A Membrane Algorithm for the Min Storage Problem

Alberto Leporati, Dario Pagani [⋆]

Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano – Bicocca
Via Bicocca degli Arcimboldi 8, 20126 Milano, Italy

e-mail: leporati@disco.unimib.it
dario.pagani@gmail.com

Abstract. Min Storage is an NP–hard optimization problem that arises in a natural way when one considers computations in which the amount of energy provided with the input values is preserved during the computation. In this paper we propose a polynomial time membrane algorithm that computes approximate solutions to the instances of Min Storage, and we study its behavior on (almost) uniformly randomly chosen instances. Moreover, we compare the (estimated) coefficient of approximation of this algorithm with the ones obtained from several other polynomial time heuristics. The result of this comparison indicates the superiority of the membrane algorithm with respect to many other traditional approximation techniques.

1 Preliminaries

Membrane systems (also known as *P systems*) were introduced in [7] as a new class of distributed and parallel computing devices, inspired by the structure and functioning of living cells. The basic model consists of a hierarchical structure composed by several membranes, embedded into a main membrane called the *skin*. Membranes divide the Euclidean space into *regions*, that contain some *objects* (represented by symbols of an alphabet) and *evolution rules*. Using these rules, the objects may evolve and/or move from a region to a neighboring one. The rules are applied in a nondeterministic and maximally parallel way: all the objects that may evolve are forced to evolve. A *computation* starts from an initial configuration of the system and terminates when no evolution rule can be applied. The result of a computation is the multiset of objects contained into an *output membrane* or emitted from the skin of the system.

In what follows we assume the reader is already familiar with the basic notions and the terminology underlying P systems. For details, and a systematic introduction on the subject, we refer the reader to [9]. The latest information about P systems can be found in [11].

[⋆] This research was supported by the European Research Training Network Segravis.

In [6], Nishida has proposed a new type of approximation algorithms for optimization problems, named *membrane algorithms*. A membrane algorithm operates on a particular type of P system, which is a linear collection of separated regions determined by nested membranes. Each region contains a number of candidate solutions, and a local optimization algorithm. At each computation step, the local optimization algorithms which occur in the system are concurrently executed on the currently available solutions. New candidate solutions are produced in each region as a result; the best and the worst of them are sent to the immediately inner and immediately outer region, respectively. By repeating this process, a good solution will likely appear in the innermost region after a suitable number of computation steps. The algorithm terminates after a prefixed number of iterations has been performed, or some halting condition is verified (such as, for example, the solution(s) of the innermost region is (resp., are) not changed for a predetermined number of steps). In [6], a membrane algorithm which computes approximate solutions to the instances of the Travelling Salesman Problem (TSP) is described; moreover, the results of some computer experiments are presented, showing that this algorithm is indeed a good approximation heuristic for TSP.

In this paper we elaborate after Nishida's membrane algorithm for TSP, and we propose a new membrane algorithm for another NP–hard optimization problem, Min Storage [5]. This problem arises in a natural way if we consider *conservative computations*, that is, computations in which the amount of energy associated with the input values is first preserved during the computation of the output values, and then it is completely returned with them. In [5], it has been proved that Min Storage is strongly NP–hard, and that it is 2–approximable. This means that there exists a polynomial time algorithm that, for every instance $\mathcal{E}$ of Min Storage, returns a feasible solution $\text{sol}(\mathcal{E})$ which is at most the double of the optimal solution $\text{opt}(\mathcal{E})$. In this paper we present the results of some computer experiments in which the new membrane algorithm for Min Storage is compared with several "classical" polynomial time heuristics. As we will see, the membrane algorithm performs considerably well (with an estimated coefficient of approximation which is well under 2), especially when the number of membranes and the number of iterations in the algorithm are sufficiently large (where the term "sufficiently" has been experimentally determined).

The paper is organized as follows. In section 2 we recall the definition of Min Storage and some of its properties. In section 3 we present a membrane algorithm to solve the problem. Precisely, we propose two versions: MA4MS (Membrane Algorithm for Min Storage) which uses a particular kind of crossover and mutation to produce new candidate solutions, and MA4MS LS, which uses only a simple local search. In section 4 we discuss a method that allows to generate random instances of Min Storage with an (almost) uniform distribution of probability. Section 5 illustrates the results of some computer experiments which have been performed on MA4MS and MA4MS LS. As we will see, these results led us to abandon MA4MS and to use only the version with the local search for further investigation. In section 6 we describe several "classical" polynomial time heuristics for MIN STORAGE. Section 7 illustrates the results of other computer experiments, which have been performed to compare the performance (in terms of average coefficient of approximation) of our membrane algorithm against the above classical heuristics. Section 8 concludes the paper.

2 The Problem

Let us first introduce the problem we want to solve. As stated in the Introduction, this problem comes from the study of *conservative* (that is, *energy preserving*) computations. We refer the interested reader to [5] for two possible interpretations of the problem in this setting.

Let $\mathcal{E} = \langle e_1, e_2, \ldots, e_k \rangle$ be a finite sequence of integer numbers. For a fixed $i \in \{1, 2, \ldots, k\}$, the *i-th prefix sum of* $\mathcal{E}$ is the value $\sum_{j=1}^{i} e_j$. Let C be a positive integer; we say that $\mathcal{E}$ is *C–feasible* if for each $i \in \{1, 2, \ldots, k\}$ the i-th prefix sum of $\mathcal{E}$ is in the closed interval $[0, C]$.

Problem 1. NAME: CONSCOMP.

- INSTANCE: a set $\mathcal{E} = \{e_1, e_2, \ldots, e_k\}$ of integer numbers such that $e_1 + e_2 + \ldots + e_k = 0$, and an integer number $C > 0$.
- QUESTION: is there a permutation $\pi \in S_k$ (the symmetric group of order k) such that the sequence $e_{\pi(1)}, e_{\pi(2)}, \ldots, e_{\pi(k)}$ is C–feasible? □

The fact that the resulting sequence $e_{\pi(1)}, e_{\pi(2)}, \ldots, e_{\pi(k)}$ is C–feasible can be explicitly written as:

$$0 \leq \sum_{j=1}^{i} e_{\pi(j)} \leq C \qquad\qquad \forall\, i \in \{1, 2, \ldots, k\}$$

The following theorem, which has been proved in [5], shows that it is very unlikely that a polynomial time algorithm exists that correctly classifies every instance of CONSCOMP as positive or negative.

