Computational Complexity of Simple P Systems

Gabriel Ciobanu, Andreas Resios

"A.I.Cuza" University of Iaşi
Romania
gabriel@info.uaic.ro

Summary. We introduce a new class of membrane systems called simple P systems, and study its computational complexity using the classical theory. We start by presenting the knapsack problem and analyzing its space and time complexities. Then we study the computational complexity of simple P systems by considering the static allocation of resources enabling the parallel application of the rules. We show that the problem of allocating resources for simple P systems is **NP**-complete by reducing it to the knapsack problem. Thus we express the computational complexity of this class of P systems in terms of classical complexity theory.

1 Introduction

We describe the computational complexity of the simple P systems in classical complexity theory, extending the short note presented initially in [5]. Membrane computing is a rather young field of *natural computing* which has been developing very fast in the last decade. It combines the power of distributed parallel rewriting systems with the power and context evolution to achieve computational universality. We use a subclass of transition P systems, simple P systems, where the left side of the rules can contain only a single object with different multiplicity. We use a classical combinatorial **NP**-complete problem, the *knapsack* problem, to show that the static allocation of rules for this class is **NP**-complete.

The structure of this paper is as follows. In Section 2 we present a pseudo-polynomial algorithm for the knapsack problem and study its complexity. Then we give a short presentation of transition P systems, and introduce the simple P systems in Section 3. Then we show that static allocation of the available resources to rules is **NP**-complete. In Section 5 we present a new approach to study the complexity of simple P systems. Conclusion and references end the paper.

2 Knapsack Problem

The knapsack problem, a classical combinatorial optimization problem, refers to finding a maximum total value given a set of objects values and weights, and a weight limit. The discrete version refers to the fact that we can only include an object as a whole, not just a part of it. Mathematically, the discrete knapsack problem can be formulated as follows: given a bag of capacity c and n objects, labelled from $1, \ldots, n$, each having value p_i and weight w_i, maximize $\sum_{i=1}^{n} p_i x_i,\ x_i \in \{0, 1\}$, subject to $\sum_{1}^{n} w_i x_i \leq c$, where $x_i = 1$ means that we take object i. This problem is known to be an **NP**-complete problem. However, there exists a *pseudo-polynomial* time algorithm using dynamic programming with running time of $O(n \cdot c)$. An algorithm is said to run in pseudo-polynomial time if its running time is polynomial in the numeric value of the input (however this can be exponential with respect to the length). Formally, we say that a function f is pseudo-polynomial if $f(n)$ is no greater than a polynomial function of the problem size n and an additional property of the input $k(n)$. Note that pseudo-polynomial time becomes polynomial time if the values are encoded in unary base.

The knapsack problem can be expressed as a optimization problem: as follows:

- Objective function
$$\max \sum_{i=1}^{n} p_i x_i$$
- Restrictions
  - $x_i \in \{0, 1\}$;
  - $\sum_{1}^{n} w_i x_i \leq c$.

To solve this problem using dynamic programming we need to define the notion of a state and the transition between two states. For this problem a state is defined by the number of objects we take into consideration. Thus we start with an initial state where we do not have any object to choose from, and we make transitions to the next state until we reach a final state. A transition is represented by the choice between inserting the object in the bag or not. We denote by $f_i(X)$ the function which answers the question "What is the optimal value obtained by using only i objects and X weight?". Thus a state i is defined by the function f_i. The function f_i can be computed as follows:

$$f_i(X) = \begin{cases} -\infty & , X < 0 \\ 0 & , i = 0 \wedge X \geq 0 \\ \max\{f_{i-1}(X), f_{i-1}(X - w_i) + p_i\} & , \text{otherwise} \end{cases} \tag{1}$$

The answer to the knapsack problem is given by the value of $f_n(c)$. The functions f_i can be stored as a table, and can be computed starting from the initial state to the last state. Note that the current state depends only on the previous state, thus we can store only the last two lines in the table. We use the example given in Table 1 where we consider $c = 10$.

i	1	2	3
w_i	3	5	6
p_i	10	30	20

Table 1. Knapsack instance

X	0	1	2	3	4	5	6	7	8	9	10
f_0	0	0	0	0	0	0	0	0	0	0	0
f_1	0	0	0	10	10	10	10	10	10	10	10
f_2	0	0	0	10	10	30	30	30	40	40	40
f_3	0	0	0	10	10	30	30	30	40	40	40

Table 2. Values for f corresponding to instance defined in Table 1

We obtain Table 2 corresponding to the recurrence defined by f Note that function f_i has many repeating values. To solve the problem of space, we need to store only the different values for f_i. A possible solution is to associate with each different value of f_i a 3-uple $(k, W_{i,k}, T_{i,k})$ with the following meaning: k represents the profit, $W_{i,k}$ represents the sum of the objects weight which we can use to achieve profit k, and $T_{i,k}$ represents the objects which are used to achieve that profit.

