UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES
INFORMÀTICS I COMPUTACIÓ

TESIS DOCTORAL:

Redes de Procesadores Genéticos

Presentada por: Marcelino Campos Francés

Dirigida por: Dr. José María Sempere Luna
Dr. Damián López Rodríguez

Septiembre, 2015

Agradecimientos

Este apartado va dedicado principalmente a mis padres a los que quiero agradecer todo lo que han hecho por mi. No conozco a nadie que sea más trabajador que mi padre con el objetivo de que a su familia no le falte de nada y un orgullo por el trabajo bien hecho que me ha sabido transmitir. Él no entiende de todas estas cosas complicadas a las que me dedico, de vez en cuando me pregunta como va todo en general y se preocupa de que no me falte de nada y sé que lo que más ilusión le hará es que ponga aquí la frase que tantas veces hemos repetido, va por ti: "MI PADRE COLABORÓ". Y mi madre, que es la que nos ha criado a mi hermana y a mi mientras mi padre trabajaba, a ella le debo gran parte de mi forma de ser, gracias mamá por cuidar siempre de todos. Sé que nos hacemos mayores y salimos del nido pero nunca olvidamos el legado ni los valores que nos han dado nuestros padres, y a ellos va dedicado esta tesis, que es el fin de una etapa que sin ellos no habría sido posible.

Luego hay una pequeña lista de gente a quien también me gustaría nombrar: Cristina, la persona que más quiero y con la que comparto y quiero compartir todo en esta vida, te quiero amor. Chema y Damián, mis tutores, a los que quiero agradecer toda lo que han hecho por mí para que consiguiera sacar esta tesis adelante. Mi hermana, siempre cuidando el uno del otro. Leti, muchos años compartiendo grandes momentos dentro y fuera de la facultad. A la gente con la que como en el poli, sois unos cabrones que siempre os tengo que esperar pero moláis. A los compañeros (y amigos) de carrera, que se les recuerda con mucho cariño. A los amigos de fuera de la universidad que siempre está ahí. Y finalmente a la gente que aún no haya mencionado pero que esté leyendo estas líneas. Gracias a todos.

Resumen

Desde la rama de la biocomputación, la computación con modelos inspirados en la biología, esta investigación presenta un nuevo modelo de computación: las Redes de Procesadores Genéticos (NGP). Este nuevo modelo parte, por un lado, de la familia de modelos de redes de procesadores, más concretamente de las Redes de Procesadores Evolutivos (NEP) y las Redes de Procesadores de Splicing (NSP), y por otra parte se inspira en los Algoritmos Genéticos. Así pues, se puede definir de manera informal el nuevo modelo como una red de procesadores bioinspirados donde las operaciones utilizadas por los procesadores son operaciones de cruce y mutación. Uno de los mayores intereses del estudio de las NGP es la capacidad conjunta de las operaciones de cruce y mutación, las NEP son un modelo completo que utiliza operaciones de evolución, es decir, inserción, substitución y borrado, las NSP son un modelo completo que utiliza operaciones de splicing, pero las NEP dejan de ser un modelo completo al usar sólo operaciones de substitución, al igual que le pasa a las NSP si restringimos el contexto de sus reglas de splicing a vacío. El estudio del nuevo modelo aquí presentado da respuesta a qué es lo que pasa cuando juntamos en un sólo modelo las operaciones de sustitución de las NEP (llamadas reglas de mutación) y las operaciones de splicing con contexto vacío de las NSP (llamadas reglas de cruce).

Cuando se trabaja con redes de procesadores bioinspirados se definen principalmente dos tipos de redes, las redes aceptoras y las redes generadoras. Estos tipos de redes sirven principalmente para trabajar a un nivel teórico o para resolver problemas de decisión. Para trabajar a un nivel más práctico como por ejemplo con problemas de optimización, se propone un nuevo tipo de red, las Redes de Procesadores Genéticos como Algoritmos Genéticos Paralelos, llamadas así por inspirarse en los Algoritmos Genéticos Paralelos.

A nivel teórico, se ha podido demostrar la completitud computacional del modelo, con lo que su potencia de computación se sitúa al mismo nivel que el de las maquinas de Turing. A raíz de este resultado y dada la gran similitud entre las NGP y los Algoritmos Genéticos Paralelos, en este trabajo

se demuestra que éstos también son un modelo de computación completo. Por otra parte se ha podido realizar una caracterización de la jerarquía de Chomsky utilizando las NGP, para ello se simula cada una de las gramáticas que definen las cuatro familias de lenguajes de dicha jerarquía observando el mínimo número de procesadores necesarios para cada simulación, lo cual da una idea apreciable de la diferencia de complejidad entre las diferentes familias.

No falta en este trabajo un estudio de la parte más práctica del modelo con la realización de algunas tareas. Primero se ha diseñado e implementado un simulador que permite la ejecución de Redes de Procesadores Genéticos en cualquiera de sus tres vertientes aquí definidas, como aceptoras, como generadoras o como Algoritmos Genéticos Paralelos, esto permite realizar pruebas con diferentes problemas de optimización. A continuación se ha realizado un estudio para ver si el nuevo modelo era capaz de resolver problemas NP en tiempo polinómico, para ello se ha trabajado con el problema de ver si existe algún ciclo Hamiltoniano en un grafo. Finalmente se ha probado el simulador con dos problemas de optimización en los que se ha detectado un buen comportamiento del mismo, los problemas utilizados han sido el problema de la mochila multidimensional y el problema del viajante de comercio.

Resum

Des de la branca de la biocomputació (la computació amb models inspirats amb la biologia) aquesta investigació presenta un nou model de computació: Les Xarxes de Processadors Genètics (NGP). Aquest nou model ve, d'una banda, de la família de models de xarxes de processadors, més concretament de les Xarxes de Processadors Evolutius (NEP) i de les Xarxes de Processadors de Splicing (NSP) i d'altra banda s'inspira als Algoritmes Genètics. Així doncs, es pot definir d'una manera informal el nou model com una xarxa de processadors bioinspirats on les operacions utilitzades per els processadors són operacions de creuament i mutació. Un dels elements més interessants de l'estudi de les NGP és la capacitat conjunta de les operacions de creuament i mutació, les NEP són un model complet que utilitza operacions evolutives, és a dir, insercions, substitucions i esborrats, les NSP són un model complet que utilitza operacions de splicing, però les NEP deixen de ser un model complet al gastar sols operacions de substitució, al igual que li passa a les NSP si restringim el context de les seues regles de splicing a buit. L'estudi del nou model presentat ací dóna resposta a què és el que passa quan ajuntem a un sol model les operacions de substitució de les NEP (anomenades regles de mutació) i les operacions de splicing amb context buit de les NSP (anomenades regles de creuament).

Quan es treballa amb xarxes de processadors bioinspirats es defineixen principalment dos tipus de xarxes, les xarxes aceptores i les xarxes generadores. Aquests tipus de xarxes s'utilitzen principalment per a treballar a nivell teòric o per a resoldre problemes de decisió. Per treballar a un nivell més pràctic, com per exemple amb problemes d'optimització, es proposa un nou tipus de xarxa, les Xarxes de Processadors Genètics com Algoritmes Genètics Paral·lels, anomenats així per estar inspirats en els Algoritmes Genètics Paral·lels.

A nivell teòric, s'ha pogut demostrar la completitut computacional del model, amb el que la seua potència computacional es situa al mateix nivell que les màquines de Turing. Degut a aquest resultat i donada la gran similitud entre les NGP i els Algoritmes genètics Paral·lels, en aquest tre-

ball es demostra que aquestos també són un model computacional complet. D'altra banda, s'ha pogut realitzar una caracterització de la jerarquia de Chomsky utilitzant les NGP, aquest procés es realitza simulant cada una de les gramàtiques que defineixen les quatre famílies de llenguatges d'aquesta jerarquia observant el mínim nombre de processadors necessaris per a cada simulació, el que ens dóna una idea apreciable de la diferència de complexitat entre les diferents famílies.

No falta a aquest treball un estudi de la part més pràctica del model com la realització d'algunes tasques. Primer s'ha dissenyat i implementat un simulador que permet l'execució de Xarxes de Processadors Genètics a qualsevol de les seues tres varietats ací definides, com aceptores, com a generadores o com a Algoritmes Genètics Paral·lels, amb el que podem realitzar proves amb diferents problemes d'optimització. A continuació s'ha realitzat un estudi per vore si el nou model era capaç de resoldre problemes NP en un temps polinòmic, estudi que hem realitzat utilitzant el problema de saber si existeix algun cicle Hamiltonià en un graf. Finalment s'ha provat el simulador amb dos problemes d'optimització als que s'ha comprovat que té un bon comportament, els problemes utilitzats són el problema de la motxilla multidimensional i el problema del viatjant de comerç.

Summary

In this research, a new model of computing is presented, within the framework of natural computing and computer models inspired by biology: Networks of Genetic Processors (NGP). This new model is based on the one hand, by the family of networks of bio-inspired processors, specifically Networks of Evolutionary Processors (NEP) and Networks of Splicing Processors (NSP), and on the other hand by Genetic Algorithms. We can define the new model as a network of biologically-inspired processors where operations used by processors are crossover and mutation. One of the major interests studying the NGP is the computational power of the operations of crossover and mutation acting together. The NEP is a complete model that uses operations of symbol mutation: insertion, substitution and deletion, the NSP is a complete model that uses splicing operations, but the NEP is no longer a complete model using only substitution operations, as it happens to the NSP if we restrict the context of its splicing rules to the empty context. The study of the new model presented here responds to what happens when we put together in a single model the substitution rules from the NEP (called mutation rules) and the splicing rules with empty context from the NSP (called crossover rules).

When we work with networks of biologically-inspired processors there are two basic types of networks, accepting networks and generating networks. These types of networks are mainly used to work at a theoretical level or to solve decision problems. To work on a more practical level such as solving optimization problems, we propose a new type of network, Networks of Genetic processors as Parallel Genetic Algorithms, inspired by the Parallel Genetic Algorithms.

In this work we prove the computational completeness of our new model by showing that it is equivalent to the Turing machine. We prove the computational completeness of Parallel Genetic Algorithms by using this result and the similarity between the NGP and Parallel Genetic Algorithms. Moreover, we propose a characterization of the Chomsky hierarchy by using the NGP. Here, we simulate every grammar in the language classes of the Chomsky's

hierarchy by using a NGP with an small number of processors required for each simulation. Hence, it gives an appreciable idea of the descriptional complexity of the different families of languages.

Finally, in this work there is an experimental study of the behavior of the model for the resolution of some practical problems. First, we design and implement a simulator that allows the execution of networks of Genetic Processors in any of its three defined types: accepting networks, generating networks or as Parallel Genetic Algorithms. This allows us to test the model with different optimization problems. Then, we make a study to see if the new model could solve NP problems in polynomial time. We use the decision problem of Hamiltonian cycle in a graph. Finally, we test the simulator with two optimization problems showing a good computational behavior. The problems are the Multidimensional Knapsack problem and the Traveling Salesman problem.

Índice general

Capítulo 1

Conceptos Básicos

1.1. Teoría de Lenguajes

1.1.1. Conceptos Básicos

A continuación se va a proponer la definición de algunos conceptos básicos sobre la teoría de lenguajes con el objetivo de establecer la notación necesaria para seguir esta tesis. Esta información se puede encontrar en [31] y [51], donde aparecen todos estos temas de forma más completa y desarrollada.

Un alfabeto es un conjunto finito no vacío de elementos denominados símbolos. Una cadena es una secuencia finita y ordenada de símbolos de un alfabeto. Denominaremos ε a la cadena vacía, que se define como la cadena que no tiene ningún símbolo. Dada una cadena w, su longitud es el número de símbolos que contiene y se denota por $|w|$ ($|\varepsilon| = 0$). Llamamos Σ^* al conjunto infinito de todas las cadenas definidas sobre el alfabeto Σ. Un lenguaje L sobre Σ es un subconjunto de Σ^*. Los elementos de L se llaman palabras. Dada la cadena $x \in \Sigma^*$, definimos $alph(x)$ como el subconjunto mínimo $W \subseteq \Sigma$ tal que $x \in W^*$. Dada la cadena $x \in \Sigma^*$, definimos el conjunto de segmentos de x como $seg(x) = \{\beta \in \Sigma^* : x = \alpha\beta\gamma \wedge \alpha, \gamma \in \Sigma^*\}$. Dado un conjunto de elementos A, definimos el conjunto partes de A como $\mathcal{P}(A)$ y está formado por todos los subconjuntos de A.

Dado el conjunto X, un multiconjunto sobre X es un par (X, f) donde $f : X \to \mathbb{N} \cup \{\infty\}$ es una función que cumple que, para $x \in X$, $f(x)$ indica el número de veces que x está en el multiconjunto. Definimos el soporte del multiconjunto M como el conjunto de todos aquellos elementos tales que $f(x) \neq 0 : \forall x \in X$. El tamaño de cualquier multiconjunto M es el número de elementos que posee y lo denotamos como $|M|$, formalmente se define como $|M| = \sum_{x \in X} f(x)$.

1.1.2. Gramáticas

Una gramática es un modelo generador constituido por un conjunto de reglas de formación que definen las cadenas de caracteres admisibles por un determinado lenguaje formal o lenguaje natural.

Definimos una gramática como una tupla $G = (N, T, P, S)$, donde N es el alfabeto de símbolos auxiliares, T es el alfabeto de símbolos terminales ($N \cap T = \emptyset$), $S \in N$ es el axioma o símbolo inicial y P es el conjunto de reglas de producción. Cada regla de producción está formada por el par (α, β) o $\alpha \to \beta$ donde tenemos el antecedente o parte izquierda $\alpha \in (N \cup T)^* N (N \cup T)^*$ y el consecuente o parte derecha $\beta \in (N \cup T)^*$.

Dadas las palabras $v, w \in (N \cup T)^*$, diremos que v deriva directamente en w ($v \underset{G}{\Rightarrow} w$) si $v = v_1 \alpha v_2$, $w = v_1 \beta v_2$ y $\alpha \to \beta \in P$. Dadas las palabras $v, w \in (N \cup T)^*$, diremos que v deriva en w ($v \overset{*}{\underset{G}{\Rightarrow}} w$) si existe la secuencia α_1, α_2, ..., α_n donde $v = \alpha_1$ y $w = \alpha_n$ y se cumple $\alpha_i \underset{G}{\Rightarrow} \alpha_{i+1}$ $i = 1 \ldots n-1$. Diremos que $\beta \in (T \cup N)^*$ es una forma sentencial si $S \overset{*}{\underset{G}{\Rightarrow}} \beta$. De la misma manera diremos que $x \in T^*$ es una palabra generada por G si $S \overset{*}{\underset{G}{\Rightarrow}} x$. Así pues el lenguaje generado por G se define como: $L(G) = \{x \in T^* : S \overset{*}{\underset{G}{\Rightarrow}} x\}$. Diremos que dos gramáticas son equivalentes si generan el mismo lenguaje.

La jerarquía de Chomsky permite agrupar todas las gramáticas existentes en cuatro tipos diferentes dependiendo de las características de las reglas utilizadas por las mismas; dichas caracterizaciones son muy generales, lo cual dificulta su manipulación; este problema puede resolverse usando las llamadas formas normales. Una forma normal no es más que otra manera de caracterizar las gramáticas pero con un conjunto limitado de tipos de reglas permitidos con lo que es mucho más sencillo trabajar con ellas (simularlas o trabajar con ellas en un ordenador). Dada una forma normal que caracterice uno de los tipos de gramáticas de la jerarquía de Chomsky y dada una gramática G perteneciente a dicho tipo, existe siempre una gramática G' equivalente a G y que cumple las especificaciones dictadas por la forma normal. A continuación se describen cada uno de los cuatro tipos de gramáticas de la jerarquía de Chomsky y la forma normal utilizada (si es necesaria):

- Gramáticas Regulares (o de tipo 3): Las gramáticas regulares lineales por la derecha son aquellas cuyas producciones son de la forma:
  - $A \to aB$, con $A, B \in N$ y $a \in T$
  - $A \to a$, con $A \in N$ y $a \in T \cup \{\varepsilon\}$

Otro tipo de gramáticas regulares son las lineales por la izquierda, donde sus producciones son de la siguiente forma:

- $A \to Ba$, con $A, B \in N$ y $a \in T$
- $A \to a$, con $A \in N$ y $a \in T \cup \{\varepsilon\}$

Estos dos tipos de gramáticas son totalmente equivalentes, a la hora de usar gramáticas regulares en este trabajo se utilizarán lineales por la derecha.

- Gramáticas Incontextuales (o de tipo 2): Las gramáticas incontextuales son aquellas cuyas producciones son de la forma:

  - $A \to \alpha$, con $A \in N$ y $\alpha \in (N \cup T)^*$.

  En esta tesis se va a trabajar con la forma normal de Chomsky, donde las reglas tienen la siguiente forma:

  - $A \to BC$, con $A, B, C \in N$
  - $A \to a$, con $A \in N$ y $a \in T$

  Hay que resaltar que cuando trabajamos con la forma normal de Chomsky se pierde la capacidad de generar la palabra ε, lo cual no plantea ningún problema pues se puede añadir la palabra en la simulación si fuese necesario.

- Gramáticas Sensibles al Contexto (o de tipo 1): Las gramáticas sensibles al contexto son aquellas cuyas producciones son de la forma:

  - $S \to \varepsilon$
  - $\gamma A \delta \to \gamma \alpha \delta$, con $A \in N$, $\gamma, \delta \in (N \cup T)^*$ y $\alpha \in (N \cup T)^+$

  En esta tesis se va a trabajar con la forma normal de Kuroda, donde las reglas tienen la siguiente forma:

  - $A \to a$, con $A \in N$ y $a \in T$
  - $A \to B$, con $A, B \in N$
  - $A \to BC$ con $A, B, C \in N$
  - $AB \to CD$ con $A, B, C, D \in N$

Opcionalmente se puede introducir la palabra ε en el lenguaje generado por la gramática añadiendo la regla $S \to \varepsilon$ y no dejando aparecer el símbolo S en ninguna parte derecha de ninguna regla.

- Gramáticas No Restringidas (o de tipo 0): Las producciones de estas gramáticas no tienen ninguna restricción. Para trabajar con esta familia se va a utilizar la forma normal de Kuroda extendida, donde las reglas son de la siguiente forma:

  - $A \to a$, con $A \in N$ y $a \in T$
  - $A \to B$, con $A, B \in N$
  - $A \to BC$ con $A, B, C \in N$
  - $AB \to AC$ con $A, B, C, D \in N$
  - $AB \to CB$, con $A, B, C \in N$
  - $AB \to B$, con $A, B \in N$

  Al igual que en el anterior tipo de gramática, se puede introducir la palabra ε en el lenguaje generado por la gramática añadiendo la regla $S \to \varepsilon$ y no dejando aparecer el símbolo S en ninguna parte derecha de ninguna regla.

Una familia de lenguajes es un conjunto de lenguajes que comparten alguna propiedad, si la propiedad escogida es ser generado por alguno de los diferentes tipos de gramáticas de la jerarquía de Chomsky se obtienen las siguientes cuatro familias:

- La Familia de los Lenguajes Regulares (REG): son aquellos que pueden ser generados mediante una gramática Regular (o de tipo 3).

- La Familia de los Lenguajes Incontextuales (CF): son aquellos que pueden ser generados mediante una gramática Incontextual (o de tipo 2).

- La Familia de los Lenguajes Sensibles al Contexto (CS): son aquellos que pueden ser generados mediante una gramática Sensible al Contexto (o de tipo 1).

- La Familia de los Lenguajes Recursivamente Enumerables (RE): son aquellos que pueden ser generados mediante una gramática No Restringida (o de tipo 0).

Al observar los cuatro tipos de gramáticas se puede ver que toda gramática Regular es a su vez una gramática Incontextual, cada gramática Incontextual es a su vez una gramática Sensible al Contexto y cada gramática Sensible al Contexto es una gramática No Restringida, por tanto las familias de lenguajes de la jerarquía de Chomsky cumplen lo siguiente:

$$REG \subset CF \subset CS \subset RE$$

1.1.3. Máquina de Turing

En 1937 la máquina de Turing fue presentada por el matemático inglés Alan Mathison Turing en un artículo que puede considerarse como el origen oficial de la informática teórica. Mediante la introducción de la máquina de Turing, en este artículo, se pudo formalizar el concepto de algoritmo y resultó ser la precursora de las computadoras. También, gracias a la máquina de Turing, se pudo demostrar la existencia de problemas irresolubles.

Una máquina de Turing esta compuesta por tres elementos (figura 1.1), una cinta potencialmente infinita por la derecha dividida en celdas en la que en cada celda hay un símbolo, un cabezal que puede moverse por la cinta pudiendo leer y escribir en cada celda y un control finito que nos indica en qué estado está la máquina y qué movimientos puede realizar. El cabezal se va moviendo a derecha e izquierda por la cinta reescribiendo los símbolos de cada celda dependiendo del estado en el que está y del carácter leído, pudiendo cambiar de estado en cada paso. Inicialmente, una cadena de entrada x se carga en la cinta introduciendo cada uno de sus símbolos en una celda empezando por la celda más a la izquierda, en el resto de celdas se considera que aparece el símbolo B señalando que están en blanco. El cabezal empieza en el primer símbolo de x y el control finito almacena el estado inicial.

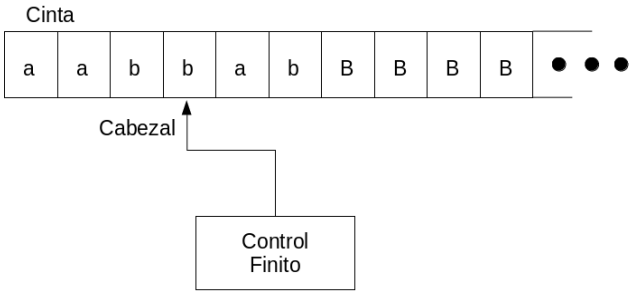

Figura 1.1: Máquina de Turing.

Más formalmente una máquina de Turing determinista es una tupla de la forma $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, donde Q es un conjunto finito de estados, Σ es el alfabeto de entrada, Γ es el alfabeto de cinta (se cumple que $\Sigma \subset \Gamma$), $q_0 \in Q$ es el símbolo inicial, B es el símbolo especial que representa el blanco en la cinta ($B \in (\Gamma - \Sigma)$), $F \subset Q$ es el conjunto de estados finales y $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ es la función de transición. Así pues, la función de transición queda definida de la siguiente manera: $\delta(q, a) = (p, b, z)$ donde $q, p \in Q$, $a, b \in \Gamma$ y donde $z \in \{L, R\}$. El resultado de esta función es el siguiente: si en el cabezal de la cinta se lee el símbolo a, conteniendo el control finito el estado q, la máquina, de forma determinista, substituirá el símbolo a por b, cambiará el estado q por p y realizara el movimiento z (izquierda o derecha).

Para definir el estado de una máquina de Turing utilizaremos una descripción instantánea y la definimos como una cadena de la forma $\alpha_1 q \alpha_2$, donde $\alpha_1, \alpha_2 \in \Gamma^*$ y $q \in Q$. La descripción instantánea nos indica que la cinta de la máquina de Turing contienen la cadena $\alpha_1\alpha_2$ seguida de infinitos blancos, que el cabezal esta apuntando al primer símbolo de la cadena α_2 y que la máquina se encuentra en el estado q. La descripción instantánea inicial es $q_0 x$, que indica que estamos en el estado inicial q_0 y hemos cargado en la cinta la cadena x.

Si partimos de la descripción instantánea $x_1 x_2 ... x_{i-1} q x_i x_{i+1} ... x_n$, podemos aplicar un movimiento a la izquierda o a la derecha como sigue:

- Si se aplica la regla $\delta(q, x_i) = (p, y, L)$ la descripción instantánea que sigue a la anterior es $x_1 x_2 ... p x_{i-1} y x_{i+1} ... x_n$ y lo denotaremos por:

$$x_1 x_2 ... x_{i-1} q x_i x_{i+1} ... x_n \underset{M}{\mapsto} x_1 x_2 ... p x_{i-1} y x_{i+1} ... x_n$$

- Si se aplica la regla $\delta(q, x_i) = (p, y, R)$, la descripción instantánea que sigue a la anterior es $x_1 x_2 ... x_{i-1} y p x_{i+1} ... x_n$:

$$x_1 x_2 ... x_{i-1} q x_i x_{i+1} ... x_n \underset{M}{\mapsto} x_1 x_2 ... x_{i-1} y p x_{i+1} ... x_n$$

Dada la máquina de Turing M, $\underset{M}{\mapsto}$ es una relación entre sus descripciones instantáneas. Definimos la clausura reflexiva y transitiva de $\underset{M}{\mapsto}$ como $\overset{*}{\underset{M}{\mapsto}}$. Entonces $\alpha_1 q \beta_1 \overset{*}{\underset{M}{\mapsto}} \alpha_2 p \beta_2$ indica que la máquina ha hecho un número finito de movimientos (puede que cero) para obtener la descripción instantánea de la derecha a partir de la de la izquierda.

Existen tres situaciones donde la máquina se detiene. La primera se da cuando no tenemos ningún movimiento definido para el estado en que se

encuentra la máquina y el símbolo que está siendo analizado por el cabezal de cinta. La segunda sucede cuando la máquina de Turing intenta realizar un movimiento a la izquierda estando en la primera celda de la cinta. Y por último la máquina siempre para cuando llega a un estado final. Indicamos la situación de parada con el símbolo $\downarrow$, entonces $\alpha_1 q \beta_1 \overset{*}{\underset{M\downarrow}{\mapsto}} \alpha_2 p \beta_2$ indica que la máquina de Turing M parte de la descripción $\alpha_1 q \beta_1$, se aplican un conjunto finito de movimientos para obtener la descripción $\alpha_2 p \beta_2$ y después para.

Diremos que una palabra w es aceptada por una máquina de Turing M si y sólo si a partir de la descripción inicial ($q_0 w$ donde q_0 es el estado inicial) se llega a una descripción con estado final. Así pues el lenguaje aceptado por la máquina de Turing M se define como $L(M) = \{w \in \Sigma^* : q_0 w \overset{*}{\underset{M\downarrow}{\mapsto}} \alpha q \beta \wedge q \in F\}$.

Una Máquina de Turing no determinista se define como una tupla formada por $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, donde todos los elementos se definen igual que en el caso determinista salvo la función δ, que se define como $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$. Así pues, la función de transición queda definida de la siguiente manera: $\delta(q, a) = \{(q_1, a_1, z_1), \ldots, (q_p, a_p, z_p)\}$, donde $q_i \in Q$, $a_i \in \Gamma$ y $z_i \in \{L, R\}$ para $1 \leq i \leq p$. El resultado de esta función es el siguiente: si en el cabezal de la cinta se lee el símbolo a, conteniendo el control finito el estado q, la máquina, de forma indeterminista, seleccionará una tupla (q_j, a_j, z_j) substituyendo a por a_j, cambiando el estado q por q_j y realizando el movimiento z_j (izquierda o derecha). De este modo, la máquina puede tener diferentes resultados dada la misma situación. Los casos de parada son los mismos que en el modelo determinista. La máquina aceptará la cadena de entrada w si existe una selección de movimientos tales que $q_0 w \overset{*}{\underset{M\downarrow}{\mapsto}} \alpha q \beta$ con $q \in F$.

