Insertion Sort with Self-reproducing Comparator P System

Davood P.Y.B.
LPU–Laguna
Calamba City, Laguna
davoodpb@gmail.com

N.P. Balba
LPU–Laguna
Calamba City, Laguna
neilbalba@yahoo.com

J.P. Pabico
Institute of Computer Science
UP Los Baños
jppabico@up.edu.ph

ABSTRACT
We present in this paper a self-reproducing comparator P system that simulates insertion sort. The comparator \( \Pi \) is a degree–2 membrane and structured as \( \mu = [c_0|h_1|_h_2],h_3 \). A maximizing \( \Pi \) compares two multisets \( a \) and \( b \) where min(\( |a|,|b| \)) is stored in compartment \( h_1 \) while max(\( |a|,|b| \)) is stored in compartment \( h_2 \). A conditional reproduction rule triggers \( \Pi \) to clone itself out via compartment division followed by endocytosis of the cloned compartment. We present the process of sorting as a collection of transactions implemented in hierarchical levels where each level has different concurrent or serialized steps.

1. INTRODUCTION
Membrane computing (MC) is a computer science theoretical discipline which aims to develop new computational models from the physiological processes in biological cells, particularly cellular membranes. The acceptance of MC among theoretical computer scientists grows swiftly starting in 1998 when Gheorghe Paun introduced the idea that a biological entity possesses computing capability [Paun, 2000]. The initial goal of MC was to learn from the seemingly computational aspects of the physiological processes in biological cells.

Paun officially proposed MC in 2000 [Paun, 2000] and after that various types of membrane systems – known as P systems – were defined, all of them inspired from the computational aspects of bio-chemical processes. The theoretical aspects translated to practical applications as numerous researchers report its applicability to solving computational problems in biomedicine, linguistics, computer graphics, economics, approximate optimization, and cryptography. The understanding of MC has hasten significantly with the introduction of several software products for simulating and implementing P systems such as SNUPS (simulator of numerical P systems) and SRP-SUMGPU (simulation of recognizer P systems by using Manycore GPU [Gutierrez-Naranjo et al., 2006, Martinez del Amor et al., 2009].

1.1 Insertion Sort
Insertion sort is a comparison-based algorithm in which the elements of the input list are sorted one at a time. In this algorithm, the sorted sub-list is always maintained in the lower position of the list. Then the new item is inserted into the previous sorted sub-list such that the new sub-list is also sorted.

We consider the list with one item as a sorted list. Then we iterate, considering one item of the list each repetition and growing the sorted sub-list. In other words, at each iteration, we work on an item from the input list and find its position in sorted sub-list comparing and swapping (if needed) the item with the elements of sorted sub-list started from the end. The iteration stops when swapping stops. The new sub-list is sorted and ready for the next iteration and new item from the list [Astrachan, 2003, Bender et al., 2006, Bentley, 1999, Knuth, 1998].

Insertion sort is slow compared to the advanced sorting algorithms such as quicksort, merge sort and heapsort [Bender et al., 2004]. Based on time complexity and number of comparison, insertion sort is as slow as bubble sort is; the worst case and the average case for both of insertion sort and bubble sort is \( O(n^2) \) while the best case is \( O(n) \) where \( n \) is the number of elements of the input list. Although regarding time complexity, insertion sort is weak as bubble sort is, it has some strengths that make it bright despite being disregarded by some computer scientists [Astrachan, 2003, Knuth, 1998]. The strengths of insertion sort can be listed as:

1. Simplicity: Jon Bentley used C programming language and implemented insertion sort only in three lines. He also implemented the optimized version of insertion sort in five lines [Bentley, 1999].

2. Adaptiveness: insertion sort efficient for lists in which the elements are already considerably sorted. In this case the time complexity of insertion sort is \( O(nm) \) when each element in the input list is no more than \( m \) places far from its sorted
2. CELL-LIKE P SYSTEMS

Cell-like P system\(^1\) consists of many membranes arranged hierarchically. These membranes bound compartments. The compartments are the area where multisets of abstract objects are placed. The multisets are sets of objects (or symbols) with multiplicities, while the objects are the “chemicals” in the compartments, “swimming” in some substance in liquid form Ardelean et al. 2008, Wu et al. 2016. The compartments are identified with its index \(i\) and is symbolized as \([i]\). A membrane \(j\) with \(n\) compartments inside it and structured as a flat rooted tree at \(j\) (Figure 1a) can be written as

\[
[j[k+1][k+2] \cdots [k+n]k+n].
\]

(1)

Meanwhile, a deep rooted membrane \(i\) with one compartment inside it but that in itself deep-rooted (Figure 1b) is symbolized as

\[
i[i+1[i+j+2 \cdots [j+n]j+n+ \cdots j+2j+1].
\]

(2)

A membrane with only one compartment inside it is both flat- and deep-rooted. In general, a membrane’s structure is a combination of these two basis structures.

