Triangle: Language, Literature, Computation, n. 7, 2012
Publicacions Universitat Rovira i Virgili · ISSN: 2013-939X
https://revistes.urv.cat/index.php/triangle

NEPs Applied to Solve Specific Problems*

Alfonso Ortega de la Puente[1], Marina de la Cruz Echeandía[1], Emilio del Rosal[1], Rafael Nuñez Hervás[3], Antonio Jiménez Martínez[1], Carlos Castañeda[1], José Miguel Rojas Siles[2], Diana Pérez[1], Robert Mercaş[4], Alexander Perekrestenko[4], and Manuel Alfonseca[1]

[1] Departamento de Ingeniería Informática, Escuela Politécnica Superior
Universidad Autónoma de Madrid
Madrid, Spain
E-mail: {alfonso.ortega, marina.cruz, emilio.delrosal, antonio.jimenez, carlos.castanneda, manuel.alfonseca}@uam.es

[2] Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
Universidad Politécnica de Madrid
Madrid, Spain
E-mail: josemiguel.rojas@upm.es

[3] Escuela Politécnica Superior
Universidad San Pablo CEU
Madrid, Spain
E-mail: rnhervas@ceu.es

[4] GRLMC-Research Group on Mathematical Linguistics
Universitat Rovira i Virgili
Tarragona, Spain
E-mail: {robertgeorge.mercas, alexander.perekrestenco}@estudiants.urv.cat

* Work partially supported by the Spanish Ministry of Science and Innovation under coordinated research project TIN2011-28260-C03-00 and research projects TIN2011-28260-C03-01, TIN2011-28260-C03-02 and TIN2011-28260-C03-03.

1 Solving NP-problems with Lineally Bounded Resources

In the following pages we will use NEPS to solve several small instances of well known NP problems. We will show computational implementations of NEPs.

In previous sections we have shown some results that prove the computational power of NEPs and the possibility of lineally bounding the temporal performance of their algorithms to solve NP problems. Their excellent performance depends on the following facts: NEPs are inherently parallel, and it is assumed that each of their processors has the spatial resources needed to store the results of applying all the possible rules to its contents. All these results are assumed to be generated at the same time.

NEPs have not been implemented in real hardware. So, in practice, NEPs have to be simulated on the architecture of one of the available computers. In practice it is very difficult to achieve lineal termporal performance because all these platforms need to explicitly handle all the possible results of all the processors. If all of them are simultaneously stored to be processed in parallel, the likely exponential temporal complexity turns into exponential spatial complexity. Nevertheless, it seems possible that the overall performance can be improved if we choose the proper platform. It is a trade-off between spatial and temporal needs.

Elsewere in this volume, we have described different approaches to the simulation of NEPs in different platforms (including their efficient access to clusters of computers).

In this chapter we will not take into account the final platform but only will show how NP-problems can be computationally solved with NEPs and be simulated with jNEP. It is clear that we can improve the likely exponential temporal performance if we choose the proper final platform to run our programs.

1.1 Solving the SAT problem with jNEP

Reference [12] 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 [12].

We have previously explained in this same volume that ANSP stands for Accepting Networks of Splicing Processors. In short, an ANSP is a NEP in

which the transformation rules of its nodes are *splicing rules*. The transformation performed by those rules is very similar to the genetic crossover. To be more precise, a *splicing rule* σ is a quadruple of words written as $\sigma = [(x, y); (u, v)]$. Given this *splicing rule* σ and two words (w,z), the action of σ on (w,z) is defined as follows:

$$\sigma(w, z) = \{t \mid w = \alpha x y \beta, z = \gamma u v \delta \text{ for any words } \alpha, \beta, \gamma, \delta \text{ and } t = \alpha x v \delta \text{ or } t = \gamma u y \beta\}$$

We can use jNEP to actually build and run the ANSP that solves the boolean satisfiability problem (SAT). We will see how the features of NEPs and the *splicing rules* can be used to tackle this problem. The following is a broad summary of the configuration file for such an 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>
    <!-- INPUT NODE -->
    <NODE initCond="{_(_A_)_AND_(_B_OR_C_)_}"
          auxiliaryWords="{_[A=1]_# {_[A=0]_# {_[B=1]_# {_[B=0]_# {_[C=1]_# {_[C=0]_#">
      <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>
    <!-- OUTPUT NODE -->
    <NODE initCond="">
      <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>
<!-- COMP NODE -->
<NODE initCond="" auxiliaryWords="#_[A=0]_} #_[A=1]_} #_} #_1_)_}">
  <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>
<!-- A=1 NODE -->
<NODE initCond=""  auxiliaryWords="#_1_)_} #_)_}">
  <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>
<!-- A=0 NODE -->
<NODE initCond=""  auxiliaryWords="#_1_)_} #_)_}">
  <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 configuration 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 can 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.

```
*************** NEP INITIAL CONFIGURATION ***************
--- Evolutionary Processor 0 ---
{_(_A_)_AND_(_B_OR_C_)_}
```

Our ANSP works as follows. Firstly, the first node creates all the possible combinations for the values of the 3 variables. We show below the jNEP output for the first step:

```
*************** NEP CONFIGURATION - EVOLUTIONARY STEP -**
****************** TOTAL STEPS: 1 **********************
--- Evolutionary Processor 0 ---
{_# {_[A=1]_(_A_)_AND_(_B_OR_C_)_} {_[A=0]_(_A_)_AND_(_B_OR_C_)_}
```

As shown, the splicing rules of the initial node have appended the two possible values of A to two copies of the logical formula. The rules concerned are:

```
<RULE ruleType="splicing" wordX="{" wordY="(" wordU="{_[A=1]" wordV="#"/>
<RULE ruleType="splicing" wordX="{" wordY="(" wordU="{_[A=0]" wordV="#"/>
```

This kind of rules (Manea's splicing rules) uses some auxiliary words that are never removed from the nodes. In our ANSP we use the following auxiliary words:

```
auxiliaryWords="{_[A=1]_# {_[A=0]_# {_[B=1]_# {_[B=0]_# {_[C=1]_# {_[C=0]_#"
```

The end of this first stage arises after $2n-1$ steps, where n is the number of variables:

```
--- Evolutionary Processor 0 ---
{_#
{_[C=0]_[B=0]_[A=0]_(_A_)_AND_(_B_OR_C_)_} {_[C=0]_[B=0]_[A=1]_(_A_)_AND_(_B_OR_C_)_}
```

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

We should point out that NEPs take advantage of the fact that all the rules can be applied to one word in the same step. This is because the model states that each word has an arbitrary number of copies in its processor. Therefore, the above task (which is $\Theta(2^n)$) can be completed in n steps, since each step double the number of words by including in each word a new variable with the value 1 or 0.

After this first stage, the words can leave the initial node and travel to the other nodes. In the net, there is one node per variable and value; in other words, there is one node for $A = 1$, another for $C = 0$ and so on. Each of these nodes reduces, from right to left, the word representing the formula according to the variable values. For example, the sixth node is responsible for $C = 1$ and, thus, makes the following modification to the word `{_[C=1]_[B=1]_[A=1]_(_A_)_AND_(_B_OR_C_)_}`:

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

{_[C=1]_[B=1]_[A=1]_(_A_)_AND_(_B_OR_1_)_}
```

However, the ninth node is responsible for $C = 0$ and, therefore, produces the following change:

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

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

In this way, the nodes share the results of their modifications until one of them produces a word in which the formula is empty and only contains the left side with the variable values. This kind of words is allowed to pass through the input filter of the output node and will therefore enter it At this point the NEP halts, since the stopping condition of the NEP states that a non-empty output node is the signal to stop the computation.

For further details, plase refer to [12] and the implementation in jnep.edelrosal.net.

1.2 Solving an instance of the Hamiltonian path problem with jNEP

Hamiltonian path problem

This well-known NP-complete problem searches an undirected graph for a Hamiltonian path; that is, one that visits each vertex exactly once.

In [1], Adleman proposed to solve this problem with polynomial resources by means of DNA manipulations in the laboratory. Figure 1 shows the graph he used. In this case, the solution is obvious (path 0-1-2-3-4-5-6) Despite its simplicity, Adleman described a general algorithm applicable to almost any graph with the same performance.

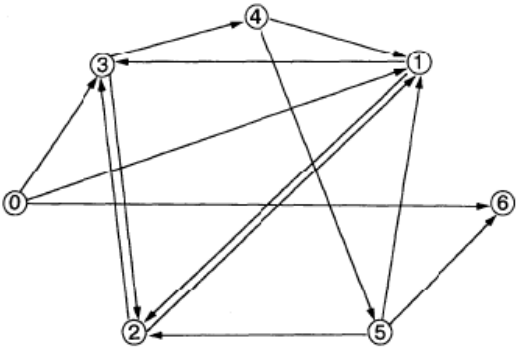

Fig. 1. Graph studied by Adleman

Adleman's algorithm can be summarized as follows:

1. Randomly generate all the possible paths.
2. Select those paths that begin and end in the proper nodes.
3. Select only the paths that contain exactly the total number of nodes.
4. Remove those paths that contain some node more than once.
5. The remaining paths are solutions to the problem.

The present study follows a similar approach (we have already introduced it in this same volume). Remember that the NEP graph is very similar to the one studied above: an extra node is added to ease the definition of the stopping condition. The set i,0,1,2,3,4,5,6 is used as the alphabet. Symbol i is the initial content of the initial node (v0) Each node (except the final one)

adds its number to the string received from the network. Input and output filters are defined to allow the communication of all the possible words without any special constraint. The input filter of the final node excludes any string which is not a solution. It is easy to imagine a regular expression for the set of solutions (those words with the proper length, the proper initial and final node and where each node appears only once). The NEP basic model defines filters by means of regular expressions. It is also easy to devise a set of additional nodes that performs the previous filter following Adleman's checks (proper beginning and end, proper length, and number of occurrences of each node). For the sake of simplicity we have explicitly used the solution word (i_0_1_2_3_4_5_6) instead of a more complex regular expression or a greater NEP.

The reader will find at http://jnep.e-delrosal.net the complete XML file for this problem (Adleman.xml).

The XML file for this example defines the alphabet with this tag

