Int. J. of Computers, Communications & Control, ISSN 1841-9836, E-ISSN 1841-9844
Vol. III (2008), Suppl. issue: Proceedings of ICCCC 2008, pp. 480-485

Simulating NEPs in a cluster with jNEP

Emilio del Rosal, Rafael Nuñez, Carlos Castañeda, Alfonso Ortega

Abstract: This paper introduces *jNEP*: a general, flexible, and rigorous implementation of NEPs (the basic model) and some interestenting variants; it is specifically designed to easily add the new results (filters, stopping conditions, evolutionary rules, and so on) of the research in the area. jNEP is written in Java; there are two different versions that implement the concurrency of NEPs by means of the Java classes *Process* and *Thread*s respectively. There are also extended versions that run on clusters of computers under JavaParty. *jNEP* reads the description of the currently simulated NEP from a XML configuration file. This paper shows how *jNEP* tackles the SAT problem with polynomial performance by simulating an ANSP.
Keywords: NEPs, natural computing, simulation, clusters of computers

1 Introduction

1.1 NEPs

NEP stands for *Network of Evolutionary Processors*. NEPs are an abstract model of distributed/parallel symbolic processing presented in [1, 2]. NEPs are inspired by biological cells. These are represented by words which describe their DNA sequences. Informally, at any moment of time, the evolutionary system is described by a collection of words, where each word represents one cell. Cells belong to species and their community evolves according to mutations and division which are defined by operations on words. Only those cells are accepted as surviving (correct) ones which are represented by a word in a given set of words, called the genotype space of the species. This feature parallels the natural process of evolution. Each node in the net is a very simple processor containing words which performs a few elementary tasks to alter the words, send and receive them to/from other processors. Despite the simplicity of each processor, the entire net can carry out very complex tasks efficiently. Many different works demonstrate the computational completeness of NEPs [4][10] and their ability to solve NP problems with linear or polynomial resources [11][2]. The emergence of such a computational power from very simple units acting in parallel is one of the main interests of NEPs.

NEPs can be used to accept families of languages. When they are used in this way they are called Accepting NEPs (ANEPs). Several variants of NEPs have been proposed in the scientific literature. NEP (the original model) [2], hibrid nets of evolutionary processor (HNEP) [4] and nets of splicing processors NEPS or NSP [10]. This last model uses a splicing processor, which adds a new operation (splicing rules) to mimic crossover in genetic systems. In section 3.1 we show an example of ANSP (the accepting variant of NSPs) solving the SAT problem. Nevertheless, all of them share the same general characteristics.

A NEP is built from the following elements: a) a set of symbols which constitutes the alphabet of the words which are manipulated by the processors, b) a set of processors, c) an underlying graph where each vertex represents a processor and the edges determine which processors are connected so they can exchange words, d) an initial configuration defining which words are in each processor at the beginning of the computation and e) one or more stopping rules to halt the NEP.

An evolutionary processor has three main components: a) a set of evolutionary rules to modify its words, b) some input filters that specifies which words can be received from other processors and c) an output filter that delimits which words can leave the processor to be sent to others. The variants of NEPs mainly differ in their evolutionary rules and filters. They perform very simple operations, like altering the words by replacing all the occurrences of a symbol by another, or filtering those words whose alphabet is included in a given set of words.

NEP's computation alternates evolutionary and communication steps: an evolutionary step is always followed by a communication step and vice versa. Computation follows the following scheme: when the computation starts, every processor has a set of initial words. At first, an evolutionary step is performed: the rules in each processor modify the words in the same processor. Next, a communication step forces some words to leave their processors and also forces the processors to receive words from the net. The communication step depends on the constraints imposed by the connections and the output and input filters. The model assumes that an arbitrary number of copies of each word exists in the processors, therefore all the rules applicable to a word are actually applied, resulting in a new word for each rule. The NEP stops when one of the stopping conditions is met, for example, when the set of words in a specific processor (the ouput node of the net) is not empty. A detailed formal description of NEPs can be found in [1], [4] or [10].

1.2 Clusters of computers

Running NEPs simulators on cluster is one of the possible ways of exploiting the inherent parallel nature of NEPs. The Java Virtual Machine (JVM), which can be considered the standard Java, cannot be run on clusters.