A multiset can be seen as a string \(S\) where its multiplicity (number of each symbol \(|S| = \text{length}(S)\)) is significant, not the order of symbols \(s \in S\). Since the objects are swimming and moving freely inside a compartment, the permutation or order of objects is not important Chen et al. 2015, Singh et al. 2014, Wu et al. 2016. For example, consider the multisets \(M_1 = aabbc\), \(M_2 = bacbb\), and \(M_3 = cbab\). All of these multisets are the same (i.e., \(M_1 = M_2 = M_3\)) because in each of the multisets, the number of object \(a\), shortened as \(|a|\) is two, \(|b| = 3\), and \(|c| = 1\). As it can be observed the order of objects in multisets are not important since the objects are inside the liquid and can freely move.

It is noticeable that, in each compartment there are some rules \(R\) and the objects inside the compartment evolve according to these rules. The number of objects in the multiset may change based on the application of the rules. Moreover, the rules do not only direct how the objects change but also how the objects communicate across membranes Chen et al. 2013, Singh et al. 2014.

3. Stability: insertion sort does not change the relative order of items in the input list with equal indices Astrachan, 2003, Bender et al. 2006, Bentley 1999, Knuth 1998.

4. In-placement: insertion sort needs only one extra unit memory space for swapping of two elements of the input list. The extra unit is small as size of elements of input list.

5. Online-ness: insertion sort sorts a list as it receives the elements of the list one-by-one Astrachan, 2003, Bentley, 1999, Knuth, 1998.

We call this change a “transmutation.” For example, consider the multisets \(M = aabbc\) reduced above (i.e., all multisets have the string \(aabbbc\) R). According to the first rule, \(M_1\) will be changed to another multiset \(M_2 = anbcb\). According to \(R_3\), one \(a\) and one \(c\) are transmuted to one \(d\). It is clearly seen that based on \(R_1\), the objects and their number from the original multiset were changed: \(|a| = 1\), \(|b| = 3\), and \(|c| = 1\) (we do not talk about object \(c\) any more, since \(|c| = 0\)). In general, given two strings \(S_n = s_1s_2 \ldots s_n\) and \(S_m = s_{n+1}s_{n+2} \ldots s_{n+m}\), a transmuting rule has the form

\[
s_1s_2 \ldots s_n \rightarrow s_{n+1}s_{n+2} \ldots s_{n+m}
\]

(3)

![Figure 1: Two general types of hierarchical structure of membranes rooted at \([i]\): (a) flat structure; and (b) deep structure.](image)

2.1 Transmuting Objects

The first type of rules change some number of objects to some another number of the same or different objects. We call this change a “transmutation.” For example, consider the rule \(R_1 : ca \rightarrow d\) and the objects \(aabbc\) in our previous multisets \(M_1, M_2\) and \(M_3\) that we introduced above (i.e., all multisets have the string \(aabbc\)). With the application of \(R_1\), this string will be changed to another multiset \(M_2 = anbc\). According to \(R_1\), one \(a\) and one \(c\) are transmuted to one \(d\). It is clearly seen that based on \(R_1\), the objects and their number from the original multiset were changed: \(|a| = 1\), \(|b| = 3\), and \(|d| = 1\) (we do not talk about object \(c\) any more, since \(|c| = 0\)). In general, given two strings \(S_n = s_1s_2 \ldots s_n\) and \(S_m = s_{n+1}s_{n+2} \ldots s_{n+m}\), a transmuting rule has the form

\[
s_1s_2 \ldots s_n \rightarrow s_{n+1}s_{n+2} \ldots s_{n+m}
\]
where the objects as \( a_1, a_2, \ldots, a_n \) in an \( n \)-long string \( S_n \) were transmuted into objects \( s_{n+1}, s_{n+2}, \ldots, s_{n+m} \) of a different \( m \)-long string \( S_m \).

### 2.2 Translocating Objects
Some rules change the locations of the objects. These rules transmute the objects from some membranes to some other membranes in a manner similar to how two or more processes exchange data in a method called *interprocess communication*. In physiological processes, common to all biological systems, such transfer is called as *cytosis*. In this paper, we call this change as “translocation,” which is illustrated by the following example.