```
<ALPHABET symbols="i_0_1_2_3_4_5_6" />
```

and the initial content of node 0 as

```
<NODE initCond="i">
```

The rules for adding the number of the node to its string are defined as follows (here for node 2)

```
<RULE ruleType = "insertion" actionType = "RIGHT" symbol = "2"/>
```

There are several ways of defining filters for the desired behavior (to allow the communication of all the possible words without any special constraint). We have used only the permitted input and output filters. A string can enter a node if it contains any of the symbols of the alphabet and no string is forbidden.

```
<FILTERS>
  <INPUT type="2"
    permittingContext="i_0_1_2_3_4_5_6"
    forbiddingContext="" />
  <OUTPUT type="2"
    permittingContext="i_0_1_2_3_4_5_6"
    forbiddingContext="" />
</FILTERS>
```

The behavior of the NEP is sketched as follows:

1. In the initial step the only non empty node is 0 and contains the string i
2. After the first step, 0 is added to this string and, node 0 therefore contains i_0

3. This string is moved to the nodes connected to node 0. In the next steps only nodes 1, 3 and 6 contain i_0.
4. These nodes add their number to the received string. In the next step their contents are, respectively, i_0_1, i_0_3 and i_0_6
5. This process is repeated as many times as necessary to produce a string that meets the conditions of the solution. In this final step the solution string i_0_1_2_3_4_5_6 is sent to node 7 and the NEP stops.

Defining filters in the NEP model poses some difficulties to the design of NEPs and, thus, to the development of a simulator. These filters are defined [7] [6] by means of two paris of filters (forbidden and allowed) to each operation (input and output). There are also several ways of combining and applying the filters to translate them into a set of strings. This mechanism contains obvious redundancies that make it difficult to design NEPs. It could be advisable a more general agreement of the researchers to ease and simplify the development of NEPs simulators.

1.3 Solving a graph coloring problem with jNEP

This problem describes a map whose regions have to be colored with only three colors. Adjacent regions must be colored in different colors. We have used the NEP defined in [6]. The map is translated into an undirected graph whose nodes stand for the regions and whose edges represent the adjacency relationship between regions. Figure 2 shows one of the examples we have studied. It is straightforward to prove that there is no solution to this map.

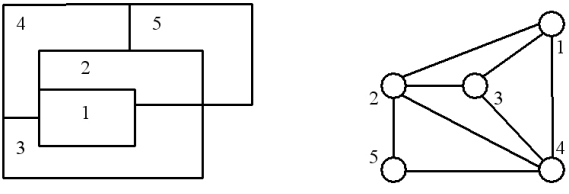

Fig. 2. Example of a map and its adjacency graph. In this case, there is no solution for the 3-colorability problem

The NEP has a complete graph with two special nodes (for the initial and final steps) and a set of seven nodes associated to each edge of the adjacency graph. These nodes perform the tasks outlined below.

The initial (final) node is responsible for starting (stopping) the computation. The seven nodes associated with an edge of the map are grouped in three pairs (one for each color). There is, in addition, a special node to communicate with the set of nodes of the next edge. Each pair is responsible for the main operation in the NEP: to check that a coloring constraint is not violated for the current edge. It performs this task in the following way:

Let us suppose that the color red is the one associated with the pair of nodes. The first node in the NEP associates the color red to the first node of the edge in the map. The second node in the NEP simultaneously keeps all the allowed coloring (two, in this case) for the second node of the edge: (blue and green) It is clear that the only acceptable colorings for this edge are red-blue and red-green.

The behavior of the complete NEP could be described as follows:

1. The initial node generates all the possible assignments of colors to all the regions in the map and adds a symbol to identify the first edge to be checked. These strings are communicated to all the nodes of the graph.
2. The set of nodes associated to each edge accepts only the strings marked with the symbol of the edge. These nodes remove all the strings that violate the coloring constraint for the regions of the edge. One special node in the set replaces the edge mark with that which corresponds to the next edge. In this way, the process continues.
3. The final node of the NEP collects the strings that satisfy the constraints of all the edges. It is straightforward to see that these strings are the solutions.

Some fragments of the XML file for this example (3Coloring.xml) are shown below to describe the above behavior in greater detail:

The alphabet of the NEP is defined as follows:

```
<ALPHABET
symbols="b1_r1_g1_b2_r2_g2_b3_r3_g3_b4_r4_g4_b5_r5_g5_B1_R1_G1_B2_R2_G2_B3_R3_G3
_B4_R4_G4_B5_R5_G5_a1_a2_a3_a4_a5_X1_X2_X3_X4_X5_X6_X8_X9"/>
```

This alphabet contains the following subsets of symbols: a1,...,a5 represents the initial situation of the regions (uncolored). b1, r1, g1,..., b5, r5, g5 represents the assignment of the colors to the regions. B1, R1, G1,..., B5,

R5, G5 is a copy of the previous set to be used while checking the constraint associated with a pair of adjacent regions.

The string contained in the initial node at the beginning represents the complete uncolored map and the number of the first edge to be tackled (X1)

```
<NODE initCond="a1_a2_a3_a4_a5_X1">
```

The rules of the initial node assign all the possible colors to all the regions. The following rules refer to the second region:

```
<RULE ruleType = "substitution"
      actionType = "ANY"
      symbol="a2" newSymbol="b2"/>
<RULE ruleType="substitution"
      actionType="ANY"
      symbol="a2" newSymbol="r2"/>
<RULE ruleType="substitution"
      actionType="ANY"
      symbol="a2" newSymbol="g2"/>
```

The node in the NEP that assigns a color (Red, in this case) to the first region (1 in the example) of an edge in the map uses the following rule:

```
<RULE ruleType="substitution"
      actionType="ANY" symbol="r1"
  newSymbol="R1"/>
```

The other node ensures that the adjacent region (2 in this case) has a different color by means of these rules:

```
<RULE ruleType="substitution"
      actionType="ANY"
      symbol="b2"
      newSymbol="B2"/>
<RULE ruleType="substitution"
      actionType="ANY" symbol="g2"
      newSymbol="G2"/>
```

The node used for starting the process in the next edge removes any special (capitalized) color symbol and sets the edge marking to the next one. The following rules correspond to the first edge

```
<RULE ruleType="substitution"
      actionType="ANY" symbol="R1"
      newSymbol="r1"/>
<RULE ruleType="substitution"
      actionType="ANY" symbol="B1"
      newSymbol="b1"/>
<RULE ruleType="substitution"
      actionType="ANY" symbol="G1"
      newSymbol="g1"/>
<RULE ruleType="substitution"
```

