Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Extensión del algoritmo de Earley a gramáticas MGIG, con aplicación como SAT-solver

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Agustín Santiago Gutiérrez

Director: José Castaño

Buenos Aires, 2016

EXTENSIÓN DEL ALGORITMO DE EARLEY A GRAMÁTICAS MGIG, CON APLICACIÓN COMO SAT-SOLVER

El algoritmo de Earley es un algoritmo de parsing no determinístico general, capaz de reconocer eficientemente cualquier gramática libre de contexto. Es bien sabido que estas gramáticas no pueden expresar muchos lenguajes de interés, como aquellos que impliquen reconocer variables de un lenguaje de programación previamente declaradas, o en general aquellos cuya definición incorpore restricciones de interdependencias cruzadas entre tokens, como es el caso del lenguaje de instancias satisfacibles del problema SAT.

Con el objetivo de modelar algunos lenguajes con ese tipo de restricciones, se propone una extensión de las gramáticas libres de contexto, las MGIG. Las MGIG son un nuevo formalismo de gramática, basado en las GIG, que extiende las gramáticas libres de contexto utilizando una estructura de datos auxiliar llamada multistack, que es manipulada durante una leftmost-derivation y permite modelar ciertas restricciones cruzadas. El algoritmo de Earley se modifica acorde a la nueva gramática, incorporando la estructura de datos a los ítems que genera el algoritmo y modificando sus pasos para reconocer las MGIG.

El objetivo principal de esta tesis radica en la implementación de un algoritmo de reconocimiento del lenguaje generado por una MGIG, basado en el algoritmo de Earley. Una vez implementado, verificamos el desempeño del mismo con diferentes parámetros y variantes posibles sobre el ejemplo motivador de SAT, para compararlo con otros enfoques existentes para resolver el problema SAT.

Palabras claves: Earley, Parsing, GIG, MGIG, Multistack automata, SAT.

AN EXTENSION OF EARLEY'S ALGORITHM TO MGIG GRAMMARS, WITH AN APPLICATION AS SAT-SOLVER

The Earley parser is a general non-deterministic algorithm that efficiently parses any given context free grammar. It is a well known fact that these grammars cannot express many languages of interest, such as those that imply the recognition of previously declared variables in a programming language, or more generally, any language such that its definition involves crossing dependencies among tokens, such as the language of satisfiable instances of SAT.

With the aim of modeling some languages having such crossing dependencies, we propose an extension of context-free grammars, the MGIG. MGIG are a new grammar formalism, based on the previously existing GIG, that extends context free grammars by using an auxiliary data structure called a multistack, which is manipulated during a leftmost-derivation, and enables the modeling of certain crossing dependencies. Earley's algorithm is modified according to the new grammar, by incorporating the data structure to the items generated by the algorithm, and by modifying its operations in order to recognize an MGIG.

The main aim of this thesis is the implementation of a recognition algorithm for MGIG, based on the Earley parser. Once implemented, we verify its performance under different sets of parameters on the motivating example of the SAT language, in order to compare it with other existing SAT-solvers.

Keywords: Earley, Parsing, GIG, MGIG, Multistack automata, SAT.

AGRADECIMIENTOS

A Mamá, por absolutamente todo en esta vida.
A Papá, por sembrar en mí el amor por la ciencia.
A Facu, por ser la mejor persona del mundo.
A Mel, por ser la mejor amiga que se puede tener.
A mis abuelos, por el apoyo incondicional.
A la OMA y a la OIA, porque solamente gracias a ellas soy la persona que soy.

A mi director José Castaño, por aguantarme.
A los jurados, por tomarse el tiempo de leer esta tesis.

Índice general

1. INTRODUCCIÓN

1.1. Contexto

Existe consenso general sobre la idea de que modelar el lenguaje natural y los fenómenos biológicos [42] requiere un poder expresivo que excede la maquinaria de las gramáticas libres de contexto. En particular, se pueden mencionar los siguientes lenguajes relevantes:

(a) *Lenguaje de copia*, $\{ww \,|w \in \{0,1\}^*\}$

(b) *Dependencias múltiples*, e.g.: $\{a^n b^n c^n d^n \mid n \geq 1\}$

(c) *Dependencias cruzadas*, como por ejemplo $\{a^n b^m c^n d^m \mid n,\ m \geq 1\}$.

En [28] se proponen lenguajes con poder expresivo sensible al contexto restringido (*Mildly context-sensitive languages*), capaces de describir los ejemplos anteriores. Estos modelos han sido generalizados mediante niveles de control, niveles de inmersión o mediante stacks. Se establecen jerarquías de niveles de lenguajes, de manera tal que un lenguaje de nivel k incluye estrictamente a un lenguaje de nivel k-1 [34, 53, 43, 15].

La complejidad del problema de reconocimiento depende del nivel del lenguaje dentro de la jerarquía, de manera que para un lenguaje de nivel k la complejidad es $\mathcal{O}(n^{3 \cdot 2^{k-1}})$ [53]. Adoptando una visión basada en autómatas, los niveles de lenguaje se corresponden con la cantidad de pilas utilizadas. Existen varios modelos de *2-stack pushdown automata* (e.g. [21], [49], [7]), y también modelos generalizados de multi-stacks (e.g. [48, 15, 52]).

Es bien sabido que un pushdown-automata con dos pilas equivale a una máquina de Turing (e.g., [26]). Los modelos de multistacks restringen la operación sobre los stacks adicionales (inmersos u ordenados) permitiendo operaciones de *read* o *pop* sobre el tope de un stack adicional únicamente si los stacks precedentes están vacíos, como por ejemplo en *multiple pushdown automata* (MPDA) [15] (También denominados OMPDAs, Ordered Multiple pushdown automata). Estos modelos se dicen *restricted to reading* [52]. El uso de múltiples stacks con restricciones de lectura y escritura se ha analizado en [50, 51, 52, 54].

Se supone que los lenguajes naturales con un ordenamiento flexible de sus constituyentes involucran dependencias cruzadas más complejas [5]. Stabler [45] describe varias dependencias cruzadas. Desde otra perspectiva, las "dependency grammars" y el "dependency parsing" han recibido mucha atención, en particular "discontinuous constituents" y "non-projective dependencies" [37, 35, 23, 36]. La planaridad y la *well-nestedness* son conceptos que se han utilizado para caracterizar "non-projective dependencies" [36]. Satta [40] caracterizó las dependencias cruzadas ilimitadas como un problema fundamental para el reconocimiento de *Linear Context-Free Rewriting Systems (LCFRS)*: pueden codificarse en el "fan-out" o en el "rank" del sistema, y el problema de reconocimiento universal es NP-completo. Es un patrón recurrente que los formalismos propuestos para modelar ciertos fenómenos resultan en problemas de reconocimiento NP-hard o NP-completos [40, 38, 4, 41, 39, 47]. En *LCFRS parsing*, varios trabajos se han concentrado en las estructuras denominadas *well-nested* [22, 29, 6, 32, 31, 30].

El formalismo de gramática que propondremos (MGIG) es capaz de modelar dependencias cruzadas ilimitadas, como las que presenta el lenguaje de las fórmulas proposicionales

satisfacibles (SAT) [40]. , ilustrado en Fig. 1.1 donde cada bloque representa una cláusula y las flechas indican dependencias entre variables:

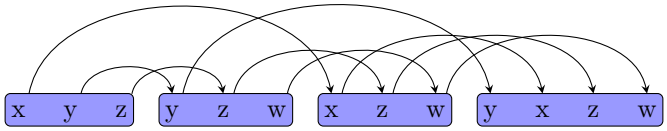


Fig. 1.1: Unrestricted crossing dependencies in a propositional formula

El problema SAT también se abordó desde perspectivas basadas en lenguajes, autómatas finitos y parsing en [12, 14, 8]. Particularmente en [8], se presenta por primera vez la noción de MGIG, y se plantea el enfoque de encarar el problema SAT como un problema de parsing de MGIG, mediante técnicas como las que desarrollamos en esta tesis.

El poder expresivo de las MGIGs conlleva una contribución adicional desde un punto de vista teórico: permite separar la maquinaria requerida por el lenguaje de múltiples copias, múltiples dependencias y dependencias cruzadas ilimitadas: los primeros dos pueden resolverse mediante GILs (i.e. [9]). En otras palabras, la única ganancia de poder expresivo al agregar "pushdown storages" en el sentido de Wartena [52]) es aumentar la capacidad de multi-planaridad.

1.2. Trabajos previos y temática de estudio

En este trabajo se estudia principalmente la noción de MGIG, un nuevo tipo de gramáticas formales que generaliza a las gramáticas libres de contexto, y que están basadas en las GIG[10]. La motivación original para definir este nuevo tipo de gramáticas es obtener suficiente poder expresivo como para poder expresar el lenguaje de instancias satisfacibles del problema SAT. De esta forma, con los métodos desarrollados en este trabajo podemos encarar el problema SAT desde una perspectiva novedosa, planteándolo **directamente** como un problema de parsing de una gramática formal, ya que bastará para determinar si una cierta instancia es satisfacible con verificar si esa instancia de entrada pertenece al lenguaje generado por una gramática particular.

Lo que plantearemos será entonces un tipo de gramática más difícil de reconocer que las libres de contexto, capaz de expresar SAT, pero que a su vez, no genere un problema de reconocimiento más difícil que el propio SAT: veremos en 2.3.8 y 3.3.1 que el problema de reconocimiento del lenguaje generado por una MGIG es NP-completo, y por lo tanto es equivalente en complejidad a SAT, salvo por transformaciones de tiempo polinomial. Otros enfoques anteriores de problemas de parsing NP-completos se han centrado fundamentalmente en la demostración de que el problema es NP-completo, y no en la capacidad de expresar de forma natural y directa el lenguaje de las instancias satisfacibles de SAT con una gramática concreta.

Otro enfoque diferente, pero relacionado con el área de teoría de lenguajes, se estudia en [14]. Allí se caracteriza a las **valuaciones** que satisfacen una instancia particular de SAT, como la intersección de los lenguajes generados por una colección muy sencilla de expresiones regulares (Finite State Intersection Grammar, *FSIG*), una por cada cláusula de la fórmula, con lo cual resolver SAT queda reducido a determinar si el correspondiente lenguaje intersección es vacío. Sin embargo, en dicho enfoque no hay una gramática formal

directamente asociada que se esté intentado parsear, ni se puede pensar en producir un árbol de derivación asociado, con lo cual el problema no queda caracterizado directamente como un problema de parsing. Además, la maquinaria de teoría de lenguajes asociada (los autómatas finitos) opera sobre un lenguaje de valuaciones, y no directamente sobre la fórmula de entrada original como entrada para el autómata.

Además de definir y estudiar las propiedades de las MGIG, nuestro principal aporte en este trabajo será el desarrollo e implementación de algoritmos para el reconocimiento de MGIG, basados en el algoritmo de parsing no determinístico de Earley[18].

1.3. Breve reseña de MPDA, GIG y MGIG, desde la teoría de autómatas

1.3.1. Multi-pushdown automata

Un autómata de múltiples pilas (*Multi-pushdown automata*) es una generalización natural del autómata de pila (*Pushdown automata*).

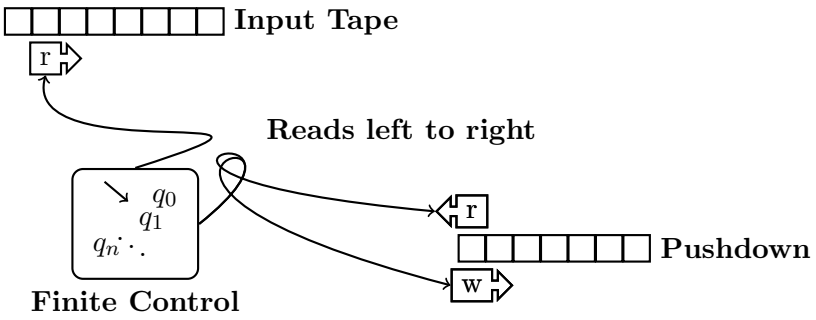


Fig. 1.2: Push Down Automata

En el autómata de pila convencional, la entrada se lee en una sola pasada de izquierda a derecha, y según una lógica de "control finito" (es decir, una relación finita que define al autómata, especificando las transiciones válidas), se opera mientras se procesa la entrada con una única stack auxiliar LIFO, sobre la cual se puede tanto leer como escribir.

De manera similar, un autómata de múltiples pilas tiene $n \geq 1$ pilas de lectoescritura, y las correspondientes transiciones de la relación finita de control especifican operaciones de lectoescritura combinada en todas las pilas a la vez. Como esta versión general resulta demasiado poderosa (ya hemos mencionado que con $n \geq 2$, el modelo tiene un poder expresivo equivalente a una máquina de Turing no determinística), se utiliza generalmente una restricción que llamamos autómata de n-pilas (*n-Multi-pushdown automata*), en la cuál se puede escribir libremente a todas las n pilas en cada operación, pero únicamente es posible leer **de la primera pila no vacía**.

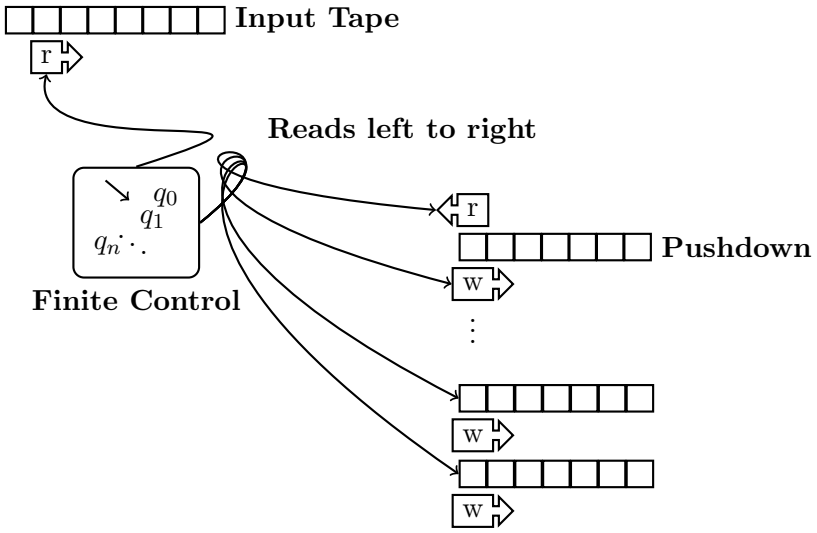


Fig. 1.3: n-Push Down Automata

1.3.2. Global Index Grammars

En [10, 11, 9] se estudian las Global Index Grammars (GIG), un tipo de gramática más general y expresiva que las gramáticas libres de contexto asociadas al autómata de pila, que puede ser reconocida en tiempo polinomial (particularmente, el problema de reconocimiento de una GIG prefijada es $O(n^6)$, siendo n la longitud de la cadena de entrada).

El autómata asociado a las GIG es un autómata con **dos pilas de lectoescritura**: Podemos pensar que la primera pila, sin restricciones de lecto-escritura, está asociada al comportamiento libre de contexto dado por la gramática usual. La segunda pila en cambio, permite modelar restricciones adicionales sobre las producciones que pueden aplicarse en la derivación. La restricción esencial que se impone al autómata para el caso de GIG, y que evita obtener el poder expresivo de una máquina de Turing, es que la escrituras (es decir, las operaciones de *push*) deben estar necesariamente ligadas a transiciones que **lean de la cadena de entrada**. En otras palabras, no está permitido escribir en la segunda pila durante transiciones ϵ / λ. Esto acota fuertemente la cantidad de escrituras que se pueden realizar sobre esta segunda pila, en términos del tamaño de la entrada.

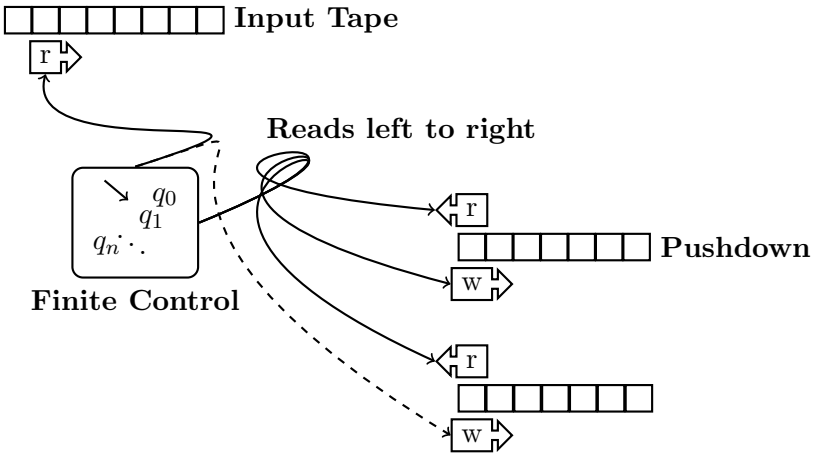


Fig. 1.4: GIG -Push-Down Automata

1.3.3. Multiple Global Index Grammar

En este trabajo desarrollamos las Multiple Global Index Grammar (MGIG), una variante de GIG donde se opera con múltiples pilas adicionales, en lugar de una sola.

En este modelo, al igual que en GIG es siempre necesario leer de la entrada para poder escribir a las stacks adicionales. La novedad con respecto a anteriores modelos de múltiples pilas, es que las pilas se mantienen conceptualmente **ordenadas** de acuerdo a un **criterio de ordenación**, que es parte de la gramática. De esta forma, únicamente es posible leer de la primera pila en este ordenamiento, mientras que las escrituras se realizan siempre en la menor pila posible (ya que a su vez, cada pila se encuentra internamente ordenada con el mismo criterio).

De esta forma, las diferentes pilas adicionales no son independientes entre sí, sino que es más conveniente pensarlas como una única estructura "compuesta", que en este trabajo denominamos *multistack*.

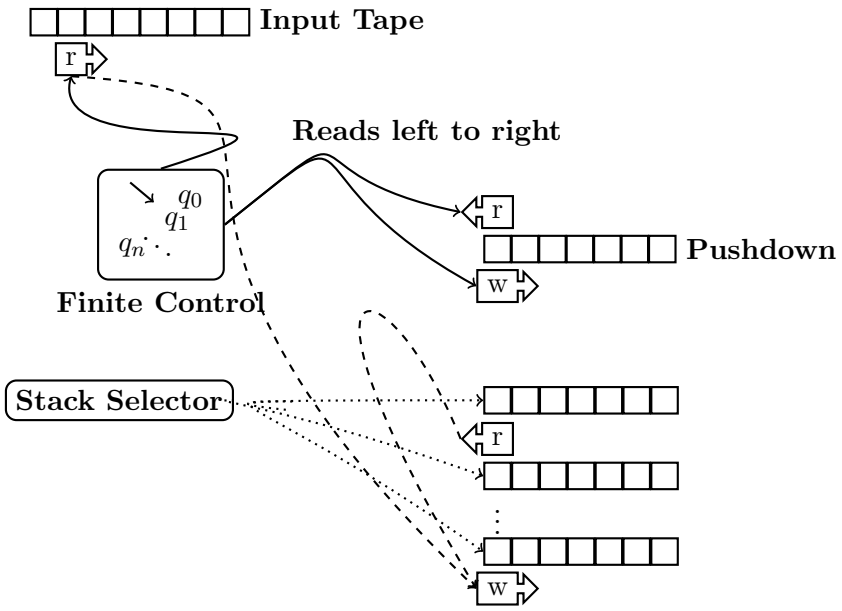


Fig. 1.5: MGIG Multi-Push Down Automata

1.3.4. Sobre el uso de autómatas en este trabajo

Si bien hemos presentado en esta sección, a modo de introducción, una comparación entre los modelos de autómatas asociados a los diversos formalismos, durante todo el desarrollo de nuestro trabajo nos hemos centrado en un enfoque directamente basado en gramáticas y su reconocimiento mediante variantes del Earley parser, y no en autómatas.

1.4. Estructura del presente trabajo

En 1, presentamos la temática general de estudio de la tesis, enmarcándola en el contexto de trabajos previos, y definimos la organización general del trabajo.

En 2, definimos formalmente la noción de MGIG, repasando para ello las definiciones formales de GLC y GIG, en las cuales las MGIG están basadas, y estudiamos posibles variantes, consecuencias y limitaciones de las definiciones utilizadas. Por ejemplo, en 2.3.6 probamos que las MGIG tienen al menos tanto poder expresivo como las GIG, y en 2.3.8 demostramos que el problema de reconocimiento general para MGIG pertenece a NP.

En 3, mostramos la conexión entre el problema SAT y el reconocimiento de MGIG. Para esto definimos en 3.2 un lenguaje formal $\mathcal{SAT}$, cuyo reconocimiento es esencialmente equivalente a resolver el problema SAT. En 3.3, mostramos cómo mediante el reconocimiento de una MGIG concreta es posible determinar si una cadena pertenece al lenguaje $\mathcal{SAT}$, completando así la prueba de equivalencia entre reconocimiento de MGIG y SAT, resultando ambos problemas NP-completos.

En 4, extendemos el algoritmo de Earley para obtener un algoritmo general de reconocimiento de MGIG. Mencionamos varias alternativas de implementación del algoritmo, y estudiamos su complejidad.

En 5, estudiamos la utilización del algoritmo obtenido en 4 como mecanismo para resolver SAT. Identificamos en 5.1.1 una familia particular de gramáticas MGIG que pueden ser reconocidas con una versión más sencilla y eficiente del algoritmo, entre las cuales se

encuentran las gramáticas dadas para SAT en 3, y demostramos la correctitud de esta versión. Se mencionan en A.8 otras versiones del algoritmo que consideramos durante el desarrollo de la tesis.

En 6 mostramos y comentamos los resultados obtenidos al probar el algoritmo.

En 7 resumimos las principales conclusiones del trabajo, y posibles direcciones de trabajo futuro.

En el apéndice A, describimos todo lo directamente relacionado con nuestra implementación particular del algoritmo Earley-MGIG. Además de las limitaciones y decisiones particulares en la implementación del algoritmo en sí, se mencionan algunos pequeños programas utilitarios auxiliares que se implementaron y la estructura del proyecto Python en general. Por ejemplo en A.5 incluimos una descripción de un formato de archivo que hemos definido para la descripción práctica de una MGIG.

En el apéndice B, transcribimos una copia textual de la totalidad de las mediciones de tiempos obtenidas al ejecutar el programa sobre las distintas instancias. Estas mismas mediciones se encuentran en la versión digital del trabajo.

2. DEFINICIONES E INTRODUCCIÓN A MGIG

2.1. Gramática libre de contexto (CFG)

Una *Gramática Libre de Contexto (CFG, Context Free Grammar)* es una tupla que consta de un conjunto finito N de símbolos llamados *no terminales*, un conjunto finito T de símbolos llamados *terminales* (con $N \cap T = \emptyset$), un símbolo destacado $S \in N$ el *símbolo inicial*, y un conjunto finito de *reglas de producción* de la forma $A \to \alpha$, donde $A \in N$, y $\alpha \in (N \cup T)^*$. Cada regla de producción se interpreta como una regla de reescritura posible, indicando que es posible reemplazar el no terminal A por la cadena α.

Es de esta forma que en el contexto de una gramática G, dadas dos cadenas s_1 y s_2 sobre el alfabeto $\Sigma = N \cup T$, decimos que s_2 *se deriva en un paso de* s_1, y notamos $s_1 \Rightarrow s_2$, si existen cadenas $\alpha, \beta \in \Sigma^*$, un no terminal $X \in N$ y una regla de producción $X \to \gamma$ en G tales que:

$$s_1 = \alpha X \beta \text{ y } s_2 = \alpha \gamma \beta \tag{2.1}$$

Es decir, cuando s_2 se obtuvo a partir de s_1 efectuando exactamente un reemplazo según lo indica alguna de las reglas de producción de la gramática.

A partir de la derivación en un paso, se define la *relación de derivación* como su clausura reflexiva y transitiva: Es decir, s_2 se deriva de s_1, y lo notamos $s_1 \overset{*}{\Rightarrow} s_2$, cuando $s_1 = s_2$ o bien $s_1 \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_n \Rightarrow s_2$, para $n \geq 0$. A la secuencia ordenada de $n+1$ reglas de producción aplicadas en esta cadena junto con las posiciones de los no terminales que se reemplazan en cada paso se la denomina una *derivación* de s_2 a partir de s_1.

A las cadenas $s \in \Sigma^*$ tales que $S \overset{*}{\Rightarrow} s$ se las denomina *formas sentenciales*.

El *lenguaje generado* por una gramática es el conjunto $\mathcal{L} = \{s \in T^* | S \overset{*}{\Rightarrow} s\}$, es decir todas las cadenas de terminales que se derivan del símbolo inicial, o equivalentemente todas las formas sentenciales en T^*. Decimos que un lenguaje es un CFL (*Context Free Language*) si es generado por una CFG. Denotamos también CFL al conjunto de todos estos lenguajes.

Una *derivación por izquierda (leftmost derivation)* de s_2 a partir de s_1, es una derivación de s_2 a partir de s_1 en la cual los reemplazos que se realizan en cada paso tienen $\alpha \in T^*$ con la notación utilizada en 2.1; es decir, si en cada paso se reescribe siempre el no terminal de más a la izquierda en la cadena. Análogamente puede definirse una derivación por derecha.

A partir de una derivación, puede definirse su *árbol de derivación*, que será un árbol con raíz y ordenado (es decir, la lista de hijos de cada nodo tiene un orden bien definido), de manera constructiva: se comienza con nodo raíz que contiene al símbolo inicial S. Luego por cada paso de la derivación, en orden, si se reemplaza un cierto no terminal X según una regla $X \to t_1 t_2 \cdots t_k$ con $t_i \in \Sigma$, se agregan k nuevos nodos con símbolos $t_1, t_2, \cdots, t_k$ como hijos del nodo n_X correspondiente al no terminal X que se está reemplazando, y se etiqueta a n_X con la regla de producción utilizada. Notar que en cada paso, por cada símbolo en la forma sentencial correspondiente a dicho paso existe exactamente una hoja distinta correspondiente a dicho símbolo en el árbol, y en particular la forma sentencial se obtiene tomando en orden todas las hojas del árbol. Más aún, en un árbol de derivación

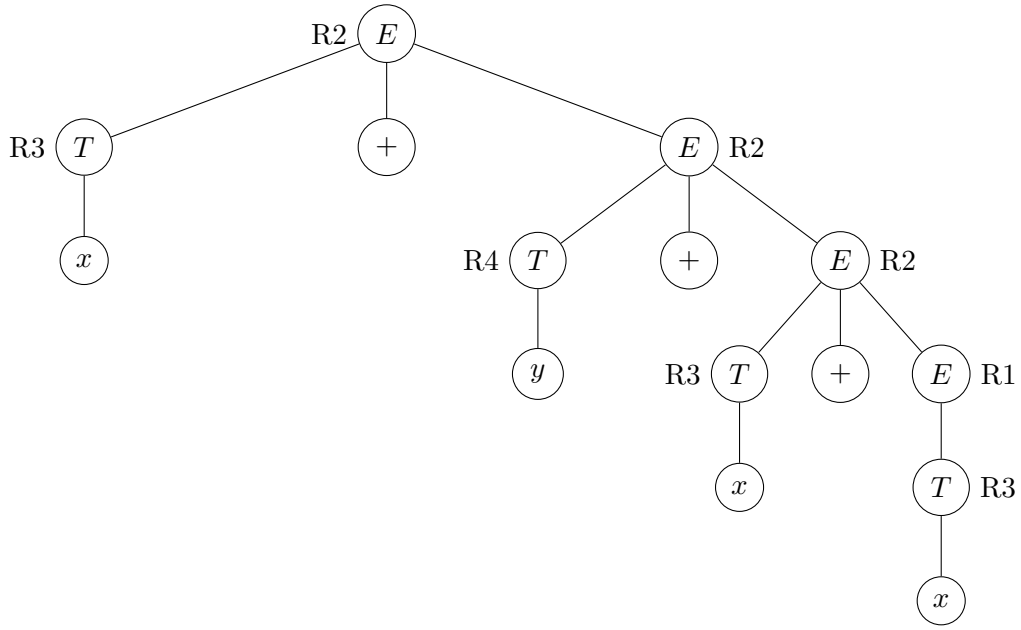
para una palabra $s \in \mathcal{L}(G)$, las hojas en orden corresponden a los símbolos de la palabra s.

Por claridad, cuando se aplica una regla de la forma $A \to \lambda$, en lugar de dejar al no terminal A como hoja en el árbol de derivación, al aplicar esta regla creamos un único hijo del nodo correspondiente, etiquetado con el símbolo especial λ.

Ejemplo 1 : Con la siguiente gramática para $T = \{+, x, y\}$ con símbolo inicial E:

$$\begin{array}{lrcl} \text{(R1)} & E & \to & T \\ \text{(R2)} & E & \to & T + E \\ \text{(R3)} & T & \to & x \\ \text{(R4)} & T & \to & y \end{array}$$

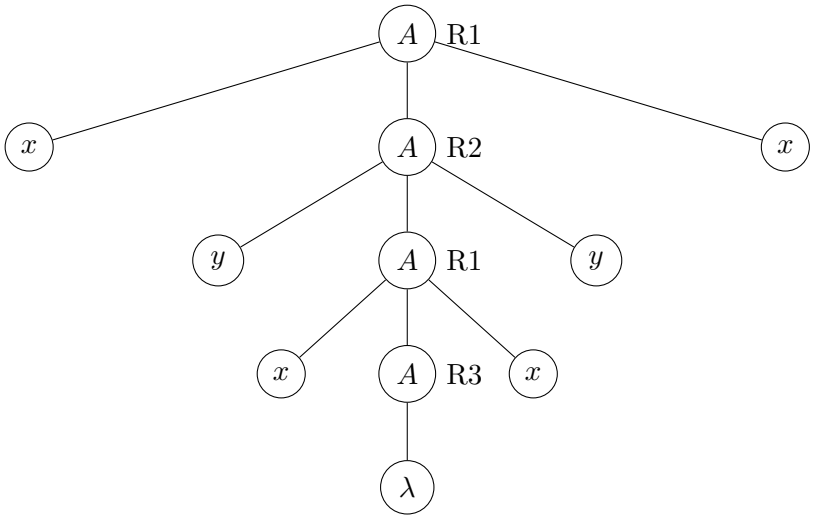
Un árbol de derivación para la cadena $x + y + x + x$ es:



Ejemplo 2 Con la siguiente gramática para $T = \{x, y\}$ con símbolo inicial A:

$$\begin{array}{lrcl} \text{(R1)} & A & \to & xAx \\ \text{(R2)} & A & \to & yAy \\ \text{(R3)} & A & \to & \lambda \end{array}$$

Un árbol de derivación para la cadena $xyxxyx$ es:



Un árbol de derivación captura el *conjunto* de reemplazos realizados durante una derivación, pero abstrayéndose de su orden. Existe una biyección entre las derivaciones por izquierda (o análogamente, por derecha) y los árboles de derivación, dado que una derivación por izquierda se obtiene realizando un recorrido de DFS (de izquierda a derecha) sobre el árbol de derivación.

En el caso de las CFG, no se pierde generalidad en la definición de lenguajes, al limitarse a trabajar con derivaciones por izquierda (o por derecha), ya que en cualquier derivación pueden reordenarse libremente los reemplazos realizados de manera de reemplazar siempre el no terminal más a la izquierda (o derecha) sin alterar la cadena final de terminales producida.

2.2. Global Index Grammar (GIG)

Las *Global Index Grammar (GIG)* son una modificación a las CFG que aumentan su poder expresivo, es decir, permiten generar lenguajes que no pueden ser generados mediante una CFG. Descritas en [10] [11], una GIG es una CFG a la cual se le agrega además un conjunto finito I de índices (De forma que N, T, I sean disjuntos dos a dos), un símbolo especial $\#$ utilizado como base de pila (que no pertenece a N, T, I) y se extiende la forma posible de las producciones, modificándose de manera acorde la noción de derivación.

Concretamente, las producciones de una GIG son de una de las siguientes formas, donde $\alpha, \beta \in (N \cup T)^*$, $x \in I$, $y \in I \cup \{\#\}$ y $a \in T$:

1. $A \to \alpha$ (producciones *epsilon*)
2. $A \underset{[y]}{\to} \alpha$ (producciones *epsilon* con restricciones)
3. $A \underset{x}{\to} a\beta$ (producciones *push*)
4. $A \underset{\overline{x}}{\to} \alpha$ (producciones *pop*)

La relación de derivación se modifica al considerar que además de la cadena que se reescribe, se mantiene en todo momento una *pila* de índices. Los distintos tipos de producciones llevan asociadas operaciones en la pila, que deben realizarse necesariamente al

mismo tiempo que el reemplazo en la cadena descrito por la producción. Además, las derivaciones están restringidas (por definición de las GIG) a *derivaciones por izquierda* únicamente.

Formalmente podemos pensar la relación de derivación en base a cadenas de la forma $\eta\#\gamma$ con $\eta \in I^*$ y $\gamma \in (N \cup T)^*$, de manera que son válidas las siguientes derivaciones en un paso correspondientes a cada una de las producciones anteriores respectivamente:

1. $\eta\#\gamma_1 A\gamma_2 \Rightarrow \eta\#\gamma_1\alpha\gamma_2$, con $\gamma_1 \in T^*$ y $\gamma_2 \in (N \cup T)^*$ (epsilon)
2. - Si $y \in I : y\eta\#\gamma_1 A\gamma_2 \Rightarrow y\eta\#\gamma_1\alpha\gamma_2$, con $\gamma_1 \in T^*$ y $\gamma_2 \in (N \cup T)^*$
   - Si $y = \# : \#\gamma_1 A\gamma_2 \Rightarrow \#\gamma_1\alpha\gamma_2$, con $\gamma_1 \in T^*$ y $\gamma_2 \in (N \cup T)^*$

   (epsilon con restricción)
3. $\eta\#\gamma_1 A\gamma_2 \Rightarrow x\eta\#\gamma_1 a\beta\gamma_2$, con $\gamma_1 \in T^*$ y $\gamma_2 \in (N \cup T)^*$ (push)
4. $x\eta\#\gamma_1 A\gamma_2 \Rightarrow \eta\#\gamma_1\alpha\gamma_2$, con $\gamma_1 \in T^*$ y $\gamma_2 \in (N \cup T)^*$ (pop)

La relación de derivación $\overset{*}{\Rightarrow}$ es al igual que en el caso de las CFG, la clausura reflexivo transitiva de $\Rightarrow$

Notar que una característica importante de las GIG es la restricción de las producciones *push*, permitiéndose únicamente reglas de este tipo cuando la parte derecha comienza con un símbolo terminal. Tal restricción permite dar garantías de complejidad: Concretamente el tamaño de la pila en cualquier paso de una derivación exitosa de una cadena de terminales s no podrá nunca superar $|s|$, ya que por la forma de las reglas *push*, cada vez que se aplica una regla de ese tipo se agrega un símbolo terminal a la forma sentencial.

