Solving NP-Complete Problems with Networks of Evolutionary Processors

Juan Castellanos[1], Carlos Martín-Vide[2], Victor Mitrana[3], and Jose M. Sempere[4]

[1] Dept. Inteligencia Artificial - Facultad de Informática, Universidad Politécnica de Madrid - Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain - jcastellanos@fi.upm.es - http://www.dia.fi.upm.es

[2] Research Group in Mathematical Linguistics - Rovira i Virgili University - Pça. Imperial Tàrraco 1, 43005 Tarragona, Spain - cmv@correu.urv.es

[3] Faculty of Mathematics, University of Bucharest[‡] - Str. Academiei 14, 70109 Bucharest, Romania - mitrana@funinf.math.unibuc.ro

[4] Department of Information Systems and Computation - Polytechnical University of Valencia, - Valencia 46071, Spain - jsempere@dsic.upv.es

Abstract. We propose a computational device based on evolutionary rules and communication within a network, similar to that introduced in [4], called network of evolutionary processors. An NP-complete problem is solved by networks of evolutionary processors of linear size in linear time. Some furher directions of research are finally discussed.

1 Introduction

A basic architecture for parallel and distributed symbolic processing, related to the Connection Machine [8] as well as the Logic Flow paradigm [5], consists of several processors, each of them being placed in a node of a virtual complete graph, which are able to handle data associated with the respective node. Each node processor acts on the local data in accordance with some predefined rules, and, then local data becomes a mobile agent which can navigate in the network following a given protocol. Only such data can be communicated which can pass a filtering process. This filtering process may require to satisfy some conditions imposed by the sending processor, by the reveiving processor or by both of them. All the nodes send simultaneously their data and the receiving nodes handle also simultaneously all the arriving messages according to some strategies, see, e.g., [6, 8].

Starting from the premise that data can be given in the form of strings, [4] introduces a concept called network of parallel language processors in the aim of investigating this concept in terms of formal grammars and languages. Networks of language processors are closely related to grammar systems, more specifically to parallel communicating grammar systems [3]. The main idea is that one can

[‡] Research supported by the Dirección General de Enseñanza Superior e Investigación Cientifica, SB 97–00110508

J. Mira and A. Prieto (Eds.): IWANN 2001, LNCS 2084, pp. 621-628, 2001.
© Springer-Verlag Berlin Heidelberg 2001

place a language generating device (grammar, Lindenmayer system, etc.) in any node of an underlying graph which rewrite the strings existing in the node, then the strings are communicated to the other nodes. Strings can be successfully communicated if they pass some output and input filter.

In the present paper, we modify this concept in the following way inspired from cell biology. Each processor placed in a node is a very simple processor, an evolutionary processor. By an evolutionary processor we mean a processor which is able to perform very simple ooperations, namely point mutations in a DNA sequence (insertion, deletion or substitution of a pair of nucleotides). More generally, each node may be viewed as a cell having a genetic information encoded in DNA sequences which may evolve by local evolutionary events, that is point mutations. Each node is specialized just for one of these evolutionary operations. Furthermore, the data is each node is organized in the form of multisets, each copy being processed in parallel such that all the possible evolutions events that can take place do actually take place.

These networks may be used as language (macroset) generating devices or as computational ones. Here, we consider them as computational mechanisms and show how an NP-complete problem can be solved in linear time.

It is worth mentioning here the similarity of this model to that of a P system, a new computing model inspired by the hierarchical and modularized cell structure recently proposed in [11].

2 Preliminaries

We start by summarizing the notions used throughout the paper. An *alphabet* is a finite and nonempty set of symbols. Any sequence of symbols from an alphabet V is called *string (word)* over V. The set of all strings over V is denoted by V^* and the empty string is denoted by ε. The length of a string x is denoted by $|x|$.

A *multiset* over a set X is a mappingg $M : X \longrightarrow \mathbf{N} \cup \{\infty\}$. The number $M(x)$ expresses the number of copies of $x \in X$ in the multiset M. When $M(x) = \infty$, then x appears arbitrarily many times in M. The set $supp(M)$ is the support of M, i.e., $supp(M) = \{x \in X \mid M(x) > 0\}$. For two multisets M_1 and M_2 over X we define their union by $(M_1 \cup M_2)(x) = M_1(x) + M_2(x)$. For other operations on multisets the reader may consult [1].