Consider two rules \( R_2 : a \rightarrow e \) and \( R_3 : d \rightarrow f \) that are applied independently to our example multiset \( M_4 \) with string \( abbbd \) that is located in membrane \( [focuss]_{focuss} \). The membrane is located in a deeply-rooted membrane whose structure is defined by \( [\text{out}]_{focuss}[\text{in}]_n [\text{focuss}]_{\text{out}} \). The multiset is transmuted to \( bbgg \), since object \( a \) is in itself transmuted to \( c \) and at the same time translocated to another compartment \([\text{in}]_n \) inside the current membrane \([\text{focuss}]_{\text{focuss}} \). Meanwhile, object \( d \) is transmuted to \( f \) and \( g \), with \( f \) transferred outside to \([\text{out}]_{\text{out}} \) of the current membrane \([\text{focuss}]_{\text{focuss}} \). In general, a translocating rule has the form

\[
s_1 s_2 \ldots s_n \rightarrow s_{n+1} s_{n+2} [s_{n+3} s_{n+4}] \ldots s_{n+m} \ 
\]

In biological processes, a translocation of objects from \([\text{focuss}]_{\text{focuss}} \) to \([\text{in}]_n \) is called as *endocytosis*, while from \([\text{focuss}]_{\text{focuss}} \) to \([\text{out}]_{\text{out}} \) is called *exocytosis*.

### 2.3 Cloning Membranes
At some situations that could be useful in computation, we allow a membrane \( \mu_1 \) to reproduce by cloning itself to another membrane \( \mu_2 \). Cloning allows the duplication of \( \mu_1 \) to several copies of itself, similar to how the biological cells divide in a process called mitosis. Three scenarios can be inferred from the cloning action with respect to the initial location of the cloned membrane: (1) outside, (2) beside, and (3) inside of the cloned membrane. Let \( R_{\text{out}}, R_{\text{side}}, \) and \( R_{\text{in}} \) be the rules that respectively define these scenarios. Further, let \( \mu_1 = [1], \mu_2 = [2] \) be the original membrane to be cloned, \( \mu_3 = [3] \) be the cloned membrane, and \( \mu_4 \) represent one or more membranes within \( \mu_1 \). Then, in general:

\[
\begin{align*}
R_{\text{out}} & : [1] \mu_1 \rightarrow [2][1] \mu_2, \mu_4 & (5) \\
R_{\text{side}} & : [1] \mu_1 \rightarrow [1] \mu_2 [2] \mu_2 & (6) \\
R_{\text{in}} & : [1] \mu_1 \rightarrow [1] [2] \mu_2 \mu_4 & (7)
\end{align*}
\]

### 2.4 Computation by Rule Application
The rules belonging to all three types can be implemented and applied in many ways. These rules emulate biological processes wherein biochemical reactions happen in concurrently. Thus, biological processes exhibit maximal parallelism. However, since not all computations are explicit parallelizable, the following modes were defined to describe the type of concurrency a computation has: sequential, minimal parallel, bounded parallel, and maximal parallel. In sequential mode, only one rule is used in each computation step. This is because there are computational steps that are inherently serial because of input-output dependencies. In minimal parallel mode, at least one rule must be used when a set of rules can be used concurrently. In bounded parallel mode, the number of membranes that will compute or the number of rules to be used is restricted. In all modes mentioned, objects to apply the rule to, as well as the rules themselves, are chosen non-deterministically [Ardelean et al., 2008, Gutierrez-Naranjo et al., 2006, Martinez del Amor et al., 2009, Paun, 2000, 2006, Wu et al., 2016].

In MC, a collection of transitions creates a computation. A computation generates a result as long as it halts, i.e., to have reached to a configuration where no rule can be applied [Ardelean et al., 2008, Gutierrez-Naranjo et al., 2006, Martinez del Amor et al., 2009, Paun, 2000, 2006, Wu et al., 2016].

### 3. COMPARATOR P SYSTEM
We now define a membrane that can sort two integers we call a comparator P System (or \( \Pi_X \) for short). Here, \( \Pi_X \) is able to sort two integers \( \alpha \) and \( \beta \), such that \( \alpha = |S_1| \) and \( \beta = |S_2| \). The multisets \( S_1 \) and \( S_2 \) are homogeneous multisets, i.e., they contain only one type of objects, and the object in \( S_1 \) is different from the object in \( S_2 \). For brevity, we represent the multiplicity of the objects in the multisets as \( \alpha \) to mean that object \( a \) has \( \alpha \) copies in the multiset. For example, \( s_1 = aaaaa \) and \( s_2 = bbb \) respectively represent the integers 5 and 3. The structure of \( \Pi_X \) is \( \mu_X = [a_0]_1 [a_1]_2 [a_2]_2 [a_3]_0 \) [Ardelean et al., 2008].