Several attempts have tried to overcome this limitation, for example: Java-Enabled Single-System-Image Computing Architecture 2 (JESSICA2) [8], the cluster virtual machine for Java developed by IBM (IBM cJVM) [3], Proactive PDC [12], DO! [9], JavaParty [6], and Jcluster [7].

The simulator described in this paper has been developed with both JVM and JavaParty.

2 jNEP

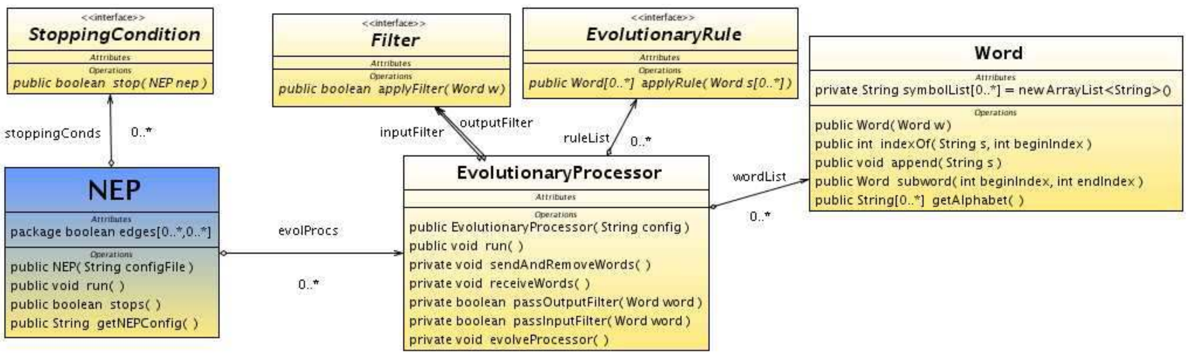
A lot of research effort has been devoted to the definition of different families of NEPs and to the study of their formal properties, such as their computational completness and their ability to solve NP problems with polynomial performance. However, no relevant effort, apart from [5], has tried to develop a NEP simulator or any kind of implementation. Unfortunately, the software described in this reference gives the possibility of using only one kind of rules and filters and, what is more important, violates two of the main principles of the model: 1) NEP's computation should not be deterministic and 2) evolutionary and communication steps should alternate strictly. Indeed, the software is focused in solving decision problems in a parallel way, rather than simulating the NEP model with all its details.

jNEP tries to fill this gap in the literature. It is a program written in Java which is capable of simulating almost any NEP in the literature. In order to be a valuable tool for the scientific community, it has been developed under the following principles: a) it rigorously complies with the formal definitions found in the literature; b) it serves as a general tool, by allowing the use of the different NEP variants and is ready to adapt to future possible variants, as the research in the area advances; c) it exploits as much as possible the inherent parallel/distributed nature of NEPs.

The jNEP code is freely available in http://jnep.e-delrosal.net.

2.1 jNEP design

jNEP offers an implementation of NEPs as general, flexible and rigorous as has been described in the previous paragraphs. As shown in figure 1, the design of the NEP class mimics the NEP model definition. In jNEP, a NEP is composed of evolutionary processors and an underlying graph (attribute *edges*) to define the net topology and the allowed inter processor interactions. The *NEP* class coordinates the main dynamic of the computation and rules the processors (instances of the *EvolutionaryProcessor* class), forcing them to perform alternate evolutionary and communication steps. It also stops the computation when needed. The core of the model includes these two classes, together with the *Word* class, which handles the manipulation of words and their symbols.



<<interface>>
StoppingCondition
Attributes
Operations
public boolean stop(NEP nep)
<<interface>>
Filter
Attributes
Operations
public boolean applyFilter(Word w)
<<interface>>
EvolutionaryRule
Attributes
Operations
public Word[0..*] applyRule(Word s[0..*])
Word
Attributes
private String symbolList[0..*] = new ArrayList<String>()
Operations
public Word(Word w)
public int indexOf(String s, int beginIndex)
public void append(String s)
public Word subword(int beginIndex, int endIndex)
public String[0..*] getAlphabet( )
stoppingConds
0..*
inputFilter
outputFilter
ruleList
0..*
NEP
Attributes
package boolean edges[0..*,0..*]
Operations
public NEP(String configFile)
public void run( )
public boolean stops( )
public String getNEPConfig( )
evolProcs
0..*
EvolutionaryProcessor
Attributes
Operations
public EvolutionaryProcessor(String config)
public void run( )
private void sendAndRemoveWords( )
private void receiveWords( )
private boolean passOutputFilter(Word word)
private boolean passInputFilter(Word word)
private void evolveProcessor( )
wordList
0..*