Theorem 1. CONSCOMP *is* NP*–complete.*

The CONSCOMP problem naturally leads to the formulation of the following optimization problem.

Problem 2. NAME: MIN STORAGE.

- INSTANCE: a set $\mathcal{E} = \{e_1, e_2, \ldots, e_k\}$ of integer numbers such that $e_1 + e_2 + \ldots + e_k = 0$.
- SOLUTION: a permutation $\pi \in S_k$ such that $\sum_{j=1}^{i} e_{\pi(j)} \geq 0$ for each $i \in \{1, 2, \ldots, k\}$.
- MEASURE: $\max\limits_{1 \leq i \leq k} \sum_{j=1}^{i} e_{\pi(j)}$. □

Informally, the output of Min Storage is the minimum value of C for which there exists a permutation $\pi \in S_k$ such that the sequence $e_{\pi(1)}, e_{\pi(2)}, \ldots, e_{\pi(k)}$ is C–feasible. Notice that a trivial upper bound for the value of C is:

$$\sum_{i \in \{1,2,\ldots,k\} \,:\, e_i > 0} e_i = \frac{1}{2} \sum_{i=1}^{k} |e_i|$$

while a trivial lower bound is $\max_{1 \leq i \leq k} |e_i|$.

It is immediately seen that Min Storage is in the class NPO [1, page 27]. In fact, checking whether some given integers $e_1, e_2, \ldots, e_k$ sum up to zero can be trivially done in polynomial time; each feasible solution has linear length and besides it can be verified in polynomial time whether a given permutation $\pi \in S_k$ is a feasible solution; finally, the measure function can be computed in polynomial time. Since the underlying decision problem ConsComp is NP–complete, we can immediately conclude that Min Storage is NP–hard [1, page 30]. Just like the ConsComp decision problem, this means that it is very unlikely that a polynomial time algorithm exists that gives the correct solution to every instance of Min Storage.

Since the Min Storage problem is NP–hard, a natural question is how well its optimal solutions can be approximated in polynomial time. Precisely, we ask ourselves whether there exists a PTAS (Polynomial Time Approximation Scheme) or even a FPTAS (Fully Polynomial Time Approximation Scheme) for Min Storage. Concerning these questions, in [5] it has been proved that ConsComp is NP–complete in the strong sense, by showing a polynomial reduction from 3–Partition [2, page 224], a well known strongly NP–complete problem. As a consequence, Min Storage is strongly NP–hard, and thus it doesn't admit a FPTAS [1, page 116]. The next natural question is whether there exists a PTAS for Min Storage; this possibility is currently under investigation.

In [5] it has also been proved that the algorithm shown in Figure 1 is a 2–approximation algorithm for Min Storage. The proof derives from the fact that, denoted by M the value $\max_{1 \leq i \leq k} |e_i|$, the variable *st* (that records the energy currently stored into the system) assumes values only from the interval $[0, 2M - 1]$. The variable *max*, which contains the value returned at the end of the computation, records the maximum of the values assumed by *st* into the subinterval $[M, 2M - 1]$. Since the optimal solution cannot be less than M, the value returned by the algorithm is at most the double of the optimal solution. A direct inspection of the pseudo–code reveals that the time complexity of the algorithm is linear with respect to k, the length of the input sequence. Hence, Min Storage is in the class APX of problems which admit a constant factor polynomial time approximation algorithm.

3 A Membrane Algorithm for Min Storage

Let us now introduce a membrane algorithm that produces approximate solutions to any instance of Min Storage. As stated in the Introduction, the

```
APPROX MIN STORAGE($\mathcal{E}$)
$M \leftarrow \max_{1 \le i \le k} |e_i|$
$E_p = E_n = \emptyset$
for $i \leftarrow 1$ to $k$
    do if $e_i \ge 0$
          then $E_p = E_p \cup \{e_i\}$
          else $E_n = E_n \cup \{e_i\}$
$max \leftarrow st \leftarrow 0$
while $E_p \neq \emptyset$
    do if $st < M$
          then $x \leftarrow$ an element of $E_p$
               $st \leftarrow st + x$
               if $st > max$ then $max \leftarrow st$
               $E_p = E_p \setminus \{x\}$
          else $x \leftarrow$ an element of $E_n$
               $st \leftarrow st + x$
               $E_n = E_n \setminus \{x\}$
return $max$
```

Fig. 1. Pseudocode of a 2–approximation algorithm for MIN STORAGE

algorithm is based on a structure composed by nested membranes. Each of the regions determined by the membranes contains a certain number of candidate solutions, and a local optimization algorithm. Formally, let m be the number of nested membranes, and let 0 and $m-1$ be the innermost and the outermost regions, respectively. Just like in the membrane algorithm for TSP, proposed by Nishida in [6], we put one candidate solution in region 0 and two candidate solutions in the other regions.

Another fundamental component of the system is the *transport* mechanism, that allows candidate solutions to move to the immediately inner or to the immediately outer region. The idea underlying the algorithm is to move good solutions towards the innermost region, and bad solutions towards the outermost region. When the computation halts, the best candidate solution produced by the system is thus contained into the innermost region, which is by definition the region in which the output is observed at the end of the computation.

A first difficulty in adapting the TSP membrane algorithm proposed by Nishida to the MIN STORAGE problem is that, differently from TSP, not all the permutations of the elements $\mathcal{E} = \{e_1, e_2, \ldots, e_k\}$ given in the instance give rise to feasible solutions. This is due to the fact that in a feasible solution π of MIN STORAGE all prefix sums are non negative; clearly, this property is not preserved if we exchange two randomly chosen elements of the solution. To overcome this difficulty, we have slightly modified the measure function associated with MIN STORAGE as follows:

$$F(\pi) = \begin{cases} \max\limits_{1 \le i \le k} \sum\limits_{j=1}^{i} e_{\pi(j)} & \text{if } \sum_{j=1}^{i} e_{\pi(j)} \ge 0 \text{ for all } i \in \{1, \ldots, k\} \\ \sum\limits_{i=1}^{k} |e_i| - \mathrm{NumVPS}_\pi & \text{otherwise} \end{cases}$$

where NumVPS_π is the number of non negative (that is, valid) prefix sums determined by π. In this way, all feasible solutions get a lower measure with respect to non feasible solutions. Moreover, every permutation can be measured, and we can also choose what among two non feasible solutions to prefer: the one which has the lowest number of negative prefix sums. As an alternative approach, we could impose to work only with feasible solutions (discarding non feasible ones when they appear), and adopt the usual measure function for Min Storage. However, since the probability to generate a non feasible solution is very high, this approach has been considered infeasible from a computational point of view.

