UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Inferencia gramatical de lenguajes incontextuales mediante computación natural y evolutiva

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Llanes Lacomba, Rodrigo

Tutor/a: Sempere Luna, José María

CURSO ACADÉMICO: 2021/2022

Resum

La inferència gramatical és el procés d'aprenentatge automàtic d'una gramàtica formal a partir d'una sèrie de mostres. Amb aquesta finalitat han sigut proposades i provades una àmplia varietat de tècniques. En aquest treball concretament, provem l'eficàcia de la computació evolutiva combinada amb els sistemes P en teixit, implementant un simulador en el llenguatge de programació Python3 i entrenant-lo amb diferents paràmetres i dades d'entrenament, per tratar de trobar-ne la configuració òptima. Els resultats finals, concorden amb els mostrats per Taishin I. Nishida, mostrant eficàcia a l'hora d'inferir alguns llenguatges, com $a^n b^n$ o el llenguatge de Dyck.

Paraules clau: inferència gramatical, computació evolutiva, sistemes P en red, Python3

Resumen

La inferencia gramatical, es el proceso de aprendizaje automático de una gramática formal a partir de una serie de muestras. Con este fin han sido propuestas y probadas una amplia variedad de técnicas. En este trabajo concretamente, probamos la eficacia de la computación evolutiva combinada con los sistemas P en tejido, implementando un simulador en el lenguaje de programación Python3 y entrenándolo con distintos parámetros y datos de entrenamiento, para tratar de encontrar su configuración óptima. Los resultados finales, concuerdan con los mostrados por Taishin Y. Nishida, mostrando eficacia a la hora de inferir algunos lenguajes, como $a^n b^n$ o el lenguaje de Dyck.

Palabras clave: inferencia gramatical, computación evolutiva, sistemas P en red, Python3

Abstract

Grammatical inference is the process of automatically learning a formal grammar from a set of samples. For which a wide variety of techniques have been proposed and tested. In this paper we specifically test the effectiveness of evolutionary computation combined with Tissue P-systems by implementing a simulator in the Python3 programming language and training it with different parameters and training data in order to find its optimal configuration. The final results are in agreement with those shown by Taishin Y. Nishida, showing efficiency in inferring some languages, such as $a^n b^n$ or Dyck's language.

Key words: grammatical inference, evolutionary computation, tissue P systems, Python3

Índice general

Apéndices

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

El proceso de aprendizaje y comprensión de las reglas de un lenguaje para un computador no es tan sencillo como para un ser humano, por ello se han desarrollado una amplia variedad de técnicas y metodologías con distintas tecnologías cuyo objetivo es optimizar este proceso de aprendizaje.

En este TFG nos enfocaremos en ampliar el trabajo realizado por Taishin Y. Nishida en su artículo "Evolutionary P Systems: The notion and an Example"[Nis20], en el que estudia diversos métodos basados en la computación con membranas y los algoritmos evolutivos (sobre los que hablaremos en profundidad más adelante). Nos centraremos en uno de los sistemas que propone, realizando una implementación del mismo y estudiando la influencia de distintos parámetros, con el fin de optimizarlo.

1.1 Motivación

La motivación principal tras este trabajo es la misma que hay tras cualquier investigación, ampliar los conocimientos existentes sobre el área de estudio con el fin de poder aportar nuestro pequeño grano de arena a futuras investigaciones o desarrollos que puedan encontrar útiles los resultados obtenidos.

Más específicamente en el campo de la Inferencia Gramatical, nuestro motor principal será la mejora de estas técnicas que tantos usos han demostrado tener en campos tan diversos como la bioinformática o la programación, por citar algunos de ellos.

1.2 Objetivos

Este trabajo tiene 3 objetivos principales, en primer lugar desarrollar un simulador funcional que pueda ser usado para inferir las reglas de un lenguaje formal en base a muestras etiquetadas.

En segundo lugar realizar una experimentación lo más exhaustiva posible, para determinar como afectan los distintos parámetros a la eficacia de este sistema con el fin de encontrar una configuración óptima.

Y por último, dejar la puerta abierta a nuevas investigaciones que puedan surgir a partir de los frentes abiertos que dejamos por el camino.

1.3 Estructura de la memoria

Esta memoria se compone de siete capítulos, el primero (1), en el cual nos encontramos, es una breve introducción al trabajo.

En el segundo capítulo (2), presentaremos unas nociones básicas sobre teoría de lenguajes e inferencia gramatical, para posteriormente en el capítulo 3 hablar concretamente de los algoritmos genéticos.

En el cuarto capítulo (4), introducimos y desarrollamos el concepto de computación con membranas, los sistemas P y los sistemas P en tejido.

En el capítulo 5, para empezar a profundizar en el tema concreto de este trabajo, repasaremos el artículo de Taishin Y. Nishida, las técnicas empleadas por él y sus resultados. Para, de este modo, en el capítulo sexto (6) poder ver la implementación realizada en base a su trabajo, las pruebas y mejoras propuestas, y los resultados obtenidos.

Y para acabar remataremos este trabajo con el capítulo 7, en el que discutiremos las conclusiones obtenidas y dejaremos las puertas abiertas a futuros trabajos.

CAPÍTULO 2
Inferencia gramatical

En este capítulo, nos centraremos en explicar los conceptos del campo de la inferencia gramatical, necesarios para la comprensión de este trabajo. La fuente principal empleada para la documentación de este capítulo es el libro "Grammatical Inference: Learning Automata and Grammars"[Hig13] cuyo contenido seleccionaremos y resumiremos, con el fin de dar una imagen general de este campo, pero sin desviarnos demasiado del tema central de esta memoria.

2.1 Nociones básicas

Al trabajar con lenguajes, surge la necesidad de modelarlos matemáticamente, para poder trabajar con ellos, estudiarlos o incluso, como pretendemos, predecirlos. Para ello introduciremos una serie de conceptos básicos extraídos de [HMU01].

El primer concepto necesario es el de alfabeto. Definimos un alfabeto Σ como un conjunto finito no vacío, de elementos a los que llamaremos símbolos. Aunque típicamente se usan alfabetos de 2 símbolos, en casos reales, podrían usarse alfabetos mucho más largos o incluso de un tamaño indefinido (pero siempre finito). Sobre estos alfabetos se pueden realizar las operaciones típicas de los conjuntos como la unión, intersección, diferencia, etc. Por ejemplo, podemos definir los siguientes alfabetos y las siguientes operaciones entre ellos:

$$\Sigma_1 = \{a, b\} \qquad \Sigma_2 = \{b, c\}$$
$$\Sigma_1 \cup \Sigma_2 = \{a, b, c\}$$
$$\Sigma_1 \cap \Sigma_2 = \{b\}$$
$$\Sigma_1 - \Sigma_2 = \{a\}$$

Estos alfabetos, al igual que sucede en la vida cotidiana, representan las letras que pueden formar nuestras palabras, a las que también llamaremos cadenas o strings. Una cadena x sobre un alfabeto Σ se define como una secuencia finita y ordenada de símbolos de Σ. Una cadena de Σ_1 podría ser por lo tanto:

$$x_1 = aabb$$

La longitud de una cadena w se denota como $|w|$ y es el número de símbolos que contiene dicha cadena. La cadena vacía es un caso particular, en el que $|w| = 0$, es decir, que la cadena no contiene ningún símbolo, en este caso se denotará la cadena como ε.

Los conjuntos de cadenas se conocen como lenguajes. Un lenguaje L es por lo tanto, cualquier conjunto de cadenas, no necesariamente finito, incluyendo el conjunto vacío. Por ejemplo, podemos definir un lenguaje L_1 como:

$$L_1 = \{a^n b^n : n > 0\}$$

Dos operadores ampliamente utilizados son el cierre de Kleene o estrella de Kleen (*) y el operador más o estrella positiva ($^+$). En el caso de la estrella de Kleen, sobre un alfabeto Σ se define Σ^* como el conjunto de todas las cadenas posibles creadas a partir de concatenar los símbolos de Σ (incluida la cadena vacía). Por ejemplo:

$$\Sigma = \{a, b\}$$
$$\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, ...\}$$

La estrella positiva, sobre un alfabeto Σ, se define a partir de la estrella de Kleen, como el conjunto resultante de eliminar la cadena vacía de Σ^*, por ejemplo:

$$\Sigma = \{a, b\}$$
$$\Sigma^+ = \{a, b, aa, ab, ba, bb, aaa, ...\}$$

Otra forma ampliamente extendida de denotar un lenguaje son las gramáticas formales, una gramática formal $G = (N, \Sigma, P, S)$ se compone de un conjunto de símbolos no terminales N, un conjunto de símbolos terminales (disjunto de N) Σ, un símbolo inicial perteneciente a N (normalmente S) y un conjunto de producciones, cada una de las cuales viene representada por un par de cadenas w_1 y w_2, normalmente denotadas como $w_1 \to w_2$.

A partir de todos estos elementos, podemos definir el concepto de derivación como la operación $\underset{G}{\Rightarrow}$ entre dos cadenas pertenecientes a $(N \cup \Sigma)^*$. Si $A \to B$ es una producción en P y $\alpha, \beta \in (N \cup \Sigma)^*$, entonces $\alpha A \beta \underset{G}{\Rightarrow} \alpha B \beta$. En cuyo caso decimos que la producción $A \to B$ se aplica a $\alpha A \beta$ para obtener $\alpha B \beta$, o que $\alpha A \beta$ deriva directamente a $\alpha B \beta$ en G.

En base a este operador, podemos definir $\overset{*}{\underset{G}{\Rightarrow}}$, suponiendo que $\alpha_1, \alpha_2, ..., \alpha_m$ son cadenas sobre $(N \cup \Sigma)^*$, $m \geq 1$ y

$$\alpha_1 \underset{G}{\Rightarrow} \alpha_2, \alpha_2 \underset{G}{\Rightarrow} \alpha_3, ..., \alpha_{m-1} \underset{G}{\Rightarrow} \alpha_m$$

.

Entonces decimos que $\alpha_1 \underset{G}{\Rightarrow} \alpha_m$ o que α_1 deriva a α_m en G. Por último y a partir del operador $\overset{*}{\underset{G}{\Rightarrow}}$, obtenemos que el lenguaje generado por G, denotado como $L(G)$, es $\{w | w \in \Sigma^* \wedge S \overset{*}{\underset{G}{\Rightarrow}} w\}$

Para ilustrar todo lo visto anteriormente, proponemos el siguiente ejemplo de gramática formal:

$$G = (\{S, A\}, \{a, b\}, P, S)$$

$$P = \begin{cases} S \to aAb \\ A \to aAb \\ A \to \varepsilon \end{cases}$$

Esta gramática está describiendo al lenguaje L_1 visto anteriormente, pues empezando por el símbolo inicial S y realizando las transformaciones descritas por las producciones podemos llegar a generar todas las cadenas del conjunto $\{a^n b^n : n > 0\}$, o lo que es lo mismo $L(G) = L_1$.

Una herramienta muy útil a la hora de visualizar la derivación de una cadena, es el árbol de derivación, como el que podemos observar en 2.1. Un árbol, es un árbol de derivación si:

- Todos los nodos están etiquetados con símbolos pertenecientes a $N \cup \Sigma \cup \{\varepsilon\}$.
- La etiqueta del nodo raíz es S.
- Todas los nodos interiores están etiquetados con símbolos pertenecientes a N.
- Si un nodo n tiene una etiqueta A y los nodos $n_1, n_2, ..., n_k$ son sus hijos, ordenados de izquierda a derecha con etiquetas $X_1, X_2, ..., X_k$ respectivamente, entonces $A \rightarrow X_1X_2...X_k$ debe ser una producción de P.

De este modo el árbol describe el proceso de derivación de una cadena desde el símbolo inicial de una gramática, como en la figura 2.1, en la que el árbol describe la derivación de la cadena *aaabbb* con la gramática formal descrita anteriormente.

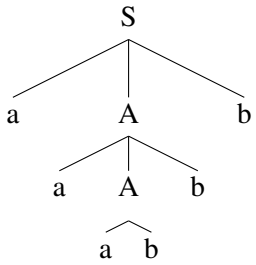

Figura 2.1: Arbol de derivación de *aaabbb* en L_1

2.2 Tipos de lenguajes

Como puede deducirse existe una cantidad infinita de lenguajes, por lo que dada su amplia variedad, surge la necesidad de clasificarlos. Una posible taxonomía es la Jerarquía de Chomsky, mostrada en la figura 2.2.