Un árbol de computación es una representación de la ejecución de una máquina de Turing no determinista dada una entrada para dicha máquina. Cada nodo del árbol representa una descripción instantánea y cada arista representa la transición a la siguiente descripción instantánea. Cada hoja del árbol es la finalización de una posible computación (ya sea de aceptación o de rechazo). Si la cadena de entrada de la máquina de Turing es w, la raíz del árbol de computación será la descripción instantánea $q_0 w$.

Dado el modelo de la máquina de Turing vamos a definir dos nuevas familias de lenguajes:

- Diremos que un lenguaje pertenece a la familia de los lenguajes recursivos ($\mathcal{R}$) si existe una máquina de Turing M que lo acepta y que para cualquier entrada M siempre se detiene.

- Diremos que un lenguaje pertenece a la familia de los lenguajes recur-

sivamente enumerables (RE) si existe una máquina de Turing M que lo acepta (aunque puede pasar que M no se detenga al introducirle una palabra que no pertenece al lenguaje).

Como podemos observar, con la definición de estos dos tipos de lenguajes conseguimos relacionar la máquina de Turing con la Jerarquía de Chomsky, pues la familia de los lenguajes recursivamente enumerables (también conocidos como tipo 0 en la Jerarquía de Chomsky) es la más grande y la que engloba a los demás en las ambas clasificaciones.

Vamos a introducir un nuevo concepto, diremos que un modelo de computación es completo si tiene la misma capacidad de computación que una máquina de Turing. Hemos visto que todos los lenguajes que pueden ser aceptados (o generados) por una máquina de Turing forman la familia de lenguajes recursivamente enumerables, con lo que también podemos decir que un modelo de computación es completo si es capaz de aceptar o generar cualquier lenguaje recursivamente enumerable.

1.1.4. Complejidad

Una vez tenemos la máquina de Turing, podemos utilizar sus características para introducir algunos conceptos de complejidad computacional extraídos de [19], para ello, en cada ejecución, tendremos en cuenta la cantidad de movimientos realizados (costes temporales) y las celdas utilizadas (costes espaciales).

Dada M una máquina de Turing determinista, si para cada cadena de entrada w de longitud n, M hace como máximo $T(n)$ movimientos antes de parar decimos que M es una máquina de Turing determinista acotada temporalmente por $T_M(n)$ o con una complejidad temporal $T_M(n)$. El lenguaje aceptado por M diremos que tiene una complejidad temporal determinista $T_M(n)$. En el caso que M sea no determinista diremos que M es una máquina de Turing no determinista acotada temporalmente por $T_M(n)$ si para cada entrada $w \in L(M)$ existe una secuencia mínima de $T(n)$ movimientos para que M acepte w. En este caso diremos que el lenguaje aceptado por M tiene una complejidad temporal no determinista $T_M(n)$. Llegados a este punto podemos definir clases de lenguajes dependiendo de la función de complejidad temporal $T(n)$ como sigue:

- $DTIME(T(n))$ es la clase de lenguajes aceptados por máquinas de Turing deterministas con una complejidad temporal determinista $T(n)$.
- $NTIME(T(n))$ es la clase de lenguajes aceptados por máquinas de Turing no deterministas con una complejidad temporal no determinista

$T(n)$.

Si fijamos un conjunto de funciones enteras, entonces podemos definir diferentes clases de lenguajes. En este caso, dado un conjunto de funciones enteras $\mathcal{C}$, tenemos que:

- $DTIME(\mathcal{C}) = \bigcup_{f \in \mathcal{C}} DTIME(f)$
- $NTIME(\mathcal{C}) = \bigcup_{f \in \mathcal{C}} NTIME(f)$

Sea *poly* el conjunto de todas las funciones polinómicas enteras con coeficientes no negativos, entonces definimos $\mathcal{P} = DTIME(poly)$ y $\mathcal{NP} = NTIME(poly)$.Uno de los problemas abiertos matemáticos más importantes hoy en día es saber si la clase $\mathcal{P}$ y la clase $\mathcal{NP}$ coinciden, es decir, si $\mathcal{P} = \mathcal{NP}$.

De la misma manera en que hemos definido todas estas clases teniendo en cuenta el número de movimientos necesarios para resolver un problema utilizando una máquina de Turing, podemos considerar la cantidad de espacio necesario y estaríamos hablando de la complejidad espacial. Así pues definimos $S(n)$ como el número máximo de celdas de cinta diferentes usadas para la computación de cualquier palabra w de longitud n. Dada M una máquina de Turing determinista, si para cada cadena de entrada w de longitud n, M utiliza como máximo $S(n)$ celdas diferentes antes de parar decimos que M es una máquina de Turing determinista acotada espacialmente por $S_M(n)$ o con una complejidad espacial $S_M(n)$. En el caso que M sea no determinista diremos que M es una máquina de Turing no determinista acotada espacialmente por $S_M(n)$ si para cada entrada $w \in L(M)$ existe un número mínimo $S(n)$ de celdas diferentes usadas necesarias para que M acepte w. En este caso diremos que el lenguaje aceptado por M tiene una complejidad espacial no determinista $S_M(n)$. De la misma forma que con la complejidad temporal podemos definir las siguientes clases:

- $DSPACE(S(n))$ es la clase de lenguajes aceptados por máquinas de Turing deterministas con una complejidad espacial determinista $S(n)$.
- $NSPACE(S(n))$ es la clase de lenguajes aceptados por máquinas de Turing no deterministas con una complejidad espacial no determinista $S(n)$.

Si fijamos un conjunto de funciones enteras, entonces podemos definir diferentes clases de equivalencia de lenguajes. En este caso, dado un conjunto de funciones enteras $\mathcal{C}$, tenemos que:

- $DSPACE(\mathcal{C}) = \bigcup_{f \in \mathcal{C}} DSPACE(f)$

- $NSPACE(\mathcal{C}) = \bigcup_{f \in \mathcal{C}} NSPACE(f)$

Sea *poly* el conjunto de todas las funciones polinómicas enteras con coeficientes no negativos, entonces definimos $\mathcal{PSPACE} = DSPACE(poly)$ y $\mathcal{NPSPACE} = NSPACE(poly)$. A diferencia de las clases $\mathcal{P}$ y $\mathcal{NP}$, sí se ha demostrado que $\mathcal{PSPACE} = \mathcal{NPSPACE}$. Para ver la relación entre las clases que son definidas por su complejidad temporal y por las clases que son definidas por su complejidad espacial tenemos el siguiente resultado: $\mathcal{NP} \subseteq \mathcal{PSPACE}$.

Otra forma de ver el tema de la complejidad es observando el coste de resolución de los problemas. Podemos expresar un problema P en términos de una relación $P \subseteq I \times S$, donde I es el conjunto de instancias del problema, y S el conjunto de soluciones.

Un problema de decisión es un problema (matemáticamente definido) donde la solución es una afirmación o una negación de un predicado sobre los parámetros del problema, más formalmente son aquellos problemas en los que podemos reducir la relación P a la función $f : I \to S$ donde $S = \{YES, NO\}$. Todo problema de decisión D define un lenguaje formal L_D que contiene las instancias del problema que tienen solución afirmativa. De esta manera podemos decir que la complejidad computacional de un problema de decisión es la complejidad computacional de su lenguaje asociado y viceversa. Para la resolución de problemas de decisión podemos utilizar algoritmos deterministas o algoritmos no deterministas:

- Un algoritmo determinista es un algoritmo que, dada una entrada en particular, siempre se comporta de la misma manera. Se trata de un algoritmo que sigue una serie de pasos que dependen de la entrada (para la misma entrada siempre seguirá los mismos pasos) dándonos la solución al terminar.

- Un algoritmo no determinista es una algoritmo que, para la misma entrada, puede tener comportamientos diferentes en diferentes ejecuciones (a diferencia de los algoritmos deterministas). El funcionamiento de una ejecución de un algoritmo de este tipo se realiza de la siguiente manera: una vez recibida la entrada, el algoritmo va generando soluciones al azar, cada vez que genera una de estas soluciones comprueba si es una solución válida para la entrada, este proceso se repite hasta que el algoritmo encuentra una solución correcta, en caso de no encontrar dicha solución el proceso se repite indefinidamente.

La complejidad de un algoritmo no determinista se mide sólo de la parte de comprobación de la solución, considerando que la generación de la solución es instantánea (no tiene coste). Así pues podemos definir la clase de

complejidad $\mathcal{P}$ como el conjunto de problemas de decisión que pueden ser resueltos mediante un algoritmo determinista en tiempo polinómico y la clase $\mathcal{NP}$ como el conjunto de problemas de decisión que pueden ser resueltos mediante un algoritmo no determinista polinómico.

Un problema de optimización es un problema que busca una solución para maximizar o minimizar un criterio determinado. Formalmente podemos caracterizar un problema de optimización P como una cuádrupla (I_P, SOL_P, m_P, $goal_p$), donde I_P es el conjunto de instancias, SOL_P es la función que asocia cada entrada $x \in I_P$ con el conjunto de posibles soluciones, m_P es una función que mide la bondad de los pares (x, y), donde $x \in I_P$ y $y \in SOL_P$ (este valor será positivo si y es una posible solución de x) y $goal_p \in \{MIN, MAX\}$ nos especifica si estamos ante un problema de maximización o minimización. Estos problemas también poseen ciertas restricciones, con lo cual las soluciones correctas serán sólo aquellas que las cumplan todas. Para los problemas de optimización diremos que su complejidad computacional es el coste del cálculo de la solución óptima sujeta a las restricciones del problema.

Sean P_1 y P_2 dos problemas de decisión, diremos que P_1 se puede reducir a P_2 mediante una reducción de Karp ($P_1 \preceq P_2$) si y sólo si existe un algoritmo M tal que el coste temporal de M es polinómico y para toda instancia x del problema P_1, el algoritmo M calcula una instancia y del problema P_2 tal que $x \in P_1$ si y sólo si $y \in P_2$.

Sea $\mathcal{C}$ una clase de complejidad. Decimos que el problema P es completo para $\mathcal{C}$ si P se encuentra en $\mathcal{C}$ y cada problema perteneciente a $\mathcal{C}$ se puede reducir mediante una reducción de Karp a P. Podemos agrupar el conjunto de estos problemas en la clase $\mathcal{C}$-Completo.

Sea $\mathcal{C}$ una clase de complejidad. Decimos que el problema P es duro para $\mathcal{C}$ si cada problema perteneciente a $\mathcal{C}$ se puede reducir mediante una reducción de Karp a P (nótese que no es necesario que P esté en $\mathcal{C}$). Podemos agrupar el conjunto de estos problemas en la clase $\mathcal{C}$-duro.

Estas dos nuevas clases nos permiten observar de una manera más concreta la complejidad de los problemas, podemos decir que dada una clase de complejidad $\mathcal{C}$, los problemas que pertenecen a $\mathcal{C}$-Completo serán los más difíciles de la clase y que cualquier problema de $\mathcal{C}$-Duro será tanto o más difícil que cualquier problema de $\mathcal{C}$. Más concretamente nos interesa la clase $\mathcal{NP}$-Completo, pues son los problemas más complicados de $\mathcal{NP}$ a los que, hoy en día, se les pretende buscar algoritmos óptimos para resolverlos. Si se encontrase un problema que fuera a su vez de la clase $\mathcal{P}$ y $\mathcal{NP}$-Completo obtendríamos una respuesta afirmativa al problema de si $\mathcal{P} = \mathcal{NP}$. Finalmente en la figura 1.2 tenemos un mapa de cómo están situadas las principales clases de complejidad.

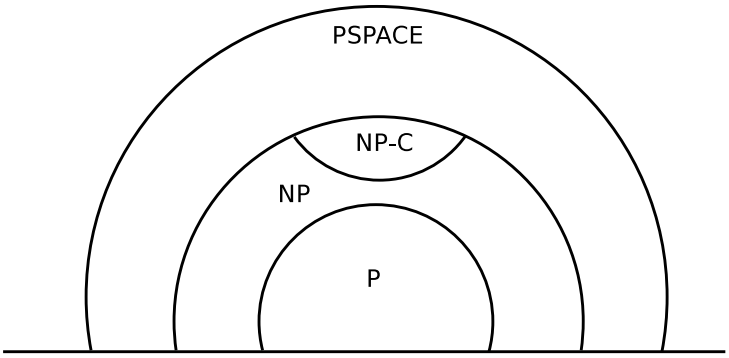

Figura 1.2: Diagrama de las principales clases de complejidad.

Capítulo 2

Computación Natural

2.1. Computación Convencional vs. Computación Natural no Convencional

En este capítulo se van a definir algunos modelos en los que se inspiran las Redes de Procesadores Genéticos, pero antes daremos un breve repaso a algunos aspectos de la computación, tanto a la computación clásica como a la computación natural.

En ciencias de la computación [31] existen dos partes claramente diferenciables, por un lado aparecen las ideas y modelos fundamentales subyacentes en el cálculo y por otro las técnicas de ingeniería para el diseño de sistemas de computación.

Los inicios de la computación "clásica", en lo que a la parte más teórica se refiere, tuvo lugar en cuatro campos distintos: el estudio de los biólogos de las redes neuronales, la investigación de los ingenieros eléctricos para el desarrollo de la teoría de circuitos como herramienta para el diseño hardware, el trabajo realizado por los matemáticos en el campo de la lógica y las investigaciones de los lingüistas en los campos de las gramáticas y lenguajes naturales. Se puede decir que el modelo de computación más importante para esta ciencia es sin duda la máquina de Turing (definida en el capítulo anterior); este modelo es capaz de diferenciar aquellas cuestiones o problemas que pueden resolverse mediante una computadora de los que no (lo que es computable y lo que no). La máquina de Turing también es un modelo muy utilizado para la realización de estudios de complejidad; por ejemplo, fue utilizado para acuñar al SAT como el primer problema que se pudo demostrar que era NP-completo. Otros modelos muy usados son las gramáticas (definidas en el capítulo anterior) y los autómatas [31]; los autómatas son modelos aceptores de palabras formados por estados y transiciones etiquetadas, don-

de, empezando por un estado inicial, cada símbolo de una palabra sirve para transitar a otro estado si el símbolo coincide con la etiqueta de la transición, se dice que la palabra se acepta si al terminar los símbolos de esta llegamos a un estado final. Estos modelos han sido de gran ayuda en problemas como la creación de analizadores léxicos, la búsqueda de patrones, la búsqueda de archivos, la creación de compiladores o problemas de aprendizaje, modelización y clasificación en el campo de la inteligencia artificial.

La computación natural [33] es el campo de investigación que trabaja sobre modelos y técnicas de computación inspirados en la naturaleza y que a la vez intenta describir el mundo natural que nos rodea en términos de procesos de información.

Podemos dividir la computación natural según la fuente de inspiración en la que se basa. Así tendremos por un lado la computación basada en fenómenos físicos y por otro la computación basada en fenómenos biológicos/bioquímicos.

Para mostrar la computación natural basada en fenómenos físicos vamos a ver un par de ejemplos. El primero es la computación óptica [22]; la idea de este modelo es substituir el uso de la electricidad en las computadoras convencionales por luz (los electrones por fotones), con esto se consigue una mayor capacidad y velocidad de transferencia y una gran cantidad de canales independientes para la comunicación. El rendimiento de una máquina capaz de trabajar con fotones es muy alto, porque aparte de las grandes ventajas en cuanto a trasferencia de datos, también nos da muchas facilidades para ejecuciones paralelas. El segundo ejemplo de este tipo de computación es la computación cuántica [52]; en este paradigma se substituyen los bits por qubits. Al utilizar las leyes de la mecánica cuántica un qubit puede tener los valores de 0 y 1, pero además puede valer 0 y 1 a la vez y una serie de valores complejos, lo que nos permite disponer en la práctica de una capacidad espacial exponencial dándonos opción a la computación en paralelo. Como vemos, estos dos modelos permiten la computación en paralelo, lo cual hace que algunos problemas puedan tener un nivel de complejidad menor que con un modelo de computación clásico.

En cuanto a la computación basada en fenómenos biológicos/bioquímicos vamos a hacer un recorrido desde lo más simple a lo más complejo. El primer nivel lo ocuparían los modelos de computación basados en cadenas de ADN, ARN o proteínas. Probablemente el referente más famoso de este tipo sea el utilizado por Adleman en su experimento realizado en el 1994 [2], donde consiguió resolver el problema del camino hamiltoniano con la manipulación de cadenas de ADN en tubos de ensayo. Lo que hace Adleman en su experimento es aprovechar la capacidad de replicación del ADN para conseguir un número exponencial de posibles soluciones en un tiempo polinómico, para

luego con un procedimiento de filtrado ver si existen soluciones al problema que se intenta resolver. Este experimento demostró que la computación con ADN es posible y si a nivel práctico en la actualidad no es de mucha utilidad, a nivel teórico conseguimos un avance muy importante, la generación de soluciones es exponencial en un tiempo polinómico, lo cual quiere decir que podemos rebajar el nivel de complejidad en la resolución de problemas. Otro proceso similar al utilizado por Adleman es el de computación de ADN basada en ciliados [36], donde se utiliza la recombinación para la creación de posibles soluciones, donde la recombinación es una extensión de la operación de splicing. El siguiente nivel lo ocuparían los modelos basados en el funcionamiento de una célula. Un ejemplo muy claro lo tenemos en la computación con membranas de los sistemas P [49], este modelo está compuesto por un conjunto de membranas, en cada membrada tenemos un multiconjunto de símbolos y un conjunto de reglas que podemos aplicar. En cada paso puede darse que ciertos elementos pasen de una membrana a otra o que alguna membrana pueda desaparecer. Otro ejemplo sería las redes de procesadores evolutivos [15], modelo que consiste en un conjunto de nodos conectados entre sí por una red donde en cada nodo se aplican reglas evolutivas (inserción, substitución y borrado) sobre un multiconjunto de cadenas, siendo posible la comunicación de dichas cadenas de un nodo a otro. Por último tenemos el nivel de modelos basados en tejidos. Un ejemplo son las Redes Neuronales [5], que se basan en las interacciones entre las neuronas creando una red de nodos con entrada y salidas ponderadas y cuyo funcionamiento consiste, para cada nodo, en la activación de la salida dependiendo del conjunto de entradas. Como segundo ejemplo de modelos basados en tejidos podemos poner la ampliación de los sistemas P y tendremos los sistemas P de tejidos [43], que trabaja con múltiples sistemas P emulando un conjunto de células y sus interacciones entre ellas. Finalmente también debemos mencionar el autómata celular [58], modelo que surgió en los años 40 y fue el primer modelo inspirado en la naturaleza que se conoce. Este modelo está inspirado en como actúan y se comunican un conjunto de células iguales y está constituido por una rejilla formada por celdas, donde cada celda contiene un valor real y en cada paso del proceso, cada celda recalcula su valor basándose en su antiguo valor y en el de sus celdas vecinas.

2.2. Algoritmos Genéticos

2.2.1. Introducción

Un Algoritmo Genético (GA) [46] es un algoritmo que simula la evolución biológica. Los GA se utilizan tanto como métodos de búsqueda para resolver problemas como modelos para sistemas evolutivos. Cuando trabajamos con un GA, tenemos un conjunto de cadenas guardadas y, a lo largo de un periodo de tiempo, las cadenas se ven modificadas de la misma manera que una población de individuos mediante la selección natural. Aunque este proceso sea simple comparado con el mundo natural, los GA son capaces de evolucionar de una forma compleja. Cada una de las cadenas almacenadas se llama individuo y dependiendo de para qué estemos utilizando el GA, puede representar una solución, una estrategia de juego, imágenes visuales, programas, etc.

Los GA están basados en ideas sobre genética de poblaciones. Al principio se crea una población (compuesta por una serie de individuos) de forma aleatoria. En los modelos más comunes, los individuos son representados mediante cadenas binarias y se puede ver como una codificación de una posible solución del problema que pretendemos resolver. Algunas variaciones entre individuos hacen que unos sean mejores que otros (por ejemplo que la solución que representan sea mejor). Estas diferencias son las que nos sirven para seleccionar el conjunto de individuos que se utilizará en el siguiente paso de tiempo. La selección consiste en eliminar los individuos menos aptos y quedarnos con los mejores (por ejemplo, si hablamos de soluciones, nos quedaremos con aquellas que den una mejor solución). En el modelo básico tenemos dos posibles operaciones: cruce (o crossover), que consiste en organizar las cadenas en pares, hacer un corte, e intercambiar una mitad con la otra cadena; y mutación, que consiste en seleccionar una cadena al azar y cambiar el valor de uno de sus bits, elegido también al azar. Resumiendo, en un GA tenemos una población inicial escogida al azar y en cada paso se modifica dicha población mediante operaciones de cruce o mutación, luego se evalúan las posibles soluciones y se escogen las n mejores, que pasarán a ser la nueva población. En general el proceso se repite un número k de pasos, pero se puede aplicar cualquier método de parada de los algoritmos que funcionan por aproximaciones.

2.2.2. Definición

Un Algoritmo Genético (AG) es un algoritmo probabilista formado por una población de individuos, $P(t) = \{x_1^t, ..., x_n^t\}$ donde t es la iteración en

la que se encuentra el algoritmo. Cada individuo representa una posible solución al problema que estemos estudiando y llamaremos S a la estructura de la codificación de todas ellas. Cada solución x_i^t se evalúa para obtener una medida de su adaptación o "fitness". Entonces, en el paso $t+1$, se elige una nueva población seleccionando a los mejores individuos (paso de selección). A algunos miembros de la nueva población se les aplican los llamados operadores "genéticos"(paso de modificación) con lo que se forman nuevas soluciones. Existe el operador unario m_i (mutación) que trabaja creando nuevos individuos cambiando un pequeño elemento de un individuo ya existente ($m_i : S \to S$) y el operador c_j (cruce) que combina partes (dos o más) de varios individuos para formar uno nuevo ($c_j : S \times ... \times S \to S$). Después de una serie de generaciones el programa converge y como solución nos quedaremos con el individuo (solución) que mejor fitness haya tenido en toda la computación (el mejor que ha podido encontrar el GA). A continuación se muestra la estructura que sigue un GA:

Procedimiento Algoritmo Genético

$t \leftarrow 0$
inicializar $P(t)$
evaluar $P(t)$
while (no se den las condiciones de parada) **do**
  $t \leftarrow t + 1$
  seleccionar $P(t)$ de $P(t-1)$
  aplicar operadores genéticos a $P(t)$
  evaluar $P(t)$
end while

2.2.3. Algoritmos Genéticos Paralelos

Si miramos el mundo podemos considerar a los humanos como una población global (esta es la visión de los Algoritmos Genéticos), pero si somos más estrictos podemos verla como un conjunto de poblaciones, que evolucionan de forma independiente, pero con la posibilidad de que algunos individuos pasen de unas poblaciones a otras. Esta es la visión básica del funcionamiento de los Algoritmos Genéticos Paralelos [4, 3, 14, 54].

Podemos definir un Algoritmo Genético Paralelo como un grafo dirigido donde cada nodo es un Algoritmo Genético clásico y cada arista nos indica la

posibilidad de migrar individuos de una población a otra. Con este modelo, aparte de los parámetros de cada Algoritmo Genético, tenemos que fijar lo siguiente: entre qué poblaciones se va a permitir la migración, la frecuencia de migración, los individuos que migran cada vez, comó se escogen estos individuos y qué individuos se borran después del proceso de migración.

La utilización de Algoritmos Genéticos Paralelos nos aporta, principalmente, más variabilidad al proceso, dado que cada población evoluciona de forma independiente, cada una puede intentar alcanzar un máximo local diferente (de la función de fitness), entonces podemos pensar que la combinación de individuos de estas poblaciones pueden darnos la variabilidad necesaria para llegar a una mejor solución. Hay que tener en cuenta que elegir los parámetros correctos es un proceso crítico, si aplicamos demasiada variabilidad el sistema no conseguirá converger a una buena solución y si aplicamos demasiado poca perderemos la potencia de utilizar varios Algoritmos Genéticos en paralelo. Se ha demostrado empíricamente que con los Algoritmos Genéticos Paralelos se obtienen mejores resultados que con los Algoritmos Genéticos clásicos.

2.3. Redes de Procesadores Bioinspirados

2.3.1. Introducción

Las Redes de Procesadores Bioinspirados son una clase de modelos de computación constituida por una arquitectura distribuida que trabaja sobre cadenas en paralelo. Cada procesador de esta arquitectura actúa sobre unos datos locales (un multiconjunto) aplicando una serie de operadores o reglas predefinidas. Por otra lado, estos datos locales, son enviados a través de una red de conexiones para formar parte o no de los datos locales de otro procesador. La forma de transmitir o aceptar los datos es gestionada por unos filtros de entrada y salida situados en cada uno de los procesadores. Todos los procesadores envían simultáneamente aquellos datos que pasen su filtro de salida y cada uno de ellos recibe aquellos datos que pasen su filtro de entrada.

En estos modelos los operadores o reglas que se puedan aplicar en cada procesador son lo que marcará el tipo de red con la que trabajemos. A continuación presentaremos dos modelos de redes previos al propuesto en esta tesis, las Redes de Procesadores Evolutivos, que van a trabajar con reglas de evolución (Substitución, Inserción y Borrado) y las Redes de Procesadores de Splicing, donde sus procesadores trabajan con operaciones de splicing.

2.3.2. Redes de Procesadores Evolutivos

Castellanos *et al.* propusieron un modelo inspirado en las mutaciones y evoluciones que suceden en el genoma del ADN, las Redes de Procesadores Evolutivos (NEP) [16, 15]. Este modelo es un modelo de Red de Procesadores Bioinspirados con la característica que sus procesadores pueden aplicar operaciones evolutivas (substitución, inserción y borrado) al conjunto de cadenas que contienen.