The structure of the algorithm is analogous to the one proposed by Nishida for TSP. Given an instance $\mathcal{E}$ of Min Storage, the algorithm works as follows:

1. put one random solution in region 0, and two random solutions in every region from 1 to $m-1$;
2. repeat the following steps d times:
   (a) in each region, apply the local optimization algorithm to produce new candidate solutions;
   (b) for every region $i \in \{1, 2, \ldots, m-1\}$, send the best among the solutions contained in the region (both old and new) to region $i-1$ (that is, towards the interior of the system). Similarly, for all $i \in \{0, 1, \ldots, m-2\}$ send the worst solution of region i to region $i+1$;
   (c) in each region $i \in \{1, 2, \ldots, m-1\}$, remove all solutions but the best two. In region 0, remove all solutions but the best one;
3. return the solution contained in region 0 as the output of the algorithm.

With respect to the membrane algorithm for TSP, we have used different local optimization algorithms. Precisely, for region 0 we have used a kind of local search: given a solution π, this operation explores the solutions which can be found in its neighborhood (which depends upon a specified element of π); if one of such solutions has a better measure than π, then it substitutes π. Formally, the neighborhood of π is defined as follows.

Definition 1. *Let $\pi \in S_k$ be a candidate solution, and let $\alpha \in \{1, 2, \ldots, k\}$. The* neighborhood *$Neigh(\pi, \alpha)$ of π, with respect to position α, is the set of $k-1$ solutions defined as follows:*

$$Neigh(\pi, \alpha) = \bigcup_{i \neq \alpha} \{\pi_{i,\alpha}\}$$

where $\pi_{i,\alpha}$ is the solution obtained from π by exchanging the elements in positions i and α.

The local search in region 0 is thus executed as follows:

```
LocalSearch4MS($\pi, \alpha$)

$Best \leftarrow \pi$
$min \leftarrow F(\pi)$
for $i \leftarrow 1$ to $k-1$ do
    $\pi' \leftarrow$ select an element from $Neigh(\pi, \alpha)$
    if $F(\pi') < min$
```

```
        then $min \leftarrow F(\pi')$
             $Best \leftarrow \pi'$
    $Neigh(\pi, \alpha) = Neigh(\pi, \alpha) \setminus \{\pi'\}$
return $Best$
```

In order to improve the probability to generate feasible solutions, the position α with respect to which we build the neighborhood of π is chosen as the first position for which the corresponding prefix sum is negative; if all prefix sums are non negative, then α is chosen at random in the set $\{1, 2, \ldots, k\}$.

In a first version of our membrane algorithm, that we have called MA4MS (Membrane Algorithm for Min Storage), we have tried to use a kind of *crossover* operation between candidate solutions as a local optimization algorithm for regions $1, 2, \ldots, m-1$. The idea, very well known in the domain of *genetic algorithms*, is to start from two solutions A and B, cut them in the same position and then recombine them as shown on the left of Figure 2. However, if we apply this operation to permutations, it is very likely that we obtain sequences in which some elements are missing and some are repeated; that is, in general we do not obtain two permutations as a result. Hence we have tried to use a variant of the standard crossover operation, named *partially matched crossover* (or PMX, for short) [4]. Just like the standard crossover, PMX operates on two permutations, say π and π'. This time, however, the two permutations are not recombined; rather, two "cutpoints" are randomly selected (let us call them l and r, respectively, with $l \leq r$) and then the elements of π are permuted according to positions $\pi'(l), \pi'(l+1), \ldots, \pi'(r)$. Similarly, the elements of π' are permuted according to positions $\pi(l), \pi(l+1), \ldots, \pi(r)$. As an example, let us assume that $\pi = \langle 2,3,1,7,4,6,5 \rangle$, $\pi' = \langle 5,1,7,6,4,2,3 \rangle$, $l = 2$ and $r = 5$. This situation is depicted on the right of Figure 2. Now, in both permutations π and π' the elements 3 and 1 are exchanged, then the elements 1 and 7 are exchanged, and so on. The pseudocode of the PMX operation is the following:

```
PMX($\pi, \pi'$)

$l, r \leftarrow$ random($1, \ldots, k$)
if $l > r$ then exchange $l$ and $r$
for $i \leftarrow l$ to $r$ do
    find the position $j \in \{1, 2, \ldots, k\}$ such that $\pi(j) = \pi'(i)$
    find the position $j' \in \{1, 2, \ldots, k\}$ such that $\pi'(j') = \pi(i)$
    exchange $\pi(i)$ and $\pi(j)$
    exchange $\pi'(i)$ and $\pi'(j')$
return $\pi, \pi'$
```

Once two new solutions have been generated using the PMX operation, a "mutation" operation is applied on each of them with the probability $p = \frac{i}{m}$, which is directly proportional to the depth of the region into the system. This means, in particular, that the mutation is never performed in the innermost

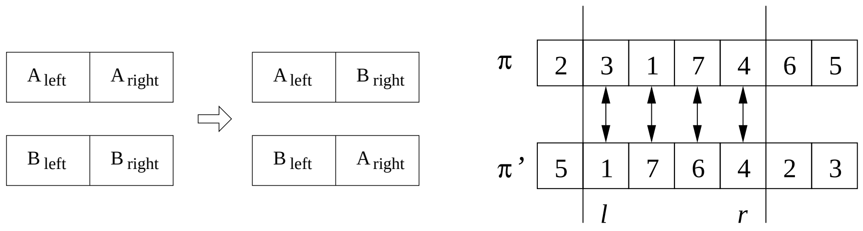

Fig. 2. The standard crossover operation (left) and the first step in partially matched crossover (right)

region, whereas it is almost always applied in the outermost region. This operation simply chooses two positions in a random way (according to a uniform probability distribution) and then exchanges the elements in such positions.

As we will see later, we have also considered a second version of the above membrane algorithm, in which no PMXs and no mutations are performed. Instead, LocalSearch4MS is used as the local optimization algorithm in every membrane of the system. We have called such variant MA4MS LS.