Notar además que la condición $\gamma_1 \in T^*$ significa que solo consideramos derivaciones por izquierda. Esta restricción a derivaciones por izquierda es importante porque al estar acopladas las reescrituras realizadas a la cadena de terminales y no terminales con las operaciones realizadas en la pila, el orden en que se hacen los reemplazos es importante (ya que dos ordenamientos que lleven a una misma cadena de terminales y no terminales, podrían resultar en pilas distintas, o uno de ellos podría realizar operaciones de pila inválidas), y por lo tanto no es cierto como lo era en las CFG, que toda derivación puede ser reordenada equivalentemente a una derivación por izquierda. Las GIG funcionan, por definición, únicamente en base a derivaciones por izquierda.

El *lenguaje generado* por una GIG es el conjunto $\mathcal{L} = \{s \in T^* | \#S \overset{*}{\Rightarrow} \#s\}$, es decir todas las cadenas de terminales que se derivan del símbolo inicial, empezando y terminando con la pila vacía. Decimos que un lenguaje es un GIL (*Global Index Language*) si es generado por una GIG. Denotamos también GIL al conjunto de todos estos lenguajes.

Los árboles de derivación para GIG se definen de exactamente la misma manera que para CFG, con la observación de que un nodo ahora puede ser etiquetado con una regla de producción de GIG, con operaciones de pila. Por lo tanto, si no se lee el árbol de derivación en el orden correspondiente a una derivación por izquierda, no hay garantías de obtener una derivación válida en la GIG.

Podemos mencionar que la forma de las reglas correspondientes a operaciones de pila es más general de lo necesario (en términos de poder expresivo). Con la misma construcción que se menciona en 2.3.6, se puede ver que solamente son necesarias reglas de la forma $A \underset{t}{\rightarrow} a$, $A \underset{\bar{t}}{\rightarrow} \lambda$ y $A \underset{[t]}{\rightarrow} \lambda$ (además de producciones usuales libres de contexto), para obtener el mismo poder expresivo de las GIG.

2.2.1. CFL $\subsetneq$ GIL

Es inmediato que todo lenguaje libre de contexto es generado por una GIG, ya que las CFG son por definición (tomando un conjunto de índices vacío) también GIG, con el mismo lenguaje generado. Como ejemplo de un lenguaje que no es libre de contexto pero puede ser generado por una GIG, consideremos:

$$\{a^n b^n c^n | n \geq 0\}$$

Es bien conocido que este lenguaje no es libre de contexto [27], pero sin embargo la siguiente GIG sencilla lo genera (con símbolo inicial S):

$$S \to AC$$

$$A \underset{t}{\to} aAb$$

$$A \to \lambda$$

$$C \underset{\bar{t}}{\to} cC$$

$$C \to \lambda$$

Esto muestra que las GIG son una extensión propia de las CFG en términos de poder expresivo.

Para destacar que es relevante a la definición de GIG el hecho de que las derivaciones deben ser por izquierda, consideremos la siguiente modificación del ejemplo anterior:

$$S \to AC$$

$$A \underset{\bar{t}}{\to} aAb$$

$$A \to \lambda$$

$$C \underset{t}{\to} cC$$

$$C \to \lambda$$

Si se permitiera cualquier orden para la derivación, el lenguaje generado por esta GIG sería el mismo que en el caso anterior, pues la única restricción que fuerza el manejo de pila en ambos casos es que se utilicen la misma cantidad de veces las reglas número 2 y 4. Sin embargo, por definición esta GIG genera únicamente la cadena vacía, ya que es imposible usar la segunda regla, pues en una derivación por izquierda, siempre se encuentra la pila vacía al momento de expandir un no terminal A, y por lo tanto no se puede hacer pop ya que no se encuentra el símbolo requerido en el tope de la pila.

2.3. Multiple Global Index Grammar (MGIG)

2.3.1. Definición

Una *Multiple Global Index Grammar (MGIG)* es un nuevo tipo de gramática, que puede verse como una modificación de las GIG ya existentes. Esto resulta similar al caso de las GIG si se consideran como una modificación de las conocidas CFG. Sin embargo, una diferencia para tener en cuenta para la definición particular de MGIG que daremos es que, debido a las restricciones que se impondrán sobre la forma de las producciones que modifican la pila en el caso de las MGIG, las MGIG no pueden considerarse como una **extensión** de las GIG, ya que algunas producciones que son válidas en una GIG resultan por definición inválidas para utilizar en una MGIG. Esto no era así para las GIG respecto de las CFG, ya que toda producción que tenga la forma utilizada en las CFG puede utilizarse como parte de una GIG, siendo así las GIG una extensión propia de las CFG. No obstante, las MGIG constituyen al igual que las GIG una extensión propia de las CFG.

Las MGIG son definidas por primera vez en [8], donde el autor las presenta como una forma de encarar el problema SAT como un problema de parsing. La definición formal que daremos a continuación está enfocada en nuestra implementación y aplicación particular al caso de SAT, con lo cual no es idéntica a la del paper mencionado, pero como veremos luego ambas versiones resultan esencialmente equivalentes.

Al igual que en el caso de las GIG, una MGIG es una CFG a la cual se le agrega además un conjunto finito I de índices (De forma que N, T, I sean disjuntos dos a dos), un símbolo especial # utilizado como base de pila (que no pertenece a N, T, I) y se extiende la forma posible de las producciones, modificándose de manera acorde la noción de derivación. En el caso de una MGIG, se agrega a la tupla adicionalmente una relación $O \subseteq I \times I$ de preorden total[1].

Concretamente, las producciones de una MGIG son de una de las siguientes formas, donde $\alpha \in (N \cup T)^*$, $a \in T$, $A \in N$, $y \in I \cup \{\#\}$ y $\mu, \mu_s \in I$ con μ_s *sufijo*[2] de μ:

1. $A \to \alpha$ (producciones *epsilon*)
2. $A \underset{[y]}{\to} \alpha$ (producciones *epsilon* con restricciones)
3. $A \underset{\mu}{\to} a|aA$ (producciones *push*)
4. $A \underset{\overline{\mu}}{\to} aA|A|\lambda$ (producciones *pop*)
5. $A \underset{\overline{\mu}|\mu_s}{\to} aA|A|\lambda$ (producciones *pop-push*)

La relación de derivación se modifica en las MGIG de manera similar al caso de las GIG, en las que se agregaba una pila de índices que se debe manipular adecuadamente junto a la forma sentencial de acuerdo a la regla de producción utilizada. En el caso de las MGIG, no tendremos una pila de índices sino una *lista ordenada de pilas* de índices. Las derivaciones siguen restringidas a *derivaciones por izquierda* únicamente.

[1] Es decir, O es reflexiva, transitiva, y tal que $\forall x, y \in I$ se tiene $(x, y) \in O \vee (y, x) \in O$

[2] La noción de sufijo es explicada más detalladamente en 2.3.2

Formalmente podemos pensar la relación de derivación en base a cadenas de la forma $\xi\gamma$ con $\xi = \eta_1\#\eta_2\#\cdots\#\eta_n\#$, $\eta_i \in I^+$, $n \geq 0$ y $\gamma \in (N \cup T)^*$. A la primera parte ξ la llamaremos la (o el) *multistack*, y es una extensión natural de la pila global de índices que tenían las GIG, y que ahora está formada por la concatenación de múltiples pilas de índices. γ por otro lado corresponde a la forma sentencial en el caso de una gramática libre de contexto. Notar que dada una forma sentencial cualquiera así definida, se pueden identificar unívocamente ambas partes ξ y γ, de manera que al definir la relación de derivación se operará con γ, mediante reescrituras usuales, y con ξ mediante *operaciones de multistack* (push, pop y pop-push), que definimos formalmente a continuación.

De esta manera, son válidas las siguientes derivaciones en un paso correspondientes a cada uno de los tipos de producciones mencionados arriba, respectivamente (1-5):

1. $\eta_1\#\eta_2\#\cdots\#\eta_n\#\gamma_1 A\gamma_2 \Rightarrow$
   $\eta_1\#\eta_2\#\cdots\#\eta_n\#\gamma_1\alpha\gamma_2$

   Con $\gamma_1 \in T^*$ y $\gamma_2 \in (N \cup T)^*$

2. - Si $\mu \in I$: $\mu\eta_1\#\eta_2\#\cdots\#\eta_n\#\gamma_1 A\gamma_2 \Rightarrow \mu\eta_1\#\eta_2\#\cdots\#\eta_n\#\gamma_1\alpha\gamma_2$
   - Si $\mu = \# : \gamma_1 A\gamma_2 \Rightarrow \gamma_1\alpha\gamma_2$

   Con $\gamma_1 \in T^*$ y $\gamma_2 \in (N \cup T)^*$

3. $\eta_1\#\eta_2\#\cdots\#\eta_{i-1}\#\eta_i\#\eta_{i+1}\#\cdots\#\eta_n\#\gamma_1 A\gamma_2 \Rightarrow$
   $\eta_1\#\eta_2\#\cdots\#\eta_{i-1}\#\mu\eta_i\#\eta_{i+1}\#\cdots\#\eta_n\#\gamma_1\alpha\gamma_2$

   Con $\gamma_1 \in T^*$, $\gamma_2 \in (N \cup T)^*$, $\alpha = a|aA$ según indique la regla de producción aplicada, y siendo i el mínimo número de pila tal que $(\mu, prim(\eta_i)) \in O$. Notamos $prim(s)$ al primer elemento de la cadena s, que usaremos para referirnos al *tope de pila*.

   Si esto no ocurre para ninguna de las n pilas, se agrega μ en una nueva pila al final, es decir, la derivación en un paso queda:

   $\eta_1\#\eta_2\#\cdots\#\eta_n\#\gamma_1 A\gamma_2 \Rightarrow$
   $\eta_1\#\eta_2\#\cdots\#\eta_n\#\mu\#\gamma_1\alpha\gamma_2$

   Se puede notar que si nos focalizamos únicamente en las multistacks, al aplicar un paso $\xi_1\gamma_1 \Rightarrow \xi_2\gamma_2$, la transformación de la multistack $\xi_1 \mapsto \xi_2$ que ocurre al aplicarlo depende únicamente de μ, pero en particular no depende de α, y es además una transformación determinista por la definición dada. A esta transformación la llamaremos un *push* de μ en la multistack ξ_1, transformación cuyo resultado es ξ_2.

4. $\mu\eta_1\#\eta_2\#\cdots\#\eta_{i-1}\#\eta_i\#\eta_{i+1}\#\cdots\#\eta_n\#\gamma_1 A\gamma_2 \Rightarrow$
   $\eta_2\#\cdots\#\eta_{i-1}\#\eta_1\#\eta_i\#\eta_{i+1}\#\cdots\#\eta_n\#\gamma_1\alpha\gamma_2$

   Con $\eta_1 \in I^+$, $\gamma_1 \in T^*$, $\gamma_2 \in (N \cup T)^*$, $\alpha = aA|A|\lambda$ según indique la regla de producción aplicada, y siendo i el mínimo número de pila tal que $(prim(\eta_1), prim(\eta_i)) \in O$.

   Si esto no ocurre para ninguna de las n pilas, se agrega la nueva pila η_1 al final, es decir, la derivación en un paso queda:

   $\mu\eta_1\#\eta_2\#\cdots\#\eta_n\#\gamma_1 A\gamma_2 \Rightarrow$
   $\eta_2\#\cdots\#\eta_n\#\eta_1\#\gamma_1\alpha\gamma_2$

   En el caso de ser η_1 vacía, directamente se elimina la primera pila que contenía μ, es decir:

$\mu\#\eta_2\#\cdots\#\eta_n\#\gamma_1 A\gamma_2 \Rightarrow$
$\eta_2\#\cdots\#\eta_n\#\gamma_1\alpha\gamma_2$

Notar que de manera similar al caso anterior, la transformación en la multistack es determinista y no depende de α (en este caso tampoco depende de μ, pero solo es posible aplicar esta regla en la derivación cuando el símbolo tope de la primera pila, que es el que se va a popear, coincide con el μ indicado en la regla pop de la gramática). De manera análoga al caso anterior, a esta operación la llamamos un *pop* de μ sobre la multistack.

5. Para simplificar el razonamiento y exposición, conviene pensar a las reglas pop-push como una combinación atómica de un pop seguido de un push.

   Más precisamente, si tenemos una regla de producción de la forma:

$$R_{\text{pop-push}} \equiv A \underset{\overline{\mu}|\mu_s}{\rightarrow} \alpha$$

   Siendo $\alpha = aA|A|\lambda$; es válida la derivación en un paso:

   $\xi_1\gamma_1 A\gamma_2 \Rightarrow \xi_3\gamma_1\alpha\gamma_2$, con ξ_1, ξ_3 multistacks y $\gamma_1 \in T^*$, $\gamma_2 \in (N \cup T)^*$,

   exactamente cuando existe una multistack ξ_2 tal que ξ_2 es el resultado de hacer un pop de μ en ξ_1, y ξ_3 es el resultado de hacer un push de μ_s en ξ_2.

La relación de derivación $\overset{*}{\Rightarrow}$ es como en los casos de CFG y GIG, la clausura reflexivo transitiva de $\Rightarrow$.

Se mantiene la restricción a realizar únicamente derivaciones por izquierda como en el caso de GIG.

El *lenguaje generado* por una MGIG es el conjunto $\mathcal{L} = \{s \in T^* | S \overset{*}{\Rightarrow} s\}$, es decir, todas las cadenas de terminales que se derivan del símbolo inicial, empezando y terminando con la multistack vacía. Decimos que un lenguaje es un MGIL (*Multiple Global Index Grammar*) si es generado por una MGIG. Denotamos también MGIL al conjunto de todos estos lenguajes.

Dada una cadena $s \in \mathcal{L}$, un *árbol de derivación* de s a partir de la MGIG se define de igual manera que para CFG y GIG, con la única salvedad de que ahora los nodos deben estar etiquetados con producciones propias de una MGIG y de manera tal que la derivación por izquierda correspondiente al árbol sea una derivación válida en MGIG.

2.3.2. Sobre la relación de sufijos

Debemos especificar más claramente qué se entiende por sufijo, ya que tal noción no está definida para un conjunto de índices completamente abstracto I como el que veníamos manejando por ejemplo en el caso de las GIG. La forma práctica de entender la noción de sufijo es considerar a los índices de I no como elementos atómicos abstractos, sino como cadenas de caracteres sobre algún alfabeto Σ_I. De esta forma un $x \in I$ será sufijo de un $y \in I$ cuando lo es como cadena en Σ_I^* [3]. En una aplicación práctica, los índices utilizados en la gramática podrían ser tokens resultantes del análisis léxico de un texto de entrada, y la relación de "ser sufijo" entre índices aquí considerada podría ser directamente la relación correspondiente entre los tokens del texto original.

[3] Sufijo propio: para estos efectos una cadena no debe ser considerada sufijo de sí misma

Notar que nunca utilizamos la relación de "ser sufijo" en la definición formal conjuntista de la MGIG (la relación no está contenida en la tupla), y tampoco la utilizaremos cuando a continuación definamos las reglas de derivación asociadas a la MGIG. La importancia de la relación radica únicamente en **restringir** la forma posible del **conjunto de producciones pop-push** presentes en la gramática, de forma que para que la MGIG sea válida el conjunto de índices debe poder interpretarse como cadenas en un Σ_I^* de tal manera que en toda producción de tipo pop-push, el índice μ_s sea sufijo del μ de acuerdo a esta interpretación.

Por ejemplo, si existieran a la vez las reglas:

$$A \underset{\overline{\mu_1}|\mu_2}{\rightarrow} aA$$

$$A \underset{\overline{\mu_2}|\mu_3}{\rightarrow} A$$

$$A \underset{\overline{\mu_3}|\mu_1}{\rightarrow} \lambda$$

Aunque cada una de estas reglas de producción por sí misma tiene la forma permitida, y puede formar parte de una MGIG, no estaríamos en presencia de una MGIG, ya que no existe interpretación posible de los índices μ_1, μ_2, μ_3 como cadenas de forma que μ_2 sea sufijo de μ_1, μ_3 sea sufijo de μ_2 y μ_1 sea sufijo de μ_3, al ser la relación de sufijos entre cadenas de caracteres una relación de orden estricto y por lo tanto acíclica.

Por otra parte, cualquier relación acíclica admite una interpretación con cadenas y sufijos como la que hemos pedido, de manera que la única restricción sobre las producciones pop-push de las MGIG es que los pares (μ, μ_s) presentes en el conjunto de producciones no contengan ningún ciclo. Esto es porque una relación acíclica admite un ordenamiento topológico, y dados los índices en orden topológico $x_1, \cdots, x_n$, podemos asignar a x_i la cadena a^{n+1-i}, lo cual garantiza que todo par presente en la relación relaciona una cadena con un sufijo de la misma, en concordancia con lo requerido por la definición de MGIG.

2.3.3. Observaciones, y variantes posibles en la definición

Se debe notar que la característica fundamental de las GIGs que requiere que toda operación push tenga un símbolo terminal en la posición inicial del lado derecho se mantiene en las MGIGs, ya que será importante cuando realicemos cálculos y demostraciones de complejidad de los algoritmos. Para aumentar el poder expresivo de las GIGs sin eliminar esta restricción es que se introducen las reglas pop-push, que pueden hacer push sin introducir un terminal en la forma sentencial, pero **solo a costa de reemplazar un índice en el multistack por un sufijo del mismo**. Esto permitirá acotar la cantidad de operaciones de pila realizadas.

También se puede observar que en la definición dada hemos exigido una forma más restringida para las reglas de producción que operan con la pila, en comparación a las que se permiten en las GIGs. Veremos en 2.3.6. que tal restricción no quita poder expresivo alguno, y que de hecho puede restringirse aún más la forma de las producciones sin perder poder expresivo.

Por este mismo motivo, las pequeñas diferencias entre nuestra definición y la dada en [8] no modifican el poder expresivo ni la forma de los algoritmos involucrados. Las dos diferencias fundamentales son:

- En [8] se permite que $I \cap T \neq \emptyset$, mientras que nosotros hemos exigido en la definición formal que sean conjuntos disjuntos. Esta diferencia es puramente formal, ya que si tenemos un cierto símbolo $t \in T$ que se usa tanto como terminal como índice de multistack, podemos cambiarlo por dos símbolos $t_T \in T$ y $t_I \in I$ para ser usados respectivamente como terminal y como índice de multistack, ya que el uso de estos en la gramática nunca es ambiguo porque nunca se mezclan índices (que aparecen debajo de $\rightarrow$ en las producciones, y en el multistack ξ en las formas sentenciales) con terminales (que aparecen en el lado derecho de las producciones, y en la "parte derecha" γ de las formas sentenciales).

- En [8] se permiten producciones que modifican el multistack de la forma $A \rightarrow aB|B$ con $A \neq B$, mientras que nosotros nos hemos restringido a $A \rightarrow aA|A$. El motivo es que estas últimas son las únicas que necesitamos para modelar SAT, que es nuestro objetivo principal y será estudiado en 3. Ambas versiones son equivalentes en poder expresivo, en virtud de la reducción dada en 2.3.6.

La definición de MGIG dada anteriormente es la que utilizamos en todo este trabajo, y en particular la que consideramos para realizar la implementación de los algoritmos. Describimos a continuación algunas posibles variaciones que se podrían realizar sobre esta definición de MGIG que no han sido exploradas en este trabajo.

En [13], el autor considera una definición de MGIG alternativa, donde no solamente las producciones que realizan operaciones de pila se restringen a tener una forma regular, sino **todas** las producciones tienen ahora tal forma. Es decir todas las producciones tienen la forma $A \underset{\mu}{\rightarrow} uB$ o $A \underset{\mu}{\rightarrow} u$, siendo $u \in T \cup \{\lambda\}$ y manteniendo la restricción de que $u \neq \lambda$ cuando μ indica una operación de push, así como la restricción de pop-push a sufijos. Las MGIG así definidas resultan ser un subconjunto de las MGIG que utilizaremos en este trabajo, y por lo tanto su problema de reconocimiento está en NP por la misma demostración que damos en 2.3.8. Como notamos en 3.4.5, el problema SAT se puede seguir modelando con gramáticas este tipo, y por lo tanto el problema de reconocimiento de MGIG para esta variante será al igual que veremos para nuestra definición, NP-completo. También en el mismo trabajo, en la formulación más restringida de MGIG utilizada, la lista de stacks ya no se ordena de acuerdo a una relación de preorden O sino que se ordena directamente por tiempo de creación, utilizándose siempre la última stack creada.

Otras variantes naturales de MGIG son aquellas que resultan de eliminar o reducir restricciones que hemos impuesto en la definición. Un primer ejemplo serían las gramáticas que se obtienen al eliminar la restricción de pop-push a sufijos. Como utilizaremos esa restricción en nuestra demostración de que el reconocimiento de MGIG está en NP (2.3.8), es una pregunta pendiente si eliminar esa restricción aumenta el poder expresivo de MGIG o no, y si el problema de reconocimiento sigue estando en NP.

Por otra parte, en la definición dada, la forma de aplicar una regla de producción es totalmente determinista, de manera que la única elección es qué regla aplicar, y por lo tanto al igual que en GIG una derivación está caracterizada por la secuencia ordenada de reglas de producción que se aplican. Ahora bien, en un push se debe ubicar un elemento en la multistack (o como resultado de un pop, se debe ubicar un espacio adecuado para una pila resultante), para lo cual se busca **la primera** pila en donde encaja (que es justamente la que tiene un tope de pila mínimo). Como es posible tener empates con nuestra definición de O (al ser la relación un preorden), se podría pensar en una versión de la gramática donde se tiene la libertad de pushear a **cualquiera** de las pilas mínimas donde el elemento

agregado encaja, y similarmente popear de **cualquiera** de las pilas con mínimo tope de pila. Al no fijar el elemento utilizado, tal versión introduce más no determinismo y por lo tanto un espacio de derivaciones posibles mayor, pero no necesitamos tal no determinismo para el problema central que nos planteamos modelar mediante MGIG (SAT).

Se puede notar que cambiar la relación de preorden total por un orden total sobre los símbolos no evita este problema, ya que es posible tener varias pilas con **el mismo** tope de pila (como resultado de una operación pop), y entonces ni siquiera el ordenamiento total puede distinguirlas. Un ejemplo de esto sería una secuencia de operaciones como `push C, push B, push A, push B, pop A`, que da lugar a $C\# \Rightarrow BC\# \Rightarrow ABC\# \Rightarrow ABC\#B\# \Rightarrow BC\#B\#$.

Finalmente, también se podría eliminar la restricción de que O sea un preorden total. Si limitamos O únicamente a ser un preorden, y mantenemos el funcionamiento de la multistack de manera que los elementos insertados x se insertan siempre en el primer lugar donde hay un y tal que $(x, y) \in O$, entonces para el preorden mínimo[4], la multistack termina funcionando como una cola FIFO de "contadores", uno por cada símbolo, de manera que cuando un pop quita la última aparición de un símbolo, el próximo push lo ubicará recién al final de toda la multistack. En particular si nunca se hace push de un índice que ya esté presente en el multistack, el funcionamiento es exactamente el de una cola FIFO.

Este comportamiento está intuitivamente bastante alejado de la idea de multistack planteada, y no hemos explorado esta posibilidad. Una variación aún mayor se podría obtener permitiendo una relación O completamente arbitraria. En este caso, notemos que por ejemplo con la relación vacía, nunca se insertará algo en una pila existente, prefiriéndose siempre crear una pila nueva **al final** del multistack. En este caso, se tendrá un modelo muy similar a GIG pero en lugar de tener una sola pila global LIFO, se tiene una sola cola global FIFO.

2.3.4. Ejemplo

Consideremos el lenguaje:

$$\mathcal{L} = \{wc^n | w \in \{a, b\}^* \text{ contiene } s \text{ aes y } t \text{ bes y } 0 \leq n \leq t - s\}$$

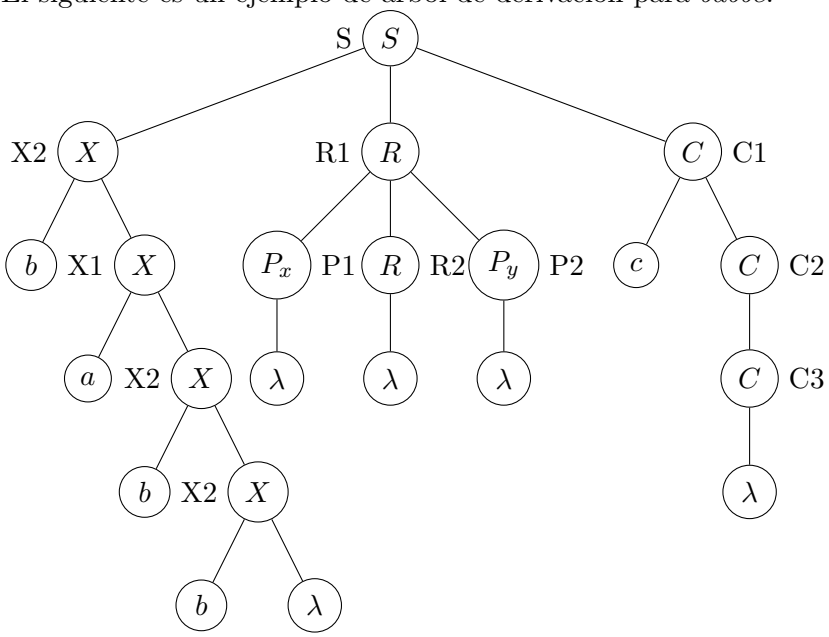
La siguiente MGIG lo genera, con símbolo inicial S, y $I = \{x, y\}$ con O tal que $x < y$:

[4] Es decir, aquel en que $(x, y) \in O \Leftrightarrow x = y$

$$
\begin{array}{rrcl}
(S) & S & \to & XRC \\
(X1) & X & \underset{x}{\to} & aX \\
(X2) & X & \underset{y}{\to} & bX \\
(X3) & X & \to & \lambda \\
(R1) & R & \to & P_xRP_y \\
(R2) & R & \to & \lambda \\
(C1) & C & \underset{\overline{y}}{\to} & cC \\
(C2) & C & \underset{\overline{y}}{\to} & C \\
(C3) & C & \to & \lambda \\
(P1) & P_x & \underset{\overline{x}}{\to} & \lambda \\
(P2) & P_y & \underset{\overline{y}}{\to} & \lambda
\end{array}
$$

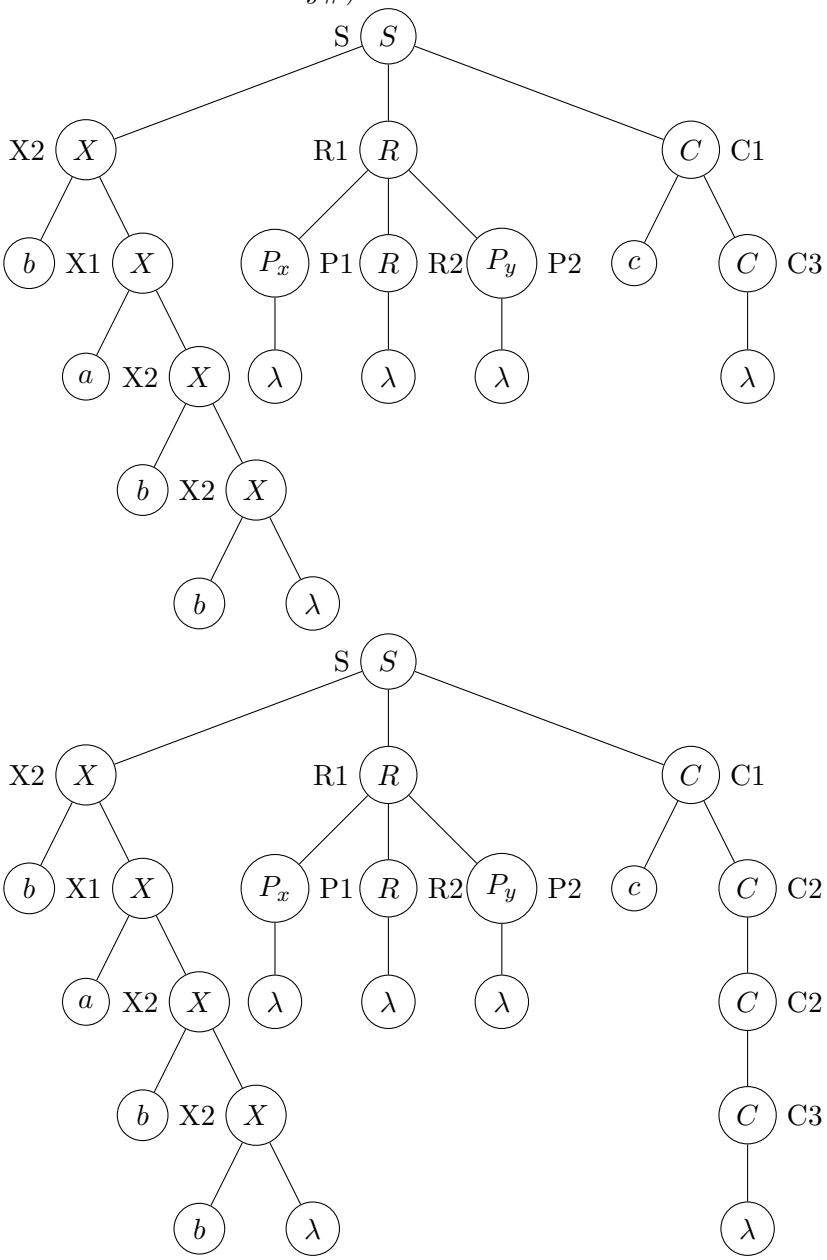
Si bien hay bastantes reglas, se puede notar la simpleza conceptual de la construcción: Primero en la fase X se recorre w por completo y se pushean a la pila las cantidades de a y b directamente (índices x e y). Luego como el orden especifica $x < y$, en la multistack tendremos primero las x, para cuyo proceso simplemente nos aseguraremos de hacer un pop de y por cada pop de x (R1). Esto no es más que computar $t - s$ explícitamente, ya que esa será la cantidad de y que quedarán en la pila al terminar de procesar la fase R en una derivación exitosa (Ya que no se puede dejar ninguna x o no podrá ser vaciada la pila). Finalmente debemos permitir a lo sumo $t - s$ ocurrencias de c, para lo cual se hace pop de un y por cada c insertada en la forma sentencial (C1), y como no es necesario llegar a usar $t - s$ ocurrencias de c, permitimos vaciar el remanente de y en la pila (C2).

El siguiente es un ejemplo de árbol de derivación para *babbc*:



Los siguientes dos son ejemplos de árboles *inválidos*, es decir que no son árboles de derivación para *babbc* con respecto a $\mathcal{L}$. El motivo es que si bien ambos son árboles de derivación válidos para *babbc* si observamos únicamente las características correspondientes al caso libre de contexto, al considerar las operaciones de pila realizadas en una derivación por izquierda como la que indica el árbol se producen operaciones inválidas (como en el

segundo árbol, donde la segunda aplicación de C2 intenta hacer pop de una multistack vacía), o bien no se termina con la multistack vacía (como en el primer árbol, en el cual al finalizar la multistack es $y\#$).



Esta construcción directa aprovecha el poder expresivo de MGIG y codifica todas las restricciones pedidas por el lenguaje de ejemplo de manera directa, y de manera fundamentalmente independiente para cada restricción, con lo cual es sencillo modelar lenguajes similares con restricciones levemente distintas. Por ejemplo hubiera sido igual de fácil modelar $\mathcal{L}_2 = \{wc^s d^t | w \in \{a, b\}^*$ contiene s aes y t bes$\}$ de manera casi idéntica.

Estas propiedades para lenguajes de esta forma son muy distintas al caso de CFG. Por ejemplo el mismo lenguaje $\mathcal{L}$ es reconocido mediante la siguiente CFG:

$$\begin{aligned} S &\rightarrow I \\ S &\rightarrow IbSc \\ I &\rightarrow aIbI \\ I &\rightarrow bIaI \end{aligned}$$

Sin embargo esta gramática aprovecha propiedades muy particulares de la definición de $\mathcal{L}$ y no generaliza fácilmente a cambios menores de las restricciones, como es el caso de $\mathcal{L}_2$.

2.3.5. CFL $\subsetneq$ MGIL

Es inmediato que todo lenguaje libre de contexto es generado por una MGIG, ya que las CFG son por definición (tomando un conjunto de índices y una relación de orden entre índices ambos vacíos) también MGIG, con el mismo lenguaje generado. Al igual que vimos para el caso de GIG, mostraremos algunos ejemplos de lenguajes que no son libres de contexto que son reconocidos fácilmente mediante GIG.

2.3.5.1 $a^n b^n c^n$

Como primer ejemplo de un lenguaje que no es libre de contexto que puede ser generado por una MGIG, consideremos:

$$\{a^n b^n c^n | n \geq 0\}$$

Como ya explicamos para el caso de las GIGs, este lenguaje no es libre de contexto, pero sin embargo la siguiente MGIG sencilla lo genera (con símbolo inicial A, y una relación de orden entre índices donde $I_{bc} < I_c$):

$$A \underset{I_{bc}}{\rightarrow} aA$$

$$A \rightarrow B$$

$$B \underset{\overline{I_{bc}}|I_c}{\rightarrow} bB$$

$$B \rightarrow C$$

$$C \underset{\overline{I_c}}{\rightarrow} cC$$

$$C \rightarrow \lambda$$

Esto muestra que las MGIG son una extensión propia de las CFG. Notar que no hemos podido utilizar directamente la misma gramática que utilizamos previamente para GIG en este ejemplo, ya que la regla push utilizada en aquel no satisface los requisitos necesarios para formar parte de una MGIG. En cambio hemos utilizado de una manera muy natural las reglas de tipo push y pop-push junto a la noción de orden entre índices, para codificar la introducción de una **dependencia a futuro** (que queda almacenada en una stack diferente a la stack que se encuentra al frente del multistack). Esta idea central de poder

hacer push de dependencias futuras **ordenadas** (de acuerdo a la relación de preorden $O \subseteq I \times I$, que permite forzar un cierto orden para el procesamiento de los índices) es una característica novedosa esencial de las MGIG.

Si bien acabamos de mostrar una gramática para este lenguaje que resulta directa y natural aprovechando las características de las MGIG, en 2.3.6 veremos cómo adaptar cualquier GIG a la forma pedida por la definición de MGIG, lo cual daría una gramática alternativa basada en la que dimos en la sección anterior para GIG.