A *network of evolutionary processors* (NEP for short) of size n is a construct

$$\Gamma = (V, N_1, N_2, \ldots, N_n),$$

where:

- V is an alphabet,
- for each $1 \leq i \leq n$, $N_i = (M_i, A_i, PI_i, FI_i, PO_i, FO_i)$ is the i-th evolutionary node processor of the network. The parameters of every processor are:
  - M_i is a finite set of evolution rules of one of the following forms only
    - $a \to b$, $a, b \in V$ (substitution rules),
    - $a \to \varepsilon$, $a \in V$ (deletion rules),

- $\varepsilon \to a$, $a \in V$ (insertion rules),

More clearly, the set of evolution rules of any processor contains either substitution or deletion or insertion rules.

- A_i is a finite set of strings over V. The set A_i is the set of initial strings in the i-th node. Actually, in what follows, we consider that each string appearing in a node of the net at any step has an arbitrarily large number of copies in that node, so that we shall identify multisets by their supports.
- PI_i and FI_i are subsets of V representing the input filter. This filter, as well as the output filter, is defined by random context conditions, PI_i forms the permitting context condition and FI_i forms the forbidding context condition. A string $w \in V^*$ can pass the input filter of the node processor i, if w contains each element of PI_i but no element of FI_i. Note that any of the random context conditions may be empty, in this case the corresponding context check is omitted. We write $\rho_i(w) = \underline{true}$, if w can pass the input filter of the node processor i and $\rho_i(w) = \underline{false}$, otherwise.
- PO_i and FO_i are subsets of V representing the output filter. Analogously, a string can pass the output filter of a node processor if it satisfies the random context conditions associated with that node. Similarly, we write $\tau_i(w) = \underline{true}$, if w can pass the input filter of the node processor i and $\tau_i(w) = \underline{false}$, otherwise.

By a configuration (state) of an NLP as above we mean an n-tuple $C = (L_1, L_2, \ldots, L_n)$, with $L_i \subseteq V^*$ for all $1 \leq i \leq n$. A configuration represents the sets of strings (remember that each string appears in an arbitrarily large number of copies) which are present in any node at a given moment; clearly the initial configuration of the network is $C_0 = (A_1, A_2, \ldots, A_n)$. A configuration can change either by an evolutionary step or by a communicating step. When changing by a evolutionary step, each component L_i of the configuration is changed in accordance with the evolutionary rules associated with the node i.
Formally, we say that the configuration $C_1 = (L_1, L_2, \ldots, L_n)$ directly changes for the configuration $C_2 = (L'_1, L'_2, \ldots, L'_n)$ by a evolutionary step, written as

$$C_1 \Longrightarrow C_2$$

if L'_i is the set of strings obtained by applying the rules of R_i to the strings in L_i as follows:

- If the same substitiution rule may replace different occurrences of the same symbol within a string, all these occurrences must be replaced within different copies of that string. The result is the multiset in which every string that can be obtained appears in an arbitrarily large number of copies.
- Unlike their common use, deletion and insertion rules are applied only to the end of the string. Thus, a deletion rule $a \to \varepsilon$ can be applied only to a string which ends by a, say wa, leading to the string w, and an insertion rule $\varepsilon \to a$ applied to a string x consists of adding the symbol a to the end of x, obtaining xa.

– If more than one rule, no matter its type, applies to a string, all of them must be used for different copies of that string.

More precisely, since an arbitrarily large number of copies of each string is available in every node, after a evolutionary step in each node one gets an arbitrarily large number of copies of any string which can be obtained by using any rule in the set of evolution rules associated with that node. By definition, if L_i is empty for some $1 \leq i \leq n$, then L_i' is empty as well.