Definición

Sean P y F dos subconjuntos disjuntos del alfabeto V ($P, F \subseteq V \wedge P \cap F = \emptyset$) y sea $w \in V^*$. Definimos los predicados $\varphi^{(1)}$ y $\varphi^{(2)}$ como sigue:

1. $\varphi^{(1)}(w, P, F) \equiv (P \subseteq alph(w)) \wedge (F \cap alph(w) = \emptyset)$ (*predicado fuerte*)
2. $\varphi^{(2)}(w, P, F) \equiv (alph(w) \cap P \neq \emptyset) \wedge (F \cap alph(w) = \emptyset)$ (*predicado débil*)

Podemos extender estos predicados para que actúen sobre segmentos en vez de sobre símbolos. Sean P y F dos conjuntos finitos disjuntos de cadenas sobre V ($P, F \subseteq V^* \wedge P \cap F = \emptyset$) y sea $w \in V^*$. Extendemos los predicados $\varphi^{(1)}$ y $\varphi^{(2)}$ como sigue:

1. $\varphi^{(1)}(w, P, F) \equiv (P \subseteq seg(w)) \wedge (F \cap seg(w) = \emptyset)$ (*predicado fuerte*)
2. $\varphi^{(2)}(w, P, F) \equiv (seg(w) \cap P \neq \emptyset) \wedge (F \cap seg(w) = \emptyset)$
   (*predicado débil*)

Definimos una Red de Procesadores Evolutivos (NEP) de la forma siguiente:

$$\Gamma = (V, N_1, N_2, ..., N_n, G)$$

donde V es un alfabeto de trabajo, $N_i : 1 \leq i \leq n$ son cada uno de los procesadores de la red y $G = (\{N_1, N_2, ..., N_n\}, E)$ es un grafo no dirigido que muestra la forma de conectar los procesadores en la red, donde $\{N_1, N_2, ..., N_n\}$ son los nodos del grafo y E las aristas. Un procesador se define como $N_i = (M_i, A_i, PI_i, PO_i, FI_i, FO_i, \beta_i)$ y sus parámetros son los siguientes:

- M_i es un conjunto de reglas de evolución, donde las reglas de cada procesador son de uno de estos tres tipos:
  - $a \to b : a, b \in V$ (reglas de sustitución)
  - $a \to \varepsilon : a \in V$ (reglas de borrado)
  - $\varepsilon \to b : b \in V$ (reglas de inserción)

- A_i es el conjunto inicial de cadenas sobre V del nodo N_i. En adelante consideraremos que hay infinitas copias de cada una de las cadenas.
- PI_i es un subconjunto finito de V^* y define un filtro de segmentos permitidos de entrada.
- PO_i es un subconjunto finito de V^* y define un filtro de segmentos permitidos de salida.
- FI_i es un subconjunto finito de V^* y define un filtro de segmentos no permitidos de entrada.
- FO_i es un subconjunto finito de V^* y define un filtro de segmentos no permitidos de salida.
- $\beta_i \in \{(1), (2)\}$ define el tipo de filtros de entrada/salida del procesador. Más concretamente, para cualquier palabra $w \in V^*$ definimos el filtro de entrada $\rho(w) = \varphi^{\beta}(w, PI, FI)$ y el filtro de salida $\tau(w) = \varphi^{\beta}(w, PO, FO)$. $\rho(w)$ (resp. $\tau(w)$) indica si la palabra w pasa el filtro de entrada (resp. filtro de salida) del procesador. Podemos extender los filtros para trabajar sobre lenguajes. Entonces, $\rho(L)$ (resp. $\tau(L)$) es el conjunto de palabras de L que pueden pasar el filtro de entrada (resp. filtro de salida) del procesador.

Definimos configuración (o estado) de una NEP como la tupla $C = (L_1, L_2, ..., L_n)$ donde $L_i \subseteq V^*$ para todo $1 \leq i \leq n$. Una configuración representa el conjunto de cadenas (con multiplicidad infinita) presentes en cada procesador en un momento dado. Entonces, la configuración inicial de la NEP será $C_0 = (A_1, A_2, ..., A_n)$. La configuración puede cambiar cuando se aplica un paso de evolución o un paso de comunicación. Cuando se aplica un paso de evolución, cada componente L_i de la configuración es modificado de acuerdo con las reglas de evolución del procesador i teniendo en cuenta que podemos aplicar distintas reglas sobre distinta copias de una misma cadena (pues cada cadena tiene un número infinito de copias).

Formalmente decimos que la configuración $C_1 = (L_1, L_2, ..., L_n)$ cambia directamente a la configuración $C_2 = (L'_1, L'_2, ..., L'_n)$ mediante un paso de evolución ($C_1 \Rightarrow C_2$) si L'_i es el conjunto resultado de aplicar las reglas de M_i a las cadenas de L_i de la siguiente manera:

(1) Si la misma regla de sustitución o borrado puede ser aplicada en diferentes apariciones del mismo símbolo en una cadena, la regla se aplica en cada una de dichas apariciones pero en copias distintas de la misma cadena. El resultado es un multiconjunto en donde cada cadena obtenida mediante la aplicación de la regla aparece con infinitas copias.

(2) Una regla de inserción es aplicada en todas las posiciones de una cadena, pero cada una en una copia diferente. El resultado es un multiconjunto donde cada cadena que pueda ser obtenida mediante la aplicación de una regla de inserción aparece un número infinito de veces.

(3) Si más de una regla, no importa el tipo, puede ser aplicada a una cadena, todas ellas deben ser aplicadas en diferentes copias de dicha cadena.

En conclusión, como tenemos infinitas copias de cada cadena, después del paso de evolución tendremos un número infinito de copias de cada cadena obtenida mediante la aplicación de cada regla del conjunto de reglas del nodo donde estén. Por definición, si $L_i = \emptyset$ entonces $L'_i = \emptyset$.

Cuando se aplica un paso de comunicación, cada nodo N_i envía todas las copias de todas las cadenas que pasen el filtro de salida a todos los procesadores conectados a N_i y recibe todas las copias de aquellas cadenas enviadas por los procesadores conectados a N_i que puedan pasar el filtro de entrada. En este paso se puede perder cadenas, las cadenas perdidas serán aquellas que consigan salir de cualquiera de los procesadores pero que no consigan entrar en otro.

Formalmente, diremos que la configuración $C_1 = (L_1, L_2, ..., L_n)$ cambia directamente a la configuración $C_2 = (L'_1, L'_2, ..., L'_n)$ mediante un paso de comunicación $(C_1 \vdash C_2)$ si

$$\begin{aligned} L'_i =& L_i \setminus \{w \mid w \in L_i \cap \varphi^{\beta}(w, PO_i, FO_i)\} \cup \\ & \bigcup_{\{N_i,N_j\} \in E} \{x \mid x \in L_j \cap \varphi^{\beta}(w, PO_j, FO_j) \cap \varphi^{\beta}(w, PI_i, FI_i)\} \end{aligned}$$

para cada $1 \leq i \leq n$.

Dada una NEP $\Gamma = (V, N_1, N_2, ..., N_n, G)$, su computación es una secuencia de configuraciones $C_0, C_1, C_2, ...$, donde C_0 es la configuración inicial, $C_{2i} \Rightarrow C_{2i+1}$ y $C_{2i+1} \vdash C_{2i+2}$ para todo $i \leq 0$.

Según planteemos la ejecución de las redes podemos dividirlas en dos tipos, las aceptoras y las generadoras. Si trabajamos con redes generadoras, el resultado de una computación (ya sea finita o infinita) es el lenguaje formado por todas las palabras que aparecen en un nodo especial que llamaremos nodo de salida. Más formalmente, dada la secuencia de computación $C_0, C_1, C_2, ...$ y si definimos el nodo k como el nodo de salida, llamaremos $L_k(\Gamma)$ al lenguaje formado por todas aquellas palabras que hayan aparecido en el procesador k en cualquier paso de la computación. En cuanto a las redes aceptoras, una palabra pertenecerá al lenguaje aceptado por la red si al ponerla al principio de la ejecución en un nodo especial que llamaremos nodo de entrada, durante

la computación, aparece alguna cadena en otro nodo especial que llamaremos nodo de salida.

La complejidad temporal de computación para un conjunto no vacío de cadenas Z es el mínimo número de pasos t en una computación $C_0, C_1, ..., C_t$ tal que $Z \subseteq L_k$ donde L_k pertenece a C_t.

A la hora de definir NEP, las topologías de redes que más se utilizan son estrella, anillo o completa.

Una de las variantes más interesantes de las NEP son las Redes de Procesadores Evolutivos con los filtros en las conexiones [20]. En esta variante se cambian los filtros de entrada y de salida de cada nodo por filtros en las conexiones, viéndose así cada conexión como un canal de doble dirección donde los dos nodos conectados comparten el mismo filtro para comunicarse entre ellos. Como consecuencia de este cambio no existe la posibilidad que se pierdan cadenas en el paso de comunicación. Aunque parece que al trabajar con este tipo de redes perdamos capacidad para controlar la computación, está demostrado que esta no disminuye con respecto a las NEP clásicas.

Las NEP son capaces de resolver problemas NP-Completos en tiempo lineal; un ejemplo de ello la podemos ver en [16], donde esto se muestra con el problema de la correspondencia postal con pesos.

2.3.3. Redes de Procesadores de Splicing

Manea *et al.* propusieron una alternativa a las NEP, las Redes de Procesadores de Splicing (NSP) [39], que substituyen las operaciones evolutivas por operaciones de splicing sobre las cadenas del modelo.

En 1987, Tom Head definió por primera vez los sistemas de splicing [30] con el objetivo de establecer una relación entre la teoría de lenguajes y macromoléculas que contienen información (ADN, ARN o moléculas de proteínas). Estos sistemas se caracterizan por tener dos conjuntos de tripletas llamadas patrones, donde cada elemento de la tripleta es un segmento (puede ser vacío). Un conjunto nos marca los cortes que podemos hacer en una primera cadena sabiendo que nos vamos a quedar con la parte izquierda y el otro los cortes que podemos hacer en una segunda cadena sabiendo que nos quedamos con la parte derecha siendo el resultado la concatenación de los dos trozos que nos quedan. Para que se pueda realizar la operación de splicing la parte central de los dos patrones debe ser la misma. Así pues, si tenemos respectivamente los patrones (c, x, d) y (e, x, f) y como cadenas iniciales $ucxdv$ y $pexfq$ el resultado de la operación serán las cadenas $ucxfq$ y $pexdv$.

Años después Dennis Pitxon nos dio su definición de la operación de splicing [47]. En esta nueva versión ya no tenemos dos conjuntos diferentes de patrones, sino un sólo conjunto de reglas. Cada regla esta compuesta por

tres segmentos, el primero nos indica donde cortar en la primera cadena quedándonos con su parte izquierda, el segundo donde cortar en la segunda quedándonos con su parte derecha y el tercero es un nuevo segmento que aparecerá entre los dos trozos que nos quedan de dichas cadenas, ni el primer segmento ni el segundo formarán parte de la cadena resultado al realizar la concatenación. Si tenemos la regla (x, y, z) y las cadenas uxv y $u'yv'$ el resultado será la cadena uzv'.

Finalmente es Gheorghe Păun quien al poco tiempo nos da la definición que se usa en la actualidad de la operación de splicing [48]. En esta versión cada regla contiene dos pares de segmentos, el primer par nos indica el punto exacto (entre qué segmentos) hay que cortar la primera palabra y el segundo el corte para la segunda palabra dando lugar a dos cadenas resultado de concatenar el primer trozo de la primera con el segundo de la segunda y el segundo trozo de la primera con el primer trozo de la segunda. A continuación pasamos a definir formalmente la operación de splicing.

Definición

Una regla de splicing sobre un alfabeto V es una cuádrupla de la forma $[(u_1, u_2); (u_3, u_4)]$, con $u_1, u_2, u_3, u_4 \in V^*$, a estos segmentos los llamaremos contexto. Dada la regla $r = [(u_1, u_2); (u_3, u_4)]$ y un par de cadenas $x, y \in V^*$, decimos que:

$$\begin{aligned}\sigma_r(x, y) =&\{\{y_1u_3u_2x_2 | x = x_1u_1u_2x_2, y = y_1u_3u_4y_2\},\\ &\{x_1u_1u_4y_2 | x = x_1u_1u_2x_2, y = y_1u_3u_4y_2\}\}\end{aligned}$$

para cada $x_1, x_2, y_1, y_2 \in V^*$. Podemos extender la definición de la siguiente manera: Dado el conjunto de reglas de splicing R y los lenguajes L_1 y L_2

$$\sigma_R(L_1, L_2) = \bigcup_{r\in R, w_1\in L_1, w_2\in L_2} \sigma_r(w_1, w_2)$$

Definimos una Red de Procesadores de Splicing (NSP) de la forma siguiente:

$$\Gamma = (V, N_1, N_2, ..., N_n, G)$$

donde V es un alfabeto de trabajo, $N_i : 1 \leq i \leq n$ son cada uno de los procesadores de la red y $G = (\{N_1, N_2, ..., N_n\}, E)$ es un grafo no dirigido que muestra la forma de conexión de la red, donde $\{N_1, N_2, ..., N_n\}$ son los nodos del grafo y E las aristas. Un procesador se define como $N_i = (M_i, A_i, PI_i, PO_i, FI_i, FO_i, \beta_i)$ y sus parámetros son los siguientes:

- M_i es el conjunto de reglas de splicing en el procesador.
- A_i es un el conjunto inicial de cadenas sobre V del nodo N_i. En adelante consideraremos que cada cadena tiene una cantidad infinita de copias.
- PI_i es un subconjunto finito de V^* y define un filtro de segmentos permitidos de entrada.
- PO_i es un subconjunto finito de V^* y define un filtro de segmentos permitidos de salida.
- FI_i es un subconjunto finito de V^* y define un filtro de segmentos no permitidos de entrada.
- FO_i es un subconjunto finito de V^* y define un filtro de segmentos no permitidos de salida.
- $\beta_i \in \{(1), (2)\}$ define el tipo de filtros de entrada/salida del procesador. Más concretamente, para cualquier palabra $w \in V^*$ definimos el filtro de entrada $\rho(w) = \varphi^{\beta}(w, PI, FI)$ y el filtro de salida $\tau(w) = \varphi^{\beta}(w, PO, FO)$. $\rho(w)$ (resp. $\tau(w)$) indica si la palabra w pasa el filtro de entrada (resp. filtro de salida) del procesador. Podemos extender los filtros para trabajar sobre lenguajes. Entonces, $\rho(L)$ (resp. $\tau(L)$) es el conjunto de palabras de L que pueden pasar el filtro de entrada (resp. filtro de salida) del procesador.

El funcionamiento de una NSP es similar al de las NEP, con la diferencia que los procesadores de las NEP aplican operaciones evolutivas y los procesadores de las NSP's aplican operaciones de splicing (ver apartado anterior).

En [40] se ha demostrado que las NSP son un modelo computacional completo comprobando que para cualquier máquina de Turing se puede construir una NSP que acepta el mismo lenguaje. En el mismo trabajo también se demuestra que las NSP pueden resolver problemas NP-Completos en tiempo lineal, para ello se usa el problema de la satisfacibilidad (SAT).

Capítulo 3

Redes de Procesadores Genéticos

En este capítulo vamos a introducir un nuevo modelo de computación, las Redes de Procesadores Genéticos [13](NGP). Este nueva variante de Red de Procesadores Bioinspirados substituye las operaciones de inserción, borrado y substitución (de las NEP) o la operación de splicing (de las NSP) por la mutación clásica y el cruce entre cadenas (splicing con el contexto vacío). Por otra parte este modelo puede verse como un conjunto de Algoritmos Genéticos trabajando en paralelo.

Las NEP's son un modelo de computación completo, pero esto no está demostrado si trabajamos sólo con operaciones de substitución. Lo mismo sucede con las NSP's, se ha demostrado su completitud, pero no se ha podido hacer cuando trabajamos con operaciones de splicing con contexto vacío. Partiendo de esta base las Redes de Procesadores Genéticos son de sumo interés para el estudio del comportamiento al unir estas dos operaciones en un mismo tipo de red.

Los Algoritmos Genéticos Paralelos consisten en un conjunto de poblaciones diferentes evolucionando de forma independiente y unos mecanismos de comunicación para que diferentes poblaciones puedan intercambiar algunas soluciones. Las Redes de Procesadores Genéticos contienen ambos elementos, con lo cual se convierten en un modelo ideal para el estudio de la capacidad computacional de los Algoritmos Genéticos Paralelos.

Por otro lado en todos los modelos de los que bebe las NGP se han realizado estudios sobre la eficiencia de resolver problemas de optimización, con lo que es interesante observar su comportamiento cuando trabajamos en este campo.

3.1. Definición

Una regla de cruce es una operación sobre cadenas que definimos de la siguiente forma: Sean x e y dos cadenas, $x \bowtie y = \{x_1y_2, y_1x_2 : x = x_1x_2$ y $y = y_1y_2\}$. $x, y \in x \bowtie y$ dado que podemos tener ϵ como una de las dos partes de x o y. Si extendemos la operación sobre lenguajes tenemos $L_1 \bowtie L_2 = \bigcup_{x \in L_1, y \in L_2} x \bowtie y$.

Por otra parte la operación de cruce puede verse como una regla de splicing con contexto vacío, es decir como la regla de splicing $r = [(\varepsilon, \varepsilon); (\varepsilon, \varepsilon)]$.

Dado un alfabeto V, una regla de mutación $a \to b$, con $a, b \in V$, puede ser aplicada a la cadena xay para producir la nueva cadena xby (la regla de mutación se puede ver como la regla de substitución en las NEP).

Dado un alfabeto V, el lenguaje $L \subseteq V^*$, los contextos P y F (recordemos que los contextos pueden ser tanto conjuntos de símbolos como de segmentos) y dado $\beta \in \{(1), (2)\}$, definimos $\varphi^\beta(L, P, F) = \{w \in L | \varphi^\beta(w; P, F)\}$.

Un procesador genético se puede ver como una máquina abstracta capaz de aplicar reglas de mutación y de cruce sobre una población de cadenas.

Dado el alfabeto V, definimos un procesador N sobre V como la tupla $(M_R, A, PI, FI, PO, FO, \alpha, \beta)$ donde:

- M_R es el conjunto de reglas de mutación sobre V.
- A es un multiconjunto finito inicial de cadenas sobre V (Normalmente se trabaja con un número infinito de copias para cada cadena a no ser que se especifique lo contrario).
- $PI, FI \subseteq V^*$ son dos conjuntos finitos de segmentos que describen respectivamente los contextos permitidos y prohibidos de entrada.
- $PO, FO \subseteq V^*$ son dos conjuntos finitos de segmentos que describen respectivamente los contextos permitidos y prohibidos de salida.
- $\alpha \in \{1, 2\}$ define el modo de funcionamiento del procesador dependiendo de los siguientes valores:
  - Si $\alpha = 1$ el procesador sólo aplica reglas de mutación
  - Si $\alpha = 2$ el procesador sólo aplica reglas de cruce y $M_R = \emptyset$
- $\beta \in \{(1), (2)\}$ define el tipo de filtros de entrada/salida del procesador. Más concretamente, para cualquier palabra $w \in V^*$ definimos el filtro de entrada $\rho(w) = \varphi^\beta(w, PI, FI)$ y el filtro de salida $\tau(w) = \varphi^\beta(w, PO, FO)$. $\rho(w)$ (resp. $\tau(w)$) indica si la palabra w pasa el filtro

de entrada (resp. filtro de salida) del procesador. Podemos extender los filtros para trabajar sobre lenguajes. Entonces, $\rho(L)$ (resp. $\tau(L)$) es el conjunto de palabras de L que pueden pasar el filtro de entrada (resp. filtro de salida) del procesador.

Una Red de Procesadores Genéticos (NGP) de tamaño n se define como una tupla $\Pi = (V, N_1, N_2, ..., N_n, G, \mathcal{N})$ donde V es un alfabeto, $G = (X_G, E_G)$ es un grafo, N_i $(1 \leq i \leq n)$ es un procesador genético sobre V y $\mathcal{N} : X_G \to \{N_1, N_2, ..., N_n\}$ es una correspondencia que asocia cada procesador N_i a cada nodo $i \in X_G$.

Una configuración de una NGP $\Pi = (V, N_1, N_2, ..., N_n, G, \mathcal{N})$ se define como la tupla $C = (L_1, L_2, ...L_n)$, donde L_i es un multiconjunto definido sobre V para todo $1 \leq i \leq n$. Una configuración representa (al igual que en las NEP) los multiconjuntos de cadenas presentes en cada uno de los procesadores de la red en un momento dado. Así pues, la configuración inicial (en el paso 0) será $C_0 = (A_1, A_2, ..., A_n)$.

Cada copia de cualquier cadena de L_i puede ser modificada cuando se aplica un paso genético de acuerdo con las reglas de mutación y cruce asociadas con el procesador N_i. Formalmente decimos que la configuración $C_1 = (L_1, L_2, ..., L_n)$ cambia directamente a la configuración $C_2 = (L'_1, L'_2, ..., L'_n)$ mediante un paso genético $(C_1 \Rightarrow C_2)$ si L'_i es el conjunto resultado de aplicar las reglas de mutación y cruce de N_i a las cadenas de L_i. Como tenemos un número infinito de copias de cada cadena, después del paso genético, tendremos infinitas copias de cada posible resultado que se pueda obtener al aplicar las reglas de mutación o cruce. Por definición, si L_i es vacío para algún $1 \leq i \leq n$ entonces L'_i también es vacío. Cuando se aplica un paso de comunicación, cada nodo N_i envía todas las copias de todas las cadenas que pasen el filtro de salida a todos los procesadores conectados a N_i y recibe todas las copias de aquellas cadenas enviadas por los procesadores conectados a N_i que puedan pasar el filtro de entrada. Formalmente, diremos que la configuración C cambia directamente a la configuración C' mediante un paso de comunicación $(C \vdash C')$ si

$$C'(x) = (C(x) - \tau_x(C(x))) \cup \bigcup_{\{x,y\} \in E_G} (\tau_y(C(y)) \cap \rho_x(C(y))) \text{ para todo } x \in X_G$$

Podemos distinguir tres tipos de NGP: las aceptoras, las generadoras y las Redes de Procesadores Genéticos como Algoritmos Genéticos Paralelos.

3.1.1. Redes de Procesadores Genéticos Aceptoras

Las Redes de Procesadores Genéticos como aceptoras (ANGP) poseen dos procesadores concretos aparte de los demás, el de entrada y el de salida, respectivamente N_{input} y N_{output}. El proceso empieza colocando en N_{input} la cadena que queremos analizar (la llamaremos x) y luego se pone la máquina en funcionamiento; x será aceptada en el momento que una cadena cualquiera entre en el procesador N_{output}.

Una ejecución en una ANGP se detendrá en el momento en el que entre una cadena en el nodo N_{output} o si después de dos pasos genéticos consecutivos o dos pasos de comunicación consecutivos la configuración no cambia (no se puede aplicar ninguna regla en ningún nodo y no se transmite ni se recibe ninguna cadena). Si ninguna de estas dos condiciones se da la ejecución continuará indefinidamente. Por último, si en ningún momento de la ejecución ninguna cadena entra en N_{output}, diremos que la cadena a analizar x es rechazada, asumiendo que, con la condición de parada que utilizamos, no se iba a realizar ningún paso nuevo que pudiera introducir ninguna cadena en N_{output}.

3.1.2. Redes de Procesadores Genéticos Generadoras

Las Redes de Procesadores Genéticos como Generadoras (GNGP) poseen un procesador concreto aparte de los demás, el procesador de salida N_{output}. Las cadenas generadas por una GNGP son todas aquellas que en algún instante de tiempo t han estado en el procesador N_{output}.

Una ejecución de una GNGP consiste en ir recogiendo todas las palabras que son depositadas en N_{output}, como este proceso no tiene fin, el único criterio de parada que aplicaremos es detener la máquina si en dos pasos consecutivos, ya sean genéticos o de comunicación, la configuración no cambia (no se puede aplicar ninguna regla en ningún nodo y no se transmite ni se recibe ninguna cadena), la razón de aplicar este criterio de parada es que a partir de que se da la situación descrita no se va a generar ninguna palabra que no se haya generado ya. Si esta condición no se da la ejecución continuará indefinidamente.

3.1.3. Redes de Procesadores Genéticos como Algoritmos Genéticos Paralelos

Las Redes de Procesadores Genéticos como Algoritmos Genéticos Paralelos son un tipo de NGP's creada expresamente para trabajar con problemas de optimización. En este tipo de red las cadenas en los procesadores codifican

las posibles soluciones a un problema dado. En los filtros de cada procesador se define la función de fitness y además se comprueba que las soluciones pasen todas las restricciones del problema. En estas redes algunos procesadores se definen como procesadores de salida, estos son los que durante la computación irán comprobando las posibles soluciones para hallar la mejor solución aparecida durante todo el proceso. En los experimentos que nosotros hemos realizado, hemos utilizado todos los procesadores de la red como procesadores de salida.

En este tipo de redes, la ejecución consiste en ir generando posibles soluciones a un problema, con lo que sólo se detendrá en el momento que el sistema se estanque y no se puedan crear nuevas soluciones, así pues, el criterio para parar la ejecución es que después de dos pasos genéticos o de comunicación la configuración no cambie (no se puede aplicar ninguna regla en ningún nodo y no se transmite ni se recibe ninguna cadena).

3.2. Completitud

Decimos que un modelo de computación es completo si es capaz de aceptar o generar todos los lenguajes recursivamente enumerables (RE). Alternativamente podemos decir que dicho modelo tiene la capacidad computacional de una máquina de Turing o que es capaz de simular una máquina de Turing.

Teorema 1 *Las Redes de Procesadores Genéticos Aceptoras son computacionalmente completas.*

La demostración se basa en la simulación de la computación de cualquier cadena en una máquina de Turing M arbitraria. Demostraremos que siempre que M llegue a un estado de aceptación, la ANGP realiza una computación finita con al menos una cadena en el procesador de salida. Por otro lado, si la máquina M rechaza una cadena o no para, la computación de la ANGP tampoco aceptará dicha cadena.

Sea $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ una máquina de Turing arbitraria. Consideramos una descripción instantánea de M en la forma xq_iay, donde $x, y \in \Gamma^*$, $a \in \Gamma$ y $q_i \in Q$. Definimos los alfabetos $\Gamma' = \{a' : a \in \Gamma\}$ y $\bar{\Gamma} = \{\bar{a} : a \in \Gamma\}$.

Al principio la red codifica la descripción instantánea inicial q_0w como $q_0\bar{\$}wF$. El procesador N_c es el encargado de este proceso y lo definimos de la siguiente forma:

$$N_c = (\emptyset, \{w, q_0\bar{\$}, F\}, \emptyset, \emptyset, \{q_0\bar{\$}wF\}, \{aq_0, Fa : a \in (\Sigma \cup \{q_0, F, \bar{\$}\})\}, 2, (2))$$

donde w es la cadena de entrada utilizada en la demostración. El procesador N_c sólo utiliza operaciones de cruce entre w, $q_0\bar{\$}$ y F con $q_0\$wF \in q_0\bar{\$} \bowtie$

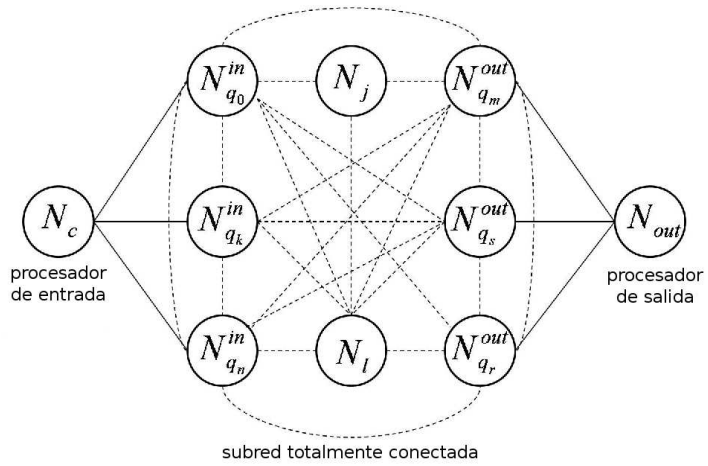

Figura 3.1: ANGP propuesta para simular el funcionamiento de una máquina de Turing arbitraria al procesar una cadena arbitraria.

$w) \bowtie F$. El filtro de salida (en este caso filtro de segmentos) nos asegura que la cadena que sale tiene la forma $q_0\bar{\$}wF$ (con los segmentos permitidos) y que no haya ningún símbolo a la derecha de F ni a la izquierda de q_0 (con los segmentos prohibidos). Además ninguna cadena puede entrar a este procesador dado que el filtro de entrada de segmentos permitido está vacío y el procesador trabaja en modo débil.

N_{out} es el procesador de salida y se define de la siguiente forma:

$$N_{out} = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, 2, (1))$$

Se puede observar que N_{out} acepta cualquier cadena. Probaremos que este procesador recibe una cadena si y sólo si la máquina de Turing para en un estado final.