2.3.5.2 Reconocimiento de lenguajes de copias

Otro ejemplo interesante es el lenguaje de copias:

$$\mathcal{L}_{copy} = \{ww | w \in T^*\}$$

Que no es un CFL cuando $|T| > 1$ [27]. La siguiente MGIG genera este lenguaje, por ejemplo para el caso $T = \{a, b, c\}$:

$$I = \{Px, Py, Pz, x, y, z\}, \text{ con un preorden tal que}$$
$$Pt_1 < t_2 \quad \forall t_1, t_2 \in \{x, y, z\}, \text{ y no se da el } < \text{ en ningún otro caso}$$

$$\begin{array}{llll}
(A1) & A & \underset{Px}{\rightarrow} & aA \\
(A2) & A & \underset{Py}{\rightarrow} & bA \\
(A3) & A & \underset{Pz}{\rightarrow} & cA \\
(A4) & A & \rightarrow & B \\
(B1) & B & \underset{\overline{Px}|x}{\rightarrow} & B \\
(B2) & B & \underset{\overline{Py}|y}{\rightarrow} & B \\
(B3) & B & \underset{\overline{Pz}|z}{\rightarrow} & B \\
(B4) & B & \rightarrow & C \\
(C1) & C & \underset{\overline{x}}{\rightarrow} & aC \\
(C2) & C & \underset{\overline{y}}{\rightarrow} & bC \\
(C3) & C & \underset{\overline{z}}{\rightarrow} & cC \\
(C4) & C & \rightarrow & \lambda
\end{array}$$

Por la forma regular de las reglas en este ejemplo, es claro que en una derivación exitosa se usarán una cierta cantidad de veces las reglas A1,A2,A3, luego la A4, luego una cierta cantidad de veces las B1,B2,B3, luego la B4, luego una cierta cantidad de veces las C1,C2,C3 y finalmente la regla C4. Esto fuerza a que por cada uso de las A1,A2,A3 será necesario posteriormente un uso de las correspondientes C1,C2,C3. Además cuando se hace push de Pt, solo las reglas B1,B2,B3 utilizan esos valores de la pila, y por lo tanto es necesario utilizarlas. Cada aplicación de B1,B2,B3 a un Pt hace push de un índice t *en una segunda* pila (pues por la relación de preorden en uso, t no encaja en la primera pila que contiene valores de la forma Pt_2). En otras palabras, los índices t que quedan almacenados en esta segunda pila quedan almacenados en orden inverso a los Pt originales de la primera pila, y como estos a su vez quedan en orden inverso a la secuencia de caracteres a, b, c generada durante la creación de dicha pila, en la pila restante al aplicar la regla B4 en una derivación exitosa los índices se irán popeando en exactamente el mismo

orden en que estaban los a, b, c de dicha cadena. Esto hace que se genere una segunda copia idéntica a esa cadena generada por las reglas A1,A2,A3.

La argumentación anterior muestra que toda cadena generada por la MGIG está en $\mathcal{L}_{copy}$, y de la misma manera es claro que toda cadena de $\mathcal{L}_{copy}$ puede ser generada de esta forma, con solo elegir el fin de la primera copia de w como el momento para aplicar la regla A4. Esto prueba que $\mathcal{L}_{copy}$ es generado mediante esta MGIG.

Es evidente que extendiendo esta construcción de manera natural, agregando más fases de procesamiento y más índices de multistack, podríamos generar para cada n el lenguaje:

$$\mathcal{L}_{n-copies} = \{w^n | w \in T^*\}$$

A modo ilustrativo damos el ejemplo para $\mathcal{L}_{3-copies}$ con $T = \{a, b\}$ y un preorden donde $PPPt_1 < PPt_2 < Pt_3 < t_4$:

$$\begin{array}{lcl}
A & \underset{PPPx}{\rightarrow} & aA \\
A & \underset{PPPy}{\rightarrow} & bA \\
A & \rightarrow & B \\
B & \underset{\overline{PPPx}|PPx}{\rightarrow} & B \\
B & \underset{\overline{PPPy}|PPy}{\rightarrow} & B \\
B & \rightarrow & C \\
C & \underset{\overline{PPx}|Px}{\rightarrow} & aC \\
C & \underset{\overline{PPy}|Py}{\rightarrow} & bC \\
C & \rightarrow & D \\
D & \underset{\overline{Px}|x}{\rightarrow} & D \\
D & \underset{\overline{Py}|y}{\rightarrow} & D \\
D & \rightarrow & E \\
E & \underset{\overline{x}}{\rightarrow} & aE \\
E & \underset{\overline{y}}{\rightarrow} & bE \\
E & \rightarrow & \lambda
\end{array}$$

2.3.6. GIL $\subseteq$ MGIL

Veamos que para cualquier GIG, es posible construir una MGIG que genera el mismo lenguaje. Notemos que tomando una relación de preorden $O = I \times I$, es decir, todos los índices iguales entre sí, las operaciones de pila funcionan exactamente igual que en el caso de GIG, trabajando con una única pila.

Por lo tanto la única restricción que tenemos es que si la GIG de partida contiene reglas de producción con operaciones de pila, que no están en la forma requerida por la definición de MGIG, debemos reemplazarlas por una o más reglas de MGIG equivalentes. La construcción de dichas reglas es inmediata:

push Si se tiene una regla $A \underset{\mu}{\rightarrow} a\beta$, se reemplaza por el par de reglas:

$$A \rightarrow P_{\mu}^{A}\beta$$

$$P_{\mu}^{A} \underset{\mu}{\rightarrow} a$$

Donde P_{μ}^{A} es un no terminal nuevo que no exista en la GIG original.

pop Si se tiene una regla $A \underset{\overline{\mu}}{\rightarrow} \alpha$, se reemplaza por el par de reglas:

$$A \rightarrow P_{\overline{\mu}}^{A} \alpha$$

$$P_{\overline{\mu}}^{A} \underset{\overline{\mu}}{\rightarrow} \lambda$$

Donde $P_{\overline{\mu}}^{A}$ es un no terminal nuevo que no exista en la GIG original.

Como existe una única regla de producción para los no terminales nuevos que se agregan como resultado de esta construcción, es inmediato que una derivación exitosa debe utilizar necesariamente la regla para el P_{μ}^{A} o $P_{\overline{\mu}}^{A}$ correspondiente inmediatamente después de la regla libre de contexto introducida en reemplazo de la original de la GIG. Como el efecto sobre la forma sentencial de aplicar ambas reglas de producción en sucesión es idéntico al de haber aplicado la regla original de la GIG, ambas son equivalentes y generan el mismo lenguaje.

2.3.7. Sobre la pregunta GIL vs MGIL

Hemos probado que GIL $\subseteq$ MGIL, y por lo tanto es natural preguntarse si vale la igualdad. En [13], el autor analiza el poder expresivo de GIG y MGIG relacionándolos con la noción de multiplanaridad de dependencias. La definición de MGIG utilizada en dicho trabajo es más restringida, pues permite únicamente producciones de tipo regular (ya no libres de contexto). Sin embargo el ejemplo dado allí es igualmente relevante para nuestros propósitos, y lo reproducimos a continuación.

En el artículo el autor muestra una GIG para reconocer lenguajes de múltiples copias, con un enfoque diferente al mostrado anteriormente mediante MGIG. Además de esto se da el siguiente ejemplo de lenguaje que **no parece ser GIL**:

$$\mathcal{L}_{cr4} = \{a^n b^m c^l d^n j^k e^l f^m g^k h^m i^l | n, m, l, k \geq 0\}$$

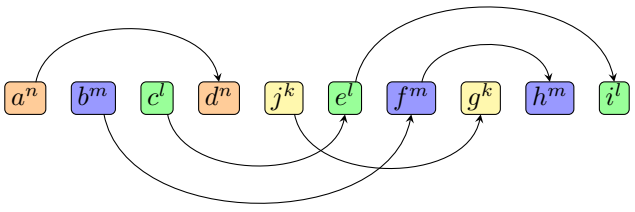


Fig. 2.1: Lenguaje $\mathcal{L}_{cr4}$, con las dependencias representadas

El razonamiento es que heurísticamente, debe ser posible dibujar de manera 2-planar todas las dependencias de cualquier forma sentencial (incluyendo la pila) en una derivación

exitosa a través de una GIG, pero por la forma en que las dependencias se entrecruzan en este ejemplo, no hay manera 2-planar de dibujarlas sin importar cómo se arme la derivación. Esta afirmación de que $\mathcal{L}_{cr4}$ no es GIL es una conjetura motivada por este razonamiento pero en el artículo mencionado no está demostrado por ahora. Una forma posible de demostrarlo podría ser desarrollar un "Pumping Lemma" adecuado para GIG, similar a los conocidos para CFL y para lenguajes regulares.

Este ejemplo candidato a no ser GIL es fácilmente reconocible mediante la siguiente MGIG (con índices $I_d < I_{ei} < I_{fh} < I_g < I_h < I_i$, y símbolo inicial A):

$$\begin{array}{rcl} A & \to & B \\ B & \to & C \\ C & \to & D \\ D & \to & J \\ J & \to & E \\ E & \to & F \\ F & \to & G \\ G & \to & H \\ H & \to & I \\ I & \to & \lambda \end{array}$$

$$\begin{array}{rcl} A & \underset{I_d}{\to} & aA \\ B & \underset{I_{fh}}{\to} & bB \\ C & \underset{I_{ei}}{\to} & cC \\ D & \underset{\overline{I_d}}{\to} & dD \\ J & \underset{I_g}{\to} & jJ \\ E & \underset{\overline{I_{ei}}|I_i}{\to} & eE \\ F & \underset{\overline{I_{fh}}|I_h}{\to} & fF \\ G & \underset{\overline{I_g}}{\to} & gG \\ H & \underset{\overline{I_h}}{\to} & hH \\ I & \underset{\overline{I_i}}{\to} & iI \end{array}$$

Es evidente que la construcción dada es general, y puede expresar cualquier grafo finito de dependencias de manera análoga. Esto muestra el poder expresivo irrestricto en cuanto a cruce de dependencias (finitas) en MGIGs.

Es interesante mencionar que el ejemplo de SAT central a este trabajo no nos sirve directamente para el propósito de mostrar un MGIL que no sea GIL, ya que todas las gramáticas que utilizamos son para una cantidad n de variables **fija**, por lo que son lenguajes esencialmente finitos (salvo cláusulas repetidas). Lo interesante en términos de poder expresivo en ese caso es **el tamaño** de la gramática necesaria para codificar SAT, que en MGIG es polinomial en el tamaño de la fórmula de entrada (y se codifica de una manera simple y natural en términos de las cláusulas que faltan satisfacer en un momento dado), mientras que en GIG no conocemos algo esencialmente mejor que una enumeración de las (finitas) fórmulas satisfacibles en la gramática misma, lo cual daría una gramática gigantesca y que "no expresa" el problema SAT en sí.

2.3.8. REC-MGIG $\in$ NP

Llamaremos REC-MGIG al problema de reconocer si una cadena de entrada pertenece al lenguaje generado por una MGIG dada. Veremos que REC-MGIG está en NP. Esto no es evidente a priori, ya que las MGIG constituyen una extensión al poder expresivo de las CFG, y existen otras extensiones sencillas muy naturales que no se cree que estén en NP.

Concretamente, las gramáticas sensibles al contexto, que son las de tipo 1 en la jerarquía de Chomsky y están justo por encima de las CFG en dicha jerarquía, son tales que el problema de reconocer si una cadena de entrada pertenece al lenguaje generado por una gramática sensible al contexto dada es PSPACE-completo[19]. Si un problema PSPACE-completo estuviera en NP, sería NP = PSPACE, lo cual pese a ser aún un problema abierto, contradice la conjetura más aceptada de que NP $\subsetneq$ PSPACE. Por lo tanto, el reconocimiento de gramáticas sensibles al contexto no parece estar en NP.

Demostración Para demostrar que REC-MGIG $\in$ NP, propondremos un certificado de tamaño polinomial en la entrada, y mostraremos un algoritmo polinomial que decida si el certificado es válido para la entrada, de forma que una instancia de REC-MGIG debe ser aceptada exactamente cuando existe un certificado que el algoritmo considera válido. Esta es la técnica usual para demostrar pertenencia a NP[20]. Para la demostración usaremos que REC-LCFG $\in P$, donde hemos llamado REC-LCFG al problema de decidir, dada una gramática libre de contexto y una cadena $\gamma \in (N \cup T)^*$, si γ es una forma sentencial posible de obtener a partir del símbolo inicial mediante **derivación por izquierda**. Esto se prueba en la sección siguiente.

El tamaño de la entrada para el problema REC-MGIG es $\Omega(|s| + |I| + R)$ símbolos, siendo R el tamaño total de las reglas de producción, s la cadena de entrada, y las otras variables las correspondientes a la MGIG de entrada[5].

Por otra parte, proponemos como certificado una descripción resumida de una derivación para la cadena s a partir de la MGIG. Esta descripción será una lista ordenada de l *pasos de derivación*, donde cada paso vendrá dado por una de las siguientes:

1. Un número entre 1 y R, que indique la producción a utilizar en el correspondiente paso de la derivación.

2. Una cadena $\gamma_m \in (N \cup T)^*$, que se deriva mediante derivación por izquierda a partir del no terminal de más a la izquierda de la forma sentencial (el cual debe reescribirse en este paso), utilizando únicamente producciones epsilon válidas para el multistack actual.

Los pasos de tipo 2 no son más que una abreviación de muchos pasos libres de contexto usuales, en los cuales no se realiza ninguna operación de multistack. Tal abreviación es un requisito técnico para que nuestra demostración funcione, ya que necesitaremos que exista un certificado de tamaño polinomial en el tamaño de la entrada.

Notaremos con M a la longitud máxima de una forma sentencial durante la derivación correspondiente. Esta es también una cota para la longitud de las cadenas γ_m (por ser subcadenas de la forma sentencial correspondiente). Tenemos que el tamaño del certificado propuesto es $\Omega(l)$ (pues hay que codificar l *pasos de derivación*) y $O(lM \lg R)$ símbolos

[5] Es posible (y razonable) codificar una MGIG con $\Theta(|I| + R)$ números entre 1 y $|I| + |N| + |T|$. La relación de preorden total $O \subseteq I \times I$ puede darse como la lista ordenada de clases de equivalencia que induce el preorden, que es una lista de listas con un total de $|I|$ elementos.

(pues como R es el tamaño de la gramática, cada símbolo se codifica con $\lg R$ bits, y cada paso de derivación involucra una cadena de como mucho M símbolos).

De lo anterior surge que la entrada para el algoritmo verificador contendrá $\Omega(|s|+|I|+R+l)$ símbolos. Comprobemos que es posible verificar la validez de esta descripción de la derivación de la cadena de entrada s en tiempo polinomial en el tamaño de la entrada del algoritmo verificador, es decir, en tiempo polinomial en $|s|+|I|+R+l$. Para eso usaremos que $M = O((|s| + |I| + R)^k)$ para cierto $k \in \mathbb{Z}_{>0}$, lo cual probaremos más adelante.

Basta para ello simular computacionalmente los l pasos de la derivación verificando su validez, y verificando que al final se obtenga la cadena deseada con la multistack vacía, ya que el resultado de cada una de las operaciones es determinista.

Cada paso de derivación que se corresponda directamente con una regla de producción de la MGIG toma claramente tiempo polinomial, ya que cada operación es simplemente un reemplazo del no terminal por el lado derecho de la regla de producción, y eventualmente una operación de multistack, ambas manipulaciones claramente polinomiales en $R + M$.

Además, cuando un paso indica que se debe derivar por reglas epsilon una cadena γ_m de longitud a lo sumo M, es posible hacer esta verificación en tiempo polinomial en $R+M$: Basta verificar que la cadena γ_m es una forma sentencial obtenible mediante derivación por izquierda de la CFG que contiene todas las reglas epsilon sin restricción de la MGIG, así como las reglas epsilon con restricciones que la multistack actual satisface: y esto puede hacerse en tiempo polinomial aprovechando que REC-LCFG $\in P$.

Es claro que cuando esta verificación es exitosa, la cadena s está en el lenguaje generado por la MGIG: En efecto, en este caso se ha comprobado que la secuencia de l pasos del certificado describe una derivación de s, con la salvedad de que en los pasos de operaciones epsilon, se da directamente una cadena objetivo que puede obtenerse por aplicación de reglas libres de contexto (mediante derivación por izquierda, como se hace en el caso de las MGIG), lo cual corresponde simplemente a abreviar pasos en la derivación.

Para concluir la demostración de que REC-MGIG $\in$ NP, falta probar que para toda entrada en el lenguaje REC-MGIG (es decir, una MGIG y una cadena s en su lenguaje generado) existe un certificado **de tamaño polinomial** de manera que la verificación produce un resultado positivo: es decir, que existe un árbol de derivación de s que puede describirse en l pasos de los anteriores, con formas sentenciales de longitud a lo sumo M, y con $lM \lg R = O((|s| + |I| + R)^k)$ para algún $k \in \mathbb{Z}_{>0}$.

Para esto comenzamos acotando la cantidad de veces que puede ser utilizada cada tipo de regla de producción (además de las epsilon).

push: Supongamos que tenemos una derivación exitosa de s en l pasos a partir de una cierta MGIG. Como cada regla de tipo push implica por la forma de las mismas que se agrega un terminal a la parte derecha de la forma sentencial, ninguna derivación en un paso puede eliminar terminales en la parte derecha de la misma, y se comienza con cero terminales mientras que se termina con $|s|$ terminales, de aquí se deduce que una derivación exitosa puede contener como máximo $|s|$ pasos en los que se utilice una regla de tipo push.

pop: Cada vez que hace pop, la cantidad de índices en total en el multistack disminuye en uno, y dicha cantidad únicamente aumenta al hacer un push. Por lo tanto la cantidad de operaciones pop posibles en una derivación exitosa es a lo más la cantidad de push, y ya hemos visto que esto es como mucho $|s|$.

pop-push: En estas operaciones la cantidad total de índices en el multistack se mantiene constante, ya que se hace pop de un índice pero luego push de un **sufijo** del mismo. La garantía de que se hace push de un sufijo es fundamental para acotar la cantidad posible de operaciones de este tipo en una derivación exitosa.

Dado un índice $\mu \in I$, podemos definir su *profundidad*, $p(\mu)$, como la cantidad de elementos $\mu_s \in I$ tales que μ_s es sufijo de μ (con la noción de sufijo explicada en la definición de MGIG, dada una interpretación de los índices como cadenas acorde a dicha definición). Claramente $p(\mu) \leq |I| \forall \mu \in I$.

Podemos definir la *profundidad total del multistack* como la suma de las profundidades de todos los índices en el multistack (con multiplicidad). Es claro que la única operación que aumenta la profundidad total del multistack es la aplicación de una regla de tipo push, y que las reglas de tipo pop-push implican una disminución de la profundidad total. Cada aplicación de una regla de tipo push puede aumentar la profundidad total en a lo más $|I|$, luego como la profundidad total es un entero no negativo y la cantidad total de push es a lo sumo $|s|$, concluimos que la cantidad total de aplicaciones de reglas de tipo pop-push en una derivación exitosa de s es a lo más $|s||I|$

Llamaremos T a la cantidad de operaciones de tipos no epsilon presentes en la derivación. En los puntos anteriores mostramos que $T = O(|s||I|)$.

Veamos ahora que podemos obtener una cota polinomial para la altura h del árbol de derivación. En efecto, tomemos un árbol de derivación para s a partir de la MGIG de altura mínima, y con la mínima cantidad posible de ramas (caminos desde la raíz hasta una hoja) con longitud igual a dicha altura mínima. Supongamos que dicha altura es $h > (T+1)(T+|s|+2)R$. Tomemos una rama del árbol de longitud h.

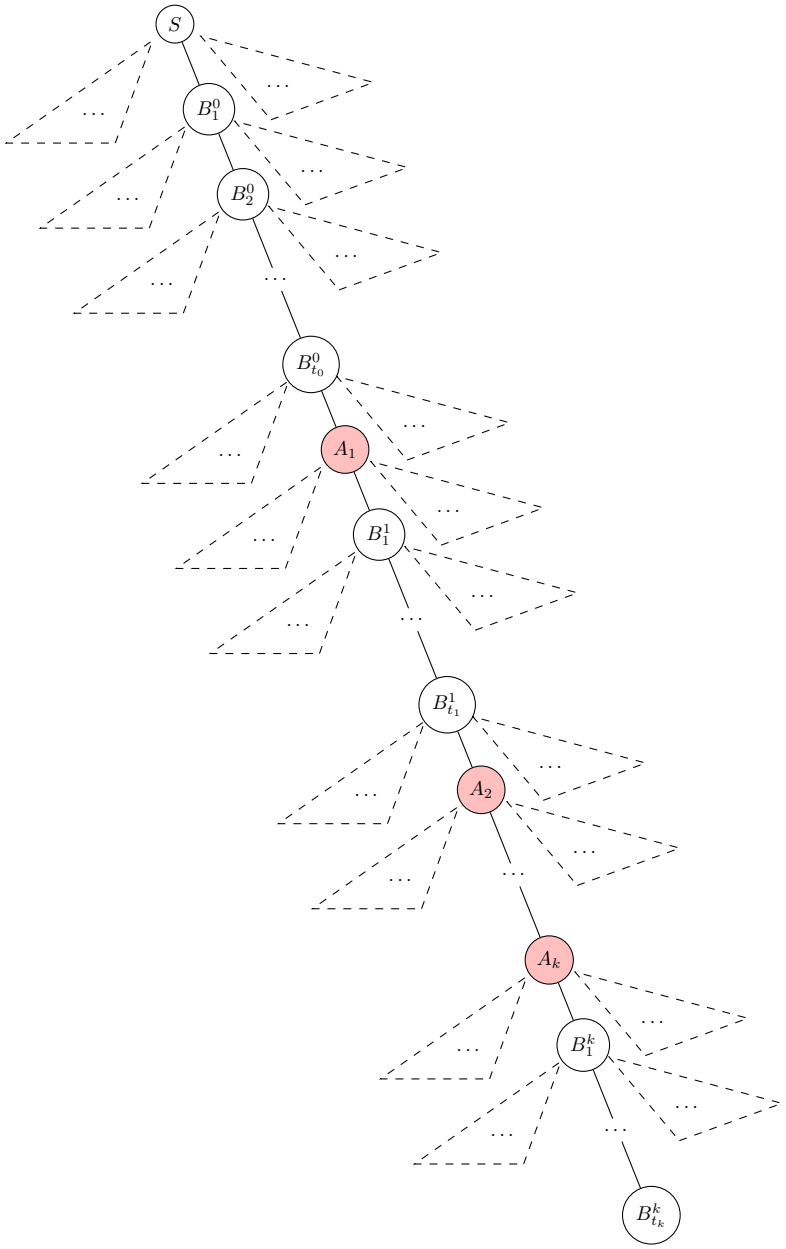


Fig. 2.2: Rama de longitud máxima en el árbol de derivación, con los $k \leq T$ nodos correspondientes a operaciones de multistack en color

Como se realizan únicamente T operaciones que no sean de tipo epsilon en la derivación, existe una subrama intermedia de longitud al menos $h' = \frac{h}{T+1} > (T+|s|+2)R$, de manera tal que en el recorrido en una derivación por izquierda no se produce ninguna operación que no sea de tipo epsilon en un nodo de esta subrama. Además, existirán al menos $n = \frac{h'}{R} > T + |s| + 2$ nodos distintos en esta subrama que corresponderán a un mismo no terminal $A \in N$, ya que no puede haber más que R no terminales distintos alcanzables en una derivación.

En términos de la derivación por izquierda, entre estos puntos de la derivación tendremos:

$$\eta\gamma_1 A\gamma_2 \stackrel{*}{\Rightarrow} \eta\gamma_1\alpha_1 A\beta_1\gamma_2 \stackrel{*}{\Rightarrow} \cdots \stackrel{*}{\Rightarrow} \eta\gamma_1\alpha_1\alpha_2\cdots\alpha_{n-1}A\beta_{n-1}\cdots\beta_2\beta_1\gamma_2$$

Con η el multistack, $\gamma_1, \alpha_1\alpha_2\cdots\alpha_{n-1} \in T^*$, $A \in N$, $\gamma_2, \beta_1, \beta_2\cdots, \beta_{n-1} \in (N \cup T)^*$,

Ahora bien, dado un β_i, decimos que es inútil si al considerar todas las operaciones que se realizan durante su procesamiento en la derivación (es decir, dentro del subárbol con raíz en alguno de sus no terminales), todas son de tipo epsilon y además en ninguna se agrega un símbolo terminal a la forma sentencial. Similarmente, decimos que un α_i es inútil cuando es vacío. Como hay más de $T + |s| + 1$ pares (α_i, β_i), alguno de ellos ha de contener ambos elementos del par inútiles: pues un par que no sea inútil debe contener en los subárboles correspondientes al menos una de las T operaciones que no son de tipo epsilon, o al menos uno de los $|s|$ símbolos terminales de la cadena, y claramente ninguno de estos elementos puede utilizarse en más de un par. Pero entonces, si (α_i, β_i) son ambos inútiles, el tramo de la derivación correspondiente al $i$ésimo salto mostrado en la rama puede omitirse por completo y obtenerse un nuevo árbol de derivación válido donde la rama inicial es más corta, y todas las demás ramas mantienen su longitud o se reducen. Esto es absurdo por la elección del árbol de derivación.

De todo esto concluimos que $h \leq (T + 1)(T + |s| + 2)R$, y entonces hay un árbol de derivación con altura $h = O((|s| + |I| + R)^5)$. Esto junto al hecho de que el tamaño de la parte derecha de una forma sentencial cualquiera en la derivación correspondiente será $O(hR+|s|)$, y como ya hemos justificado que el multistack se compone de a lo más $|s|$ índices, la forma sentencial tiene longitud $O((|s| + |I| + R)^6)$ en cualquier paso intermedio de la demostración. Esto implica $M = O((|s| + |I| + R)^6)$.

Ahora es posible codificar el árbol de derivación en la forma requerida para el algoritmo verificador: Cada operación que no sea epsilon se codifica directamente con su número de producción. Luego, entre dos de esas operaciones puede ser necesario describir una secuencia de operaciones $o_1, o_2, \cdots, o_k$ de tipo epsilon realizadas en la derivación por izquierda en el árbol. Consideramos el nodo correspondiente a la operación o_1 en el árbol, que tendrá a $o_1, o_2, \cdots, o_i$ como descendientes en el árbol, para algún $i \leq k$. Todas estas operaciones constituyen una derivación por izquierda a partir de un símbolo inicial de o_1, y entonces podemos resumirlas dando directamente la forma sentencial resultante, que puede verificarse como ya hemos explicado. Si $i < k$, entonces a o_{i+1} le corresponderá un nodo que no es descendiente del de o_1, y entonces podremos resumir las operaciones $o_{i+1}, o_{i+2}, \cdots, o_j$ correspondientes a descendientes del nodo de o_{i+1} de la misma manera. Continuamos de esta forma hasta incluir en el certificado las k operaciones. Cada vez que aplicamos un de estas secuencias de operaciones (salvo quizás la última que culmina con o_k), la forma sentencial tiene un no terminal menos que al comienzo. Luego solo puede haber $O(M)$ de estas secuencias.

Teniendo en cuenta esto, sabemos entonces que el certificado tendrá T pasos de operaciones que no son tipo epsilon, y luego a lo sumo $T+1$ tramos de operaciones epsilon entre ellas. Acabamos de ver que cada uno de esos tramos tendrá $O(M)$ pasos como máximo. Luego la cantidad total de pasos es $l = O(T + (T + 1)M) = O(TM)$. De la polinomialidad de T y M ya establecida, se concluye la polinomialidad de l, y con eso demostramos que la longitud del certificado es efectivamente polinomial, como queríamos. ∎

Como comentario adicional, podemos notar que si bien hemos tomado $|I| + R$ como el tamaño de la entrada, el caso en que $|I| > R$ no es interesante, pues implica que hay índices que nunca serán utilizados en una derivación porque no aparecen en las reglas de

producción. De esta forma el tamaño de una MGIG codificada en el caso interesante es efectivamente R, y los resultados sobre la complejidad de reconocer MGIG no dependen de manera relevante de la codificación propuesta.

2.3.8.1 REC-LCFG $\in$ P

Para la demostración de este resultado auxiliar a la demostración de la sección previa, usaremos que REC-CFG $\in P$, ya que las gramáticas libres de contexto pueden ser reconocidas en tiempo polinomial tanto en la cadena de entrada como en el tamaño de la gramática, mediante, por ejemplo, el algoritmo de Earley[18].

Notar que no toda forma sentencial posible de obtener es necesariamente una forma sentencial posible de obtener mediante derivación por izquierda. Como ejemplo simple tenemos la gramática:

$$\begin{aligned} A &\to BC \\ B &\to x \\ C &\to y \end{aligned}$$

Con A como símbolo inicial. En este caso By es una forma sentencial válida, ya que $A \Rightarrow BC \Rightarrow By$, pero no es posible de obtener mediante derivación por izquierda en este caso. Sí es posible obtener BC, xC y xy mediante derivación por izquierda.

El motivo por el que necesitamos abreviar los pasos de la derivación en la demostración anterior, es que una derivación libre de contexto general, cuando existen **producciones lambda**, puede requerir necesariamente una cantidad de pasos exponencial en el tamaño de la gramática (no así una profundidad del árbol de derivación, que como hemos mostrado en la sección anterior, siempre puede tomarse de altura polinomial). Por ejemplo:

$$\begin{aligned} A &\to xB \\ B &\to C_1C_1 \\ C_i &\to C_{i+1}C_{i+1}, \forall 1 \leq i < n \\ C_n &\to \lambda \end{aligned}$$

Esta gramática tiene tamaño $O(n)$, pero la única derivación posible de x tiene longitud $\Theta(2^n)$. Algoritmos como el de Earley son capaces de producir el árbol de derivación en tiempo polinomial expresándolo como un DAG y reutilizando subárboles repetidos (como los correspondientes a cada C_i en el ejemplo).

Daremos a continuación un algoritmo polinomial para REC-LCFG. Dada la CFG $G = (N, T, S, C)$ y una cadena $\gamma \in (N \cup T)^*$, debemos decidir si γ es una forma sentencial que puede obtenerse de G realizando únicamente pasos de derivación por izquierda. Definimos $G' = (N', T', S_L, C')$ con:

$$\begin{gathered} N' = A \cup \{A_L | A \in N\} \\ T' = T \cup \{A_T | A \in N\} \\ C' = C \cup U, \text{ con} \\ U = \{A_L \to A_T | A \in N\} \cup \{A_L \to A | A \in N\} \cup \\ \cup \{A_L \to X_1X_2 \cdots X_{i-1} l(X_i) t(X_{i+1}) t(X_{i+2}) \cdots t(X_k) | \\ A \to X_1X_2 \cdots X_k \in C, 1 \leq i \leq k, X_j \in N \cup T\} \\ l(A) = A_L \forall A \in N \\ t(A) = A_T \forall A \in N \\ l(x) = t(x) = x \forall x \in T \end{gathered}$$

A modo de ejemplo, a esta gramática:

$$S \to ABC$$
$$A \to AA|z$$
$$B \to x$$
$$C \to BC|y$$

Se agregan:

$$S_L \to S_T|S|A_LB_TC_T|AB_LC_T|ABC_L$$
$$A_L \to A_T|A|A_LA_T|AA_L|z$$
$$B_L \to B_T|B|x$$
$$C_L \to C_T|C|B_LC_T|BC_T|y$$

Es decir, se agrega un no terminal nuevo por cada uno existente, y un terminal nuevo por cada no terminal existente, así como reglas de producción nuevas para los no terminales nuevos. Es claro que esta nueva gramática puede computarse en tiempo polinomial en el tamaño de la original en base a esta descripción (y su tamaño es de hecho $O(R^2)$, siendo R el tamaño de la gramática original). Por lo tanto, dado $\gamma = X_1X_2\cdots X_n$, y definiendo $\gamma' = t(X_1)t(X_2)\cdots t(X_n)$: si probamos que γ es forma sentencial de $G \Leftrightarrow \gamma' \in \mathcal{L}(G')$, tendremos un algoritmo polinomial para el problema deseado, que consiste simplemente en computar esta nueva gramática y la cadena γ' y verificar su pertenencia a $\mathcal{L}(G')$, lo cual ya hemos mencionado que puede hacerse en tiempo polinomial.

Para esto basta notar que las cadenas de terminales que se derivan en G' de $A \in N$ son las mismas que en G, pues no es posible utilizar ninguna de las nuevas reglas de producción, ya que todas reescriben no terminales que no son alcanzables con las reglas de la gramática G, y por lo tanto no son alcanzables desde A. Además, es fácil justificar inductivamente que las cadenas de terminales que se derivan de A_L son exactamente las formas sentenciales que pueden obtenerse por derivación por izquierda en G (aplicándoles l): Dada una derivación por izquierda en G que llega a una cierta forma sentencial, si consideramos su árbol de derivación (parcial) dicha forma sentencial se corresponderá a una hoja del árbol, y los no terminales que quedarán presentes serán todos los hijos de ancestros de dicha hoja, que estén más a la derecha de la rama que contiene la hoja (por ser una derivación por izquierda). De aquí que eligiendo las reglas $A_L \to X_1X_2\cdots X_{i-1}l(X_i)t(X_{i+1})t(X_{i+2})\cdots t(X_k)$ que elijan el i correspondiente a esta rama, obtendremos una derivación en G' de la forma sentencial correspondiente, y viceversa.

3. EL PROBLEMA SAT VISTO A TRAVÉS DE LAS MGIG

3.1. Introducción de SAT

El problema de la satisfacibilidad booleana (SAT) es el problema NP-completo más famoso, por ser el primero demostrado NP completo[16], a partir del cual pueden probarse fácilmente la NP-completitud de muchos otros problemas importantes[33]. En particular, siempre que hablemos de SAT nos referiremos concretamente a la versión SAT-CNF del problema, en la cual se toma como entrada una fórmula booleana proposicional en forma normal conjuntiva (es decir, una conjunción de cláusulas, donde cada cláusula es una disyunción de literales, y cada literal es una variable o una variable negada) y se debe responder si la fórmula dada es satisfacible.

3.2. Definición del lenguaje $\mathcal{SAT}$

En términos de lenguajes formales, a partir del alfabeto dado por

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, \square\}$$

donde $\square$ denota un espacio en blanco utilizado a modo de separador, podemos definir el lenguaje $\mathcal{SAT} \subseteq \Sigma^*$ que contenga todas las cadenas de la forma $s = c_1\square c_2\square c_3\square \cdots \square c_n$, $n \geq 0$ tales que cada cadena c_i tenga *forma de cláusula*, las n cadenas se encuentren ordenadas de manera no decreciente por *menor número de variable*, y además su *fórmula booleana asociada* sea satisfacible.

Una cadena $c \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b\}^+$ tiene forma de cláusula cuando es de la forma $l_1 l_2 l_3 \cdots l_k$, con $k \geq 1$, y las cadenas l_i tienen *forma de literales* con números de variable estrictamente crecientes. Por lo tanto el número de variable de l_1 es el menor número de variable de la cláusula.