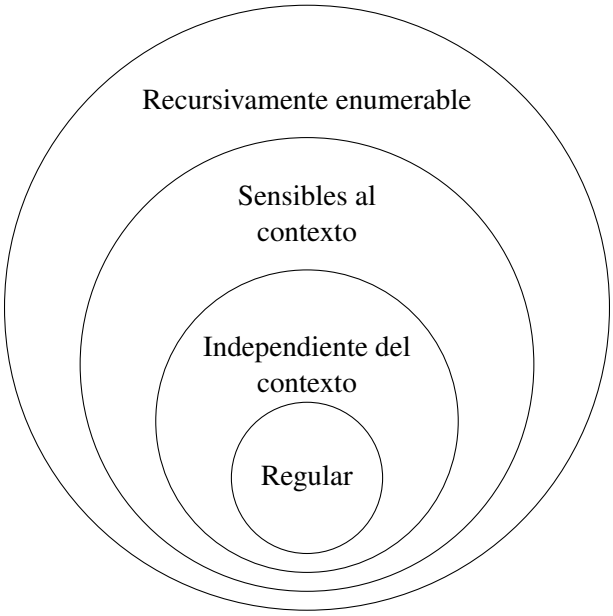

Figura 2.2: Jerarquía de Chomsky

Esta jerarquía divide los lenguajes en un grupo principal (los lenguajes recursivamente enumerables) que contiene a los otros tres subgrupos, a su vez el segundo grupo (los lenguajes sensibles al contexto), contiene a los otros dos y así sucesivamente.

Antes de enunciar cada uno de estos grupos definiremos una serie de símbolos que emplearemos más adelante:

$$\alpha, \beta, \delta \in (N \cup \Sigma)^*$$
$$\gamma \in (N \cup \Sigma)^+$$
$$A, B \in N$$
$$a \in \Sigma$$

Los lenguajes recursivamente enumerables o de tipo 0, son todos aquellos reconocibles por una máquina de Turing y comprenden a todos los posibles lenguajes descritos por una gramática formal, sin ningún tipo de restricción, por lo tanto sus producciones son de la forma:

$$\alpha A \beta \rightarrow \delta$$

Los lenguajes sensibles al contexto o de tipo 1, son todos aquellos reconocibles por un autómata linealmente acotado, las gramáticas que definen estos lenguajes no pueden contener reglas compresoras, es decir, todas sus producciones deben ser de la forma:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

Los lenguajes independientes del contexto o de tipo 2, son todos aquellos reconocibles por un autómata de pila, la gran mayoría de lenguajes de programación forman parte de este grupo. Las producciones de las gramáticas que definen estos lenguajes solo pueden tener un símbolo no terminal en la parte izquierda, por lo tanto sus producciones son de la forma:

$$A \rightarrow \delta$$

Por último los lenguajes regulares o de tipo 3, son los lenguajes más simples, son todos aquellos que pueden ser definidos por una expresión regular. Las producciones de las gramáticas que definen estos lenguajes solo pueden tener un símbolo no terminal en la parte izquierda y un símbolo terminal, un terminal y uno no terminal o la cadena vacía en la derecha, por lo tanto sus producciones tienen la forma:

$$A \rightarrow \varepsilon$$
$$A \rightarrow a$$
$$A \rightarrow aB$$

Concretamente nuestro trabajo se va a centrar en los lenguajes independientes del contexto o de tipo 2.

2.3 Aprendizaje automático de lenguajes

Cuando hablamos de inferencia gramatical hacemos referencia al conjunto de técnicas que tienen como finalidad obtener una gramática objetivo a partir de una serie de muestras que pueden presentarse de distintas formas, textos, palabras etiquetadas, conjuntos de palabras relacionadas...

Esta gramática objetivo debe describir fielmente el lenguaje que pretendemos modelizar, pero es posible que esa gramática no exista, por lo tanto nuestro algoritmo debe buscar la representación más fiel posible de los datos usados para describirlo.

Existen cuatro paradigmas principales del aprendizaje, el deductivo, inductivo, por analogía y por referencia. La inferencia gramatical se encuentra dentro del grupo del aprendizaje inductivo, que trata de encontrar leyes generales a partir de muestras de información específica, en concreto dentro del subgrupo de la inferencia inductiva, en la cual el entorno proporciona los ejemplos y contraejemplos necesarios.

La inferencia gramatical presenta cuatro propiedades básicas:

2.3.1. Concepto

El concepto es el método empleado para asignar una hipótesis a cada objeto, este puede aceptor, si el algoritmo genera autómatas o generador, si se generan gramática. En nuestro caso será generador, pues intentaremos encontrar una gramática formal que genere únicamente las palabras del lenguaje objetivo.

2.3.2. Protocolo de la información

El protocolo, es el modo en que la información empleada para el entrenamiento será proporcionada. El protocolo puede ser de dos tipos, texto, si recibe únicamente cadenas positivas o informante, si recibe tanto cadenas positivas como negativas, etiquetadas. En nuestro caso será informante de tipo arbitario, pues las cadenas proporcionadas al sistema no tienen porque seguir ningún orden.

2.3.3. Criterio de éxito

Existen dos criterios de éxito principales, que son:

- Identificación en el límite, propuesto por E. Mark Gold en [Gol67], es un modelo que recibe como entrada una presentación (una sucesión de cadenas), el aprendizaje se ve como un proceso infinito, en el que el aprendiz, va procesando una por una las cadenas de entrada y para cada una devuelve una gramática que trata de representar el lenguaje. Eventualmente, en un número finito de pasos, el aprendiz devolverá la gramática correcta, aunque no podrá necesariamente detectar su correctitud.
- Aprendizaje correcto probablemente aproximado, propuesto por L. G. Valiant en [Val84], es un modelo que recibe una serie de muestras como entrada, para las que devolverá una generalización tratando de aumentar la probabilidad de que el error sea bajo.

En nuestro caso particular, no existe de momento ninguna demostración formal de cual es el criterio de éxito del modelo que vamos a emplear, pero existen trabajos en curso intentando despejar esta incógnita.

2.3.4. Método de inferencia

El método de inferencia, es el tipo de procedimiento que va a seguir el algoritmo para encontrar la mejor hipótesis, y puede ser enumerativo, cuando el algoritmo solo recorre una serie de hipótesis predefinidas en busca de la hipótesis objetivo o constructivo, cuando el algoritmo construye la nueva hipótesis a partir de la información recibida.

Dentro de los métodos constructivos están los métodos incrementales, cuando la nueva hipótesis se construye a partir de la anterior y de la nueva información recibida. Y los métodos no incrementales, en los que la nueva hipótesis se construye cada vez, a partir de toda la información recibida hasta el instante actual.

El modelo estudiado en este trabajo empleará un método de inferencia constructivo incremental.

2.4 Dificultades de las gramáticas independientes del contexto

Por último cabría destacar las dificultades que surgen a la hora de trabajar con estas gramáticas.

Para empezar, la equivalencia entre gramáticas independientes del contexto es un problema indecidible, como explica John E. Hopcrof en [HMU01], por lo que nos va a ser imposible crear un algoritmo que compruebe si una de las gramáticas que hemos generado es la objetivo o no. Esta circunstancia nos llevará a realizar aproximaciones para poder evaluar de la mejor manera posible las gramáticas intentando a su vez mantener la eficiencia del programa.

Otro problema que podría surgir, es el hecho de que el tamaño de las palabras de los lenguajes independientes del contexto puede ser exponencialmente más grande que el tamaño de la propia gramática, por ejemplo, fijémonos en la gramática:

$$N_0 \rightarrow N_1 N_1$$

$$\ldots$$

$$N_i \rightarrow N_{i+1} N_{i+1}$$

$$N_n \rightarrow a$$

En este caso, el lenguaje tendrá una única palabra de tamaño 2^{n-1} siendo n el número de producciones de la gramática. Esto es problemático, pues en lenguajes de estas características las muestras pueden ser excesivamente grandes, con costes computacionales altísimos en consecuencia. En nuestro caso intentaremos evitar estos lenguajes en las pruebas, seleccionando a mano previamente los lenguajes a utilizar, aunque como veremos en el capítulo 6 no siempre será posible.

Por último cabría destacar el problema de la inteligibilidad, un mismo lenguaje puede tener una infinidad de gramáticas independientes del contexto que lo describen, algunas de las cuales se acoplarán mejor al lenguaje y otras peor, esto puede desembocar en problemas de ineficiencia y alto coste computacional, porque la gramática generada, pese a ser correcta, es excesivamente compleja, para el lenguaje objetivo, existiendo alternativas mejores.

CAPÍTULO 3
Algoritmos genéticos

Existen distintas técnicas de inferencia gramatical, basadas en distintos enfoques, de todas ellas, nosotros nos vamos a centrar concretamente en los algoritmos genéticos, pese a que más adelante en el capítulo 4 ampliaremos este método combinándolo con la computación con membranas. Gran parte de la información obtenida, acerca de este campo, está extraida de [RM00].

Los algoritmos genéticos pretenden, como su nombre indica, mejorar una población (en nuestro caso de gramáticas) mediante la modificación genética y mecanismos evolutivos propios de la naturaleza, que discutiremos en las siguientes secciones.

3.1 Codificación y decodificación

Una parte fundamental de los algoritmos genéticos son las funciones de codificación y decodificación, que nos permiten construir una cadena con toda la información de un objeto (su genoma) y a partir de la misma, reconstruir el objeto original.

Aunque en 5.3 y en 6.1 ya discutiremos las funciones empleadas en [Nis20] y en este trabajo respectivamente, vamos a proponer unas funciones de codificación y decodificación sencillas para ejemplificar lo explicado:

- A modo de función de codificación, podemos simplemente concatenar las cadenas de caracteres que representan cada una de las producciones de la gramática que deseamos codificar, eliminando las ocurrencias de ε, pues es la cadena vacía y empleando como separador el símbolo #. Por ejemplo, para codificar la gramática de la sección 2.1, podríamos hacer:

$$\left.\begin{array}{r} S \to aAb \\ A \to aAb \\ A \to \varepsilon \end{array}\right\} \Rightarrow "S \to aAb \# A \to aAb \# A \to "$$

- Para decodificarlo, dividimos el genoma empleando el símbolo # como separador, y si a uno de los dos lados de una producción está vacío, añadimos ε. Por ejemplo, para decodificar el genoma anterior:

$$"S \to aAb \# A \to aAb \# A \to " \Rightarrow \begin{cases} S \to aAb \\ A \to aAb \\ A \to \end{cases} \Rightarrow \begin{cases} S \to aAb \\ A \to aAb \\ A \to \varepsilon \end{cases}$$

Antes de pasar a la siguiente sección, cabe destacar que en todas las funciones de decodificación que abordamos en este trabajo, se asume que hay un símbolo inicial, conjunto de símbolos

terminales y no terminales conocidos, que no varían y podemos emplear después de la decodificación para generar la gramática, en caso contrario, esa información también debería ser incluida en el genoma por la función de codificación.

3.2 Inicialización

Para inicializar el sistema, se generará una población de un tamaño arbitrario predefinido n de genomas aleatorios o pseudo aleatorios. Normalmente, se intentará que estos genomas iniciales definan correctamente una gramática, por lo que serán pseudo aleatorios, pero también se podrían generar cadenas completamente aleatorias y luego filtrar los resultados eliminando las cadenas incoherentes.

3.3 Evaluación y selección

Las funciones de evaluación, determinará lo buenas o malas que son las gramáticas obtenidas en cada iteración del algoritmo, al igual que las de codificación y decodificación, las estudiaremos más adelante en 5.1 y en 6.1, pero plantearemos una versión simplificada para comprenderlo mejor.

La función de evaluación recibe una gramática ya decodificada y una palabra marcada como positiva o negativa, a partir del símbolo inicial de la gramática, y mediante derivaciones, hará una búsqueda en anchura, generando todas las palabras de la gramática, hasta encontrar la que busca o llegar a un límite máximo de palabras generadas, en cuyo caso buscará la palabra con más letras en común y devolverá como resultado c/n siendo c el número de letras en común y n el número de letras de la palabra de entrada.

Como podemos observar, en caso de que encuentre la palabra exacta, devolverá un valor de 1, mientras que conforme aumenta la diferencia disminuirá el valor devuelto hasta alcanzar el 0, cuando no encuentre ninguna palabra con coincidencias. A este valor de lo buena o mala que es la gramática para reconocer una palabra, lo llamaremos fitness.