De ahora en adelante utilizaremos $x, y \in \Gamma^*$, $a, b, c \in \Gamma$ y $q_i, q_j \in Q$. También tenemos una ANGP completa (cada nodo está conectado con todos los demás) que estará conectada a los procesadores N_c y N_{out}. Podemos ver la topología de la red en 3.1 a la que llamaremos $\hat{K}$.

La arquitectura de la red se describe de la siguiente manera: En el procesador de entrada la cadena de entrada se codifica para que pueda ser computada; se transforma la cadena inicial en la descripción instantánea inicial de acuerdo con la codificación que hemos señalado antes. El procesador de salida recibe una cadena siempre y cuando la cadena de entrada sea aceptada por la máquina de Turing. Finalmente, la subred completa está compuesta por varios procesadores: los procesadores N_q^{in} actúan como los estados antes de

aplicar un paso de la computación de la máquina de Turing; N_q^{out} toman el rol de los estados finales de la máquina de Turing; y, finalmente, el resto de procesadores, como N_j o N_i son definidos para llevar a cabo la simulación de cada paso de computación en la máquina de Turing. A continuación vamos a definir formalmente todos estos procesadores.

Definimos la red con la tupla $R = (V, N_c, N_1, N_2, ..., N_n, N_{out}, \hat{K}, f)$, donde $V = \Gamma \cup \Gamma' \cup \bar{\Gamma} \cup Q \cup \{F, \#, \$, \bar{\$}\}$, con $F, \#, \$, \bar{\$} \notin \Gamma$ y $\hat{K}$ es el grafo que representa la red mostrado en la figura 3.1. Los procesadores se definen como sigue:

1. Para cada estado $q \in Q$

   $N_q^{in} = (M_q^{in}, \emptyset, \{q, \bar{\$}\}, \{\#\}, \emptyset, \{j', \bar{j} : j \in \Gamma\} \cup \{\bar{\$}\}, 1, (1))$

   $M_q^{in} = \{k' \to k : k \in \Gamma\} \cup \{\bar{k} \to k : k \in \Gamma\} \cup \{\bar{\$} \to \$\}$

2. Para cada estado $q \in Q_f$

   $N_q^{out} = (\emptyset, \emptyset, \{q, \$\}, \{\#, \$F\}, \emptyset, \emptyset, 1, (1))$

3. $N_B = (\emptyset, \{\#BF\}, \{\$F, \#BF\}, \emptyset, V, \emptyset, 2, (2))$

4. $N_{B2} = (\emptyset, \{\#BF\}, \{\#BF\}, \{a\#, Fa : a \in V\}, V, \emptyset, 1, (2))$

5. Para cada estado $q_i \in Q$ y cada símbolo $a \in \Gamma$ tal que $\delta(q_i, a) = (q_j, b, R)$

   $N_{q_iaR} = (\{q_i \to q_j, \$ \to b', a \to \bar{\$}\}, \emptyset, \{q_i, \$a\}, \{\#, \$F\}, \{q_j, b'\bar{\$}\}, \{c\bar{\$} : c \in \Gamma\} \cup \{q_j\bar{\$}, \bar{\$}\bar{\$}\}, 1, (1))$

6. Para cada estado $q_i \in Q$ y cada símbolo $a \in \Gamma$ tal que $\delta(q_i, a) = (q_j, b, L)$ y para cada símbolo $c \in \Gamma$

   $N_{q_iacL1} = (\{\$ \to \bar{c}, q_i \to q_j, a \to b'\}, \emptyset, \{q_i, \$a\}, \{\#, \$F\}, \{q_j, c\bar{c}b'\}, \emptyset, 1, (1))$

7. Para cada estado $q_i \in Q$ y cada símbolo $a \in \Gamma$ tal que $\delta(q_i, a) = (q_j, b, L)$ y para cada símbolo $c \in \Gamma$

$$N_{q_iacL2} = (\{c \rightarrow \bar{\$}\}, \emptyset, \{q_j, c\bar{c}b'\}, \{\$, \#\} \cup \{kb',\ k'b' : k \in \Gamma \cup Q\}, \{q_j, \bar{\$}\bar{c}b'\},\ \{\bar{\$}k : k \in \Gamma\} \cup \{\$F\}, 1, (1))$$

Vamos a explicar el rol de cada uno de los procesadores que hemos definido. Las cadenas de la red codifican descripciones instantáneas de la máquina de Turing durante la computación. Los procesadores N_q^{in} son los que contendrán la codificación de la descripción instantánea después de la simulación de cada movimiento. Los procesadores N_q^{out} son los que reciben las codificaciones de descripciones instantáneas con estados finales. Los procesadores N_B y N_{B2} son los encargados de añadir un símbolo B después del símbolo F. Esto es necesario hacerlo cuando visitamos por primera vez nuevas celdas al mover el cabezal a la derecha. Los procesadores N_{q_iaR} son los encargados de realizar un movimiento que desplace el cabezal hacia la derecha. Los procesadores N_{q_iacL1} y N_{q_iacL2} son los encargados de realizar un movimiento que desplace el cabezal hacia hacia la izquierda. Aquí necesitamos considerar todos los posibles símbolos a la izquierda del cabezal, con lo que la simulación es más compleja que con los movimientos hacia la derecha.

Una vez claro el rol de cada procesador, vamos a describir el funcionamiento de la red: Primero $N_{q_0}^{in}$ recibe la codificación de la descripción instantánea inicial de N_c. Los demás procesadores N_q^{in} no recibirán ninguna cadena con un estado diferente a q. La codificación de la cadena de entrada w a $q_0\bar{\$}wF$ en el procesador N_c ya ha sido explicada previamente. Cada movimiento de la máquina de Turing se simula enviando la codificación de la descripción instantánea al procesador correspondiente, que es el que realizará el movimiento (se puede observar que hay procesadores que realizan un movimiento a la derecha, o grupos de dos procesadores que realizan un movimiento a la izquierda). Los procesadores simulan el movimiento y devuelven la nueva descripción instantánea al procesador N_q^{in} donde q es el estado que aparece en la instantánea. Cuando aparece la codificación $q_ix\$F$ significa que el cabezal ha llegado al símbolo más a la derecha de la cinta y que en esa celda tenemos un blanco. En este caso la cadena se envía al procesador N_B y este devuelve la cadena $q_ix\$BF$ al resto de procesadores. Cuando tenemos una codificación de la forma $q_ix\$yF$ con $q_i \in Q_f$, esta se envía al procesador $N_{q_i}^{out}$ que a su vez la envía al procesador N_{out}, donde la red se detendrá aceptando la cadena.

El procesador $N_{q_i}^{in}$ es el único que puede admitir cadena con los símbolos $\bar{\$}$ y q_i. El resto de procesadores N_q^{in} no admitirá una cadena que contenga un

estado diferente a q y los demás no admiten cadena con el símbolo $\bar{\$}$ (también podemos ver que algunos de ellos necesitan que la cadena contenga el segmento $\$a$). Por otra parte, la operatividad de la red hace que los procesadores N_q^{in} distribuyan la información a toda la red y reciban los resultados del resto de procesadores. El símbolo $\bar{\$}$ se utiliza para asegurarnos que sólo los procesadores N_q^{in} puedan recibir la descripción instantánea codificada después de simular un movimiento de la máquina de Turing.

La demostración formal de que la ANGP simula una máquina de Turing se realiza mediante inducción sobre el número de movimientos que lleva a cabo la máquina de Turing. La idea básica es que al principio y al final de cada movimiento de la máquina de Turing, la codificación de la descripción instantánea se encuentra en uno de los procesadores N_q^{in}. Vamos a demostrar el siguiente predicado formalmente:

Si $q_0 w \overset{*}{\underset{M}{\mapsto}} \alpha q \beta$ entonces

$$(\exists k \geq 0)[C_0 \Rightarrow C_1 \vdash C_2 \cdots \vdash C_k \text{ con } q\alpha\$\beta F \in C_k(N_q^{in})]$$

Caso Base

En el Caso Base de la inducción tenemos que la configuración inicial de la red representa la descripción instantánea inicial de la máquina de Turing $q_0 w = \alpha q \beta$. Si w es la cadena de entrada, el procesador $N_{q_0}^{in}$ recibe la cadena $q_0\$wF$ del procesador N_c con un paso de comunicación.

Hipótesis de Inducción

Si $q_0 w \overset{*}{\underset{M}{\mapsto}} \alpha q \beta$ en un máximo de j pasos, entonces

$$(\exists k \geq 0)[C_0 \Rightarrow C_1 \vdash C_2 \cdots \vdash C_k \text{ con } q\alpha\$\beta F \in C_k(N_q^{in})]$$

Paso de Inducción

Para el paso de inducción consideraremos que $q_0 w \overset{*}{\underset{M}{\mapsto}} \alpha_1 q_i \beta_1 \underset{M}{\mapsto} \alpha q \beta$. Debemos analizar todos los posibles movimientos que pueden darse al pasar de $\alpha_1 q_i \beta_1$ a $\alpha q \beta$ dependiendo del símbolo leído, el estado en el que se encuentra la máquina y el movimiento que va a realizar el cabezal (izquierda o derecha). El paso de $q_0 w$ a $\alpha_1 q_i \beta_1$ se realiza, como máximo, con j movimientos, entonces podemos afirmar por hipótesis de inducción que $q_i \alpha_1 \$ \beta_1 F \in C_k(N_q^{in})$.

Antes de ver todos los posibles movimientos, vamos a analizar el caso en el cual lleguemos a tener una codificación instantánea de la forma $q_i x\$F$ en el procesador $N_{q_i}^{in}$. En este caso la red se encarga de transformar la cadena $q_i x\$F$ en la cadena $q_i x\$BF$ añadiendo un blanco al final para poder aplicar el

siguiente movimiento. Sólo el procesador N_B puede recibir la cadena $q_i x\$F$, debido a que ningún otro procesador puede aceptar cadenas con el segmento $\$F$. Cuando esto pasa se realiza una operación de cruce entre la cadena $q_i x\$F$ y $\#BF$. Una de las cadenas obtenidas después de un paso genético es $q_i x\$BF$. En el siguiente paso de comunicación todas las cadenas de N_B salen del procesador (esto incluye también la cadena $\#BF$) pero sólo la cadena $q_i x\$BF$ pasará el filtro de entrada del procesador correspondiente para luego poder realizar el movimiento definido por $\delta(q_i, B)$. Ningún procesador N_q^{in} recibirá la nueva cadena, pues no tiene el símbolo $\bar{\$}$, esta irá directamente al procesador encargado de simular el siguiente movimiento. Al mismo tiempo N_{B2} envía la cadena $\#BF$, que sólo será admitida por N_B. Los procesadores N_B y N_{B2} trabajan sincronizados para que la cadena $\#BF$ esté siempre presente en el procesador N_B.

Una vez mostrado como la red solventa el problema de añadir blancos por la derecha, vamos a analizar cada movimiento:

Caso 1. Suponemos que el procesador $N_{q_i}^{in}$ contiene la cadena $q_i x\$ayF$ (que se corresponde con la descripción instantánea $\alpha_1 q_i \beta_1$ con $\alpha_1 = x$ y $\beta_1 = ay$) y el próximo movimiento a simular es $\delta(q_i, a) = (q_j, b, R)$. El único procesador que puede recibir la cadena es N_{q_iaR} dado que tiene el único filtro de entrada que acepta los segmentos q_i y $\$a$. Una vez la cadena entra en el procesador N_{q_iaR}, se le aplica la regla de mutación $q_i \to q_j$ para cambiar el estado y las reglas $\$ \to b'$ y $a \to \bar{\$}$ para simular el movimiento del cabezal y la reescritura del símbolo a por b. La regla $a \to \bar{\$}$ se puede aplicar a todos los símbolos a de la cadena, pero sólo la cadena $q_j x b' \bar{\$} y F$ (con $\bar{\$} \notin (alph(x) \cup alph(y))$) puede pasar el filtro de salida $PO = \{q_j, b'\bar{\$}\}$ y $FO = \{c\bar{\$} : c \in \Gamma\} \cup \{q_j\bar{\$}, \bar{\$}\bar{\$}\}$. Esta cadena es enviada fuera de procesador y es recibida por el procesador $N_{q_j}^{in}$ pues es el único con un filtro de entrada que admite los segmentos q_j y $\bar{\$}$. Finalmente en el procesador $N_{q_j}^{in}$ se aplican las reglas $b' \to b$ y $\bar{\$} \to \$$ dando como resultado la secuencia $q_j x b\$ yF$. Si aplicamos la regla $\delta(q_i, a) = (q_j, b, R)$ a la descripción instantánea xq_iay el resultado es xbq_jy y la cadena $q_j x b\$ yF$ se obtiene en $N_{q_j}^{in}$.

Caso 2. Suponemos que el procesador $N_{q_i}^{in}$ contiene la cadena $q_i xc\$ayF$ (que se corresponde con la descripción instantánea $\alpha_1 q_i \beta_1$ con $\alpha_1 = xc$ y $\beta_1 = ay$) y el próximo movimiento a simular es $\delta(q_i, a) = (q_j, b, L)$. El procesador N_{q_iacL1} es quien recibe la cadena, debido a que es el único que contiene en sus filtros permitidos de entrada q_i y $\$a$. En N_{q_iacL1} se aplican las mutaciones $\$ \to \bar{c}$, $a \to b'$ y $q_i \to q_j$, pero sólo la cadena $q_i x c\bar{c} b' y F$ puede pasar el filtro ya que $PO = \{q_j, c\bar{c}b'\}$. En este punto la cadena $q_j x c\bar{c} b' y F$ no contiene ningún símbolo $\$$ con lo que sólo puede entrar en el procesador N_{q_iacL2}. En este procesador aplicamos la regla $c \to \bar{\$}$ que junto con su filtro de salida

nos asegura que sólo las palabras con el segmento $\bar{\$}\bar{c}b'$ pueden salir de el. La cadena resultante $q_j x\bar{\$}\bar{c}b'yF$ deja el procesador y sólo puede ser admitida por $N_{q_j}^{in}$ debido al estado actual y al símbolo $\bar{\$}$. Finalmente el procesador $N_{q_j}^{in}$ aplica las reglas $b' \to b$, $\bar{c} \to c$, i $\bar{\$} \to \$$ que dan como resultado la cadena $q_j x\$cbyF$ que se corresponde con la descripción instantánea xq_jcby.

Caso 3. Suponemos que el procesador $N_{q_i}^{in}$ contiene la cadena $q_i\$ayF$ y el próximo movimiento a simular es $\delta(q_i, a) = (q_j, b, L)$. En este caso la máquina de Turing para y rechaza la cadena de entrada, ya que el cabezal no puede moverse a la izquierda de la primera celda de la cinta. En la simulación sólo el procesador N_{q_iacL1} puede recibir la cadena sobre la que aplicará la regla $\$ \to \bar{c}$. Dado que el símbolo \$ aparece justo a la derecha del símbolo q_i, la cadena resultado de la mutación será $q_j\bar{c}ayF$. Esta nueva cadena no puede pasar el filtro de salida por no contener el segmento $c\bar{c}$, con lo que permanecerá en el procesador. Llegados a este punto, ninguna cadena puede ser comunicada, además, las mutaciones que se pueden aplicar en cada uno de los procesadores son finitas y los procesadores N_B y N_{B2} se intercambian entre ellos la cadena $\#BF$. Todo esto hace que la red repita dos configuraciones seguidas y pare en modo de no aceptación.

Vamos a ver ahora el caso en que la máquina de Turing llega a una situación de aceptación. Supongamos que el procesador $N_{q_i}^{in}$ contiene la cadena $q_i x\$yF$ con $q_i \in Q_f$. En este caso la máquina de Turing para y acepta la cadena de entrada. La red envía la cadena a los procesadores N_q^{out}. El procesador $N_{q_i}^{out}$ recoge la cadena y la envía al procesador N_{out}. En este punto la red para y acepta la cadena de entrada.

En esta demostración ha quedado excluido el caso $\epsilon \in L(M)$. Cuando esto sucede la red procede de la siguiente forma: la descripción instantánea inicial es q_0 con $q_0 \in Q_f$ y la correspondiente codificación es $q_0\bar{\$}F$. El procesador $N_{q_0}^{in}$ recibe la cadena $q_0\bar{\$}F$ y le aplica la regla $\bar{\$} \to \$$. Posteriormente, la cadena $q_0\$F$ es enviada al procesador N_B donde se le pone el símbolo de blanco al final y finalmente puede ser aceptada por $N_{q_0}^{out}$, con lo cual la cadena vacía es aceptada por la red.

En el caso en que la máquina de Turing no tenga movimiento definido ningún procesador recibe la descripción instantánea codificada, quedando sólo activos los procesadores N_B y N_{B2}, pero estos procesadores sólo se intercambian simultáneamente la cadena $\#BF$, con lo que dos configuraciones consecutivas de la red se repetirán y la computación parará en modo de no aceptación. Finalmente podemos ver que si la máquina de Turing realiza una computación infinita la red también realizará una computación infinita y la cadena de entrada no se aceptará nunca.

Hemos propuesto una ANGP que realiza la simulación dada una cadena de entrada w y una máquina de Turing M. El número de procesadores usados en la simulación depende del tamaño de la función de transición de M. Este número está acotado por $2 \cdot |Q| + |Q| \cdot |\Gamma| + 2 \cdot |Q| \cdot |\Gamma|^2 + 4$. Hay que tener en cuenta que esto es una cota superior del número de procesadores usados, puesto que se utiliza un sólo procesador si se realiza un movimiento a la derecha, o dos si realizamos un movimiento a la izquierda. Los dos casos no pueden ocurrir simultáneamente porque estamos trabajando con el modelo determinista.

También podríamos trabajar con una red que aceptara directamente las cadenas de entrada de la máquina de Turing codificadas, sólo tendríamos que eliminar el procesador N_c y que el procesador $N_{q_0}^{in}$ recibiera la cadena codificada $q_0\$wF$.

Simulación de Máquinas de Turing no Deterministas

En el caso anterior hemos visto como una ANGP simula una máquina de Turing determinista para demostrar que el modelo es completo. Ahora vamos a ver como una ANGP puede simular de forma directa una máquina de Turing no determinista. Aunque este resultado no nos aporte nada nuevo en cuanto a la completitud del modelo, sí es interesante para enunciar algunos resultados sobre complejidad.

Teorema 2 *Cualquier máquina de Turing no determinista puede ser simulada por una ANGP.*

Una máquina de Turing no determinista se diferencia de una determinista en la definición de la función de transición. Ahora tenemos $\delta(q, a) = \{(q_1, a_1, z_1), \cdots, (q_p, a_p, z_p)\}$. En el caso determinista definimos un procesador para cada valor de la función de transición. En el caso no determinista, vamos a definir un procesador para cada posible elección de la función de transición. Vamos a usar la misma topología de red que en el caso determinista. Los procesadores N_c y N_{out} se definen de la misma forma que en el caso determinista, la definición del resto de procesadores la tenemos a continuación:

1. Para cada estado $q \in Q$

   $N_q^{in} = (M_q^{in}, \emptyset, \{q, \bar{\$}\}, \{\#\}, \emptyset, \{j', \bar{j} : j \in \Gamma\} \cup \{\bar{\$}\}, 1, (1))$, donde

   $M_q^{in} = \{k' \to k : k \in \Gamma\} \cup \{\bar{k} \to k : k \in \Gamma\} \cup \{\bar{\$} \to \$\}$

2. Para cada estado $q \in Q_f$

   $N_q^{out} = (\emptyset, \emptyset, \{q, \$\}, \{\#, \$F\}, \emptyset, \emptyset, 1, (1))$

3. $N_B = (\emptyset, \{\#BF\}, \{\$F, \#BF\}, \emptyset, V, \emptyset, 2, (2))$

4. $N_{B2} = (\emptyset, \{\#BF\}, \{\#BF\}, \{a\#, Fa : a \in V\}, \emptyset, \emptyset, 1, (2))$

5. Para cada par de estados $q_i, q_j \in Q$ y para cada par de símbolos $a, b \in \Gamma$ tal que $(q_j, b, R) \in \delta(q_i, a)$

   $N_{q_iaq_jbR} = (\{q_i \rightarrow q_j, \$ \rightarrow b', a \rightarrow \bar{\$}\}, \emptyset, \{q_i, \$a\}, \{\#, \$F\}, \{q_j, b'\bar{\$}\}, \{c\bar{\$} : c \in \Gamma\} \cup \{q_j\bar{\$}, \bar{\$}\bar{\$}\}, 1, (1))$

6. Para cada par de estados $q_i, q_j \in Q$ y para cada par de símbolos $a, b \in \Gamma$ tal que $(q_j, b, L) \in \delta(q_i, a)$ y para cada símbolo $c \in \Gamma$

   $N_{q_iaq_jbcL1} = (\{\$ \rightarrow \bar{c}, q_i \rightarrow q_j, a \rightarrow b'\}, \emptyset, \{q_i, \$a\}, \{\#, \$F\}, \{q_j, c\bar{c}b'\}, \emptyset, 1, (1))$

7. Para cada par de estados $q_i, q_j \in Q$ y para cada par de símbolos $a, b \in \Gamma$ tal que $(q_j, b, L) \in \delta(q_i, a)$ y para cada símbolo $c \in \Gamma$

   $N_{q_iaq_jbcL2} = (\{c \rightarrow \bar{\$}\}, \emptyset, \{q_j, c\bar{c}b'\}, \{\$, \#\} \cup \{kb', k'b' : k \in \Gamma \cup Q\}, \{q_j, \bar{\$}\bar{c}b'\}, \{\bar{\$}k : k \in \Gamma\} \cup \{\bar{\$}F\}, 1, (1))$

El funcionamiento de la ANGP es el mismo que el descrito en la demostración del Teorema 1. La única diferencia es que la codificación de la descripción instantánea $q\alpha\$a\beta F$ que se envía desde el procesador N_q^{in} puede ser aceptada por mas de un procesador N_{qaq_jbR} o N_{qaq_jbcL1}. En el caso de que la cadena $q\alpha\$a\beta F$ entre, al menos, en dos procesadores N_{qaq_jbR}, las transformaciones se realizarán de forma independiente en cada procesador y estas nuevas cadenas serán las que luego se acepten en los procesadores N_q^{in}. Si más de una cadena entra en el mismo procesador N_q^{in} las transformaciones sobre ellas se realizarán de forma independiente y luego serán comunicadas de acuerdo a los movimientos definidos por la máquina de Turing. En el caso

de que la cadena entre en más de un procesador N_{qaq_jbcL1}, las transformaciones en cada procesador se hacen de manera independiente a los otros. En los procesadores N_{qaq_jbcL2} las transformaciones dependen del símbolo a la izquierda del que se ha cambiado, con lo que estos cambios se realizan de forma independiente. Finalmente, en el caso de que la cadena entre en procesadores N_{qaq_jbR} y N_{qaq_jbcL1} no hay problema puesto que las cadenas entran en procesadores distintos. Por otro lado el funcionamiento de N_B y N_{B2} nos asegura que si más de dos cadenas están dentro de ellos, el cruce entre ellas se perderá en el siguiente paso de comunicación (sólo el cruce con un blanco codificado sera válido para continuar con la computación).

3.2.1. Introducción de medidas de complejidad temporal

Hemos dado una simulación de las máquinas de Turing no deterministas con las ANGP's, ahora podemos centrar nuestra atención en la complejidad temporal de este nuevo modelo. La complejidad temporal de las Redes de Procesadores Evolutivos (NEP's) se definió por primera vez en [41]. Vamos a utilizar las mismas definiciones que en dicho trabajo.

Primero, podemos establecer la siguiente medida de complejidad para las ANGP: Consideramos una ANGP R que para cualquier entrada se detiene y el lenguaje L aceptado por R. Escribimos la complejidad temporal de una computación que termina en modo aceptación de R, si x es la cadena de entrada, como $Time_R(x)$ y se define como el número de pasos (tanto de comunicación como de computación) que la red necesita para parar en modo de aceptación. Hay que tener en cuenta que la medida no está definida cuando la máquina no para o cuando repite dos configuraciones consecutivas. Esta medida satisface completamente los axiomas de Blum para medidas de complejidad abstractas [11]. Podemos definir la función parcial $Time_R : \mathbb{N} \to \mathbb{N}$ como sigue:

$$Time_R(n) = max\{Time_R(x) : x \in L(R), |x| = n\}$$

Tomando una función entera $f : \mathbb{N} \to \mathbb{N}$, podemos definir la siguiente clase de lenguajes siempre que exista una ANGP R que cumpla la propiedad requerida en la definición

$$\begin{aligned} Time_{ANGP}(f) = \{L :\ & \text{existe una ANGP, R y un número natural } n_0 \\ & \text{tal que } L = L(R) \text{ y } \forall n \geq n_0\ Time_R(n) \leq f(n)\} \end{aligned}$$

Finalmente, para un conjunto de funciones enteras $\mathbb{C}$ definimos

$$Time_{ANGP}(\mathbb{C}) = \bigcup_{f \in \mathbb{C}} Time_{ANGP}(f)$$

Consideramos el conjunto de funciones enteras *poly* como el conjunto de funciones polinómicas enteras y lo denotamos como $Time_{ANGP}(poly)$ para preservar la notación clásica de la teoría de la complejidad computacional. Vamos a enunciar y demostrar lo siguiente:

Lema 1 $\mathcal{NP} \subseteq PTIME_{ANGP}$.

Si un lenguaje $L \in \mathcal{NP}$, entonces existe una máquina de Turing no determinista M que acepta cualquier cadena de L en tiempo polinómico. En ese caso, podemos construir una ANGP de M que trabaje con la misma cadena de entrada de acuerdo con el teorema 2. La ANGP lleva a cabo un número constante máximo de pasos (genéticos y de comunicación) para simular un movimiento de la máquina de Turing.

Podemos hacer el siguiente análisis para garantizar que la afirmación previa es cierta: Primero, si una máquina de Turing realiza una transición usando un movimiento a la derecha (por ejemplo (q, a, R)), entonces la ANGP realiza tres pasos genéticos para simular dicho movimiento (cambia el estado, substituye el símbolo y mueve el cabezal de posición). Entonces, la nueva cadena se comunica al procesador N_q^{in} que realiza dos pasos genéticos para cambiar el símbolo a' por a y $\bar{\$}$ por \$. Segundo, si la máquina de Turing realiza un cambio de configuración por un movimiento hacia la izquierda (por ejemplo (q, a, L)), entonces la ANGP realiza tres pasos genéticos para simular este movimiento (cambia el estado, substituye el símbolo y mueve el cabezal de posición) en el procesador $N_{q_i a q_j b c L1}$ y uno en el procesador $N_{q_i a q_j b c L2}$. La nueva cadena se comunica al procesador N_q^{in} donde se realizan tres pasos genéticos, para cambiar los símbolos a' por a, $\bar{b}$ por b y $\bar{\$}$ por \$. Finalmente, si la máquina de Turing explora nuevas celdas con el símbolo blanco, la red realizará dos pasos genéticos (en este caso con la operación de cruce en el procesador N_B) para añadir este símbolo extra a la cadena.

La ANGP propuesta en el teorema 2 simula el funcionamiento de la máquina de Turing considerando todas las combinaciones de transiciones posibles simultáneamente. Si la máquina de Turing trabaja en tiempo polinómico entonces la ANGP también trabaja en tiempo polinómico, dado que la simulación de cada transición de la máquina de Turing requiere un número constante de pasos en la red. Por lo tanto $\mathcal{NP} \subseteq PTIME_{ANGP}$.