By using this approach, we need to keep a list of 3-uples instead of the full table. We now show an example of how we use this list to solve the problem. We start with the list containing only $\{(0, 0, \emptyset)\}$. At each iteration we construct a new list A_i that contains the 3-uples with the valid profits that we could obtain by using the current object. The new list is obtain by merging the previous list with the new constructed list: $L_{i+1} = \mu(L_i, A_i)$, where $\mu(A, B)$ is the merging of two list of 3-uples. For the instance previously presented we have:

$$
\begin{aligned}
L_0 &= \{(0, 0, \emptyset)\} \\
A_0 &= \{(10, 3, \{1\})\} \\
L_1 &= \{(0, 0, \emptyset), (10, 3, \{1\})\} \\
A_1 &= \{(30, 5, \{2\}), (40, 8, \{1, 2\})\} \\
L_2 &= \{(0, 0, \emptyset), (10, 3, \{1\}), (30, 5, \{2\}), (40, 8, \{1, 2\})\} \\
A_2 &= \{(20, 6, \{3\}), (30, 9, \{1, 3\})\}) \\
L_3 &= \{(0, 0, \emptyset), (10, 3, \{1\}), (20, 6, \{3\}), (30, 5, \{2\}), (40, 8, \{1, 2\})\}
\end{aligned}
$$

The solution to the problem is given by the last element of L_3. In our example it is given by $(40, 8, \{1, 2\})$, which means we can obtain a profit of 40 by taking items 1 and 3 that weight 8.

The Algorithm 1 solves the knapsack problem using this approach. We now express the time and space complexity of Algorithm 1. We know that $|L_{i+1}| \leq 2 \cdot |L_i|$ because $|A_i| \leq |L_i|$. The computation of L_{i+1} from L_i and A_i is done in $O(|L_i| + |A_i|) = O(|L_i|)$ time. We know that the 3-uples from L_i satisfy the relation:

Algorithm 1 Knapsack(n,w,p,M)

```
1: L_0 ← {(0, 0, ∅)}
2: for i ← 1 to n do
3:    A_{i−1} ← ∅
4:    for all (k, W_{a,k}, T_{a,k}) in L_{i−1} do
5:       if W_{a,k} + w_i ≤ M then
6:          A_{i−1} ← A_{i−1} ∪ {(k + p_i, W_{a,k} + w_i, T_{a,k} ∪ {i})}
7:       end if
8:    end for
9:    L_i ← Merge(L_{i−1}, A_{i−1})
10: end for
11: return last(L_n)
```

$$0 \leq |L_i| \leq k \leq \sum_{j=1}^{i} p_j \leq n \cdot \max\{p_1, \ldots, p_n\}$$

It follows that $|L_i| \leq n \cdot \max\{p_1, \ldots, p_n\}$. In conclusion Algorithm 1 has the time complexity:

$$O(\sum_{i=1}^{n} |L_i|) = O(n^2 \cdot \max\{p_1, \ldots, p_n\})$$

Note that this complexity is pseudo-polynomial, and if $max\{p_1, \ldots, p_n\} > 2^n$ then this algorithm runs in exponential time w.r.t the size of its input.
The space complexity is:

$$O(\sum_{i=1}^{n} |L_i|) = O(\sum_{i=1}^{n} 2^i) = O(2^n)$$

In the *Merge* procedure presented as Algorithm 2, when we have two items with the same profit we choose the one with the lowest weight. This assures that for each possible profit we have the minimum weight.

3 Simple P Systems

Membrane computing represents an unconventional paradigm of computing which combines the power of distributed parallel rewriting systems with the power and context evolution. Local rules and the evolution contexts are biological metaphors: the rules are developmental rules in cells, and contexts denote division mechanisms of cells and active/mobile membranes. There are several ingredients in membrane systems which are meaningful from the point of view of biological media. The basic model and many variants are described in [7], and some applications are presented in [4]. Membrane computing is used both to model cells or biological systems and to study the computability and complexity of a new and unconventional computing device.

Algorithm 2 Merge(A,B)

```
1: S ← ∅, i ← 1, j ← 1
2: n ← min(|A|, |B|)
3: while i ≤ n && j ≤ n do
4:    (k1, W_{a,k1}, T_{a,k1}) ← A[i], (k2, W_{b,k2}, T_{b,k2}) ← B[j]
5:    if k1 < k2 then
6:       S ← S ∪ {(k1, W_{a,k1}, T_{a,k1})}, i ← i + 1
7:    else if k1 > k2 then
8:       S ← S ∪ {(k2, W_{b,k2}, T_{b,k2})}, j ← j + 1
9:    else if k1 = k2 then
10:      if W_{a,k1} < W_{b,k2} then // chose the one with minimum weight
11:         S ← S ∪ {(k1, W_{a,k1}, T_{a,k1})}
12:      else
13:         S ← S ∪ {(k2, W_{b,k2}, T_{b,k2})}
14:      end if
15:      i ← i + 1, j ← j + 1
16:   end if
17: end while
18: if i ≤ n then
19:   for j ← i to |A| do
20:      S ← S ∪ A[j]
21:   end for
22: else
23:   for i ← j to |B| do
24:      S ← S ∪ B[i]
25:   end for
26: end if
27: return S
```

The transition P system is represented by *regions* delimited by a *membrane structure* that contains *multisets* of objects that evolve according to associated rules. A computation consists of a number of transition between system configurations and the result is represented either by the objects present in the final configuration in a specific membrane or by the objects which leave the outermost membrane of the system (the skin membrane) during the computation.