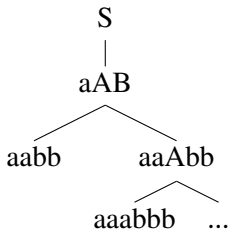

Figura 3.1: Árbol resultante de una búsqueda en anchura en L_1

Pero esta función de evaluación tiene dos claros inconvenientes, por un lado, el coste computacional es muy elevado, pues para cadenas muy grandes debe generar todas las de menor tamaño primero, y por otro lado, podría no llegar a generar la palabra que se está buscando porque llega al límite, obteniendo un mal fitness siendo que si que la reconoce. Esta problemática la abordaremos en las secciones comentadas anteriormente, analizando distintos enfoques.

La selección, por otro lado puede o no depender de la evaluación, en este paso, se seleccionará a parte de la población para mutarla y recombinarla. Esta selección puede ser aleatoria, escogiendo objetos al azar. Metódica, siguiendo algún patrón a la hora de seleccionar los objetos, como por ejemplo, los que tienen mejor fitness. O una mezcla de ambas, por ejemplo, la mitad serán los mejores y la otra mitad objetos al azar para aumentar la variabilidad.

3.4 Recombinación y cruce

Sobre los genomas seleccionados, realizaremos las siguientes operaciones:

- Recombinación: A partir de dos genomas, obtendremos dos genomas nuevos, cortando los dos originales por una posición arbitraria e intercambiando sus secciones.

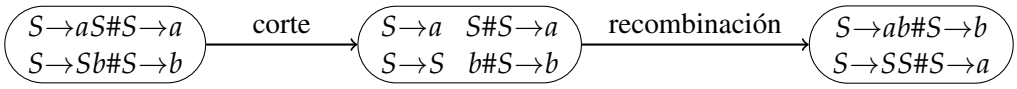

Figura 3.2: Ejemplo de recombinación

- Mutación: Se escoge un genoma o grupo de genomas y se les modifica uno o varios símbolos arbitrariamente.

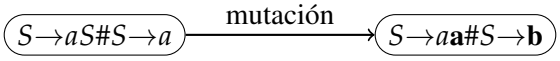

Figura 3.3: Ejemplo de mutación

Los puntos de corte, o los genomas mutados pueden ser arbitrarios o metódicos, depende del enfoque, al igual que comentamos en la sección 3.2, si se hace aleatoriamente, se corre el riesgo de que la modificación rompa la coherencia del genoma, por lo que en pasos posteriores se deberá filtrar la población para eliminar o al menos ignorar estos genomas defectuosos. Esto es lo que hacemos en nuestra implementación, mutamos símbolos arbitrarios, pero posteriormente puntuamos negativamente los genomas incoherentes, para poco a poco ir desechándolos.

3.5 Reemplazo

El último paso es el reemplazo, para este paso se puede realizar una segunda evaluación, incluyendo esta vez a los nuevos genomas, o emplear los resultados de la evaluación realizada antes de la selección. Siguiendo el procedimiento que se desee, se eliminarán los genomas sobrantes, para mantener el tamaño de la población constante.

En nuestro trabajo, empleamos la evaluación previa a la selección, eliminando los peores genomas y sustituyéndolos por los nuevos.

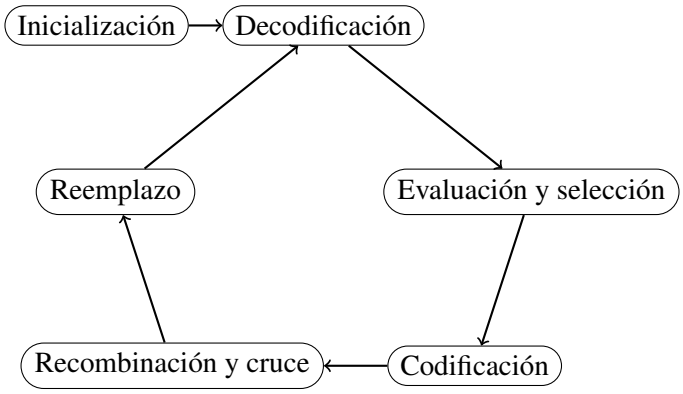

Figura 3.4: Etapas de un algoritmo genético

Una vez definidas todas las etapas del algoritmo, podemos construirlo, como se puede observar en la figura 3.4. Solo quedaría destacar que el algoritmo puede parar tras un número definido de iteraciones (el método que nosotros emplearemos) o cuando se obtiene un alto valor de fitness.

CAPÍTULO 4
Computación con membranas

A continuación, en las subsiguientes secciones, explicaremos los fundamentos de la computación natural, más específicamente de la computación con membranas o Sistemas P. Para posteriormente discutir su función en este trabajo y sus posibles sinergias con la inferencia gramatical y los algoritmos genéticos.

4.1 Nociones básicas

La computación natural, es un termino que engloba todos los sistemas computacionales que:

- Se inspiran en la naturaleza, para desarrollar métodos de solución de problemas, por ejemplo, las redes neuronales artificiales, que imitan a las redes neuronales humanas.
- Hacen uso de elementos de la propia naturaleza para realizar un cómputo, como la computación con ADN, descrita y probada por Leonard M. Adleman en [Adl98].

Nuestro campo de estudio, la computación con membranas, se encuentra en esta primera definición, pues citando a Gheorghe Păun en [Pau00] y [Pau02]:

> *Introducimos un nuevo modelo de computabilidad, de tipo paralelo distribuido, basado en la noción de estructura de membrana. Dicha estructura está formada por varias membranas de tipo celular, colocadas de forma recurrente dentro de una única membrana "piel".*

Como comenta Păun en este artículo, en el que propone el modelo de computación con membranas, la idea es crear un modelo computacional distribuido y paralelo, basado en las estructuras de membranas de las células.

Más en profundidad, la idea consiste en tener una estructura de membranas, representable como un diagrama de Venn sin intersecciones, con una única membrana que engloba al resto, a la que llamaremos piel. Cada membrana define una región donde podemos encontrar objetos, que representarían a los elementos químicos, aminoácidos, etc. Estos objetos pueden evolucionar, en función de una serie de reglas propias de cada membrana, una vez no se puedan aplicar más reglas, la computación ha terminado y el resultado se extrae del estado de una membrana previamente definida, conocida como membrana de salida.

Formalmente, un sistema P de transición de grado $n, n \geq 1$ es una construcción de la forma:

$$\Pi = (V, \mu, w_1, ..., w_n, (R_1, p_1), ..., (R_n, p_n), i_0)$$

donde:

1. V es un alfabeto cuyos símbolos serán los objetos.

2. μ es una estructura de membranas de grado n, con cada membrana etiquetada con un valor único, nosotros utilizaremos las etiquetas $1, 2, ..., n$. Esto se representa como una cadena parentizada, siguiendo el ejemplo de la figura 4.1, $\mu = (_1(_2(_3)_3(_4)_4)_2)_1$. El conjunto de cadenas parentizadas puede definirse como el lenguaje generado por la gramática independiente del contexto:
$$S \rightarrow (S)S$$
$$S \rightarrow \varepsilon$$

3. $w_i, 1 \leq i \leq n$ son cadenas sobre V representando los multiconjuntos asociados a la región i de μ. Un multiconjunto se define como el par (A, m) donde A es un conjunto y $m : A \rightarrow \mathbb{N}$ es una función de A a $\mathbb{N}$ (los número naturales), de este modo podemos añadir la multiplicidad a los conjuntos, permitiendo que estos contengan varias ocurrencias del mismo símbolo.

4. $R_i, 1 \leq i \leq n$ es un conjunto finito de reglas sobre V asociadas a la región i de μ. p_i es el orden parcial entre las distintas reglas de R_i especificando un orden de prioridad si fuera necesario. Una regla es un par (u, v), normalmente expresado como $u \rightarrow v$, donde u es una cadena sobre V y $v = v'$ o $v = v'\delta$, donde v' es una cadena sobre
$$(V \times \{here, out\}) \cup (V \times \{in_j \mid 1 \leq j \leq n\})$$

5. i_0 es un número entre 1 y n, que indica cual es la membrana de salida.

Este sistema, como demuestras Gheorghe Păun es capaz de reconocer los lenguajes recursivamente enumerables, con configuraciones sencillas de dos o tres membranas. Para observar la potencia del modelo y poder comprender de manera más intuitiva su funcionamiento, vamos a proceder con un ejemplo:

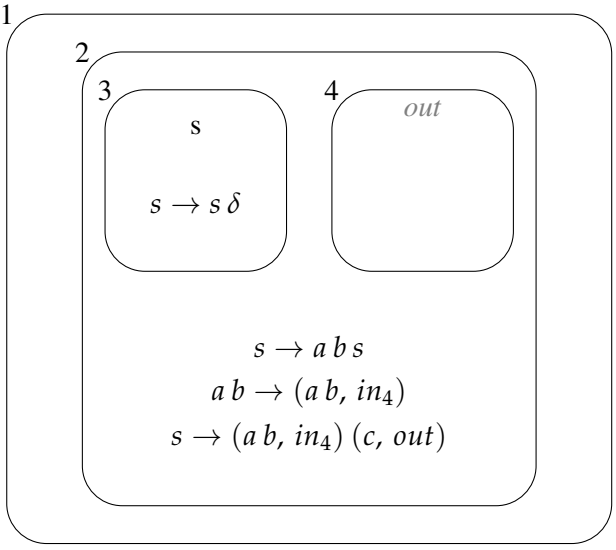

Figura 4.1: Ejemplo de sistema P

Antes de empezar a analizar el ejemplo que se muestra en la figura 4.1, cabe advertir, que no pretende ser eficiente, sino un reflejo de las funcionalidades principales de los sistemas P.

Lo primero que vemos es la estructura de membranas con una piel (membrana 1) que solamente contiene una membrana (2) que a su vez envuelve a dos sub membranas (3 y 4), una de las cuales está marcada como membrana de salida (4).

El único objeto que hay, se encuentra en la membrana 3 y es una s, en esta membrana también encontramos una regla, que nos indica que un objeto s puede transformarse en el mismo más δ, este nuevo símbolo, es lo que llamamos un objeto de disolución, cuando se produce este símbolo, se disuelve la membrana en la que se encuentra, destruyendo las reglas de la misma y enviando sus objetos a su membrana padre, en este caso a la membrana 2. Es importante remarcar que δ no puede aparecer en la región de la piel.

En la membrana número 2 se encuentran el resto de reglas:

- «$s \rightarrow a\ b\ s$», en esta regla, a partir de una s, mantenemos la s y generamos a y b.
- «$a\ b \rightarrow (a\ b,\ in_4)$», en esta regla, enviamos una a y una b a la membrana 4.
- «$s \rightarrow (a\ b,\ in_4)\ (c,\ out)$», en esta última regla, dada una s, enviamos una a y una b a la membrana 4 y una c a la membrana padre de la membrana donde se encuentra la regla, la 1. Perdiendo la s en el proceso.

Las reglas se aplican de un modo similar a las producciones de una gramática formal, pero operando sobre multisets o multiconjuntos.

De este modo, una regla de una membrana, se puede aplicar si su parte izquierda está presente en dicha membrana, eliminando esos objetos, generando los de su parte derecha y enviándolos a su membrana padre o a alguna de sus submembranas si así se especifica en la regla (como se puede ver en las dos últimas reglas de la membrana 2). Por este mismo motivo, podrían activarse distintas reglas de manera simultanea, si algunas de estas reglas consumen los mismos objetos, se aplicarán de manera indeterminista.

Si se deseara evitar esto último, o al menos poder controlarlo en cierta medida, también existe la posibilidad de establecer prioridades para las reglas, de modo que si se pueden aplicar dos reglas y una tiene prioridad sobre la otra, siempre se aplicará la más prioritaria.

Una vez entendidos y definidos todos los componentes de un sistema P, veamos un ejemplo de su ejecución en la figura 4.2.