```
      actionType="ANY" symbol="R2"
      newSymbol="r2"/>
<RULE ruleType="substitution"
      actionType="ANY" symbol="B2"
      newSymbol="b2"/>
<RULE ruleType="substitution"
      actionType="ANY" symbol="G2"
      newSymbol="g2"/>
<RULE ruleType="substitution"
      actionType="ANY" symbol="X1"
      newSymbol="X2"/>
```

We found it difficult to apply the input and output filters as they are in [6]. In our opinion, greater standardization is advisable to minimize these situations. Notice that nodes associated with the last edge (in this case with number 8) mark their strings with the following number, which does not correspond to any edge in the graph (9 in our example). This is important for the design of the final node that checks the stopping condition (Non Empty Node Stopping Condition). This final node only accepts strings with the corresponding mark (one that does not correspond to any edge in the adjacency graph).

Figure 3 shows another map to be colored with 3 colors. It is generated by splitting region 3 and 4 in figure 2. Figure 3 also summarizes the sequence of steps for one of the possible solutions. It is worth noticing that all the solutions are simultaneously kept in the configurations of the NEP.

The behavior of the NEP for this map could be summarized as follows: the initial content of the initial node is a1_a2_a3_a4_a5_X1. This node produces all the possible coloring combinations. In the second step of the computation, for example, it contains the following strings:

b1_a2_a3_a4_a5_X1 r1_a2_a3_a4_a5_X1
g1_a2_a3_a4_a5_X1 a1_b2_a3_a4_a5_X1
a1_r2_a3_a4_a5_X1 a1_g2_a3_a4_a5_X1
a1_a2_b3_a4_a5_X1 a1_a2_r3_a4_a5_X1
a1_a2_g3_a4_a5_X1 a1_a2_a3_b4_a5_X1
a1_a2_a3_r4_a5_X1 a1_a2_a3_g4_a5_X1
a1_a2_a3_a4_b5_X1 a1_a2_a3_a4_r5_X1
a1_a2_a3_a4_g5_X1

The NEP still needs a few more steps to get all the combinations. Then, the coloring constraints are applied simultaneously to all the possible solutions and those assignments that violate some constraint are removed. We describe below a sequence of strings generated by the NEP that corresponds to the

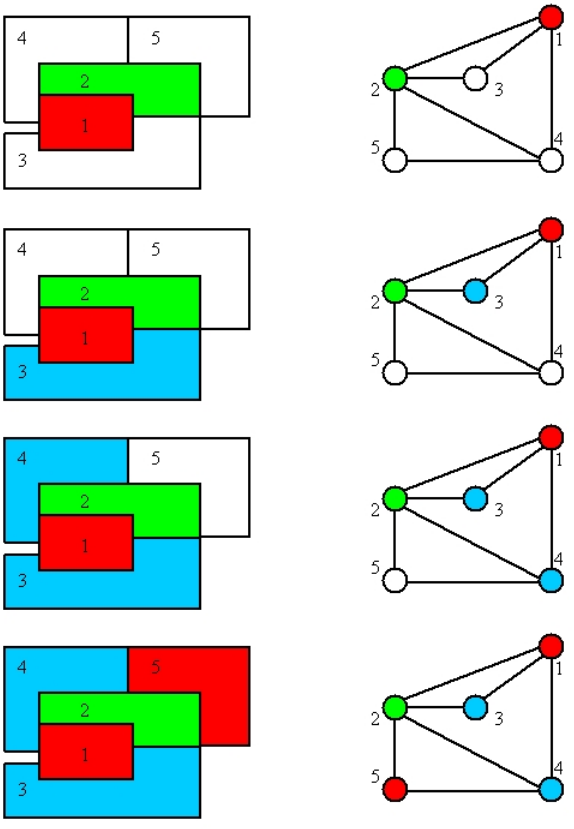

Fig. 3. Sequence of steps in the solution of a 3-coloring problem by jNEP

solution graphically shown in figure 2: r1_g2_b3_b4_r5_X1 is generated in the initial steps. After checking the 1st edge (regions 1 and 2) the NEP contains two strings: R1_g2_b3_b4_r5_X1 and R1_G2_b3_b4_r5_X1

After checking the 2nd edge (regions 1 and 3) R1_g2_B3_b4_r5_X2. And after checking edges 3, 4, 5, 6 and 8 (remember that edge 7 was removed to make the map colorable) associated, respectively, with the pairs of regions 1-4, 2-3, 2-4, 2-5 and 4-5, the following strings are in the NEP:

R1_g2_b3_B4_r5_X3 r1_G2_B3_b4_r5_X4

r1_G2_b3_B4_r5_X5 r1_G2_b3_b4_R5_X6
r1_g2_b3_B4_R5_X8.

Finally, the complete solution is found to be r1_g2_b3_B4_R5_X9 and r1_g2_b3_b4_r5_X9

This NEP processes all the solutions at the same time. It removes all the coloring combinations that violate any constraint. In the last step the final node contains all the solutions found. [6] describes one of the kinds of NEPs (simple NEPs) that is simulated by jNEPs. As we have briefly mentioned before, we have observed that the authors have used slightly different filters for the 3-coloring problem. We could not use these filters and we had to change some of them (most of the output filters) for the NEP to behave properly. The complete XML file is available at http://jnep.e-delrosal.net.

2 Some Applications of NEPs to Language Processing

2.1 PNEPs: top-down parsing for natural languages

Motivation

Syntactic analysis is one of the classical problems related to language processing, and applies both to *artificial* languages (formal languages such as, for instance, programming languages) and to *natural* ones (those that people use to write and talk).

There is an ample range of parsing tools that computer scientists and linguists can use. They share a common goal (parsing), but have obvious differences: some are based on the theoretical foundations of Computer Science (automata, Chomsky grammars) while others mix several formal and informal techniques [14]: for example, generalized deterministic parsing, linear-time substring parsing, parallel parsing, parsing as intersection, non-canonical methods or non-Chomsky systems [15].

The characteristics of the particular language determine the suitability of the parsing technique. Two of the main differences between natural and formal languages are ambiguity and the size of the required representation. Ambiguity creates many difficulties for parsing, so programming languages are usually designed to be non ambiguous. On the other hand, ambiguity is an almost implicit characteristic of natural languages, so it should be taken into account by parsing techniques. To compare the size of different representations, the same formalism should be used. Context-free grammars are

widely used to describe the syntax of languages. It is possible to informally compare the sizes of context free grammars for some programming languages (such as C) and for some natural languages (such as Spanish). We conjecture that the representations required to parse natural languages are frequently greater than those required for high level imperative programming languages.

Parsing techniques for programming languages usually restrict the representation (grammar) used in different ways: it must be unambiguous, recursion is restricted, lambda rules must be removed, they must be (re)written according to some specific normal form, etc. These conditions mean that the designer of the grammar has more work to do, and that non-experts in the field of formal languages will have greater difficulty in properly understanding the grammar. This may be one of the reasons why formal representations such as grammars are little used or even unpopular. Natural languages usually do not fulfill these constraints.

These paragraphs focus on formal representations (based on Chomsky grammars) that can be used for syntactic analysis, and specially those which do not comply with these kinds of constraints. In this way, our approach will be applicable to both natural and formal languages.

Formal parsing techniques for natural languages are inefficient. The sentences that these techniques can usually parse are short (usually less than a typical computer program).

This chapter also focus on new models to increase the efficiency of parsing for languages with non-restricted context free grammars: we propose the use of NEPs as efficient parsing tools. In other sections of the current volume we show how we can efficiently access parallel hardware, such as clusters of computers, in order to simulate NEPs. Our goal is to provide the scientific community with efficient parsing tools that can be run on parallel platforms when they are available.

In the paragraphs below we will introduce the peculiarities of the syntactic analysis of natural languages, and PNEPs, an extension to NEPs that makes them suitable for efficient parsing of any kind of context free grammars, particularly those applicable to languages that share characteristics with natural languages (inherent ambiguity, for example). We have designed a top-down parser for context free grammars without additional constraints. Bellow we informally describe the algorithm, formally define it, detail a jNEP implementation and discuss some examples.

Introduction to analysis of natural languages with NEPs

Computational Linguistics researches linguistic phenomena that occur in digital data. Natural Language Processing (NLP) is a subfield of Computational Linguistics that focuses on building automatic systems that can interpret or generate information written in natural language [26]. This is a broad area which poses a number of challenges, both for theory and for applications.

Machine Translation was the first NLP application in the fifties [27]. In general, the main problem found in all cases is the inherent ambiguity of the language [22].

A typical NLP system has to cover several linguistic levels:

- **Phonological**: Sound processing to detect expression units in speech.
- **Morphological**: Extracting information about words, such as their part of speech and morphological characteristics [21, 2]. The best systems have an accuracy of 97% in this level [4].
- **Syntactical**: Using parsers to detect valid structures in the sentences, usually in terms of a certain grammar. One of the most efficient algorithms is the one described by Earley and its derivatives [11, 24, 28]. It provides parsing in polynomial time, with respect to the length of the input (linear in the average case; n^2 and n^3, respectively, for unambiguous and ambiguous grammars in the worst case) These sections focus on this step. Syntactical analysis for natural language requires considerable of computational resources. Parsers can usually only completely analyze short sentences. Shallow parsing tries to overcome this difficulty. Instead of a complete derivation tree for the sentence, this parsing technique actually builds partial derivation trees for its elemental components.
- **Semantic**: Finding the most suitable knowledge formalism to represent the meaning of the text.
- **Pragmatic**: Interpreting the meaning of the sentence in a context which makes it possibe to react accordingly.

The last two levels are still far from being solved [13].

Figure 4 shows the way in which typical NLP systems usually cover the linguistic levels described above.

A computational model that can be applied to NLP tasks is a network of evolutionary processors (NEPs). NEP as a generating device was first introduced in [10] and [9]. The topic is further investigated in [7], while further different variants of the generating machine are introduced and analyzed in [5, 17, 18, 19, 20].

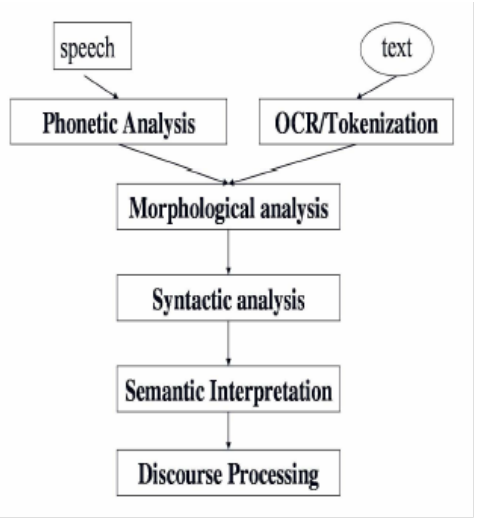

Fig. 4. Typical phases in natural language processing

The first attempt was made to apply NEPs for syntactic NLP parsing in [3]. We have the same goal: to test the suitability of NEPs for this task. We have previously mentioned some performance characteristics of one of the most popular families of NLP parsers (those based on Earley's algorithm). We will conclude that the complexity of our approach is similar.

While [3] outlines a bottom up approach to natural language parsing with NEPs, we suggest a top-down strategy and show its possible use in a practical application.

Top down parsing with NEPs and jNEP

Informal description Other authors have studied the relationships between NEPs, regular, context-free, and recursively enumerable languages [14-18]. [21] shows how NEPs simulate the application of context free rules ($A \rightarrow \alpha, A \in V, \alpha \in V^*$ for alphabet V): a set of additional nodes is needed to implement a rather complex technique to rotate the string and locate A in one of the string ends, then delete it and add all the symbols in α. PNEPs use context free rules rather than classic substitution ($A \rightarrow B, A, B \in V$), as well as insertion and deletion NEP rules. In this way, the expressive power of NEP processors is bounded, while providing a more natural and comfortable way to describe the parsed language for practical purposes.