Figure 1: Simplified class diagram of jNEP

We keep *jNEP* as general and rigorous as possible by means of the following mechanisms: Java interfaces and the develop of different versions to widely exploit the parallelism availble in the hardware platform.

jNEP offers three interfaces: a) *StoppingCondition*, which provides the method *stop* to determine whether a *NEP* object should stop according to its state; b) *Filter*, whose method *applyFilter* determines which objects of class *Word* can pass it and c) *EvolutionaryRule*, which *appl*ies a *Rule* to a set of *Word*s to get a new set. *jNEP* tries to implement a wide set of NEPs' features. The *jNEP user guide* (http://jnep.e-delrosal.net) contains the updated list of filters, evolutionary rules and stopping conditions implemented.

Currently *jNEP* has two list of choices to select the parallel/distributed platform on which it runs (any combination of them is also available in http://jnep.e-delrosal.net). Concurrency is implemented by means of two different Java approaches: *Thread*s and *Process*es. The first needs more complex synchronization mechanisms. The second uses heavier concurrent threads. The supported platforms are standard JVM and clusters of computers (by means of JavaParty).

3 jNEP in practice

jNEP is written in Java, therefore to run jNEP one needs a Java virtual machine (version 1.4.2 or above) installed in a computer. Then one has to write a configuration file describing the NEP. The *jNEP user guide* (available at http://jnep.e-delrosal.net) contains the details concerning the commands and requirements needed to launch jNEP. In this section, we want to focus on the configuration file which has to be written before running the program, since it has some complex aspects important to be aware of the potentials and possibilities of jNEP.

The configuration file is an XML file specifying all the features of the NEP. Its syntax is described below in BNF format, together with a few explanations. Since BNF grammars are not capable of expressing context-dependent aspects, context-dependent features are not described here. Most of them have been explained informally in the previous sections. Note that the traditional characters <> used to identify non-terminals in BNF have been replaced by `[ ]` to avoid confusion with the use of the <> characters in the XML format.

- [configFile] ::= <?xml version="1.0"?> <NEP nodes="[integer]"> [alphabetTag] [graphTag] [processorsTag] [stoppingConditionsTag] </NEP>
- [alphabetTag] ::= <ALPHABET symbols="[symbolList]"/>
- [graphTag] ::= <GRAPH> [edge] </GRAPH>
- [edge] ::= <EDGE vertex1="[integer]" vertex2="[integer]"/> [edge]
- [edge] ::= λ
- [processorsTag] ::= <EVOLUTIONARY_PROCESSORS> [nodeTag] </EVOLUTIONARY_PROCESSORS>

The above rules show the main structure of the NEP: the alphabet, the graph (specified through its vertices) and the processors. It is worth remembering that each processor is identified implicitly by its position in the processors tag (first one is number 0, second is number 1, and so on).

- [stoppingConditionsTag] ::= <STOPPING_CONDITION> [conditionTag] </STOPPING_CONDITION>
- [conditionTag] ::= <CONDITION type="MaximumStepsStoppingCondition" maximum="[integer]"/> [conditionTag]
- [conditionTag] ::= <CONDITION type="WordsDisappearStoppingCondition" words="[wordList]"/> [conditionTag]
- [conditionTag] ::= <CONDITION type="ConsecutiveConfigStoppingCondition"/> [conditionTag]
- [conditionTag] ::= <CONDITION type="NonEmptyNodeStoppingCondition" nodeID="[integer]"/> [conditionTag]
- [conditionTag] ::= λ

The syntax of the stopping conditions shows that a NEP can have several stopping conditions. The first one which is met causes the NEP to stop. The different types try to cover most of the stopping conditions used in the literature. If needed, more of them can be added to the system easily. The *jNEP user guide* explains their semantics in detail.