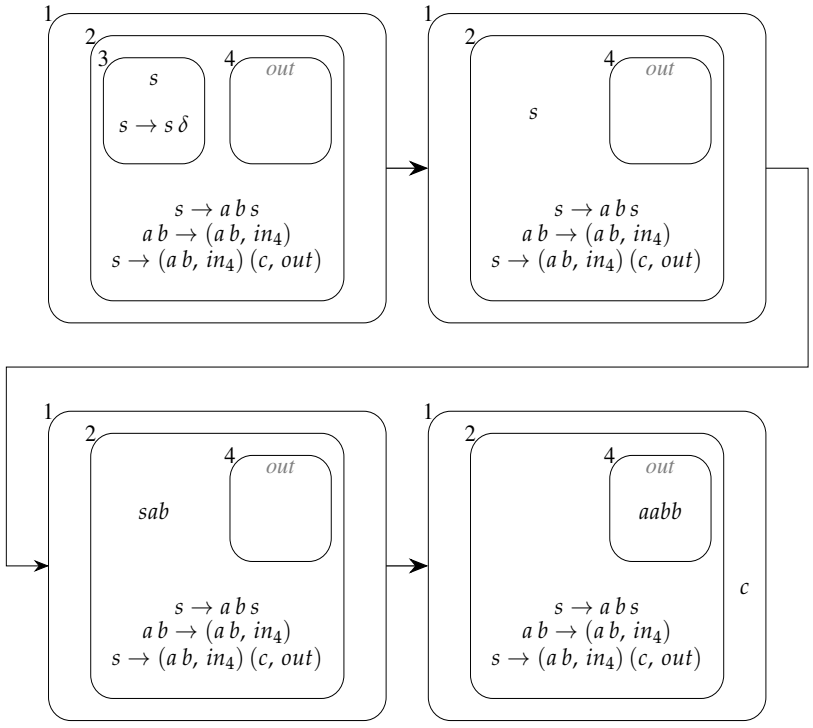

Figura 4.2: Ejemplo de transiciones de un sistema P

- En el primer paso, la única regla que puede aplicarse es la de la membrana 3, «$s \rightarrow s\,\delta$», por lo tanto, se mantiene la s, se genera δ y esto disuelve la membrana 3, enviando a s a la membrana 2.
- En el segundo paso pueden aplicarse dos reglas de la misma membrana sobre el mismo objeto, «$s \rightarrow a\,b\,s$» y «$s \rightarrow (a\,b,\ in_4)\,(c,\ out)$», por lo tanto, de manera indeterminista se aplicará una de las dos reglas, en nuestro caso escogemos la primera, manteniendo a s y generando a y b.
- En este último paso, se pueden aplicar las dos mismas reglas sobre s, pero esta vez aplicamos la segunda, enviando una c a la membrana 1, y una a y una b a la membrana 4. Y simultáneamente, se aplica la regla «$a\,b \rightarrow (a\,b,\ in_4)$», enviando una a y una b a la misma membrana.
- Llegados a este punto ya no es posible aplicar ninguna reglas, así que llegamos a un estado de parada, y el resultado es el multiset $\{a, a, b, b\}$.

En el caso del ejemplo anterior, en función del número de veces que se aplique la regla «$s \rightarrow a\,b\,s$» en lugar de su alternativa, determinará el número de aes y bes que aparezcan en el multiset de salida, siendo n el número de veces que se aplica la regla, se generarán $n+1$ aes y $n+1$ bes.

4.2 Variantes de sistemas P

Ahora que comprendemos el funcionamiento básico de un sistema P, es hora de ahondar un poco más en el mundo de la computación con membranas, pues existen distintos tipos de sistemas P, como las alternativas propuestas en el artículo original de Gheorghe Păun en [Pau00] o los discutidos en [Son+21]. En esta sección vamos a analizar las principales propuestas y sus características.

- **Sistemas P de transición:** Los sistemas P de transición son la variante más básica de sistemas P, es la versión original propuesta por Păun y la que hemos explicado en la sección anterior.
- **Sistemas P basados en reescritura:** Los sistemas P de transición pueden ser entendidos como sistemas carentes de estructuras de datos, los resultados se expresan de manera numérica, en la cardinalidad de los multisets. Esto puede ser más fiel a la realidad desde un punto de vista bioquímico pero es ineficiente desde un punto de vista clásico.

  Para solucionarlo surgen los sistemas P basados en reescritura, en los que la principal diferencia radica en el uso de cadenas para sustituir a los objetos de los sistemas P de transición y sustituir las reglas por reglas de reescritura, muy similares a las producciones de una gramática formal, pero añadiendo los símbolos in, out y δ. En cuyo caso la cadena modificada, se transporta a la membrana especificada, o en el caso de δ, se diluye la membrana actual. Un ejemplo de regla de este tipo sería:

  $$S \rightarrow aSb(out)$$

- **Sistemas P de recombinación:** Siguiendo con la idea de emplear cadenas en lugar de objetos, surgen los sistemas de recombinación, en este caso cambiando las reglas de reescritura por reglas de recombinación o splicing, que tendrán la forma $r = u_1\#u_2\$v_1\#v_2$ de manera que esta regla se aplicará sobre x e y generando w si y solo si:

  $$x = x_1u_1u_2x_2$$

$$y = y_1 v_1 v_2 y_2$$

$$w = x_1 u_1 v_2 y_2$$

Esto se denota $(x, y) \vdash_r w$ y si nos fijamos, es el mismo comportamiento de las operaciones de recombinación del ADN, de las que ya hemos hablado en la sección 2.3.

- **Sistemas P con membranas activas:** Inspirado en el punto de vista biológico y matemático surgen los sistemas P con membranas activas, estos destacan especialmente por la capacidad de modificar su estructura de membranas durante la computación.

  Para adaptar el sistema a esta nueva capacidad, se extraen las reglas de las membranas y se definen dos nuevas propiedades de las mismas, la etiqueta y la carga o polaridad. Ahora las reglas son comunes a todas las membranas, pero cada regla tendrá una etiqueta y una carga, y solo las membranas que compartan ambas propiedades con la regla podrán aplicarla.

  Además, se añaden las reglas para dividir membranas, cambiar sus polaridades y cambiar sus etiquetas.

- **Sistemas P basados en comunicación:** Estos sistemas son una simplificación de los sistemas P de transición, pues no permiten reglas de evolución para los objetos y toda la computación se hace mediante transmisiones de objetos entre las membranas.

- **Sistemas P en tejido con reglas de transporte e intercambio:** Los sistemas P en red o tejido, consisten en un conjunto de sistemas P independientes, conectados entre ellos por canales de comunicación, en este caso el canal de comunicación incluye reglas de transporte e intercambio, que básicamente permiten enviar multisets de un sistema a otro o intercambiar multisets entre dos sistemas.

- **Sistemas P con proteínas en sus membranas:** Esta variante de los sistemas P comparte similitudes con los sistemas P con membranas activas, en la capacidad de marcar sus membranas y condicionar la aplicación de sus reglas a esto, solo que en este caso, en lugar de emplear etiquetas y cargas, se emplean proteínas, que funcionan como objetos asociados a las membranas, que también pueden ser afectados por sus reglas.

- **Sistemas P neuronales con estímulos:** Siendo las células encargadas del "computo" en el cuerpo humano, y estando trabajando con células para realizar cálculos, lo lógico sería tratar de simular neuronas, y de ese intento surgen los sistemas P neuronales con estímulos.

  Esta versión de los sistemas P tiene lógicamente, mucho en común con las redes neuronales artificiales, disponemos de una serie de membranas o neuronas, conectadas entre ellas por sinapsis, a través de las cuales, enviarán estímulos o spikes, que provocarán la aplicación de reglas en las neuronas destino, propagando el computo.

4.3 Sistemas P en tejido y sinergias con algoritmos genéticos

Teniendo en cuenta que las células trabajan en tejidos, lo natural sería imitar este comportamiento para nuestros modelos computacionales, esta idea es propuesta en [Mar+03], pues citando al artículo original:

> *Partiendo de la forma en que se produce la comunicación intercelular por medio de canales de proteínas (y también del conocimiento estándar sobre el funcionamiento de las neuronas), proponemos un modelo de computación denominado sistema P en tejido, que procesa símbolos en un sentido de reescritura de multiconjuntos, en una red de células.*

Definiendo formalmente este sistema, tenemos que, un sistema P en tejido, de grado $m \geq 1$, es una construcción de la forma

$$\Pi = (O, \sigma_1, ..., \sigma_m, \text{syn}, i_{\text{out}}),$$

donde:

- O es un alfabeto no vacío de objetos.
- syn $\subseteq \{1, 2, ..., m\} \times \{1, 2, ..., m\}$ (Representando las sinapsis entre las celulas).
- $i_{\text{out}} \in \{1, 2, ..., m\}$ indicando la célula de salida.
- $\sigma_1, ..., \sigma_m$ son células de la forma:

  $$\sigma_i = (Q_i, s_{i,0}, w_{i,0}, P_i), 1 \leq i \leq m,$$

  donde:

  - Q_i es un conjunto finito de estados.
  - $s_{i,0} \in Q_i$ es el estado inicial.
  - $w_{i,0} \in O^*$ es el multiset inicial de objetos.
  - P_i es un conjunto finito de reglas de la forma $sw \rightarrow s'xy_{\text{go}}z_{\text{out}}$, donde $s, s' \in Q_i$, $w, x \in O^*$, $y_{\text{go}} \in (O \times \{\text{go}\})^*$ y $z_{\text{out}} \in (O \times \{\text{out}\})^*$, con la restricción de $z_{\text{go}} = \lambda$ para todo $i \in \{1, 2, ..., m\}$ diferente de $i_t extout$.

Un sistema P de transición se dice que es cooperativo si contiene como mínimo una regla $sw \rightarrow s'w'$ tal que $|w| > 1$, y no cooperativo en caso contrario. Los objetos que aparecen en el multiconjunto w son llamados impulsos y los que aparecen en w' son llamados excitaciones. Veamos el siguiente ejemplo para comprenderlo todo mejor:

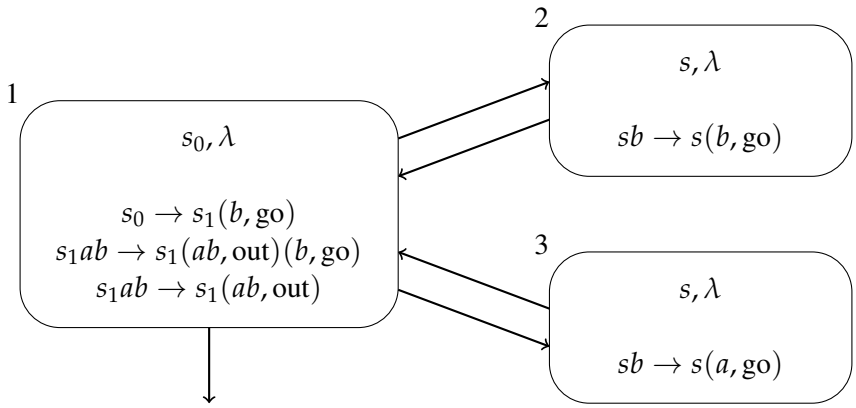

Figura 4.3: Ejemplo de sistema P en tejido

En la figura 4.3, podemos observar un tejido compuesto por tres células:

1. La célula de salida, empieza vacía en un estado inicial s_0 que tras aplicar la única regla posible «$s_0 \rightarrow s_1(b, \text{go})$» pasará al estado s_1, esta regla, aparte de cambiar el estado enviará un objeto b a las otras dos células.

   Por otro lado, la regla «$s_1ab \rightarrow s_1(ab, \text{out})(b, \text{go})$», en el estado s_1, dados los objetos ab, enviará ambos objetos al entorno y una b a las otras dos células.

   Por último la regla «$s_1ab \rightarrow s_1(ab, \text{out})$», en las mismas condiciones que la regla anterior (estado s_1 y dados los objetos ab), enviará ab al entorno, pero ningún símbolo a las otras dos células, terminando así con la ejecución, pues no se podrá aplicar ninguna regla más.

2. Esta célula empieza vacía y en el estado s, el célula no variará. Solo contiene una regla «$sb \rightarrow s(b, \text{go})$», que se aplicará al recibir el objeto b por parte de la célula 1, devolviendo el mismo objeto a dicha célula sin realizar ningún cambio en su estado.

3. Al igual que la célula anterior, esta empieza vacía y en el estado s, el célula no variará. Contiene únicamente la regla «$sb \rightarrow s(a, \text{go})$», que al igual que en (2) se aplicará al recibir el objeto b por parte de la célula 1, pero en este caso, devolverá un objeto a la célula (1) sin realizar ningún cambio en su estado.

A partir de la descripción anterior, podemos deducir, que este sistema P en tejido, generará el multiset $a^n b^n$ para $n \geq 1$, siendo n el número de veces que se ejecuta la regla «$s_1 ab \rightarrow s_1(ab, \text{out})(b, \text{go})$». Coincidiendo con el sistema P descrito en la sección 4.1.