When changing by a communication step, each node processor sends all copies of the strings it has which are able to pass its output filter to all the other node processors and receives all copies of the strings sent by any node processor providing that they can pass its input filter.
Formally, we say that the configuration $C_1 = (L_1, L_2, \ldots, L_n)$ directly changes for the configuration $C_2 = (L_1', L_2', \ldots, L_n')$ by a communication step, written as

$$C_1 \vdash C_2$$

if for every $1 \leq i \leq n$,

$$L_i' = L_i \setminus \{w \in L_i \mid \tau_i(w) = \underline{true}\} \cup \bigcup_{j=1, j\neq i}^{n} \{x \in L_j \mid \tau_j(x) = \underline{true} \text{ and } \rho_i(x) = \underline{true}\}.$$

Let $\Gamma = (V, N_1, N_2, \ldots, N_n)$ be an NEP. By a computation in Γ we mean a sequence of configurations $C_0, C_1, C_2, \ldots$, where C_0 is the initial configuration, $C_{2i} \Longrightarrow C_{2i+1}$ and $C_{2i+1} \vdash C_{2i+2}$ for all $i \geq 0$.

If the sequence is finite, we have a finite computation. The result of any finite computation is collected in a designated node called the output (master) node of the network. If one considers the output node of the network as being the node k, and if $C_0, C_1, \ldots, C_t$ is a computation, then the set of strings existing in the node k at the last step - the k-th component of C_t - is the result of this computation. The time complexity of the above computation is the number of steps, that is t.

3 Solving NP-Complete Problems

In this section we attack one problem known to be NP-complete, namely the Bounded Post Correspondence Problem (BPCP) [2, 7] which is a variant of a much celebrated computer science problem, the Post Correspondence Problem (PCP) known to be unsolvable [9] in the unbounded case, and construct a NEP for solving it. Furthermore, the proposed NEP computes all solutions.

An instance of the PCP consists of an alphabet V and two lists of strings over V

$$u = (u_1, u_2, \ldots, u_n) \quad \text{and} \quad v = (v_1, v_2, \ldots, v_n).$$

The problem asks whether or not a sequence $i_1, i_2, \ldots, i_k$ of positive integers exists, each between 1 and n, such that

$$u_{i_1} u_{i_2} \ldots u_{i_k} = v_{i_1} v_{i_2} \ldots v_{i_k}.$$

The problem is undecidable when no upper bound is given for k and NP-complete when k is bounded by a constant $K \leq n$. A DNA-based solution to the bounded PCP is proposed in [10].

Theorem 1 *The bounded PCP can be solved by an NEP in size and time linearly bounded by the product of K and the length of the longest string of the two Post lists.*

Proof. Let $u = (u_1, u_2, \ldots, u_n)$ and $v = (v_1, v_2, \ldots, v_n)$ be two Post lists over the alphabet $V = \{a_1, a_2, \ldots, a_m\}$ and $K \geq n$. Let

$$s = K \cdot \max\left(\{|u_j| \mid 1 \leq j \leq n\} \cup \{|u_j| \mid 1 \leq j \leq n\}\right).$$

Consider a new alphabet

$$U = \bigcup_{i=1}^{m} \{a_i^{(1)}, a_i^{(2)}, \ldots, a_i^{(s)}\} = \{b_1, b_2, \ldots, b_{sm}\}.$$

For each $x = i_1 i_2 \ldots i_j \in \{1, 2, \ldots, n\}^{\leq K}$ (the set of all sequences of length at most K formed by integers between 1 and n), we define the string

$$u^{(x)} = u_{i_1} u_{i_2} \ldots u_{i_j} = a_{t_1} a_{t_2} \ldots a_{t_{p(x)}}.$$

We now define a one-to-one mapping $\alpha : V^* \longrightarrow U^*$ such that for each sequence x as above $\alpha(u^{(x)})$ does not contain two occurrences of the same symbol from U. We may take

$$\alpha(u^{(x)}) = a_{t_1}^{(1)} a_{t_2}^{(2)} \ldots a_{t_{p(x)}}^{(p(x))}.$$

The same construction applies to the strings in the second Post list v. We define

$$F = \{\alpha(u^{(x)}\alpha(v^{(x)}) \mid x \in \{1, 2, \ldots, n\}^{\leq K}\} = \{z_1, z_2, \ldots, z_l\}$$

and assume that $z_j = b_{j,1} b_{j,2} \ldots b_{j,r_j}$, $1 \leq j \leq l$, where $|z_j| = r_j$. By the construction of F, no letter from U appears within any string in F for more than two times. Furthermore, if each letter of $z = \alpha(u^{(x)})\alpha(v^{(x)})$ appears twice within z, then x is a solution of the given instance.