Dejamos abierto el problema de saber si $\mathcal{NP} \subset PTIME_{ANGP}$ para futuros trabajos.

3.2.2. Ciclo Hamiltoniano. Estudio de como resolver un problema NP y su complejidad mediante las ANGP's

Una vez definida la complejidad para las ANGP's vamos a poner como ejemplo un problema (que pertenezca a la clase NP) y a realizar un estudio de su complejidad utilizando NGP's. El problema seleccionado es el problema de decisión del ciclo hamiltoniano en un grafo no dirigido.

Definición del Problema

Dado un grafo, diremos que tiene un ciclo hamiltoniano si existe un camino que pasa por cada vértice exactamente una sola vez y que además empieza y termina por el mismo vértice. Podemos definir el problema de decisión del ciclo hamiltoniano como, dado un grafo, averiguar si posee o no un ciclo hamiltoniano.

Configuración de la NGP

Para decidir el alfabeto y la codificación vamos a tener en cuenta que en el problema seleccionado tenemos una talla (número de nodos) a determinar por la entrada del mismo (el grafo que define la instancia del problema) y esto nos obligaría a codificar el identificador de cada vértice del grafo como una cadena (y controlarlo con los filtros) para poder tener un alfabeto finito. Como la idea inicial es hacer un estudio sencillo y fácilmente entendible, vamos a acotar el problema a 50 vértices y a utilizar los 50 primero números enteros (del 0 al 49) para codificarlos. Esta medida la podemos asumir porque la reducción de la complejidad que implica es constante.

Pasemos a definir la red, dado el grafo $G = (V, E)$, donde V es el conjunto de n vértices acotado a un máximo de 50 y E es el conjunto de aristas, definimos la red con la tupla $R = (V, N_0, N_1, N_2, ..., N_{49}, N_c, N_{pc}, N_{out}, K_{53}, f)$ donde $V = \{0, 1, 2, ..,49\} \cup \{0', 1', 2', ..,49'\} \cup \{\#, \#', 0^*\}$ y K_{53} es una red de 53 nodos, que contiene dos nodos centrales N_c y N_{pc} conectados entre si y de los cuales existen conexiones a todos los nodos $N_i : 0 \leq i \leq 49$, el nodo N_{out} aparece independiente y con una sola conexión con N_{pc}.

Los procesadores quedan definidos de la siguiente forma:

Para cada vértice $q \in V$
$N_q = (\{\# \rightarrow q'\}, \emptyset, \{p\# : (p, q) \in E\} \cup \{0^*\# : (0, q) \in E\}, \{q\}, \{q'\}, \emptyset, 1, (2))$

$N_c = (\emptyset, \{\#'\}, \{n' : 0 \leq n \leq 49\}, \{\#, \#'\}, \{n' : 0 \leq n \leq 49\} \cup \{n : 0 \leq n \leq 49\} \cup \{\#'\#', 0^*\}, \emptyset, 2, (2))$

$N_{pc} = (\{q' \to q : 0 \leq q \leq 49\} \cup \{\#' \to \#\}, \{0^*\#\}, \{n'\#' : 0 \leq n \leq 49\}, \emptyset, \{n : 0 \leq n \leq 49\} \cup \{0^*\#\}, \{n' : 0 \leq n \leq 49\} \cup \{\#'\}, 1, (2))$

$N_{out} = (\emptyset, \emptyset, \{n : 0 \leq n \leq 49\}, \emptyset, \emptyset, \emptyset, 1, (1))$

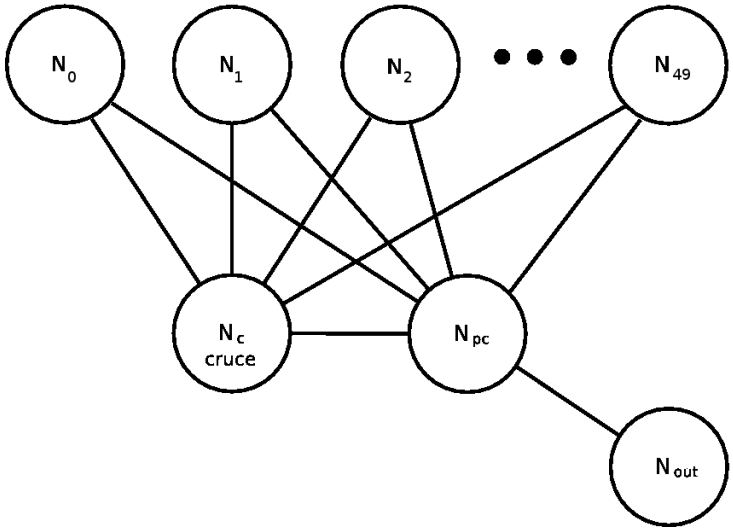

Figura 3.2: Red para solucionar el problema de decisión de saber si un grafo contiene un ciclo hamiltoniano para grafos de un tamaño máximo de 50 nodos.

Funcionamiento de la Red

El proceso empieza en el nodo N_{pc}, con la cadena $0^*\#$, el significado de la cadena es que vamos a empezar por el vértice 0 y el símbolo # nos sirve para marcar el final de la cadena y poder añadir nuevos vértices. Esta cadena sale del procesador y entra en aquellos $N_i : 0 \leq i \leq 49$ tal que (0,i) es una arista del grafo. Cada procesador $N_i : 0 \leq i \leq 49$ tiene una regla que substituye el símbolo # por el símbolo i' (su identificador de vértice marcado) seguidamente, la secuencia pasa al nodo N_c que se usa para anexarle al final el símbolo $\#'$ y después al nodo N_{pc} que sirve para limpiar todas las marcas de los símbolos. En este punto, en el procesador N_{pc}, tendremos una cadena en la que lo más interesante es el símbolo anterior a # (que codifica el vértice del que venimos), vamos a llamarlo j, en el siguiente paso la cadena entra en aquellos nodos de entre $N_i : 0 \leq i \leq 49$ que cumplan que (j, i) es una arista del grafo y que no se haya pasado ya por el vértice

que representa el procesador N_i (esto lo controla el filtro de entrada de cada procesador). A partir de aquí el proceso se repite una y otra vez. Si durante la computación conseguimos una cadena que contenga todos los vértices (los que tiene el grafo, sin contar el símbolo 0^* que es por el que comenzamos y con el que conseguimos contruir el ciclo), esta conseguirá pasar el filtro de entrada de N_{out} y podremos detener la computación y responder SI a la solución del problema. Si por el contrario el grafo no permite un ciclo hamiltoniano, en la computación se verá reflejado en que ninguna cadena podrá entrar en ningún procesador $N_i : 0 \leq i \leq 49$, con lo que no habrá movimiento de cadenas, la red repetirá dos configuraciones consecutivas y el proceso terminará respondiendo NO.

Experimentación y Cálculo de Complejidad

Recordemos que la complejidad de una NGP al resolver un problema se calcula contando el número de pasos (tanto genéticos como de comunicación) que se necesitan para realizar la computación.

Una vez definida la red que vamos a usar para la resolución del problema, ya sólo nos queda realizar la experimentación para diferentes tamaños de grafos, para ello hemos decidido mostrar sólo resultados de instancias que sí tenían ciclo hamiltoniano, pues esos son los casos en que la computación dura más. También comentar que en los experimentos se ha variado tanto el número de vértices, como el número de aristas. En la tabla 3.3 podemos ver los resultados.

Si estudiamos el resultado de la experimentación, la tabla no hace más que corroborar los resultados que se pueden deducir repasando el funcionamiento de la red, primero podemos ver que el número de aristas no influye en la complejidad, esto se debe a que la red está hecha para ir probando todos los posibles caminos en paralelo con lo que si aumenta el número de aristas aumenta la carga en paralelo, pero no la complejidad. Este hecho sí nos pasa factura a la hora de simular la red en un ordenador (ya que el nivel de paralelismo queda reducido al número de procesadores de la máquina), donde a mayor número de aristas mayor es el tiempo que se tarda en realizar la computación.

En cuanto a la complejidad en la tabla se infiere que si N es el número de vértices, el número de pasos que necesita la computación es $8 * N$. Si observamos el funcionamiento de la red, simular un paso de un vértice a otro equivale al movimiento de una cadena por 4 procesadores, lo cual nos da un total de 8 pasos (4 pasos genéticos y 4 pasos de comunicación), así pues para un grafo de N vértices se necesitan $8 * N$ pasos, con todo esto podemos afirmar que tenemos una complejidad lineal $O(N)$.

Número de Vértices	Número de Aristas	Pasos de la Computación
5	8	40
5	10	40
5	12	40
10	18	80
10	20	80
10	22	80
15	28	120
15	30	120
15	32	120
20	38	160
20	40	160
20	42	160

Figura 3.3: Experimentación con NGP's para la resolución del problema de decisión del ciclo hamiltoniano.

Resumen y Conclusiones

Una vez definida la complejidad para las NGP's queríamos poner esta medida en práctica y al mismo tiempo comprobar la potencia de nuestro modelo, para ello hemos elegido un problema NP, el problema de comprobar si un grafo contiene un ciclo hamiltoniano. Hemos construido la red y hemos realizado una experimentación variando los vértices y las aristas del grafo. Finalmente hemos observado que la complejidad es lineal con el número de vértices. La conclusión más importante que podemos extraer es que conseguimos resolver un problema NP con Complejidad lineal utilizando nuestro modelo, siempre con la salvedad que para realizar un modelo de este tipo necesitamos diseñar un sistema paralelo, cosa que hoy en día puede ser llevado a cabo aunque con ciertas limitaciones.

3.2.3. Redes de Procesadores Genéticos y Algoritmos Genéticos Paralelos

En este apartado vamos a introducir el modelo de los Algoritmos Genéticos Paralelos y cómo se relaciona con las Redes de Procesadores Genéticos.

Los Algoritmos Genéticos Paralelos (PGA's) fueron propuestos para mejorar la eficiencia de los Algoritmos Genéticos (GA) estándar en su búsqueda de la soluciones óptimas. Viendo las referencias [4, 3, 14, 54], los principales aspectos para proponer GA distribuidos y paralelos son los siguientes:

- La distribución de los individuos en diferentes poblaciones (maestro-esclavo, múltiples poblaciones o islas, poblaciones de grano fino o jerárquicas y poblaciones híbridas) y la topología del modelo (anillo, completa, rejilla, etc.).

- La sincronización de la evolución y la comunicación de las poblaciones.

- El fenómeno de la migración: El índice de migración (el porcentaje de individuos que pasan de una población a otra, la selección de migración (que individuos van a migrar) y la frecuencia de migración.

Todos estos elementos han sido introducidos en el modelo de la NGP que hemos propuesto en esta tesis: una topología de poblaciones completa, un reloj universal para sincronizar los pasos de computación y comunicación, un índice y una frecuencia de migración del cien por cien y una selección de migración basada en los filtros de entrada y de salida de los procesadores. Las Redes de Procesadores Genéticos como aceptoras que hemos definido previamente no están hechas para resolver problemas de optimización, sino problema de decisión. Entonces, para poder ver nuestro modelo como una propuesta clásica de PGA's necesitamos formular los PGA's como técnicas para resolver problemas de decisión.

A continuación vamos a proponer dos criterios razonables para trabajar con PGA's como modelos de resolución de problemas de decisión:

- *Criterio de Aceptación I (AC-I)*

  Sea w una cadena de entrada. Diremos que un PGA acepta w si, después de un número finito de iteraciones (aplicar operaciones, selección con la función de fitness y migraciones individuales), w aparece en una población predefinida de supervivientes.

- *Criterio de Aceptación II (AC-II)*

  Sea w una cadena de entrada. Diremos que un PGA acepta w si, después de un número finito de iteraciones (aplicar operaciones, selección con la función de fitness y migraciones individuales), el individuo x_{yes} aparece en una población predefinida de supervivientes. Diremos que el PGA rechaza la cadena de entrada si, después de un número finito de iteraciones (aplicar operaciones, selección con la función de fitness y migraciones individuales), el individuo x_{not} aparece en una población predefinida de supervivientes.

Demostraremos que los dos criterios de aceptación son equivalentes a través del siguiente resultado:

Teorema 3 *Sea D un problema de decisión y L_D su lenguaje de aceptación. D puede ser resuelto por un Algoritmo Genético Paralelo con Criterio de Aceptación I si y sólo si puede ser resuelto con Criterio de Aceptación II.*

Demostración Supongamos que D puede ser resuelto por un Algoritmo Genético Paralelo A_{D-I} con $AC-I$. Después de un número finito de operaciones y selección de supervivientes por la función de fitness, A_{D-I} tiene una población predefinida con el individuo w. Podemos pues formular un PGA A_{D-II} con $AC-II$ de la siguiente manera: A_{D-II} simula A_{D-I} y obtiene en la población predefinida de A_{D-I} la cadena w. Supongamos que g es la función de fitness de la población predefinida de A_{D-I}, definimos g' como sigue:

$$g'(x) = \begin{cases} g(y)+1 \text{ (con } y = argmax_g(z)) \text{ si } x = w \\ g(x) \text{ en otro caso} \end{cases}$$

La función g' nos asegura que w obtendrá el mayor valor de fitness en la población predefinida de A_{D-I}. El índice y la frecuencia de migración pueden ser ajustados para asegurar que w emigra a una nueva población predefinida en A_{D-II}. Una vez la cadena llega a ésta nueva población esta muta a x_{yes} (introduciendo las reglas de mutación correspondientes). De esta manera la cadena es aceptada por A_{D-II}. Vemos el esquema en la figura 3.4.

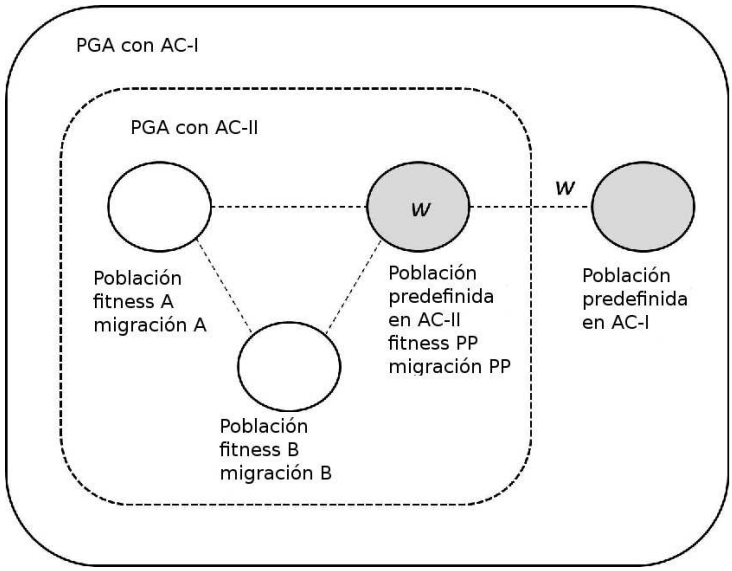

Figura 3.4: Esquema para convertir AC-I en AC-II

En cuanto a la situación inversa supongamos que D puede ser resuelto por un Algoritmo Genético Paralelo A_{D-II} con $AC-II$. Después de un

número finito de operaciones y selección de supervivientes por la función de fitness, A_{D-II} tiene una población predefinida con el individuo x_{yes} or el individuo x_{not}. Podemos definir A_{D-I} de la siguiente forma: primero, A_{D-I} simula A_{D-II}. Si el resultado es la cadena x_{not}, A_{D-I} no hace nada (de esta manera w no aparecerá nunca en la población predefinida y A_{D-I} no aceptará w). Para el caso en que el resultado de la simulación sea la cadena x_{yes} necesitamos definir la función de fitness g' como sigue:

$$g'(x) = \begin{cases} g(y) + 1 \text{ (con } y = argmax_g(z)) \text{ si } x = x_{yes} \\ g(x) \text{ en otro caso} \end{cases}$$

La función g' nos asegura que x_{yes} obtendrá el mayor valor de fitness en la población predefinida de A_{D-II}. El índice y la frecuencia de migración pueden ser ajustados para asegurar que x_{yes} emigra a una nueva población predefinida en A_{D-I}. Una vez la cadena llega a esta nueva población, esta muta a w (introduciendo las reglas de mutación correspondientes). De esta manera la cadena es aceptada por A_{D-I}. Vemos el esquema en la figura 3.5.

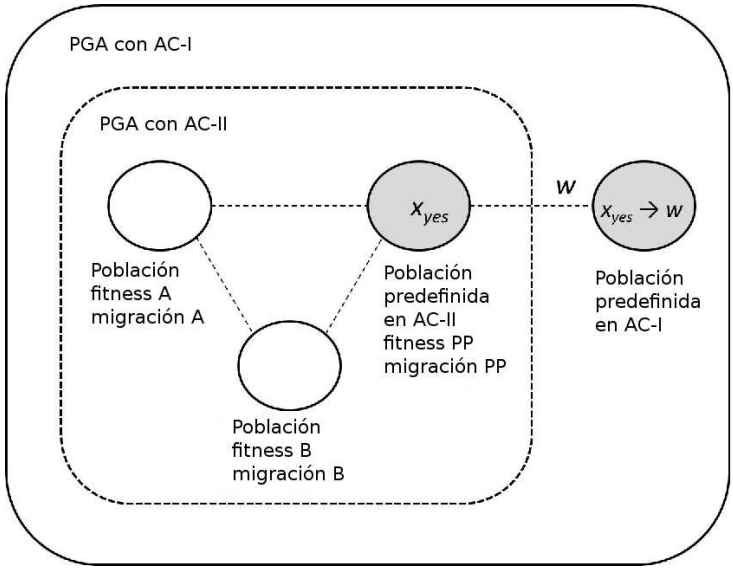

Figura 3.5: Esquema para convertir AC-II en AC-I

Una vez demostrado que los criterios $AC-I$ y $AC-II$ podemos enunciar el siguiente resultado que expone la potencia computacional de los PGA's como métodos para resolver problemas de decisión.

Teorema 4 *Los Algoritmos Genéticos Paralelos con población múltiple, síncronos y con migración completa son computacionalmente completos.*

Demostración Es suficiente considerar una ANGP como un PGA con población múltiple y con migración síncrona y completa.

3.3. Complejidad Descriptiva para la Jerarquía de Chomsky

Anteriormente hemos definido una medida temporal (computacional) para las Redes de Procesadores Genéticos, quizá esta sea la medida más importante en cuanto a complejidad, pero hay otras medidas cuyo estudio también es muy interesante y en este apartado vamos a hacer un estudio sobre la Complejidad Descriptiva. Una posible definición de la Complejidad Descriptiva podría ser la cantidad de elementos estáticos (procesadores, reglas, símbolos, etc) mínimos que le hacen falta a un modelo para resolver un problema. A continuación vamos a dar una caracterización de las familias de lenguajes de la jerarquía de Chomsky con la utilización de las Redes de Procesadores Genéticos, más concretamente su variante generadora, las GNGP. El número de procesadores necesarios para cada red nos da una idea de la Complejidad Descriptiva de cada una de las familias de lenguajes.

3.3.1. Gramáticas Regulares

Vamos a ir de menor a mayor complejidad empezando por las gramáticas regulares y vamos a ver que cualquier lenguaje regular puede ser generado por una GNGP de 3 procesadores.

Dados $L = L(G)$ y la gramática lineal por la derecha $G = (N, T, P, S)$, definimos la GNGP $R = (V, N_1, N_2, N_3, K_3, id)$, donde $V = N \cup T \cup \hat{N} \cup [TN]$, con $\hat{N} = \{\hat{A} : A \in N\}$ y $[TN] = \{[aB] : a \in T, B \in N\}$. Los procesadores de R se definen como sigue:

1. $N_1 = (M_1, \{S\}, N, FI_1, PO_1, \emptyset, 1, (2))$
   $M_1 = \{A \to [bC] : (A \to bC) \in P\} \cup \{A \to a : (A \to a) \in P\}$
   $FI_1 = [TN] \cup \hat{N}$
   $PO_1 = [TN]$

2. $N_2 = (\emptyset, A_2, PI_2, \emptyset, PO_2, \emptyset, 2, (2))$
   $A_2 = \hat{N}$
   $PI_2 = [TN]$
   $PO_2 = VV \cup (V - \hat{N})$

3. $N_3 = (M_3, \emptyset, PI_3, \emptyset, V, FO_3, 1, (2))$
   $M_3 = \{[aA] \to a : A \in N \wedge a \in T\} \cup \{\hat{A} \to A : A \in N\}$
   $PI_3 = \{[aA]\hat{A} : a \in T \wedge A \in N\}$
   $FO_3 = [TN] \cup \hat{N}$

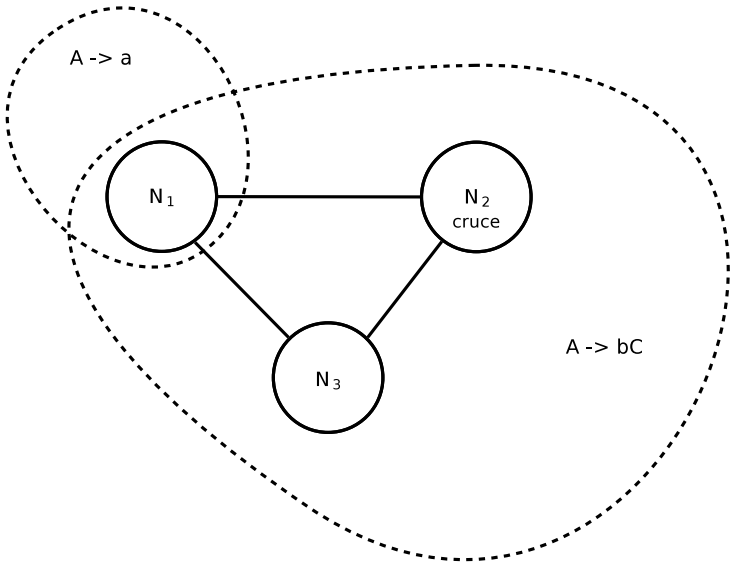

Figura 3.6: Red que simula el funcionamiento de una gramática regular.

La red simula el proceso de derivación de la gramática regular. El procesador N_1 es el encargado de recoger las cadenas de salida y aplicar las producciones de la gramática. El procesador N_2 se ocupa cuando se está aplicando una regla del tipo $A \to bC$ añadiendo al final un símbolo que en el futuro será el nuevo auxiliar (cuando aplicamos este tipo de reglas estamos añadiendo un símbolo más a la forma sentencial y es en este procesador con una operación de cruce donde se añade este símbolo). Podemos observar que por el filtro de salida de N_2 las cadenas $\hat{N}$ nunca saldrán de dicho procesador. Finalmente N_3 substituye los símbolos usados en los otros dos procesadores para que la aplicación de la regla quede concluida. Para facilitar la demostración vamos a proponer el siguiente enunciado:

$$S \overset{*}{\underset{G}{\Rightarrow}} \alpha \text{ sii existe } C_k(L_1, L_2, L_3) \text{ con } \alpha \in L_1 \cap (N \cup T)^*$$

Primero vamos a demostrar que si $S \overset{*}{\underset{G}{\Rightarrow}} \alpha$, entonces la configuración $C_k(L_1, L_2, L_3)$ aparecerá en la red, para ello vamos a utilizar un proceso de inducción sobre la gramática G para obtener la cadena α.

Caso Base

En el Caso Base vamos a realizar un paso de derivación de la gramática $S \underset{G}{\Rightarrow} \alpha$ (hemos omitido el caso trivial en que tenemos la regla $S \overset{*}{\underset{G}{\Rightarrow}} S$). Si se realiza un paso de derivación se pueden aplicar dos tipos de reglas, si $\alpha = a$ entonces $S \to a \in M_1$ y en el primer paso genético S muta en a y el enunciado se cumple. Si por el contrario $\alpha = aB$, entonces $S \to [aB] \in M_1$ y $[aB] \to a \in M_3$, siendo el proceso el siguiente: durante el primer paso genético en N_1 la cadena S muta a $[aB]$ y en el paso de comunicación esta pasa al procesador N_2. En este procesador, mediante cruce se obtiene la cadena $[aB]\hat{B}$ y se pasa al procesador N_3. Entonces, el símbolo $[aB]$ muta al símbolo a y el símbolo $\hat{B}$ muta al símbolo B en dos pasos genéticos. Finalmente la cadena aB pasa al procesador N_1 y el enunciado se cumple.

Hipótesis de Inducción

Suponemos que para cada número entero p, tal que $S \overset{p}{\underset{G}{\Rightarrow}} \alpha$, con $1 \leq p$ existe una configuración $C_k = (L_1, L_2, L_3)$ en R tal que $\alpha \in L_1$.

Paso de Inducción

Tenemos que $S \overset{p}{\underset{G}{\Rightarrow}} \beta \underset{G}{\Rightarrow} \alpha$ donde $\beta = wA$ con $w \in T^*$ y $A \in N$ y por hipótesis de inducción existe $C_k(L_1, L_2, L_3)$ con $wA \in L_1$. Ahora pueden ocurrir dos casos al pasar de wA a α: La primera posibilidad es que tengamos $A \to b \in P$ y $\alpha = wb$. En este caso la cadena wA muta a wb en el procesador N_1 dado que $A \to b \in M_1$ y el enunciado se cumple. La segunda posibilidad la tenemos cuando $A \to bC \in P$ y $\alpha = wbC$. En este caso la cadena wA muta a $w[bC]$ en el procesador N_1, puesto que $A \to [bC] \in M_1$. La cadena $w[bC]$ pasa entonces al procesador N_2 donde por cruce se obtiene la cadena $w[bC]\hat{C}$ que pasa al procesador N_3. En los dos siguiente pasos genéticos la cadena $w[bC]\hat{C}$ muta a wbC que finalmente es enviada a N_1, con lo que el enunciado se cumple.

Ahora nos queda por ver que si se da una configuración $C_k(L_1, L_2, L_3)$ y $\alpha \in L_1 \cap (N \cup T)^*$ entonces $S \overset{*}{\underset{G}{\Rightarrow}} \alpha$. Esto lo podemos demostrar por inducción de forma similar al enunciado anterior. Dado que cada cadena $w \in L(G)$ surge de $S \overset{*}{\underset{G}{\Rightarrow}} w$ por el enunciado anterior existe una configuración $C_k(L_1, L_2, L_3)$ tal que $w \in L_1$ y por tanto $w \in L(R)$.

3.3.2. Gramáticas Incontextuales

Seguimos con el siguiente tipo de gramáticas, las gramáticas incontextuales, vamos a ver que podemos generar cualquier lenguaje incontextual

con una GNGP de 4 procesadores.