- [nodeTag] ::= <NODE initCond="[wordList]" [auxWordList]> [evolutionaryRulesTag] [nodeFiltersTag] </NODE> [nodeTag]
- [nodeTag] ::= λ
- [auxWordList] ::= λ | auxiliaryWords="[wordList]"
- [evolutionaryRulesTag] ::= <EVOLUTIONARY_RULES> [ruleTag] </EVOLUTIONARY_RULES>
- [ruleTag] ::= <RULE ruleType="[ruleType]" actionType="[actionType]" symbol="[symbol]" newSymbol="[symbol]"/> [ruleTag]
- [ruleTag] ::= <RULE ruleType="splicing" wordX="[symbolList]" wordY="[symbolList]" wordU="[symbolList]" wordV="[symbolList]"/> [ruleTag]
- [ruleTag] ::= <RULE ruleType="splicingChoudhary" wordX="[symbolList]" wordY="[symbolList]" wordU="[symbolList]" wordV="[symbolList]"/> [ruleTag]

- [ruleTag] ::= λ
- [ruleType] ::= insertion | deletion | substitution
- [actionType] ::= LEFT | RIGHT | ANY
- [nodeFiltersTag] ::= [inputFilterTag] [outputFilterTag]
- [nodeFiltersTag] ::= [inputFilterTag]
- [nodeFiltersTag] ::= [outputFilterTag]
- [nodeFiltersTag] ::= λ
- [inputFilterTag] ::= <INPUT [filterSpec]/>
- [outputFilterTag] ::= <OUTPUT [filterSpec]/>
- [filterSpec] ::= type=[filterType] permittingContext="[symbolList]" forbiddingContext="[symbolList]"
- [filterSpec] ::= type="SetMembershipFilter" wordSet="[wordList]"
- [filterSpec] ::= type="RegularLangMembershipFilter" regularExpression="[regExpression]"
- [filterType] ::= 1 | 2 | 3 | 4

The preceding set of rules describe the elements of the processors: their initial conditions, rules, and filters. We have applied the same philosophy as in the case of stopping conditions, which means that our systems supports almost all kinds found in the literature at the moment. Future types can also be added. The reader may refer to the *jNEP user guide* for further detailed information.

- [wordList] ::= [symbolList] [wordList]
- [wordList] ::= λ
- [symbolList] ::= `a string of symbols separated by the character '\_'`
- [boolean] ::= true | false
- [integer] ::= `an integer number`
- [regExpression] ::= `a Java regular expression`