PNEPs implement a top down parser for context free grammars. Like any other parsers, PNEPs have to build the derivation tree of the string being parsed. We have added a simple mechanism to solve this task. We have used indexes to identify the rules added to the sentential form when each rule is applied. The generated string includes these indexes, making it possible to reconstruct the derivation tree. Several examples are shown below. A PNEP for a given grammar can explicitly generate all the possible derivations of each string in the language generated by the grammar. Its temporal complexity is bounded by the length of the analyzed string (actually by the depth of the derivation tree, that is, by the logarithm of the length). This bound can be used to stop the computation when processing incorrect strings, thus avoiding the possibility that the PNEP runs for an infinite number of steps. Nevertheless, this naive approach seems to be spatially inefficient, because of the high number of strings and derivations simultaneously considered which the processors have to store. We have added two additional mechanisms to overcome this inefficiency:

Discarding non promising sentential forms

The first check we have implemented is the lightest, and it is present in almost all the parsers: discard any sentential form that contains a terminal symbol that the analyzed string does not contain. Parsers usually check sequentially for this condition, starting at the right end of the string. NEPs filters make it possible to check the condition regardless of the positions in the sentential form where the incorrect symbols are. We have implemented this feature by means of an additional node which contains just one deletion rule that deletes nothing (no symbol). In this way we can prevent the whole string from being lost when the evolutionary step is executed in the node. The pruning actually happens during the communication step, because it is implemented by the input filters. A string can pass the filter if it contains only non terminals, or terminals that belong to the input string being parsed. PNEPs duplicate the number of steps needed to parse a string, but reduce the number of strings stored by the processors.

Forcing a left-most derivation order

Applying all the possible rules in parallel to a sentential form produces a lot of different derivations that are actually the same derivation tree. They only

differ in the order in which the non terminal symbols of the same sentential form are derived. We extend the NEP model with a new specialized kind of context free evolutive rule that applies only to the left-most non terminal. The symbol $\rightarrow_l$ will be used to represent this kind of rule. The result of applying the rule $r : A \rightarrow_l s$, where $s \in V^*$ (V stands for the NEP's alphabet) on a given string w, can be formally defined as follows:

$$r(w) =$$
$$t \text{ where } w = w_1 A w_2,\ t = w_1 s w_2,\ not\_contains(w_1, A),\ s \in V^*,\ w_1 \in V^*, \text{ and } w_2 \in V^*$$

For example, the rule $r : A \rightarrow_l s$ will change the following words as shown below:

$Aw_1 \Rightarrow sw_1$ (changes the left-most ocurrence of non-terminal A, which is the left-most non-terminal)
$BAw_1 \Rightarrow Bsw_1$ (changes the left-most ocurrence of non-terminal A, although non-terminal B is on its left)
$cdAw_1 \Rightarrow cdsw_1$ (now there are terminals to the left of A)

From context free grammars to PNEPs

The PNEP is built from the grammar in the following way:

1. We assume that each derivation rule in the grammar has a unique index that can be used to reconstruct the derivation tree.
2. There is a node for each non terminal (*deriving* nodes) that applies to its strings all the derivation rules for its left-most non terminal.
3. There is an additional node (*discarding* node) which discards non promising sentential forms. It receives all the sentential forms generated and sends to the net those that just contain non terminal symbols or terminals which are also contained in the input string.
4. The *deriving* nodes are connected only to the *discarding* node.
5. There is an output node, in which the *parsed string* can be found: this is a version of the input, enriched with information that will make it possible to reconstruct the derivation tree (the rules indexes).
6. The output node is connected with all the *deriving* nodes.

Obviously the same task can be performed using a trivial PNEP with only one *deriving* node for all the derivation rules. However, the proposed PNEP is easier to analyze and makes it easier to distribute the work among several nodes.

We will use the following grammar as an example for some of the steps outlined above. Let us consider grammar $G_{a^n b^n o^m}$ induced by the following derivation rules (notice that indexes have been added in front of the corresponding right hand side):

$$X \rightarrow (1)SO \ , \ S \rightarrow (2)aSb|(3)ab \ , \ O \rightarrow (4)Oo|(5)oO|(6)o$$

It is easy to prove that the language corresponding to this grammar is $\{a^n b^n o^m \mid n, m > 0\}$. Furthermore, the grammar is ambiguous, since every sequence of o symbols can be generated in at least two different ways: by producing the new terminal o with rule 4 or rule 5.

The input filters of the output node describe parsed copies of the initial string. In other words, strings whose symbols are preceded by strings of any length (including 0) of the possible rules indexes. As an example, a parsed version of the string *aabboo* would be 12*a*3*abb*5*o*6*o*.

Formal description We will now describe the way in which our PNEP is defined, starting from a certain grammar. Given the context free grammar $G = \{\Sigma_T = \{t_1, ..., t_n\}, \Sigma_N = \{N_1, ..., N_m\}, A, P\}$ with $A \in \Sigma_N$ its axiom and $P = \{N_i \rightarrow \gamma_j \mid j \in \{1, .., k\}, i \in \{1, ..., n\} \wedge \gamma_j \in (\Sigma_T \cup \Sigma_N)^*\}$ its set of k production rules; the PNEP is defined as

$$\Gamma_G = (V = \Sigma_T \cup \Sigma_N \cup \{1, ..., k\}, node_{output}, N_1, N_2, ..., N_m, N_1^t, N_2^t, ..., N_m^t, G)$$

where

1. N_i is the family of *deriving* nodes. Each node contains the following set of rules: $\{N_i \rightarrow_l \gamma_j\}$ ($\{N_i \rightarrow \gamma_j\}$ are the derivation rules for N_i in G)
2. N^t is the *discarding* node. As has been mentioned above it only contains the deletion rule $\rightarrow$
3. $node_{output}$ is the output node
4. G is a graph that contains an edge for
   - Each pair (N_i, $node_{output}$)
   - Each pair (N_i, N_i^t)
5. The *input node* A is the only one with a non empty initial content (A)
6. The filters for each node are designed to produce the behavior informally described above. In general, the *deriving* nodes have empty output filters

For example, the PNEP for grammar $G_{a^n b^n o^m}$ described above has a node for nonterminal S with the following substitution rules: $\{S \rightarrow 2aSb, S \rightarrow 3ab\}$. The input filter for this node allows all strings containing some copy of their non terminal S to be input in the node.

The input filter for the output node $node_{output}$ has to describe what we have called *parsed strings*. Parsed strings will contain numbers, corresponding to the derivation rules which have been applied, among the symbols of the initial string. For $PI_{node_{output}}$, we can easily create membership conditions. For example, in order to parse the string *aabbo* with the grammar given above, the regular expression can be $\{1,2,3,4,5,6\}^*a\{1,2,3,4,5,6\}^*a\{1,2,3,4,5,6\}^*b\{1,2,3,4,5,6\}^*b\{1,2,3,4,5,6\}^*o$. Our PNEP will stop computing whenever a string enters the output node.

For the discarding node, PI_{N^t} is a random context filter of type 2, where $P = \{a, b, o, X, S, O\}$ and $F = \emptyset$. The derivation nodes have a random context PI_{N_i} of type 1, where $P = \{N_i\}$ and $F = \emptyset$. Finally, any other filters are designed to accept any word without additional constraints.

The complete PNEP for our example ($\Gamma_{a^n b^n o^m}$) is defined as follows:

- Alphabet $V = \{X, O, S, a, b, o, 1, 2, 3, 4, 5, 6\}$
- Nodes
  - $node_{output}$:
    - $A_{output} = \emptyset$ is the initial content;
    - $M_{output} = \emptyset$ is the set of rules;
    - $PI_{output} = \{$ (regular expression membership filter);
    - $\{\{1,2,3,4,5,6\} * a\{1,2,3,4,5,6\} * a\{1,2,3,4,5,6\} * b\{1,2,3,4,5,6\} * b\{1,2,3,4,5,6\} * o\}\}$;
    - $PO_{output} = \emptyset$ is the output filter
  - N_X:
    - $A_X = \{X\}$;
    - $M_X = \{X \rightarrow_l 1SO\}$;
    - $PI_X = \{P = \{X\}, F = \emptyset\}$;
    - $PO_X = \emptyset$
  - N_S:
    - $A_S = \emptyset$;
    - $M_S = \{S \rightarrow_l 2aSb, S \rightarrow_l 3ab\}$;
    - $PI_S = \{P = \{S\}, F = \emptyset\}$;
    - $PO_S = \emptyset$
  - N_O:
    - $A_O = \emptyset$;
    - $M_O = \{O \rightarrow_l 4oO, O \rightarrow_l 5Oo, O \rightarrow_l 5o\}$;
    - $PI_O = \{P = \{O\}, F = \emptyset\}$;
    - $PO_O = \emptyset$
  - N^t:
    - $A = \{\}$;
    - $M = \{\rightarrow\}$;

  - $PI = \{P = \{X, O, S, a, b, o\}, F = \emptyset\}$;
  - $PO = \{F = \emptyset, P = \emptyset\}$
- Its graph contains an edge for each pair $\{(N_X, N^t)$, (N_S, N^t), (N_O, N^t), $(N_X, node_{output})$, $(N_S, node_{output})$, $(N_O, node_{output})\}$
- It stops the computation when some string enters $node_{output}$

The following shows some of the strings generated by all the nodes of the PNEP in succesive communication steps, when parsing the string *aboo* (each set corresponds to a different step):

$\{X\} \Rightarrow \{1SO\} \Rightarrow \{..., 13abO, ...\} \Rightarrow \{..., 13ab4Oo, 13ab5oO, ...\} \Rightarrow \{..., 13ab46oo, ..., 13ab5o6o, ...\}$

The last set contains two different derivations for *aboo* by $(G_{a^n b^n o^m})$, which can enter the output node and stop the computation of the PNEP.

It is easy to reconstruct the derivation tree from the parsed strings in the output node, by following their sequence of numbers. For example, consider the parsed string $13ab6o$ and its sequence of indexes 136; *abo* is generated in the following steps: $X \Rightarrow$ (rule 1 $X \Rightarrow SO$) SO , $SO \Rightarrow$ (rule 3 $S \Rightarrow ab$) abO , $abO \Rightarrow$ (rule 6 $O \Rightarrow o$) abo

jNEP description of PNEPs In a previous section we have described the structure of the XML input files for *jNEP*.

In order to keep jNEP as general as possible, we have added new xml descriptions for each extension needed in PNEP.

Context free rules are represented in the xml file as follows:

```
<RULE ruleType=‘‘contextFreeParsing’’ symbol=‘‘[symbol]’’ newString=‘‘[symbolList]’’/>
```

Those applied to the left-most non terminal, however, use this syntax:

```
<RULE ruleType="leftMostParsing" symbol="NON-TERMINAL" string="SUBSTITUTION_STRING"
nonTerminals="GRAMMAR_NON-TERMINALS"/>
```

Three of the sections of the xml representation of the PNEP $\Gamma_{a^n b^n o^m}$ defined above (the output node, the *derivating* node for axiom X and the *discarding* node) are shown below.

```
<NODE initCond="">
   <EVOLUTIONARY_RULES>
   <RULE ruleType="deletion" actionType="RIGHT" symbol=""/>
   </EVOLUTIONARY_RULES>
   <FILTERS>
     <INPUT type="RegularLangMembershipFilter"
      regularExpression="[1-6]*a[1-6]*b[1-6]*o[1-6]*o"/>
     <OUTPUT type="1" permittingContext="" forbiddingContext="a_b_o_o"/>
   </FILTERS>
 </NODE>
```

```
<NODE initCond="X">
  <EVOLUTIONARY_RULES>
    <RULE ruleType="leftMostParsing" symbol="X" string="1_S_0" nonTerminals="S_0_X"/>
  </EVOLUTIONARY_RULES>
  <FILTERS>
    <INPUT type="1" permittingContext="X" forbiddingContext=""/>
  </FILTERS>
</NODE>


<NODE initCond="">
  <EVOLUTIONARY_RULES>
    <RULE ruleType="deletion" actionType="RIGHT" symbol=""/>
  </EVOLUTIONARY_RULES>
  <FILTERS>
    <INPUT type="2" permittingContext="a_b_o_o_0_1_2_3_4_5_S_0_X"
                    forbiddingContext=""/>
  </FILTERS>
</NODE>
```

The nodes for other non terminal symbols are similar, but with an empty ("") *initial condition* and their corresponding derivation rules.

An example for natural language processing

As described in a previous section, the complexity of the grammars used for syntactic parsing depends on the desired target. These grammars are usually very complex, which makes them one of the bottlenecks in NLP tasks.

We will use the grammar deduced from the following derivation rules, whose axiom is the non terminal Sentence. This grammar is similar to grammars devised by other authors in previous attempts to use NEPs for parsing (natural) languages [19]. We have added the index of the derivation rules that will be used later.

Sentence → (1) NounPhraseStandard PredicateStandard
| (2) NounPhrase3Singular Predicate3Singular
NounPhrase3Singular → (3) DeterminantAn VowelNounSingular
| (4) DeterminantSingular NounSingular
| (5) Pronoun3Singular
NounPhraseStandard → (6) DeterminantPlural NounPlural
| (7) PronounNo3Singular
NounPhrase → (8) NounPhrase3Singular | (9) NounPhraseStandard
PredicateStandard → (10) VerbStandard NounPhrase
Predicate3Singular → (11) Verb3Singular NounPhrase
DeterminantSingular → (12) a | (13) the | (14) this
DeterminantAn → (15) an
VowelNounSingular → (16) apple
NounSingular → (17) boy
Pronoun3Singular → (18) he | (19) she | (20) it
DeterminantPlural → (21) the | (22) several | (23) these
NounPlural → (24) apples | (25) boys
PronounNo3Singular → (26) I | (27) you | (28) we | (29) they
VerbStandard → (30) eat
Verb3Singular → (31) eats

As we have described above, NLP syntax parsing usually takes the results of the morphological analysis as input. In this way, the previous grammar can be simplified by removing the derivation rules for the last 9 non terminals (from DeterminantSingular to Verb3Singular): these symbols become terminals for the new grammar.

Notice, also, that this grammar implements grammatical agreement by means of context free rules. For each non terminal, we had to use several different *specialized versions*. For instance, NounPhraseStandard and NounPhrase3Singular are specialized versions of non terminal NounPhrase. These rules increase the complexity of the grammar.

We can build the PNEP associated with this context-free grammar by following the steps described in the corresponding section.

Let us consider the English sentence *the boy eats an apple*. Some of the strings generated by the nodes of the PNEP in successive communication steps while parsing this string are shown below (we show the initials, rather than the full name of the symbols).

A left derivation of the string is highlighted: { S } ⇒ { ..., 2 NF3S P3S, ... } ⇒ { ..., 2 4 DS NS P3S, ... } ⇒ { ..., 2 4 13 the NS P3S, ... } ⇒ { ..., 2 4 13 the 17 boy P3S, ... } ⇒ { ..., 2 4 13 the 17 boy 11 V3S NF, ... } ⇒ { ...,

2 4 13 the 17 boy 11 31 eats NF, ... } ⇒ { ..., 2 4 13 the 17 boy 11 31 eats 8 NF3S, ... } ⇒ { ..., 2 4 13 the 17 boy 11 31 eats 8 3 DA VNS, ... } ⇒ { ..., 2 4 13 the 17 boy 11 31 eats 8 3 15 an VNS, ... } ⇒ { ..., 2 4 13 the 17 boy 11 31 eats 8 3 15 an 16 apple, ... }

The following fragments of the jNEP output for this case show more detail of the contents of some nodes of the PNEP during its execution.

Notice that

- Node 16 is the *discarding* node, node 17 is the output node and the rest are the *deriving* nodes.
- The indexes of the rules added to the string in order to build the derivation tree include two numbers:
  1. The first one identifies their non terminal
  2. The second identifies the right hand side

  For example, index 1-8 refers to the eighth right hand side of the first non terminal.
- The string [...] means that a piece of output is not shown to save space. Comments are also written between square brackets.