Dado $L = L(G)$ y la gramática incontextual en forma normal de Chomsky $G = (N, T, P, S)$, definimos la GNGP $R = (V, N_1, N_2, N_3, N_4, K_4, id)$, donde $V = N \cup T \cup \hat{N} \cup \hat{T} \cup \bar{N} \cup \bar{T} \cup [NT] \cup [NN] \cup [TN] \cup [TT] \cup [NT]' \cup [NN]' \cup [TN]' \cup [TT]' \cup [[NN]] \cup \overline{[[NN]]} \cup [[NN]]^f$. Los procesadores de R se definen como sigue:

1. $N_1 = (M_1, \{S\}, V, FI_1, PO_1, \emptyset, 1, (2))$
   $M_1 = \{A \to [[BC]] : (A \to BC) \in P\} \cup \{A \to a : (A \to a) \in P\}$
   $FI_1 = \bar{N} \cup \bar{T} \cup \hat{N} \cup \hat{T} \cup \{[AB] : A, B \in (N \cup T)\} \cup \{[AB]' : A, B \in N\}$
   $\cup \{\overline{[[AB]]} : A, B \in N\} \cup \{[[AB]]^f : A, B \in N\}$
   $PO_1 = \{[[AB]] : A, B \in N\}$

2. $N_2 = (M_2, \emptyset, PI_2, \emptyset, V, FO_2, 1, (2))$
   $M_2 = \{A \to [BA] : A, B \in (N \cup T)\} \cup \{A \to [BA]' : A, B \in (N \cup T)\}$
   $\cup \{A \to \bar{A} : A \in (N \cup T)\} \cup \{[[AB]] \to \overline{[[AB]]} : A, B \in N\}$
   $\cup \{[[AB]] \to [[AB]]^f : A, B \in N\}$
   $PI_2 = \{[[AB]] : A, B \in N\}$
   $FO_2 = \{[[AB]] : A, B \in N\} \cup N \cup T$

3. $N_3 = (\emptyset, (\hat{N} \cup \hat{T}), PI_3, FI_3, PO_3, \emptyset, 2, (2))$
   $PI_3 = \{[AB]' : A, B \in (N \cup T)\} \cup \{[[AB]]^f : A, B \in N\}$
   $FI_3 = \{[AB][CD] : B \neq C\} \cup \{[AB][CD]' : B \neq C\} \cup \{\overline{[[AB]]}[CD] : B \neq C\}$
   $\cup \{\overline{[[AB]]}[CD]' : B \neq C\} \cup \{[AB]\overline{[[CD]]} : A, B \in (N \cup T) \wedge C, D \in N\}$
   $\cup \{[AB]'C : A, B \in (N \cup T) \wedge C \in V\} \cup \{\bar{A}[BC]' : A, B, C \in (N \cup T)\}$
   $\cup \{\bar{A}[BC] : A, B, C \in (N \cup T)\} \cup N \cup T \cup \hat{N} \cup \hat{T} \cup \{[[AB]]^f C : A, B \in N \wedge C \in V\}$
   $\cup \{[AB][[CD]]^f : A, B \in (N \cup T) \wedge C, D \in N\}$
   $PO_3 = \{AB : A, B \in V\} \cup (V - (\hat{N} \cup \hat{T}))$

4. $N_4 = (M_4, \emptyset, PI_4, \emptyset, V, FO_4, 1, (2))$
   $M_4 = \{[AB] \to A : A, B \in (N \cup T)\} \cup \{[AB]' \to A : A, B \in (N \cup T)\}$
   $\cup \{\overline{[[AB]]} \to A : A, B \in N\} \cup \{[[AB]]^f \to A : A, B \in N\}$
   $\cup \{\hat{A} \to A : A \in (N \cup T)\} \cup \{\bar{A} \to A : A \in (N \cup T)\}$
   $PI_4 = \{[[AB]]^f \hat{B} : A, B \in N\} \cup \{[AB]'\hat{B} : A, B \in (N \cup T)\}$
   $FO_4 = \{[AB] : A, B \in (N \cup T)\} \cup \{\overline{[[AB]]} : A, B \in N\} \cup \hat{N} \cup \hat{T} \cup \bar{N} \cup \bar{T}$
   $\cup \{[AB]' : A, B \in (N \cup T)\} \cup \{[[AB]]^f : A, B \in N\}$

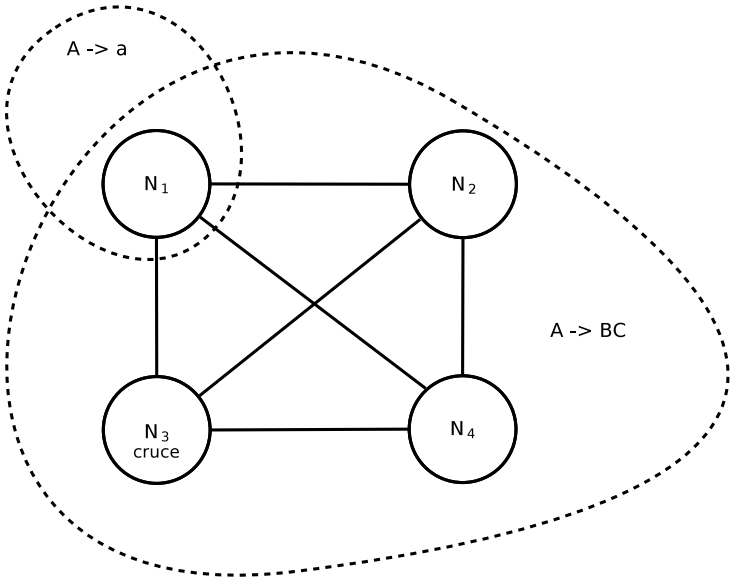

Figura 3.7: Red que simula el funcionamiento de una gramática incontextual.

La red simula el proceso de derivación de la gramática incontextual en forma normal de Chomsky. El procesador N_1 es el encargado de recoger las cadenas de salida y señalar las producciones de la gramática. Al trabajar con la forma normal de Chomsky sólo tenemos dos tipos de reglas, $A \to a$ y $A \to BC$. Las reglas del tipo $A \to a$ se simulan en el procesador N_1 en un sólo paso genético, mutando el símbolo A en a. Para las reglas del tipo $A \to BC$ utilizaremos la red entera de la siguiente manera: en N_1 mutamos el símbolo A en el símbolo $[[BC]]$ y la cadena pasa al procesador N_2. En este procesador tratamos todos los símbolos posteriores a $[[BC]]$ de modo que cada símbolo $X \in N \cup T$ muta a $[YX] : X, Y \in N \cup T$, con esto y con los filtros conseguimos que las cadenas que entran en el procesador N_3 tengan la siguiente forma $...[[\bar{B}C]][CA_1][A_1A_2][A_2A_3]...[A_{n-2}A_{n-1}][A_{n-1}A_n]'$ (el último queda marcado de forma diferente). N_3, mediante cruce añade al final de la cadena el símbolo que corresponde de tal forma que queda $...[A_{n-1}A_n]'\hat{A}_n$ y llega a N_4. Finalmente N_4 limpia la cadena transformando cada símbolo $[YX]$ en Y y la cadena limpia pasa otra vez al procesador N_1. Con todo este proceso lo que hacemos es cambiar cada símbolo (a partir del que se va a aplicar la regla) por el siguiente como si desplazáramos todos los símbolos una posición a la derecha para poder poner el nuevo símbolo (ya que al aplicar la regla de la gramática debemos substituir un único símbolo por dos), también añadimos al final el símbolo que necesitamos por cruce porque la cadena, después de aplicar la regla, debe tener un símbolo más.

3.3.3. Gramáticas Sensibles al Contexto

El siguiente tipo de gramáticas son las gramáticas sensibles al contexto. A continuación veremos que cualquier lenguaje sensible al contexto puede ser generado por una GNGP de 6 procesadores.

Dada la siguiente gramática sensible al contexto en forma normal de Kuroda $G = (N, T, P, S)$, definimos la GNGP $R = (V, N_1, N_2, N_3, N_4, N_5, N_6, K_6, id)$ con $V = N \cup T \cup \hat{N} \cup \hat{T} \cup \bar{N} \cup \bar{T} \cup [NT] \cup [NN] \cup [TN] \cup [TT] \cup [NT]' \cup [NN]' \cup [TN]' \cup [TT]' \cup [[NN]] \cup [[NN]]^f \cup \overline{[[NN]]} \cup [[NNN] \cup [NNN]]$. Los procesadores de R se definen como sigue:

1. $N_1 = (M_1, \{S\}, V, FI_1, PO_1, \emptyset, 1, (2))$
   $M_1 = \{A \to [[BC]] : (A \to BC) \in P\} \cup \{A \to a : (A \to a) \in P\}$
   $\cup \{A \to B : (A \to B) \in P\} \cup \{A \to [[ACD] : \exists B \in N \wedge (AB \to CD) \in P\}$
   $FI_1 = \bar{N} \cup \bar{T} \cup \hat{N} \cup \hat{T} \cup \{[AB] : A, B \in (N \cup T)\} \cup \{[AB]' : A, B \in N\}$
   $\cup \{[[AB]]^f : A, B \in N\} \cup \{\overline{[[AB]]} : A, B \in N\}$
   $PO_1 = \{[[AB]] : A, B \in N\} \cup \{[[ACD] : \exists B \in N \wedge (AB \to CD) \in P\}$

2. $N_2 = (M_2, \emptyset, PI_2, \emptyset, V, FO_2, 1, (2))$
   $M_2 = \{A \to [BA] : A, B \in (N \cup T)\} \cup \{A \to [BA]' : A, B \in (N \cup T)\}$
   $\cup \{A \to \bar{A} : A \in (N \cup T)\} \cup \{[[AB]] \to \overline{[[AB]]} : A, B \in N\}$
   $\cup \{[[AB]] \to [[AB]]^f : A, B \in N\}$
   $PI_2 = \{[[AB]] : A, B \in N\}$
   $FO_2 = \{[[AB]] : A, B \in N\} \cup N \cup T$

3. $N_3 = (\emptyset, (\hat{N} \cup \hat{T}), PI_3, FI_3, PO_3, \emptyset, 2, (2))$
   $PI_3 = \{[AB]' : A, B \in (N \cup T)\} \cup \{[[AB]]^f : A, B \in N\}$
   $FI_3 = \{[AB][CD] : B \neq C\} \cup \{[AB][CD]' : B \neq C\} \cup \{\overline{[[AB]]}[CD] : B \neq C\}$
   $\cup \{\overline{[[AB]]}[CD]' : B \neq C\} \cup \{[AB]\overline{[[CD]]} : A, B \in (N \cup T) \wedge C, D \in N\}$
   $\cup \{[AB]'C : A, B \in (N \cup T) \wedge C \in V\} \cup \{\bar{A}[BC]' : A, B, C \in (N \cup T)\}$
   $\cup \{\bar{A}[BC] : A, B, C \in (N \cup T)\} \cup N \cup T \cup \hat{N} \cup \hat{T} \cup \{[[AB]]^f C : A, B \in N \wedge C \in V\}$
   $\cup \{[AB][[CD]]^f : A, B \in (N \cup T) \wedge C, D \in N\}$
   $PO_3 = \{AB : A, B \in V\} \cup (V - (\hat{N} \cup \hat{T}))$

4. $N_4 = (M_4, \emptyset, PI_4, \emptyset, V, FO_4, 1, (2))$
   $M_4 = \{\underline{[AB]} \to A : A, B \in (N \cup T)\} \cup \{[AB]' \to A : A, B \in (N \cup T)\}$
   $\cup \{\overline{[[AB]]} \to A : A, B \in N\} \cup \{[[AB]]^f \to A : A, B \in N\}$
   $\cup \{\hat{A} \to A : A \in (N \cup T)\} \cup \{\bar{A} \to A : A \in (N \cup T)\}$
   $PI_4 = \{[[AB]]^f \hat{B} : A, B \in N\} \cup \{\underline{[AB]'\hat{B}} : A, B \in (N \cup T)\}$
   $FO_4 = \{[AB] : A, B \in (N \cup T)\} \cup \{\overline{[[AB]]} : A, B \in N\} \cup \hat{N} \cup \hat{T} \cup \bar{N} \cup \bar{T}$
   $\cup \{[AB]' : A, B \in (N \cup T)\} \cup \{[[AB]]^f : A, B \in N\}$

5. $N_5 = (M_5, \emptyset, PI_5, \emptyset, V, \emptyset, 1, (2))$
   $M_5 = \{B \to [BCD]] : \exists A \in N \wedge (AB \to CD) \in P\}$
   $PI_5 = \{[[ACD]B : (AB \to CD) \in P\}$

6. $N_6 = (M_5, \emptyset, PI_6, \emptyset, V, FO_6, 1, (2))$
   $M_6 = \{[[ACD] \to C : \exists B \in N \wedge (AB \to CD) \in P\}$
   $\cup \{[BCD]] \to D : \exists A \in N \wedge (AB \to CD) \in P\}$
   $PI_6 = \{[[ACD][BCD]] : (AB \to CD) \in P\}$
   $FO_6 = \{[[ACD] : A, C, D \in N\} \cup \{[BCD]] : B, C, D \in N\}$

En la red vamos a simular el proceso de derivación de una gramática sensible al contexto en forma normal de Kuroda. El procesador N_1 es el encargado de recoger las cadenas de salida y marcar donde aplicar las producciones de la gramática. Cuando se trabaja con una gramática en forma normal de Kuroda se tienen cuatro tipos de reglas diferentes, $A \to a$, $A \to B$, $A \to BC$ y $AB \to CD$. Los dos primeros tipos de reglas, $A \to a$ y $A \to B$ se aplicarán íntegramente en el procesador N_1, mediante la aplicación de una regla de mutación. De las reglas del tipo $A \to BC$ se encargan los procesadores N_1, N_2, N_3 y N_4 y el proceso es el mismo que con las gramáticas incontextuales. Finalmente, los procesadores N_1, N_5 y N_6 son los encargados de las reglas del tipo $AB \to CD$ y lo hacen de la siguiente forma: primero en el procesador N_1 se muta un símbolo A en $[[ACD]$. En N_5 sólo entraran cadenas con el segmento $[[ACD]B$ (así controlamos que los símbolos AB estaban juntos) y la B muta a $[BCD]]$. N_6 recibe cadenas con el segmento $[[ACD][BCD]]$, con lo que sólo nos queda mutar el símbolo $[[ACD]$ por C y $[BCD]]$ por D. Finalmente, la cadena ya con la regla aplicada vuelve al procesador N_1.

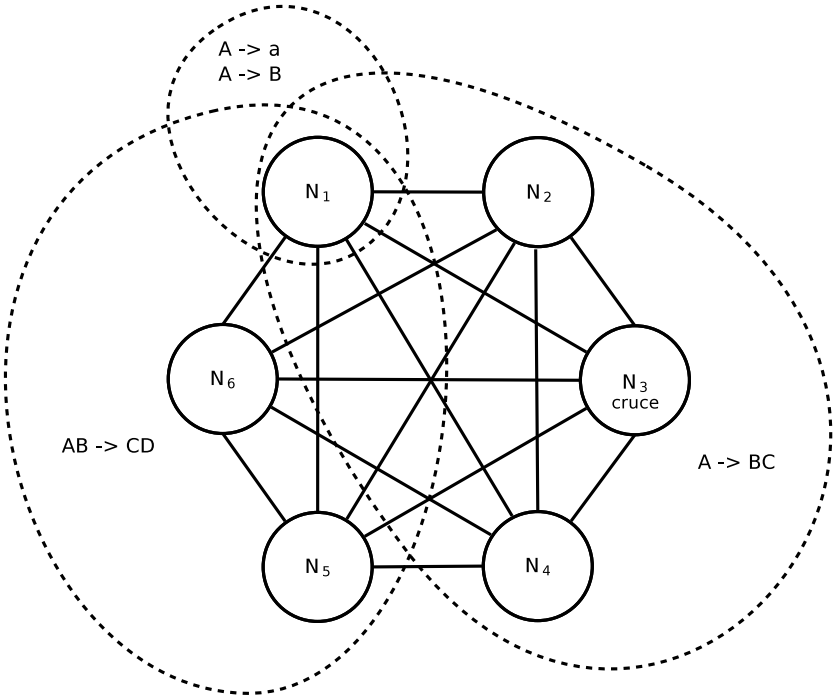

Figura 3.8: Red que simula el funcionamiento de una gramática sensible al contexto.

3.3.4. Gramáticas No Restringidas

Finalmente terminamos la jerarquía de Chomsky con las gramáticas no restringidas, seguidamente vamos a ver que se puede generar cualquier lenguaje recursivamente enumerable con una GNGP de 8 procesadores.

Dada la siguiente gramática sin restricciones en forma normal $G = (N, T, P, S)$, definimos la GNGP $R = (V,\ N_1,\ N_2,\ N_3,\ N_4,\ N_5,\ N_6,\ N_7,\ N_8,\ K_8,\ id)$, con $V = N \cup T \cup \hat{N} \cup \hat{T} \cup \bar{N} \cup \bar{T} \cup \overline{[NT]} \cup [NN] \cup [TN] \cup [TT] \cup [NT]' \cup [NN]' \cup [TN]' \cup [TT]' \cup [[NN]] \cup \overline{[[NN]]} \cup [[NNN] \cup [NNN]] \cup \widetilde{N} \cup \widetilde{T} \cup < NN > \cup < NT > \cup < TN > \cup < TT > \cup < NX > \underline{\cup < TX >} \cup \overline{< NN >} \cup \overline{< NT >} \cup \overline{< TN >} \cup \overline{< TT >} \cup \overline{<< NN >>} \cup \overline{\overline{<< NN >>}}$, con $X \notin (T \cup N)$. Los procesadores de R se definen como sigue:

1. $N_1 = (M_1, \{S\}, V, FI_1, PO_1, \emptyset, 1, (2))$
   $M_1 = \{A \to [[BC]] : (A \to BC) \in P\} \cup \{A \to a : (A \to a) \in P\}$
   $\cup \{A \to B : (A \to B) \in P\} \cup \{A \to [[ACD] : \exists B \in N \wedge (AB \to CD) \in P\}$
   $\cup \{A \to << AB >> : (AB \to B) \in P\}$
   $FI_1 = \bar{N} \cup \bar{T} \cup \hat{N} \cup \hat{T} \cup \{[AB] : A, B \in (N \cup T)\} \cup \{[AB]' : A, B \in N\}$
   $\cup \{[[AB]]^f : A, B \in N\} \cup \{\overline{[[AB]]} : A, B \in N\} \cup \widetilde{N} \cup \widetilde{T}$

$\cup \{<< AB >>: A, B \in N\} \cup \{< AB >: A, B \in (N \cup T \cup \{X\})\}$
$\cup \{\overline{< AB >} : A, B \in (N \cup T)\}$
$PO_1 = \{[[AB]] : A, B \in N\} \cup \{[[ACD] : \exists B \in N \wedge (AB \to CD) \in P\}$
$\cup \{<< AB >>: A, B \in N\}$

2. $N_2 = (M_2, \emptyset, PI_2, \emptyset, V, FO_2, 1, (2))$
$M_2 = \{A \to [BA] : A, B \in (N \cup T)\} \cup \{A \to \underline{[BA]}' : A, B \in (N \cup T)\}$
$\cup \{A \to \bar{A} : A \in (N \cup T)\} \cup \{[[AB]] \to \overline{[[AB]]} : A, B \in N\}$
$\cup \{[[AB]] \to [[AB]]^f : A, B \in N\}$
$PI_2 = \{[[AB]] : A, B \in N\}$
$FO_2 = \{[[AB]] : A, B \in N\} \cup N \cup T$

3. $N_3 = (\emptyset, (\hat{N} \cup \hat{T}), PI_3, FI_3, PO_3, \emptyset, 2, (2))$
$PI_3 = \{[AB]' : A, B \in (N \cup T)\} \cup \{[[AB]]^f : A, B \in N\}$
$FI_3 = \{[AB][CD] : B \neq C\} \cup \{[AB][CD]' : B \neq C\} \cup \{\overline{[[AB]]}[CD] : B \neq C\}$
$\cup \{\overline{[[AB]]}[CD]' : B \neq C\} \cup \{[AB]\overline{[[CD]]} : A, B \in (N \cup T) \wedge C, D \in N\}$
$\cup \{[AB]'C : A, B \in (N \cup T) \wedge C \in V\} \cup \{\bar{A}[BC]' : A, B, C \in (N \cup T)\}$
$\cup \{\bar{A}[BC] : A, B, C \in (N \cup T)\} \cup N \cup T \cup \hat{N} \cup \hat{T} \cup \{[[AB]]^f C : A, B \in N \wedge C \in V\}$
$\cup \{[AB][[CD]]^f : A, B \in (N \cup T) \wedge C, D \in N\}$
$PO_3 = \{AB : A, B \in V\} \cup (V - (\hat{N} \cup \hat{T}))$

4. $N_4 = (M_4, \emptyset, PI_4, FI_4, V, FO_4, 1, (2))$
$M_4 = \{\underline{[AB]} \to A : A, B \in (N \cup T)\} \cup \{[AB]' \to A : A, B \in (N \cup T)\}$
$\cup \{\overline{[[AB]]} \to A : A, B \in N\} \cup \{[[AB]]^f \to A : A, B \in N\}$
$\cup \{\hat{A} \to A : A \in (N \cup T)\} \cup \{\bar{A} \to A : A \in (N \cup T)\}$
$\cup \{\widetilde{A} \to A : A \in (N \cup T)\} \cup \{\overline{<< AB >>} \to B : A, B \in N\}$
$\cup \{\overline{\overline{<< AB >>}} \to B : A, B \in N\} \cup \{\overline{< AB >} \to B : A, B \in (N \cup T)\}$
$\cup \{< AB > \to B : A, B \in (N \cup T)\}$
$PI_4 = \{[[AB]]^f \hat{B} : A, B \in N\} \cup \{[AB]'\hat{B} : A, B \in (N \cup T)\}$
$\cup \{\overline{< AB >} : A, B \in N\} \cup \{\overline{\overline{<< AB >>}} : A, B \in N\}$
$FI_4 = \{\overline{< AB >}C : A, B \in N \wedge C \in V\} \cup \{\overline{\overline{<< AB >>}}C : A, B \in N \wedge C \in V\}$
$FO_4 = \{[AB] : A, B \in (N \cup T)\} \cup \{\overline{[[AB]]} : A, B \in N\} \cup \hat{N} \cup \hat{T} \cup \bar{N} \cup$

$\bar{T} \cup \widetilde{N} \cup \widetilde{T}$
$\cup \{[AB]' : A, B \in (N \cup T)\} \cup \{[[AB]]^f : A, B \in N\}$
$\cup \{< AB > A, B \in (N \cup T)\} \cup \{\overline{< AB >} : A, B \in (N \cup T)\}$
$\cup \{\overline{<< AB >>} : A, B \in N\} \cup \{\overline{\overline{<< AB >>}} : A, B \in N\}$

5. $N_5 = (M_5, \emptyset, PI_5, \emptyset, V, \emptyset, 1, (2))$
   $M_5 = \{B \to [BCD]] : \exists A \in N \wedge (AB \to CD) \in P\}$
   $PI_5 = \{[[ACD]B : (AB \to CD) \in P\}$

6. $N_6 = (M_5, \emptyset, PI_6, \emptyset, V, FO_6, 1, (2))$
   $M_6 = \{[[ACD] \to C : \exists B \in N \wedge (AB \to CD) \in P\}$
   $\cup \{[BCD]] \to D : \exists A \in N \wedge (AB \to CD) \in P\}$
   $PI_6 = \{[[ACD][BCD]] : (AB \to CD) \in P\}$
   $FO_6 = \{[[ACD] : A, C, D \in N\} \cup \{[BCD]] : B, C, D \in N\}$

7. $N_7 = (M_7, \emptyset, PI_7, \emptyset, V, FO_7, 1, (2))$
   $M_7 = \{A \to \widetilde{A} : A \in (N \cup T)\} \cup \{A \to < AB >: A, B \in (N \cup T)\}$
   $\cup \{A \to \overline{< AB >} : A, B \in (N \cup T)\} \cup \{A \to < AX >: A \in (N \cup T)\}$
   $\cup \{<< AB >> \to \overline{<< AB >>} : A, B \in N\}$
   $\cup \{<< AB >> \to \overline{\overline{<< AB >>}} : A, B \in N\}$
   $PI_7 = \{<< AB >> B : (AB \to B) \in P\}$
   $FO_7 = \{<< AB >>: A, B \in N\} \cup N \cup T$

8. $N_8 = (\emptyset, \emptyset, PI_8, FI_8, V, \emptyset, 2, (2))$
   $PI_8 = \{\overline{<< AB >>} : A, B \in N\} \cup \{\overline{\overline{<< AB >>}} : A, B \in N\}$
   $FI_8 = \{\overline{<< AB >>}\widetilde{C} : A, B \in N \wedge C \in (N \cup T)\}$
   $\cup \{< AX > B : A \in (N \cup T) \wedge B \in V\} \cup \{< AB > \widetilde{C} : A, B, C \in (N \cup T)\}$
   $\cup \{\widetilde{C} < AB >: A, C \in (N \cup T) \wedge B \in (N \cup T \cup \{X\})\}$
   $\cup \{\overline{< AB >}\widetilde{C} : A, B, C \in (N \cup T)\} \cup \{\widetilde{C}\overline{< AB >} : A, B, C \in (N \cup T)\}$
   $\cup \{\overline{< AB >} < CD >: A, B, C, D \in (N \cup T)\}$
   $\cup \{\overline{< AB >< CD >} : A, B, C, D \in (N \cup T)\}$
   $\cup \{\overline{< AB >}<< CD >> : A, B \in (N \cup T) \wedge C, D \in N\}$

$$\cup \{\overline{<< AB >>} < CD >: B \neq C\} \cup \{\overline{<< AB >>< CD >} : B \neq C\}$$
$$\cup \{\overline{< AB >} < CX >: B \neq C\} \cup \{< AB >< CD >: B \neq C\}$$
$$\cup \{< AB > \overline{< CD >} : B \neq C\}$$
$$\cup \{\overline{<< AB >>} < CX >: A, B \in N \wedge C \in (N \cup T)\}$$
$$\cup \{\overline{\overline{<< AB >>}} CD : A, B \in N \wedge C, D \in V\}$$
$$\cup \{\overline{\overline{<< AB >>}} \widetilde{C} : A, B \in N \wedge C \in (N \cup T)\}$$
$$\cup \{\overline{\overline{<< AB >>}} < CD >: A, B \in N \wedge C, D \in (N \cup T)\}$$
$$\cup \{< CD > \overline{\overline{<< AB >>}} : A, B \in N \wedge C, D \in (N \cup T)\}$$
$$\cup \{\overline{\overline{<< AB >>}< CD >} : A, B \in N \wedge C, D \in (N \cup T)\}$$
$$\cup \{< CD >\overline{\overline{<< AB >>}} : A, B \in N \wedge C, D \in (N \cup T)\}$$

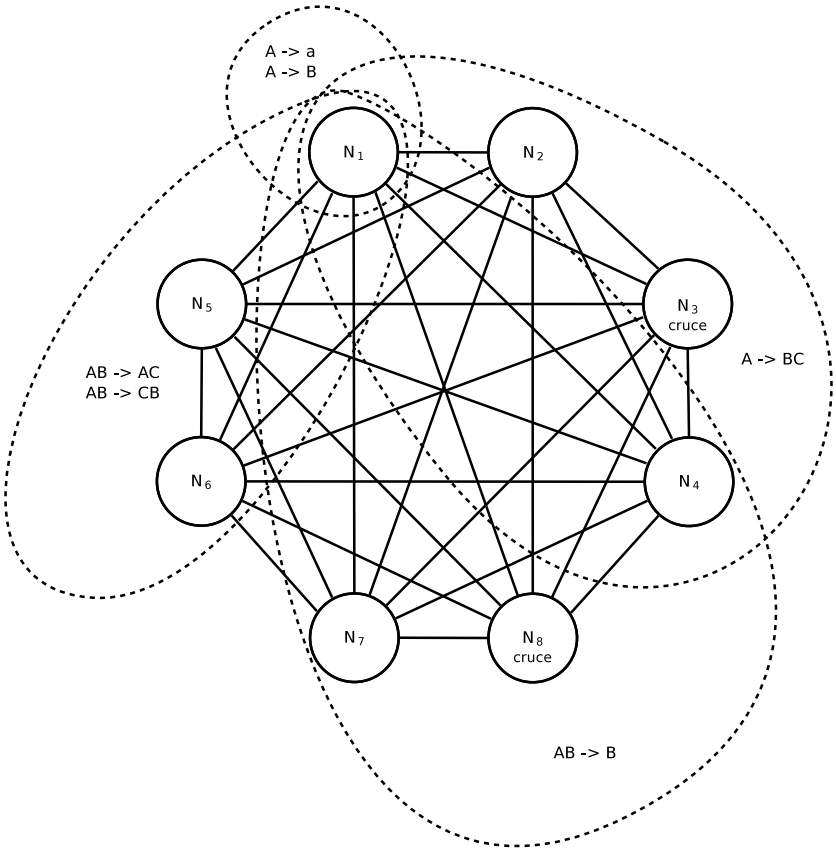

Figura 3.9: Red que simula el funcionamiento de una gramática sin restricciones.