3.1 An example: solving the SAP problem with linear resources

Reference [10] describes a NEP with splicing rules (ANSP) which solves the boolean satisfiability problem (SAT) with linear resources, in terms of the complexity classes also present in [10]. We can use jNEP to actually build and run this ANSP. The following is a broad summary of the config file for such a ANSP, applied to the solution of the SAT problem for three variables. The entire file can be downloaded from jnep.e-delrosal.net.

```
<NEP nodes="9">
  <ALPHABET symbols="A_B_C_!A_!B_!C_AND_OR_(_)_[A=1]_[B=1]_[C=1]_[A=0]_[B=0]_[C=0]_#_UP_{_}_1"/>
  <!-- WE IGNORE THE GRAPH TAG TO SAVE SPACE. THIS NEP HAVE A COMPLETE GRAPH -->
  <STOPPING_CONDITION>
    <CONDITION type="NonEmptyNodeStoppingCondition" nodeID="1"/>
  </STOPPING_CONDITION>
  <EVOLUTIONARY_PROCESSORS>
    <NODE initCond="{_(_A_)_AND_(_B_OR_C_)_}" auxiliaryWords="{_[A=1]_# {_[A=0]_# {_[B=1]_#
                                                              {_[B=0]_# {_[C=1]_# {_[C=0]_#">  <!-- INPUT NODE -->
      <EVOLUTIONARY_RULES>
        <RULE ruleType="splicing" wordX="{" wordY="(" wordU="{_[A=1]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{" wordY="(" wordU="{_[A=0]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{" wordY="[A=0]" wordU="{_[B=0]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{" wordY="[A=0]" wordU="{_[B=1]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{" wordY="[A=1]" wordU="{_[B=0]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{" wordY="[A=1]" wordU="{_[B=1]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{" wordY="[B=0]" wordU="{_[C=0]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{" wordY="[B=0]" wordU="{_[C=1]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{" wordY="[B=1]" wordU="{_[C=0]" wordV="#"/>
        <RULE ruleType="splicing" wordX="{" wordY="[B=1]" wordU="{_[C=1]" wordV="#"/>
      </EVOLUTIONARY_RULES>
      <FILTERS>
        <INPUT type="4" permittingContext="" forbiddingContext="[A=1]_[B=1]_[C=1]_[A=0]_[B=0]_[C=0]_#_UP_{_}_1"/>
        <OUTPUT type="4" permittingContext="[C=1]_[C=0]" forbiddingContext=""/>
      </FILTERS>
    </NODE>
    <NODE initCond="">  <!-- OUTPUT NODE -->
      <EVOLUTIONARY_RULES>
      </EVOLUTIONARY_RULES>
      <FILTERS>
        <INPUT type="1" permittingContext="" forbiddingContext="A_B_C_!A_!B_!C_AND_OR_(_)"/>
        <OUTPUT type="1" permittingContext="" forbiddingContext="[A=1]_[B=1]_[C=1]_[A=0]_[B=0]_[C=0]_#_UP_{_}_1"/>
      </FILTERS>
    </NODE>
    <NODE initCond="" auxiliaryWords="#_[A=0]_} #_[A=1]_} #_} #_1_)_}"> <!-- COMP NODE -->
      <EVOLUTIONARY_RULES>
        <RULE ruleType="splicing" wordX="" wordY="A_OR_1_)_}" wordU="#" wordV="1_)_}"/>
        <RULE ruleType="splicing" wordX="" wordY="!A_OR_1_)_}" wordU="#" wordV="1_)_}"/>
        <RULE ruleType="splicing" wordX="" wordY="B_OR_1_)_}" wordU="#" wordV="1_)_}"/>
        <RULE ruleType="splicing" wordX="" wordY="!B_OR_1_)_}" wordU="#" wordV="1_)_}"/>
        <RULE ruleType="splicing" wordX="" wordY="C_OR_1_)_}" wordU="#" wordV="1_)_}"/>
        <RULE ruleType="splicing" wordX="" wordY="!C_OR_1_)_}" wordU="#" wordV="1_)_}"/>
        <RULE ruleType="splicing" wordX="" wordY="AND_(_1_)_}" wordU="#" wordV="}"/>
        <RULE ruleType="splicing" wordX="" wordY="[A=1]_(_1_)_}" wordU="#" wordV="[A=1]_}"/>
        <RULE ruleType="splicing" wordX="" wordY="[A=0]_(_1_)_}" wordU="#" wordV="[A=0]_}"/>
      </EVOLUTIONARY_RULES>
      <FILTERS>
        <INPUT type="1" permittingContext="1" forbiddingContext=""/>
        <OUTPUT type="1" permittingContext="" forbiddingContext="#_1"/>
      </FILTERS>
    </NODE>
    <NODE initCond=""  auxiliaryWords="#_1_)_} #_)_}">  <!-- A=1 NODE -->
      <EVOLUTIONARY_RULES>
        <RULE ruleType="splicing" wordX="" wordY="A_)_}" wordU="#" wordV="1_)_}"/>
        <RULE ruleType="splicing" wordX="" wordY="(_!A_)_}" wordU="#" wordV="UP"/>
        <RULE ruleType="splicing" wordX="" wordY="OR_!A_)_}" wordU="#" wordV=")_}"/>
        <RULE ruleType="splicing" wordX="" wordY="B_)_}" wordU="#" wordV="UP"/>
        <RULE ruleType="splicing" wordX="" wordY="C_)_}" wordU="#" wordV="UP"/>
      </EVOLUTIONARY_RULES>
      <FILTERS>
        <INPUT type="1" permittingContext="[A=1]" forbiddingContext="[A=0]_1"/>
        <OUTPUT type="1" permittingContext="" forbiddingContext="#_UP"/>
      </FILTERS>
    </NODE>
    <NODE initCond=""  auxiliaryWords="#_1_)_} #_)_}">  <!-- A=0 NODE -->
      <EVOLUTIONARY_RULES>
        <RULE ruleType="splicing" wordX="" wordY="OR_A_)_}" wordU="#" wordV=")_}"/>
        <RULE ruleType="splicing" wordX="" wordY="(_A_)_}" wordU="#" wordV="UP"/>
        <RULE ruleType="splicing" wordX="" wordY="!A_)_}" wordU="#" wordV="1"/>
        <RULE ruleType="splicing" wordX="" wordY="B_)_}" wordU="#" wordV="UP"/>
        <RULE ruleType="splicing" wordX="" wordY="C_)_}" wordU="#" wordV="UP"/>
      </EVOLUTIONARY_RULES>
      <FILTERS>
        <INPUT type="1" permittingContext="[A=0]" forbiddingContext="[A=1]_1"/>
        <OUTPUT type="1" permittingContext="" forbiddingContext="#_UP"/>
      </FILTERS>
    </NODE>
    <!-- NODES FOR 'B' AND 'C' ARE ANALOGOUS TO THOSE FOR 'A'. WE DO NOT PRESENT THEM TO SAVE SPACE-->
  </EVOLUTIONARY_PROCESSORS>
</NEP>
```