```
***************                NEP INITIAL CONFIGURATION              ***************
--- Evolutionary Processor 0 ---
[...]
--- Evolutionary Processor 9 ---

--- Evolutionary Processor 10 ---
Sentence
--- Evolutionary Processor 11 ---
[...]
--- Evolutionary Processor 16 ---

--- Evolutionary Processor 17 ---


*************** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 1 **************
[...]
--- Evolutionary Processor 10 ---
10-1_NounPhraseStandard_PredicateStandard 10-0_NounPhrase3Singular_Predicate3Singular
[...]
--- Evolutionary Processor 16 ---

--- Evolutionary Processor 17 ---


*************** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS:  2 ************
--- Evolutionary Processor 0 ---
[...]
--- Evolutionary Processor 15 ---
```

```
--- Evolutionary Processor 16 ---
10-1_NounPhraseStandard_PredicateStandard 10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 17 ---


*************** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 3 **************
--- Evolutionary Processor 0 ---
[...]
--- Evolutionary Processor 15 ---

--- Evolutionary Processor 16 ---
10-1_NounPhraseStandard_PredicateStandard 10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 17 ---


*************** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS:  4 *************
--- Evolutionary Processor 0 ---

--- Evolutionary Processor 1 ---

--- Evolutionary Processor 2 ---

--- Evolutionary Processor 3 ---
10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 4 ---

--- Evolutionary Processor 5 ---

--- Evolutionary Processor 6 ---

--- Evolutionary Processor 7 ---
10-1_NounPhraseStandard_PredicateStandard
--- Evolutionary Processor 8 ---

--- Evolutionary Processor 9 ---

--- Evolutionary Processor 10 ---

--- Evolutionary Processor 11 ---

--- Evolutionary Processor 12 ---

--- Evolutionary Processor 13 ---
10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 14 ---

--- Evolutionary Processor 15 ---
10-1_NounPhraseStandard_PredicateStandard
--- Evolutionary Processor 16 ---

--- Evolutionary Processor 17 ---


*************** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 5 **************
[...]
*************** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS:  6 *************
[...]
```

```
*************** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 7 **************
[...]
*************** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS:  8 ************
[...]
*************** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 9 **************
--- Evolutionary Processor 0 ---

--- Evolutionary Processor 1 ---

--- Evolutionary Processor 2 ---
10-0_3-0_2-2_he_Predicate3Singular 10-0_3-0_2-1_she_Predicate3Singular
10-0_3-0_2-0_it_Predicate3Singular
--- Evolutionary Processor 3 ---
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
10-0_3-0_Pronoun3Singular_Predicate3Singular
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
--- Evolutionary Processor 4 ---
10-1_7-1_4-1_several_NounPlural_PredicateStandard
10-1_7-1_4-2_the_NounPlural_PredicateStandard
10-1_7-1_4-0_these_NounPlural_PredicateStandard
--- Evolutionary Processor 5 ---
10-0_3-2_5-0_an_VowelNounSingular_Predicate3Singular
--- Evolutionary Processor 6 ---

--- Evolutionary Processor 7 ---

--- Evolutionary Processor 8 ---
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
--- Evolutionary Processor 9 ---
10-1_7-0_9-2_you_PredicateStandard 10-1_7-0_9-0_they_PredicateStandard
10-1_7-0_9-3_I_PredicateStandard
10-1_7-0_9-1_we_PredicateStandard
--- Evolutionary Processor 10 ---

--- Evolutionary Processor 11 ---
10-0_3-1_11-2_a_NounSingular_Predicate3Singular
10-0_3-1_11-0_this_NounSingular_Predicate3Singular
10-0_3-1_11-1_the_NounSingular_Predicate3Singular
--- Evolutionary Processor 12 ---
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
--- Evolutionary Processor 13 ---
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
10-0_3-0_Pronoun3Singular_Predicate3Singular
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 14 ---
10-1_7-1_DeterminantPlural_NounPlural_PredicateStandard
--- Evolutionary Processor 15 ---
10-1_NounPhraseStandard_PredicateStandard
10-1_7-1_DeterminantPlural_NounPlural_PredicateStandard
10-1_7-0_PronounNo3Singular_PredicateStandard
--- Evolutionary Processor 16 ---

--- Evolutionary Processor 17 ---


*************** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS:  10 ************
```

```
--- Evolutionary Processor 0 ---
[...]
[AT THIS POINT, PARSING TREES WITH INCORRECT TERMINALS HAVE BEEN PRUNED]
--- Evolutionary Processor 16 ---
10-0_3-2_5-0_an_VowelNounSingular_Predicate3Singular
10-1_7-1_DeterminantPlural_NounPlural_PredicateStandard
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
10-0_3-0_Pronoun3Singular_Predicate3Singular
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
10-1_7-1_4-2_the_NounPlural_PredicateStandard
10-0_3-1_11-1_the_NounSingular_Predicate3Singular
10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 17 ---


*************** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 11 *************
--- Evolutionary Processor 0 ---
[...]
--- Evolutionary Processor 16 ---
10-0_3-2_5-0_an_VowelNounSingular_Predicate3Singular
10-1_7-1_DeterminantPlural_NounPlural_PredicateStandard
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
10-0_3-0_Pronoun3Singular_Predicate3Singular
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
10-1_7-1_4-2_the_NounPlural_PredicateStandard
10-0_3-1_11-1_the_NounSingular_Predicate3Singular
10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 17 ---


*************** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS:  12 ***********
--- Evolutionary Processor 0 ---

--- Evolutionary Processor 1 ---

--- Evolutionary Processor 2 ---
10-0_3-0_Pronoun3Singular_Predicate3Singular
--- Evolutionary Processor 3 ---
10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 4 ---
10-1_7-1_DeterminantPlural_NounPlural_PredicateStandard
--- Evolutionary Processor 5 ---
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
--- Evolutionary Processor 6 ---

--- Evolutionary Processor 7 ---

--- Evolutionary Processor 8 ---
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
10-0_3-1_11-1_the_NounSingular_Predicate3Singular
--- Evolutionary Processor 9 ---

--- Evolutionary Processor 10 ---

--- Evolutionary Processor 11 ---
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
--- Evolutionary Processor 12 ---
```

```
10-0_3-2_5-0_an_VowelNounSingular_Predicate3Singular
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
--- Evolutionary Processor 13 ---
10-0_3-2_5-0_an_VowelNounSingular_Predicate3Singular
10-0_3-1_DeterminantSingular_NounSingular_Predicate3Singular
10-0_3-0_Pronoun3Singular_Predicate3Singular
10-0_3-2_DeterminantAn_VowelNounSingular_Predicate3Singular
10-0_3-1_11-1_the_NounSingular_Predicate3Singular
10-0_NounPhrase3Singular_Predicate3Singular
--- Evolutionary Processor 14 ---
10-1_7-1_DeterminantPlural_NounPlural_PredicateStandard
10-1_7-1_4-2_the_NounPlural_PredicateStandard
--- Evolutionary Processor 15 ---
10-1_NounPhraseStandard_PredicateStandard
10-1_7-1_DeterminantPlural_NounPlural_PredicateStandard
10-1_7-0_PronounNo3Singular_PredicateStandard
10-1_7-1_4-2_the_NounPlural_PredicateStandard
--- Evolutionary Processor 16 ---

--- Evolutionary Processor 17 ---


*************** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 13 *************
[...]
*************** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS:  14 ***********
[...]
*************** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 15 *************
[...]
[...]
[...]
*************** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS:  38 ***********
--- Evolutionary Processor 0 ---
[...]
--- Evolutionary Processor 16 ---
10-0_3-2_5-0_an_12-0_apple_13-0_6-0_eats_0-0_7-0_PronounNo3Singular
10-0_3-2_5-0_an_12-0_apple_13-0_6-0_eats_0-1_3-2_5-0_an_VowelNounSingular
10-0_3-2_5-0_an_12-0_apple_13-0_6-0_eats_0-1_3-1_11-1_the_8-0_boy
[...]
10-0_3-1_11-1_the_8-0_boy_13-0_6-0_eats_0-0_NounPhraseStandard
10-0_3-1_11-1_the_8-0_boy_13-0_6-0_eats_0-1_3-2_DeterminantAn_VowelNounSingular
10-0_3-1_11-1_the_8-0_boy_13-0_6-0_eats_0-0_7-1_DeterminantPlural_NounPlural
[...]
--- Evolutionary Processor 17 ---
10-0_3-1_11-1_the_8-0_boy_13-0_6-0_eats_0-1_3-2_5-0_an_12-0_apple


----------------------- NEP has stopped!!! -----------------------

Stopping condition found: net.e_delrosal.jnep.stopping.NonEmptyNodeStoppingCondition