Concretamente, en este trabajo, emplearemos un sistema P en tejido, pero en lugar de reglas de reescritura emplearemos reglas de recombinación y mutación, pues nuestro tejido trabajará con genomas, más en concreto trabajará con un grupo de genomas que codifican gramáticas formales, de este modo, cada célula tratará de optimizar sus genomas para generar la gramática más fiel posible al objetivo, y enviará sus mejores candidatos a la célula de salida. Los mecanismos empleados para esto y la forma exacta de su implementación serán discutidos más adelante en el capítulo 6.

CAPÍTULO 5
Trabajo original de Taishin Y. Nishida

Una vez entendidos los conceptos básicos de Inferencia gramatical y computación con membranas, llega la hora de abordar el artículo central sobre el que trata este trabajo, «Evolutionary p systems: The notion and an example» de Taishin Y. Nishida [Nis20]. En este artículo, se estudia la eficacia de distintos modelos de computación con membranas a la hora de inferir gramáticas incontextuales. En las siguientes secciones analizaremos el artículo, para posteriormente centrarnos en el modelo que vamos a estudiar.

5.1 Modelos

En este artículo, se evalúan dos modelos y en cada modelo se prueban dos funciones fitness diferentes. Estas dos funciones son:

- La función NE1 (negative 1), hace uso de la distancia de Hamming, que mide la distancia entre dos cadenas. Siendo $(w)_i$ el i-esimo símbolo de una cadena, la distancia de Hamming se define como:

$$d(w_t, w_g) = \sum_{i=1}^{s} \delta((w_t)_i, (w_g)_i) + \frac{1}{2}||w_t| - |w_g||$$

$$\delta(x, y) = \begin{cases} 0, & x = y \\ 1, & x \neq y \end{cases}$$

  Dada la distancia de Hamming, una cadena muestra w_t y una cadena generada con la gramatica formal codificada por el genoma a evaluar, mediante el algoritmo de parsing $LL(1)$ w_g, el fitness de dicho genoma será:

$$fitness(w_t, w_g) = \begin{cases} \frac{3}{2}|w_t| - d(w_t, w_g), & \text{si } w_t \text{ es positiva} \\ d(w_t, w_g), & \text{si } w_t \text{ es negativa} \end{cases}$$

  El algoritmo de parsing $LL(1)$ es un analizador sintáctico descendente, de derivación a izquierdas. Este algoritmo genera una tabla de análisis a partir de la cual, trata de encontrar el orden de aplicación de las producciones de una gramática independiente del contexto, partiendo del símbolo inicial, para generar la cadena objetivo. Para más información acerca de este algoritmo y sus aplicaciones consultar [AU07].

- La otra función de fitness empleada es IM (immune-like evaluation), y como su nombre indica, copia su comportamiento del sistema inmunitario. Este comportamiento es idéntico al de NE1, pero si la cadena objetivo es negativa, y la distancia resulta 0, esa gramática es automáticamente destruida.

Estos dos métodos son probados en los siguientes dos modelos:

- **Cell-like**, este modelo se compone de una única célula, con una única membrana piel, que contiene todos los genomas juntos, tras cada etapa de entrenamiento, la mejor gramática de la membrana es enviada a una membrana de salida donde no se realiza ninguna operación sobre los mismos.
- **Tissue**, un modelo de tejido, con n sistemas P conectados a un sistema central, que será el de salida (5.1). En cada una de las membranas "exteriores" se encuentran conjuntos de genomas que evolucionarán paralelamente al resto, pero sin contacto entre ellos, esto permite que se desarrollen distintos "enfoques" simultáneamente. Al igual que en el modelo Cell-like, al final de cada etapa de entrenamiento la mejor gramática de cada membrana es enviada a la membrana de salida.

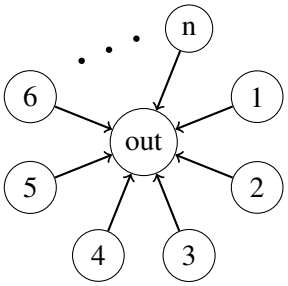

Figura 5.1: Ejemplo de modelo basado en sistemas P en tejido

El modelo que exploraremos en este trabajo, será el sistema-P en tejido con una variante de la evaluación NE1, y una serie de modificaciones, sobre las que veremos más en el siguiente capítulo.

5.2 Proceso de entrenamiento

El proceso de entrenamiento empleado por Taishin Y. Nishida, consiste primero, en generar en cada sistema del modelo, una población de genomas, que codifican gramáticas formales, para después aplicar los siguientes pasos, durante un número arbitrario de operaciones:

- Primero se genera una muestra, que consistirá en una cadena etiquetada como positiva, si pertenece al lenguaje objetivo, o negativa, si no pertenece.
- Posteriormente se evalúan los genomas presentes en las membranas, en base a esa muestra, empleando una función de fitness, que le asigna una puntuación a la gramática codificada por cada genoma, en función de su capacidad para predecir correctamente la muestra. Las funciones empleadas, son las comentadas en la sección 5.1.
- Una vez evaluados los genomas, se seleccionan los que han obtenido mejor fitness, destruyendo al resto.
- Estos genomas supervivientes, son recombinados y mutados, para repoblar la membrana.
- Y por último, el mejor genoma de cada sistema es enviado al sistema de salida.

5.3 Codificación y decodificación

Para poder codificar las gramáticas, se duplicarán los símbolos no terminales, añadiéndoles un marcador, para posteriormente concatenar todas las producciones, de este modo, una gramática se codificaría de la siguiente forma:

$$\left.\begin{array}{r} S \to aB \\ B \to Sb \\ B \to b \end{array}\right\} \Rightarrow \left.\begin{array}{r} S \to \bar{a}\bar{B} \\ B \to \bar{S}\bar{b} \\ B \to \bar{b} \end{array}\right\} \Rightarrow S\bar{a}\bar{B}B\bar{S}\bar{b}B\bar{b}$$

De este modo la decodificación es trivial, pues simplemente hay que separar el genoma en grupos de caracteres continuos sin marcar y marcados.

CAPÍTULO 6
Implementación y experimentación

Partiendo del modelo de tejido con NE1, vamos a realizar una serie de modificaciones tanto al método empleado como al modelo, para acercarlo más a los frameworks de experimentación "tradicional" de inteligencia artificial, y de este modo poder estudiar sus parámetros.

Posteriormente detallaremos las peculiaridades de su implementación y realizaremos una experimentación para ver los resultados.

6.1 Modificaciones

La primera modificación realizada es cambiar la codificación de los genomas. En lugar de marcar caracteres, codificaremos gramáticas en forma normal de Chomsky, esta forma solo reconoce dos tipos de producciones:

$$A \to BC$$

$$A \to a$$

La codificación se hará concatenando los símbolos de las producciones, de la siguiente forma:

$$\left.\begin{aligned} S &\to AC \\ A &\to a \\ C &\to SB \\ C &\to b \\ B &\to b \end{aligned}\right\} \Rightarrow SACAaCSBCbBb$$

La decodificación es trivial, pues empleando la cadena del genoma como una pila, se puede elaborar un algoritmo sencillo, como el de la figura 6.1.

Otra modificación importante realizada, es un cambio en el método de evaluación de las gramáticas, el método empleado por Taishin Y. Nishida supone emplear una variante del algoritmo $LL(1)$ el cual no funciona para cualquier gramática y además es indeterminista, por lo que el fitness de una gramática puede variar en función del azar. Por lo tanto nosotros utilizaremos una modificación del algoritmo CYK, que pese a ser más costoso computacionalmente, puede evaluar todo tipo de gramáticas y los resultados no pueden variar.

El algoritmo CYK, construye una tabla con el fin de determinar si una cadena es generada o no por una gramática, si al acabar el algoritmo, en la última fila de la tabla, se encuentra el símbolo inicial de la gramática, el resultado es afirmativo, en caso contrario negativo (ver figura

```
input: Stack genome
output: Set

res = ∅
while genome ≠ ∅
    r ← pop(genome)
    l1 ← pop(genome)
    if isTerminal(l1) then
        append(res, r → l1)
    else
        l2 ← pop(genome)
        append(res, r → l1l2)
    end if
end while
return res
```

Figura 6.1: Algoritmo de decodificación

6.2). Nosotros modificaremos este comportamiento, de manera que si el símbolo inicial no aparece en la última fila (n), pero aparece en la fila i-esima, el fitness será $\frac{i}{n}$.

```
input: Gramatica en forma normal de Chomsky G=(N, T, P, S), String w
output: Boolean

for i=1 to n
    V_i1 = {A : A → w_i ∈ P}
end for

for j=2 to n
    for i=1 to n-j+1
        V_ij = ∅
        for k=1 to j-1
            V_ij = V_ij ∪ {A : A → BC ∈ P, B ∈ V_ik, C ∈ V_{i+k,j-k}}
        end for
    end for
end for

return S ∈ V_1n
```

Figura 6.2: Algoritmo CYK

También cabe destacar, que las muestras empleadas constan de un 50 % de muestras positivas y un 50 % de muestras negativas, que dividiremos en conjuntos de entrenamiento y conjuntos de test. Para ello empleamos una variable *samplessize*, con ella configuramos el tamaño de los conjuntos de entrenamiento, se dividirán todas las muestras en n conjuntos de tamaño *samplessize*, con cada uno de estos conjuntos se entrenará un modelo y el resto se emplearán para evaluarlo.

Para terminar, también incluimos los conceptos de *batchsize* y *epochs*. El *batchsize* surge ante la idea de que si en cada iteración evaluamos con respecto a una única muestra, una gramática muy buena que solo falla esa muestra, dará un mal fitness y podría ser eliminada, mientras que una gramática terrible que solo acierta esa, dará un buen fitness y podría sobrevivir. Por ello, en cada iteración, se entrena con respecto a *batchsize* muestras, de este modo se pretenden suavizar estos posibles fallos.

Por otro lado tenemos el concepto de *epochs* o épocas, esto surge de la idea de emplear una misma muestra varias veces para el entrenamiento, en lugar de solo una, para ello realizamos

epochs pasadas completas a las muestras de entrenamiento. Este no es un concepto nuevo, sino que es una técnica ampliamente utilizada en el mundo del aprendizaje automático, sobretodo con redes neuronales, pretendiendo que el sistema pueda extraer más información con distintas pasadas a los mismos datos.

Para comprender mejor estos tres conceptos y su relación entre ellos, vamos a dar un ejemplo:

Pongamos que tengo 100 muestras, con un *samplessize* de 20, un *batchsize* de 2 y 3 *epochs*. En este caso, esas 100 muestras se dividirán en 5 grupos de 20 (por el *samplessize*), con cada uno de esos grupos de 20 entrenaremos un sistema distinto y emplearemos las otras 80 como conjunto de test, para después comparar sus resultados y obtener un valor medio. Para cada uno de estos sistema, dividiremos sus 20 muestras de entrenamiento en 10 grupos de 2 (por el *batchsize*), por lo que realizaremos 10 iteraciones, en cada una de las cuales se le proporcionarán 2 muestras al sistema, esas 10 iteraciones se realizarán 3 veces con las mismas muestras, pues son 3 *epochs*, por lo que nos quedan 30 iteraciones.

6.2 Implementación

La implementación se ha realizado en Python3 y de manera modularizada, con el fin de poder expandir las funcionalidades, por ello, la estructura del proyecto tiene la siguiente forma:

```
Project root
├── __init__.py
├── fitness.py
├── genome.py
├── grammar.py
├── main.py
├── tissue.py
└── tools
    ├── __init__.py
    ├── case_builder.py
    ├── latex_converter.py
    └── experiments_visualization.py
```

De este modo, cada módulo controla una funcionalidad determinada, podemos observar por ejemplo, fitness.py que controla todo lo relacionado con la evaluación de los sistemas, o genome.py que se encarga de los procesos de mutación y recombinación de los genomas. Esta estructura permite modificar el funcionamiento de un único módulo, sin que esto afecte al resto de la aplicación.

Por otro lado, podríamos dividir las funcionalidades del simulador en dos grandes grupos:

- El simulador propiamente dicho, accesible desde una aplicación de consola, que debe recibir como entrada dos ficheros de configuración .json, el primero se encarga de definir los parámetros del experimento, como el número de sistemas, número de genes por sistema, epochs, número de mutaciones, número de símbolos terminales por gramática, número de no terminales, etc. Mientras el segundo proporciona las muestras que se emplearán para el entrenamiento y test. Separar estos dos ficheros, nos permite reutilizar los parámetros de un experimento con distintos datasets.

