSCS. EPTCS, vol. 5, pp. 27–40 (2009)
2. García-Quismondo, M., Gutiérrez-Escudero, R., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Riscos-Núñez, A.: An overview of P-lingua 2.0. In: Păun, Gh., Pérez-Jiménez, M.J., Riscos-Núñez, A., Rozenberg, G., Salomaa, A. (eds.) Workshop on Membrane Computing. Lecture Notes in Computer Science, vol. 5957, pp. 264–288. Springer (2009)
3. Gutiérrez-Naranjo, M.A., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Solving the N-queens puzzle with P systems. In: Gutiérrez-Escudero, R., Gutiérrez-Naranjo, M.A., Păun, Gh., Pérez-Hurtado, I., Riscos-Núñez, A. (eds.) Seventh Brainstorming Week on Membrane Computing. vol. I, pp. 199–210. Fénix Editora, Sevilla, Spain (2009)
4. Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J.: Depth-first search with P systems. In: Gheorghe, M., Hinze, T., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Int. Conf. on Membrane Computing. Lecture Notes in Computer Science, vol. 6501, pp. 257–264. Springer (2010)
5. Hoos, H.H., Stützle, T.: Stochastic Local Search : Foundations & Applications (The Morgan Kaufmann Series in Artificial Intelligence). Morgan Kaufmann, 1 edn. (Sep 2004)
6. Pérez-Jiménez, M.J., Romero-Jiménez, Á., Sancho-Caparrini, F.: Complexity classes in models of cellular computing with membranes. Natural Computing 2(3), 265–285 (2003)
7. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach (2nd Edition). Prentice Hall (December 2002)
8. Sosic, R., Gu, J.: 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 (1994)