Definition 1. *A* transition P system *of degree* $n, n \geq 1$*, is a construct*

$$\Pi = (O, T, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n)\,,$$

where

1. O *is an alphabet of* objects*;*
2. $T \subseteq O$ *(the output alphabet);*
3. μ *is a membrane structure of degree* n*;*
4. $w_i, 1 \leq i \leq n$*, strings that represent multisets over* V *associated with the regions of* μ*;*

5. *R_i represents the rules from region μ_i of μ; an* evolution rule *is a pair (u, v) written as $u \to v$, where u is a string over O and $v = v'$ or $v = v'\delta$, where v' is a string over $\{a_{here}, a_{out}, a_{in_j} | a \in O, 1 \leq j \leq n\}$, and δ is a special symbol not in O; the length of u is called the* radius *of rule $u \to v$. $R_i, 1 \leq i \leq n$, are finite sets of evolution rules over O.*

Definition 2. *A* simple P system *is a transition P system where the left side of a rule can contain a single object with an arbitrary multiplicity. Formally, a rule $u \to v \in R_i$ has $u = k \cdot a$, where $k \in \mathbb{N}$ and $a \in O$.*

The *membrane structure* is a tree structure, where each node represents a membrane. The relation between a child node and a parent node symbolizes that the parent membrane contains the child membrane. To define the *membrane structure* we consider the language MS over the alphabet $\{[,]\}$ recursively defined as follows:

1. $[\,] \in MS$;
2. if $\mu_1, \ldots, \mu_n \in MS$, $n \geq 1$ then $[\mu_1 \ldots \mu_n] \in MS$;
3. nothing else is in MS.

We define a binary relation $\sim$ over the elements of MS: $x \sim y$ if and only if we can write $x = \mu_1\mu_2\mu_3\mu_4$, $y = \mu_1\mu_3\mu_2\mu_4$, for $\mu_1\mu_4 \in MS$ and $\mu_2, \mu_3 \in MS$. We denote by $\sim$ the reflexive and transitive closure of $\sim$ (note that $\sim$ is an equivalence relation). We denote by $\overline{MS}$ the set of equivalence classes of MS with respect to $\sim$. A *membrane structure* is an element of $\overline{MS}$, where each pair of matching parentheses $[,]$ is a membrane. The degree of a membrane structure is defined as the number of membranes it contains. A natural way to represent membrane structure is by a Venn diagram because this emphasizes the topological structure between computing compartments.

A configuration of the system is given by the membrane structure and the contents of each region. The initial configuration is a $(n+1)$-tuple $(\mu, w_1, \ldots, w_n)$. Having the possibility to dissolve a membrane, we can obtain a configuration which has only some of the initial membranes. Thus we define a configuration of Π as any sequence $(\mu', w'_{i_1}, \ldots, w'_{i_k})$, with μ' a membrane structure obtained by dissolving from μ all membranes different from $i_1, \ldots, i_k$, with w_{i_j} strings over O, $1 \leq j \leq k$, and $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$.

For two configurations $C_1 = (\mu', w'_{i_1}, \ldots, w'_{i_k})$, $C_2 = (\mu'', w''_{j_1}, \ldots, w''_{j_k})$ of Π we write $C_1 \Rightarrow C_2$, and we say that we have a *transition* from C_1 to C_2 if we can pass from C_1 to C_2 by using the evolution rules from $R_{i_1}, \ldots, R_{i_k}$. A sequence of transitions between configurations of a given P system, Π is called a *computation* with respect to Π. A computation *halts* if there is no rule applicable to the objects from the last configuration. The result of a computation is $\Psi_T(w)$, where w describes the multiset of output objects sent out by the system during the computation. $\Psi_T(w)$ is the Parikh mapping associated with T; it is defined by $\Psi_T(w) = (|w|_{a_1}, \ldots, |w|_{a_n})$, where $T = \{a_1, \ldots, a_n\}$, $w \in T^*$, and $|w|_{a_i}$ denotes the number of occurrences of a_i in w. The set of such vectors $\Psi_T(w)$ is denoted by $Ps(\Pi)$, and we say it is generated by Π.