  A parte de estos dos ficheros, también se pueden emplear parámetros opcionales como la verbosidad o un parámetro para repetir cada experimento n veces con el fin de obtener resultados más fiables.

  Al terminar la ejecución, el simulador generará un fichero .json de resultados, donde se muestran los parámetros empleados, las gramáticas obtenidas y sus puntuaciones.

- Por otro lado tenemos las herramientas auxiliares, accesibles desde la misma aplicación de consola. En este módulo tenemos un visualizador, que recibe como entrada un fichero de resultados del simulador y muestra una gráfica con los resultados.

  Un generador de latex, que también recibe un conjunto de fichero de resultados del simulador y genera dos archivos de latex con una tabla y una gráfica respectivamente.

  Y por último tenemos el generador de casos. Dado que generar las muestras es un proceso costoso, no tiene sentido repetirlo en cada experimento, por ello desarrollamos el generador de casos, al que se le proporciona la gramática (en formato json) para la que se quieren generar los casos, el número de casos positivos y el número de casos negativos a generar. El programa generará un fichero json con los casos etiquetados, que utilizará posteriormente el simulador.

Si se desea profundizar más en el funcionamiento y uso del simulador, en el apéndice A se trata en mayor profundidad.

6.3 Resultados

A lo largo de esta sección, analizaremos los experimentos realizados, para ver la variación de la eficacia de este modelo en función de una serie de parametros. A modo de benchmark utilizaremos los siguientes tres lenguajes:

$$L_1 = \{a^n b^n | n \geq 1\}$$

$$L_2 = \{ww^R | w \in \{a,b\}^+\}, \text{ donde } w^R \text{ es la inversa de } w$$

$$L_3 = (\text{el lenguaje de Dyck sobre} \{a,b\})$$

Esto no es casual, pues emplearemos los mismos lenguajes empleados por Taishin Y. Nishida en [Nis20]. De este modo podremos comparar sus resultados con los nuestros, a fin de contrastarlos.

Para calcular lo acertado o erróneo que sea el resultado devuelto por el sistema, emplearemos la métrica accuray, cuyo cálculo es trivial, una vez obtenida la gramática veremos que muestras positivas y negativas, acepta y que muestras rechaza. Con estos datos la accuracy se calculará así:

$$\frac{(positivas - positivas_{aceptadas}) + (negativas - negativas_{rechazadas})}{positivas + negativas}$$

6.3.1. Tamaño del conjunto de entrenamiento

El primer experimento que vamos a analizar es la influencia del tamaño del conjunto de entrenamiento en la eficacia del modelo, podemos observar los resultados en 6.3 y 6.1.

Este experimento muestra claramente la limitación clave de este trabajo, el coste computacional. Como se puede ver en 6.3 para el lenguaje $a^n b^n$ se han utilizado conjuntos de entrenamiento de menor tamaño, esto es debido a que el lenguaje crece de manera lineal, pues la n-sima cadena tendrá un tamaño $2n$ y dado que el algoritmo empleado para evaluar las gramáticas funciona en base al algoritmo CYK (como vimos en 6.1), el coste de evaluar en función de una cadena w es de $|w|^3$ siendo $|w|$ la longitud de la cadena w, esto hace que el coste se dispare para cadenas largas, lo que acaba siendo un factor limitante.

Por otro lado también puede observarse claramente algo que ya destacó Taishin Y. Nishida en su trabajo [Nis20] y es la dificultad de inferir el lenguaje ww^r que empleando 500 muestras para su entrenamiento no es capaz de llegar a un accuracy del 0.6 mientras que el lenguaje $a^n b^n$ con tan solo 50 muestras llega al 0.8. Esto saca a relucir el hecho de que el modelo tiene facilidad/dificultad para inferir según que lenguajes.

Por último cabe destacar la influencia del tamaño de los conjuntos de entrenamiento:

- Para ww^r parece que cuanto mayor sea el conjunto mejor sera el accuracy.
- Mientras que la media de $a^n b^n$ sufre un pico en 50 muestras para posteriormente desplomarse en 75, esto podría deberse a haberlo sometido a muchas muestras muy diferentes, sobretodo teniendo en cuenta el rápido aumento del tamaño de las cadenas de $a^n b^n$ y esto ha debido dificultar el aprendizaje.
- Para acabar el lenguaje de Dyck parece sufrir una caida en las 300 muestras pero se recupera en las 500, esto puede deberse a la complejidad del lenguaje y que debido a esto unas muestras determinadas "confundan" al modelo, pero al añadir más se facilite su aprendizaje.

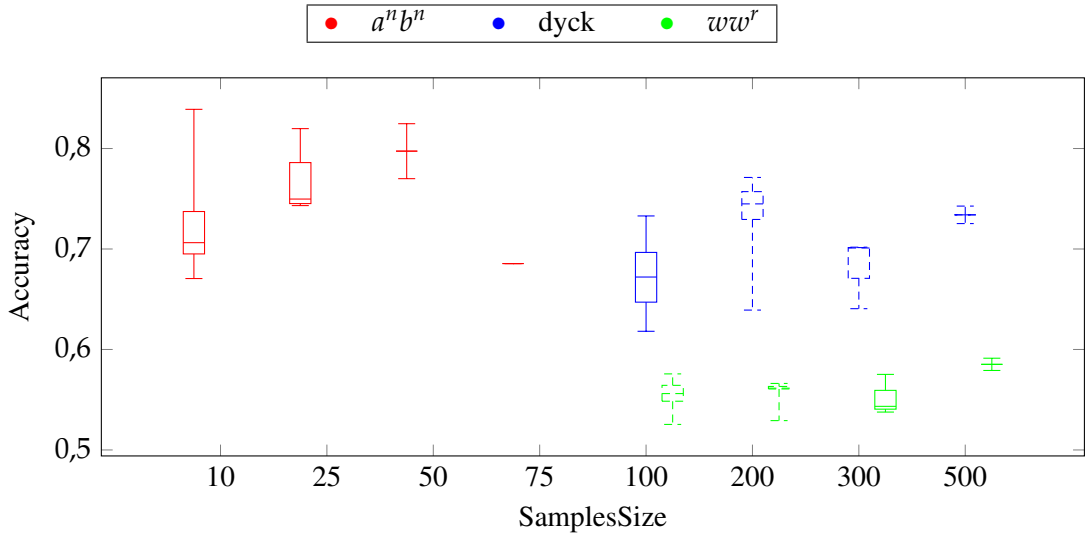

Figura 6.3: Gráfica de la influencia del tamaño del conjunto de entrenamiento en el accuracy

6.3.2. Tamaño de lote

A continuación estudiaremos la influencia del batch size o tamaño de lote, para el que pueden observarse los resultados en 6.4 y 6.2. Para este parámetro podemos observar cómo para cada lenguaje varía de una forma diferente:

- El aprendizaje de $a^n b^n$ parece no verse influenciado por el tamaño del lote, pues independientemente de su valor la precisión se mantiene estable.
- Por otro lado, el lenguaje de Dyck parece tener un máximo empleando un tamaño de lote de 1, para después desplomarse. Aunque parece que conforme el tamaño de lote aumenta, la precisión se va recuperando, es posible que para un mayor corpus de entrenamiento con mayores tamaños de lote, pudiéramos alcanzar un nuevo máximo.
- Por último el lenguaje ww^r muestra una clara tendencia a la baja conforme aumenta el tamaño de lote.

Lenguaje	SamplesSize	Mediana	Min.	Max.	Q_1	Q_3
$a^n b^n$	10	0.7063	0.6706	0.839	0.6951	0.7373
$a^n b^n$	25	0.7497	0.7432	0.8198	0.7452	0.786
$a^n b^n$	50	0.7974	0.77	0.8247	0.7974	0.7974
$a^n b^n$	75	0.6855	0.6855	0.6855	0.6855	0.6855
dyck	100	0.6721	0.6181	0.7329	0.6471	0.6966
dyck	200	0.7449	0.6392	0.7712	0.7294	0.7571
dyck	300	0.701	0.6406	0.7018	0.6708	0.7014
dyck	500	0.734	0.7253	0.7427	0.734	0.734
ww^r	100	0.5561	0.5254	0.5757	0.5485	0.5643
ww^r	200	0.5615	0.5292	0.5662	0.5607	0.5632
ww^r	300	0.5434	0.5377	0.5752	0.5406	0.5593
ww^r	500	0.5852	0.5791	0.5913	0.5852	0.5852

Tabla 6.1: Tabla de la influencia del tamaño del conjunto de entrenamiento en el accuracy

Este resultado es interesante, pues saca a relucir el hecho de que, dependiendo del lenguaje que deseemos inferir, los parámetros que optimicen el aprendizaje del modelo, pueden variar. Esto no suele ser algo deseable, pues se suele buscar siempre un sistema lo más generalista posible, pero esto nos abre un abanico de preguntas como: ¿Existe alguna configuración de parámetros que optimice el aprendizaje en general para cualquier lenguaje? ¿Existe algún patrón en los lenguajes que nos permita intuir sus parámetros ideales?

La respuesta a estas preguntas influirá de manera decisiva en la eficacia y eficiencia de este modelo.

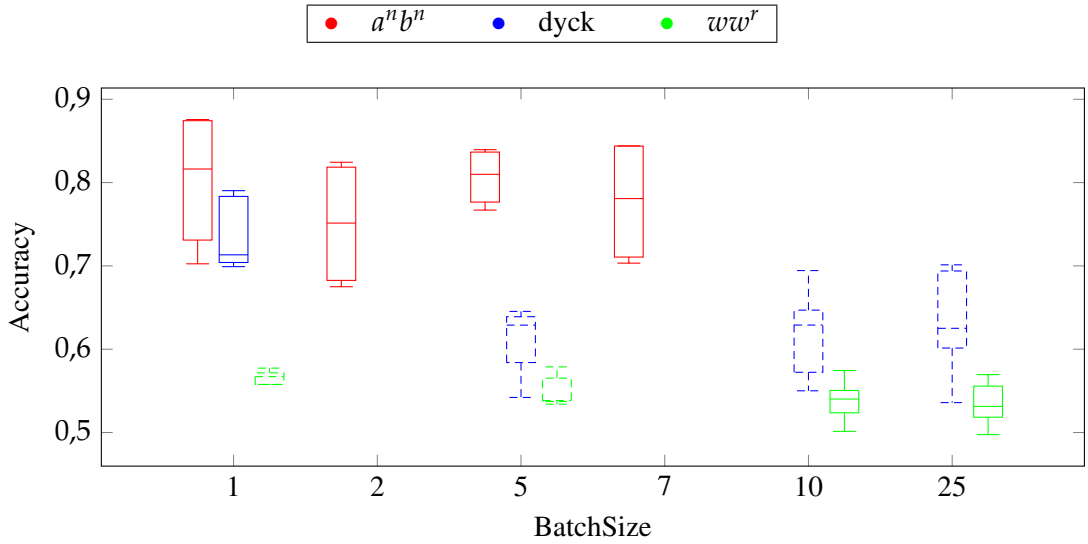

Figura 6.4: Gráfica de la influencia del tamaño de lote en el accuracy

6.3.3. Épocas

Otro parámetro ampliamente utilizado en el mundo de la inteligencia artificial son las epochs o épocas, cuya influencia hemos estudiado y reflejamos en 6.5 y 6.3.

El efecto de las épocas en la eficacia del modelo parece positivo en todos los lenguajes, y aunque para $a^n b^n$ parece ser negativo, tenemos ciertas sospechas de lo que está ocurriendo.