Una cadena $l \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b\}^+$ tiene forma de literal con número de variable $i \in \mathbb{N}$ si es $e_i a$ o $e_i b$, donde notamos e_i a la cadena que corresponde a la escritura en decimal del número i. Notemos que dada una cadena con forma de literal, su número de variable está unívocamente determinado, con lo cual las cadenas con forma de cláusula dadas antes están bien definidas.

Definimos el literal asociado a una cadena l con forma de literal como

$$literal(l) = \begin{cases} x_i & \text{si } l = e_i a \\ \neg x_i & \text{si } l = e_i b \end{cases}$$

(es decir, la i-ésima variable afirmada o negada según corresponda). Dada una cadena con forma de cláusula $c = l_1 l_2 l_3 \cdots l_k$, definimos su cláusula asociada como

$$clause(c) = literal(l_1) \vee literal(l_2) \vee \cdots \vee literal(l_k)$$

Dada una cadena $s = c_1\square c_2\square c_3\square \cdots \square c_n$, donde las c_i tienen forma de cláusula, definimos su fórmula booleana asociada como

$$formula(s) = clause(c_1) \wedge clause(c_2) \wedge \cdots \wedge clause(c_n)$$

Con esta definición, el lenguaje $\mathcal{SAT}$ contiene todas las codificaciones de instancias satisfacibles del problema SAT motivador. Toda instancia f de SAT puede ser codificada fácilmente en una cadena $s = \text{codif}(f)$ de manera que f es satisfacible si y solo si $\text{codif}(f) \in \mathcal{SAT}$. Para esto basta con ordenar las cláusulas y los literales dentro de cada cláusula según el número de variable (eliminando eventuales literales repetidos, y cláusulas tautológicas que contengan a la vez un literal y su negación) y luego escribir la fórmula en forma normal conjuntiva con la notación descrita.

Notar sin embargo que la codificación indicada **no es única**: Concretamente, el ordenamiento de las cláusulas entre sí no está completamente determinado, y con esta definición pueden desempatarse cláusulas con un mismo menor número de variable con cualquier criterio. Por ejemplo, `1a2b3a 1a 2b` y `1a 1a2b3a 2b` son dos codificaciones posibles de la fórmula $\neg x_2 \wedge (x_3 \vee x_1 \vee \neg x_2) \wedge x_1$, y ambas pertenecen a $\mathcal{SAT}$ porque esta fórmula es satisfacible con $x_1 = true$ y $x_2 = false$.

Por conveniencia posterior, definimos para cada $n \in \mathbb{Z}_{>0}$ los subconjuntos:

$n\text{-}\mathcal{SAT} = \{s \in \mathcal{SAT} | s$ contiene cláusulas formadas únicamente por variables con números entre 1 y $n\}$

3.3. Reconociendo $\mathcal{SAT}$

Habiendo definido el lenguaje $\mathcal{SAT}$, surge el problema natural de decidir algorítmicamente si una cadena de entrada pertenece al lenguaje. Como hemos mostrado, partiendo de una fórmula booleana, es posible generar una codificación s de forma que la fórmula es satisfacible si y solo si $s \in \mathcal{SAT}$. Por lo tanto, como claramente una codificación con la propiedad mencionada puede computarse en tiempo polinomial en una máquina de Turing determinista (la codificación no es más que escribir en forma de texto cada cláusula ordenándolas correctamente por un criterio simple), el problema SAT se reduce polinomialmente mediante esta codificación al problema de decidir si una cadena pertenece a $\mathcal{SAT}$. En este sentido, el lenguaje $\mathcal{SAT}$ modela esencialmente SAT, ya que no son importantes para dicho problema las restricciones adicionales de ordenamiento de cláusulas que hemos impuesto.

Como SAT es NP-completo, concluimos inmediatamente que el problema de reconocer si una cadena pertenece a $\mathcal{SAT}$ es NP-hard. Nuestro plan a continuación es desarrollar un algoritmo para este problema, basado en técnicas de teoría de compiladores y gramáticas formales.

3.3.1. Análisis léxico de la cadena de entrada

El análisis léxico de la cadena de entrada es inmediato a partir de la definición dada utilizando, como es usual para esta fase del procesamiento, expresiones regulares:

- En primer lugar, los separadores pueden detectarse en forma directa por ser una cadena constante " □ ".
- Notando con d a una expresión regular que detecta los dígitos (formada directamente por el *or* de todos ellos), una cláusula tiene la forma $(d^+(a|b))^+$, siendo cada uno de los elementos $d^+(a|b)$ un literal de la cláusula.

Por lo tanto el análisis léxico de los elementos fundamentales de nuestro lenguaje $\mathcal{SAT}$ no presenta dificultades relevantes, y cualquier motor de expresiones regulares usual

puede utilizarse fácilmente para descomponer la cadena de entrada en las cláusulas (y eventualmente literales) que la constituyen.

3.3.2. Reconocedor de $\mathcal{SAT}$ basado en sintaxis

Una vez que tenemos la entrada separada en tokens correspondientes a las cláusulas, nos gustaría encontrar una gramática que describa adecuadamente las relaciones que son necesarias entre dichos tokens para obtener una cadena en $\mathcal{SAT}$, y de esa manera abordar directamente el problema de reconocer el lenguaje generado por esa gramática utilizando un parser o reconocedor apropiado.

3.3.2.1 Mediante CFG

Como la pertenencia de una cadena al lenguaje generado por una CFG puede reconocerse en tiempo polinomial en el tamaño de la gramática y la cadena (por ejemplo mediante el algoritmo de Earley[18]), intentar reconocer $\mathcal{SAT}$ mediante una gramática de tamaño polinomial en el tamaño de la fórmula de entrada no parece un camino fructífero (pues de tener éxito se obtendría un algoritmo polinomial para reconocer $\mathcal{SAT}$, lo cual implicaría $P = NP$).

Si queremos utilizar una CFG para reconocer $\mathcal{SAT}$, deberíamos utilizar entonces una CFG de tamaño en general exponencial (o al menos super-polinomial) en el tamaño de la fórmula.

3.3.2.2 Mediante GIG

En [10], el autor muestra un algoritmo polinomial de complejidad $O(n^6)$ para el problema de decidir si una cadena de entrada dada pertenece al lenguaje generado por una GIG fijada. Además, la complejidad incorporando la GIG como parámetro de entrada al algoritmo es $O(|I|^3|G|^2n^6)$, siendo $|I|$ y $|G|$ los tamaños del conjunto de índices y de la gramática respectivamente.

En virtud de esto, aplican las mismas consideraciones que en el caso de CFG: Un enfoque prometedor para reconocer $\mathcal{SAT}$ utilizando GIGs requerirá el planteo de una GIG de tamaño mayor que polinomial en el tamaño de la cadena de entrada, para el caso general.

Dicho esto, existe un caso particular de fórmulas CNF satisfacibles que puede ser reconocido mediante GIG, y es el caso de las *Ordered Complete CNF* (OCCNF). Estas fórmulas contienen cláusulas que contienen un literal para cada una de las variables existentes, de manera que una cláusula prohíbe exactamente una valuación. Con GIG es posible procesar las cláusulas ordenadas y "contar" para verificar si todas las posibles cláusulas aparecen (en cuyo caso la fórmula era insatisfacible) o no. Este caso es estudiado en [12].

3.3.2.3 Mediante MGIG

A diferencia de los casos anteriores, será posible mediante una MGIG de tamaño polinomial en la longitud de la cadena de entrada correspondiente, generar un reconocedor de $\mathcal{SAT}$. A continuación mostraremos la construcción de dicha gramática.

La idea de la construcción será modelar un proceso de elección no determinista por el cual vamos eligiendo de izquierda a derecha los valores de verdad de cada variable, y se generan para cada una todas las cláusulas que la contienen como variable de menor número.

Para las cláusulas que no son satisfechas automáticamente por el valor de verdad elegido, se mantienen en la multistack las cláusulas (sufijos de cláusulas originales, que vuelven a ser cláusulas) que todavía falta satisfacer, las cuales se van popeando a medida que se hacen elecciones que las satisfagan. Al final entonces podremos producir una secuencia de cláusulas solo cuando la fórmula sea satisfacible, pues la única forma de vaciar la pila es satisfacer todas las cláusulas pendientes generadas.

Dado un $n \in \mathbb{Z}_{>0}$, consideramos entonces la MGIG que llamaremos *n-TAS* de la siguiente manera:

Terminales: Tomando $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b\}$, el conjunto T de símbolos terminales estará dado por:

$$T = \{s \in \Sigma^+ | s \text{ tiene forma de cláusula en las } n \text{ variables } x_1, x_2, \cdots, x_n\}$$

Notar que T es finito, ya que la cantidad de cláusulas posibles con n variables es $3^n - 1$ (no admitimos cláusulas vacías en nuestra definición de forma de cláusula), al poder cada variable aparecer negada, afirmada, o no aparecer en la cláusula.

No terminales: Tomaremos los siguientes $3n$ símbolos como no terminales:

$$N = \{S_1, S_2, \cdots, S_n, A_1, A_2, \cdots, A_n, B_1, B_2, \cdots, B_n\}$$

Índices: Para los índices utilizaremos una copia (por ejemplo agregando un caracter especial `i` al final para distinguirlos) del mismo conjunto de terminales T, es decir, usaremos cláusulas como índices en las pilas. De acuerdo a nuestra definición conjuntista de las MGIG, los conjuntos I y T deben ser disjuntos, pero utilizaremos la misma notación para ambos pues siempre queda claro en base al uso si estamos hablando de un índice o de un terminal.

Relación de orden: Como relación de orden, compararemos las cláusulas por el menor número de variable que aparece en ella (es decir, el primer número que aparece en la cadena con forma de cláusula que la representa).

En otras palabras $(c_1, c_2) \in O \Leftrightarrow minvar(c_1) \leq minvar(c_2)$

Símbolo inicial: S_1

Reglas de producción:

- Elección de los valores de verdad de las variables
  - $S_i \to A_i S_{i+1} | B_i S_{i+1}$, para $1 \leq i < n$
  - $S_n \to A_n | B_n$
- Cláusulas satisfechas con la menor variable
  - $A_i \to iaw^* A_i$
  - $B_i \to ibw^* B_i$

  - Ambas para $1 \leq i \leq n$, donde w^* denota cualquier posible continuación (potencialmente de longitud cero) del token.

- Cláusulas **no** satisfechas con la menor variable (se inserta el sufijo con las variables aún no fijadas en la multistack de cláusulas pendientes)
  - $A_i \underset{w^+}{\rightarrow} ibw^+A_i$
  - $B_i \underset{w^+}{\rightarrow} iaw^+B_i$
  - Ambas para $1 \leq i \leq n$, donde w^+ denota cualquier posible continuación (de longitud positiva) del token.

- Cláusulas pendientes que se satisfacen con la variable actual (se sacan de la multistack de pendientes)
  - $A_i \underset{\overline{iaw^*}}{\rightarrow} A_i$
  - $B_i \underset{\overline{ibw^*}}{\rightarrow} B_i$
  - Ambas para $1 \leq i \leq n$, con w^* como en las anteriores.

- Cláusulas pendientes que **no** se satisfacen con la variable actual (se deja como pendiente un sufijo sin la variable actual)
  - $A_i \underset{\overline{ibw^+}|w^+}{\rightarrow} A_i$
  - $B_i \underset{\overline{iaw^+}|w^+}{\rightarrow} B_i$
  - Ambas para $1 \leq i \leq n$, con w^+ como en las anteriores.

- Fin del procesamiento de la variable
  - $A_i \rightarrow \lambda$
  - $B_i \rightarrow \lambda$
  - Ambas para $1 \leq i \leq n$.

Llamamos n-$\mathcal{TAS}$ al lenguaje generado por la gramática n-TAS definida. Lo que nos interesa es mostrar que esta gramática expresa el problema SAT, más específicamente:

Teorema 3.3.1: n-$\mathcal{TAS} = n$-$\mathcal{SAT}$

Demostración

- En primer lugar, consideremos una fórmula en n-$\mathcal{SAT}$. Queremos mostrar que existe una derivación exitosa de la fórmula con la gramática n-TAS.

  Llamemos $s \in n$-$\mathcal{SAT}$ a la cadena correspondiente. Por la forma en la cual se encuentran ordenadas las cláusulas (por definición) en las cadenas de n-$\mathcal{SAT}$, podemos escribir:

$$s = s_1 s_2 s_3 \cdots s_n$$

  Donde s_i es una cadena que contiene únicamente cláusulas que comienzan con un literal correspondiente a la variable i (es decir, comienzan con ia o con ib). Por

definición de n-$\mathcal{SAT}$, la fórmula es satisfacible y existe entonces una valuación v que la satisface (De acuerdo al valor correspondiente en la valuación v, notamos $v(i) = A_i$ o $v(i) = B_i$, para cada $1 \leq i \leq n$). Proponemos la siguiente derivación para s:

$$S_1 \Rightarrow v(1)S_2 \underset{\text{der 1}}{\overset{*}{\Rightarrow}} \xi_1 s_1 S_2 \Rightarrow \xi_1 s_1 v(2) S_3 \underset{\text{der 2}}{\overset{*}{\Rightarrow}} \xi_2 s_1 s_2 S_3 \Rightarrow \cdots$$
$$\cdots \underset{\text{der n-1}}{\overset{*}{\Rightarrow}} \xi_{n-1} s_1 s_2 \cdots s_{n-1} S_n \Rightarrow \xi_{n-1} s_1 s_2 \cdots s_{n-1} v(n) \underset{\text{der n}}{\overset{*}{\Rightarrow}} s_1 s_2 \cdots s_{n-1} s_n = s$$

Notar que afirmamos que el multistack final ξ_n será vacío, así como lo es el inicial ξ_0. Faltan definir los tramos de demostración que hemos denotado "der i". Cada uno de estos tramos procesará el no terminal $v(i)$, produciendo la cadena de terminales s_i y modificando el multistack ξ_{i-1} transformándolo en ξ_i. Más precisamente, en el tramo "der i":

1. Se utilizan sucesivamente producciones de tipo pop y pop-push para eliminar de ξ_{i-1} todas las ocurrencias de cláusulas cuya primera variable sea i. Notar que para las pop-push, el sufijo a pushear es no vacío, pues de lo contrario v no sería una valuación que satisface s, en virtud del invariante que se explica a continuación.
2. Luego por cada cláusula c en s_i, se utiliza una producción que permita derivar c:
   - Si c comienza con un literal con valor de verdad correspondiente a $v(i)$, se utiliza directamente una regla libre de contexto sin modificar el multistack, para agregar c a la forma sentencial.
   - Si c comienza con un literal con el valor de verdad contrario a $v(i)$, se utiliza una regla de tipo push, que permite agregar c a la forma sentencial al mismo tiempo que se pushea el sufijo de c sin su primera variable. Notar que dicho sufijo es no vacío, pues sino v no podría satisfacer la fórmula dada por s.

Es un invariante de esta derivación propuesta que en cada paso, el multistack ξ_i contiene exactamente las cláusulas pendientes que deberán satisfacerse con los valores de verdad que se asignen a las variables de índice $j > i$ (además de las cláusulas en s_j para $j > i$), para que dichos valores de verdad formen junto con los valores ya elegidos para las primeras i variables una valuación que satisface s. Esto es porque únicamente se pushean elementos cuando la cláusula que se agrega a la forma sentencial no está satisfecha aún, en cuyo caso se pushea exactamente el sufijo restante de cláusula. Similarmente, únicamente se hace pop de elementos cuando ya están satisfechos, y pop-push de elementos cuando la elección de la variable actual no los satisface, reinsertando en el multistack el sufijo con las restantes variables, alguna de las cuales deberá satisfacer necesariamente la cláusula ya que v es una valuación que satisface s. Además al terminar cada paso únicamente quedan cláusulas con variables $j > i$ en el multistack, o ya habrían sido popeadas.

Como consecuencia de este invariante y del hecho de que v satisface s, resulta ser ξ_n vacío, pues la fórmula ya está satisfecha por v y no quedan variables. Todas las operaciones propuestas preservan este invariante en cada paso, y además construyen una derivación de s de la forma indicada anteriormente. Por lo tanto $s \in n$-$\mathcal{TAS}$

- En segundo lugar, veamos que si una cierta fórmula dada por la cadena s es derivada exitosamente mediante n-TAS, entonces dicha cadena s necesariamente pertenece a n-$\mathcal{SAT}$.

  Para esto basta notar que por la forma de las reglas de producción de la gramática, toda derivación exitosa en n-TAS tiene esencialmente la forma de la derivación que hemos construido en el paso anterior. En particular, es posible identificar una valuación v asociada a una derivación exitosa, dada por $v(i) = A_i$ si se utilizó $S_i \to A_i S_{i+1}$ (o $S_i \to A_i$ para $i = n$), o bien $v(i) = B_i$ en caso contrario (en cuyo caso se habrá utilizado necesariamente la regla con B_i). Además, es posible separar la derivación en n partes según el índice i de variable que se está procesando.

  Por la forma de las reglas de la gramática, en una derivación exitosa cada uno de estos pasos cumplirá el mismo invariante que se explicó en la parte anterior. En efecto, si no se eliminaran todas las ocurrencias de la variable i del multistack en el paso i, la derivación no podrá ser exitosa pues no hay reglas que permitan hacerlo en ningún otro paso posterior. Similarmente, si no se producen todas las cláusulas c de s_i en la forma sentencial en el paso i, no hay reglas que permitan hacerlo en ningún paso posterior así que la derivación no podría ser exitosa. Luego una derivación exitosa debe realizar todos los pasos que hemos explicado en nuestra derivación anterior, y solamente es posible realizar todos ellos cuando la valuación v asociada a la derivación satisface s. Por lo tanto, dada una derivación válida de s hemos construido una derivación v que satisface la fórmula asociada a s, y por lo tanto tenemos $s \in n$-$\mathcal{SAT}$ (que las cláusulas están ordenadas adecuadamente es inmediato por forma de la gramática).

∎

Observación La gramática n-TAS tiene $\Theta(3^n)$ reglas de producción, que resulta exponencial en n tal y como la hemos definido para el caso general, de manera independiente de la cadena de entrada. Esto se debe a que contiene reglas de las formas indicadas para cada una de las $3^n - 1$ cláusulas posibles en n variables. Sin embargo, dada una cadena de entrada s generada a partir de una fórmula booleana de n variables con la codificación anteriormente explicada para el $\mathcal{SAT}$, es claro que las únicas reglas de n-TAS que podrán ser utilizadas en una derivación de s serán aquellas en las que las cláusulas que se mencionan sean **sufijos** de cláusulas en s.

En efecto:

- Si se utiliza una regla correspondiente a cláusula satisfecha con su menor variable, entonces la cláusula mencionada en la regla está en s.
- Si se utiliza una regla correspondiente a cláusula no satisfecha con su menor variable, la cláusula mencionada en el lado derecho de la regla está en s (Y la que se pushea es un sufijo de la cláusula en s).
- Si se utiliza una regla correspondiente a cláusula pendiente que se satisface con la variable actual, la cláusula de la que se hace pop es una cláusula pusheada anteriormente en la derivación, y podemos asumir inductivamente que debe ser un sufijo de una cláusula en s.
- Si se utiliza una regla correspondiente a cláusulas pendientes que no se satisface con la variable actual, la cláusula de la que se hace pop es una cláusula pusheada

anteriormente en la derivación, y podemos asumir inductivamente que debe ser un sufijo de una cláusula en s. La que se pushea es a su vez un sufijo de esta última, y por lo tanto será sufijo de una cláusula en s.

- Las demás reglas no mencionan cláusulas, y son $O(n)$ en total.

De aquí se deduce inmediatamente que si definimos la gramática n-TAS(s) como el subconjunto de la n-TAS que solo contiene las cláusulas que son sufijo de cláusulas en s, con sus respectivas reglas de producción, entonces $s \in n\text{-}\mathcal{TAS} \Leftrightarrow s \in n\text{-}\mathcal{TAS}(s)$, pues la segunda gramática contiene exactamente las reglas de n-TAS que podrían ser utilizadas en una derivación exitosa de s.

Notar que si $l = |s|$, la cantidad de sufijos mencionados es $O(l)$, y el tamaño de cada regla de producción es $O(1)$ símbolos [1], con lo cual la gramática n-TAS(s) tiene un tamaño $O(l + n)$, que es $O(l)$ si las n variables aparecen en s, lo cual puede asumirse sin cambiar la fórmula booleana representada con un simple renombre de las variables y tomando n del menor tamaño posible.

Corolario 3.3.2: El problema de reconocer si una cadena de entrada dada pertenece al lenguaje generado por una MGIG dada como entrada es NP-hard. Esto es evidente ya que por la observación anterior y el teorema de equivalencia entre $n\text{-}\mathcal{TAS}$ y $n\text{-}\mathcal{SAT}$, es posible reducir el problema SAT para una fórmula de tamaño l al de identificar si una cadena de longitud $O(l)$ pertenece a una MGIG de tamaño polinomial en l [2], y esta reducción es claramente polinomial pues consiste en escribir la codificación s de la fórmula, y las reglas de la gramática que ya hemos explicado y que se derivan de manera directa de la fórmula original.

En virtud de esto, como ya hemos probado que REC-MGIG $\in$ NP, concluimos que REC-MGIG $\in$ NP-completo.

3.4. Otras gramáticas para resolver SAT

En este trabajo hemos utilizado otras MGIG además de *n-TAS*, que si bien producen lenguajes levemente diferentes, igualmente sirven para resolver el problema SAT. Como estas resultan de modificar las reglas de producción de *n-TAS*, que es la lógicamente más simple, hemos comenzado introduciendo las ideas y resultados principales con *n-TAS*.

3.4.1. $n\text{-}\mathcal{DTAS}$

Definimos el lenguaje $n\text{-}\mathcal{DSAT}$, "deterministic-SAT", de manera análoga al $n\text{-}\mathcal{SAT}$, pero con la restricción adicional de que las cláusulas con un mismo menor número de variable se ordenan por valor de verdad correspondiente. Más precisamente, para la variable i-ésima, las cláusulas que comienzan ia se ubican siempre antes que las que comienzan ib.

Esto permite introducir mayor determinismo en la gramática correspondiente, separando el procesamiento de cada variable en tres fases ordenadas, cada una con su propio símbolo no terminal:

[1] Si cada sufijo es considerado un símbolo

[2] Ver la codificación explicada en 2.3.8. Utilizando esa codificación, tenemos $O(l)$ números de $O(\lg l)$ bits de longitud.

- En la primera fase, con símbolo no terminal A_i^R / B_i^R según el valor de verdad elegido, se hace pop / pop-push de la multistack de todas las cláusulas pendientes que contengan la variable actual (y que por lo tanto se encontrarán al comienzo de la multistack).
- En la segunda fase, con símbolo no terminal A_i / B_i^P, se generan las cláusulas que comienzan con ia (que por la restricción introducida en el ordenamiento, vienen todas juntas antes de las ib). En el caso de elegir el valor de verdad a para la i-ésima variable, simplemente se generarán las cláusulas mediante las reglas libres de contexto correspondientes a A_i. Si se eligió el valor de verdad b, se hace push de los sufijos que quedan pendientes de satisfacer, de acuerdo a las reglas correspondientes al no terminal B_i^P.
- En la tercera fase, con símbolo no terminal A_i^P / B_i, se generan las cláusulas que comienzan con ib. Esto es análogo a la segunda fase con los roles de a y b intercambiados.

Definimos entonces la correspondiente gramática n-$DTAS$ de manera análoga a n-TAS pero separando las reglas gramaticales de acuerdo a la fase a la cual corresponden. Tenemos entonces las siguientes reglas de producción para n-$DTAS$:

- Elección de los valores de verdad de las variables
  - $S_i \to A_i^R S_{i+1} | B_i^R S_{i+1}$, para $1 \leq i < n$
  - $S_n \to A_n^R | B_n^R$
- Primera fase: Cláusulas pendientes que se satisfacen con la variable actual (se sacan de la multistack de pendientes)
  - $A_i^R \underset{\overline{iaw^*}}{\to} A_i^R$
  - $B_i^R \underset{\overline{ibw^*}}{\to} B_i^R$
  - Ambas para $1 \leq i \leq n$
- Primera fase: Cláusulas pendientes que **no** se satisfacen con la variable actual (se deja como pendiente un sufijo sin la variable actual)
  - $A_i^R \underset{\overline{ibw^+|w^+}}{\to} A_i^R$
  - $B_i^R \underset{\overline{iaw^+|w^+}}{\to} B_i^R$
  - Ambas para $1 \leq i \leq n$
- Segunda / Tercera fase: Cláusulas satisfechas con la menor variable
  - $A_i \to iaw^* A_i$
  - $B_i \to ibw^* B_i$
  - Ambas para $1 \leq i \leq n$
- Segunda / Tercera fase: Cláusulas **no** satisfechas con la menor variable (se inserta el sufijo con las variables aún no fijadas en la multistack de cláusulas pendientes)

- $A_i^P \underset{w^+}{\rightarrow} ibw^+A_i^P$
- $B_i^P \underset{w^+}{\rightarrow} iaw^+B_i^P$
- Ambas para $1 \leq i \leq n$

- Secuencia de procesamiento de la variable
  - $A_i^R \rightarrow A_i$
  - $A_i \rightarrow A_i^P$
  - $A_i^P \rightarrow \lambda$
  - $B_i^R \rightarrow B_i^P$
  - $B_i^P \rightarrow B_i$
  - $B_i \rightarrow \lambda$
  - Todas las anteriores con $1 \leq i < n$.
  - $A_n^R \rightarrow A_n$
  - $A_n \rightarrow \lambda$
  - $B_n^R \rightarrow B_n$
  - $B_n \rightarrow \lambda$

De manera análoga a lo que ocurre con *n-TAS*, resulta $n\text{-}\mathcal{DTAS} = n\text{-}\mathcal{DSAT}$

3.4.2. n-$\mathcal{RTAS}$

Definimos el lenguaje n-$\mathcal{RSAT}$, "restricted-SAT", de manera análoga al n-$\mathcal{DSAT}$, pero con la restricción adicional de que para todo número de variable salvo el n, debe existir al menos una cláusula que comience con ib.

Esto permite evitar casi todas las producciones lambda de la correspondiente gramática *n-RTAS*, que queda definida por las siguientes reglas de producción:

- Elección de los valores de verdad de las variables
  - $S_i \rightarrow A_i^R S_{i+1} | B_i^R S_{i+1}$, para $1 \leq i < n$
  - $S_n \rightarrow A_n^R | B_n^R$

- Primera fase: Cláusulas pendientes que se satisfacen con la variable actual (se sacan de la multistack de pendientes)
  - $A_i^R \underset{\overline{iaw^*}}{\rightarrow} A_i^R$
  - $B_i^R \underset{\overline{ibw^*}}{\rightarrow} B_i^R$
  - Ambas para $1 \leq i \leq n$

- Primera fase: Cláusulas pendientes que **no** se satisfacen con la variable actual (se deja como pendiente un sufijo sin la variable actual)
  - $A_i^R \underset{\overline{ibw^+|w^+}}{\rightarrow} A_i^R$

- $B_i^R \underset{\overline{iaw^+}|w^+}{\rightarrow} B_i^R$
- Ambas para $1 \leq i \leq n$

- Segunda / Tercera fase: Cláusulas satisfechas con la menor variable
  - $A_i \rightarrow iaw^* A_i$
  - $B_i \rightarrow ibw^* B_i$
  - $B_i \rightarrow ibw^*$
  - Las tres para $1 \leq i \leq n$
  - $A_n \rightarrow \lambda$
  - $B_n \rightarrow \lambda$

- Segunda / Tercera fase: Cláusulas **no** satisfechas con la menor variable (se inserta el sufijo con las variables aún no fijadas en la multistack de cláusulas pendientes)
  - $A_i^P \underset{w^+}{\rightarrow} ibw^+ A_i^P$
  - $A_i^P \underset{w^+}{\rightarrow} ibw^+$
  - $B_i^P \underset{w^+}{\rightarrow} iaw^+ B_i^P$
  - Las tres para $1 \leq i \leq n$

- Secuencia de procesamiento de la variable
  - $A_i^R \rightarrow A_i$
  - $A_i \rightarrow A_i^P$
  - $B_i^R \rightarrow B_i^P$
  - $B_i^P \rightarrow B_i$
  - Todas las anteriores con $1 \leq i < n$.
  - $A_n^R \rightarrow A_n$
  - $B_n^R \rightarrow B_n$

De manera análoga a lo que ocurre en los casos anteriores, resulta $n\text{-}\mathcal{RTAS} = n\text{-}\mathcal{RSAT}$

3.4.2.1 Sobre la generalidad de la restricción

En principio podría parecer que esta restricción limita las fórmulas (es decir, las instancias de SAT) a las cuales puede aplicarse el algoritmo reconocedor. Sin embargo, es fácil a partir de una fórmula arbitraria dada como entrada, generar una fórmula lógicamente equivalente que satisfaga las relaciones adicionales impuestas por esta gramática, que a diferencia de las presentadas anteriormente va más allá de un simple reordenamiento de cláusulas.

Concretamente, podemos transformar una fórmula de n variables en una equivalente de $n+1$ variables, que se obtenga de la original mediante el agregado de cláusulas que contengan la nueva variable. La idea será utilizar esta nueva variable para agregar cláusulas que comiencen con ib, pero se satisfagan de manera automática al contener la nueva

variable, cuyo valor de verdad estará fijado a $(n+1)b$, y de esta manera logramos que exista alguna cláusula que comienza ib para todo $1 \leq i \leq n+1$.

Concretamente, agregamos al final una cláusula de un único literal $(n+1)b$, y por cada número de variable i tal que la fórmula original no contiene ninguna cláusula que comience ib, agregamos la cláusula $ib(n+1)b$. Está claro que si esta fórmula es satisfacible, la original lo era con la misma asignación, ya que esta fórmula se obtiene agregando cláusulas a la anterior y es por lo tanto lógicamente más fuerte. Recíprocamente, si una asignación de las n variables originales hace verdadera a la fórmula original, con la misma asignación se satisfacen todas las cláusulas de la nueva fórmula, salvo quizás las que hemos agregado, pero entonces fijando el valor de la variable $n+1$ a b (que es posible sin afectar a las n originales) se satisfacen automáticamente todas las cláusulas restantes.

3.4.3. n-$\mathcal{SRTAS}$

Definimos el lenguaje n-$\mathcal{SRSAT}$, "super-restricted-SAT", como una pequeña restricción adicional del n-$\mathcal{RSAT}$: se exige que también exista una cláusula que comience ib para $i = n$, es decir, para la última variable, para la cual no se exigía esta restricción en el caso anterior.

Esta forma más estricta nos permite eliminar todas las producciones lambda de la gramática. La separación entre n-$\mathcal{SRSAT}$ y n-$\mathcal{RSAT}$ la mantenemos fundamentalmente porque comenzamos trabajando inicialmente con n-$\mathcal{RSAT}$, aunque posteriormente se refinó la gramática a esta forma más estricta y homogénea.

Al igual que ocurre con n-*RTAS*, no se pierde generalidad en esta versión, ya que se puede utilizar exactamente la misma transformación explicada anteriormente para llevar cualquier instancia de SAT a una equivalente en la forma adecuada para esta gramática. Las reglas de producción de la gramática son las siguientes:

- Elección de los valores de verdad de las variables
  - $S_i \to A_i^R S_{i+1} | B_i^R S_{i+1}$, para $1 \leq i < n$
  - $S_n \to B_n^R$
- Primera fase: Cláusulas pendientes que se satisfacen con la variable actual (se sacan de la multistack de pendientes)
  - $A_i^R \xrightarrow[iaw^*]{} A_i^R$, para $1 \leq i < n$
  - $B_i^R \xrightarrow[ibw^*]{} B_i^R$, para $1 \leq i \leq n$
- Primera fase: Cláusulas pendientes que **no** se satisfacen con la variable actual (se deja como pendiente un sufijo sin la variable actual)
  - $A_i^R \xrightarrow[ibw^+ | w^+]{} A_i^R$
  - $B_i^R \xrightarrow[iaw^+ | w^+]{} B_i^R$
  - Ambas para $1 \leq i < n$
- Segunda / Tercera fase: Cláusulas satisfechas con la menor variable

- $A_i \to iaw^*A_i$, para $1 \leq i < n$
- $B_i \to ibw^*B_i$, para $1 \leq i \leq n$
- $B_i \to ibw^*$, para $1 \leq i \leq n$

- Segunda / Tercera fase: Cláusulas **no** satisfechas con la menor variable (se inserta el sufijo con las variables aún no fijadas en la multistack de cláusulas pendientes)
  - $A_i^P \underset{w^+}{\to} ibw^+A_i^P$
  - $A_i^P \underset{w^+}{\to} ibw^+$
  - $B_i^P \underset{w^+}{\to} iaw^+B_i^P$
  - Las tres para $1 \leq i < n$

- Secuencia de procesamiento de la variable
  - $A_i^R \to A_i$
  - $A_i \to A_i^P$
  - $B_i^R \to B_i^P$
  - $B_i^P \to B_i$
  - Todas las anteriores con $1 \leq i < n$.
  - $B_n^R \to B_n$

De manera análoga a lo que ocurre en los casos anteriores, resulta n-$\mathcal{SRTAS}$ = n-$\mathcal{SRSAT}$

3.4.4. Variante: separación de la elección del valor de verdad de las variables a no terminales V_i

Para cada una de las anteriores, es posible dar una variación en la forma de generar la secuencia de no terminales S_i, que luego darán lugar individualmente a las elecciones de los valores de verdad de cada variable.

En lugar de, en una misma regla, elegir el valor de verdad para la variable i y pasar al siguiente no terminal S_{i+1}, se puede separar estas dos partes introduciendo nuevos no terminales V_i, que se corresponden con la parte de la cadena de entrada que tiene cláusulas que comienzan en i, y que en nuestras derivaciones "generamos" inmediatamente a continuación de elegir el valor de verdad de la variable i.

Concretamente, en lugar de tener las reglas de la siguiente forma (en varias de las gramáticas anteriores se usa $A_i^R|B_i^R$ en lugar de $A_i|B_i$, pero la forma de las reglas aquí dadas será análoga):

- $S_i \to A_iS_{i+1}|B_iS_{i+1}$, para $1 \leq i < n$
- $S_n \to A_n|B_n$

Tendremos las siguientes reglas de generación de los V_i a partir del símbolo inicial S_1, y n reglas análogas entre sí para elegir el valor de verdad de cada variable, correspondiente a cada V_i:

- $S_i \to V_i S_{i+1}$ para $1 \leq i < n$
- $S_n \to V_n$
- $S_i \to A_i | B_i$, para $1 \leq i \leq n$

Para referirnos a la versión correspondiente a alguna de las gramáticas anteriores con esta modificación, lo indicaremos específicamente, o bien utilizaremos la letra V como prefijo (en alusión a los nuevos no terminales), por ejemplo V-n-*RTAS* o V-n-*TAS*.