4 Complexity of the Parallel Application of Rules by Static Allocation of Resources

We now study the computational power of simple P systems coming from the maximal parallel and nondeterministic application of the rules. To express this in terms of computational complexity, we envision a device capable of solving this problem and we express its complexity. We call this device a *resource allocator*, and abbreviate it by RA. Two operational semantics of membrane systems were defined in [1] and [2]. They differ only in the way the maximal parallel application of rules is described, and reflect the fact that resource allocation to rules can be done either statically or dynamically. A dynamic resource allocation is based on applying rules one by one in a nondeterministic manner until there is no applicable rule left [1].

An alternative is given by static allocation [2], where the existing resources are distributed in a nondeterministic and maximal way to the rules which then are applied in parallel. The equivalence between static and dynamic allocation semantics is proved in [2].

Therefore the purpose of the resource allocator is to allocate multisets of objects to rules such that the evolution is then done in a maximal parallel and nondeterministic way. Given this setup, the maximal parallel application of rules depends on RA being able to solve an instance of the discrete knapsack problem. The nondeterminism comes from the fact that we can choose different multisets of rules that correspond to the solution of the knapsack problem given the contents of RA and we choose one in a nondeterministic way. We can associate a resource allocator with every membrane of a simple P system. Given the parallel evolution of membranes, we note that every resource allocator resolves a particular instance associated with its membrane, and each one operates independent of the others. We represent multisets as a string over their support alphabet. A resource allocator can be formally defined as a mapping $RA : O^* \rightarrow (O^*)^{|R|+1}$, where O is the alphabet of objects, $w \in O^*$ and R is the set of rules associated with a membrane:

$$RA(w) = \{(w_1, w_2, \ldots, w_{|R|}, w') \mid w_i \not\subseteq w', i = \overline{1, |R|} \wedge \sum_{i=1}^{|R|} w_i + w' = w\} \quad (2)$$

The resource mapping problem can be formulated as follows: given a resource allocator RA of a simple P system, decide which of the rules are applied such that the system evolves in a maximal parallel and nondeterministic way. Formally, given (μ, w, R, RA) where

- μ is a membrane structure of a P system,
- w is the multiset of objects associated with μ,
- R is the set of rules associated with μ,
- RA is the resource allocator associated with μ,

maximize $|\sum_{i=1}^{|R|} w_i|$, where $(w_1, w_2, \ldots, w_{|R|}, w') \in RA(w)$ and $\exists\ u_1, \ldots, u_{|R|}$ such that $u_i \to v_i \in R\ \wedge u_i \neq u_j, \forall i \neq j\ \wedge\ \exists\ k_i \in \mathbb{N}\ w_i = k_i \cdot u_i, \forall i = \overline{1, |R|}$ and $\forall u \to v \in R$ we have that $u \nsubseteq w'$.

Note that the maximal parallel rewriting comes from the fact that the RA chooses nondeterministically which multiset of rules to apply (by assigning resources $w_1, w_2, \ldots, w_{|R|}$ to each rule), and the set is maximal because we cannot apply another rule using the remaining resources w'. When we have multiple solutions to the problem, we chose one nondeterministically.

To clarify the definition we give the following example: suppose that the resource allocator RA has to distribute the multiset $10a$ to the rules $4a \to b$, $3a \to c$ and $2a \to d$. We can now distribute the resources according to our definition in multiple ways. We show only a few:

$$10a \Rightarrow 2 \cdot 4a + 0 \cdot 3a + 1 \cdot 2a \text{ (no remaining a)}$$
$$10a \Rightarrow 0 \cdot 4a + 3 \cdot 3a + 0 \cdot 2a \text{ (remaining 1 a)}$$
$$10a \Rightarrow 1 \cdot 4a + 1 \cdot 3a + 1 \cdot 2a \text{ (remaining 1 a)}$$

Note that it is not possible to use the remaining resources to apply another rule, i.e. in our example we cannot have more than one remaining a because that means that we can apply another rule.

We show that the problem of resource mapping in simple P systems (shortly **RMP**) can be reduced to the *discrete knapsack problem* (shortly **KNAP**). In order to prove this, we first make a Karp reduction from **KNAP** to **RMP**.

Definition 3. *Let A,B be two decision problems over the alphabet Σ. We say that A can be Karp (or polynomial) reduced to B, and write $A \leq_m B$ if $\exists f : \Sigma^* \to \Sigma^*$, where f can be computed in deterministic polynomial time such that $x \in A \Leftrightarrow f(x) \in B, \forall x \in \Sigma^*$.*