At the beginning of the computation, two multisets \( \alpha \) and \( \beta \) are in \([a_0]_0 \). The compartments \([a_1]_1 \) and \([a_2]_2 \) are empty. Then, all transactions are performed in two steps in order using the following ruleset:

1. \( R_{1}[a_0] : a b \rightarrow a [a_1] b [a_2] \)
2. \( R_{2}[a_0] : a \rightarrow b [a_2] \)
\( R_{3}[a_0] : b \rightarrow [a_2]_2 \)
\( R_{4}[a_1] : b \rightarrow b [a_0] \)

In step 1, the rule \( R_{1}[a_0] \) is applied to the multisets in \([a_0]_0 \) translocating an equal number of objects \( a \) and \( b \) to \([a_2]_2 \). If \( x > y \), then \( (x - y) \) of \( a \) are left behind at \([a_0]_0 \) and all \( y \) of \( b \) are in \([a_2]_2 \). If \( x = y \), then all objects \( a \) and \( b \) are moved to \([a_1]_2 \). If \( x < y \), then \((y - x) \) of \( b \) are left behind at \([a_0]_0 \) and all \( x \) of \( a \) are in \([a_1]_2 \).

In step 2, the rules \( R_{2}[a_0] \), \( R_{3}[a_0] \), and \( R_{4}[a_1] \) can at least be minimally applied in parallel. \( R_{2}[a_0] \) translocates the \( a \)'s in \([a_0]_0 \) to \([a_2]_2 \) and at the same time transmutes them as \( b \)'s. \( R_{3}[a_0] \) translocates the \( b \)'s in \([a_0]_0 \) to \([a_2]_2 \). \( R_{4}[a_1] \) translocates the \( b \)'s in \([a_1]_2 \) to \([a_0]_0 \).

\(^2\)At best maximally applied in parallel.
Upon closer investigation, it may seem that \( R_3[h_0] \) has a dependency to \( R_4[h_1] \), in which case we can apply the rules as minimally parallel. However, if we assume that all rules will only fire as long as an object in the corresponding compartment is present, then we can always assume non-dependency and therefore consider the process as maximally parallel. However, when there is only one object \( b \) remaining in \([h_1]h_1\), then \( R_4[h_1] \) fires first followed by \( R_1[h_0] \). This seemingly serial order of the two supposedly concurrent rules instantaneously transfers the remaining \( b \) from \([h_1]h_1\) to \([h_2]h_2\) through \([h_0]h_0\). The time spent, as well as the overhead cost, for passing through transient membrane \([h_0]h_0\) are considered zero.

When no more rule among the ruleset can be applied, then the halting state happens. In this case, the larger between \( x \) and \( y \) will be in \([h_2]h_2\) while the lower will be in \([h_1]h_1\) [Ardelean et al., 2008]. We call each \( \Pi^+ \) as a maximizing comparator and is represented as \( \Pi^+ \).

We can likewise define a minimizing \( \Pi^- \) as \( \Pi^- \) where the location of the lower and the higher numbers are reversed in the compartments within \( \Pi^- \). In this case, \( \max(x, y) \) will be in \([h_1]h_1\), while \( \min(x, y) \) will be in \([h_2]h_2\).

### 4. Membrane Sorter

We now propose a membrane sorter containing several modified \( P^+ \)'s that uses the insertion algorithm to sort a list of \( n \) integers \( \{x_1, x_2, \ldots, x_n\} \), where each integer \( x_i \) is represented as the length \( |S_i| \) of a multiset \( S_i \). We introduce a modification to the \( P^+ \) described above introducing a rule that implements a cloning out, but with a little twist. We will only allow the clone to copy the first level children compartment of its parent compartment. We define a function \( \text{root}(\mu_1) \) that returns the root of a compartment whose structure is \( \mu_1 \). Given that \( \mu_1 = [1 \, \mu_2, 1] \), where \( \mu_2 \) might be flatly- or deeply-rooted structure, then \( \text{root}(\mu_1) = [1 \, 1] \).

Our additional rule which must be triggered conditionally is:

\[
R_{[h_0]} : \Pi^+_0 \rightarrow [e_0 \, e_1 \, \Pi^+_1]_e_0 \quad (8)
\]

Here, the compartment \([h_0]h_0\) of the parent \( \Pi^+_0 \) acts as the \([e_2]e_2\) of the cloned \( \Pi^+_1 \) such that the structure of the clone \( \mu_c = \left[ e_0 \, e_1 \, h_0 \, h_1 \, h_2 \, h_2 \right]_e_0 \). Note here that the ruleset of the parent is inherited by the clone. Note further that the firing of \( R_{[h_0]} \) is conditional such that it will not fire if \([h_0]h_0\) is already contained in \([e_0]e_0\).