Desde el comienzo del trabajo, trabajamos con las gramáticas anteriores, sin esta modificación, pero como explicaremos más adelante esta variante de la gramática resulta ser fuertemente determinada a derecha 5.1.2, y esta forma tiene algunas ventajas teóricas como se menciona en 5.2.

3.4.5. Variante: de MGIG con forma regular

Ya hemos mencionado anteriormente que en [13], el autor trabaja con una noción de MGIG más restringida, donde **todas** las reglas de producción deben tener forma regular, y no solo aquellas que indiquen operaciones del multistack.

Si bien no hemos explorado su utilización en este trabajo, y en particular ninguna de las anteriores gramáticas que hemos utilizado tiene esta forma, mostramos en esta sección una posible gramática para reconocer n-$\mathcal{SAT}$ que tiene la forma regular más restringida considerada en [13]. Esto muestra que el problema general de reconocimiento de MGIG sigue siendo NP-completo incluso con dicha restricción.

La gramática es sencilla, y esencialmente consiste en "mezclar" todos los pasos de las gramáticas anteriores, sin separar entre la generación de los A_i y B_i, y su procesamiento.

Concretamente tendremos las siguientes reglas de producción (ahora utilizaremos un único símbolo inicial S):

- **Elección de los valores de verdad de las variables** (únicas reglas nuevas diferentes a n-TAS)
  - $S \to A_1 | B_1$
  - $A_i \to A_{i+1} | B_{i+1}$, para $1 \leq i < n$
  - $B_i \to A_{i+1} | B_{i+1}$, para $1 \leq i < n$
  - $A_n \to \lambda$
  - $B_n \to \lambda$
- Cláusulas satisfechas con la menor variable
  - $A_i \to i a w^* A_i$
  - $B_i \to i b w^* B_i$
  - Ambas para $1 \leq i \leq n$, donde w^* denota cualquier posible continuación (potencialmente de longitud cero) del token.
- Cláusulas **no** satisfechas con la menor variable (se inserta el sufijo con las variables aún no fijadas en la multistack de cláusulas pendientes)
  - $A_i \underset{w^+}{\to} i b w^+ A_i$

- $B_i \underset{w^+}{\rightarrow} iaw^+B_i$
- Ambas para $1 \leq i \leq n$, donde w^+ denota cualquier posible continuación (de longitud positiva) del token.

- Cláusulas pendientes que se satisfacen con la variable actual (se sacan de la multistack de pendientes)
  - $A_i \underset{\overline{iaw^*}}{\rightarrow} A_i$
  - $B_i \underset{\overline{ibw^*}}{\rightarrow} B_i$
  - Ambas para $1 \leq i \leq n$, con w^* como en las anteriores.
- Cláusulas pendientes que **no** se satisfacen con la variable actual (se deja como pendiente un sufijo sin la variable actual)
  - $A_i \underset{\overline{ibw^+}|w^+}{\rightarrow} A_i$
  - $B_i \underset{\overline{iaw^+}|w^+}{\rightarrow} B_i$
  - Ambas para $1 \leq i \leq n$, con w^+ como en las anteriores.

4. ALGORITMO DE RECONOCIMIENTO DE MGIG

Para poder completar la construcción de un SAT-solver basado en las ideas expuestas anteriormente de utilizar un reconocedor del lenguaje generado por una MGIG, falta dar un algoritmo de reconocimiento de MGIG. En este capítulo daremos un algoritmo general para reconocimiento y parsing de una MGIG arbitraria dada como entrada. En el capítulo siguiente estudiaremos más en detalle el caso particular de SAT, y cómo para dicho caso (y para otros casos de MGIG que cumplan las hipótesis necesarias) se puede simplificar el algoritmo general que aquí proponemos.

4.1. Reconocedor general de MGIG

El algoritmo que damos a continuación se basa fuertemente en el algoritmo de Earley[18] para reconocimiento de CFG, y constituye una extensión natural de las ideas presentadas por el autor de dicho algoritmo cuando se trabaja con MGIG en lugar de CFG.

4.1.1. Descripción del algoritmo de Earley

Para procesar una cadena de entrada de longitud n, el algoritmo de Earley construye $n+1$ conjuntos $S_0, S_1, \cdots, S_n$ de *ítems*. Un ítem tiene la forma $(i, A \to X \bullet Y)$ con $0 \leq i \leq n$ y $A \to XY$ una regla de producción de la gramática, siendo A un no terminal y X e Y cadenas posiblemente vacías de terminales y no terminales.

Si llamamos $\alpha = \alpha_0\alpha_1 \cdots \alpha_{n-1}$ a la cadena de entrada, siendo α_i cada uno de los tokens individuales, la

- ***Propiedad fundamental*** del algoritmo de Earley es que un ítem $(i, A \to X \bullet Y)$ pertenece al conjunto S_j al terminar el algoritmo, si y solo si existe una derivación por izquierda de la forma:

  $$S' \overset{*}{\Rightarrow} \alpha_0\alpha_1 \cdots \alpha_{i-1}A\gamma \Rightarrow \alpha_0\alpha_1 \cdots \alpha_{i-1}XY\gamma \overset{*}{\Rightarrow} \alpha_0\alpha_1 \cdots \alpha_{i-1}\alpha_i\alpha_{i+1} \cdots \alpha_{j-1}Y\gamma$$

  Donde se entiende que en el árbol de esta derivación, el paso correspondiente a la regla $A \to \cdots$ cuya aplicación se muestra es el padre de todos los nodos correspondientes al Y de la última forma sentencial.

Es decir, es posible "comenzar a parsear" una regla de producción de la forma $A \to XY$ en la posición i de la cadena de entrada, y "habiendo leído hasta la posición j", encontrarse en el punto indicado por $\bullet$ del lado derecho de la producción, "faltando parsear Y".

De esta forma, el algoritmo puede considerarse un algoritmo de programación dinámica, ya que la composición de cada conjunto S_i se obtiene a partir de los S_j con $j < i$ previamente calculados, y además se presenta la característica superposición de subproblemas ya que un mismo ítem en un mismo conjunto puede ser agregado a través de varias posibles derivaciones, evitándose al trabajar con los conjuntos de ítems repetir el cómputo de un mismo ítem múltiples veces.

El algoritmo comienza inicialmente con $S_0 = \{(0, S' \to \bullet S)\}$, siendo S' un símbolo *nuevo*, y S el símbolo inicial de la gramática utilizada. Se reconocerá exitosamente la

cadena de entrada si finalmente obtenemos que $(0, S' \to S\bullet) \in S_n$. Esto es consecuencia inmediata de la propiedad que acabamos de enunciar. Alternativamente, es posible no agregar un símbolo nuevo, y colocar todas las reglas de producción que contengan a S del lado izquierdo en el conjunto S_0 al comienzo. Similarmente, en este caso habría que verificar la presencia de cualquier ítem de la forma $(0, S \to \gamma\bullet)$ en el conjunto S_n. Utilizaremos siempre por simplicidad el criterio de agregar un símbolo nuevo S' con un único ítem inicial $(0, S' \to \bullet S)$ en S_0, y un único ítem final $(0, S' \to S\bullet)$ en S_n.

Para construir los mencionados conjuntos, el algoritmo de Earley procesa cada uno de los ítems exactamente una vez (un mismo ítem que aparece en varios conjuntos es procesado exactamente una vez por conjunto). El procesamiento de un ítem puede agregar nuevos ítems a los conjuntos, que serán procesados a su vez posteriormente. Inicialmente existe un único ítem en S_0, y el proceso terminará cuando ya no quede ningún ítem sin procesar.

El procesamiento de un ítem puede clasificarse en uno de tres posibles casos, dependiendo de la forma del ítem (más específicamente, dependiendo del símbolo presente en la regla de producción inmediatamente a la derecha del $\bullet$):

Scanner: Dado un ítem $(i, A \to X \bullet tY) \in S_j$, con i, A, X, Y como antes y $t \in T$ un símbolo terminal, si $t = \alpha_j$ se agrega $(i, A \to Xt \bullet Y)$ a S_{j+1}.

Predictor: Dado un ítem $(i, A \to X \bullet BY) \in S_j$, con i, A, X, Y como antes y $B \in N$ un símbolo no terminal, se agrega al conjunto S_j un ítem $(j, B \to \bullet\gamma)$ por cada regla $B \to \gamma$ de la gramática.

Completer: Dado un ítem $(i, A \to \gamma\bullet) \in S_j$, se agrega al conjunto S_j un ítem $(k, B \to XA \bullet Y)$ por cada ítem $(k, B \to X \bullet AY)$ presente en el conjunto S_i.

Es fácil verificar la correctitud del algoritmo de Earley si se tiene presente la propiedad fundamental anteriormente enunciada sobre la existencia de una derivación con ciertas características asociada a cada ítem de los conjuntos. En efecto, el único ítem inicial claramente satisface la propiedad, y los ítems que se agregan en cada una de las operaciones cumplen la propiedad, si se asume que los ítems ya existentes la cumplían. De esta forma es claro inductivamente que por cada ítem presente en un conjunto de Earley, existe una derivación con la forma indicada por la propiedad fundamental.

Similarmente, dada una derivación y un ítem que debería estar presente en un conjunto de acuerdo a la propiedad, es posible verificar que alguna de las tres operaciones agregará necesariamente este ítem (asumiendo inductivamente que lo mismo ocurre para todos los prefijos de la derivación en cuestión). La operación en cuestión que agregará este ítem dependerá de lo que haya **a la izquierda** del $\bullet$ en el ítem: Si es un terminal, será agregado vía Scanner, si es un no terminal, vía Completer, y de no haber nada, vía Predictor.

4.1.2. Modificación para reconocer MGIG en lugar de CFG

Teniendo bien presente la idea presentada en la propiedad fundamental de la sección anterior, es natural extender el algoritmo de Earley para el reconocimiento de MGIG agregando información sobre multistack en los ítems. Llamaremos Earley-MGIG al algoritmo extendido.

Concretamente, consideraremos los conjuntos $S_0 \cdots S_n$ de ítems de la forma $(i, \xi_i, \xi, A \to X \bullet Y)$, donde ξ_i y ξ son multistacks. Modificamos la propiedad fundamental de la siguiente manera:

- ***Propiedad fundamental*** del algoritmo de Earley-MGIG:

  Un ítem $(i, \xi_i, \xi, A \underset{\mu}{\to} X \bullet Y)$ pertenece al conjunto S_j al terminar el algoritmo, si y solo si existe una derivación por izquierda de la forma:

  $$S' \overset{*}{\Rightarrow} \xi_i \alpha_0 \alpha_1 \cdots \alpha_{i-1} A \gamma \Rightarrow \mu(\xi_i) \alpha_0 \alpha_1 \cdots \alpha_{i-1} X Y \gamma \overset{*}{\Rightarrow} \xi \alpha_0 \alpha_1 \cdots \alpha_{i-1} \alpha_i \alpha_{i+1} \cdots \alpha_{j-1} Y \gamma$$

  Donde hemos notado $\mu(\xi)$ a la multistack que resulta de aplicar a ξ la operación de multistack indicada por μ (potencialmente ninguna)

  Nuevamente, se entiende que en el árbol de esta derivación el paso correspondiente a la regla $A \to \cdots$ cuya aplicación se muestra es el padre de todos los nodos correspondientes al Y de la última forma sentencial.

La intuición es análoga al caso de Earley para CFG: la presencia de $(i, \xi_i, \xi, A \underset{\mu}{\to} X \bullet Y)$ en el conjunto S_j indica que es posible "comenzar a parsear" una regla de producción de la forma $A \underset{\mu}{\to} XY$ en la posición i de la cadena de entrada, teniendo el multistack ξ_i justo antes de comenzar el parsing de la regla mencionada, y "habiendo leído hasta la posición j", encontrarse en el punto indicado por $\bullet$ del lado derecho de la producción, "faltando parsear Y" y siendo el multistack actual ξ.

Las operaciones de scanner y completer no presentan cambios relevantes con respecto al caso anterior, salvo el hecho de que "arrastran" la información de multistacks presente en los ítems. En el predictor además es necesario realizar la actualización (update) del multistack, que hemos notado con $\mu(\xi)$ para indicar que se hace update de ξ de acuerdo a lo indicado por el μ de la regla de producción.

Scanner: Dado un ítem $(i, \xi_i, \xi, A \to X \bullet tY) \in S_j$, con i, A, X, Y como antes y $t \in T$ un símbolo terminal, si $t = \alpha_j$ se agrega $(i, \xi_i, \xi, A \to Xt \bullet Y)$ al conjunto S_{j+1}.

Predictor: Dado un ítem $(i, \xi_i, \xi, A \to X \bullet BY) \in S_j$, con i, A, X, Y como antes y $B \in N$ un símbolo no terminal, se agrega al conjunto S_j un ítem $(j, \xi, \mu(\xi), B \to \bullet\gamma)$ por cada regla $B \underset{\mu}{\to} \gamma$ de la MGIG. Esto se hace si es posible llevar a cabo la operación indicada por μ (es decir, en el caso de involucrar un pop, si el índice que se está popeando se encuentra presente como primer tope en ξ).

Completer: Dado un ítem $(i, \xi_i, \xi, A \to \gamma\bullet) \in S_j$, se agrega al conjunto S_j un ítem $(k, \xi'_i, \xi, B \to XA \bullet Y)$ por cada ítem $(k, \xi'_i, \xi_i, B \to X \bullet AY)$ presente en el conjunto S_i.

Notando λ a la multistack vacía, en este caso se comienza con $S_0 = \{(0, \lambda, \lambda, S' \to \bullet S)\}$, y se reconoce la cadena de entrada cuando al terminar la ejecución $(0, \lambda, \lambda, S' \to S\bullet) \in S_n$. Esto es correcto por la propiedad fundamental.

La demostración de la validez de la propiedad fundamental en este caso es completamente análoga al caso de Earley para CFG, ya que nuevamente cada paso preserva inductivamente su validez.

4.1.3. Implementación: update de multistack en scanner vs predictor

En la sección anterior se presentó el algoritmo de Earley-MGIG, de la manera lógicamente más simple posible. En particular, el **predictor** es el paso de Earley en el cual **se elige** la siguiente regla de producción a utilizar, y por lo tanto se corresponde con el momento en el que lógicamente se produce la modificación del multistack en la derivación correspondiente.

A la hora de utilizar el algoritmo de Earley para reconocimiento de CFG, es una práctica habitual complementar el algoritmo de Earley que trabaja directamente sobre la gramática con un lexer o analizador léxico que separe la cadena de entrada en tokens: de esta forma, los **terminales** de la CFG (o en nuestro caso, MGIG) se corresponden con **conjuntos posibles de tokens**, asociables a dicho terminal. Es habitual a su vez que el conjunto de tokens correspondientes a un terminal de la gramática venga dado como una expresión regular asociada a cada terminal, de manera que en la implementación en lugar de verificar la igualdad $\alpha_j = t$ en el paso scanner, se verifica que el token α_j pertenezca al lenguaje generado por la correspondiente expresión regular asociada a t. Este es el enfoque que hemos tenido en nuestra implementación.

Teniendo en cuenta esta consideración implementativa habitual en CFG, resulta natural extender estas consideraciones para con los μ de las reglas de producción de MGIG. Más precisamente, en la descripción del algoritmo que hemos realizado, hemos mantenido por simplicidad la asunción de que en reglas como $A \underset{\mu}{\to} \omega A$, tanto el $\omega \in T$ como el $\mu \in I$ son de la forma dada en nuestra descripción formal de MGIG (problema que ya hemos probado que es en sí mismo NP-completo y alcanza para modelar SAT). Al proceder como es habitual con CFG y trabajar con ω como una expresión regular que especifica los tokens aceptables, resulta natural en el caso de MGIG considerar que el μ correspondiente de la regla venga dado también como una expresión regular, que se debe matchear contra el correspondiente ω encontrado en el texto de entrada. En otras palabras, es natural que el μ dependa del ω, y eso nos permite condensar implementativamente en una sola regla de la gramática muchas reglas similares. Ya hemos utilizado esta convención a modo de abreviación previamente, cuando al describir gramáticas para SAT hemos hablado de reglas como $A_i \underset{w^+}{\to} ibw^+A_i$ o $A_i \underset{\overline{iaw^*}}{\to} A_i$, entendiendo que con esto nos referíamos a un conjunto de reglas de esa forma. Con estas consideraciones implementativas, podemos resumir efectivamente las reglas de esta forma en el algoritmo, y por ejemplo considerar un pop como válido cuando el tope se corresponde con la expresión regular indicada en la regla (comparado al chequeo de igualdad mencionado en el algoritmo "directo"), o hacer push de un sufijo **que dependerá del símbolo terminal en el lado derecho de la regla**.

Esta última consideración nos lleva a modificar la implementación directa propuesta anteriormente, que actualiza los multistacks únicamente en la operación predictor, de manera que las multistacks en el caso de reglas push, pop o pop-push **que contengan un terminal en el lado derecho**, se actualicen en el paso **scanner**, ya que es en dicho paso en el que se lee el siguiente símbolo de la entrada y se lo matchea contra la correspondiente expresión regular indicada en la regla de producción.

Notar que esto rompe la propiedad fundamental anteriormente mencionada, pero no se afecta a la correctitud del algoritmo ya que la propiedad se restituye inmediatamente luego del paso Scanner. Más precisamente, dada una regla $A \underset{\mu}{\to} \omega A$ de tipo push, pop o pop-push, cuando el predictor la seleccione se generará un ítem de Earley de la forma

$A \underset{\mu}{\rightarrow} \bullet\omega A$ en un cierto conjunto S_j, pero sin alterar la multistack presente en el ítem desde el cuál fue generado. No obstante como el símbolo a continuación del $\bullet$ es un terminal, este ítem solamente será utilizado por el algoritmo en el paso de scanner, que por ser una regla de este tipo modificará la multistack de acuerdo a la operación correspondiente y en base al ω concreto que se lea de la cadena de entrada. De esta forma el único ítem que queda con la multistack "atrasada" con respecto a lo que se espera de acuerdo a la propiedad fundamental es el que tiene la forma $A \underset{\mu}{\rightarrow} \bullet\omega A$ generado por el predictor.

4.1.4. Sobre la necesidad de almacenar información sobre multistack de partida en los ítems

Una optimización deseable al algoritmo propuesto consiste en la eliminación de los multistack iniciales en los ítems de Earley. Es decir, el tamaño de los conjuntos y consecuentemente la cantidad total de ítems distintos explorados durante el algoritmo, así como la memoria utilizada y complejidad de la implementación, podrían reducirse en principio si únicamente se guardara la segunda multistack de los ítems. Lo que se pierde al no guardar tal información en los ítems es la posibilidad de hacer la verificación durante el paso completer de que la "multistack actual" del ítem que se va a completar coincida con la "multistack inicial" del ítem que se usa para completarlo, debiendo asumir que dicha verificación resulta siempre exitosa. Veremos más adelante que esta optimización es posible para ciertas gramáticas, y la utilizaremos en concreto a la hora de aplicar el algoritmo al caso de SAT.

En el caso general de reconocimiento del lenguaje de una MGIG arbitraria, no es posible prescindir de tener dos multistacks, como se puede observar al analizar el siguiente ejemplo:

$$\begin{array}{lcl} S & \rightarrow & AXa \\ S & \rightarrow & BXb \\ A & \underset{a}{\rightarrow} & x \\ B & \underset{b}{\rightarrow} & x \\ X & \underset{\bar{b}}{\rightarrow} & xX \\ X & \rightarrow & x \end{array}$$

Como no hay ninguna regla que permita hacer pop de un símbolo a, una derivación exitosa no puede utilizar la tercera regla. Como dicha regla es la única para el no terminal A, tampoco será posible utilizar la primera regla, que contiene un A del lado derecho. Como la primera regla es la única que contiene al terminal a del lado derecho, ninguna cadena que contenga un a pertenece al lenguaje generado por esta MGIG. Sin embargo, el algoritmo de reconocimiento daría por válida la cadena $xxxa$ si solamente se utilizara una multistack en los ítems. Esto se debe a que como consecuencia de la siguiente derivación correcta:

$$S \Rightarrow BXb \Rightarrow b\#xXb \Rightarrow xxXb \Rightarrow xxxb$$

Al completarse el símbolo X en el último paso, se completarán durante el algoritmo los ítems de S_1 con producciones $S \rightarrow A \bullet Xa$ y $S \rightarrow B \bullet Xb$, y como la multistack actual es vacía, el ítem $S \rightarrow AX \bullet a$ generado en el primer caso permite completar incorrectamente

una derivación "exitosa" de $xxxa$. El problema radica justamente en que el no terminal X puede completarse con multistack vacía solamente si se comenzó su procesamiento con una multistack con b, a diferencia de una multistack con a, con lo cual es necesario mantener esta información en los ítems para resolver correctamente estos casos con el algoritmo propuesto.

La dificultad esencial que presenta este ejemplo radica en que hay dos reglas distintas que contienen X del lado derecho, con **continuaciones diferentes** (a en la primera regla, b en la segunda regla). Veremos más adelante que esta es la única dificultad que evita la eliminación de la información de multistack inicial (e incluso del índice de comienzo del parsing), lo cual nos permitirá aplicar dicha optimización al caso de SAT.

Por último, vale la pena mencionar una propiedad relevante, y es que la modificación del algoritmo en la cual se elimina el multistack inicial únicamente puede producir falsos positivos, pero nunca un falso negativo. Esto es evidente ya que cualquier ítem que el algoritmo original agrega a algún conjunto, es agregado por el algoritmo sin la multistack al mismo conjunto (pero sin la multistack inicial), ya que los pasos realizados son idénticos pero sin la verificación de que coincidan los multistack durante el completer.

4.2. Modificación para producir un árbol de derivación

Las versiones que hemos dado de los algoritmos Earley y Earley-MGIG funcionan como *reconocedores*, es decir que dada una cadena de entrada únicamente determinan si pertenece o no al lenguaje generado por la gramática. Es natural querer utilizar el algoritmo para construir un árbol de derivación en el caso de que la cadena de entrada pertenezca al lenguaje.

Es sabido que la cantidad de árboles de derivación para una palabra en el lenguaje de una gramática libre de contexto puede ser exponencial en la longitud de la palabra, y por lo tanto lo mismo ocurrirá en el caso de MGIG. También hemos mostrado anteriormente que es posible que el tamaño de un árbol de derivación sea necesariamente exponencial en la longitud de la cadena de entrada (si existen producciones λ). Sin embargo, el algoritmo de Earley permite construir un grafo dirigido de punteros entre los ítems de los conjuntos S_i, de tamaño total polinomial, de manera que **todos** los posibles árboles de derivación se obtienen para cierta elección de caminos en el grafo.

Más precisamente, a cada ítem de Earley podemos asignarle una lista de *predecesores*, de la siguiente manera:

- Los ítems de la forma $A \to \bullet\gamma$ fueron agregados por el paso Predictor (excepto el ítem inicial, $S' \to \bullet S$), corresponden al **comienzo** del procesamiento de un no terminal A, y por lo tanto no tendrá predecesores.

- Los ítems de la forma $A \to \alpha\omega \bullet \beta$, con $\omega \in T$, fueron agregados por el paso Scanner, a partir de un ítem anterior I_a de la forma $A \to \alpha \bullet \omega\beta$. A todos los posibles ítems I_a los consideraremos anteriores del ítem que estamos considerando, y por lo tanto en cada paso Scanner se agregará un ítem a la lista de predecesores del ítem que se genera[1].

[1] Los ítems agregados a través de Scanner tendrán un único predecesor, ya que el ítem anterior está determinado a partir del ítem en consideración, y pertenece necesariamente al conjunto inmediatamente anterior. No obstante, lo describimos en términos de una *lista* de predecesores por homogeneidad, ya que tal lista es necesaria para el paso Completer.

- Los ítems de la forma $A \to \alpha B \bullet \beta$, con $B \in N$, fueron agregados por el paso Completer, a partir de **dos** ítems anteriores: un ítem I_a de la forma $A \to \alpha \bullet B\beta$, que fue completado, y un ítem I_b de la forma $B \to \gamma\bullet$ que se utilizó para completar B. A la tupla (I_a, I_b) la consideraremos un predecesor del ítem que estamos considerando, de manera que su correspondiente lista de predecesores estará formada por pares de ítems anteriores. Por lo tanto en cada paso Completer se agregará un par de ítems (I_a, I_b) a la lista de predecesores del ítem que se genera.

Teniendo almacenadas estas listas de predecesores que son calculadas de manera natural durante la ejecución del algoritmo, se tiene una representación implícita de todos los árboles de derivación posibles. En el caso de CFG esta representación tiene tamaño polinomial $O(n^3)$. En el caso de MGIG, el tamaño podría ser exponencial en n y dependerá del tamaño de los conjuntos generados.

Para obtener un árbol de derivación, se puede proceder recursivamente comenzando por el ítem final de la forma $(0, S' \to S\bullet)$. En un paso recursivo en que se procesa la regla $A \to t_1 t_2 \cdots t_k\bullet$, se obtiene un árbol de derivación para A como raíz, utilizando esa regla como primer paso de derivación.

Para ello elegimos k veces en sucesión, un elemento de la lista de antecesores, y en el caso de un paso Completer, tomamos el primer elemento de la tupla (I_a, I_b). De esta manera tendremos k ítems de la forma $A \to t_1 t_2 \cdots t_{k-1} \bullet t_k$, $\cdots$, $A \to t_1 \bullet t_2 \cdots t_{k-1} t_k$, $A \to \bullet t_1 t_2 \cdots t_{k-1} t_k$. Los k elementos $t_1, t_2, \cdots, t_k \in N \cup T$ serán los hijos de la raíz A en el árbol de derivación. Y además, para cada hijo no terminal $t_i \in N$, tenemos asociado un ítem I_b, a partir del cual podemos reconstruir un subárbol correspondiente a ese hijo recursivamente.

El procesamiento y reconstrucción que hemos descrito aplica inalterado tanto al algoritmo original de Earley como al Earley-MGIG descrito. Todo árbol de derivación válido para la cadena de entrada puede ser construido por este procedimiento, eligiendo adecuadamente los elementos que sean necesarios de las listas de predecesores.

4.3. Representación de multistacks

4.3.1. Trie de stacks

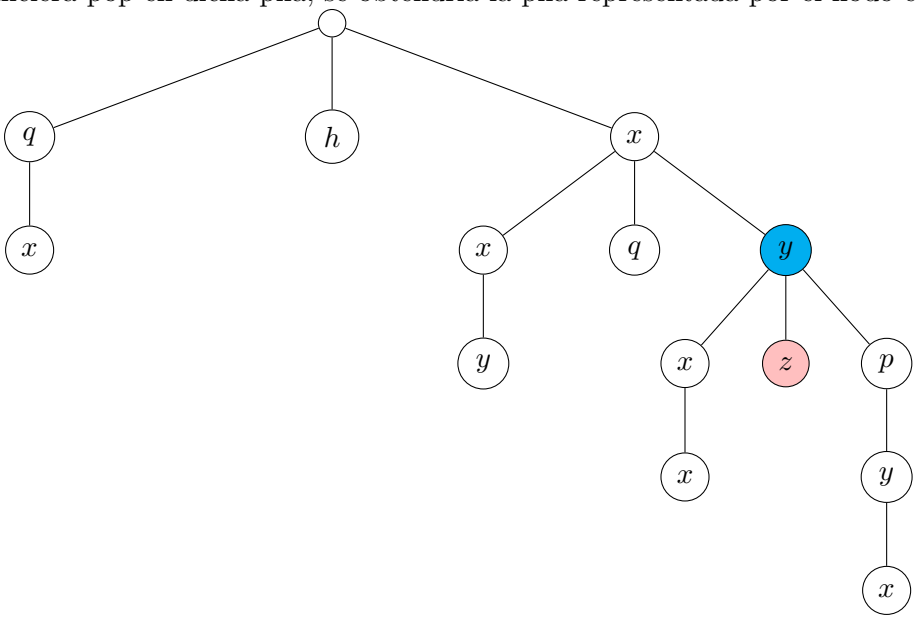
Un multistack es por definición una lista ordenada de stacks, siendo un stack una cadena de índices $\eta \in I^+$. En la implementación del algoritmo, representaremos una multistack como una lista de ordenada de punteros o identificadores de stack, donde cada puntero representará una stack en particular.

Los punteros (o identificadores) identificarán un nodo de un **trie**[17], correspondiente a la cadena η que identifica al stack. Los nodos más cercanos a la raíz del trie se corresponderán con los índices más cercanos a la base de pila, y el nodo apuntado por el correspondiente puntero en un multistack se corresponde con el tope de la pila correspondiente. Cada nodo del trie almacenará un diccionario de nodos hijos en el trie, indexado por el índice correspondiente en I, así como un puntero al nodo padre para permitir hacer pop del último índice fácilmente.

Mantendremos un único trie global durante la ejecución de todo el algoritmo. De esta manera las operaciones de push, pop y comparación de stacks por igualdad pueden

realizarse en $O(1)$[2], con un uso total de memoria en el trie a lo sumo proporcional al total de operaciones realizadas durante la ejecución del algoritmo.

En el trie que se muestra a continuación, el nodo coloreado en rosa corresponde una pila en la cual se han insertado en orden x, y y luego z, con lo cual z está en el tope de pila y es el próximo elemento a popear, mientras que x se encuentra en la base de pila. Si se hiciera pop en dicha pila, se obtendría la pila representada por el nodo en azul.



En esta figura por comodidad, mostramos los caracteres del alfabeto (en nuestro caso serán índices del conjunto I) en los nodos del árbol del trie. Conceptualmente, los caracteres se ubican en un trie en las aristas, de modo que la cadena representada por un nodo viene dada por los caracteres presentes en las aristas del camino desde la raíz hasta el nodo. En nuestra implementación, por comodidad, almacenamos estos índices tanto en nodos como en aristas. Más precisamente, en cada nodo v tenemos un diccionario cuyas claves son índices, y sus valores son los correspondientes nodos hijos de v que se alcanzan siguiendo aristas marcadas con el correspondiente índice. Además, cada nodo guarda el índice de la arista que lo conecta con su padre, como se muestra en la figura.

4.3.2. Unificación de ítems por lista de topes de multistack

4.3.2.1 Descripción de la representación

Para intentar mejorar la eficiencia del algoritmo reconocedor, la idea fundamental con la que hemos experimentado y trabajado durante el desarrollo de esta tesis es la noción de **unificación** de multistacks en el algoritmo de Earley. En el contexto de este trabajo, cuando hablamos de unificación nos referimos al proceso de fusionar múltiples multistacks en un solo "nodo" que las represente a todas a la vez, de manera que puedan ser procesadas todas en conjunto operando únicamente con el nodo que las resume, en lugar de tener que realizar las operaciones individualmente sobre cada multistack posible.

[2] Asumiendo un costo $O(1)$ en las operaciones del diccionario de hijos interno, por ejemplo con una buena implementación de tablas hash, o mediante el uso de arreglos directamente cuando es razonable considerar que el conjunto de índices tiene tamaño $O(1)$

La idea central está basada fuertemente en la noción análoga introducida por el autor de [46], en el cual la utiliza para dar un algoritmo de Parsing-GLR con resultados de igual generalidad que el algoritmo de Earley, pero basados en un autómata clásico LR. Durante el desarrollo de este trabajo hemos considerado versiones más básicas y eficientes de unificación, pero todas esas versiones previas resultaron incorrectas incluso en el caso particular del lenguaje TAS, que es el que nos interesa fundamentalmente en este trabajo. Estas versiones se mencionan brevemente en la sección siguiente. Describimos a continuación la versión final (y correcta) de unificación utilizada.

La observación fundamental es que dada una multistack de la cual conocemos únicamente su **lista de topes de stack**, no necesitamos conocer su composición interna para poder operar con ella **mediante operaciones push**. De aquí surge que si tenemos dos multistacks con **idéntica lista de topes de stack**, su procesamiento por operaciones push será completamente idéntico.

Por ejemplo, para el caso de TAS, dadas las multistacks `1a2b 3a#2b#` y `1a2b#2b 3a 4a#`, la lista de topes de stack es { `1a2b , 2b` } en ambos casos. Esto es suficiente información para procesar cualquier push ya que por ejemplo, si ahora se hace push de `3a`, con esa información alcanza para saber que se creará una tercera pila con dicho push, quedando así la nueva lista de topes de stacks { `1a2b , 2b , 3a` }. O si se pushea `2a`, tenemos suficiente información para saber que se hará el push sobre la segunda pila, quedando la lista de topes { `1a2b , 2a` }.

Teniendo esto en consideración, si existen dos ítems de Earley-MGIG en un mismo conjunto, que difieran únicamente en sus multistacks, pero de forma tal que ambos tengan la misma lista de topes de pila, el procesamiento futuro de dichos ítems será completamente idéntico en tanto no aparezcan operaciones de pop o pop-push. Lo que hacemos entonces es representar ambos ítems como un único ítem que codifique todas sus multistacks posibles. De esta forma, **cada ítem** en el algoritmo con unificación tiene asociada una representación de sus posibles multistacks, con lo cual representa en realidad a todos los ítems que coincidan en los demás valores y difieran únicamente en el multistack.

Describimos a continuación la representación del conjunto de las multistacks asociadas a un cierto ítem. En cada ítem, almacenamos un diccionario cuyas claves serán **lista de topes de stack**, y que asocia a cada una de ellas un **conjunto de descripciones de multistack**. El conjunto de multistacks representado por tal conjunto de descripciones, es la unión de los conjuntos representados por cada descripción individual del conjunto.

Una **descripción de multistack** particular es un par ordenado (b, l), donde b es un puntero a un conjunto de descripciones de multistack que se toma como base, y l es una **lista ordenada de descripciones de stack** que describen cada stack que compone el multistack, en el orden del multistack. Una **descripción de stack** particular es un par ordenado (i, t), que indica que para obtener esta stack se debe hacer push de la cadena de índices t (dada como un nodo del Trie de stacks) sobre la stack i-ésima del multistack ξ que se toma como base. Si se indica en t la cadena vacía (es decir, la raíz del Trie de stacks), la correspondiente stack se obtiene como una copia idéntica de la stack i-ésima del multistack ξ. Si se indica un valor nulo para i, la correspondiente stack es una nueva stack iniciada directamente con el valor t (equivalentemente, se obtiene de pushear t sobre una stack vacía). Como la descripción anterior depende del ξ, esta descripción de multistack describe un conjunto de multistacks: este es el conjunto de todas las multistacks que se obtienen de la manera indicada, para todas las ξ que describe el conjunto de descripciones de multistack dado por b.

4.3.2.2 Ejemplo

Por ejemplo, si tenemos las reglas:

$$\begin{array}{rcl} S' & \rightarrow & S \\ S & \rightarrow & X \\ S & \underset{a}{\rightarrow} & S \end{array}$$

En el diccionario del ítem correspondiente a $S' \rightarrow \bullet S$ en S_0, tendremos inicialmente al comenzar el algoritmo un único mapeo, para la lista vacía de topes de stack [], con valor $\{(\text{NULL}, [])\}$. Denotaremos con $\star$ a este mapeo, de forma que un puntero $\star$ es un puntero a este mapeo.