Lemma 1. ***KNAP*** $\leq_m$ ***RMP****.*

Proof. We consider only the decision version of **RMP** and **KNAP**, because every optimization problem can be reduced to a decision one. We consider that the value of each object is equal with its weight, thus the knapsack problem becomes the *subset sum* problem (*subset sum* is also a **NP**-complete problem). We use the name **KNAP** because we are mainly interested in an implementation of the resource allocator. We transform in polynomial time an instance of **KNAP** into an instance of **RMP**. We denote the transformation by $f(c, n, W, P)$, and we show that $\mathbf{KNAP}(c, n, W, P) =$ yes implies $\mathbf{RMP}(f(c, n, W, P)) =$ yes, and $\mathbf{KNAP}\ (c, n, W, P) =$ no implies $\mathbf{RMP}(f(c, n, W, P)) =$ no. The transformation f has to create an instance of the **RMP** problem. We need to define a membrane structure μ, the set R of rules, a multiset w of objects in μ, and the resource allocator RA.
We use the same notations as above:

$$\begin{aligned}
&\mu \text{ is an arbitrary membrane structure,}\\
&w = a^c,\ a \in O,\\
&R = \{a^{w_i} \to b \mid b \in O,\ w_i \in W,\ \forall i = \overline{1,n}\},\\
&RA \text{ is a resource allocator defined as in equation (2).}
\end{aligned} \tag{3}$$

The transformation f is defined by the equations presented in (3). The membrane structure of μ can be chosen arbitrary because it is not involved in the distribution of object to rules. To express the capacity c of the knapsack, we define the contents of the membrane structure μ as a multiset composed of a single object a with multiplicity c. For every object we define a rule, such that if the object is used in the knapsack problem, then it will be used by RA only once. The transformation can be done in polynomial time with respect to the number of objects.

For the first part of the implication we start from $\mathbf{KNAP}(c, n, W, P) =$ yes, and we need to show that $\mathbf{RMP}(f(c, n, W, P)) =$ yes. Let us assume that $\mathbf{RMP}(f(c, n, W, P)) =$ no. This implies that there exists a better solution to the instance $f(c, n, W, P)$. Let the solution be $|\sum_{i=1}^{|R|} \bar{w}_i| \leq |w|$, where $(\bar{w}_1, \bar{w}_2, \ldots, \bar{w}_{|R|}, w') \in RA(w)$. We know that $|\sum_{i=1}^{|R|} \bar{w}_i| > |\sum_{i=1}^{|R|} w_i|$ because we assumed we have a better solution. Using this solution we construct a solution to $\mathbf{KNAP}(c, n, W, P)$ as follows:

$$x'_i = \begin{cases} 1 & , \quad \text{if } 0 < |\bar{w}_i| \\ 0 & , \quad \text{otherwise} \end{cases} \tag{4}$$

Note that if we have $|\bar{w}_i|$, then rule r_i has been used. We then have: $\sum_{i=1}^{n} p_i x'_i = \sum_{i=1}^{n} |\bar{w}_i| x'_i = |\sum_{i=1}^{n} \bar{w}_i| > |\sum_{i=1}^{n} w_i| = \sum_{i=1}^{n} p_i x_i$, because we assumed that we have a better solution. Thus we have $\mathbf{KNAP}\ (c, n, W, P) =$ no, which is a contradiction.

Now we prove that

$$\mathbf{KNAP}(c, n, W, P) = \text{no} \quad \text{implies} \quad \mathbf{RMP}(f(c, n, W, P)) = \text{no}$$

We consider that the decision problem was for the optimal value of T. Let us assume that we have $\mathbf{RMP}\ (f(c, n, W, P)) =$ yes, and the solution of this instance is formed by the following allocation: $(\bar{w}_1, \bar{w}_2, \ldots, \bar{w}_{|R|}, w') \in RA(w)$ and $|\sum_{i=1}^{|R|} \bar{w}_i| = T$. Like in the first implication, we construct an instance of **KNAP** such that $\mathbf{KNAP}(c, n, W, P) =$ yes. We construct this instance of **KNAP** as follows:

$$x'_i = \begin{cases} 1 & , \quad \text{if } 0 < |\bar{w}_i| \\ 0 & , \quad \text{otherwise} \end{cases} \tag{5}$$

We have $\sum_{i=1}^{n} p_i x'_i = T$. This is a contradiction, because we have $\mathbf{KNAP}$ $(c, n, W, P) =$ no.

Theorem 1. ***RMP*** *is* ***NP****-complete.*

Proof. We have

- **KNAP** $\leq_m$ **RMP**, by Lemma 1,
- **KNAP** is **NP**-complete,
- **NP** is closed under $\leq_m$.

Therefore **RMP** is **NP**-complete.

5 Complexity of Simple P Systems

We now present a way to use classic complexity theory classes to study the complexity of simple P systems. To use such an approach we need to take into account the distribution of objects to rules, rather than the number of steps performed in a computation because parallel evolution can consume more resources than sequential evolution w.r.t the number of steps.

We now extend the approach from [5], where the authors show that a P system can evolve using an **NP** oracle that solves the resource allocation problem for each of the membranes from the system. We avoid the oracle by using the resource allocator described in Section 4 which ensures the maximal parallelism and a nondeterministic evolution. Thus the parallel evolution of simple P systems can be viewed as a sequence of independent steps. Maximal parallel evolution means that we cannot apply another rule with the contents left in the membrane after the application of the selected rules. The maximal parallel application of rules depends on *RA* being able to solve an instance of the discrete knapsack problem and retrieve the solutions within a profit range that corresponds to this kind of evolution.