------------------------------------------------------------------
```

If we analyze an incorrect sentence, such as *the boy eat the apple*, the PNEP will continue the computation after the steps summarized above, because in this case it is impossible to find a parsed string. To modify our PNEP to stop

when this happens, it is enough to take into account that the length of the input string is a bound for the number of steps needed (it is always possible to get equivalent context free grammars without chaining and lambda rules; in addition, the length of a given string is usually less than the depth of its derivation trees).

2.2 PNEPs for shallow parsing

Motivation

The goal of the following paragraphs is to modify and use PNEPs for shallow parsing. Shallow parsing will be described later. It is a parsing technique frequently used in natural language processing to overcome the inefficiency of other approaches to syntactic analysis.

Some of the authors of this contribution were involved in developing IBERIA, a corpus of scientific Spanish which is able to process the sentences at the morphological level.

We are very interested in adding syntactic analysis tools to IBERIA. The current contribution has this goal.

Below we will introduce shallow parsing and FreeLing, a well-known free platform that offers parsing tools such as a Spanish grammar and shallow parsers for this grammar.

Then we will show how PNEPs can be used for shallow parsing and describe a jNEP implementation. Finally some examples will be given.

Introduction to FreeLing and shallow parsing

Let us summarize some of the main difficulties encountered by parsing techniques when building complete parsing trees for natural languages:

- Spatial and temporal performance of the analysis. The Early algorithm and its derivatives [11, 24, 28] are some of the most efficient approaches. They, for example, provide parsing in polynomial time, with respect to the length of the input. Its time complexity for parsing context-free languages is linear in the average case, while in the worst case it is n^2 and n^3, respectively, for unambiguous and ambiguous grammars.
- The size and complexity of the corresponding grammar, which is also difficult to design. Natural languages, for instance, are usually ambiguous.

The goal of shallow parsing is to analyze the main components of the sentences (for example, noun groups, verb groups, etc.) rather than complete sentences. It ignores the actual syntactic structure of the sentences, which are considered to be merely sets of these basic blocks. Shallow parsing tries to overcome, in this way, the performance difficulties that arise when building complete derivation trees.

Shallow parsing produces sequences of subtrees. These subtrees are frequently shown as children of a fictitious root node. This way of presenting the results of the analysis can confuse the inexperienced reader, because the final tree is not a real derivation tree: neither is its root the axiom of the grammar nor its branches correspond to actual derivation rules.

Shallow parsing includes different particular algorithms and tools (for instance FreeLing [25] or cascades of finite-state automata [16])

FreeLing is *An Open Source Suite of Language Analyzers* that provides the scientist with several tools and techniques. FreeLing includes a Spanish context-free grammar, adapted for shallow parsing, that does not contain a real axiom. This grammar has almost two hundred non-terminals and approximately one thousand rules. The actual number of rules is even greater, because they use regular expressions rather than terminal symbols. Each rule, then, represents a set of rules, depending on the terminal symbols that match the regular expressions.

The terminals of the grammar are *part-of-speech* tags produced by the morphological analysis. So they include labels like "plural adjective", "third person noun" etc.

Figure 8 shows the output of FreeLing for a very simple sentence like "Él es ingeniero"[5].

FreeLing built three subtrees: two noun phrases and a verb. After that, FreeLing just joins them under the fictitious axiom. Figure 5 shows a more complex example.

PNEP extension for shallow parsing

The main difficulty involved in adapting PNEPs to shallow parsing is the fictitious axiom. PNEPs are designed to handle context free grammars that must have an axiom.

We have also found additional difficulties in the way in which FreeLing reduces the number of derivation rules required by its grammar. As we have

[5] *He is an engineer*

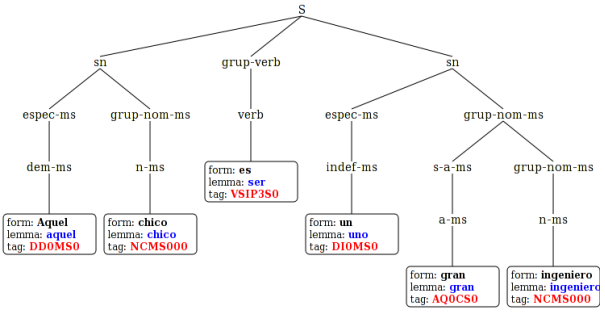

Fig. 5. FreeLing output for "Aquel chico es un gran ingeniero" (*That guy is a great engineer*)

mentioned above, FreeLing uses regular expressions rather than terminal symbols. This kind of rules actually represents a set of rules: those whose terminals match the regular expressions. We have also added this mechanism to PNEPs in the corresponding filters that implement the matching.

In the paragraphs below we will explain both problems in greater detail.

The virtual root node and the partial derivation trees (for the different components of the sentence) force some changes in the behavior of PNEPs. Firstly, we have to derive many trees at once, one for each constituent, instead of only one tree for the complete sentence. Therefore, all the nodes that will apply derivation rules for the nonterminals associated with the components in which the shallow parser is focused will contain their symbol in the initial step. In [23] the nodes of the axiom were the only non empty nodes. More formally:

- Initially, in the original PNEP [23], the only non empty node is associated with the axiom and contains a copy of the axiom. Formally (N_A and Σ_N stand, respectively, for the node associated with the axiom and the set of nonterminal symbols of the grammar under consideration)
  $I_{N_A} = A$
  $\forall N_i \in \Sigma_N, i \neq A \rightarrow I_{N_i} = \emptyset$
- The initial conditions of the PNEP for shallow parsing are:
  $\forall N_i, I_{N_i} = i$

In this way, the PNEP produces every possible derivation sub-tree beginning from each non-terminal, as if they were axioms of a virtually independent grammar. However, those sub-trees have to be concatenated and then joined

to the same parent node (virtual root node of the fictitious axiom). We get this behavior with splicing rules [8], [18] in the following way: (1) the PNEP marks the end and the beginning of the sub-trees with the symbol %, (2) splicing rules are applied to concatenate couples of sub-trees, taking the beginning of the first one and the end of the second as the splicing point.

To be more precise, a special node is responsible for the first step. Its specification in jNEP is the following:

```
<NODE initCond="">
  <EVOLUTIONARY_RULES>
    <RULE ruleType="insertion" actionType="RIGHT" symbol="%"/>
    <RULE ruleType="insertion" actionType="LEFT" symbol="%"/>
  </EVOLUTIONARY_RULES>
  <FILTERS>
    <INPUT type="2" permittingContext="SET_OF_VALID_TERMINALS"
                    forbiddingContext=""/>
    <OUTPUT type="RegularLangMembershipFilter"
            regularExpression="%%.*|%.*%|.*%%"/>
  </FILTERS>
</NODE>
```

During the second step the splicing rules concatenate the sub-trees. We could choose a specialized node (just one node) or a set of nodes depending on the degree of parallelism we prefer. The splicing rule required could be defined as follows:

```
<RULE ruleType="splicingChoudhary" wordX="terminal1" wordY="%"
                                   wordU="%" wordV="terminal2"/>
```

Where terminal2 follows terminal1 in the sentence at any place. It should be remembered that % marks the end and beginning of the derivation trees. If the sentence has n words, there are n-1 rules/points for concatenation. It is important to note that only splicing rules that create a valid sub-sentence are actually concatenated. [6]

For example, if the sentence to be parsed is a_b_c_d, we would need the following rules:

```
<RULE ruleType="splicingChoudhary" wordX="a" wordY="%"
                                   wordU="%" wordV="b"/>
<RULE ruleType="splicingChoudhary" wordX="b" wordY="%"
                                   wordU="%" wordV="c"/>
<RULE ruleType="splicingChoudhary" wordX="c" wordY="%"
                                   wordU="%" wordV="d"/>
```

They could concatenate two sub-sentences like b_c and d, resulting in b_c_d.

[6] In fact, we are using Choudhary splicing rules [8] with a little modification to ignore the symbols that belong to the trace of the derivation.

Our PNEP for the FreeLing's Spanish grammar

The jNEP configuration file for our PNEP adapted to FreeLing's grammar is large. It has almost 200 hundred nodes and some nodes have dozens of rules. We will show, however, some of its details. Let the sentence to be parsed be "Él es ingeniero". The output node has the following definition:

```
<NODE initCond="">
  <EVOLUTIONARY_RULES>
    <RULE ruleType="deletion" actionType="RIGHT" symbol=""/>
  </EVOLUTIONARY_RULES>
    <FILTERS>
      <INPUT type="RegularLangMembershipFilter"
             regularExpression=
                "%[0-9\-]*(PP3MS000|PP\*)[0-9\-]*(VSIP3S0|VSI\*)
                  [0-9\-]*(NCMS000|NCMS\*|NCMS00\*)%"/>
      <OUTPUT type="1" permittingContext=""
              forbiddingContext="PP*_PP3MS000_VSI*_VSIP3S0
                                 _NCMS*_NCMS00*_NCMS000"/>
    </FILTERS>
</NODE>
```

We have explained above that the input sentence includes part-of-speech tags instead of actual Spanish words. This sequence of tags, together with the indexes of the rules that will be used to build the derivation tree, are in the input filter for the output node. We can also see some tags written as regular expressions. We have added this kind of tags because FreeLing also uses regular expressions to reduce the size of the grammar.

As an example, we show the specification of one of the deriving nodes. We can see below that the non-terminal group-verb has many rules. The rule with trace ID 70-7 is the one that is actually needed to parse our example.

```
<NODE initCond="grup-verb" id="70">
 <EVOLUTIONARY_RULES>
  <RULE ruleType="leftMostParsing" symbol="grup-verb"
        string="70-0_grup-ve[...]
  <RULE ruleType="leftMostParsing" symbol="grup-verb"
        string="70-1_grup-ve[...]
  <RULE ruleType="leftMostParsing" symbol="grup-verb"
        string="70-7_verb" [...]
   [...]
 </EVOLUTIONARY_RULES>
 <FILTERS>
   <INPUT type="1" permittingContext="grup-verb"
                   forbiddingContext=""/>
 </FILTERS>
</NODE>
```

The output of jNEP is also considerable. However, we can show at least the main dynamics of the process (see Figures 6 and 7). The comments between brackets provide explanations to facilitate understanding.