4 Generating Random Instances of Min Storage

We have performed some computer experiments on randomly chosen instances of Min Storage, in order to study the behavior of our membrane algorithm. All the random choices made during the experiments were performed according to the discrete uniform distribution. Hence the first problem we faced was to generate random instances for Min Storage in a uniform way. Formally, we can state the problem as follows.

Problem 3. Let $e_1, e_2, \ldots, e_k$ be independent variables uniformly distributed over the set of integers from the interval $[-M, M]$: how can we extract in a uniform way those k–tuples for which $e_1 + e_2 + \ldots + e_k = 0$? □

A possible solution to this problem could be to extract each element e_i and to discard the entire k–tuple if the sum is not zero; however the probability of success, Prob $\left[\sum_{i=1}^{k} e_i = 0\right]$, is fairly small. In order to compute such probability we observe that the distribution of the sum of k discrete independent uniformly distributed random variables is a k–th order convolution. Hence, the evaluation of the probability of success amounts to compute how many k–tuples with elements in $[-M, M]$ whose sum is zero we can build. To the best knowledge of the authors, this calculation seems to require the examination of an exponential number of k–tuples, and thus it is not feasible. As a consequence, we can compute an estimate of the probability of success by approximating the distribution of the random variable $Y = e_1 + e_2 + \ldots + e_k$ with an appropriate normal distribution.

First of all, let us compute the mean and variance of each random variable e_i. Since e_i is uniformly distributed over the interval $[-M, M]$ of integers, it holds:

$$\mathrm{E}\left[e_i\right] = \sum_{x=-M}^{M} x \cdot \frac{1}{2M+1} = \frac{1}{2M+1} \sum_{x=-M}^{M} x = 0$$

and

$$\begin{aligned}\mathrm{var}\left[e_i\right] &= \mathrm{E}\left[e_i^2\right] - \left(\mathrm{E}\left[e_i\right]\right)^2 = \mathrm{E}\left[e_i^2\right] \\ &= \sum_{x=-M}^{M} x^2 \cdot \frac{1}{2M+1} = \frac{2}{2M+1} \sum_{x=1}^{M} x^2 = \frac{M(M+1)}{3}\end{aligned}$$

For linearity we obtain:

$$\mathrm{E}\left[Y\right] = \sum_{i=1}^{k} \mathrm{E}\left[e_i\right] = 0$$

Since $e_1, e_2, \ldots, e_k$ are independent variables, the variance of their sum is the sum of their variances, hence:

$$\mathrm{var}\left[Y\right] = \sum_{i=1}^{k} \mathrm{var}\left[e_i\right] = k \cdot \frac{M(M+1)}{3}$$

A direct consequence of the Central Limit theorem is that we can approximate the distribution of Y with the normal distribution $N\left(0, k\frac{M(M+1)}{3}\right)$ having the same mean and variance. In our experiments we have considered $k = 100$ and $M = 10^6$; this means that $\mathrm{var}\left[Y\right] \approx 10^{14}$, and the probability of success is:

$$\mathrm{Prob}\left[Y = 0\right] \approx 6.91 \cdot 10^{-8}$$

As stated above, this is a very small value. On the other hand, let us notice that there is a bijective correspondence between the set of all k–tuples whose sum is zero and the set of all $(k-1)$–tuples whose sum is in the interval $[-M, M]$. This observation suggests that we could extract $k-1$ elements in a uniform way and check whether their sum is in the interval $[-M, M]$. If this is the case then we put $e_k = -\sum_{i=1}^{k-1} e_i$, thus producing an instance; otherwise, we discard the entire $(k-1)$–tuple and we try with a different set of $k-1$ elements. Now the probability of successfully generate an instance is $\mathrm{Prob}\left[-M \leq \sum_{i=1}^{k-1} e_i \leq M\right]$. Once again, we can approximate the distribution of the random variable $Z = \sum_{i=1}^{k-1} e_i$ with the normal distribution $N\left(0, (k-1)\frac{M(M+1)}{3}\right)$. Thus, if

$$\phi(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2x^2}\right) \qquad \text{and} \qquad \Phi(x) = \int_{-\infty}^{x} \phi(u)\,\mathrm{d}u$$

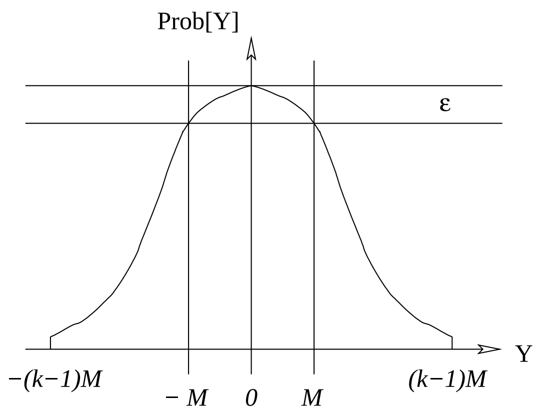

Fig. 3. Gaussian approximation of the distribution of $Z = \sum_{i=1}^{k-1} e_i$

are the probability density function and the cumulative distribution function of the standardized normal distribution $N(0, 1)$, it holds:

$$\mathrm{Prob}\left[-M \leq \sum_{i=1}^{k-1} e_i \leq M\right] \approx \Phi\left(\frac{M}{\sigma}\right) - \Phi\left(-\frac{M}{\sigma}\right) = 2\Phi\left(\frac{M}{\sigma}\right) - 1$$

where $\sigma = \sqrt{(k-1)\frac{M(M+1)}{3}}$. For $k = 100$ and $M = 10^6$ we obtain:

$$\mathrm{Prob}\left[-10^6 \leq \sum_{i=1}^{99} e_i \leq 10^6\right] \approx 0.138 \tag{1}$$

Intuitively, for fixed values of M and k we approximate the real distribution of Z (that can assume every integer value in the interval $[-(k-1)M, (k-1)M]$) with a normal distribution (see Figure 3), and we consider the portion of the curve contained into the vertical strip included between $-M$ and M. For growing values of k, such strip becomes small with respect to the entire curve, and thus the portion of the curve into the strip tends to become an horizontal segment. This means that we find k–tuples whose sum is zero with almost a uniform distribution. We can estimate the error due to the fact that the portion of curve into the strip is not horizontal by looking at the difference between the higher and the lower values it assumes in this interval:

$$\varepsilon = \mathrm{Prob}\left[\sum_{i=1}^{k-1} e_i = 0\right] - \mathrm{Prob}\left[\sum_{i=1}^{k-1} e_i = M\right]$$

For $k = 100$ and $M = 10^6$, the error is $\varepsilon \approx 1.04 \cdot 10^{-9}$. As a consequence, we can safely assume that our strategy produces k–tuples whose sum is equal to zero with a uniform probability distribution; moreover, as stated in (1), about 13.8%

of the times it will produce one of such k–tuples. A computer experiment has confirmed this last result.