En esta red vamos a simular el proceso de derivación de una gramáti-

ca sin restricciones en forma normal. El procesador N_1 es el encargado de recoger las cadenas de salida y marcar qué producciones y dónde aplicarlas en la gramática. Esta vez, el tipo de reglas que nos podemos encontrar son $A \to a$, $A \to B$, $A \to BC$, $AB \to AC$, $AB \to CB$ y $AB \to B$. Al igual que en la simulación anterior $A \to a$ y $A \to B$ se aplicarán íntegramente en el procesador N_1, mediante la aplicación de una regla de mutación. Las reglas del tipo $A \to BC$ son tratadas por los procesadores N_1, N_2, N_3 y N_4, de la misma forma que en las gramáticas incontextuales o las sensibles al contexto. Las reglas del tipo $AB \to AC$ y $AB \to CB$ son un subconjunto de las reglas $AB \to CD$ utilizadas en gramáticas sensibles al contexto, así que los procesadores N_1, N_5 y N_6 las aplicarán del mismo modo que en sensibles al contexto. Sólo nos quedan las reglas del tipo $AB \to B$ que se simulan de la siguiente manera: el procesador N_1 marca el símbolo A mutándolo a $<< AB >>$. El procesador N_7 sólo acepta cadenas con el segmento $<< AB >> B$ (para controlar que el segmento al que se le aplica la regla es el correcto) y lo que hace es mutar cada símbolo F después de $<< AB >>$ a $< FY >$ (si F fuera el último símbolo lo mutaría a $< FX >$ donde $X \notin N$). La cadena, cuando entra a N_8 y debido a los filtros, tiene el siguiente aspecto: $...\overline{<< AB >>} < BA_1 >< A_1A_2 >< A_2A_3 > ...\overline{< A_{n-1}A_n >} < A_nX >'$ (Hay un caso que se trata aparte, que el símbolo a eliminar sea el penúltimo, pero el proceso es parecido). En este procesador se le aplica una operación de cruce que sirve para eliminar el último símbolo, N_4 es el encargado de recoger y limpiar el resultado mutando los símbolo de la forma $< FY >$ ($\overline{< FY >}$, $\overline{<< FY >>}$ o $\overline{\overline{<< FY >>}}$) a Y. Finalmente, la cadena ya con la regla aplicada vuelve al procesador N_1. Con todo este proceso podemos decir que lo que hacemos es cambiar cada símbolo por el anterior (hasta el que queremos borrar) y por cruce eliminamos el último símbolo que es el que nos sobra.

Capítulo 4

Aplicaciones

Hasta ahora hemos definido un nuevo modelo de computación, las Redes de Procesadores Genéticos, hemos demostrado que se trata de un modelo completo y hemos definido varios tipos de redes para poder trabajar con diferentes tipos de problemas. Llegados a este punto, queremos saber cómo funciona este modelo con problemas reales. Para ello necesitamos un simulador que nos permita ver el comportamiento del modelo cuando lo utilizamos para resolver dichos problemas. En este apartado tratamos la creación e implementación de un simulador de Redes de Procesadores Bioinspirados (definidos en la sección 2.3).

Muchos de los diferentes modelos de Redes de Procesadores Bioinspirados comparten una misma estructura, lo que varía son las operaciones que puede realizar cada procesador sobre el conjunto de cadenas que contiene en cada momento, ejemplo de ello es que las NEP usan operadores evolutivos, las redes de Procesadores de Splicing utilizan operaciones de splicing y las NGP utilizan mutaciones y cruces. A la hora de construir un simulador esta característica nos permite programar la estructura de procesadores, filtros, multiconjuntos, etc y tener una batería de diferentes tipos de reglas para simular diferentes tipos de Redes de Procesadores Bioinspirados.

4.1. Implementación de un simulador de Redes de Procesadores Bioinspirado

En este apartado vamos a proponer la implementación de un procesador bioinspirado (en un principio evolutivo, pero nos sirve para todo tipo de redes bioinspiradas) en Java [32]. La implementación está constituida por la definición de un conjunto de interfaces que van a marcar el comportamiento y funcionalidad de las diferentes clases. Al tener como objetivo conseguir un

simulador para todo tipo de redes bioinspiradas, la definición por interfaces nos da la ventaja de poder definir diferentes clases para cada interface dependiendo de la funcionalidad (o tipo de red) que necesitemos. El ejemplo más claro de lo que acabamos de comentar es la interface *IFilter*, que dependiendo del filtro que necesitemos podemos tener diferentes clases, *filterLOW* para filtro de palabras, *filterLOS* para filtro de segmentos, *filterAut* para tener un autómata como filtro, etc.

Las dependencias entre clases y interfaces están representadas en la figura 4.1, en el cual se muestra un diagrama de clases UML.

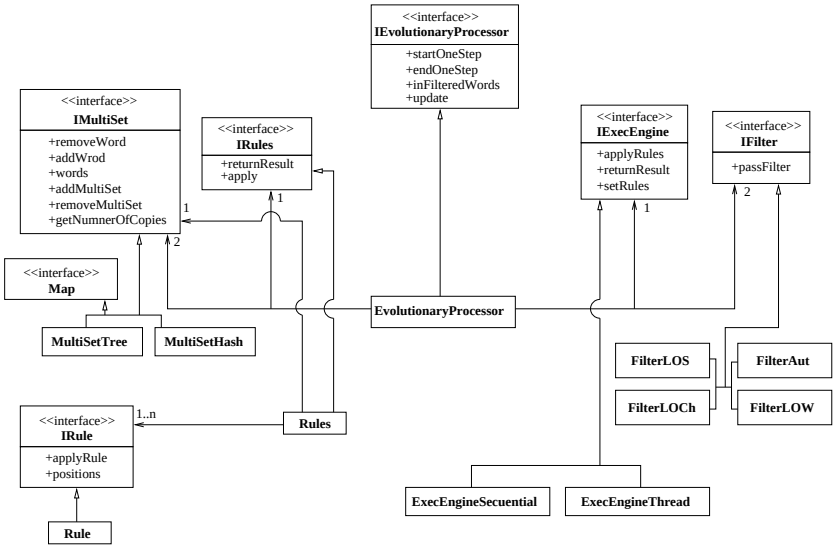

Figura 4.1: Diagrama de clases UML de un procesador bioinspirado.

En la figura 4.1 podemos ver que la clase principal es *EvolutionaryProcessor*, que implementa la interface *IEvolutionaryProcessor*. Esta clase tiene definidos todos sus componentes mediante un conjunto de interfaces. En el diagrama podemos ver un conjunto de reglas, dos filtros (el de entrada y el de salida), un multiconjunto de palabras y un motor de ejecución.

Las reglas dependen del tipo de red que queramos simular, cuando trabajemos con NEP's tendremos reglas de inserción, substitución y borrado, pero cuando trabajemos con NGP's sólo tendremos substituciones (pues el cruce no necesita ninguna información adicional, sólo las cadenas sobre las que queremos operar). Tenemos filtros de muchos tipos, cuando trabajamos con problemas de decisión se suelen utilizar filtros de palabras (LOW), de segmentos (LOS) o de símbolos ($LOCh$) y cuando trabajamos con problemas de optimización los filtros suelen implementar las restricciones del problema y la función de fitness. También nos pareció interesante definir un filtro

compuesto por un autómata (Aut). Para los multiconjuntos tenemos una interface ya implementada en Java, $Map<K,V>$, de la que usamos dos clases, una constituida por un árbol Red-Black y otra con tablas Hash.

Finalmente comentar que hemos realizado dos implementaciones de la simulación, una donde todos los procesadores hacen sus pasos de computación y comunicación de forma secuencial y otro en el que cada procesador trabaja con un hilo de ejecución.

La operatividad de un procesador se puede separar en dos fases. Primero realizamos una fase de cálculo (paso de ejecución) que empieza con la llamada del método *startOneStep* en el procesador. Este método llama al método *applyRules* que aplica el conjunto de reglas al multiconjunto que se le pasa previamente. Las reglas se aplican sobre las palabras del multiconjunto (dependiendo el tipo de red bioinspirada que estemos simulando) y los resultados se guardan. Cuando llamamos al método *EndOneStep*, éste coge las palabras que se han guardado y las separa en dos conjuntos, las que pasan el filtro de salida y las que no. Podemos ver la operatividad de esta fase en un procesador evolutivo en la figura 4.2.

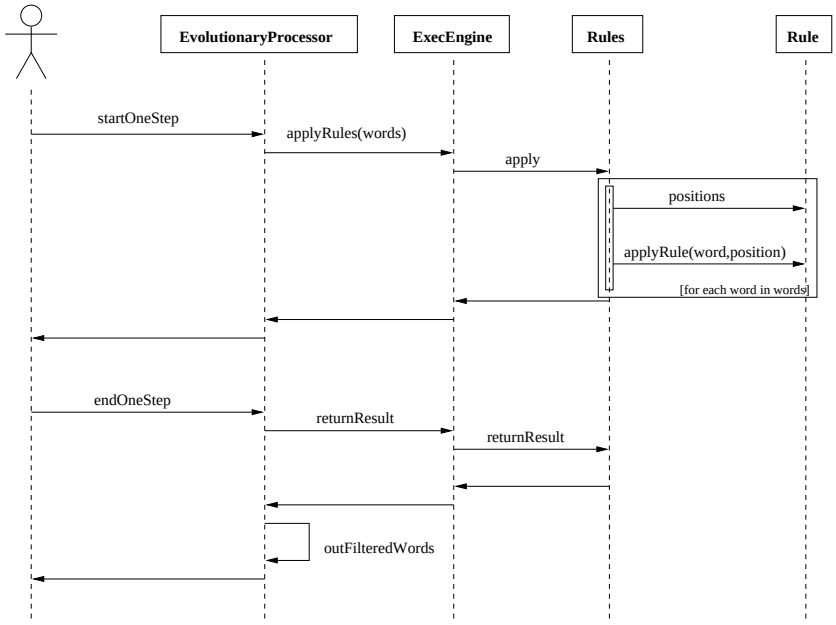

Figura 4.2: Operatividad en el procesador evolutivo.

La segunda fase es la fase de comunicación. Después de ejecutar *endOneStep*, en cada procesador, tenemos almacenadas las palabras que han pasado el filtro de salida y que van a ser repartidas por los procesadores a los que esté conectado. El método *inFilteredWords* repasa estas palabras y descarta

aquellas que no pasen el filtro de entrada, el nuevo conjunto de palabras, junto con las que se han quedado en el procesador en la fase anterior por no pasar el filtro de salida se guardan en un buffer. Finalmente el método *update* actualiza el estado del procesador pasando todas las palabras del buffer al multiconjunto propio del procesador.

Interfaces y código fuente

La siguiente interfaz sólo la usamos para NEP's y es la que se encarga de manejar una regla evolutiva (inserción, borrado o substitución) de un procesador:

```
public interface IRule {

     Vector<String> applyRule(Vector<String> s, int pos);
     int[] positions(Vector<String> s);
     String getType();
}
```

El método principal es *applyRule* que, al ser ejecutado, escoge una cadena y una posición en donde puede ser aplicada la regla y nos devuelve una nueva palabra resultado de aplicar la regla a la palabra en la posición especificada. Las posiciones donde puede ser aplicada la regla es lo que nos devolverá el método *positions* en un array al pasarle como parámetro la cadena en cuestión. El método *getType* nos sirve para obtener información sobre el tipo de regla que estamos usando.

La clase *RuleGram* implementa esta interface. Como estamos trabajando con reglas evolutivas, necesitamos guardar lo que es la regla, para ello la clase posee dos cadenas, una para la cabeza y otra para el cuerpo ($a \to b$). Cuando queramos representar una regla de inserción pondremos la cabeza a vacío ($\lambda \to b$) y cuando queramos representar una regla de borrado, pondremos el cuerpo a vacío ($a \to \lambda$).

La interface encargada de representar todas las reglas de un procesador es la siguiente:

```
public interface IRules {

     void apply(IMultiset m);
     IMultiset returnResult();
     String getXML();
```

```
     String getType();
}
```

El método principal es *apply*, que básicamente aplica el conjunto de reglas al multiconjunto de palabras. Este método es realmente complejo, pues tiene en cuenta las dos posibilidades que se pueden dar con el multiconjunto: que sus cadenas tengan multiplicidad infinita, o que sus cadenas tengan multiplicidad finita. En el primer caso el método selecciona cada una de las cadena y le aplica todas las reglas que se puedan en todas las posiciones posibles, añadiendo todas las soluciones que vayan saliendo en un nuevo multiconjunto en el cual todas las cadenas también tendrán multiplicidad infinita. Si nos encontramos con la segunda opción, es decir, para cada cadena tenemos un número n diferente de copias, lo que hacemos es aplicar n reglas al azar y guardamos cada uno de los resultados con multiplicidad 1 en un nuevo multiconjunto (si se repiten cadenas en vez de poner cadenas iguales se incrementa el número de copias de la cadena ya existente). El método *returnResult* sirve para devolver el multiconjunto donde se ha almacenado el resultado de un paso de ejecución. Los métodos *getXML* y *getType* son usados para obtener información de la clase, el primero nos devuelve un archivo de texto en formato XML con la información del conjunto de reglas y el segundo método nos devuelve el tipo de reglas (evolutivas, genéticas...).

A continuación presentamos la interface que representa los filtros tanto de entrada como de salida:

```
public interface IFilter {
     boolean passFilter(Vector<String> s);
     void wordsPassFilter(IMultiSet words,
                          IMultiSet wYes,IMultiSet wNo);
     String getXML();

     String getType();
}
```

En este interface, los métodos principales son *passFilter* y *wordsPassFilter*. Usaremos *passFilter* cuando estemos trabajando con problemas de decisión, este método es el encargado de decir si una palabra pasa o no el filtro. Usaremos *wordsPassFilter* cuando trabajemos con problemas de optimización, la clase (dependiendo del tipo de filtro) tendrá un número máximo de población, este método nos reparte un multiconjunto en dos, uno de palabras que pasan el filtro y otro de palabras que no lo pasan, para ello tiene

en cuenta las restricciones del problema y que sólo pueden pasar un cierto número de cadenas (para evaluarlas tendremos una función de fitness). El método *getXML* nos devuelve una cadena que representa el filtro en XML y el método *getType* nos indica con qué tipo de filtro estamos trabajando.

En esta interface hemos hecho muchas implementaciones diferentes, un ejemplo es la clase *FilterLOW*, que implementa filtros de palabras mediante una lista de cadenas permitidas. Una palabra pasará el filtro si se encuentra en la lista de cadenas permitidas y no lo pasará en caso contrario. Otras implementaciones son *filterLOS* para filtros de segmentos, *filterLOCh* para filtros de símbolos, *filterAut* en donde para pasar el filtro una palabra tiene que ser aceptada por un autómata, es decir un lenguaje regular, o algunos filtros específicos para problemas de optimización.

Pasemos ahora a ver la interface encargada de los multiconjuntos:

```
import java.util.Iterator;
import java.util.Vector;
public interface IMultiSet {

     void addMultiSet(IMultiSet m);
     int getNumberOfCopies(Vector<String> word);
     void addWord(Vector<String> word,  Integer copies);
     void removeWord(Vector<String> word);
     Iterator<Vector<String>> words();
     void removeMultiSet(IMultiSet ms);

     String getType();
     int size();
     String getXML();
}
```

Un multiconjunto es un conjunto de palabras con multiplicidad variable, en esta interface ponemos la herramientas básicas para poder trabajar con un multiconjunto. Primero debemos comentar que la posibilidad de tener infinitas copias de una cadena se indica poniendo un valor igual a -1 en el número de copias, en otro caso dicho número representa el número de copias que contiene. *addWord* añade n copias de la cadena dada, si la cadena ya estaba en el multiconjunto el resultado es que tenemos un número de copias igual al anterior más n. *removeWord* borra todas las copias de una cadena en el multiconjunto. *addMultiset* añade el conjunto de cadenas de un multiconjunto dado con el multiconjunto que estamos trabajando y *removeMultiset*

borra todas las copias de todas las cadenas de un multiconjunto dado en el multiconjunto actual. Para trabajar más cómodamente con el multiconjunto tenemos el método *words*, con el que obtenemos un iterador con cada una de las palabras del multiconjunto (para cuando sólo nos interese trabajar con las palabras) y *getNumberOfCopies* que, dada una palabra, nos devuelve la cantidad de copias que hay en el multiconjunto. Finalmente tenemos el método *getType* que nos devuelve el tipo de multiconjunto utilizado, el método *getXML* que nos devuelve un XML con la descripción y contenido del multiconjunto y el método *size*, que nos indica el número de cadenas que contiene el multiconjunto.

Para esta interface tenemos dos posibles implementaciones, *MultiSetHash* y *MultiSetTree*, la diferencia está en el tipo de estructura interna utilizada. En la primera se usan tablas Hash y en la segunda Árboles Red-Black. Usamos estas estructuras para almacenar las cadenas (y que las propias cadenas sean la clave) y su número de copias.

La interface que tenemos a continuación representa el motor de ejecución de cada procesador:

```
public interface IExecEngine {
     void applyRules(IMultiSet m);
     IMultiSet returnResult();
     void setRules(IRules r);
}
```

El motor de ejecución es un elemento crítico de la implementación, ya que la eficiencia de esta interface es, en mayor medida, la que determina el tiempo de ejecución del simulador, con lo que tenemos que tener especial cuidado con su implementación. Antes de hacer cualquier ejecución necesitamos asignar el conjunto de reglas que va a utilizar el procesador, para ello usamos el método *setRules*. Para realizar el proceso de aplicar la reglas al multiconjunto tenemos el método *applyRules* y para finalizar el método *returnResult* que es el encargado de devolver el resultado.

Hemos creado dos clases que implementan esta interfaz, *ExecEngineSequential* y *ExecEngineConcurrent*, la primera clase está creada para ejecutar el simulador con un hilo, ideal para cuando utilicemos un ordenador monoprocesador. La segunda clase lanza cada ejecución de cada procesador en un hilo diferente de ejecución, con lo cual podemos aprovecharnos de ordenadores con varios procesadores o para lanzar el simulador en paralelo en varios ordenadores.

La siguiente interface representa un procesador de la red:

```
public interface IProcessor {
     void startOneStep();
     IMultiSet endOneStep();

     void inFilteredWords(IMultiSet m);
     void update();
     Object info(String t);
     IMultiSet getState();
     IRules getRules();
     IFilter getOutputFilter();

     IFilter getInputFilter();
     String getId();
     String getEngine();
}
```

Esta es la interface principal de un procesador que nos define la operatividad de sus aplicaciones. En la interface, cada paso de ejecución está implementado por dos métodos, *startOneStep* que se encarga de hacer la llamada a *applyRules* del motor de ejecución pasándole el multiconjunto de palabras y *endOneStep* que se encarga de llamar a *returnResult* del motor de ejecución el cual le devuelve el multiconjunto resultado, después de esto, el método *endOneStep* comprueba que palabras del último multiconjunto recibido pasan el filtro de salida, devolviendo un nuevo multiconjunto con éstas y guardando el resto en un multiconjunto auxiliar. *inFilteredWords* recibe un multiconjunto con las palabras que le llegan de los otros procesadores (a los que esté conectados) y mira qué cadenas pasan el filtro de entrada, guardándolas en un multiconjunto auxiliar. Llegados a este punto tenemos las cadenas resultado de aplicar un paso de ejecución y un paso de comunicación al multiconjunto del procesador en un multiconjunto auxiliar, el método *update* es el encargado de actualizar es el estado del procesador y que el multiconjunto auxiliar pase a ser el propio del procesador. Para resumir, el paso de ejecución lo realizan los métodos *startOneStep* y *endOneStep*, y el paso de comunicación lo realizan *endOneStep*, luego *update* actualiza el estado del procesador. El resto de métodos son para obtener información del procesador, con el método *info* obtenemos toda la información del procesador, o podemos pedir información más especifica con los métodos *getState*, *getRules*, *getOutputFilter*, *getInputFilter*, *getId* y *getEngine*.

Para el almacenamiento de la descripción de un procesador y sus elementos hemos optado por utilizar un formato XML, la razón es que es un sistema compatible con muchas plataformas y con un alto grado de capacidad de descripción. Otra ventaja es que al ser estándar hay gran cantidad de intérpretes y los diferentes elementos y estructuras son fáciles de definir. Con esto sólo necesitamos pasarle al simulador los XML necesarios para la ejecución de una red. Hemos definido archivos XML para el procesador y cada uno de sus elementos.

Para el criterio de parada el modelo teórico tiene definidas varias posibilidades, que se repitan dos configuraciones seguidas en la ejecución o que en las redes aceptoras entre una cadena en el nodo de salida. En el simulador hemos implementado dos, la primera es la ejecución de un número concreto de pasos, es decir, realizar una simulación de n pasos. El segundo consiste en que la simulación se detiene si no se puede aplicar ninguna regla en ningún procesador, aunque a priori parezca una condición un poco pobre, tiene dos ventajas; la primera es que nos viene muy bien para simulaciones que no hemos definido bien y en las que, por ejemplo, perdemos todas las cadenas. La segunda es que es muy sencilla de implementar y muy rápida de ejecutar. Realizar una comprobación entre dos configuraciones consecutivas parecería una buena idea, pero ralentizaría mucho la simulación, preferimos tener una condición más sencilla (que nos resulta útil) pero que se evalué en tiempo constante.

4.2. Resolución de Problemas Mediante Simulación de Redes de Procesadores Genéticos

Llegado a este punto y una vez presentado el simulador de Redes de Procesadores Genéticos, vamos a ver como se comporta cuando lo usamos para resolver un par de problemas, el Problema de la Mochila Multidimensional y el Problema del Viajante de Comercio. Se han elegido estos dos problemas por tratarse de problemas con complejidad NP-completo, además son problemas muy conocidos y estudiados debido a su gran cantidad de usos en nuestra sociedad. Para cada uno de los problemas hemos seleccionado un artículo donde se trabaja con dicho problema y hemos probado nuestro simulador utilizando las mismas instancias. También hemos mostrado los resultados obtenidos con algoritmos genéticos, puesto que nos ha parecido muy interesante realizar esta comparación.

4.2.1. Mochila Multidimensional

Introducción

El problema de la mochila multidimensional (MKP) es un problema de optimización NP-Completo [26]. Con este problema se pueden modelizar y solucionar problemas reales tales como presupuestación de capital (Lorie y Savage [38], [45] o [8]) o problemas de selección de proyectos (Manne y Markovitz [42]). También tenemos otros ejemplos en donde se ha utilizado el MKP como problemas stock de cortes [29], de carga [57], de colocación de procesadores y bases de datos en sistemas distribuidos [27] o por ejemplo encontrar buenas políticas de inversión para el sector del turismo y desarrollo de un país [28]. En el artículo [23] tenemos una extensa visión general del MKP. Todo esto hace que el MKP sea un problema muy utilizado como benchmark para validar y/o comparar modelos o métodos de resolución de problemas.

Muchos métodos han sido utilizados para solucionar este problema, el hecho de que estemos trabajando con un problema NP-Completo hace que no sea fácil encontrar métodos/algoritmos que proporcionen buenas soluciones para tallas grandes del problema en un tiempo aceptable, algunos ejemplos significativos son el uso de algoritmos exactos como programación dinámica [29] o el método de ramificación y poda [57], algoritmos de aproximación [56], análisis probabilístico como por ejemplo el uso de Lagrangiana [44][6], la utilización de heurísticas [55] y la utilización de metaheurísticas como pueden ser los Algoritmos Genéticos [34]. En el artículo [23] tenemos un extenso resumen sobre los métodos de resolución del MKP.

Definición del Problema

En el Problema de la Mochila clásico tenemos una mochila con una capacidad b de peso y una colección de n elementos con diferentes valores p_j y pesos r_j. El objetivo es elegir el conjunto de elementos cuya suma de valores sea la mayor pero sin exceder la capacidad de la mochila. En el Problema de la Mochila Multidimensional (MKP) tenemos m mochilas con diferentes capacidades b_j y al seleccionar un elemento i, éste se introduce en todas las mochilas con un peso diferente r_{ij} para cada una. El objetivo es elegir el conjunto de elementos cuya suma de valores sea la mayor pero sin exceder la capacidad de ninguna de las m mochilas. Podemos formular el problema de la siguiente forma:

$$x_i \in \{0,1\}\ \forall i = 1...n$$

$$x_i = \begin{cases} 1 \text{ si el objeto } i \text{ es incluido en la mochila} \\ 0 \text{ en otro caso} \end{cases}$$

Función objetivo:

$$\max \sum_{i=1}^{n} p_i x_i$$

Conjunto de restricciones:

$$\sum_{i=1}^{n} r_{ij} x_i \leq b_j\ \forall j = 1...m$$

Codificación de las soluciones

Una posible solución x para este problema la vamos a representar mediante la cadena $x_1 x_2 ... x_n$ donde el elemento i es seleccionado para la solución si $x_i = 1$ y no lo es si $x_i = 0$.

Las únicas mutaciones que se pueden aplicar sobre las cadenas son $0 \rightarrow 1$ y $1 \rightarrow 0$. El significado de aplicar la mutación $0 \rightarrow 1$ al elemento i es que el elemento i pasa de no formar parte a formar parte de la solución. Al aplicar $1 \rightarrow 0$ al elemento i lo que estamos haciendo es quitar el elemento i de la solución.

En este problema la operación de cruce es bastante sencilla, se realizan los cortes y los cambios como en las definiciones previas. El significado no es otro que escoger un trozo de selección de la primera solución y el resto de una segunda solución, obteniendo una nueva selección con elementos heredados de sus antecesores. El único problema es que al no controlar el corte pueden aparecer cadenas demasiado largas o cadenas demasiado cortas. Esto lo solucionamos desechando las cadenas que tengan menos de n elementos y con las cadenas de más de n elementos sólo tendremos en cuenta para la solución los primeros n elementos.

Experimentación y Resultados

Para realizar la experimentación con el problema de la Mochila Multidimensional, hemos utilizado Redes de Procesadores Genéticos como Algoritmos Genéticos Paralelos, que son las que definimos expresamente para trabajar con problemas de optimización. Como función de fitness tomamos

la propia funcion objetivo del problema. Cada población será representada por un procesador del modelo. Los pasos de ejecución del modelo represan la aplicación de operaciones del algoritmo genético y los pasos de comunicación representan la migración entre poblaciones. El banco de pruebas se ha obtenido de [7], concretamente el último problema del archivo "mknap1.txt" que consta de una talla de 5 mochilas y 50 objetos, la razón para esta elección es que se trata de un test lo bastante complicado para que no sea trivial pero tampoco demasiado complicado para no alargar innecesariamente la experimentación.