```
***************NEP INITIAL CONFIGURATION***************
--- Evolutionary Processor 0 ---
[THE INITIAL WORD OF EVERY DERIVATION NODE IS ITS CORRESPONDING
NON-TERMINAL IN THE GRAMMAR]
[...]
--- Evolutionary Processor 70 ---
grup-verb
[...]
--- Evolutionary Processor 112 ---
sn
[...]
--- Evolutionary Processor 190 ---
[THE OUTPUT NODE IS EMPTY]
*************** NEP CONFIGURATION - EVOLUTIONARY STEP -
TOTAL STEPS: 1 ***************
[FIRST EXPANSION OF THE TREES]
[...]
--- Evolutionary Processor 70 ---
70-6_verb-pass 70-7_verb 70-0_grup-verb_patons_patons_patons[...]
[...]
--- Evolutionary Processor 112 ---
112-104_grup-nom 112-103_grup-nom-ms 112-97_pron-mp 112-95_pron-ns[...]
[...]
*************** NEP CONFIGURATION - COMMUNICATION STEP -
TOTAL STEPS:  2 ***************
--- Evolutionary Processor 0 ---
[THE FIRST TREES WITH ONLY TERMINALS APPEAR AT THE BEGINNING OF
SPLICING SUB-NET]
--- Evolutionary Processor 178 ---
57-3_NCMS00* 151-35_VSI* 1-2_PP3MS000 99-0_NCMS* 121-2_VSI*
[...]
[THE REST GO TO THE PRUNING NODE]
--- Evolutionary Processor 189 ---
112-87_psubj-mp_indef-mp 8-3_s-a-ms 44-6_prep_s-a-fp [...]
```

Fig. 6. jNEP output for "Él es ingeniero". 1 of 2

As jNEP shows, the output node contains more than one derivation tree. We design the PNEP in this way because ambiguous grammars have more than one possible derivation tree for the same sentence. In this case, our PNEP will produce all the possible derivation trees, while FreeLing is only able to show the most likely.

Figure 8 also clearly corresponds to the output of jNEP when our PNEP is run for shallow parsing.

```
*************** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS:  4 ************
[THE PROCESS OF MARKING THE END AND THE BEGINNING STARTS]
[...]
--- Evolutionary Processor 178 ---
1-2_PP3MS000_% %_151-35_VSI* 57-3_NCMS00*_% %_1-2_PP3MS000 %_99-0_NCMS* 99-0_NCMS*_%
151-35_VSI*_% 121-2_VSI*_% %_121-2_VSI* %_57-3_NCMS00*
[...]
*************** NEP CONFIGURATION - EVOLUTIONARY STEP - TOTAL STEPS: 7 **************
[THE SPLICING SUB-NET STARTS TO CONCATENATE THE SUB-TREES]
[...]
--- Evolutionary Processor 178 ---
156-3_1-2_PP3MS000_% 77-13_57-3_NCMS00*_% %_70-7_151-35_VSI* 34-11_99-0_NCMS*_%
%_111-4_1-2_PP3MS000 111-4_1-2_PP3MS000_% 70-7_151-35_VSI*_% %_77-13_57-3_NCMS00*
%_34-11_99-0_NCMS* %_156-3_1-2_PP3MS000
[...]
--- Evolutionary Processor 187 ---
%_121-2_VSI*_99-0_NCMS*_% %_% %_151-35_VSI*_% %_99-0_NCMS*_% %_121-2_VSI*_%
%_151-35_VSI*_99-0_NCMS*_%
--- Evolutionary Processor 188 ---
%_121-2_VSI*_57-3_NCMS00*_% %_151-35_VSI*_57-3_NCMS00*_% %_% %_151-35_VSI*_%
%_121-2_VSI*_% %_57-3_NCMS00*_%
[...]
*************** NEP CONFIGURATION - COMMUNICATION STEP - TOTAL STEPS:  18 ***********
[THE OUTPUT NODE RECEIVES THE RIGHT DERIVATION TREE. IT IS THE SAME AS THE ONE OUTPUT
 BY FREELING]
--- Evolutionary Processor 190 ---
[THE FIRST ONE IS THE OUTPUT DESIRED]
%_112-99_111-4_1-2_PP3MS000_70-7_151-35_VSI*_112-103_77-13_57-3_NCMS00*_%
%_1-2_PP3MS000_151-35_VSI*_57-3_NCMS00*_%
[...]
```

Fig. 7. jNEP output for "Él es ingeniero". 2 of 2

References

1. R. Adleman. Molecular computation of solutions to combinatorial problem. *Science*, 266:1021–1024, 1994.
2. E. Alfonseca. *An Approach for Automatic Generation of on-line Information Systems based on the Integration of Natural Language Processing and Adaptive Hypermedia techniques*. PhD thesis, Computer Science Department, UAM, 2003.
3. G. Bel Enguix, M. D. Jiménez-López, R. Mercaş and A. Perekrestenko. Networks of evolutionary processors as natural language parsers. In *Proceedings ICAART 2009*, 2009.
4. T. Brants. Tnt–a statistical part-of-speech tagger. In *Proceedings of the 6th Conference on Applied Natural Language Processing*, 2000.
5. J. Castellanos, P. Leupold, and V. Mitrana. On the size complexity of hybrid networks of evolutionary processors. *Theoretical Computer Science*, 330(2):205–220, 2005.
6. J. Castellanos, C. Martín-Vide, V. Mitrana, and J. M. Sempere. Networks of evolutionary processors. *Acta Informatica*, 39(6-7):517–529, 2003.

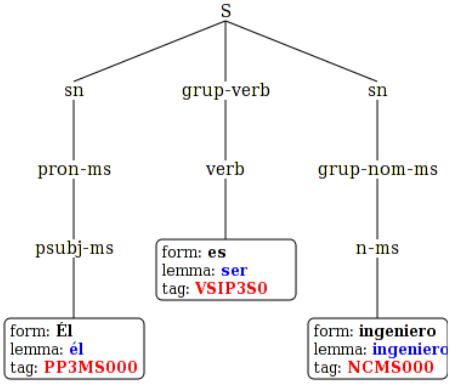

Fig. 8. Shallow parsing tree for "Él es ingeniero"

7. J. Castellanos, C. Martín-Vide, V. Mitrana, and J. M. Sempere. Solving NP-complete problems with networks of evolutionary processors. In *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, 2001, Proceedings, Part I*, 2001.
8. A. Choudhary and K. Krithivasan. Network of evolutionary processors with splicing rules. *Mechanisms, Symbols and Models Underlying Cognition, PT 1, Proceedings*, 3561:290–299, 2005.
9. E. Csuhaj-Varjú and V. Mitrana. Evolutionary systems: a language generating device inspired by evolving communities of cells. *Acta Informatica*, 36(11):913–926, May 2000.
10. E. Csuhaj-Varjú and A. Salomaa. *Lecture Notes on Computer Science 1218*, chapter Networks of parallel language processors. 1997.
11. J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.
12. C. Martín-Vide, F. Manea and V. Mitrana. Accepting networks of splicing processors: Complexity results. *Theoretical Computer Science*, 371(1-2):72–82, 2007.
13. C. Gómez, F. Javier, D. Valle Agudo, J. Rivero Espinosa, and D. Cuadra Fernández. *Procesamiento del lenguaje Natural*, chapter Methodological approach for pragmatic annotation, pages 209–216. 2008.
14. D. Grune, H.E. Bal, C. Jacobs and Langendoen, K.G. *Modern Compiler Design*. John Wiley and sons, 2002.
15. D. Grune and C. Jacobs. *Parsing Techniques: a Practical Guide*. Springer New York, 2008.
16. Z. S. Harris. *String Analysis of Sentence Structure*. Mouton, The Hague, 1962.

17. F. Manea. Using AHNEPS in the recognition of context-free languages. In *In Proceedings of the Workshop on Symbolic Networks ECAI*, 2004.
18. F. Manea and V. Mitrana. All NP-problems can be solved in polynomial time by accepting hybrid networks of evolutionary processors of constant size. *Information Processing Letters*, 103(3):112–118, July 2007.
19. M. Margenstern, V. Mitrana, and M. J. Pérez-Jiménez. Accepting hybrid networks of evolutionary processors. *DNA Computing*, 3384:235–246, 2005.
20. C. Martín-Vide, V. Mitrana, M. J. Pérez-Jiménez and F. Sancho-Caparrini. Hybrid networks of evolutionary processors. *Genetic and Evolutionary Computation. GECCO 2003, PT I, Proceedings*, 2723:401–412, 2003.
21. A. Mikheev. Periods, capitalized words, etc. *Computational Linguistics*, 28(3):289–318, 2002.
22. R. Mitkov. *The Oxford Handbook of Computational Linguistics*. Oxford University Press, 2003.
23. A. Ortega, E. del Rosal, D. Pérez, R. Mercaş, A. Perekrestenko and M. Alfonseca. *PNEPs, NEPs for Context Free Parsing: Application to Natural Language Processing*, chapter Bio-Inspired Systems: Computational and Ambient Intelligence, pages 472–479. LNCS. 2009.
24. S. Seifert and I. Fischer. *Parsing String Generating Hypergraph Grammars*. Springer, 2004.
25. TALP. http://www.lsi.upc.edu/ nlp/freeling/, 2009.
26. M. Volk. *Introduction to Natural Language Processing*. Course CMSC 723 / LING 645 in the Stockholm University, Sweden., 2004.
27. W. Weaver. *Translation, Machine Translation of Languages: Fourteen Essays*. 1955.
28. A. Zollmann and A. Venugopal. Syntax augmented machine translation via chart parsing. In *Proceedings of the Workshop on Statistic Machine Translation*. HLT/NAACL, New York, June. 2006.