Correspondiente a la derivación $S' \Rightarrow S \Rightarrow X$, tendremos en el diccionario del ítem correspondiente a $S \rightarrow \bullet X$ en S_0, un mapeo de [] en $\{(\star, [])\}$. Observemos que a pesar de que se copia inalterada la multistack correspondiente al otro conjunto, se crea una nueva descripción de multistacks, que contiene un puntero a la anterior tomada de referencia. Esto es importante dada la posibilidad de agregar elementos (descripciones de multistack) a estos conjuntos de descripciones de multistacks (que son las imágenes del diccionario). Si no se estableciera esta referencia, la copia directa ya no sería válida si se agregan descripciones al conjunto b tomado como base (en este caso, $\star$).

Finalmente, consideremos la derivación $S' \Rightarrow S \Rightarrow a\#S$. Asociado a ella, tendremos en el diccionario del ítem correspondiente a $S \underset{a}{\rightarrow} \bullet S$ en S_0, un mapeo de [a] en $\{(\star, [(\text{NULL}, \text{“a”})])\}$. Notamos $\star_2$ a este mapeo.

Ahora bien, cuando se considere un paso más de una derivación como $S' \Rightarrow S \Rightarrow a\#S \Rightarrow aa\#S$, tendremos una multistack más posible en el ítem $S \underset{a}{\rightarrow} \bullet S$ ya mencionado. Además, esta multistack tiene exactamente la misma lista de topes de pila, [a]. Luego se agregará la descripción de la correspondiente multistack al conjunto imagen de dicho mapeo, quedando este conjunto de la siguiente forma: $\{(\star, [(\text{NULL}, \text{“a”})]), (\star_2, [(0, \text{“a”})])\}$. Notar que en este último conjunto obtenemos una descripción que contiene una referencia al mismo conjunto, con lo cual vemos que esta representación podría, en el caso general, contener ciclos. Veremos esto mejor en 4.3.2.6.

4.3.2.3 Procesamiento de operaciones push, pop y pop-push

Una ventaja inmediata de esta representación es que si se llega a un mismo ítem con multistacks que se unifican, se procesarán de allí en adelante como si fueran una sola (hasta que una operación pop fuerce a examinar los antecesores del nodo donde se produce la unificación). Es decir, si como resultado de por ejemplo un push, se determina que se llega a un ítem ya existente con una lista de topes de stacks que ya existe para ese ítem, solamente se agregará la nueva descripción de multistack a la lista correspondiente, pero cualquier procesamiento que ya se haya hecho a partir de este nodo sigue siendo igualmente válido, y similarmente, cualquier procesamiento posterior se realizará una única vez y será válido para todas las descripciones de multistacks del conjunto.

Para realizar una operación de pop sobre una lista de descripciones de multistack, se lleva a cabo la operación sobre cada una. Por otra parte, como no es posible determinar la secuencia de topes de stack resultante como sí era posible para el push, lo que se obtendrá al hacer pop de una descripción ahora no es, en general, una única descripción

nueva, sino un **conjunto de muchas descripciones de multistack**, cuya unión representa todas las posibles que se obtienen haciendo pop desde las multistacks representadas por la descripción de partida.

En el caso de que la primera pila (de la cual se hace pop) contenga operaciones de push en la descripción, entonces sí se genera una única descripción nueva: una idéntica a la de partida pero sin el push del tope de la primera pila, y eventualmente con las pilas reordenadas / eliminadas como consecuencia de este pop. En el caso de que no sea así, para poder conocer los contenidos de la primera pila es necesario examinar las descripciones contenidas en el conjunto de base, ya que cada una de esas descripciones podría contener distintos valores para la primera pila y dar lugar así a distintas secuencias de topes de pila. Lo que se hace entonces es popear recursivamente de cada una de las descripciones de la lista base, y considerar todos los resultados de descripciones que se obtienen. Además, a cada uno de esos resultados se le agregan en la descripción de operaciones push, las operaciones push de la descripción actual, que justamente indican terminales que no son retirados por el pop.

En el caso de operaciones pop-push, el procesamiento es idéntico al anterior pero a las descripciones obtenidas, justo después de agregarle a sus descripciones las operaciones push de la descripción actual, se agrega una operación push correspondiente a la operación que se realiza, tal y como se hace con operaciones push normales.

Notar que cualquier secuencia de operaciones de push puede resumirse de forma tal que el orden de operaciones push entre distintas pilas no sea importante, indicando únicamente qué terminales fueron agregados a qué pilas (incluyendo la posibilidad de crear una nueva pila). Por este motivo basta con utilizar una lista de pares (i, t), **uno por cada stack de la multistack representada**. Notar que el índice i que indica el número de pila a utilizar como base es necesario porque, si bien el orden de las pilas no puede haber cambiado nunca como resultado de un solo push, sí puede cambiar ante pops, y en dicho caso representaremos las secuencias de push necesarias para llegar al nuevo multistack con esta misma estructura. Es también por este motivo que tiene sentido permitir nodos del Trie de stacks en general para t en lugar de especificar un único índice: Si bien las descripciones de multistack agregadas por el procesamiento de una operación push tendrán nodos que representan un único terminal, al volver hacia atrás procesando operaciones pop podrán aparecer cadenas más largas de terminales a pushear, que representan una secuencia de push realizados que no afectaron a la pila en la cual se realiza el pop.

Vale la pena destacar que como resultado de operaciones push, pop, o pop-push, siempre se crean **nuevas** descripciones de multistack, y nunca se reutilizan descripciones existentes: estas solamente se utilizan para especificar la base de una nueva descripción.

4.3.2.4 Conjunto de operaciones pop pendientes

La situación descrita anteriormente permite reducir los cómputos al compactar lo que eran muchas multistacks distintas en un solo elemento (y por ejemplo, si se filtra o se descarta como imposible de continuar la derivación desde dicho nodo, todos esos descartes se realizan una sola vez para todas las multistacks correspondientes), pero trae un inconveniente, que resulta del procesamiento de las operaciones de tipo pop: como en cualquier momento pueden agregarse descripciones a una lista como parte del algoritmo, si ya se ha realizado alguna operación de pop que utilice esta lista, dicha operación de pop fue procesada de manera incompleta, pues lógicamente debiera haberse utilizado también la descripción que se está agregando por ser parte de la lista, pero esta no existía al momento

en que el pop fue procesado.

Se debe entonces ser cuidadosos para **reprocesar** los pop en dichas situaciones. Solamente las descripciones que se han agregado desde que el pop fue procesado necesitan ser procesadas con el pop. La información necesaria para procesar uno de estos pops anteriores al agregar una nueva descripción de multistack a un conjunto es la siguiente:

- El ítem de Earley inicial que desencadenó el pop (pues determina la producción anotada con • del nuevo ítem, y el conjunto de Earley donde debe crearse).
- La lista de pares (i, t) (índice de stack y cadena que se pushea) que describen las operaciones adicionales que se deben hacer sobre el multistack que resulte luego del pop. Si el conjunto al que se agrega una descripción fuera siempre el comienzo de una operación pop, esto no sería necesario, pero es posible que se agreguen descripciones a un conjunto cualquiera, que es un paso intermedio en el recorrido del pop.
- El terminal que se debe pushear a continuación (solo para operaciones pop-push).

En nuestra representación del conjunto de descripciones de multistack, almacenaremos entonces las descripciones de multistack separadas en dos: Por un lado, aquellas para las cuales ya se han procesado todos los pops. Por otro, aquellas para las cuales no se ha procesado ningún pop. Además de esta separación, almacenamos en cada conjunto de descripciones de multistacks, un conjunto con todos los pops realizados (más precisamente, con la información anteriormente mencionada para cada uno de esos pop).

Al realizar un pop nuevo sobre un conjunto, ese se realiza únicamente sobre las descripciones de multistacks del conjunto de ya procesadas. Además, la información del nuevo pop es registrada, para que se puedan procesar las demás descripciones luego (así como todas las que sean agregadas al conjunto en un futuro).

Cuando se agrega una descripción de multistack nueva al conjunto, se agrega al conjunto de las no procesadas aún.

Cuando se deciden procesar pops pendientes para este ítem (lo cual se realiza como parte de la iteración de Earley: un conjunto de Earley solamente estará finalizado, cuando se hayan procesado por completo todos sus ítems, y no está completado si algún ítem tiene pops pendientes de procesar), se procesan con la información de los pops, todas las descripciones de multistacks del conjunto de pendientes de procesar, y además todas ellas se pasan al conjunto de las procesadas.

4.3.2.5 Impacto en memoria de la unificación por lista de topes de stack

Notar que este mecanismo de unificación trae consigo varios inconvenientes. En primer lugar, podemos mencionar una mayor utilización de memoria, debido al costo de punteros y estructuras adicionales que se referencian entre sí que tiene esta versión, en comparación con el algoritmo base, cuyos ítems únicamente almacenan una lista sencilla de punteros a nodos de un Trie que puede manejarse eficientemente.

Más importante aún, como con esta estructura cada descripción depende efectivamente de **toda la historia** de operaciones realizadas durante la derivación, aún si utilizamos una versión del algoritmo de Earley-MGIG para la cual alcance con almacenar solamente el conjunto de Earley actual y el siguiente en construcción (como mencionamos en 5.1.5), será necesario guardar las listas de descripciones de multistacks correspondientes a todos

los conjuntos, ya que las listas de conjuntos posteriores contienen referencias a las de conjuntos anteriores.

Este problema de uso de memoria podría mitigarse parcialmente, por ejemplo, si la representación se compacta al terminar de procesar cada conjunto de Earley, eliminando cualquier lista de descripciones que contenga solamente un elemento, y reemplazándola directamente por una copia del elemento correspondiente con las operaciones de push necesarias directamente codificadas en las descripciones de la misma. Explorar esta potencial optimización quedó fuera del alcance de este trabajo.

El segundo inconveniente importante es el hecho de que ya no se están almacenando en los ítems multistacks, entendidos estrictamente como listas de stacks, es decir, una lista ordenada de cadenas de índices, sino que la representación usada **distingue la historia** que lleva a tener tal multistack. Es decir, con la versión sencilla sin unificación, si llegamos a la misma multistack exacta por dos derivaciones bien distintas, en un mismo ítem de Earley, el ítem que contiene esa multistack se inserta y procesa una única vez. Realizando la unificación de la manera que hemos explicado, se distingue la historia de creación de la multistack (a través de la secuencia de referencias en las descripciones de operaciones push a realizar, la historia exacta se preserva al menos desde el último pop o pop-push).

Por ejemplo, desde la multistack inicial vacía, las secuencias `push b - push a - push c` y `push b - push c - push a`, con `a <b <c`, llevan a multistacks idénticas $ab\#c\#$, pero las descripciones de multistack serán diferentes, pues se obtienen haciendo push de índices distintos a multistacks que no estaban unificadas (en particular, tenían distintas listas de topes de stack, $[a]$ y $[b, c]$ respectivamente). Si bien para futuros push serán unificadas (por tener la misma lista de topes de stack, al ser de hecho multistacks idénticas), a la hora de procesar los pop serán procesadas de manera separada porque al tener distinta historia (es decir, depender de distintas listas anteriores de descripciones de multistacks) podrían dar lugar a distintos conjuntos de nuevas descripciones (por ejemplo, si posteriormente se agregan nuevas descripciones a esas listas que ya no sean equivalentes a las originales: la mutabilidad de los conjuntos durante el algoritmo hace imposible saber a priori si estos conjuntos que ahora son iguales lo seguirán siendo durante todo el algoritmo).

En resumen, los conjuntos de multistacks descritos por las distintas descripciones de una misma lista no son disjuntos entre sí, y esto no es sencillo de evitar porque **la dependencia del conjunto de su historia es lo que permite agregar descripciones a listas anteriores** sin arruinar el procesamiento posterior ya realizado. En otras palabras, los conjuntos de multistacks representados por una descripción pueden cambiar dinámicamente. Por otra parte, como solamente las listas correspondientes a ítems de los conjuntos de Earley que se encuentran actualmente en procesamiento pueden ser modificadas, es posible en principio realizar una eliminación de multistacks repetidas al completar el procesamiento de cada conjunto. Esta idea se relaciona con la optimización mencionada anteriormente para la memoria, y al igual que aquella su estudio quedó fuera del alcance de este trabajo.

Finalmente, como la representación de una multistack con estas ideas consiste de un "nodo unificado" que está asociado a un ítem de Earley específico, almacenar la multistack inicial en este caso implica almacenar un puntero al ítem de Earley correspondiente, de manera que en completer será necesariamente este ítem particular el que se completa. Esta limitación a solamente completar dicho ítem viene dada por el hecho de que esta representación distingue la historia de un multistack y por lo tanto solamente es correcto completar el ítem donde se inició el parsing del ítem que se usa para completar. Por

simplicidad, no hemos implementado la versión general de este algoritmo, que tiene en cuenta como aquí se describe las multistacks iniciales en los ítems de Earley, sino que solamente hemos implementado sobre el caso particular de gramáticas determinadas a derecha, que se describe más adelante en 5.1.1 y es el caso relevante para modelar por ejemplo el problema SAT.

4.3.2.6 Sobre ciclos en la representación

Algo muy importante a considerar es la posibilidad de que se formen ciclos de referencias entre conjuntos de descripciones de multistacks. En el ejemplo anteriormente mencionado, el ciclo obtenido pone de manifiesto las infinitas derivaciones parciales posibles de la forma $S' \Rightarrow S \Rightarrow a\#S \Rightarrow aa\#S \Rightarrow aaa\#S \Rightarrow \cdots$. Notemos que este ciclo en particular no puede ocurrir nunca en una MGIG debido a la restricción en las producciones push, que deben necesariamente introducir un terminal en la forma sentencial, y por lo tanto implican un avance al siguiente conjunto de Earley (de S_i a S_{i+1}).

Consideremos entonces por ejemplo, una situación en la cual el conjunto $l1$ contiene una descripción de multistack que toma por base al conjunto $l2$, que contiene una descripción que toma por base al $l3$, que contiene una descripción que toma por base al $l1$. En tal caso, el procesamiento de un pop o pop-push desde $l1$, que efectivamente recorre las descripciones hacia atrás hasta encontrar una descripción adecuada con información suficiente para realizar el pop, podría recorrer $l2$, luego $l3$, y finalmente volver a $l1$, con lo cual el procesamiento de dicha operación no terminaría nunca. Lo que puede hacerse para evitar este problema es dejar de recorrer cuando se vuelve a una lista ya visitada, en otras palabras, recorrer las listas hacia atrás en forma de DFS (deteniéndose cuando se encuentra una descripción con información adecuada para hacer el pop, o bien cuando se vuelve a una lista ya visitada por el recorrido).

Lo anterior funciona bien porque es posible demostrar que cualquier ciclo que ocurra en esta estructura, estará formado únicamente por descripciones de copia introducidas en el procesamiento de reglas libres de contexto (sin operaciones de pila). En efecto, como en las MGIG una operación de push siempre lee del input, involucra una operación scanner, y por lo tanto si una descripción de multistack fue introducida por scanner en el ítem S_{i+1}, su base necesariamente está en el S_i, lo que evita que sea parte de un ciclo de referencias.

De manera similar, al crear una descripción como resultado de un pop o pop-push en un cierto conjunto, esta se agrega tomando como base una descripción que debe necesariamente estar en un conjunto anterior al que se está procesando actualmente, ya que sobre dicha descripción fue realizado al menos un push para llegar al ítem actual en procesamiento (concretamente, aquel push que se está anulando mediante este pop) y por lo tanto estos casos tampoco pueden estar involucrados en un ciclo de referencias.

Como un ciclo de referencias solamente ocurre entonces por operaciones libres de contexto que mantienen el multistack intacto, el único efecto del ciclo es "unir" las lista de descripciones involucradas, de manera que cualquier descripción que se agregue a alguna de ellas afecta a todas las demás inmediatamente. Este efecto se logra inmediatamente al recorrer con un DFS de la forma que se mencionó anteriormente.

En nuestra implementación, asumimos simplemente que no se formará nunca ningún ciclo en el grafo de dependencias entre conjuntos de descripciones de multistacks, con lo cual evitamos la leve complicación adicional de verificar si ya hemos recorrido el conjunto que se visita en cada paso durante el procesamiento de las operaciones pop y pop-push. Esto no ocurre nunca para las gramáticas de SAT, ya que todas son libres de ciclos en sus

producciones libres de contexto.

4.4. Complejidad

4.4.1. Earley clásico para CFG

La complejidad del algoritmo de Earley para CFG es $O(RGn^3)$, siendo n la longitud de la cadena de entrada, G la cantidad de reglas de la gramática y R la longitud total de las reglas de la gramática (cantidad de caracteres necesarios para escribir todas las reglas consecutivamente).

Esta complejidad puede obtenerse sumando los costos totales de los tres pasos del algoritmo. Para esto conviene notar el tamaño de los conjuntos generados por el algoritmo. Cada conjunto contendrá $O(Rn)$ elementos, pues un ítem consta de un índice i de comienzo del parsing (n opciones) y una regla de producción anotada con $\bullet$ (R opciones). En un paso de Scanner se agrega un único ítem al conjunto siguiente, y puede realizarse en $O(1)$. Un paso de Predictor por otra parte implica agregar $O(G)$ reglas al conjunto actual, con lo cual tendrá complejidad $O(G)$. Como hay n conjuntos, de aquí obtenemos que la complejidad total de los pasos Predictor y Scanner será $O(n \cdot Rn \cdot (G+1)) = O(GRn^2)$.

La complejidad total del algoritmo de Earley está dominada por el paso Completer. El costo de un paso completer es $O(T)$, siendo T la cantidad total de ítems que son completados en dicho paso. Por lo tanto el costo total invertido por el algoritmo en Completer será $O(Rn^2 \cdot T)$, o también, $O(Rn^2 \cdot T_{item})$, siendo T_{item} la máxima cantidad total de veces que un ítem en particular es generado por completer (solo la primera de esas veces será agregado verdaderamente a su conjunto). Para generar con completer un ítem en el conjunto j de la forma $(i, X \to \alpha A \bullet \beta)$, debe haberse realizado la operación completer sobre un ítem $(k, A \to \gamma\bullet)$ en el conjunto j, habiendo completado a su vez un ítem en el conjunto k de la forma $(i, X \to \alpha \bullet A\beta)$. Como hay $O(n)$ valores posibles k y $O(G)$ reglas de la gramática posibles para completar A, resulta $T_{item} = O(Gn)$. De aquí se sigue que la complejidad total invertida en Completer es $O(Rn^2 \cdot Gn) = O(RGn^3)$, que domina la complejidad del algoritmo.

Como se nota en [18], el análisis anterior permite mejorar la complejidad indicada (sin modificar el algoritmo) en casos de interés en los cuales podamos mejorar la cota general para T_{item}. Particularmente importante es el caso de las gramáticas no ambiguas (o en general, "con poca ambigüedad"), en las cuales $T_{item} = O(1)$, y por lo tanto el algoritmo de Earley resulta automáticamente $O(Rn^2 + GRn^2) = O(GRn^2)$.

4.4.2. Earley-MGIG

Podemos realizar un análisis de complejidad análogo para Earley-MGIG, y la diferencia fundamental radicará en un crecimiento exponencial de la cantidad de ítems, debido a la gran cantidad de posibles multistacks. En lo siguiente, llamaremos M a la máxima cantidad de *multistacks* diferentes que pueden aparecer en algún ítem durante la corrida del algoritmo. Llamaremos k a la máxima cantidad de *stacks* que pueden aparecer en un mismo multistack durante la corrida del algoritmo. Claramente $k = O(M)$.

4.4.2.1 Algoritmo general

Al igual que antes tenemos n conjuntos, y queremos contar el costo total de las operaciones realizadas durante el algoritmo de Earley-MGIG.

La cantidad total de ítems es ahora $O(Rn^2M^2)$, pues en cada ítem se especifican adicionalmente a los datos usuales del algoritmo de Earley, dos multistacks (la "actual" y la "inicial").

Las operaciones de multistack pueden realizarse razonablemente en tiempo $O(k)$, pues son operaciones de punteros y eventualmente recorrer la lista de multistacks, de tamaño $O(k)$. Por este motivo las operaciones Scanner y Predictor cuestan $O(1)$ y $O(kG)$ respectivamente, con un análisis análogo al anterior. La complejidad total de los pasos Scanner y Predictor resulta $O(RGkn^2M^2)$.

Además, como antes, debemos considerar cuántas veces puede ser generado un ítem durante el paso Completer. Esta cantidad es ahora $O(GnM)$ pues además de especificar el valor k de inicio del parsing y la regla de la gramática usada para completar, hay que especificar el valor de la multistack inicial del ítem intermedio utilizado para completar, lo cual se puede hacer de $O(M)$ formas.

La complejidad final resulta $O(RGkn^2M^2 + GRn^3M^3) = O(GRn^3M^3)$

4.4.2.2 Algoritmo que guarda únicamente multistack actual

Más adelante en 5.1.5 veremos un algoritmo para un caso particular de gramáticas MGIG que es esencialmente idéntico a Earley-MGIG pero prescindiendo del multistack de inicio y del índice de inicio en los ítems. Es decir, los ítems tienen simplemente la forma $(A \to \alpha \bullet \beta, \xi)$, siendo ξ el multistack actual.

La cantidad de ítems es en este caso $O(RnM)$. Además, por las particularidades de esta gramática el paso completer se realiza en $O(1)$, ya que la forma del ítem que se generará como parte del completer está unívocamente determinada por el no terminal que se completa. Luego la complejidad total del algoritmo resulta $O(GRknM)$, dominado ahora por las operaciones de pila realizadas en el paso predictor.

4.4.2.3 Cotas generales para M

Para dar una complejidad general que dependa únicamente de los parámetros de entrada del algoritmo, podemos notar que $M = O((2|I|)^n)$. En efecto, un multistack consta de no más de n símbolos del conjunto I, pero estos se encuentran distribuidos en una lista ordenada de multistacks. Como hay $|I|^n$ cadenas en I^* de longitud n y cada una puede partirse en multistacks de 2^{n-1} maneras diferentes (pues basta elegir para cada frontera entre dos índices, si allí se cruza de un multistack al siguiente o se sigue en el mismo), tenemos que $M = O(|I|^n \cdot 2^{n-1}) = O((2|I|)^n)$. Notar que esta cota no parece ser ajustada al peor caso, ya que se cuentan por ejemplo cadenas con ordenamientos inválidos dentro y entre las stacks.

4.4.2.4 Cotas para M en el caso de gramáticas para SAT

Cuando se utilizan las gramáticas restringidas mencionadas en 3, es decir, DTAS, RTAS y SRTAS, se puede mejorar la cota de la cantidad de multistacks posibles a $M = O(2^n l)$, siendo n la cantidad de variables presentes en la instancia de SAT considerada, y l la longitud de la cadena de entrada.

Esto se debe a que las multistacks que pueden generarse durante una derivación se corresponden con las distintas valuaciones de las variables, debido a que el único momento de una derivación para tales gramáticas en el cual la regla a aplicar no está forzada, es a la hora de elegir el valor de verdad de la siguiente variable. Además la cantidad de

operaciones de multistack durante una derivación es $O(l)$, pues en las gramáticas de SAT, cada vez que se aplica una regla pop-push se elimina un literal de la multistack que no será considerado nuevamente, y hay $O(l)$ literales en la fórmula.

5. EL ALGORITMO DE RECONOCIMIENTO DE MGIG Y SU USO COMO SAT-SOLVER

En 3, se mostraron diferentes ejemplos de MGIG cuyo reconocimiento equivale de manera natural a resolver instancias arbitrarias del problema SAT. Cuando queremos enfocarnos en utilizar el reconocimiento de MGIG como mecanismo para resolver SAT, no necesitamos entonces de un algoritmo de reconocimiento general como el mostrado anteriormente, sino que alcanza con poder reconocer (alguna de) las gramáticas concretas propuestas para SAT.

Con esta idea en mente, experimentamos con la correctitud de diversos enfoques para intentar reducir el tiempo de ejecución con respecto al algoritmo para MGIG general. En este capítulo se explican estos diferentes enfoques.

5.1. Eliminación de multistack de partida

Todos los enfoques que hemos estudiado para el reconocimiento de SAT se han basado desde el comienzo del trabajo en la idea de utilizar una única multistack en los ítems de Earley, en lugar de dos como requiere el algoritmo general planteado anteriormente. En esta sección establecemos un resultado general sobre una clase de MGIG para las cuales eliminar de los ítems el multistack de partida, dejando únicamente el multistack actual, resulta en un algoritmo correcto para su reconocimiento.

En el proceso de establecer este resultado, descubrimos que de hecho no es necesario guardar el índice de comienzo del parsing en este tipo de gramáticas, necesitándose únicamente el multistack actual en el ítem; con lo cual es posible reducir aún más el tamaño de los ítems utilizados.

5.1.1. MGIG determinadas a derecha

Decimos que una MGIG es una *MGIG determinada a derecha* si satisface que para todo no terminal $A \in N$ distinto del símbolo inicial S, cada regla que contiene a A del lado derecho cumple una de las siguientes condiciones:

Tipo 1 La regla tiene la forma $A \underset{\mu}{\rightarrow} \gamma A$ con $\gamma \in (N \cup T)^*$ una cadena que no contenga a A.

Tipo 2 La regla tiene la forma $B \underset{\mu}{\rightarrow} \gamma A \beta$ con $B \in N$, $B \neq A$, $\gamma, \beta \in (N \cup T)^*$ cadenas que no contengan a A, y además B y β deben **tomar siempre los mismos valores** para todas las reglas de este tipo correspondientes al no terminal A.

- Notar que el tipo de una regla está definido con respecto a un no terminal A.

Es decir, si ignoramos las reglas tipo 1, siempre que aparezca un A a la derecha, su *continuación* β deberá ser la misma, y el no terminal a la izquierda de la regla también.

Además, el símbolo inicial no deberá aparecer nunca a la derecha, salvo en reglas tipo 1. Notar que esto equivale a tratarlo igual que a los demás símbolos no terminales si agregamos una regla $S' \rightarrow S$ y ponemos a S' como nuevo símbolo inicial.

5.1.2. MGIG fuertemente determinada a derecha

Decimos que una MGIG es una *MGIG fuertemente determinada a derecha* si es determinada a derecha, y además cumple que para todo símbolo $A \in N$ que no sea el símbolo inicial S existe una única regla de tipo 2 que lo tiene en el lado derecho.

Nuevamente, notar que el símbolo inicial S cumplirá esto al agregar la regla $S' \to S$ (que será la única regla de tipo 2 para S).

5.1.3. Relación de antecesor entre no terminales

Consideremos un no terminal A. En una MGIG determinadas a derecha, todas las reglas de tipo 2 para A (de existir) deben tener a un mismo no terminal B en el lado izquierdo. De esta forma, podemos decir que B es *el antecesor* de A. Además, todo símbolo alcanzable desde el símbolo inicial S (exceptuando al mismo S) tiene necesariamente un antecesor (que también será alcanzable), pues de lo contrario sería imposible introducirlo en una derivación.

Por lo tanto, si consideramos a los no terminales alcanzables como nodos, podemos formar un grafo dirigido para la relación de antecesor, colocando una arista de A a B siempre que B sea el antecesor de A. Como todo nodo (excepto S) tiene grado de salida exactamente 1, y además la relación de antecesor siempre produce un camino hacia S (o sino, el no terminal no sería alcanzable), esta relación genera de hecho un **árbol** con raíz en S.

5.1.4. Imposibilidad de recursión a izquierda

Notemos que las MGIG determinadas a derecha, de manera similar a lo que ocurre con gramáticas LL [2], no permiten recursión a izquierda. Más precisamente, si A es un no terminal alcanzable, entonces no puede existir una regla:

$$A \underset{\mu}{\to} A\gamma$$

Con $\gamma \neq \emptyset$, pues entonces A debería ser su propio antecesor en la MGIG, en contradicción con que el símbolo es alcanzable.

Notar que las reglas de tipo 1, que están expresamente permitidas, son un ejemplo explícito de recursión a derecha.

5.1.5. Correctitud del Earley-MGIG con una única multistack

Probaremos que el algoritmo de Earley-MGIG modificado para utilizar una única multistack es correcto cuando se lo aplica para reconocer una MGIG determinada a derecha. Ya hemos visto en 4.1.4 que esto no es válido para el caso de una MGIG general. Para eso modificaremos la propiedad fundamental teniendo en cuenta que los ítems ya no indican una multistack inicial.

Concretamente, consideraremos los conjuntos $S_0 \cdots S_n$ de ítems, que ahora tendrán la forma $(i, \xi, A \to X \bullet Y)$, donde ξ es un multistack (notar que ya no hay ξ_i). En estas condiciones, el enunciado más natural de propiedad fundamental, por analogía con las anteriores, sería el siguiente:

(Incorrecta) ***Propiedad fundamental*** del algoritmo de Earley-MGIG (sin multistack inicial en los ítems):

Un ítem $(i, \xi, A \underset{\mu}{\rightarrow} X \bullet Y)$ pertenece al conjunto S_j al terminar el algoritmo, si y solo si existe una derivación por izquierda de la forma:

$$S' \overset{*}{\Rightarrow} \xi_i \alpha_0 \alpha_1 \cdots \alpha_{i-1} A \gamma \Rightarrow \mu(\xi_i) \alpha_0 \alpha_1 \cdots \alpha_{i-1} X Y \gamma \overset{*}{\Rightarrow} \xi \alpha_0 \alpha_1 \cdots \alpha_{i-1} \alpha_i \alpha_{i+1} \cdots \alpha_{j-1} Y \gamma$$

Para **alguna** multistack ξ_i, donde hemos notado $\mu(\xi)$ a la multistack que resulta de aplicar a ξ la operación de multistack indicada por μ (potencialmente ninguna)

Notemos especialmente que esta propiedad la obtenemos de las que veníamos utilizando, cambiando en el enunciado de la propiedad el ξ_i concreto del ítem bajo estudio, por un ξ_i arbitrario o libre que debe existir y cumplir lo mismo que cumplía antes el ξ_i presente en el ítem. Es decir debe existir un ξ_i tal que si imaginamos el ítem extendido agregando el ξ_i correspondiente al algoritmo general, se cumple lo pedido por la propiedad fundamental para el caso del algoritmo general.

La propiedad fundamental modificada de esta manera resulta **incorrecta** incluso para el caso de MGIG fuertemente determinada a derecha. Como ejemplo podemos dar la siguiente gramática fuertemente determinada a derecha:

$$\begin{array}{lcl} S & \rightarrow & xS \\ S & \rightarrow & C \\ C & \rightarrow & PA \\ P & \rightarrow & x \\ P & \underset{q}{\rightarrow} & xP \\ A & \underset{\overline{q}}{\rightarrow} & yA \\ A & \rightarrow & z \end{array}$$

Al ejecutar el algoritmo para esta gramática sin guardar la multistack inicial en los ítems, teniendo $xxyz$ como cadena de entrada, los siguientes dos ítems estarán en el conjunto S_2:

$$\begin{array}{l} (0, \{q\}, C \rightarrow P \bullet A) \\ (1, \{\}, C \rightarrow P \bullet A) \end{array} \quad (*)$$

El primero de ellos corresponde a la derivación parcial

$$S \Rightarrow C \Rightarrow PA \Rightarrow q\#xPA \Rightarrow q\#xxA$$

Mientras que el segundo se corresponde con

$$S \Rightarrow xS \Rightarrow xC \Rightarrow xPA \Rightarrow xxA$$

Al ser procesados en Predictor, generarán los nuevos ítems

$$\begin{array}{c} (2, \{q\}, A \underset{\overline{q}}{\rightarrow} \bullet yA) \\ (2, \{\}, A \underset{\overline{q}}{\rightarrow} \bullet yA) \\ (2, \{q\}, A \rightarrow \bullet z) \\ (2, \{\}, A \rightarrow \bullet z) \end{array}$$

respectivamente. De estos, claramente el único que podrá ser avanzado es el primero, que tiene una multistack adecuada para permitir el pop, y un símbolo para leer que se corresponde con el siguiente de la entrada. Sin embargo, una vez completado el no terminal A, a la hora de realizar el paso Completer se completarían **ambos** ítems mostrados en $(*)$, puesto que no hay información que permita realizar el chequeo de multistacks en Completer, que es lo que los diferencia haciendo que solamente el primero funcione. De esta forma se agregarán al conjunto S_4 los ítems:

$$(0, \{\}, C \to PA\bullet)$$
$$(1, \{\}, C \to PA\bullet)$$

Pero podemos comprobar que solamente para el primero de los ítems se verifica la propiedad fundamental que deseamos mantener. En efecto, para el segundo ítem la propiedad dice que debe existir una derivación que genere $xxyz$ pero utilice la regla $C \to PA$ teniendo el primer x ya generado. Esto es imposible, ya que solamente se puede generar y con una regla, que hace pop de q, lo cual obliga a haber utilizado la única regla que hace push de q antes, y eso implica que a partir del símbolo P se deben haber generado al menos dos x, con lo cual como solamente hay dos x en la cadena generada, no es posible que hubiera habido una x adicional presente antes de expandir inicialmente el símbolo C.

Podemos notar que este "entrecruzamiento" de la información de los multistacks produce ítems con valores inválidos para el índice de inicio del parsing. Sin embargo, en este ejemplo se puede comprobar que el resto de la información de los ítems se corresponde con la propiedad fundamental. Más precisamente, si cambiamos la propiedad fundamental para que, al igual que solamente pedimos que **exista** algún ξ_i que la haga cierta, baste con que **exista** algún i, índice inicial del parsing de la regla, tal que la propiedad sea cierta, podemos comprobar que los ítems generados en este ejemplo la verifican.

Este es un resultado que resulta valer en general para cualquier MGIG determinada a derecha (excepto para ciertos ítems específicos que se corresponden a una regla de tipo 1 lista para ser completada, y en las gramáticas no fuertemente determinadas a derecha, por la ambigüedad existente entre las distintas reglas tipo 2 a utilizar para completar, ya que son todas equivalentes), y que por lo tanto vuelve redundante el almacenado del índice de inicio del parsing en los ítems de Earley a la hora de procesar dichas gramáticas. De esta manera, lo que proponemos finalmente es modificar el algoritmo de Earley-MGIG original eliminando toda la información de inicio de parsing en cada ítem, es decir, reduciendo los ítems a únicamente el multistack actual, y la regla que se está parseando anotada con $\bullet$. En el paso Completer, lo que se hará será completar directamente con **la primera**[1] de las reglas de tipo 2 correspondientes al no terminal que se está completando (en caso de no haber ninguna, el Completer simplemente no agrega ningún ítem, pues entonces el no terminal involucrado es improductivo y no puede utilizarse nunca en una derivación exitosa, ya que es imposible eliminarlo utilizando únicamente reglas de tipo 1). Notar que en particular ya **no se buscará en los conjuntos anteriores**, con lo cual podemos guardar en memoria únicamente el conjunto de Earley actual (y el siguiente en construcción) reduciendo notablemente la memoria necesaria durante el cómputo.