We are now ready to define the NEP which computes all the solutions of the given instance. It is a NEP of size $2sm + 1$

$$\Gamma = (U \cup \bar{U} \cup \hat{U} \cup \tilde{U} \cup \{X\} \cup \{X_d^{(c)} \mid 1 \leq c \leq n, 2 \leq d \leq |z|_c\}, N_1, N_2, \ldots N_{2sm+1}),$$

where

$$\begin{aligned}
\bar{U} &= \{\bar{b} \mid b \in U\} \\
&\quad \text{(the other sets, namely } \hat{U} \text{ and } \tilde{U}\text{, which form the NEP alphabet} \\
&\quad \text{are defined similarly)} \\
M_f &= \{\varepsilon \to b_f\}, \\
A_f &= \emptyset, \\
FI_f &= \{X_d^{(c)} \mid 2 \le d \le |z|_c, 1 \le c \le l \text{ such that } b_f \ne b_{c,d}\} \cup \bar{U} \cup \hat{U} \cup \tilde{U}, \\
PI_f &= FO_f = PO_f = \emptyset,
\end{aligned}$$

for all $1 \le f \le sm$,

$$\begin{aligned}
M_{sm+1} &= \{X \to X_2^{(c)} \mid 1 \le c \le l\} \cup \{X_{|z|_c} \to b_{c,1} \cup \{b_d \to \bar{b}_d \mid 1 \le d \le sm\} \\
&\quad \cup \{X_d^{(c)} \to X_{d+1}^{(c)} \mid 1 \le c \le l, 2 \le d \le |z|_c - 1\}, \\
A_{sm+1} &= \{X\}, \\
FI_{sm+1} &= PI_{sm+1} = FO_{sm+1} = PO_{sm+1} = \emptyset,
\end{aligned}$$

and

$$\begin{aligned}
M_{sm+d+1} &= \{b_d \to \tilde{b}_d, \bar{b}_d \to \hat{b}_d\}, \\
A_{sm+d+1} &= \emptyset, \\
FI_{sm+d+1} &= (\bar{U} \setminus \{\bar{b}_d\}) \cup \{X_g^{(c)} \mid 2 \le g \le |z|_c, 1 \le c \le l\}, \\
PI_{sm+d+1} &= FO_{sm+d+1} = \emptyset, \\
PO_{sm+d+1} &= \{\tilde{b}_d, \hat{b}_d\},
\end{aligned}$$

for all $1 \le d \le sm$.

Here are some informal considerations about the computing mode of this NEP. It is easy to note that in the first stage of a computation only the processors $1, 2, \ldots, sm+1$ are active. Since the input filter of the others contains all symbols of the form $X_g^{(c)}$, they remain inactive until one strings from F is produced in the node $sm+1$.

First let us explain how an arbitrary string $z_j = b_{j,1}b_{j,2}\ldots b_{j,r_j}$ from F can be obtained in the node $sm+1$. One starts by applying the rules $X \to X_2^{(j)}$, $1 \le j \le l$, in the node $sm+1$. The strings $X_2^{(j)}$, $1 \le j \le l$, obtained in the node $sm+1$ as an effect of a evolutionary step are sent to all the other processors, but for each of these strings there is only one processor which can receive it. For instance the string $X_2^{(c)}$ is accepted only by the node processor f, $1 \le f \le sm$, with $b_f = b_{c,2}$. In the next evolutionary step, the symbol $b_{j,2}$ is added to the right hand end of the string $X_2^{(j)}$ for all $1 \le j \le l$. Now, a communication step is to be done. All the strings $X_2^{(j)}b_{j,2}$ can pass the output filters of the nodes processors where they were obtained but the node processor $sm+1$ is the only one which can receive them. Here the lower subscripts of the symbol X are

increased by one and the process from above is resumed in the aim of adjoining a new letter. This process does not apply to a string $X_r^{(j)} b_{j,2} \dots b_{j,r}$ anymore, if and only if $r = |z_j|$, when $X_r^{(j)}$ is replaced by $b_{j,1}$ resulting in the string z_j. By these considerations, we infer that all the strings from F are produced in the node $sm + 1$ in $2s$ steps.