We now present the ruleset that will allow for their recursive and semantically correct implementation:

1. Follow the ruleset for \( \Pi^+_0 \) described above.
2. (a) \( R_{[h_1]} : b \rightarrow b_{[h_0]} \)
   (b) \( R_{[h_0]} : \) Follow the reproduction rule in Equation 5
   (c) \( R_{[h_0]} : b \rightarrow b_{[e_0]} \)
3. (a) \( R_{[h_0]} : b \rightarrow b_{[e_0]} \)

Outside of \([e_0]e_0\) is the environment of the membrane \([e]e\) which contains the multisets whose respective lengths represent the integers to be sorted. The environment member could be \([e_0]e_0\) owns cloned outer membrane. When there are no more multisets remaining in \([e]e\) and there are no more rules to implement, then the sorting operation stops. Compartment \([h_0]h_0\) will have a duality of function: (1) as the outer membrane of \( \Pi^+ \) and (2) as the compartment \([c_1]c_0\) of the cloned \( \Pi^+_c \). It is important to keep track of the rules to fire so that the meaning will still be semantically correct.

### 5. Conclusion

This paper presents a membrane-sorter that follows the natural processes of the insertion sort algorithm. The membrane-sorter contains a deeply-rooted comparator P systems \( \Pi^+_c \). Each \( \Pi^+_c \) compares the respective sizes of homogeneous multisets and swaps their compartments in levels 1, 2, and 3 of the proposed ruleset. The comparisons and swaps are performed until the input list is sorted. Similar to insertion sort, the membrane-sorter sorts the list online.

### 6. Acknowledgment

The respectfully extend our profound gratitude and appreciation to the administrators of Lyceum of the Philippines University – Laguna (LPU-Laguna), who contributed assistance to the fulfillment of this research paper:

1. Dr. Peter Laurel – President
2. Engr. Ricky Bustamante – Dean, College of Engineering and Computer Studies
3. Mrs. Gerby Muya – Director, Research and Statistics Center

Without the help and support of these persons, this research might not be performed.

### References

Ioan I. Ardelean, Rodica Ceterchi, and Alexandru Ioan Tomescu. A biological perspective on sorting with P systems. In Proceedings of the 6th Brainstorming Week on Membrane Computing, pages 1–9, 2008. ISBN: 978-84-612-4429-4.

Owen Astrachan. Bubble sort: An archaeological algorithmic analysis. In Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education, pages 1–5, 2003. doi: 10.1145/792548.611918.

Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro. Insertion sort is \( \mathcal{O}(n \log n) \). Theory of Computing Systems, 39:391–397, 2006.

Jon Bentley. Programming Pearls. Addison-Wesley, 2nd edition, 1999. ISBN: 0201657880.
Ke Chen, Jun Wang, Ming Li, Jun Ming, and Hong Peng. Cell-like fuzzy P system and its application of coordination control in micro-grid. In *Proceedings of the 10th International Conference on Bio-inspired Computing – Theories and Applications*, pages 18–32, 2015.

Miguel Angel Gutierrez-Naranjo, Mario J. Perez-Jimenez, and Agustin Riscos-Núñez. Available membrane computing software. In *Applications of Membrane Computing*, chapter 15, pages 411–436. Springer, 2006.

Donald Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison–Wesley, 2nd edition, 1998.

Miguel Angel Martinez del Amor, Ignacio Perez-Hurtado de Mendoza, Mario J. Perez-Jimenez, Jose M. Cecilia, Gines D. Guerrero, and Jose M. Garcia. Simulation of recognizer P systems by using manycore GPUs. In *Proceedings of the 7th Brainstorming Week on Membrane Computing*, volume 2, pages 45–57, 2009. ISBN: 9788461328369.

Gheorghe Paun. Computing with membrane. *Journal of Computer and System Sciences*, 61(1):108–143, 2000. doi: 10.1006/jcss.1999.1693.

Gheorghe Paun. Introduction to membrane computer. In *Applications of Membrane Computing*, chapter 1, pages 1–42. Springer, 2006.

Garima Singh, Kusum Deep, and Atulya K. Nagar. Cell-like P-systems based on rules of particle swarm optimization. *Applied Mathematics and Computation*, 246:546–560, 2014. doi: 10.1016/j.amc.2014.08.027.

Tingfang Wu, Zhiqiang Zhang, Gheorghe Paun, and Linqiang Pan. Cell-like spiking neural P systems. *Theoretical Computer Science*, 623:180–189, 2016. doi: 10.1016/j.tcs.2015.12.038.