Each step is composed of three stages: the first consists of the assignment of objects to rules according to the resource allocator, the second represents the distribution of the results obtained from applying the selected rules, and finally the dissolution of certain membranes.

The first stage ensures a maximal parallel application of rules and consists of the creation of an instance of the discrete knapsack problem based on the multiset from the membrane, followed by solving the instance and obtaining the results. The second stage moves the objects obtained from the previous stage according to their tags. In the third stage we dissolve all membranes that contain the special symbol δ by transferring their resources to their parents (as an exception we do not transfer the δ symbols).

For the first stage of the process we present a function that transforms an instance of the resource allocation problem into an knapsack instance such that we can obtain the solution to the **RMP** instance by solving the transformed instance.

Given a membrane we define: the capacity c of the knapsack, the number n of objects, the weight w_i and value p_i for each object i:

$$
\begin{aligned}
&c = |w|;\\
&n = |\{w_{i_k} \mid w_{i_k} \text{ defined object}\}|\,;\\
&w_{i_k} = k \cdot |u_i|, \text{ where } R = \{r_1, \ldots, r_m\},\ r_i = u_i \to v_i,\ u_i \in O^* \wedge\\
&\quad k = \overline{1,p} \text{ where } p = \max\{j \in \mathbb{N} \mid w' \subseteq w \ \wedge\ w' = j \cdot u_i\};\ i = \overline{1,|R|};\\
&p_{i_k} = w_{i_k}
\end{aligned}
\tag{6}
$$

The transformation f is defined by the equations (6). Note that we do not need to use all the contents of the membrane. We denote by w the contents of the membrane, and by v the multiset which we intend to consume. We now have $w = v + v'$ which links w and v, where $\forall u \to v \in R$ we have that $u \nsubseteq v'$. This restriction assures us that we cannot apply a rule with the remaining contents of the membrane. In the knapsack problem an item can be used only once, but in membrane systems, because of maximal parallelism, a rule can be applied several times to all objects which it can process. Thus we need to define for every rule a "class" of items which represent all the possible ways in which a rule could be used. We denote by W the set of items defined; thus we have $|W| \leq c \cdot |R|$. The profit of an object is defined as the number of symbols it consumes – because we are interested in consuming as many symbols as possible. The transformation f can be computed in polynomial time with respect to $|w|$.

In the first stage we transform an instance of the resource allocation problem to an instance of the knapsack problem by using the function f. Then we solve the created instance, and obtain the rules which can be applied in parallel, together with the multiplicity of each rule. We can now express the computational complexity of each stage with respect to the input, represented by the multiset of the membrane.

For the first stage we need to express a relation between the number of objects created and the size of the multiset. From equations (6) we have that each $u_i \to v_i$ rule can introduces a maximum of $\frac{|w|}{|u_i|}$ objects. Summing these relations for each rule we have that:

$$n \leq \sum_{i=1}^{|R|} \frac{|w|}{|u_i|} = |w| \sum_{i=1}^{|R|} \frac{1}{|u_i|}$$

Note that $\sum_{i=1}^{|R|} \frac{1}{|u_i|}$ is a constant associated with the membrane, because the rules of a membrane do not change in the process of evolution. We denote this constant with S, and have that $n \leq |w| \cdot S$. Thus the complexity of this stage is $O(n) = O(|w| \cdot S)$.

For the second stage we use a pseudo-polynomial algorithm for knapsack with a complexity of $O(n \cdot c)$, where n is the number of objects, and c is the capacity of the knapsack. By using the relations (from the first stage) between the number of objects and the capacity, we have that the second stage has a complexity of $O(|w| \cdot S \cdot |w|) = O(|w|^2 \cdot S)$.

The third stage consists of applying the rules according to the solution from the knapsack problem. Note that a maximal parallel evolution does not imply a maximum profit. According to this, we chose nondeterministically a solution that corresponds to such a behaviour. Using the knapsack algorithm we compute all valid profits, so we need to chose only the ones which correspond to such an evolution. To achieve this, we define for each symbol a minimum allowed profit expressed as the difference between the multiplicity of the symbol from the input multiset and the minimum multiplicity of a rule that uses the symbol. Formally, $p_a^{min} = w(a) - \min_{u \to v \in R}\{u(a) | u(a) > 0\} + 1$. We introduce a lower bound in terms of profit, defined by summing the minimum allowed profit for each symbol. This assures that no rule can be applied using the remaining multiset. We know that for the pseudo-polynomial algorithm for knapsack we can retrieve the selected objects in $O(n)$, thus the complexity of this step is $O(|w| \cdot S)$. During this backtracking process we nondeterministically choose a rule that corresponds to the object chosen. Thus we obtain a multiset of rules that corresponds to maximal parallel evolution because no rule can be applied with the remaining contents and the evolution is nondeterministic because of the way the rules were chosen.