Before looking at the experiments, let us recall the notion of coefficient of approximation. Let $c_A(\mathcal{E})$ be the value which is returned as a solution by a heuristic A for the instance $\mathcal{E}$ of the MIN STORAGE problem, and let $opt(\mathcal{E})$ be the optimal solution, that is, the value returned by the brute force algorithm that examines all possible feasible solutions. Then, the *coefficient of approximation* of algorithm A over the instance $\mathcal{E}$ is the value $app_A(\mathcal{E})$, where

$$app_A(\mathcal{E}) = \frac{|c_A(\mathcal{E})|}{opt(\mathcal{E})} \tag{2}$$

Note that $app_A(\mathcal{E})$ is always greater than or equal to 1, and that the closer it is to 1, the better the approximate solution is. We say that algorithm A has the *guaranteed coefficient of approximation* c if $app_A(\mathcal{E}) \leq c$, for every instance $\mathcal{E}$. For example, APPROX MIN STORAGE has a guaranteed coefficient of approximation equal to 2.

5 First Experiments with MA4MS

We have implemented the membrane algorithm MA4MS in the JAVA programming language. To simulate the parallel application of local optimization algorithms we have implemented them as *threads*, with a monitor that allows to synchronize the exchange of information between the regions of the system.

Then, we have performed some computer experiments to study the behavior of MA4MS on randomly chosen instances. To measure the performance of the algorithm we have computed an estimate of its coefficient of approximation (see also equation (2)), averaged on the number N of instances considered in the experiment:

$$app_{\mathrm{MA4MS}} = \frac{1}{N} \sum_{i=1}^{N} \frac{F_i(\pi)}{opt_i}$$

where opt_i has been put equal to the optimal solution of the i-th instance in those experiments for which the length of the instances allowed to compute it. In the experiments for which the length of the instances did not allow to compute the optimal solution with the brute force approach, we have substituted it with the theoretical lower bound $\max_{1 \leq i \leq k} |e_i|$.

In the first experiment we have tested the behavior of MA4MS by running 10000 tests, each with randomly generated instances of increasing length (10, 20, 50 and 100). The number m of regions and the number d of iterations have been put equal to 10 and 50, respectively. The results are illustrated in Figure 4 (on the left). As we can see, the average coefficient of approximation grows together with k, the length of the instances, going well above the value 2 given by APPROX MIN STORAGE. This is probably due to the fact that the partially matched crossover is not able to differentiate the solutions initially assigned to the system. Indeed, with PMX, solutions that differ in a small number of

k	app_{MA4MS}	VARIANCE
10	1.2052383	0.0433753
20	1.7645406	0.1412564
50	3.0863457	0.6402221
100	4.6576763	1.8045398

k	app_{MA4MS}	VARIANCE
10	1.0875901	0.0139737
20	1.5258556	0.0582978
50	2.6124590	0.2444580
100	3.9430665	0.5004649

Fig. 4. Results obtained for MA4MS on 10000 tests, with $m = 10$ and $d = 50$ (on the left) and with $m = 30$ and $d = 150$ (on the right), for different lengths k of the instances

positions produce new solutions which are similar. As we can see on the right of Figure 4, this problem remains even if we raise the parameters m and d to 30 and 150, respectively.

For these reasons, we have abandoned our first version of the membrane algorithm and we have repeated the above experiments with the second version, MA4MS LS, in which LOCALSEARCH4MS is used as a local optimization algorithm in all regions, instead of PMX and mutations. In Figure 5 (left) we can see the results of the second experiment described above, with $m = 30$ and $d = 150$.

k	app_{MA4MS}	VARIANCE
10	1.0032719	0.0002333
20	1.0093292	0.0003654
50	1.0600094	0.0045419
100	1.1978124	0.0162296

m	d	app_{MA4MS}
10	20	2.1003992
30	60	1.4055032
50	100	1.2228035
80	150	1.1003962
150	200	1.1066577
300	500	1.0214561

Fig. 5. Results obtained with MA4MS LS, on: (left) 10000 tests, with $m = 30$ and $d = 150$, for different lengths k of the instances; (right) groups of 10 instances of length 100, and growing values of m and d

It is apparent that MA4MS LS obtains better results, and thus we will use it in the following to perform some comparisons with some “classical” heuristics specially crafted for MIN STORAGE. Let us note that, in this new version of the algorithm, the only “forces” that drive to a good solution are local search and the transport mechanism, that moves good solutions towards region 0 and bad solutions towards region $m - 1$. No other forces (crossover, mutations, etc.) are involved, and hence it is our opinion that this is a “true” membrane algorithm, in the original spirit of Membrane Computing.

Some computer experiments have also been performed to see how the average coefficient of approximation is affected by the number m of regions and the number d of iterations. Figure 5 (on the right) shows some results obtained on groups of 10 instances, each containing 100 elements, for growing values of m and d. Notice that $d \geq m$, so that a good solution has always the possibility to reach the innermost region. Figure 6 contains two plots of the average coefficient of approximation with respect to growing values of d (up) and of m (down). All

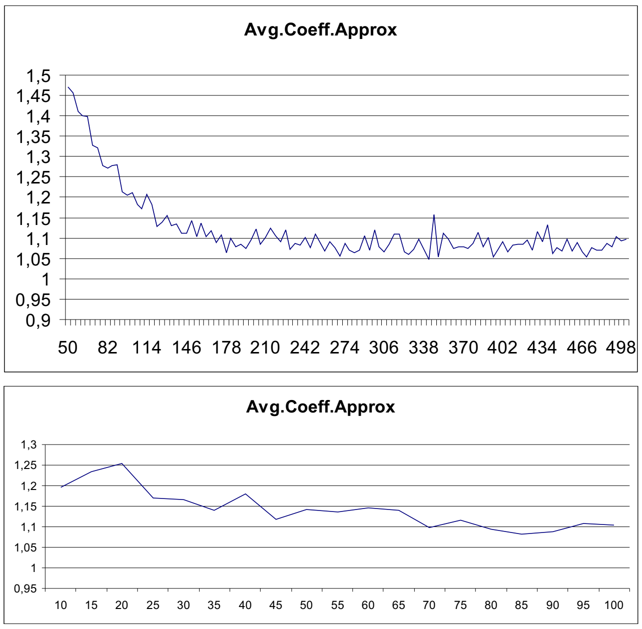

Fig. 6. Average coefficients of approximation obtained by letting vary d (up) and m (down) independently

coefficients have been computed by performing 10 tests with instances of length 100. In the first experiment the value of m has been fixed to 80, and the value of d has been let vary in the interval $[50, 500]$; in the second experiment, instead, the value of d has been fixed to 150 and the value of m has been let vary in the interval $[10, 100]$.