Lenguaje	BatchSize	Mediana	Min.	Max.	Q_1	Q_3
$a^n b^n$	1	0.8163	0.7025	0.8757	0.731	0.8743
$a^n b^n$	2	0.7514	0.675	0.8244	0.6826	0.8185
$a^n b^n$	5	0.8099	0.767	0.8395	0.7766	0.8366
$a^n b^n$	7	0.7808	0.7033	0.8441	0.7106	0.8438
dyck	1	0.7133	0.6991	0.7903	0.7041	0.7834
dyck	5	0.6289	0.5421	0.6454	0.584	0.639
dyck	10	0.629	0.5501	0.6944	0.5723	0.6469
dyck	25	0.6251	0.5359	0.7013	0.6014	0.6939
ww^r	1	0.5671	0.5577	0.5773	0.5577	0.5716
ww^r	5	0.5384	0.5341	0.5788	0.5373	0.5653
ww^r	10	0.5402	0.5014	0.5744	0.5237	0.5505
ww^r	25	0.5314	0.4975	0.5695	0.5184	0.5557

Tabla 6.2: Tabla de la influencia del tamaño de lote en el accuracy

- Para $a^n b^n$ parecen mejorar los resultado con 2 épocas, para después empeorar. Esto puede deberse al overfitting o sobre ajuste, dado que este lenguaje tiene un conjunto de entrenamiento reducido, las gramáticas pueden haberse sobre ajustado a sus cadenas de entrenamiento, reproduciéndolas muy fielmente, pero siendo incapaces de reconocer ninguna otra.
- Continuando con el lenguaje de Dyck, este muestra una clara mejora cuanto mayor es el número de épocas, empeorando ligeramente cuando llega a 10, esto puede deberse, al igual que en el apartado anterior a que se empieza a producir un poco de overfitting.
- Acabando con ww^r, para este lenguaje la eficacia crece con el número de épocas sin aparente muestra de sobre ajuste.

De estos resultados podemos observar que en general el número de épocas mejora los resultados hasta la llegada del overfitting, esto ocurre de igual manera con otros modelos de inteligencia artificial más "convencionales" como las redes neuronales. Y se trata de un resultado lógico, pues cuanto más entrenes al sistema, mejores resultados obtendrá.

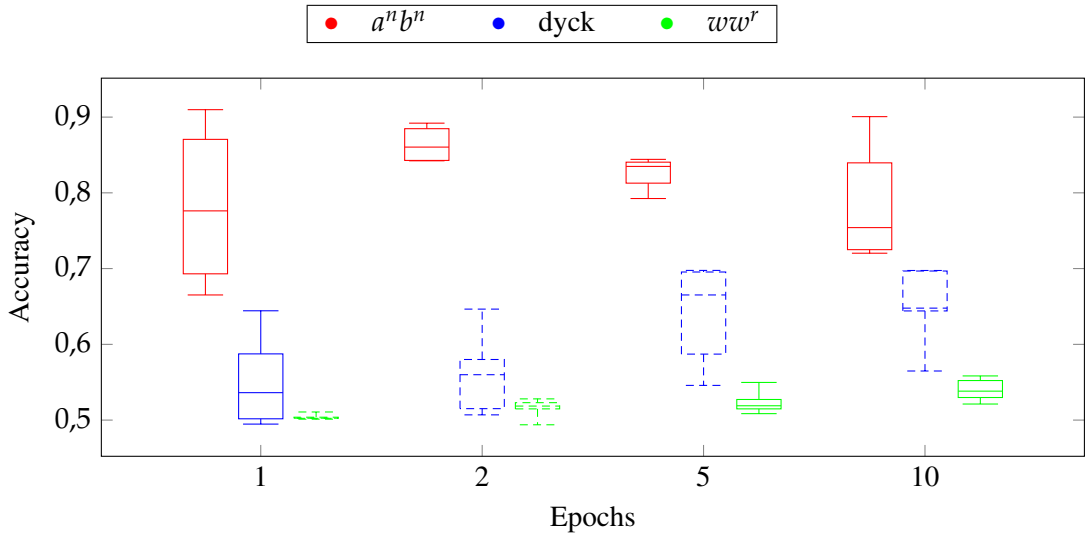

Figura 6.5: Gráfica de la influencia del número de épocas en el accuracy

Lenguaje	Epochs	Mediana	Min.	Max.	Q_1	Q_3
$a^n b^n$	1	0.7762	0.6652	0.9098	0.6931	0.8706
$a^n b^n$	2	0.8604	0.8423	0.892	0.8428	0.8847
$a^n b^n$	5	0.8349	0.7925	0.8442	0.8128	0.8405
$a^n b^n$	10	0.7541	0.7203	0.9006	0.725	0.8396
dyck	1	0.5362	0.4947	0.6443	0.5017	0.5874
dyck	2	0.5599	0.5069	0.6465	0.5151	0.58
dyck	5	0.6652	0.5458	0.6976	0.5871	0.6955
dyck	10	0.6478	0.5648	0.6975	0.6441	0.6968
ww^r	1	0.5025	0.5012	0.5107	0.5023	0.5038
ww^r	2	0.5185	0.4938	0.528	0.5148	0.523
ww^r	5	0.5189	0.5085	0.5497	0.5148	0.5272
ww^r	10	0.5382	0.5212	0.5583	0.5298	0.5522

Tabla 6.3: Tabla de la influencia del número de épocas en el accuracy

6.3.4. Número de producciones terminales y no terminales

Por último pasamos a estudiar la influencia del número de producciones terminales y no terminales de las gramáticas en la precisión, los resultados exactos pueden verse en 6.6 y 6.4.

- Empezando con $a^n b^n$, muestra dos máximos relativos en (2, 5) y (3, 5) producciones no terminales y terminales respectivamente.
- A continuación el lenguaje de Dyck, también muestra dos máximos relativos, pero esta vez (fijándonos en la mediana) en (2, 5) y (5, 5) producciones no terminales y terminales respectivamente.
- En contraposición con ww^r, que muestra un único máximo (fijándonos de nuevo en la mediana), con 5 producciones no terminales y 2 terminales.

Estos resultados tienen sentido considerando el hecho de que, estamos empleando genomas de tamaño fijo, que representan gramáticas incontextuales en forma normal de Chomsky, por lo tanto, dependiendo del lenguaje a codificar, requerirá distinto número de producciones, por lo que al sistema le resulta más sencillo inferir una buena aproximación, si ya se le predispone con un número de producciones adecuado.

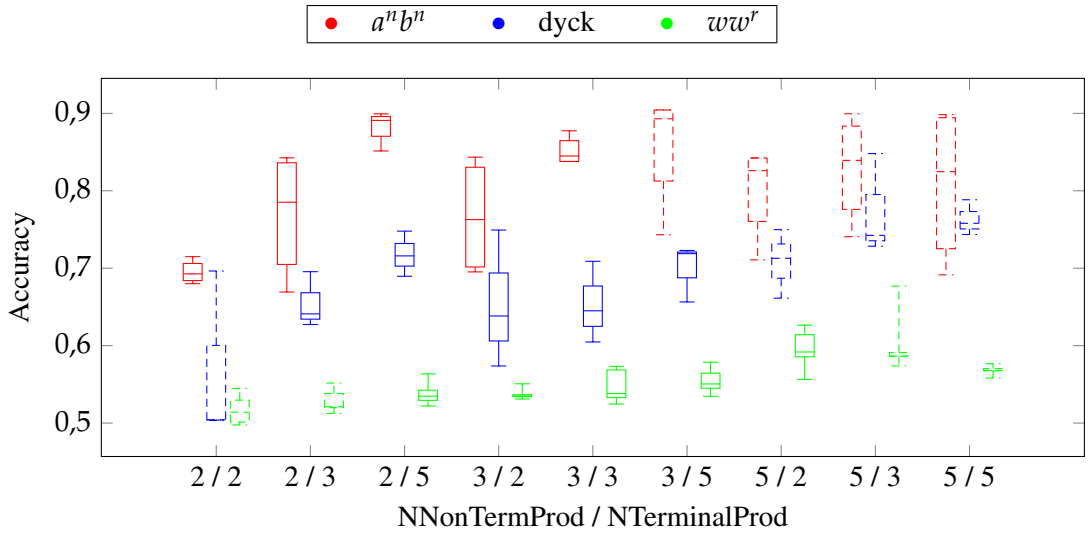

Figura 6.6: Gráfica de la influencia del número e producciones terminales y no terminales en el accuracy

Lenguaje	NNonTermProd	NTerminalProd	Mediana	Min.	Max.	Q_1	Q_3
$a^n b^n$	2	2	0.6927	0.68	0.7149	0.684	0.7061
$a^n b^n$	2	3	0.7852	0.6692	0.8426	0.7049	0.8362
$a^n b^n$	2	5	0.8909	0.8514	0.8994	0.8704	0.8959
$a^n b^n$	3	2	0.7628	0.6952	0.8434	0.7016	0.8304
$a^n b^n$	3	3	0.8449	0.8378	0.8775	0.8379	0.8647
$a^n b^n$	3	5	0.893	0.7432	0.9045	0.8126	0.9043
$a^n b^n$	5	2	0.826	0.7108	0.8425	0.7604	0.8423
$a^n b^n$	5	3	0.8393	0.7407	0.8995	0.776	0.8835
$a^n b^n$	5	5	0.8247	0.6914	0.8983	0.7252	0.8943
dyck	2	2	0.5043	0.5033	0.6964	0.5038	0.6003
dyck	2	3	0.641	0.6273	0.6955	0.6342	0.6683
dyck	2	5	0.7159	0.6896	0.7479	0.7027	0.7319
dyck	3	2	0.6383	0.5737	0.7493	0.606	0.6938
dyck	3	3	0.645	0.6047	0.7089	0.6249	0.677
dyck	3	5	0.7188	0.6564	0.7229	0.6876	0.7209
dyck	5	2	0.7128	0.6613	0.7498	0.687	0.7313
dyck	5	3	0.7424	0.7284	0.8481	0.7354	0.7952
dyck	5	5	0.758	0.7435	0.7884	0.7507	0.7732
ww^r	2	2	0.514	0.4975	0.5447	0.5012	0.5296
ww^r	2	3	0.5211	0.5124	0.5517	0.5199	0.5381
ww^r	2	5	0.5347	0.522	0.5635	0.5293	0.5424
ww^r	3	2	0.5349	0.5311	0.5507	0.5342	0.5366
ww^r	3	3	0.5381	0.5245	0.5733	0.5328	0.5685
ww^r	3	5	0.5505	0.5346	0.5786	0.5449	0.5644
ww^r	5	2	0.5918	0.5564	0.6264	0.5855	0.6141
ww^r	5	3	0.5866	0.5738	0.6768	0.5862	0.591
ww^r	5	5	0.5679	0.5581	0.5767	0.5673	0.5703

Tabla 6.4: Tabla de la influencia del número de producciones terminales y no terminales en el accuracy

CAPÍTULO 7
Conclusiones

Tras los experimentos realizados, podemos confirmar los resultados obtenidos por Taishin Y. Nishida en [Nis20], destacando en concreto la dificultad de inferir determinados lenguajes que, a priori, podrían parecernos más sencillos como ww^r. Esta característica se extiende a la influencia de algunos parámetros (como comentamos en 6.3.1) cuyo valor óptimo varía en función del lenguaje a inferir.

Por otro lado, a partir de los resultados obtenidos, cabe destacar la importancia de una buena configuración del sistema con los parámetros adecuados, obteniendo diferencias de casi un 0.25 en la precisión (como se puede observar en 6.4 con el lenguaje de Dyck) con la variación de un único parámetro.

Por último, y pese a las problemáticas observadas en el modelo, como el coste computacional del método de evaluación o la dependencia de los parámetros del lenguaje objetivo, los resultados obtenidos acaban siendo bastante aceptables, obteniendo por ejemplo, una mediana de 0.89 en el lenguaje $a^n b^n$ con únicamente 25 muestras de entrenamiento y una época (como se puede observar en 6.4). En consecuencia, opino, sería interesante seguir trabajando en esta linea, con el fin de determinar si es posible mejorar los tiempos de ejecución y la generalidad del modelo, para ello, en la siguiente sección (7.1) propongo una serie de posibles trabajos surgidos a partir de este, con el fin de ampliar los conocimientos sobre este modelo.

7.1 Posibles ampliaciones y trabajos futuros

- **Algoritmo de evaluación:** El algoritmo empleado en este trabajo para evaluar el fitness de una gramática, es una variación del algoritmo CYK (como comentamos en 6.1), con el objetivo de evitar los problemas del algoritmo empleado por Taishin Y. Nishida en [Nis20] ($LL(1)$) que es indeterminista y no funciona para cualquier gramática, pero por contra, nuestro algoritmo es mucho más costoso. Aquí se podría ampliar el trabajo, comparando los resultados obtenidos con ambos algoritmos para comprobar si la mejora vale la pena, e incluso ampliar con otros algoritmos o soluciones intermedias.

- **Profundizar en los parámetros:** El simulador aquí presentado es capaz de ajustar y estudiar muchos más parámetros de los que aquí hemos tratado, esta es otra vía evidente de estudio, comprobar la influencia de todos esos parámetros y tratar de encontrar la mejor combinación posible.