With this config file, at the end of its computation, jNEP outputs the interpretation which satisfies the logical formula contained in the file, namely:

```
    (_A_)_AND_(_B_OR_C_): {_[C=0]_[B=1]_[A=1]_} {_[C=1]_[B=1]_[A=1]_} {_[C=1]_[B=0]_[A=1]_}
```

This ANSP is able to solve any formula with three variables. The formula to be solved must be specified as the value of the *initCond* attribute for the input node.

4 Conclusions and further research lines

jNEP is one of the first and more complete implementations of the family of abstract computing devices called NEPs. *jNEP* simulates not only the basic model, but also some of its variants, and is able to run on clusters of computers.

In the future we plan to offer full acces to the cluster version by means of the web. We also plan to develop a graphic user interface to ease the definition of the NEP being simulated. *jNEP* will be used as a module in the design of an automatic programming methodology to design NEPs to solve a given problem.

Acknowledgement: This work was supported in part by the Spanish Ministry of Education and Science (MEC) under Project TSI2005-08225-C07-06.

References

[1] J. Castellanos, C. Martin-Vide, V. Mitrana, and J. M. Sempere. “Networks of evolutionary processors”. *Acta Informatica*, 39(6-7):517-529, 2003.

[2] Juan Castellanos, Carlos Martin-Vide, Victor Mitrana, and Jose M. Sempere. “Solving NP-Complete Problems With Networks of Evolutionary Processors.” *Connectionist Models of Neurons, Learning Processes and Artificial Intelligence : 6th International Work-Conference on Artificial and Natural Neural Networks*, IWANN 2001 Granada, Spain, June 13-15, Proceedings, Part I, 2001.

[3] http://www.haifa.il.ibm.com/projects/systems/cjvm/index.html

[4] E. Csuhaj-Varju, C. Martin-Vide, and V. Mitrana. “Hybrid networks of evolutionary processors are computationally complete.” *Acta Informatica*, 41(4-5):257-272, 2005.

[5] M. A. Diaz, N. Gomez Blas, E. Santos Menendez, R. Gonzalo, and F. Gisbert. “Networks of evolutionary processors (nep) as decision support systems.” In Fith International Conference. *Information Research and Applications*, volume 1, pages 192-203. ETHIA, 2007.

[6] http://wwwipd.ira.uka.de/JavaParty/

[7] http://vip.6to23.com/jcluster/

[8] http://i.cs.hku.hk/ wzzhu/jessica2/index.php

[9] Pascale Launay, Jean-Louis Pazat. “A Framework for Parallel Programming in Java.” *INRIA Rapport de Recherche* Publication Internet - 1154 decembre 1997 - 13 pages

[10] Florin Manea, Carlos Martin-Vide, and Victor Mitrana. “Accepting networks of splicing processors: Complexity results.” *Theoretical Computer Science*, 371(1-2):72-82, February 2007.

[11] Florin Manea, Carlos Martin-Vide, and Victor Mitrana. “All np-problems can be solved in polynomial time by accepting networks of splicing processors of constant size.” *DNA Computing*, pages 47-57, 2006.

[12] http://www-sop.inria.fr/sloop/javall/

Emilio del Rosal, Rafael Nuñez, Carlos Castañeda, and Alfonso Ortega
Universidad Autónoma de Madrid
Departamento de Ingeniería Informática
Av/Francisco Tomás y Valiente 7 - 28049 Madrid (Spain)
E-mail: emilio.delrosal@uam.es