We have also tried to increase the number of candidate solutions occurring in each region (but the innermost, which continues to have only one solution). As we can see in Table 1, the average coefficient of approximation decreases, but only slightly. Perhaps more interesting was to evaluate the *speed* of convergence of the algorithm, with respect to the number of candidate solutions. Table 2 reports the results obtained with tests of 10 instances of length 100. The second, third and fourth column contain the number of steps needed (on average) to lower the average coefficient of approximation below 2, 1.5 and 1.1, respectively. Finally, in the last column we report the average of the gains obtained during each iteration of the algorithm. As we can see, when the number of candidate solutions

Number of solutions	app_{MA4MS}	Variance
2	1.0609374	0.0022672
3	1.0392002	0.0018685
5	1.0231814	0.0007303
10	1.0145107	0.0005448

Table 1. Results obtained with MA4MS LS, by letting vary the number of candidate solutions in the regions (but the innermost). Each test contained 100 instances of length 100; $m = 30$ and $d = 160$

Num. sol.	C.A. < 2	C.A. < 1.5	C.A. < 1.1	Avg. Gain
2	23	50	134	0.5065
3	17	38	111	0.3915
5	20	41	99	0.3355
10	15	37	93	0.2865

Table 2. Speed of convergence of MA4MS LS, with respect to the number of candidate solutions in the regions

grows the number of steps needed to obtain a good coefficient of approximation decreases. As a drawback, also the average gain decreases.

6 Some Heuristics for Min Storage

In this section we propose some "classical" polynomial time heuristics for solving Min Storage. Subsequently, we will run the same tests for both these heuristics and MA4MS LS, in order to compare the behavior of our membrane algorithm with more traditional approximation algorithms.

All the proposed algorithms have been implemented in the C programming language, to obtain the fastest execution times as possible. All lists have been implemented as arrays. We have associated a boolean flag to each element of the lists, indicating whether the element has to be considered as deleted or not, so that we can assume that the removal of a generic element $L[i]$ from a list L takes a constant time. As for sorting operations, we have assumed to use some comparisons–based optimal algorithm such as QuickSort or MergeSort, which take $\Theta(k \log k)$ time steps to sort k elements; in our experiments, we have indeed used the QuickSort routine included in the standard C libraries.

The first heuristic we consider is the *greedy* algorithm. This algorithm maintains a list L of elements to be considered. At the beginning of the execution L contains all the elements $\{e_1, e_2, \ldots, e_k\}$ of the instance. An integer variable st, initially set to 0, indicates the amount of energy currently stored into the gate. The algorithm repeats the following operations until L becomes empty: first it finds the minimum *positive* value of $st + \ell$, with $\ell \in L$, then it updates the value of st with $st + \ell$, and finally it removes ℓ from L. An integer variable $stmax$ records the maximum value reached by st; the value of $stmax$ at the end of the execution is the result returned by the greedy algorithm. It is easily seen

```
GREEDY($\mathcal{E}$)

$L \leftarrow$ Sort($\mathcal{E}$)
$st \leftarrow 0$
$stmax \leftarrow 0$
while Length($L$) $> 0$ do
    $i \leftarrow 1$
    while $st + L[i] < 0$ do
        $i \leftarrow i + 1$
    endwhile
    $st \leftarrow st + L[i]$
    $stmax \leftarrow \max\{st, stmax\}$
    remove $L[i]$ from $L$
endwhile
return $stmax$
```

```
MIN($\mathcal{E}$)

$L_n \leftarrow$ negative values of $\mathcal{E}$
$L_p \leftarrow \mathcal{E} \setminus L_n$
sort $L_p$ and $L_n$ in increasing order
$st \leftarrow 0$
$stmax \leftarrow 0$
while Length($L_p$) $> 0$ do
    $st \leftarrow st + \min(L_p)$
    $stmax \leftarrow \max\{st, stmax\}$
    remove $\min(L_p)$ from $L_p$
    while Length($L_n$) $> 0$ and
            $st + \max(L_n) \geq 0$ do
        $st \leftarrow st + \max(L_n)$
        remove $\max(L_n)$ from $L_n$
    endwhile
endwhile
return $stmax$
```

Fig. 7. Pseudocode for the GREEDY (on the left) and MIN (on the right) algorithms

that this algorithm can also be implemented as shown on the left side of Figure 7. From the inspection of the pseudocode it is clear that, under the hypotheses made above, the execution time of the whole algorithm is $\Theta(k^2)$.

Another heuristic is the MIN algorithm, whose pseudocode is shown on the right side of Figure 7. As we can see, at each iteration of the outer `while` loop the minimum of the remaining positive elements is chosen. For each positive element considered the inner `while` loop takes as many negative elements as possible, choosing the maximum of them (that is, the one with minimum absolute value) at each iteration. After an initial sorting, each element is considered only once during the execution of the two `while` loops; hence, the total execution time of the algorithm is $\Theta(k \log k)$. We have also considered a *dual* algorithm, which we have called MAX, where at each iteration of the outer loop the *maximum* of the remaining positive elements is chosen, whereas at each iteration of the inner loop the minimum of the remaining negative values is chosen.

Another variation is the MAXMINMAX algorithm, where at each iteration of the outer `while` loop the maximum of the remaining positive values is chosen, as in MAX. This time, however, there are two inner `while` loops: first we remove (as much as possible) the minimum negative elements, that is those with highest absolute value, and then we remove as much as possible the maximum elements.

Also in this case there exists a dual algorithm, called MinMaxMin, where at each iteration of the outer loop we remove the minimum of the remaining positive elements, and in the two inner loops we remove first the maximum and then the minimum of the remaining negative elements.