Antes de enunciar la propiedad fundamental adecuada para este algoritmo específico para MGIGs determinadas a derecha, nos conviene definir las siguientes nociones:

Producción plausible Decimos que una regla de producción $A \underset{\mu}{\to} \gamma$ es *plausible* para

[1] Podemos tomar una cualquiera, pero fija.

un ítem $B \underset{\mu'}{\rightarrow} X \bullet Y$ si

- $A = B$
- $\gamma = \delta Y$
- Si $X = \alpha C \beta$ con $\beta \in T^*$, $C \in N$, entonces $\delta = \alpha' C \beta$
- Si $X \in T^*$ entonces $\delta = X$ y $\mu = \mu'$

Es decir, si tienen el mismo símbolo a la izquierda, y coinciden desde el último no terminal antes del $\bullet$, o bien las reglas coinciden por completo en caso de no existir dicho no terminal.

Ítem bueno Decimos que un ítem $(\xi, A \underset{\mu}{\rightarrow} X \bullet Y)$ es *bueno*, si $A \underset{\mu}{\rightarrow} XY$ es la **primera regla de tipo 2** para el último símbolo no terminal presente en X, o bien $X \in T^*$. En particular quedan excluidos los ítems correspondientes a una regla de tipo 1 con el $\bullet$ al final.

La idea detrás de la definición de ítem bueno, es que estos son los únicos ítems que puede agregar el paso Completer, ya que con la modificación realizada únicamente puede completar con la primera regla de tipo 2 de cada no terminal.

Ahora sí, la propiedad fundamental queda entonces modificada de la siguiente manera:

- ***Propiedad fundamental*** del algoritmo de Earley-MGIG (sin multistack inicial ni índice inicial en los ítems):

  Un ítem **bueno** $(\xi, A \underset{\mu}{\rightarrow} X \bullet Y)$ pertenece al conjunto S_j al terminar el algoritmo, si y solo si existe una derivación por izquierda de la forma:

  $$S' \overset{*}{\Rightarrow} \xi_i \alpha_0 \alpha_1 \cdots \alpha_{i-1} A \gamma \Rightarrow \mu'(\xi_i) \alpha_0 \alpha_1 \cdots \alpha_{i-1} \beta Y \gamma \overset{*}{\Rightarrow} \xi \alpha_0 \alpha_1 \cdots \alpha_{i-1} \alpha_i \alpha_{i+1} \cdots \alpha_{j-1} Y \gamma$$

  Para **alguna** multistack ξ_i, algún índice $i \leq j$, y alguna regla $A \underset{\mu'}{\rightarrow} \beta Y$ en la MGIG **plausible** para $A \underset{\mu}{\rightarrow} X \bullet Y$, donde hemos notado $\mu'(\xi_i)$ a la multistack que resulta de aplicar a ξ_i la operación de multistack indicada por μ' (potencialmente ninguna).

  Como siempre, se entiende que en el árbol de esta derivación el paso correspondiente a la regla $A \rightarrow \cdots$ cuya aplicación se muestra es el padre de todos los nodos correspondientes al Y de la última forma sentencial.

  Además, todos los ítems generados por el algoritmo son ítems buenos.

Teorema 5.1.1: Si se modifica Earley-MGIG de manera que utilice únicamente la multistack actual en los ítems (eliminando la verificación de coincidencia de multistacks en el paso Completer), y no se guarde el índice inicial del parsing de la regla en los mismos (con lo cual en Completer se completa únicamente **la primera** de las reglas de tipo 2 existentes para el no terminal completado, sin examinar los conjuntos anteriores), se sigue manteniendo la propiedad fundamental (modificada como se indicó) cuando se lo aplica sobre una MGIG determinada a derecha.

Demostración Consideremos una cierta MGIG determinada a derecha.

Veamos primero que si tenemos una derivación parcial para un ítem bueno de acuerdo a lo indicado por la propiedad fundamental en el caso de esta MGIG, entonces dicho ítem es generado en el correspondiente conjunto por el algoritmo. Lo veremos inductivamente: Sea T un natural arbitrario estrictamente mayor que la cantidad de no terminales en la MGIG, y estrictamente mayor que la longitud máxima de una regla de la misma. Realizaremos la demostración de este paso por inducción en $lT^2 + nT + p$, siendo l la longitud de la derivación parcial, n la cantidad de descendientes del no terminal a la izquierda de la regla usada en el ítem que se está considerando en el árbol de antecesores (5.1.3), y p la posición del $\bullet$ en la regla del ítem.

La única importancia de realizar la inducción en este valor particular es que por la elección de T, siempre $T^2 > n'T + p'$ y $T > p'$, con lo cual si inductivamente reducimos el caso bajo estudio a un caso:

- Con una derivación más corta.
- Con una derivación de idéntica longitud, pero con un ítem que tiene un no terminal a la izquierda con menos descendientes en el árbol de antecesores.
- Con una derivación de idéntica longitud y con un ítem con el mismo no terminal a la izquierda, pero con el $\bullet$ más a la izquierda.

entonces en cualquiera de estos casos el valor en el cual hacemos inducción se reduce, y entonces podemos realizar la demostración deseada, ya que en nuestra demostración siempre reduciremos el problema a casos que correspondan a una de estas 3 opciones[2].

Existen 3 posibilidades, según el tipo de símbolo a la izquierda del $\bullet$ en el ítem:

1. No hay nada antes del $\bullet$.

   El único ítem de este tipo que no será agregado por Predictor será $(\lambda, S' \to \bullet S)$, que por supuesto corresponde a la derivación parcial $S' \Rightarrow S$ y se encuentra presente como ítem inicial de partida al comenzar el algoritmo.

   Supongamos ahora que tenemos una derivación

   $$S' \overset{*}{\Rightarrow} \xi_i \alpha_0 \cdots \alpha_{i-1} A \gamma \Rightarrow \mu'(\xi_i) \alpha_0 \cdots \alpha_{i-1} \beta \gamma$$

   que se corresponde, llamando $\xi = \mu'(\xi_i)$, de acuerdo a lo pedido por la propiedad fundamental con el ítem bueno $(\xi, A \underset{\mu}{\to} \bullet\beta) \in S_i$ (que no sea el ítem anteriormente considerado, por lo cual $A \neq S'$). En este caso deberá ser $\mu' = \mu$ necesariamente por la definición de producción plausible. Notar que podemos obviar la última forma sentencial $\cdots \overset{*}{\Rightarrow} \xi \alpha_0 \cdots \alpha_{j-1} \beta \gamma$ presente en el enunciado de la propiedad fundamental, pues como β contiene la totalidad de los hijos del nodo correspondiente a $A \to \cdots$ en el árbol de parsing, deberá ser $i = j$ en este caso.

   En la derivación, debe crearse en algún paso el no terminal A mostrado, mediante alguna regla $X \underset{\mu''}{\to} \delta_1 A \delta_2$. Es decir:

[2] Alternativamente, para ahorrarnos definir T, en lugar de realizar una inducción en naturales podríamos considerar las triplas (l, n, p) ordenadas lexicográficamente, y realizar inducción directamente aprovechando que tal ordenamiento define un **buen orden** en el conjunto de **triplas de naturales**.

$$S' \overset{*}{\Rightarrow} \xi_k\alpha_0\cdots\alpha_{k-1}X\gamma' \Rightarrow \mu''(\xi_k)\alpha_0\cdots\alpha_{k-1}\delta_1 A\delta_2\gamma' \overset{*}{\Rightarrow} \xi_i\alpha_0\cdots\alpha_{i-1}A\gamma \Rightarrow \mu'(\xi_i)\alpha_0\cdots\alpha_{i-1}\beta\gamma$$

Con $\gamma = \delta_2\gamma'$. Si consideramos la parte de la derivación mostrada en la primera línea, esta tiene estrictamente menos pasos que la derivación completa y por lo tanto inductivamente podemos suponer que existe en S_i el ítem bueno correspondiente $(\xi_i, X \underset{\mu'''}{\to} \delta_1' \bullet A\delta_2) \in S_i$, para algún $X \underset{\mu'''}{\to} \delta_1' \bullet A\delta_2$ tal que $X \underset{\mu''}{\to} \delta_1 A\delta_2$ es una producción plausible para para tal ítem.

Como la aplicación de Predictor a tal ítem generará el ítem que buscábamos mediante la regla $A \underset{\mu}{\to} \beta$, queda probado que tal ítem será generado.

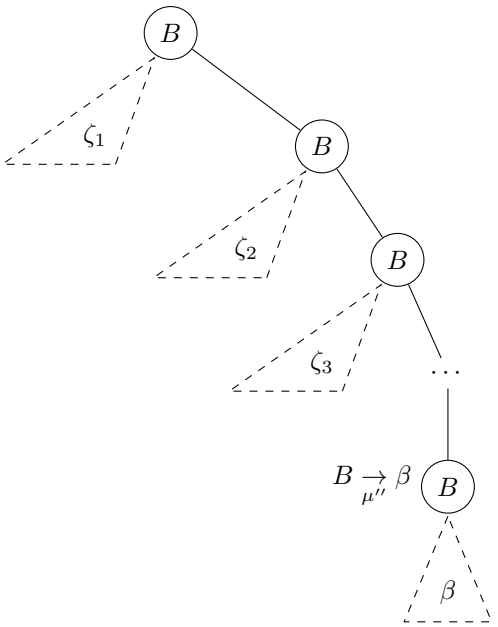
2. Hay un símbolo no terminal antes del $\bullet$.

   Supongamos que tenemos una derivación

$$S' \overset{*}{\Rightarrow} \xi_i\alpha_0\cdots\alpha_{i-1}A\gamma \Rightarrow \mu'(\xi_i)\alpha_0\cdots\alpha_{i-1}\delta BY\gamma \overset{*}{\Rightarrow} \xi\alpha_0\cdots\alpha_{j-1}Y\gamma$$

   Que corresponde a un ítem bueno $(\xi, A \underset{\mu}{\to} XB \bullet Y) \in S_j$, y en la cual se ha usado la regla $A \underset{\mu'}{\to} \delta BY$. Notar que por ser un ítem bueno, deberá ser en este caso $B \neq A$ pues estamos ante una regla de tipo 2 para B.

   Como el no terminal B que se ve en esta derivación es eliminado en la misma, debe utilizarse para ello una regla con B a la izquierda. Si esta regla es de tipo 1, al aplicarla se reemplazará B por ζB, con lo cual a su vez este último B debe ser eliminado en algún momento de la derivación. Repitiendo este argumento mientras la regla utilizada sea de tipo 1, eventualmente tenemos un no terminal B descendiente totalmente por derecha del B original en el árbol de la derivación considerada y que es eliminado mediante una regla que no es de tipo 1, digamos $B \underset{\mu''}{\to} \beta$.



Se observa entonces que esta misma derivación se corresponde de acuerdo a la propiedad fundamental con el ítem bueno $(\xi, B \underset{\mu'''}{\rightarrow} \beta'\bullet) \in S_j$ tal que $B \underset{\mu''}{\rightarrow} \beta$ es producción plausible para el mismo. Si demostramos entonces que este ítem es generado por el algoritmo, queda claro que el ítem original que nos interesa será generado a partir de este en el paso Completer.

Como se explicó en 5.1.3, A resulta ser el antecesor de B en el árbol de antecesores. Visto en el otro sentido, el nuevo no terminal B bajo estudio es un hijo de A en el árbol de antecesores, y por lo tanto tiene menos descendientes en el mismo. Por hipótesis de inducción, este ítem es generado por el algoritmo, ya que corresponde a una derivación de igual longitud pero utiliza un no terminal con menor cantidad de descendientes.

3. Hay un símbolo terminal $t \in T$ antes del $\bullet$.

   Si el ítem considerado es $(\xi, A \rightarrow XW \bullet Y) \in S_i$, con $W \in T^+$ y de manera que $X \in (N \cup T)^*$ **no** termine en un símbolo terminal, es inmediato ver que la misma derivación que corresponde a este ítem corresponde a $(\xi, A \rightarrow X \bullet WY) \in S_{i-|W|}$, con lo cual este caso se reduce inductivamente a otro con la misma derivación, el mismo símbolo no terminal a la izquierda de la regla en el ítem, y el $\bullet$ más a la izquierda (notar que si el primer ítem era bueno, el segundo también lo es).

   Una vez establecido que el segundo ítem es generado por el algoritmo, es evidente que también será generado el primero, pues este se obtiene de aquel luego de $|W|$ pasos de Scanner.

Falta ver ahora que la existencia de un ítem $(\xi, A \underset{\mu}{\rightarrow} X\bullet Y)$ en el conjunto S_j implica que este ítem es bueno, y la existencia de una derivación con la forma indicada por la propiedad. Podemos hacerlo inductivamente, asumiendo que todos los ítems ya agregados lo cumplen, y verificando que el ítem que se agrega en un paso lo verifica. Hay 4 posibilidades para cada ítem:

- Para el ítem inicial $(\lambda, S' \rightarrow \bullet S)$ (siendo λ multistack vacía) se verifica claramente la propiedad, con la derivación de un paso $S' \Rightarrow S$. El ítem es claramente bueno.
- Si un ítem fue agregado con Scanner, esto se hizo a partir de algún ítem $(\xi, A \underset{\mu}{\rightarrow} X \bullet xY)$ con $x \in T$, y es inmediato que la misma derivación que hace válida la propiedad para ese ítem, la hace válida para el actual. Notar que ambos ítems son necesariamente buenos.
- Si un ítem $(\xi, A \underset{\mu}{\rightarrow} \bullet\alpha) \in S_i$ fue agregado con Predictor, esto se hizo a partir de algún ítem $(\xi', B \underset{\mu'}{\rightarrow} X \bullet AY) \in S_i$ con $\mu(\xi') = \xi$, y es inmediato que agregando un paso $\Rightarrow \xi\alpha_0\alpha_1 \cdots \alpha_{i-1}\alpha Y\gamma$ al final de la derivación correspondiente al segundo ítem, tenemos una derivación correspondiente al primero. Notar que ambos ítems son necesariamente buenos.
- El caso interesante ocurre al agregar un ítem mediante el paso Completer. Aquí es donde al intentar extender de manera natural por un paso la derivación correspondiente al ítem que se usa para completar, tendremos que solamente es posible realizar

dicha extensión cuando los multistacks correspondientes coinciden, lo cual ya no se verifica en el algoritmo al no guardar multistacks iniciales. Veremos que la propiedad sigue valiendo gracias a que la gramática es determinada a derecha.

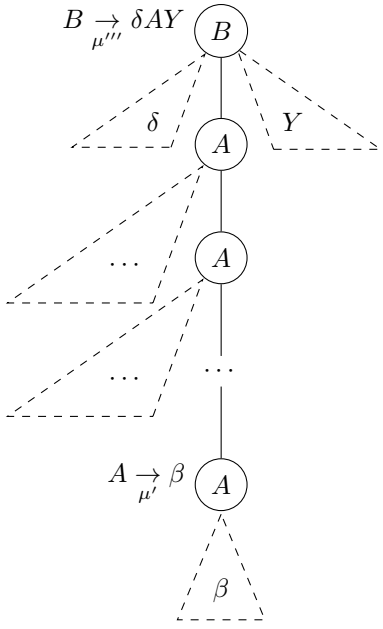
Sea $(\xi, A \underset{\mu}{\to} \alpha\bullet) \in S_j$ el ítem bueno que se está procesando con Completer.

Tenemos que ver que para el nuevo ítem $(\xi, B \underset{\mu''}{\to} XA \bullet Y) \in S_j$ (donde $B \underset{\mu''}{\to} XAY$ es la primera regla de tipo 2 para A) existe una derivación adecuada, suponiendo que la hay para el anterior. Notar que este ítem es bueno por ser una regla de tipo 2 para A, que es el símbolo anterior al $\bullet$, ya que debería ser tipo 1 para tal no terminal en un ítem que no fuese bueno.

Sabemos por la propiedad fundamental por el ítem que se está procesando, que existe una derivación:

$$S' \overset{(\star)}{\overset{*}{\Rightarrow}} \xi_i\alpha_0\alpha_1\cdots\alpha_{i-1}A\gamma \Rightarrow \mu'(\xi_i)\alpha_0\alpha_1\cdots\alpha_{i-1}\beta\gamma \overset{*}{\Rightarrow} \xi\alpha_0\alpha_1\cdots\alpha_{i-1}\alpha_i\alpha_{i+1}\cdots\alpha_{j-1}\gamma$$

Con $A \underset{\mu'}{\to} \beta$ una producción plausible para $A \underset{\mu}{\to} \alpha\bullet$. Ahora bien, si consideramos la primera parte de la derivación, indicada con $(\star)$, como al final de esta parte aparece el no terminal A en la forma sentencial, necesariamente se debe haber utilizado en este tramo alguna regla de producción que contenga a A en el lado derecho. Más aún, consideremos en el árbol de parsing de la derivación, los pasos correspondientes a ancestros del paso $A \to \cdots$ mostrado. Como la raíz del árbol corresponde a $S' \to \cdots$, existe un paso más cercano al nodo de A cuya regla no tiene a A del lado izquierdo. Es claro que en dicho paso se debe haber aplicado una regla de tipo 2 para A. Pero al ser una MGIG determinada a derecha, tal regla debe ser necesariamente de la forma $B \underset{\mu'''}{\to} \delta AY$, porque tanto el no terminal izquierdo B y la terminación AY son únicos para todas las reglas de tipo 2 para A, y en particular debe coincidir con el ítem que estamos agregando con Completer.



Resumiendo lo dicho, sabemos entonces que la derivación mostrada tiene de hecho la siguiente forma:

$$S' \stackrel{*}{\Rightarrow} \xi_k\alpha_0\alpha_1\cdots\alpha_{k-1}B\gamma' \Rightarrow \mu'''(\xi_k)\alpha_0\alpha_1\cdots\alpha_{k-1}\delta AY\gamma' \stackrel{*}{\Rightarrow}$$
$$\stackrel{*}{\Rightarrow} \xi_i\alpha_0\alpha_1\cdots\alpha_{i-1}AY\gamma' \Rightarrow \mu'(\xi_i)\alpha_0\alpha_1\cdots\alpha_{i-1}\beta Y\gamma' \stackrel{*}{\Rightarrow} \xi\alpha_0\alpha_1\cdots\alpha_{i-1}\alpha_i\alpha_{i+1}\cdots\alpha_{j-1}Y\gamma'$$

Pero esta derivación tiene justamente la forma que la propiedad fundamental exige para el ítem que agrega el completer, pues $B \underset{\mu'''}{\rightarrow} \delta AY$ es producción plausible para dicho ítem.

∎

Corolario 5.1.2: Earley-MGIG con una única multistack y sin índice de inicio del parsing en los ítems, resulta correcto como reconocedor al aplicarlo a una MGIG determinada a derecha. Esto es inmediato de la propiedad fundamental, ya que el reconocimiento depende exclusivamente de que el ítem correspondiente a $S' \rightarrow S\bullet$ con multistack vacía pertenezca al conjunto n-ésimo al terminar el algoritmo (y este es un ítem bueno). Es claro que la cadena de entrada debe ser reconocida si y solo si existe una derivación que cumple lo que indica la propiedad fundamental para este ítem.

Corolario 5.1.3: Earley-MGIG con una única multistack resulta correcto como reconocedor al aplicarlo a una MGIG fuertemente determinada a derecha. Más aún, en este caso podemos precisar que deberá ser $\beta = X$ y $\mu = \mu'$ en la propiedad fundamental indicada, pues existe una única regla de tipo 2 para cada no terminal. Esto hace que la propiedad fundamental sea análoga a las anteriores en dicho caso, sin necesidad de utilizar la noción de producción plausible con su consiguiente ambigüedad en cuanto a la regla concreta utilizada en la derivación.

5.2. Clasificación de las gramáticas de SAT

Todas las gramáticas de SAT propuestas anteriormente (TAS, DTAS, RTAS, SRTAS) son MGIG determinadas a derecha. Además, si se utilizan las versiones "V-" de las mismas, se obtienen gramáticas MGIG fuertemente determinadas a derecha. Esto puede verificarse inmediatamente a partir de la definición de las mismas.

Además, es claro que la versión de MGIG "regular" dada para SAT **no** es determinada a derecha: Por ejemplo las reglas $A_1 \rightarrow A_2$ y $B_1 \rightarrow A_2$ son un claro ejemplo de dos reglas tipo 2 para A_2 con distinto símbolo no terminal a la izquierda. Pese a ello, podría utilizarse esencialmente nuestro algoritmo para dicha gramática (y cualquier otra MGIG con forma regular), ya que todas estas consideraciones solamente afectan al paso completer, pero en una gramática con forma regular, cuando se realiza un paso de completer necesariamente se termina de procesar toda la cadena de entrada, con lo cual en estas gramáticas regulares es posible completar directamente con un ítem para $S' \rightarrow S\bullet$ (con la multistack actual) y obtener un reconocedor correcto.

5.3. Relación entre Earley-MGIG con un solo multistack y un algoritmo clásico de programación dinámica para SAT

Teniendo en cuenta que el algoritmo de Earley puede ser visto como un algoritmo de programación dinámica, y que como resultado de nuestro modelado, tenemos planteado a SAT como un problema de parsing que puede resolverse con una extensión del algoritmo de Earley, es interesante preguntarse si existirá alguna relación entre el algoritmo obtenido de esta forma, y algoritmos conocidos para SAT basados en la técnica de programación dinámica.

La respuesta es afirmativa, y podemos encontrar dicha conexión fácilmente si nos centramos en los subproblemas planteados por el algoritmo (al pensarlo como algoritmo de programación dinámica): En el algoritmo de Earley, los subproblemas asociados corresponden a cada posible ítem, en cada posible conjunto, y la respuesta a cada subproblema es la respuesta a la pregunta booleana "¿Debe existir este ítem, de acuerdo a la propiedad fundamental?". El algoritmo en lugar de computar recursivamente la presencia de los ítems, realiza un procesamiento bottom-up mediante el cual únicamente se construyen los subproblemas con respuesta afirmativa (es decir, ítems que existirán efectivamente en los conjuntos de Earley). Pero con este esquema, es posible verlo como un algoritmo de programación dinámica con la misma complejidad de peor caso que el algoritmo de Earley usual.

Estas mismas consideraciones se aplican a Earley-MGIG con los ítems aumentados y la propiedad fundamental correspondiente. En este caso, cuando consideramos los ítems reducidos de la versión que acabamos de presentar, un subproblema queda de la forma:

"¿Es posible lograr una derivación parcial que termine con multistack ξ, habiendo leído los primeros n símbolos de la cadena de entrada, y de forma tal que se está parseando la regla $A \to \alpha \bullet \beta$ (en el sentido de la propiedad fundamental)?"

Si consideramos esto en el caso de la gramática TAS, vemos que si nos concentramos en las reglas de tipo 2, conocer la regla que se está parseando en un ítem tiene únicamente el efecto importante de indicar qué número de variable se debe considerar a continuación. Similarmente, a los efectos de resolver el problema SAT, la multistack actual contiene información sobre cláusulas pendientes de satisfacer con la asignación parcial realizada.

Luego la "información importante" del ítem, considerando el contexto particular del problema SAT, queda reducida al número de conjunto que lo contiene (que corresponde a la cantidad de símbolos de entrada ya leídos), la siguiente variable a procesar, y el conjunto de cláusulas (sufijos de las cláusulas de la entrada) pendientes de satisfacer con la asignación parcial de valores de verdad a las variables ya realizada. Además, las dos primeras de estas están relacionadas, ya que por ejemplo no tiene sentido haber leído solamente terminales correspondientes a cláusulas que comienzan con la primera variable, pero tener que procesar a continuación la quinta variable, existiendo en la entrada cláusulas que comienzan por ejemplo con la segunda variable[3].

En definitiva, los puntos de elección importantes son justo al comenzar el procesamiento de una variable, cuando se elige su valor de verdad, y eso determina por completo el **conjunto** de cláusulas pendientes que quedarán luego de procesar toda esa variable.

Por todo esto, la información fundamental e imposible de simplificar en un ítem es el conjunto de cláusulas pendientes de ser satisfechas, y el número de variable que se debe

[3] Esto es considerado por nuestra implementación: Mediante el uso de los filtros explicados en A.3.4, el algoritmo de parsing se acerca aún más a este algoritmo de programación dinámica

asignar a continuación. Tomando directamente estos parámetros como definición de un subproblema, asumiendo n variables numeradas desde 0 hasta $n-1$, podemos definir la función booleana:

$$f(k, C) = \begin{array}{l}\text{“Es posible realizar una asignación booleana a las variables [k, n),} \\ \text{que satisfaga las cláusulas de la entrada que comienzan con variables en [k,n),} \\ \text{así como también todas las cláusulas de C”}\end{array}$$

Para $0 \leq k \leq n$, y C un conjunto de cláusulas pendientes que contiene únicamente las variables $[k, n)$. La subestructura recursiva de f permite un conocido algoritmo de programación dinámica para SAT: Una elección para la variable k induce dos posibles conjuntos de pendientes, $a(C)$ y $b(C)$ (alguno de ellos podría ser inválido, si la correspondiente asignación de variables produce inmediatamente una inconsistencia, dejando la cláusula vacía por satisfacer), según el valor de verdad que se elija, y que son de hecho los que en el algoritmo de parsing resultan de los push, pop y pop-push que se hacen sobre C.

Luego $f(k, C) = f(k+1, a(C)) \vee f(k+1, b(C))$ (si $a(C)$ es inconsistente se toma directamente $f(k+1, a(C)) = False$, y análogamente con $b(C)$). El caso base ocurre para $f(n, C) = (C = \emptyset)$. La respuesta a la instancia de SAT es $f(0, \emptyset)$.

Este algoritmo es mucho mejor que el backtracking usual cuando solamente se evalúan pocos valores distintos de C, ya que entonces las distintas elecciones parciales de variables que produzcan idénticos C “se unifican” y son procesadas todas juntas una única vez, de manera similar a lo que ocurre en el algoritmo de parsing con las derivaciones distintas que llevan a un mismo ítem. Pero en peor caso, en todos los algoritmos puede ser necesario considerar explícitamente las 2^n elecciones de variables.

Notemos que como los ítems guardan más información en nuestro enfoque mediante parsing (ya que pueden producirse diferentes multistacks para un mismo conjunto de cláusulas, dependiendo del orden y forma en que se hacen los push y pop) que los subproblemas del algoritmo de programación dinámica, no es esperable que se pueda obtener una mejora de performance para el caso de Earley-MGIG alterado únicamente con la modificación para gramáticas determinadas a derecha por sobre este algoritmo de programación dinámica. No obstante, la forma de unificar multistacks que serían procesadas de idéntica manera que hemos introducido en 4.3.2 por analogía con los GLR-parsers es novedosa, y en casos en los que permitan descartar rápidamente muchas multistacks que han sido unificadas, existe a priori potencial de mejorar la performance para SAT (notar que al unificar únicamente por lista de topes de stacks, muchas multistacks que potencialmente representan a muchos conjuntos de cláusulas diferentes son almacenadas en el mismo nodo, y podrían ser por ejemplo descartadas todas a la vez si ese nodo no lleva a derivaciones exitosas).

6. PRUEBAS REALIZADAS

A la hora de probar el algoritmo, existen diferentes parámetros que pueden variarse. Identificamos en particular los siguientes:

- Clases de instancias de SAT sobre las que se ejecutará el algoritmo. En nuestro trabajo utilizamos las siguientes tres familias de instancias:
  - *random*: 10 instancias aleatorias con 20 variables y 91 cláusulas, elegidas arbitrariamente de la familia `uf20-91` de SATLIB [25], "Uniform Random-3-SAT, phase transition region, unforced filtered".
  - *factoring*: 5 instancias que codifican problemas de factorización entera de los números 4 hasta 8 inclusive. Obtuvimos estas instancias de [55].
  - *pigeonhole*: 5 instancias que codifican una aplicación del principio del palomar. Estas instancias difíciles de SAT ya fueron consideradas por ejemplo en trabajos como [14, 44].
- Algoritmo de reconocimiento de MGIG utilizado:
  - `GeneralEarleyMGIGRecognizer` (Algoritmo Earley-MGIG, para reconocer MGIG arbitraria)
  - `EarleyRDMGIGRecognizer` (Algoritmo Earley-MGIG específico para gramáticas determinadas a derecha)
  - `UnifyingEarleyRDMGIGRecognizer` (Variante del anterior, con la técnica de unificación explicada en 4.3.2)
- Gramática de SAT utilizada:
  - TAS
  - DTAS
  - SRTAS

  No hemos hecho pruebas con las versiones "V-" de las gramáticas, cuyas diferencias o posible conveniencia con respecto a las gramáticas originales es principalmente teórica.

  Tampoco utilizamos la gramática RTAS, por ser extremadamente similar a la SRTAS, siendo esta última más completa y homogénea.
- Utilización de filtros: se pueden utilizar o no los filtros de ítems durante el algoritmo.
- Criterio de ordenamiento de variables y cláusulas utilizado:

  Para las cláusulas, fijamos siempre el ordenamiento lexicográfico (entendiendo al entero que identifica a una variable como un único símbolo atómico a efectos de este ordenamiento). Ese ordenamiento ya se utiliza en la misma definición del lenguaje, pues las cláusulas van ordenadas por el número de su primera variable y (en el caso de DTAS, RTAS y SRTAS) también por el valor `a` o `b` de dicho primer literal de la

cláusula. Sin embargo tal ordenamiento impuesto por el lenguaje únicamente predica sobre el primer literal de cada cláusula. Para ordenar las cláusulas por completo, las ordenamos siempre con el criterio lexicográfico completo, desempatando las cláusulas que coinciden en el primer literal por los literales que siguen, con idéntico criterio.

Por otro lado para las variables, que deben ser ordenadas para identificar cada una con un número de variable entre 1 y n, consideramos 3 ordenamientos posibles, todos basados en heurísticas sencillas sobre frecuencias de aparición de variables y literales, y ya utilizados por ejemplo en [14]:

- **Max**: Se ordenan las variables por frecuencia de aparición. Se numeran con números más bajos aquellas variables que aparecen el mayor número de veces en la fórmula. Para desempatar, se ubican primero aquellas variables que **maximicen** la diferencia entre la cantidad de veces que aparecen afirmadas y negadas.
- **Min**: Similar a la anterior, pero el criterio desempate es ahora colocar primero aquellas variables que **minimicen** la diferencia entre la cantidad de veces que aparecen afirmadas y negadas.
- **Johnson**: Este ordenamiento está basado en la heurística de Johnson [24] para el problema Max-SAT. En dicha heurística, en cada paso se elige el literal más frecuente, y se asigna el valor de verdad de la variable asociada de forma tal que el literal se satisfaga. Se eliminan entonces las cláusulas que contienen al literal, por estar ya satisfechas, y se elimina el literal inverso de todas las cláusulas donde aparezca, obteniendo así un nuevo problema con una variable menos. Se repite este procedimiento hasta que ya no queden cláusulas no vacías.

  El ordenamiento propuesto es entonces numerar las variables en el orden en que las considera la heurística de Johnson. Además, como podrían eventualmente quedar algunas variables sin utilizar (porque desaparecen por completo todas las cláusulas que la contienen, en algún paso de este procedimiento, antes de ser asignadas), para completar la numeración colocamos todas estas variables al final, ordenadas por frecuencia de aparición (sobre la fórmula original) de manera decreciente.

Además, por cada uno de los ordenamientos anteriores definimos su ordenamiento contrario, que se obtiene invirtiendo la lista ordenada por el correspondiente criterio. Por ejemplo, definimos *anti-johnson* como el criterio de ordenamiento inverso al *johnson* anteriormente definido (si $ABCD$ es una lista ordenada por el criterio *crit*, entonces $DCBA$ es una lista ordenada por *anti-crit*).

Los resultados completos de las mediciones de tiempos realizadas se pueden consultar en el apéndice B. Presentamos a continuación un resumen de los resultados obtenidos.

En primer lugar, nos propusimos verificar la diferencia de performance con y sin filtros, para las distintas gramáticas. Sobre las instancias de interés mencionadas, los resultados sin filtros resultan en todos los casos utilizar tanto tiempo y memoria, que no pudieron ser medidos en el hardware utilizado. A continuación, utilizamos filtros en todas nuestras mediciones.

Lo siguiente que observamos es una importante y consistente diferencia de tiempos entre los 3 ordenamientos propuestos, y sus versiones inversas: Con los 3 algoritmos, para los

3 criterios de ordenamiento con la gramática SRTAS sobre instancias *random*, siempre se observa un factor de aproximadamente 7 entre el tiempo de ejecución con el ordenamiento propuesto y con su versión inversa. Es decir, *max* resulta sobre nuestros tests aproximadamente 7 veces más rápido que *anti-max*, y exactamente lo mismo ocurre con *min* y *johnson*.

Resumimos a continuación los tiempos de ejecución medios y el desvío estándar **en segundos**, sobre instancias *random*, para distintas combinaciones de gramática, criterio de ordenamiento y algoritmo.

- Gramática TAS:

	johnson	*max*	*min*
General MGIG	184 ± 58	312 ± 161	236 ± 110
RD-MGIG	67 ± 12	86 ± 33	70 ± 33
Unif-RD-MGIG	57 ± 25	88 ± 34	73 ± 37

- Gramática DTAS:

	johnson	*max*	*min*
General MGIG	$0{,}99 \pm 0{,}32$	$0{,}93 \pm 0{,}28$	$1{,}04 \pm 0{,}30$
RD-MGIG	$0{,}59 \pm 0{,}19$	$0{,}66 \pm 0{,}43$	$0{,}67 \pm 0{,}42$
Unif-RD-MGIG	$0{,}78 \pm 0{,}25$	$0{,}98 \pm 0{,}40$	$0{,}81 \pm 0{,}28$

- Gramática SRTAS:

	johnson	*max*	*min*
General MGIG	$0{,}71 \pm 0{,}22$	$0{,}72 \pm 0{,}18$	$0{,}71 \pm 0{,}25$
RD-MGIG	$0{,}52 \pm 0{,}20$	$0{,}48 \pm 0{,}15$	$0{,}50 \pm 0{,}18$
Unif-RD-MGIG	$0{,}69 \pm 0{,}21$	$0{,}69 \pm 0{,}21$	$0{,}71 \pm 0{,}25$

Se desprende de estas mediciones que, para cualquiera de las 9 posibles elecciones de los parámetros, con DTAS y SRTAS el reconocimiento es muchísimo más rápido que con TAS, y además con SRTAS es más rápido que con DTAS.