Using the example in Section 4, we show how we can distribute $10a$ as $3 \cdot 3a$(remaining $1a$). Using equations (6) we obtain the items in Table 3. By applying the knapsack algorithm we get the results in Table 4. In this case, the value of $p_a^{min} = 10 - 2 + 1 = 9$. Using the recurrence relation defined in equation (1) we obtain the items used of the solution. At each step $i = \overline{1, n}$ we test whether the item $n - i + 1$ was included in the knapsack or not. According to p_a^{min} the starting value can be 9 or 10. Suppose we start with the value 9. To test if object 7 was used we find the maximum of $f_6(9) = 9$ and $f_6(9-10)+10 = -\infty$. The maximum is $f_6(9)$. We continue until we reach the f_0 line.

i	1	2	3	4	5	6	7
w_i	2	3	4	6	8	9	10
p_i	2	3	4	6	8	9	10

Table 3. Items obtained using the transformation for the example in Section 4

This process is illustrated in Table 5, where X represents the remaining weight in the knapsack, $f_{i-1}(X)$ and $f_{i-1}(X - w_i) + p_i$ represent the alternatives between including or not the object and max represent the chosen value. The recurrence is also illustrated in Table 6, where the value chosen at step i is highlighted with a box and a subscript indicating the step. The solution is given by object 6 which has weight 9 and profit 9. This object corresponds to the multiset of rules composed of three times the rule with $3a$ as left-hand side. In conclusion, the algorithm tells us we can apply the rule with $3a$ three times and process only $9a$ out of $10a$.

Thus a complexity of a single evolution step for a membrane is

$$O(|w| \cdot S) + O(|w|^2 \cdot S) + O(|w| \cdot S).$$

X	0	1	2	3	4	5	6	7	8	9	10
f_0	0	0	0	0	0	0	0	0	0	0	0
f_1	0	0	2	2	2	2	2	2	2	2	2
f_2	0	0	2	3	3	5	5	5	5	5	5
f_3	0	0	2	3	4	5	6	7	7	9	9
f_4	0	0	2	3	4	5	6	7	8	9	10
f_5	0	0	2	3	4	5	6	7	8	9	10
f_6	0	0	2	3	4	5	6	7	8	9	10
f_7	0	0	2	3	4	5	6	7	8	9	10

Table 4. Solution to the knapsack instance in Table 3

i	7	6	5	4	3	2	1
X	9	9	0	0	0	0	0
$f_{i-1}(X)$	9	9	0	0	0	0	0
$f_{i-1}(X-w_i)+p_i$	$-\infty$	9	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
max	$f_6(9)$	$f_5(0)$	$f_4(0)$	$f_3(0)$	$f_2(0)$	$f_1(0)$	$f_0(0)$

Table 5. The backtracking process

X	0	1	2	3	4	5	6	7	8	9	10
f_0	$\boxed{0}_7$	0	0	0	0	0	0	0	0	0	0
f_1	$\boxed{0}_6$	0	2	2	2	2	2	2	2	2	2
f_2	$\boxed{0}_5$	0	2	3	3	5	5	5	5	5	5
f_3	$\boxed{0}_4$	0	2	3	4	5	6	7	7	9	9
f_4	$\boxed{0}_3$	0	2	3	4	5	6	7	8	9	10
f_5	$\boxed{0}_2$	0	2	3	4	5	6	7	8	9	10
f_6	0	0	2	3	4	5	6	7	8	$\boxed{9}_1$	10
f_7	0	0	2	3	4	5	6	7	8	$\boxed{9}_0$	10

Table 6. Finding the solution to the knapsack instance in Table 3

After all membranes have evolved through these three stages, we need to distribute the resources produced by them to show how the multiset of each membrane evolves. Thus we seek to find a relation between the contents of two consecutive configurations. Formally for two configurations $C_1 = (\mu, w_{i_1}, \ldots, w_{i_k})$, $C_2 = (\mu', w'_{j_1}, \ldots, w'_{j_l})$, where $C_1 \Rightarrow C_2$ we need to express the relation between $w'_{j_p} p = \overline{1,l}$ and $w_{i_q} q \in \overline{1,k}$. The contents of a membrane change from the application of rules. Following the definition of a rule, we see that we have four different situations for a new produced symbol: the symbol remains in the membrane, the symbol goes to the parent membrane, the symbol goes to a specific membrane, and the membrane dissolves passing all its contents to the parent. We know that μ' is obtained from μ by dissolving some of the membranes, thus $l \leq k$. We introduce a function $t : \overline{1,k} \to \overline{1,l}$ where $t(p) = 1$ if the membrane with label p from μ is not dissolved and $t(p) = 0$ otherwise. We introduce the following notations:

- w_i^{alloc} the multiset allocated by the resource allocator for membrane i;
- $selected : \mathbb{N}^2 \to \mathbb{N}$, $selected(i,j) = k$, where $k \in \{k \cdot u_j \mid \exists u_j \to v_j \in R_i \wedge w_j = k \cdot u_j \wedge w_j$ has been allocated to rule $j\}$ representing the number of times rule $u_j \to v_j \in R_i$ has been selected by the resource allocator;
- $w_i^{here} = \bigcup\{k \cdot v_j(a_{here}) \cdot a \mid \exists u_j \to v_j \in R_i \wedge k = selected(i,j)\}$ representing the multiset of objects produced by the selected rules which remain in i;
- $w_i^{in} = \bigcup\{k \cdot s \cdot a \mid \exists u_j \to v_j \in R_l \wedge k = selected(l,j) \wedge s = v_j(a_{in_i}) + v_j(a_{out}),\ i$ is the parent of $l\}\}$ representing the multiset of objects which have been produced by other membranes and have been transported to membrane i;
- $w_i^{dis} = \bigcup\{w \mid \exists u_j \to v_j\delta \in R_l \wedge selected(l,j) > 0 \wedge w$ the contents of $l \wedge i$ is the parent of $l\}$ representing the multiset of objects produced by rules which dissolve a membrane.

Thus we have the following relations:

$$w'_{j_p} = \begin{cases} w_{j_p} - w_{j_p}^{alloc} + w_{j_p}^{here} + w_{j_p}^{in} + w_{j_p}^{dis} & , \quad t(j_p) = 1 \\ 0 \quad , \quad \text{otherwise} \end{cases} \tag{7}$$

This means that the local symbols are first transported, followed by the symbols from other membranes, and finally the symbols produced by dissolving a membrane. We do this in order to ensure that the produced symbols reach their destination membrane. If we do not consider dissolution as the last operation, the symbols that were supposed to reach other membranes would pass on to the parent of the dissolving membrane.

6 Conclusion

In this paper we describe the computational complexity of the simple P system in classical complexity theory, extending the approach shortly presented in [5]. We show that the complexity of certain membrane systems called *simple P systems* can be studied using the classical complexity theory. The evolution of such a system is studied by using a resource allocator which solves the resource allocation problem using a well-known combinatorial problem.

The allocation of the resources to the rules is an important step in the nondeterministic and maximally parallel evolution of a simple P system. We consider the static allocation of resources towards the parallel application of the rules, and study the computational complexity of a subclass of P systems by reducing the resource allocation problem to the knapsack problem.

Trading space for time (as many models of natural computing), one can show that **PMC** = **PSPACE**, where **PMC** is the class of problems which can be solved in polynomial time by P systems of a given type [8, 9]. Membrane computing brings **PSPACE** to polynomial time in the sense that given a problem $X \in$ **PSPACE** there exists a deterministic Turing machine that constructs in polynomial time Π_X,

a P system that solves X in polynomial time. Computing **PSPACE** in polynomial time means that we have a family of membrane systems for a given **PSPACE** problem such that the n-th membrane system solves the problem in polynomial time for inputs of size less than or equal to n. Recently, Sosik and Rodriguez-Paton provide a characterization of **PSPACE** by showing that confluent P systems with active membranes solve in polynomial time exactly the class of problems **PSPACE** [10].

References

1. O. Andrei, G. Ciobanu, D. Lucanu. A Rewriting Logic Framework for Operational Semantics of Membrane Systems, *Theoretical Computer Science* vol. 373, 163-181, 2007.
2. O. Agrigoroaiei, G. Ciobanu. Rewriting Logic Specification of Membrane Systems with Promoters and Inhibitors, to appear in *Electronic Notes of Theoretical Computer Science*, 2008.
3. C.S. Calude, Gh. Păun. Bio-steps Beyond Turing. *BioSystems* vol.77, 175-194, 2004.
4. G. Ciobanu, Gh. Păun, M.J. Pérez-Jiménez. *Application of Membrane Computing*, Springer, 2006.
5. G. Ciobanu, M. Gontineac. Mealy Membrane Automata and P Systems Complexity. In M.A.Gutierrez-Naranjo, Gh.Păun, M.J.Perez-Jimenez (Eds.): *Cellular Computing; complexity aspects*, ESF PESC Exploratory Workshop, Fenix Editora, Sevilla, 149-164, 2005.
6. T.H. Cormen, C.E. Leiserson, R.L. Rivest. *Introduction to Algorithms*, MIT Press, 1990.
7. Gh. Păun. *Membrane Computing. An Introduction*, Springer, 2002.
8. Gh. Păun. P Systems with Active Membranes: Attacking NP Complete Problems, *J. Autom. Lang. Comb.* vol.6(1), 75-90, 2001.
9. M. Perez Jimenez, A.R. Jimenez, F. Sancho-Caparrini. Complexity Classes in Models of Cellular Computing with Membranes, *Natural Computing* vol.2, 265-285, 2003.
10. P. Sosik, A. Rodriguez-Paton. Membrane Computing and Complexity Theory: A characterization of PSPACE. *Journal of Computer and System Sciences* vol 73(1), 137-152, 2007.