A further variation is given by algorithms MinMaxMinMax and MaxMin-MaxMin. In the outer loop of these algorithms the maximum or the minimum of the remaining positive elements is alternately chosen; in particular, in the former algorithm the first element chosen from the instance is the minimum of positive elements, whereas in the latter algorithm the maximum element of the instance is chosen first. The two inner loops are just like those of MaxMinMax and MinMaxMin; in particular, if the minimum of positive values has been chosen in the outer loop then we first remove the maximum negative elements and then the minimum ones, whereas we do the opposite if the maximum of positive elements was chosen. It is immediately seen that all the variations just exposed are uninfluent to the asymptotic execution time, that remains equal to $\Theta(k \log k)$.

Another approach to solve Min Storage is the Best Fit algorithm, shown in Figure 8. Best Fit assumes as a first estimate for the capacity of the gate

```
Best Fit($\mathcal{E}$)

$L_n \leftarrow$ negative values of $\mathcal{E}$
$L_p \leftarrow \mathcal{E} \setminus L_n$
sort $L_p$ and $L_n$ in increasing order
$est \leftarrow \max_{1 \le i \le k} |e_i|$
$st \leftarrow 0$
while Length($L_p$) > 0 do
    if $st + \min(L_p) > est$ then
        $est \leftarrow st + \min(L_p)$
        $st \leftarrow st + \min(L_p)$
        remove $\min(L_p)$ from $L_p$
    else
        for $i \leftarrow$ Length($L_p$) downto 1 do
            if $st + L_p[i] \le est$ then
                $st \leftarrow st + L_p[i]$
                remove $L_p[i]$ from $L_p$
            endif
        endfor
    endif
    for $i \leftarrow 1$ to Length($L_n$) do
        if $st + L_n[i] \ge 0$ then
            $st \leftarrow st + L_n[i]$
            remove $L_n[i]$ from $L_n$
        endif
    endfor
endwhile
return $est$
```

Fig. 8. Pseudocode for the Best Fit algorithms

(denoted by *est* in the pseudocode) the theoretical lower bound $\max_{1 \le i \le k} |e_i|$. During the execution of the algorithm the estimate for the capacity is adjusted, by increasing it of the smallest possible amount. Precisely, at each iteration of the outer `while` loop we add to the internal storage some positive values from the instance, and then we add some negative values. Positive values of the instance are scanned from the maximum down to the minimum; each of them is added to the internal storage (and removed from the instance), unless the resulting value exceeds *est*. Analogously, negative values are scanned from the minimum to the maximum; each of them is added to the internal storage (and removed from the instance), unless the resulting value becomes negative. If at some point no positive value can be added — that is, if $st + \min(L_p) > est$, where st is the energy currently stored into the gate — then we adjust the value of est by putting $est = st + \min(L_p)$. Now we can add $\min(L_p)$, the minimum of the remaining positive elements, to the internal storage and then try to add some negative elements. The result returned by the algorithm is the value of est at the end of the execution, that is, after all the elements of the instance have been considered. A direct inspection of the pseudocode allows us to see that the execution time of Best Fit is $\Theta(k^2)$.

7 Comparison Experiments

In this section we describe three computer experiments we have performed to compare the behavior of the proposed heuristics with MA4MS LS. Each instance was generated according to the random process described in section 4. All the classical heuristics, as well as MA4MS LS (with $m = 300$ membranes and $d = 500$ iterations), have been executed on a number of instances, and for each algorithm we have computed its average coefficient of approximation as well as the corresponding variance. The results obtained during these experiments are illustrated in Figure 9.

In the first experiment we have generated 100 instances, each one containing 12 elements. The elements were chosen from the interval $[-10^6, 10^6]$ of integers. The small number and length of instances have been chosen in order to allow

Algorithm	app_A	Var	app_A	Var	app_A	Var
Greedy	1.2052082	0.0615908	1.3844832	0.0664088	1.3948121	0.0468795
Min	1.3901304	0.0993216	1.7585659	0.0720077	1.6441352	0.0416634
Max	1.3804116	0.0979608	1.7533215	0.0733613	1.6424797	0.0449882
MaxMinMax	1.0863972	0.0143145	1.1051660	0.0055823	1.4993568	0.0784421
MinMaxMin	1.3901304	0.0993216	1.7585659	0.0720077	1.6441352	0.0416634
MaxMinMaxMin	1.1312751	0.0253605	1.1356385	0.0057628	1.5032439	0.0757982
MinMaxMinMax	1.1568342	0.0205104	1.1368552	0.0058827	1.1470192	0.0042202
Best Fit	1.0202729	0.0043468	1.0072024	0.0017742	1.1619727	0.0219509
MA4MS LS	1	0	1.0202449	0.0006565	1.0185239	0.0005584
	First experiment		Second experiment		Third experiment	

Fig. 9. Results obtained during the three computer experiments

the computation of optimal solutions through the "brute force" algorithm that examines all permutations in S_k. This means that the obtained results are the real average coefficients of approximation of the involved heuristics. Due to the length of instances, during the other two experiments we were not able to compute optimal solutions; hence, in those cases, in order to compute the coefficients of approximation we have used the theoretical lower bound $\max_{1 \leq i \leq k} |e_i|$ as the optimal solution, thus obtaining upper bounds to the real coefficients. Indeed, the first experiment was conceived to compare these upper bounds with the real coefficients of approximation, although computed over very small instances. As we can see from the tables, the values obtained are pretty similar.

In the first experiment, among the traditional heuristics Best Fit has obtained the best average coefficient of approximation, and also the smallest variance; this means that it frequently finds a good solution. On the other hand, the membrane algorithm has always found the optimal solution. This is probably due to the fact that, since the number of regions is high with respect to the length of the instances, then the probability that an optimal solution is produced during the initial generation of candidate solutions is very high. Further, a relatively high number of iterations in the algorithm allows such optimal solution to reach the innermost membrane before the algorithm halts.