Se han realizado experimentaciones con diferentes tamaños de población máxima, los que han obtenido mejores resultados han sido tamaños de 10 y 20 individuos. Los tipos de redes usados han sido estrella, completa y anillo por ser las básicas y bus por dar los mejores resultados. Debido a la naturaleza de este método de resolución que consiste en una búsqueda de la solución óptima, vemos que es importante hacer varias pruebas con elementos que introduzcan cierta variabilidad al proceso, para ello hemos implementado unos procesadores especiales, en los cuales el filtro de entrada sólo implementa las restricciones del problema (no tiene en cuenta el fitness), esto hace que se introduzca variabilidad al conjunto de las n mejores soluciones y aumenta la posibilidad de encontrar mejores máximos locales (o el global). Recordemos que una solución al problema es el beneficio más alto obtenido de los objetos que se han podido meter en las mochilas, así pues en cada ejecución tendremos como resultado dicho beneficio. Debido a que en cada ejecución podemos tener una solución distinta, para cada tipo de red hemos realizado 6 experimentos y mostramos la media para hacernos una mejor idea del funcionamiento de cada red.

Los resultados de la experimentación aparecen en la figura 4.3. Antes que nada debemos decir que todas las redes utilizadas siempre han dado mejores resultados que los Algoritmos Genéticos (cuyo valor se ha calculado previamente para poder comparar y se halla en la primera fila de la tabla), como podemos observar en la tabla. En los resultados podemos apreciar que la mejor tipología de la red es la bus y de las peores son la estrella o la completa. Esto puede pasar debido a las conexiones entre los procesadores, las topologías de bus tienen conjuntos de cadenas que pueden tardar muchos ciclos en unirse (aplicarse cruce entre ellas) y esto hace que el sistema tenga mayor variabilidad. Por otra parte las redes completas tienen un comportamiento bastante similar a un Algoritmo Genético, pues en cada paso, como las cadenas pueden ir a cualquier procesador, cualquier par de cadenas puede coincidir en el mismo procesador y se les puede aplicar cruce. Por último las redes de topología estrella pierden muchas cadenas, pues cuando todas las cadenas van al procesador central, sólo nos quedamos con un número de

Tipo de Red (población máxima 10)	Media
Algoritmos Genéticos	14935,5
Estrella (4 procesadores)	15314,4
Completa (6 procesadores)	15537,2
Bus (6 procesadores)	15450
Bus (4 procesadores) (un procesador especial)	15800,8
Bus (3 procesadores) (un procesador especial en medio)	15570
Bus (4 procesadores) (población máxima 20)	15507,6
Anillo (10 procesadores) (población máxima 20)	15672,1
Bus (4 procesadores) (un procesador especial)(p. máxima 20)	15998

Figura 4.3: Media de las soluciones obtenidas al aplicar Redes de Procesadores Genéticos como Algoritmos Genéticos Paralelos al Problema de la Mochila Multidimensional. El problema consta de 5 mochilas y 50 objetos.

cadenas igual a la población máxima definida (por ejemplo, si 4 procesadores le envían sus 10 cadenas al procesador centrar y la población máxima es 10, perdemos 30 cadenas en un sólo paso).

4.2.2. Viajante de Comercio

Introducción

El problema del viajante de comercio (TSP) es un problema de optimización NP-Completo [26]. La aplicación más directa de este problema son tal vez las relacionadas con gestión de rutas para vehículos, pero tenemos un conjunto tan amplio de aplicaciones como el cableado de computadores o redes [37], secuenciación de trabajos, diseño de dianas [21], cristalografía [9] o algunas aplicaciones industriales como corte de rollos de papel [24], agujerear en el mínimo tiempo posible una superficie siguiendo un patrón [50] o fabricación de chips VLSI [35].

Debido a la naturaleza y características del problema, el TSP y cómo solucionarlo ha sido un tema muy estudiado, alguno de los métodos clásicos más usados son la utilización de heurísticas, la programación lineal o métodos de ramificación y poda [59]. También podemos ver otros métodos muy conocidos como redes neuronales [1], algoritmos genéticos [10] o por colonias artificiales de hormigas [18].

Definición del Problema

En el Problema del Viajante de Comercio tenemos n ciudades y un conjunto de caminos entre pares de ciudades con una distancia (coste) asignado. El objetivo es encontrar una ruta que, empezando y terminando por la misma ciudad, pase una sola vez por cada una de las ciudades y minimice la distancia del recorrido.

Este problema se puede representar mediante un grafo dirigido $G = \langle N, A \rangle$, donde N es un conjunto de n nodos y A es el conjunto de aristas ponderadas del grafo, los nodos representan las ciudades y las aristas las distancias entre ellas, con lo que, en un principio, los pesos de las aristas nunca tomarán valores negativos. Como se tiene un grafo dirigido, hay que tener en cuenta que la distancia (coste) para ir desde i hasta j no tiene por qué ser igual a la distancia (coste) para llegar desde j hasta i.

Hablando en términos de grafos, el objetivo del problema es encontrar el *ciclo hamiltoniano* de menor coste dado un grafo ponderado (un ciclo hamiltoniano es aquel que pasa por todos los nodos del grafo exactamente una vez). Formulando el problema matemáticamente, se dice que la finalidad del mismo consiste en encontrar el camino C formado por la secuencia de n ciudades, $C = \{1, 2, ..., n\}$, donde $\forall i \in N, i \in C$, que consiga:

Función objetivo:

$$\min \left(\sum_{i=1}^{n-1} A[C_i, C_{i+1}]\right) + A[C_n, C_1]$$

Codificación de la Red

Una posible solución para este problema la vamos a representar mediante la cadena $C = C_1C_2...C_n$ donde cada uno de los símbolos C_i identifica una cuidad. Dicha cadena codifica el camino que empieza en C_1, va de C_1 a C_2, de C_2 a C_3, ... , de C_{n-1} a C_n y finalmente de C_n a C_1.

Partimos de la base que una posible solución será correcta si $\forall i, j$: $C_i \neq C_j, i \neq j$, esto es debido a que sólo debemos pasar una sola vez por cada ciudad. Si aplicamos una regla básica de mutación lo que hacemos es substituir un símbolo de la cadena por otro diferente, con lo que el resultado será una cadena que codifica una solución que pasa dos veces por la misma ciudad y ninguna por otra, con lo cual tenemos una solución incorrecta. Para solucionar esto hemos redefinido las reglas de mutación en este problema. Así la regla $C_i \rightarrow C_j$ lo que hace es intercam-

biar la posición de la ciudad i por la de la ciudad j, es decir, si le aplicamos esta regla a la cadena $C_1...C_{i-1}C_iC_{i+1}...C_{j-1}C_jC_{j+1}...C_n$ el resultado es $C_1...C_{i-1}C_jC_{i+1}...C_{j-1}C_iC_{j+1}...C_n$, con esto, si la cadena inicial era una solución correcta, el resultado también lo será. Se puede ver esta modificación de la mutación como dos mutaciones clásicas simultaneas, $C_i \to C_j$ y $C_j \to C_i$.

En cuanto a la operación de cruce tenemos el mismo efecto, hay una alta probabilidad de que al aplicar una operación de cruce sobre dos soluciones correctas obtengamos como resultado dos soluciones incorrectas (pues se mezclan caminos pudiendo repetir ciertas ciudades o no pasar por otras). Para solucionarlo redefinimos la operación de cruce de la siguiente forma, primero realizamos un corte en una posición aleatoria de la primera cadena, tomamos el primer segmento y completamos con el resto de ciudades que nos quedan en el mismo orden en que aparecen en la segunda cadena y luego realizamos la misma operación pero intercambiando el rol las dos cadenas. Con esto obtenemos dos soluciones correctas y mantenemos el espíritu de la operación (y de los algoritmos genéticos), puesto que nos quedamos con parte de la primera solución y, en cierta manera, parte de la segunda. Para ver un ejemplo, si aplicamos un cruce entre las cadena (04123576) y la cadena (47156230) un posible resultado (recordemos que el cruce es indeterminista) podría ser {(**0412**7563),(**47156**023)}.

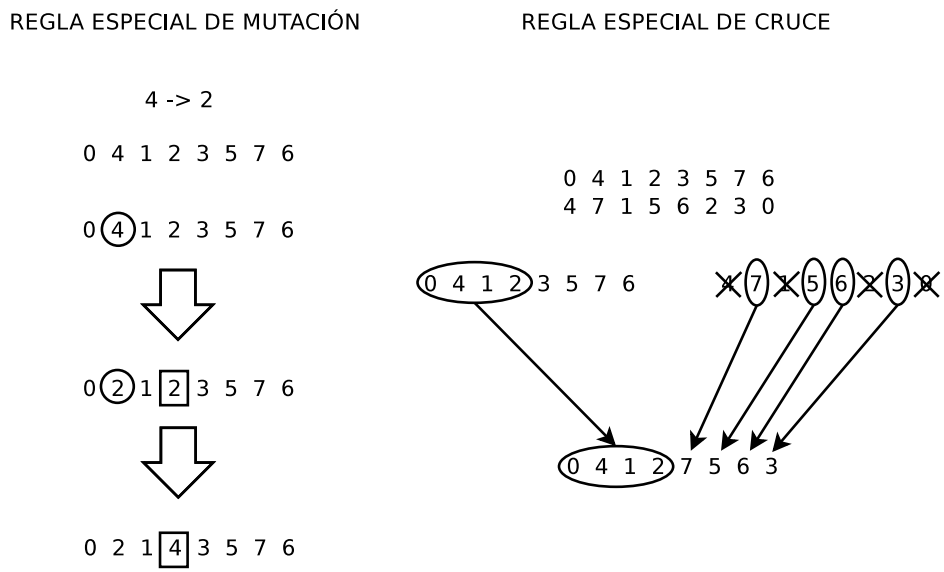

Figura 4.4: Ejemplo del funcionamiento de las reglas de cruce y mutación redefinidas para poder resolver el problema de optimización del viajante de comercio.

Experimentación y Resultados

Para la resolución del Viajante de Comercio hemos utilizado Redes de Procesadores Genéticos como Algoritmos Genéticos Paralelos. El banco de pruebas se ha obtenido de [25], con una talla de 30 ciudades y con caminos para todo par de ciudades (simulando un grafo completo). El motivo de la elección de este banco de pruebas es que 30 ciudades constituyen un problema de dificultad moderada con lo que podemos apreciar la dificultad del mismo, pero no perder demasiado tiempo en las experimentaciones, además el uso de grafos completos nos simplifica el validar las soluciones.

En este problema se han realizado experimentos para unos tamaños de poblaciones máximas de 10 y 20 donde se han conseguido los mejores resultados, aunque también se ha hecho algún experimento marginal con población máxima 30. Los tipos de redes usados, al igual que en el problema anterior, han sido estrella, completa y anillo por tratarse de las topologías básicas y la red en bus por dar muy buenos resultados. Por las mismas razones que con la mochila multidimensional, en este problema también se han utilizados procesadores especiales para algunos experimentos, recordemos que estos procesadores sólo implementan las restricciones del problema y no la función de fitness, con lo que añadimos tipos diferentes de soluciones y más variabilidad al proceso. En este problema, la solución consiste en el coste menor encontrado al recorrer todas las ciudades (empezando y terminando por la misma), con lo que dicho coste será lo que obtengamos cada vez que lanzamos una ejecución. Al igual que en el problema anterior, en cada ejecución podemos tener una solución distinta, así que hemos realizado 6 experimentos para cada tipo de red y mostramos la media, la mejor y la peor solución.

Los resultados de la experimentación aparecen el la figura 4.5. Primero, como en el problema anterior, todos los resultados mejoran los de los obtenidos con Algoritmos Genéticos (valor que, al igual que en el problema anterior, ha sido calculado para poder compararlo con las demas soluciones). La mejor media de resultados la hemos obtenido con una población máxima de 20 individuos, pero la mejor solución ha sido con 30. En cuando al tipo de redes, las conclusiones son las mismas que en la Mochila, la red en forma de bus sigue siendo la mejor y las redes completas o en forma de estrella son los que nos dan peores resultados, vemos pues que la variabilidad es un elemento importante (como siempre lo ha sido) en modelos de búsqueda de soluciones.

Población máxima 10	media	mejor	peor
Algoritmos Genéticos	852,99	675,57	982,83
Completa (7 procesadores)	550,07	495,66	624,01
Bus (16 procesadores)	528,52	485,71	601,6
Estrella (10 procesadores)	512,18	484,25	545,11
Anillo (13 procesadores)	549,79	521,13	599,56
Población máxima 20			
Algoritmos Genéticos	676,25	625,8	732,72
Bus (13 procesadores)	502,35	428,28	553,18
Bus (16 procesadores)	482,4	453,26	519,58
Bus (20 procesadores)	503,03	447,66	576,3
Completa (20 procesadores)	502,46	442,51	567,4
Bus (20 procesadores) (6 procesadores especiales)	491,07	436,95	541,59
Población máxima 30			
Bus (20 procesadores)	499,71	423,25	539,25
Completa (20 procesadores)	496,01	457,65	540,99

Figura 4.5: Media, mejor y peor solución al aplicar Redes de Procesadores Genéticos como Algoritmos Genéticos Paralelos al Problema del Viajante de Comercio. La talla del problema es de 30 ciudades y están todas conectadas.

Capítulo 5

Conclusiones

En esta tesis hemos propuesto un nuevo modelo de computación inspirado en la biología y que pasaría a formar parte del conjunto de modelos de procesadores bioinspirados: las Redes de Procesadores Genéticos (NGP). Este nuevo modelo aporta un estudio sobre el uso de la combinación de dos operaciones biológicas muy representativas: las mutaciones y el cruce.

En cuanto a la parte teórica hemos definido formalmente el modelo, hemos demostrado que es completo (mediante la simulación de una Máquina de Turing), hemos definido nuevas medidas de complejidad y hemos realizado un ejemplo de medida de complejidad sobre un problema NP, demostrando que con nuestro modelo podemos solucionar problemas NP en un número de pasos lineal. Para redondear también hemos hecho un estudio del mínimo número procesadores necesarios para simular cada una de las gramáticas de la jerarquía de Chomsky, observando que dicho número crece cuando más compleja es la gramática.

Otro elemento importante de la tesis a nivel teórico es que hemos demostrado a través de las NGP que los Algoritmos Genéticos Paralelos son un modelo completo, un resultado al que no se había llegado a pesar del tiempo que se llevan usando los Algoritmos Genéticos.

No hemos querido abandonar la parte práctica del estudio. Para empezar a poder trabajar con las NGP hemos definido tres tipos de redes diferentes, las aceptoras, las generadoras y las que trabajan como Algoritmos Genéticos Paralelos, siendo estas últimas ideales para la resolución de problemas de optimización. Con el último tipo de redes hemos probado el modelo con dos problemas, el problema de la mochila multidimensional y el problema del viajante de comercio obteniendo buenos resultados. También se muestra algunas de las clases en java del modelo implementado que hemos usado para los experimentos.

En cuanto a trabajo futuro, lo que más nos gustaría es la posible inte-

gración en hardware, para que así se aproveche de verdad la potencia que nuestro modelo.

Otros frentes que tenemos abiertos son el uso de nuestro modelo para resolver problemas de ámbito biológico, como el multialineamiento de biosequencias, ya que observando las operaciones de las NGP podría dar buenos resultados.

También nos gustaría relacionar nuestro modelos con otros modelos conocidos como los P-sistemas o profundizar más en la relación con los Algoritmos Genéticos Paralelos.

Publicaciones de la tesis

- Marcelino Campos, José M. Sempere. Solving Combinatorial Problems with Networks of Genetic Processors. International Journal "Information Technologies and Knowledge"Vol.7 No. 1, pp 65-71. 2013

En este artículo realizamos un estudio de las Redes de Procesadores Genéticos a la hora de resolver un problema de decisión complejo como es el problema de saber si existe un ciclo hamiltoniano en un grafo, definimos una codificación para que se pueda ejecutar en el modelo y obtenemos que el número de pasos necesario depende linealmente con el número de vértices del grafo de entrada.

En la esta tesis podemos ver este estudio en el punto 3.2.2.

- Marcelino Campos, José M. Sempere. Accepting Networks of Genetic Processors are computationally complete. Theoretical Computer Science Vol. 456, pp 18-29. 2012

En este artículo se define formalmente el modelo de las Redes de Procesadores Genéticos, se demuestra que es un modelo completo simulando en el una máquina de Turing, se establece una relación entre Redes de Procesadores Genéticos y Algoritmos Genéticos Paralelos para, a continuación, poder demostrar que los Algoritmos Genéticos Paralelos son computacionalmente completos.

En la tesis tenemos la presentación del modelo en el punto 3.1, la demostración de su completitud en el punto 3.2 y la parte relacionada con Algoritmos Genéticos en el punto 3.2.3.

- M. Campos, J. González, T.A. Pérez, J. M. Sempere. Implementing Evolutionary Processors in JAVA: A case study. In 13th International

Symposium on Artificial Life and Robotics (AROB 2008) (Beppu, Japan) January 31 - February 2. 2008 Proceedings edited by M. Sugisaka and H. Tanaka pp 510-515. 2008 ISBN: 978-4-9902880-2-0

En dicho artículo se explica como se abordó la implementación en Java de un simulador de Redes de Procesadores Evolutivos, posteriormente a este simulador le incorporamos la posibilidad de trabajar con reglas de mutación y cruce, con lo que lo pudimos utilizar para simular Redes de Procesadores Genéticos.

En la tesis se puede encontrar esta información en el punto 4.1.

Bibliografía

[1] E.H.L Aarts, H.P. Stehouwer. Neural networks and the travelling salesman problem. Proc, S. Gielen and B. Kappan (Eds.), Springer-Verlag, Berlin, 1993, 950-955.

[2] L. Adleman. Molecular computation of solutions to combinatorial problems. Science 266 (1994), 1021-1024.

[3] E. Alba, M. Tomassini. Parallelism and Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, Vol. 6. No. 2, pp 443-462. 2002.

[4] E. Alba, J.M. Troya. A survey of parallel distributed genetic algorithms. *Complexity*, Vol. 4, No. 4, pp 31-52. 1999.

[5] M. Arbid. The Handbook of Brain Theory and Neural Networks. *MIT Press 2003*

[6] I. Averbakh. Probabilistic properties of the dual structure of the multidimensional knapsack problem and fast statistically efficient algorithms. Mathematical Programing 62 (1994) 311-330.

[7] J.E.Beasley. *OR-Library: distributing test problems by electronic mail.* Journal of the Operational Research Society 41(11) (1990) pp1069-1072

[8] G.J. Beaujon, S.P. Marin, G.C. McDonald. Balancing and optimizing a portfolio of r & d projects. Naval Research Logistics, 48(1):18-40, 2001.

[9] R.G. Bland, D.F. Shallcross. Large travelling salesman problems arising experiments in X-ray crystallography: A preliminary report on computation. Operations Research Letters 8 (1989) 125-128.

[10] H. Braun. On solving travelling salesman problems by genetic algorithms. Parallel Problem Solving from Nature, H.-P. Schewefel and R. Männer (Eds) Lecture Notes in Computer Science, Vol. 496, Springer, Berlin, 1991, 129-133.

[11] M. Blum. A Machine-Independent Theory of the Complexity of Recursive Functions. *Journal of the ACM*, Vol 14, No. 2. pp 322-336. 1967.

[12] M. Campos, J. González, T.A. Pérez, J.M. Sempere. Implementing Evolutionary Processors in JAVA: A case study. 13th International Symposium on Artificial Life and Robotics 2008. 510-515.

[13] M. Campos, J. M. Sempere. Accepting Networks of Genetic Processors are computationally complete. Theoretical Computer Science 456 (2012) 18-29.

[14] E. Cantú-Paz. *Efficient and Accurate Parallel Genetic Algorithms.* Kluwer Academic Publisher. 2001.

[15] J. Castellanos, Carlos Martín-Vide, Victor Mitrana, José M. Sempere. Networks of evolutionary processors. *Acta Informatica* 39, pp 517-529. 2003.

[16] J. Castellanos, C. Martín-Vide, V. Mitrana, José M. Sempere. Solving NP-Complete Problems with Networks of Evolutionary Processors. In *Proceedings of the 6th International Work-conference on Artificial and Natural Neural Networks, IWANN 2001* LNCS Vol. 2084, pp 621-628. Springer. 2001

[17] S.A. Cook. The Complexity of Theorem Proving Procedures. Proceedings Third Annual ACM Symposium on Theory of Computing (1971) pp 151-158

[18] M. M. Dorigo, L.M. Gambardella. Ant colonies for the travelling salesman problem. BioSystems 43 (1997) 73-81

[19] D. Du, K. Ko. *Theory of Computational Complexity.* John Wiley & Sons, Inc. 2000.

[20] C. Drăgoi, F. Manea, V. Mitrana. *Accepting networks of evolutionary processors with filtered conections.* J Univers Comput Sci 13:1598-1614 (2007)

[21] H.A. Eiselt, G. Laporte. A combinational optimization problem arising in dartboard design. Journal of the Operational Research Society 42 (1991) 113-118.

[22] D.G. Feitelson. *Optical Computing: A Survey for Computer Scientists.* Cambridge, MA: MIT Press. ISBN 0-262-06112-0.

[23] A. Fréville. The multidimensional 0-1 Knapsack problem: An overview. European Journal of Operational Research, 127(1):1-21. 2004.

[24] R.S. Garfinkel. Minimizing wallpaper waste, Part I: A class of travelling salesman problems. Operations Research 25 (1977) 741-751.

[25] Gh. Gao Shang, Zhang Lei, Zhuang Fengting, Zhang Chunxian. *Solving Traveling Salesman Problem by Ant Colony Optimization Algorithm with Association Rule*. Third International Conference on Natural Computation 2007

[26] M.R. Garey, D.J. Johnson. Computers and Intractability. A Guide to the Theory of NP-Completeness. W.H. Freeman, San Francisco, 1979.

[27] B. Gavish, H. Pirkul. Allocation of databases and processors in a distributed computing system. Management of Distributed Data Processing, 31:215-231, 1982.

[28] C.E. Gearing, W.W. Swart, T. Var. Determining the optimal investment policy for the tourism sector of a developing country. Management Science, 20(4):487-497, 1973.

[29] P.C. Gilmore, R.E. Gomory. The theory and computation of knapsack functions. Operations Research 14 (1966) 1045-1075.

[30] T.Head *Formal language theory and DNA: an analysis of the generative capacity of specific recombinant Behaviors*. Bull. Math. Biol. 49 (1987) 737-759.

[31] J.E. Hopcroft, R. Motwani, J.D. Ullman. *Introduction to Automata Theory, Langages and Computation*. Pearson Education. Addison Wesley, http://www-db.stanford.edu/ullman/ialc.html, second edition, 2001.

[32] C.S. Horstmann, G. Cornell. Core Java. (Vols 1 and 2) Prentice Hall 2005.

[33] L. Kari, G. Rozenberg. The Many Facets of Natural Computing. *Communications of the ACM* Vol. 51 No.10, pp 72-83. 2008.

[34] S. Khuri, T. Back, J. Heithotter. The zero/one multiple knapsack problem and genetic algorithms. Proceedings of the 1994 ACM Symposium on Aplied Computing, (SAC'99), ACM Press, London, 1994, pp. 188-193.

[35] B. Korte. Applications of combinatorial optimization. Talk at the 13th International Mathematical Programing Symposium, Tokyo 1988.

[36] L.F. Landweber, L. Kari *Universal Molecular Computation in Ciliates.* Evolution as Computation (2002) 516-536 Springer

[37] J.K. Lenstra, A.H.G. Rinnooy Kan. Some simple applications of the travelling salesman problem. Operations Research Quarterly 26 (1975) 717-733.

[38] J.H. Lorie, L.J. Savage. Three problems in rationing capital. The Journal of Busines, 28(4):229-239, 1955.

[39] F. Manea, C. Martín-Vide, V. Mitrana. Accepting networks of splicing processors. In *Proceedings of the First Conference on Computability in Europe, CiE 2005*, LNCS Vol. 3526, pp 300-309. Springer. 2005.

[40] F. Manea, C. Martín-Vide, V. Mitrana. Accepting networks of splicing processors: Complexity results. In *Theoretical Computer Science* 371 (2007) 72-82.

[41] M. Margentern, V. Mitrana, M.J. Pérez-Jiménez. Accepting hybrid networks of evolutionary processors. In Procedings of the International Meeting on DNA Computing, DNA 10, in: LNCS, vol. 3384, Springer, 2005, pp. 235-246.

[42] H.M. Markowitz, A.S. Manne. On the solution of discrete programming problems. Econometrica, 25(1):84-110, 1957.

[43] C. Martín-Vide, G. Păun, J. Pazos, A. Rodríguez-Patón. *Tissue P Systems* Theoretical Computer Science 296 (2003) 295-326

[44] M. Meanti, A.H.G. Rinnooy Kan, L. Stougie, C. Vercellis. A probabilistic analysis of the multiknapsack value function. Mathematical Programing 46 (1990) 237-247.

[45] H. Meier, N. Christofides, G. Salkin. Capital budgeting under uncertainty-an integrated aproach using contingent claims analysis and integer programing. Operations Research, 49(2):196-206, 2001.

[46] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs, (Third, Revised and Extended Edition).* Springer. 1996.

[47] D. Pixton Regularity of splicing languages. Discrete Applied Mathematics 69 (1996) 101-124.

[48] Gh. Păun. On the splicing operation. Discrete Applied Mathematics 70 (1996) 57-79.

[49] Gh. Păun. Computing with Membranes. Submitted, 1998 (see also TUCS Research Report No. 208, November 1998, http://www.tucs.fi).

[50] G. Reinelt. Fast heuristics for large geometric traveling salesman problems. Report N° 185, Institut für Mathematic, Universität Augsburg (1989).

[51] G. E. Revesz. *Introduction to Formal Languages.* McGrawHill, New York (1982).

[52] E. Rieffel, W. Polak *An Introduction to Quantum Computing for Non-Physicists.* ACM Computing Surveys, Vol. 32, No. 3, September 2000, pp. 300–335.

[53] G. Rozenberg, A. Salomaa. The Mathematical Theory of L Systems. Academic Press, 1980.

[54] M. Schwehm. *Parallel Population Models for Genetic Algorithms* Universität Erlangen-Nürnberg. 1996.

[55] S. Senju, Y. Toyada. An aproach to linear programming problems with 0-1 variables. Management Sciences 15 (1968) 196-207.

[56] S. Shani. Approximate algorithms for the 0-1 knapsack problem. Journal of Association for Computing Machinery 22 (1975) 115-124.

[57] W. Shih. A branch and bound method for the multiconstraint zero-one knapsack problem. The Journal of the Operational Research Society, 30(4):369-378, 1979.

[58] S. Wolfram. "Cellular Automata", in Cellular Automata and Complexity: Collected Papers. 1994 Addison Wesley, Steven Wolfram, Reading, MA

[59] L. Wolsey. Heuristic analysis, linear programing, and branch and bound. Math. Prog. Stud. 13 (1980) 121-134.