- **Probar otros lenguajes:** En este trabajo solo hemos estudiado los lenguajes $a^n b^n$, ww^r y el lenguaje de Dyck, puesto que fueron los empleados por Taishin Y. Nishida en [Nis20], y queríamos comparar los resultados. Pero el simulador está preparado para aceptar cualquier lenguaje en forma normal de Chomsky, por lo que se podría comprobar los resultados ob-

tenidos para otros lenguajes, e incluso buscar patrones que permitan intuir los parámetros a utilizar.

Bibliografía

[Gol67] E. Mark Gold. «Language identification in the limit». En: *Information and Control* 10 (5 1967). ISSN: 00199958. DOI: 10.1016/S0019-9958(67)91165-5.

[Val84] L. G. Valiant. «A theory of the learnable». En: *Communications of the ACM* 27 (11 1984). ISSN: 15577317. DOI: 10.1145/1968.1972.

[Adl98] Leonard M Adleman. «Computing with DNA». En: *Scientific American* 279 (2 1998), págs. 54-61. ISSN: 00368733, 19467087. URL: http://www.jstor.org/stable/26070598.

[Pau00] Gheorghe Paun. «Computing with membranes». En: *Journal of Computer and System Sciences* 61 (1 2000). ISSN: 00220000. DOI: 10.1006/jcss.1999.1693.

[RM00] D. M. Rocke y Z. Michalewicz. «Genetic Algorithms + Data Structures = Evolution Programs». En: *Journal of the American Statistical Association* 95 (449 2000). ISSN: 01621459. DOI: 10.2307/2669583.

[HMU01] John E. Hopcroft, Rajeev Motwani y Jeffrey D. Ullman. «Introduction to automata theory, languages, and computation, 2nd edition». En: *ACM SIGACT News* 32 (1 mar. de 2001), págs. 60-65. ISSN: 0163-5700. DOI: 10.1145/568438.568455.

[Pau02] Gheorghe Paun. «Membrane computing: an introduction». En: (2002). ISSN: 1619-7127.

[Mar+03] Carlos Martín-Vide y col. «Tissue P systems». En: *Theoretical Computer Science* 296 (2 2003), págs. 295-326. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(02)00659-X. URL: https://www.sciencedirect.com/science/article/pii/S030439750200659X.

[AU07] R. Sethi A. Aho M. Lam y J. Ullman. *Compilers: principles, techniques, & tools*. Addison-Wesley, 2007. ISBN: 0321491696.

[Hig13] Colin de la Higuera. *Grammatical inference: Learning automata and grammars*. Vol. 9780521763165. 2013. DOI: 10.1017/CBO9781139194655.

[Nis20] Taishin Y. Nishida. «Evolutionary P systems: The notion and an example». En: vol. 12687. 2020. DOI: 10.1007/978-3-030-77102-7_7.

[Son+21] Bosheng Song y col. «A Survey of Nature-Inspired Computing: Membrane Computing». En: *ACM Computing Surveys* 54 (1 2021). ISSN: 15577341.

APÉNDICE A
Uso del simulador

Como ya hemos comentado anteriormente, para poder realizar la experimentación mostrada en este trabajo, ha sido necesario desarrollar un simulador de Sistemas P en tejido con algoritmos genéticos. Este software ha sido programado en Python3 y es accesible a través de GitHub. En este anexo explicaremos como clonar el repositorio, instalar las dependencias y utilizarlo.

A.1 Instalación

Para realizar la instalación, asumiremos que ya se dispone de una versión de python instalada superior o igual a la 3.8 y la herramienta de entornos virtuales de python venv. Si se ha descargado python de la página oficial debería estar instalada, si se ha instalado por terminal puede que se deba instalar aparte.

También cabe destacar que todos los comandos mostrados a continuación serán para sistemas unix, para windows podrían variar ligeramente, para lo que recomendamos consultar la guía de entornos virtuales de python.

El código fuente del simulador está disponible en GitHub, para utilizarlo, el primera paso será clonar dicho repositorio en local. Si se tiene instalado el cliente de git, es posible hacerlo con el siguiente comando:

```
git clone https://github.com/RodrigoLlanes/Gramatical-inference-through
    -natural-computation.git
```

Una vez clonado el repositorio y situando la terminal en la carpeta raíz del mismo, crearemos un entorno virtual, para no "ensuciar" nuestra instalación de python,

```
python3 -m venv env
```

después lo activamos

```
source env/bin/activate
```

y por último cargamos las librerías del proyecto mediante el archivo "requirements.txt".

```
python3 -m pip install -r requirements.txt
```

Una vez completados todos estos pasos, ya disponemos de un entorno python listo para ejecutar el simulador.

A.2 Uso del simulador

Para hacer uso del simulador, el primer paso es definir la gramática que deseamos inferir, para ello creamos un fichero .json con el siguiente formato:

```
{
    "name": "Dyck",
    "S": "S",
    "Vn" : ["S", "A", "B"],
    "Vt": ["a", "b"],
    "P": {
        "S": [["S", "S"], ["A", "B"]],
        "A": [["a"]],
        "B": [["S", "B"], ["b"]]
    }
}
```

Este fichero está definiendo el lenguaje de Dyck, el parámetro "name" puede emplear código latex y es opcional, solo se usa con fines estéticos. El resto de parámetros son obligatorios y representan el símbolo terminal, los símbolos no terminales, terminales y las producciones respectivamente. Por lo que la gramática tendría la forma:

$$S \to SS$$

$$S \to AB$$

$$A \to a$$

$$B \to SB$$

$$B \to b$$

Esta gramática y las otras dos usadas en este trabajo están accesibles en el repositorio, en la carpeta grammars, como gramáticas de ejemplo.

Una vez definida nuestra gramática formal en forma normal de Chomsky, crearemos el fichero con los casos de entrenamiento y test, para ello haremos uso de la primera utilidad del simulador, cbuilder:

```
python3 main.py cbuilder grammars/wwr.json 50 50 cases.json
```

Donde "grammars/wwr.json" es la ruta al fichero json que define la gramática cuyos casos queremos generar, el primer 50 es el número de casos positivos, el segundo 50 el número de casos negativos y "cases.json" el fichero de salida.

Una vez generados los casos, definiremos los parámetros de nuestro experimento, para ello crearemos un fichero json con el siguiente formato:

```
{
    "n_non_term_sym":          3,
    "n_terminal_sym":          2,
    "n_non_term_prod":         3,
    "n_terminal_prod":         2,

    "n_grammars":             30,
    "n_cells":                 5,

```

```
    "samples_size":          25,
    "epochs":                 1,
    "batch_size":     [1, 2, 5],
    "shuffle_epochs":      true,

    "n_crossovers":           5,
    "n_mutations":            5,
    "mutation_size_min":      1,
    "mutation_size_max":     10,
    "mutate_out":         false
}
```

Este fichero se encuentra a modo de ejemplo en el repositorio, en la carpeta experiments.

Los parámetros de este fichero se dividen en cuatro grupos, el primero hace referencia a las gramáticas que vamos a emplear como población de los sistemas P del tejido, número de símbolos y producciones no terminales y terminales.

El segundo hace referencia a la construcción del tejido, número de sistemas P y número de gramáticas por sistema.

El tercer grupo gestiona el proceso de entrenamiento, tamaño del conjunto de entrenamiento, épocas, tamaño de lote y la opción de "barajar" las muestras en cada época.

El último grupo controla todo lo relacionado con los algoritmos genéticos, el número de recombinaciones, el número de mutaciones, el número mínimo de símbolos cambiados por una mutación, el número máximo y la opción de aplicar las mutaciones y recombinaciones también en el sistema de salida.

Si cualquiera de estos parámetros se define como una lista (como en este ejemplo el tamaño de lote), el simulador ejecutará todas las combinaciones posibles.

Una vez definido nuestro fichero de experimento, llega la hora de ejecutarlo, para ello haremos uso de la principal utilidad del simulador, exp:

```
python3 main.py exp experiments/batch_size.json cases.json result.json
```

Donde "experiments/batch_size.json" es la ruta al fichero del experimento, "cases.json" es la ruta al fichero de casos que queremos emplear y "result.json" la ruta de salida.

Además de esto se le pueden pasar adicionalmente los parámetros -v para aumentar la verbosidad y -r junto con un número entero para repetir cada experimento ese número de veces.

En el fichero "result.json" quedará el resultado de cada ejecución, los parámetros empleados, la gramática resultante y su accuracy.

A.3 Otras utilidades

Para acabar repasaremos las dos utilidades extra del simulador que nos permiten visualizar con mayor facilidad los resultados.

A.3.1. Visualizador

El visualizador nos permite mostrar por pantalla una gráfica con el resultado de la ejecución de un experimento, para ello haremos uso del siguiente comando:

```
python3 main.py plot result.json
```

Donde “result.json” es el fichero de salida de un experimento. Aparte de esto, puede recibir un paramétro opcional -m seguido de “err”, “box” o “dot”, para configurar el modo de dibujo a barras de error, box and whiskers o puntos.

A.3.2. Generador de código latex

Esta utilidad nos permite generar tablas y gráficas latex de un conjunto de experimentos, para poder insertarlos directamente en documentos latex como este.

Para ello, seleccionamos distintos experimentos que o hagan uso del mismo archivo de experimentos o modifiquen los mismos parámetros (no es necesario que usen los mismos casos ni los mismos lenguajes) y ejecutaremos el siguiente comando:

```
python3 main.py latex table.tex plot.tex -f out.json
```

Donde “table.tex” es el fichero de salida para la tabla, “plot.tex” es el fichero de salida para la gráfica y el último parámetro -f puede recibir un número indefinido de ficheros que unirá para generar las tablas y gráficas.

APÉNDICE B
Objetivos de desarrollo sostenible

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenible	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. **Fin de la pobreza.**				x
ODS 2. **Hambre cero.**				x
ODS 3. **Salud y bienestar.**				x
ODS 4. **Educación de calidad.**		x		
ODS 5. **Igualdad de género.**				x
ODS 6. **Agua limpia y saneamiento.**				x
ODS 7. **Energía asequible y no contaminante.**				x
ODS 8. **Trabajo decente y crecimiento económico.**			x	
ODS 9. **Industria, innovación e infraestructuras.**		x		
ODS 10. **Reducción de las desigualdades.**				x
ODS 11. **Ciudades y comunidades sostenibles.**				x
ODS 12. **Producción y consumo responsables.**				x
ODS 13. **Acción por el clima.**			x	
ODS 14. **Vida submarina.**				x
ODS 15. **Vida de ecosistemas terrestres.**				x
ODS 16. **Paz, justicia e instituciones sólidas.**				x
ODS 17. **Alianzas para lograr objetivos.**				x

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Al tratarse este, de un trabajo eminentemente teórico y relacionado con el estudio de los modelos de inferencia gramatical, no parece tener una gran relación con los ODS. Pese a esto, se pueden entrever algunas relaciones con estos objetivos.

Por un lado podemos relacionarlo con el 4º ODS "Educación de calidad", pues este trabajo aporta un simulador de código abierto, que puede ser usado de manera completamente gratuita, esto abre la puerta a la experimentación libre de todo el que lo desee. Además también incluye unos resultados y una información que será accesible para cualquiera, lo que permite aumentar el conocimiento general sobre esta área para cualquier persona independientemente de su posición social, capacidad económica o cualquier otro factor de exclusión social.

Otros dos objetivos de desarrollo sostenibles con los que podría relacionarse este trabajo son el 8 y 9, "Trabajo decente y crecimiento económico" e "Industria, innovación e infraestructuras", pues el objetivo de este trabajo al fin y al cabo es estudiar la eficacia y eficiencia de un modelo de inferencia gramatical, lo que permite a las empresas, grupos de investigación o iniciativas que estén buscando un modelo de este tipo, estar informados de cual es la mejor opción para sus necesidades particulares.

Por último también podríamos encontrar cierta relación con el 13^{er} ODS, "Acción por el clima", pues como ya hemos comentado antes, este trabajo se centra en estudiar la eficacia y eficiencia de un modelo de inferencia gramatical, gracias a este y a otros trabajos similares, podemos encontrar los modelos más eficientes, que nos permitan ahorrar tiempo y, más importante, electricidad, lo que ayudará en mayor o menor medida a frenar el cambio climático.

En conclusión, pese ser este un trabajo con una gran carga teórica y no centrarse en casos prácticos, puede tener ciertas implicaciones con los objetivos de desarrollo sostenibles, que, en este caso, puedan ayudar a la democratización del conocimiento, a la ayuda al desarrollo tecnológico y económico, y a la contribución a la causa del cambio climático.