In the second experiment we have generated 100000 instances of 100 elements, each taken from the interval $[-10^6, 10^6]$ of integers. As we can see in Figure 9, for traditional heuristics we have obtained results similar to those of the first experiment. MA4MS LS has obtained both a low average coefficient of approximation and a (very) low variance; moreover, it performs better than almost all the traditional heuristics. However, the winner is Best Fit. An interesting observation is that Best Fit did not find a solution equal to $\max_{1 \leq i \leq k} |e_i|$ for only 4564 of the 100000 instances; since the optimal solution cannot be less than this value, this means that for at least 95.4% of instances Best Fit found the optimal solution. We can explain this result by saying that Best Fit performs so well because it has been intentionally conceived for Min Storage. We are currently investigating whether higher values for the parameters m and d would lead to a better performance of MA4MS LS. Let us note, however, that even if this hypothesis should be true, the execution time of the algorithm would make us prefer again Best Fit, since it is very quick. Does this mean that we should forget MA4MS LS? The answer is negative, as shown by the next experiment.

For the third experiment, we have considered a variant of the Min Storage problem, where we have relaxed the requirement that the amount of energy stored into the gate at the beginning of the computation is zero. This corresponds to a natural extension of the notion of conservative computation, obtained by letting the gate to have a positive amount ε of energy stored at the beginning of the computation, and requiring that exactly the same amount ε of energy is stored into the gate at the end of the computation. When this situation occurs, we say that the computation is *ε–conservative*. Hence up to now we have dealt with 0–conservativeness. Clearly also the variant of Min Storage concerning ε–conservative computations (with $\varepsilon \geq 0$) is NP–hard, by the restriction property [2, page 63], since it contains Min Storage as a particular case.

In the third experiment we generated 100 instances, each one composed by 100 elements taken from the interval $[-10^6, 10^6]$ of integers. For each instance we ran the proposed algorithms, varying the initial energy ε from 0 to $\max_{1\le i\le k} |e_i|$, with steps of 100. At first sight it may be surprising to see that Best Fit gives no more the best results: indeed, among the traditional heuristics MinMaxMin-Max has both the lowest average coefficient of approximation and the lowest variance. It is our opinion that Best Fit does not perform better than Min-MaxMinMax because the former algorithm starts by considering the elements of the instance from the greatest positive to the smallest positive element, each time taking the element if there is enough free storage into the gate; negative elements are considered only later. Of course this may not be the optimal strategy, especially when the initial energy stored into the gate is high with respect to gate capacity. The latter algorithm alternately chooses the minimum and the maximum of the positive elements remaining into the instance, and then it immediately considers negative elements: as a consequence, it has more chances to make the right choices. Some modifications to the Best Fit algorithm in order to perform better when there is a positive initial amount of energy into the gate are currently under consideration.

However, the absolute winner in this experiment is MA4MS LS. Once again it has a very low variance, and almost the same average coefficient of approximation as in the previous experiment; we can interpret this fact by saying that MA4MS LS is a *stable* algorithm, in the sense that its performance is not affected by small changes in the definition of the instances.

8 Conclusions

In this paper we have proposed some polynomial time approximation heuristics for Min Storage, a strongly NP–hard optimization problem that naturally arises in the context of conservative (that is, energy preserving) computations. One of the proposed heuristics is a membrane algorithm which was inspired by a previous work by Nishida [6].

We studied the behavior of all these heuristics on (almost) uniformly randomly chosen instances through several computer experiments. A first set of experiments allowed us to understand that a very simple local optimization algorithm, LocalSearch4MS, suffices to make the membrane algorithm obtain good solutions for Min Storage. We have called MA4MS LS the resulting algorithm. The results obtained from a second set of experiments suggest that Min Storage seems to be easy to solve on uniformly randomly chosen instances. In particular, one of the proposed heuristics, namely Best Fit, seems to perform very well when the initial energy stored into the gate is zero. Interestingly, the same heuristic is no more the best when the initial energy is positive.

If we look at the average coefficient of approximation obtained for MA4MS LS during the second set of experiments, we can see that we always obtain approximately the same value. Moreover, the low value obtained for the variance shows that the algorithm is also pretty stable, that is, its (average) behavior is not affected too much by small changes in the instances of the problem. If we compare this situation with the behavior of Best Fit, we can draw the following conclusions. Best Fit performs well since it is an algorithm which has been intentionally crafted for Min Storage; stated otherwise, it strongly reflects the structure of the problem. On the contrary, MA4MS LS is a *general* algorithm, that behaves in the same way independent of the problem on which it is applied.

Acknowledgements

We gratefully thank the anonymous referees, whose comments have helped us to improve a previous version of this paper.

References

1. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti–Spaccamela, M. Protasi. *Complexity and Approximation. Combinatorial Optimization Problems and Their Approximability Properties*. Springer–Verlag, 1999.
2. M. R. Garey, D. S. Johnson. *Computers and Intractability. A Guide to the Theory on NP–Completeness*. W. H. Freeman and Company, 1979.
3. G. V. Gens, E. V. Levner. Computational complexity of approximation algorithms for combinatorial problems. *Proceedings of the 8th International Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 74, Springer–Verlag, Berlin, 1979, pp. 292–300.
4. D. E. Goldberg, R. Lingle. Alleles, Loci and the Traveling Salesman Problem. In *Proceedings of the International Conference on Genetic Algorithms*, 1985, pp. 154–159.
5. A. Leporati, C. Zandron, G. Mauri. Conservative Computations in Energy–based P systems. In Giancarlo Mauri, Gheorghe Păun, Mario J. Pérez-Jiménez, et al. *Membrane Computing: 5th International Workshop, WMC 2004*, Milan, Italy, June 14–16, 2004, LNCS 3365, Springer–Verlag, 2005, pp. 344–358.
6. T. Y. Nishida. Membrane Algorithms. In Rudolf Freund, Gheorghe Păun, Grzegorz Rozenberg, Arto Salomaa (Eds.) *Membrane Computing: 6th International Workshop, WMC 2005*, Vienna, Austria, July 18–21, 2005, LNCS 3850, Springer–Verlag, 2006, pp. 55–66.
7. G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 1(61):108–143, 2000. See also Turku Centre for Computer Science — TUCS Report No. 208, 1998. Available at: *http://www.tucs.fi/Publications/techreports/TR208.php*
8. G. Păun. Computing with Membranes. An Introduction. *Bulletin of the EATCS*, 67:139–152, February 1999.
9. G. Păun. *Membrane Computing. An Introduction*. Springer–Verlag, Berlin, 2002.
10. G. Păun, G. Rozenberg. A Guide to Membrane Computing. *Theoretical Computer Science*, 287(1):73–100, 2002.
11. The P systems Web page: `http://psystems.disco.unimib.it/`