Another stage of the computation checks the number of occurrences of any letter within any string obtained in the node $sm+1$, as soon the string contains only letters in U. This is done as follows. By the way of applying the substitution rules aforementioned, each occurrence of any letter is replaced by its barred version in the node $sm + 1$. Let us consider a string produced by such an evolutionary step. Such a string has only one occurrence of a symbol $\bar{b}_d$, for some $1 \leq d \leq sm$, the other symbols being from $U \cup \hat{U} \cup \tilde{U}$. It can pass the input filter of the processor $sm + d + 1$ only, where it remains for three steps (two evolutionary steps and one comunication one) or forever. The string can leave the node $sm + d + 1$, only if it has an ooccurrence of the symbol b_d. By replacing this occurrence with $\tilde{b}_d$ and $\bar{b}_d$ with $\hat{b}_d$, the string can pass the output filter of the node processor $sm + d + 1$ and goes to the node $sm + 1$. In this way, one checked whether or not the original string had have two occurrences of the letter b_d. After $6s$ steps the computation stops and the node $sm + 1$ has only strings which were produced from those strings in F having two occurrences of any letter. As we have seen, these strings encode all the solutions of the given instance of BPCP. □

4 Concluding Remarks

We have proposed a computational model whose underlying architecture is a complete graph having evolutionary processors placed in its nodes. Being a bio-inspired system, a natural question arises: How far is this model from the biological reality and engineering possibilities? More precisely, is it possible exchange biological material between nodes? Can the input/output filter conditions of the node processors be biologically implemented? What about a technological implementation? We hope that at least some answers to these questions are affirmative.

We have presented a linear algorithm based on this model which provide all solutions of an NP-complete problem.

Further, one can go to different directions of research. In our variant, the underlying graph is the complete graph. In the theory of networks some other types of graphs are common, e.g., rings, grids, star, etc. It appears of interest to study the networks of evolutionary processors where the underlying graphs have these special forms.

A natural question concerns the computational power of this model. Is it computationally complete? However, our belief is that those variants of the model which are "specialized" in solving a few classes of problems have better chances to get implemented, at least in the near future.

References

1. J. P. Banâtre, A. Coutant, D. Le Metayer, A parallel machine for multiset transformation and its programming style, *Future Generation Computer Systems*, 4 (1988), 133–144.
2. R. Constable, H. Hunt, S. Sahni, On the computational complexity of scheme equivalence, *Technical Report* No. 74-201, Dept. of Computer Science, Cornell University, Ithaca, NY, 1974.
3. E. Csuhaj - Varju, J. Dassow, J. Kelemen, Gh. Paun - *Grammar Systems*, Gordon and Breach, 1993.
4. E. Csuhaj-Varjú, Networks of parallel language processors. In *New Trends in Formal Languages* (Gh. Păun, A. Salomaa, eds.), LNCS 1218, Springer Verlag, 1997, 299–318
5. L. Errico, C. Jesshope, Towards a new architecture for symbolic processing. In *Artificial Intelligence and Information-Control Systems of Robots '94* (I. Plander, ed.), World Sci. Publ., Singapore, 1994, 31–40.
6. S. E. Fahlman, G. E. Hinton, T. J. Seijnowski, Massively parallel architectures for AI: NETL, THISTLE and Boltzmann machines. In *Proc. AAAI National Conf. on AI*, William Kaufman, Los Altos, 1983, 109–113.
7. M. Garey, D. Johnson, *Computers and Intractability. A Guide to the Theory of NP-completeness*, Freeman, San Francisco, CA, 1979.
8. W. D. Hillis, *The Connection Machine*, MIT Press, Cambridge, 1985.
9. J. Hopcroft, J. Ulmann, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.
10. L. Kari, G. Gloor, S. Yu, Using DNA to solve the Bounded Correspondence Problem, *Theoret. Comput. Sci.*, 231 (2000), 193–203.
11. Gh. Păun, Computing with membranes, *J. Comput. Syst. Sci.* 61(2000). (see also *TUCS Research Report* No. 208, November 1998, http://www.tucs.fi.)
12. Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms*, Springer-Verlag, Berlin, 1998.