Finalmente, con esta observación medimos el tiempo de SRTAS para las dos familias de instancias restantes, lo cual se puede observar en las siguientes tablas (en el caso de *pigeonhole*, se indica directamente el tiempo de ejecución sobre el caso más grande, en lugar de un valor medio y desvío estándar, ya que dicho caso insume un tiempo mucho mayor que los demás):

- Instancias *factoring*:

	johnson	*max*	*min*
General MGIG	$1{,}77 \pm 1{,}01$	$1{,}73 \pm 0{,}99$	$1{,}78 \pm 1{,}02$
RD-MGIG	$1{,}31 \pm 0{,}74$	$1{,}34 \pm 0{,}78$	$1{,}30 \pm 0{,}74$
Unif-RD-MGIG	$1{,}34 \pm 0{,}75$	$1{,}15 \pm 0{,}63$	$1{,}45 \pm 0{,}81$

- Instancias *pigeonhole*:

	johnson	*max*	*min*
General MGIG	33,37	33,39	33,60
RD-MGIG	25,96	25,42	25,69
Unif-RD-MGIG	6,86	6,78	6,75

7. CONCLUSIÓN

Como resultado de lo observado, concluimos que la utilización directa del algoritmo de parsing de las MGIG presentadas como SAT-solver no parece ser un camino fructífero para avanzar el estado del arte en cuanto a SAT, ya que incluso teniendo en cuenta que nuestra implementación prototipo en python es una implementación lenta, los tiempos de ejecución obtenidos son excesivamente altos, producto fundamentalmente del enorme tamaño de los conjuntos de Earley que pueden ser generados por el algoritmo.

Es interesante no obstante desde el punto de vista teórico el poder expresivo de las MGIG, que permiten con gran naturalidad expresar el lenguaje SAT manteniendo un problema de reconocimiento general NP-completo, es decir, se expresa SAT naturalmente mediante un enfoque basado en gramáticas que extienden las CFG, sin salirse de NP. Nuestro principal aporte desde este punto de vista teórico es la definición y el estudio de la complejidad del problema de reconocimiento de MGIG, así como el desarrollo de un algoritmo general para su reconocimiento, y la identificación de una subclase particular, las MGIG determinadas a derecha, para las cuales es posible desarrollar un algoritmo de reconocimiento aún más simple y eficiente que en el caso general.

De las mediciones de tiempos del algoritmo, se puede destacar inmediatamente la importancia de utilizar filtros para reducir la cantidad de ítems generados en una derivación. La enorme diferencia obtenida entre utilizarlos o no, no es sorprendente si se tiene en cuenta que esta es esencialmente la misma diferencia que existe entre un algoritmo de backtracking y un algoritmo de fuerza bruta para un mismo problema.

Otro resultado que podemos verificar claramente de las mediciones es el hecho de que el ordenamiento de variables utilizado juega un papel importante en el tiempo de ejecución final. Si bien entre las tres estrategias de frecuencia particulares estudiadas no se observan finalmente diferencias importantes, sí es muy notoria la diferencia entre cualquiera de ellas y cualquiera de los correspondientes ordenamientos inversos. Se podría continuar el trabajo en esta dirección explorando más criterios de ordenamiento.

También se desprende de las mediciones que reducir el no determinismo en la gramática utilizada, es decir, utilizar gramáticas con menos derivaciones posibles para una misma cadena, ayuda a reducir notablemente el tiempo de ejecución. En nuestras mediciones se observa un claro ejemplo de esto al comparar la gramática TAS, que permite intercalar libremente las 3 etapas de pop-push, push y pop durante el procesamiento de una variable particular, con las DTAS, RTAS y SRTAS, que fijan un orden particular para dichas operaciones, obteniendo así tiempos de ejecución mucho menores.

Finalmente, podemos destacar que en las instancias difíciles evaluadas, los tiempos de ejecución de los algoritmos quedan muy claramente ordenados de forma tal que

$$t_{\text{GeneralMGIG}} > t_{\text{RD-MGIG}} > t_{\text{Unif-RD-MGIG}}$$

Si bien la tendencia $t_{\text{GeneralMGIG}} > t_{\text{RD-MGIG}}$ se observa en general en todos los casos, y es de esperarse ya que RD-MGIG no es más que una versión estrictamente recortada en cuanto a estructuras y funcionamiento de GeneralMGIG, no ocurre así con Unif-RD-MGIG, ya que esta implementación introduce mucha complejidad de estructuras y procesamiento adicionales. Esto explica por qué en general en los otros casos, nuestra implementación de RD-MGIG con unificación por lista de topes de multistack resulta más

lenta que RD-MGIG. Podemos destacar no obstante que en los casos evaluados en los que la unificación produce peores tiempos, estos son en el peor de los casos aproximadamente un 50 % peores que los tiempos de RD-MGIG, mientras que para las instancias difíciles de pigeonhole obtenemos tiempos casi 4 veces más rápidos. La conclusión que podemos extraer de esto es que la idea general de unificación, inspirada en las ideas planteadas en [46] para GLR parsing, puede aplicarse más en general a otros formalismos de gramáticas y autómatas, como son las MGIG en nuestro caso, para reducir el procesamiento total necesario, con buenos resultados. Se podría continuar futuras investigaciones considerando mejores maneras de implementar la idea general de unificación de multistacks, ya sea mejorando las estructuras de datos utilizadas para implementar nuestro mecanismo de unificación, o bien planteando un mecanismo alternativo de unificación de multistacks.

Con la intención de seguir investigando modelos basados directamente en gramáticas para el parsing de SAT, el autor estudia en [13] modificaciones a la definición formal de MGIG que permitan encarar el parsing de las gramáticas mediante un enfoque basado en autómatas con pilas más tradicionales, esperando obtener buenos resultados como los que se han obtenido en [14].

La diferencia teórica entre el enfoque mediante la verificación de existencia de un elemento en la intersección de autómatas / expresiones regulares (PSPACE completo) y mediante parsing de MGIG (NP completo) no se traduce en una diferencia práctica en el caso de SAT, ya que se obtienen hasta el momento mejores resultados con el enfoque basado en intersección de autómatas. El problema esencial encontrado es que los conjuntos de Earley durante el parsing de la gramática resultan ser demasiado grandes, y el enfoque que utilizamos no puede evitar generarlos y explorarlos por completo, resultando en algoritmos ineficientes para el caso de SAT, más allá de las mejoras que se pueden obtener con técnicas de unificación, que podrían ser útiles por ejemplo para el parsing de MGIG generales.

Otra posible línea de investigación pendiente es el desarrollo de un algoritmo determinístico para MGIG, con técnicas tradicionales de autómatas, y que por lo tanto no genere más que un árbol de derivación posible para una cadena dada. Si bien este enfoque no aplica para el caso general de una MGIG cualquiera, y a priori no sirve entonces por ejemplo para el modelado de SAT, su estudio podría ser interesante para utilización en compiladores, donde es habitual desambiguar gramáticas formales ambiguas introduciendo reglas de precedencia en los autómatas, para seleccionar una única alternativa de entre todas las posibles en un cierto paso.

Apéndice

A. SOBRE LA IMPLEMENTACIÓN REALIZADA EN ESTE TRABAJO

A.1. Estructura general del proyecto

Todo el código fuente asociado a este trabajo se encuentra organizado en un proyecto python bajo el directorio raíz `src`.

En este podemos identificar los siguientes subdirectorios:

- `mgig`: Este es el directorio más importante. Contiene todo lo directamente relacionado con nuestra implementación de algoritmos de reconocimiento de MGIG.
  - `cnfs`: Contiene archivos con instancias de SAT, en formato DIMACS-CNF, utilizadas para testing.
  - `grammars`: Contiene archivos con descripciones de gramáticas MGIG, en el formato explicado en A.5
  - `graphs`: Aquí hemos guardado imágenes de stacks y multistacks, generadas como información de debug en corridas del algoritmo durante nuestro testing (ver A.7.2).
  - `msgraph.py`: Módulo python para dibujar stacks.
  - `grammar.py`: Módulo python para cargar archivos de gramáticas MGIG.
  - `mgig.py`: Módulo python con las implementaciones de los reconocedores de MGIG basados en el algoritmo de Earley.
  - `tas.py`: Módulo con lexers, filtros y funciones auxiliares específicas para las gramáticas de TAS.
  - `run_examples.py`: Módulo utilizado para ejecutar todos los ejemplos y visualizar tiempos y resultados.
  - `examples.json`: Archivo de configuración para indicar los ejemplos que deberán ser ejecutados.
- `pygraph`: Código fuente de la librería python-graph, utilizada para para la visualización de stacks explicada en A.7.2
- `spark`: Código fuente de la librería SPARK [3], del cual hemos extraído y utilizado el lexer.
- `utils`: Funciones utilitarias no directamente relacionadas con el reconocimiento de MGIG.
  - `instanceGeneration.py`: Usado durante el testing, genera instancias de SAT aleatorias / exhaustivamente, del tamaño deseado.
  - `satsolver.py`: Contiene dos implementaciones de satsolvers: Una mediante un backtracking directo, y la otra mediante el algoritmo de programación dinámica mencionado en 5.3

A.2. Análisis léxico

Para el análisis léxico se utiliza el módulo correspondiente de la implementación de SPARK[3], en lenguaje Python.

Antes de ejecutar el algoritmo de reconocimiento de Earley (modificado para MGIG), la entrada se separa en tokens de acuerdo a regexes, que se especifican en una clase python que herede de la clase base `GenericScanner` de SPARK.

Para el caso del TAS, la implementación separa la entrada en tokens mediante regexes sencillas que separan por espacios, quedando así dividida en tokens como $3a4b$ y similares, correspondientes a cada cláusula.

Las reglas de las gramáticas también se dan usando regexes: Si bien hemos el formalismo de gramáticas de la manera más simple posible, de forma tal que no hay expresiones regulares en las reglas sino directamente índices y símbolos terminales específicos; en la implementación práctica y en la definición del TAS, permitimos el uso de expresiones regulares, lo que permite simplificar la descripción de la gramática.

De esta forma, en nuestra implementación todo símbolo terminal es representado como un objeto de la clase `EarleyTerminal`, y tiene asociada una expresión regular particular. Esta expresión regular especifica qué tokens (obtenidos del análisis léxico de la entrada) son aceptados en la definición de este terminal. De esta forma, es posible codificar reglas como $B_1 \underset{\mu}{\rightarrow} 1aw^*$, donde a la derecha de la regla se utiliza un único objeto terminal, con regex asociada `1a.*`. Además, estas regex pueden contener un grupo para capturar y utilizar en el push como índice. Así por ejemplo en una regla como $B_1 \underset{w^*}{\rightarrow} 1aw^*$, la regex correspondiente al terminal del lado derecho sería `1a(.*)`, en la cual se captura el sufijo que se pushea en este caso.

Los índices almacenados en stacks son, en nuestra implementación, cadenas de texto de python. Por otro lado, los índices que se utilizan en las reglas de la gramática de la forma pop y pop-push son objetos de clase `EarleyTerminal`, cuya regex indica las cadenas que pueden ser popeadas mediante esta regla. Además, para el caso de pop-push, la regex deberá contener un grupo para capturar que indique el sufijo del índice que se deberá pushear luego del pop. Para el caso de las reglas de tipo push no se utiliza un objeto `EarleyTerminal` con una regex, sino que se especifica directamente una cadena de texto (índice) a pushear, con la salvedad de que se puede utilizar la secuencia de escape `{0}`, la cual se reemplazará por el grupo capturado por la regex correspondiente al objeto `EarleyTerminal` del lado derecho de la regla utilizada. En este sentido, una regla de push puede codificarse por ejemplo como $B_1 \underset{\texttt{\{0\}}}{\rightarrow} \texttt{1a(.*)}$

A.3. Implementación de Earley-MGIG

A.3.1. Tipos de updates de multistack

Los updates de multistack son desencadenados por el procesamiento de reglas de tipo push, pop o pop-push. Una regla de producción puede ser en nuestra implementación de uno de dos tipos: `updateOnScanner` o `updateOnPredictor`. Como ya se ha mencionado anteriormente al exponer Earley-MGIG, el paso más natural para realizar modificaciones a los multistacks de los ítems es el paso predictor. Sin embargo, al estudiar el análisis léxico realizado en nuestra implementación, hemos mostrado cómo el índice que será pusheado **depende del token de entrada** que se lea a continuación. Por este motivo, decidimos

que las reglas con operación de pila que tengan algún símbolo terminal del lado derecho (y que por lo tanto, lo tendrán como primer símbolo, y como único símbolo terminal) sean de tipo `updateOnScanner`, de forma que la modificación al multistack de los ítems se produce en el paso Scanner, en el cuál normalmente se lee la entrada y es posible conocer entonces el siguiente token. Todas las demás reglas de la gramática serán de tipo `updateOnPredictor`.

Notar que no es razonable hacer que todas las reglas sean de tipo `updateOnScanner`, pues este paso del algoritmo solamente se ejecuta al encontrarse con un terminal, y no todas las reglas que implican operaciones de pila contienen un terminal del lado derecho. Una alternativa posible sería que todas las reglas sean de tipo `updateOnPredictor`, pero que entonces en el paso predictor se verifique el siguiente token de la entrada para saber cuál es la modificación que debe realizarse al multistack. En nuestra implementación optamos por clasificar las reglas en estos dos tipos, para mantener la propiedad de que únicamente en el paso Scanner se examinan los tokens de la entrada.

A.3.2. Limitaciones

Nuestra implementación tiene algunas restricciones con respecto a las versiones más generales de los algoritmos que explicamos anteriormente:

- No consideramos la posibilidad de reglas de producción de tipo *epsilon con restricciones*. Notar que estas nunca son utilizadas en nuestras gramáticas para SAT. De cualquier manera, la modificación para permitirlas sería muy simple, pues su tratamiento es idéntico a las reglas epsilon, pero al aplicarlas en predictor solamente se debe generar el nuevo ítem si la multistack de partida satisface la restricción indicada.

- Asumimos que no hay ciclos de referencias en el algoritmo UnifyingEarleyRDMGIGRecognizer (A.4.3). Como se explicó en 4.3.2.6 , para evitar este problema y permitir gramáticas que generen ciclos de referencias, se debería evitar la propagación hacia atrás durante un pop a descripciones de ítems ya visitadas (nuestra implementación podría no terminar en este paso, entrando en una recursión infinita). Notar que estos ciclos jamás se producen en las gramáticas para SAT.

A.3.3. $\lambda-$watchlist

Un problema conocido a la hora de implementar el algoritmo de Earley ocurre en el paso Completer cuando se permiten producciones con λ en el lado derecho. En este caso, es posible que se procese mediante Completer un ítem cuyo índice de inicio sea el mismo que la posición actual en la cadena de entrada, es decir, que los ítems para completar deban buscarse en el mismo conjunto de Earley que se está construyendo en el paso actual del algoritmo. Esta referencia a un conjunto aún incompleto podría causar que algunos ítems que deberían generarse no se generen, por no estar presente aún en el conjunto el ítem que se completaría para generarlos.

Existen diversas soluciones posibles a este problema. Una de las primeras propuestas es repetir el procesamiento mediante Completer de todos los ítems correspondientes del conjunto actual, mientras que se sigan generando ítems nuevos. Esto necesariamente terminará eventualmente con un conjunto tal que no se produce ningún ítem nuevo al aplicar ninguna operación Scanner, Predictor o Completer, y entonces habremos terminado el procesamiento del conjunto de Earley.

La solución anterior es indeseable porque implica reprocesar el mismo conjunto múltiples veces. Una alternativa implementada en SPARK [3] es la de tratar de manera diferente a los símbolos anulables: En SPARK, antes de ejecutar el algoritmo de Earley se calculan todos los no terminales de la gramática A tales que $A \stackrel{*}{\Rightarrow} \lambda$. Luego, en el algoritmo de Earley implementado en SPARK nunca se reprocesan conjuntos ni ítems de Earley, pero en el paso predictor, si en un ítem $(i, A \to \alpha \bullet B\beta)$ el no terminal B a la derecha de $\bullet$ es anulable, se agrega directamente $(i, A \to \alpha B \bullet \beta)$ al conjunto actual (es evidente por la propiedad fundamental y el hecho de que B es anulable que este ítem será agregado por el algoritmo de Earley en algún momento). Esta optimización sencilla garantiza que no se pierde nada en el paso completer, puesto que aquellos no terminales B que podrían ser completados sobre el conjunto actual son necesariamente símbolos anulables, y por lo tanto los ítems que se agregarían fueron agregados inmediatamente en el paso predictor.

En nuestro caso, no hemos utilizado esta optimización de SPARK para resolver el problema, puesto que al agregar el multistack inicial y final a los ítems de Earley, ya no es suficiente con conocer los símbolos anulables (en el sentido libre de contexto, ignorando las operaciones de pila), pues su anulabilidad puede depender de la multistack inicial, y a su vez, no está claro con qué multistack debería completarse el ítem en el paso Predictor, pues por más que el símbolo sea anulable, su procesamiento podría desencadenar muchas operaciones de pila.

La alternativa que utilizaremos consistirá en mantener para el conjunto actual lo que hemos denominado en la implementación como λ-*watchlist*. Esta estructura almacena conceptualmente todo lo que puede ser completado sobre ítems del mismo conjunto actual, de forma tal que al procesar con Completer un ítem que completa sobre el mismo conjunto actual, agregamos la información correspondiente a la λ-watchlist, y cada vez que agregamos un ítem al conjunto actual, verificamos si hay en la λ-watchlist información que permite completarlo, y realizamos el correspondiente paso completer en dicho caso.

Es decir, en lugar de realizar múltiples pasadas para que el procesamiento de un ítem i_1 mediante Completer pueda operar sobre un ítem i_2 generado con posterioridad, mantenemos toda la información de pasos Completer pasados que pueda afectar a ítems futuros del mismo conjunto actual, y entonces aplicamos estos pasos de Completer "atrasados" cuando aparecen los nuevos ítems.

En nuestra implementación, la λ-watchlist es un diccionario de pares (A, ξ_i) en un conjunto de posibles ξ, siendo A no terminal y ξ_i, ξ multistacks. Esto significa que si en el conjunto actual se agrega un ítem que tiene a la derecha del $\bullet$ un no terminal A, y cuyo multistack actual es ξ_i, este ítem puede ser completado generando uno nuevo en el conjunto actual con multistack actual ξ. Por otra parte, cuando se procesa con Completer un ítem, se agrega ξ al conjunto asociado a (A, ξ_i), siendo ξ_i la multistack inicial del ítem que se procesa mediante Completer, A el no terminal a la izquierda de la regla, y ξ la multistack actual del ítem que se procesa.

Algo importante de destacar es que este problema afecta únicamente al reconocedor general de MGIG, y no afecta a las versiones particulares desarrolladas para gramáticas determinadas a derecha, ya que en este último caso en el paso Completer no es necesario explorar ningún conjunto de Earley para saber qué regla completar. Además esta estructura utiliza el multistack inicial de los ítems, que es algo que no se almacena en los algoritmos para gramáticas determinadas a derecha. Por lo tanto, en nuestras implementaciones de `EarleyRDMGIGRecognizer` y `UnifyingEarleyRDMGIGRecognizer` no se utiliza λ-watchlist, sino que esta estructura es únicamente utilizada en `GeneralEarleyMGIGRecognizer`.

A.3.4. Filtros

Una práctica usual en algoritmos de parsing es la utilización de *filtros* que descarten estados intermedios "inválidos", que pueden ser generados por la definición de la gramática utilizada pero que se sabe por sus características que nunca podrán ser parte de una derivación exitosa. Al hacer esto, se puede reducir el cómputo y la cantidad de ítems totales generados al no generar nunca ítems de Earley correspondientes a dichos estados.

En nuestra implementación, permitimos la utilización de dos tipos de filtros:

- *ItemFilters*: Implementativamente, un item filter es una función de python

  `f(item, inputToken, multistackTops)`

  que retorna `True` si el correspondiente estado es válido de acuerdo al filtro, y `False` sino, es decir, si debe ser filtrado. Antes de agregar un nuevo ítem a un conjunto de Earley, siempre se verifica su validez con respecto a todas las funciones provistas al constructor del reconocedor en la lista `itemFilters`.

  Los filtros de este tipo que utilizamos para las gramáticas de SAT son:

  - `filtroYaPaso`: Si el número de variable del tope de la menor stack, es menor al número de variable en la regla de producción, el ítem no es válido. En términos de la derivación, tal situación significaría que ya hemos completado el procesamiento de la variable i-ésima, pero sin embargo se han dejado en la multistack cláusulas que comienzan con dicha variable, de manera que nunca podrán eliminarse de la misma.
  - `filtroFaltaToken`: Similar al anterior, pero en lugar de utilizar un número de variable en el multistack, se considera el siguiente token a leer del input correspondiente al ítem que está agregando. Si dicho número de variable es menor al número de variable correspondiente al no terminal de la producción, entonces será imposible generar correctamente la cláusula de la cadena de entrada en la derivación, pues utiliza una variable cuyo procesamiento ya terminó.
  - `filtroFaltanPops` (Solo para RTAS y SRTAS): Se consideran inválidos los ítems cuya producción tenga un terminal A_i o B_i^P, y el número de variable en el tope de la menor stack sea i. Esto es inválido porque en este caso, la fase de "pops" de la derivación ya pasó, pero quedan en la multistack cláusulas que comienzan con la variable i, y por lo tanto nunca podrán ser eliminadas.
  - `filtroEleccionInconsistente` (Solo para RTAS y SRTAS): Se consideran inválidos los ítems que tengan ia en el tope del multistack, y una producción para el no terminal B_i^R. Análogamente para ib y B_i^R. Tales estados corresponderían a una elección inconsistente de variables: en términos de la derivación, el elemento ia (o ib) en el multistack nunca podrá ser popeado.

- *PushFilters*: Implementativamente, un push filter es una función de python

  `f(mu, multistackTops)`

  que retorna `True` si el correspondiente estado es válido de acuerdo al filtro, y `False` sino, es decir, si debe ser filtrado. Antes de hacer push de una cadena `mu` a un multistack con los topes de stacks indicados por `multistackTops`, se verifica su validez con respecto a todas las funciones provistas al constructor del reconocedor

en la lista `pushFilters`. Si los filtros no permiten el push, el nuevo ítem no es generado.

El único filtro de este tipo en nuestra implementación para SAT es `filtroPushearMalaEleccion`: en este caso, se descartan aquellos ítems que se obtendrían haciendo push de una cadena de la forma ia (una cláusula con un solo literal), en una multistack que contiene a ib entre sus topes de stack, o viceversa.

A.4. Reconocedores implementados

En nuestra implementación, el módulo `mgig.py` contiene 3 clases que implementan distintas versiones del algoritmo Earley-MGIG. Estas se describen a continuación. Todas las versiones tienen idéntica interfaz: Para utilizar el reconocedor, primero se debe crear una instancia indicando las siguientes opciones como parámetros del constructor:

- `productions`: Lista de instancias de `Production`
- `initialSymbol`: Instancia de `EarleyNonTerminal`
- `orderingFunction`: Función de python que implemente la relación de preorden O. $f(a, b)$ implementa la operación $a \leq b$.
- `itemFilters` y `pushFilters`: Listas de filtros a utilizar.
- `printFullSets`: Variable booleana que indica si se debe imprimir el detalle de los conjuntos de Earley, a modo de información de debug.
- `printSetSize`: Variable booleana que indica si se debe imprimir la cantidad de ítems en los distintos conjuntos de Earley, a modo de información de debug.
- `outputStackTrieToFile`: Este parámetro indica el nombre de archivo donde debe ser generado un gráfico `.png` del Trie de stacks generado durante toda la ejecución del algoritmo. Se indica `None` si no se desea generar tal gráfico.

La mayoría de estas opciones pueden ser controladas, al ejecutar ejemplos con el módulo `run_examples.py`, mediante el archivo de configuración `examples.json`.

Una vez que se tiene una instancia del reconocedor, se puede utilizar el método `matchesGrammar(self, inputTokens)` para reconocer una secuencia de tokens de entrada (cada token es una cadena, y estas deberán corresponderse con las regex presentes en los terminales de la gramática). Este devuelve `True` cuando la secuencia de tokens de entrada pertenece al lenguaje generado por la correspondiente MGIG.

A.4.1. GeneralEarleyMGIGRecognizer

La clase `GeneralEarleyMGIGRecognizer` implementa el algoritmo general de reconocimiento de MGIG explicado en 4.1.2. Por lo tanto, puede utilizarse para reconocer una MGIG arbitraria. Esta versión utiliza la λ-watchlist.

A.4.2. EarleyRDMGIGRecognizer

La clase `EarleyRDMGIGRecognizer` implementa el algoritmo de reconocimiento de MGIG para el caso particular de gramáticas determinadas a derecha, explicado en 5.1.5. Por lo tanto, solamente puede utilizarse con MGIG determinadas a derecha, y esto es verificado por el algoritmo en el constructor: si se provee una MGIG que no es determinada a derecha, se lanza una `NotRDMGIGException`.

Esta versión no requiere ni utiliza una λ-watchlist, ya que al leer la gramática determinada a derecha pasada en el constructor, se determina para cada no terminal una regla de tipo 2 que completará, y esa regla es la que se utiliza directamente en Completer.

En esta versión las multistacks son representadas en los ítems de manera explícita, como una lista ordenada de nodos del Trie de stacks. En particular, dos multistacks que no sean exactamente idénticas dan lugar necesariamente a ítems diferentes.

A.4.3. UnifyingEarleyRDMGIGRecognizer

La clase `UnifyingEarleyRDMGIGRecognizer` implementa el mismo algoritmo de reconocimiento de MGIG determinadas a derecha que la anterior, pero utilizando la técnica de unificación de multistacks explicada en 4.3.2. Por lo tanto al igual que la anterior, solamente puede utilizarse con MGIG determinadas a derecha, lo cual es verificado por el algoritmo.

A.5. Formato de archivos de gramáticas

Para facilitar la definición y utilización de MGIG en las pruebas, hemos definido en este trabajo un sencillo formato de archivos de texto para especificar una MGIG.

Transcribimos aquí una breve especificación de este lenguaje.

A.5.1. Símbolos especiales

Los siguientes caracteres (o secuencias de caracteres) tienen un significado especial en el formato y no deben ser utilizados en el lenguaje más que para su función específica:

- Los espacios son los que delimitan tokens (excepto algunos especiales como `-->`, `:`, `;`): **NINGÚN** token puede contener espacios. Ni siquiera las regexes. Esta es una limitación innecesaria que debería ser eliminada para un formato de propósito general, pero con esta versión sencilla es suficiente para el propósito de nuestro trabajo.
- `#` (reemplazo de variable por su valor, es un reemplazo sintáctico directo)
- `:` (separa range de la regla / terminal)
- `;` (para definir variables locales antes de una regla o terminal)
- `-->` (se considera una sola unidad, debería haber exactamente uno por línea de regla)
- `=` (usado en terminales, startsymbol, range, variables locales)

A.5.2. Identificadores

Se puede utilizar identificadores para los parámetros de la gramática y las variables (range o variables introducidas manualmente).

Los identificadores están formados por caracteres alfanuméricos, y las correspondientes variables toman valores enteros. Un identificador `i` se transforma en su representación textual con `#i`. De esta forma si `i = 12`, son completamente equivalentes `S12` y `S#i`.

La totalidad de los identificadores utilizados debe ser **libre de prefijos** (ningún identificador puede ser prefijo de otro identificador), y si esto no se cumple no está garantizado cuál se aplica (ambigüedad). Por ejemplo si `a = 3` y `ab = 5`, `#abc` podría expandirse a `3bc` o a `5c`.

A.5.3. Terminales y no terminales

Se recomienda fuertemente respetar la convención de que un símbolo no terminal viene dado por una secuencia de letras mayúsculas,números y símbolos (pero no minúsculas), mientras que un símbolo terminal viene dado por una secuencia de letras minúsculas, números y símbolos (pero no mayúsculas).

Es necesario declarar los símbolos terminales, pero los no terminales son automáticos y pueden usarse directamente (todo símbolo que aparece en una regla y no es terminal se considera un no terminal).

A.5.4. Estructura del archivo de gramática

Todo archivo tendrá la siguiente estructura ordenada:

- `PARAMS <params>`

  En una primera línea, se deben indicar los *parámetros* que tomará esta gramática (por ejemplo el n que indica la cantidad de variables en las gramáticas para SAT).

- `STARTSYMBOL = S`

  En una segunda línea, se indica el símbolo no terminal que es el símbolo inicial de la MGIG.

- `TERMINALS`

  Una tercera línea `TERMINALS` por si misma indica el comienzo de la sección de declaración de terminales. Cada terminal se declarará a continuación en una línea separada, hasta que aparezca una línea `RULES` en el archivo.

  Una declaración de un símbolo terminal tiene simplemente la forma `<name> = <regex>`, indicando la regex de python (en forma "raw", sin utilizar caracteres de escape adicionales) que corresponderá con ese símbolo terminal.

  Es posible utilizar la construcción `RANGE` e introducir variables para poder declarar múltiples terminales brevemente en una misma línea. Ejemplos:

  ```
  lpar = \(
  RANGE i = 1 TO #n - 1 : #iaw* = #ia(.*)
  ```

  Si `n=3` el segundo ejemplo equivale a las líneas individuales:

  ```
  1aw* = 1a(.*)
  ```

```
2aw* = 2a(.*)
```

- `RULES`

  La sección de declaración de las reglas de producción de la MGIG es la más importante, y comienza con `RULES` en una única línea.

  Una declaración de regla tiene la forma `<nonTerminal> --> <tokenList> <mu>`, donde la lista de tokens contendrá símbolos terminales y no terminales separados por espacios. El índice μ de la regla se especifica al final de la misma, tomando `<mu>` valores de la siguiente manera:

  - $\mu = \lambda$ (producción libre de contexto) , se omite `<mu>`.
  - $\mu = x$ (push del terminal x), se pone `PUSH r`, siendo `r` una cadena de formato de python, que puede utilizar opcionalmente el wildcard {0}, que será reemplazado por el texto del terminal w correspondiente presente en la regla push.
  - $\mu = \overline{x}|y$ (pop del terminal x, push del sufijo y), se pone `PUSHPOP t`, siendo `t` el terminal que se popea, y en cuya regex, el primer grupo capturado indicará el sufijo que se volverá a pushear al multistack.
  - $\mu = \overline{x}$ (pop del terminal x), se pone `POP t`, siendo `t` el terminal que se popea.

- Construcción range y variables:

  En las líneas que expresan reglas, o las que declaran terminales, es posible utilizar la construcción `RANGE` para producir múltiples reglas de manera concisa. También pueden declararse de manera opcional variables locales a la línea. La sintaxis es:

  `RANGE <varName> = <expr> TO <expr> : <varName> = <expr> ; <varName> = <expr> ; <rule>`

  Es decir, de utilizarse, la construcción `RANGE` se termina con un `:`, y se usa en la misma línea, al comienzo de la misma y antes de la regla o declaración de terminal. Similarmente, si se usan declaraciones de variables locales a la línea, se colocan antes de la regla (pero después de `RANGE` en caso de utilizarse esta última). Una expresión es una lista de enteros, separados por caracteres `+` o `-`, que es evaluada aritméticamente a su valor concreto. Este valor es almacenado en la variable indicada en el caso de variables locales. En el caso de `RANGE`, la regla y sus variables locales se evalúa múltiples veces en sucesión, para todos los enteros en el rango indicado.

  Un ejemplo sería por ejemplo una línea como:

  `RANGE i = 1 TO #n - 1 : next = #i + 1 ; prev = #i - 1 ; <rule>`

  Notar los espacios alrededor de los operadores `+` y `-`: Los espacios son los separadores de tokens en estos archivos y por lo tanto son importantes.

A.5.5. Ejemplo

A continuación, mostramos por ejemplo un archivo que codifica la gramática n-TAS:

```
// Accepts the TAS language, that is, satisfiable instances of sat,
// with literals ordered by number within a clause, and
// clauses ordered by first literal number.
PARAMS n
STARTSYMBOL = S1
TERMINALS
RANGE i = 1 TO #n : #iaw* = #ia
RANGE i = 1 TO #n : #ibw* = #ib
RANGE i = 1 TO #n : #iaw+ = #ia(.+)$
RANGE i = 1 TO #n : #ibw+ = #ib(.+)$
RULES
// Context free assignment of truth values.
RANGE i = 1 TO #n - 1 : next = #i + 1 ; S#i --> A#i S#next
RANGE i = 1 TO #n - 1 : next = #i + 1 ; S#i --> B#i S#next
S#n --> A#n
S#n --> B#n
// True literals make a clause true.
RANGE i = 1 TO #n : A#i --> #iaw* A#i
RANGE i = 1 TO #n : A#i -->
RANGE i = 1 TO #n : B#i --> #ibw* B#i
RANGE i = 1 TO #n : B#i -->
// If a literal is false, the rest of the clause must be true later.
RANGE i = 1 TO #n : A#i --> #ibw+ A#i PUSH {0}
RANGE i = 1 TO #n : B#i --> #iaw+ B#i PUSH {0}
// If a literal is true, pending clauses can be satisfied
RANGE i = 1 TO #n : A#i --> A#i POP #iaw*
RANGE i = 1 TO #n : B#i --> B#i POP #ibw*
// If a literal is false, pending clauses must be satisfied still later
RANGE i = 1 TO #n : A#i --> A#i PUSHPOP #ibw+
RANGE i = 1 TO #n : B#i --> B#i PUSHPOP #iaw+
```

A.6. Carga de archivos en formato DIMACS-CNF

Para realizar pruebas de nuestra implementación, es conveniente para el caso de TAS poder cargar instancias arbitrarias del problema SAT desde archivos de datos. Para esto hemos implementado la función `dimacsToTas(filename)` en el módulo `tas.py`, que lee el contenido de un archivo en formato DIMACS-CNF [1], y lo convierte en una instancia (cadena de texto) que pertenezca al lenguaje $\mathcal{SAT}$ si y solo si la instancia de SAT indicada en el archivo es satisfacible. Para esto se procede codificando la instancia como se explicó en 3.2.

Además, para asegurar que es posible utilizar cualquier instancia de SAT con cualquiera de las gramáticas, hemos implementado la transformación mencionada en 3.4.2.1 en la función `tasToRestrictedTas`, en el mismo módulo `tas.py`.

A.7. Información de debug

A.7.1. Detalle de conjuntos algoritmo de Earley

Con la opción `printFullSets`, se activa la visualización completa de cada uno de los conjuntos de Earley S_i. En este caso, para cada uno de ellos se listan todas las producciones anotadas con • presentes en ítems del conjunto y, para cada una de ellas, se indican los distintos multistacks posibles, cada uno de los cuales se correspondería con un ítem distinto.

Para el caso de `UnifyingEarleyRDMGIGRecognizer`, este listado de multistacks no es explícito, sino que se muestra una descripción de las multistacks de cada ítem, acorde a la representación utilizada por esta estructura.

Por ejemplo, a continuación se muestra una visualización completa del conjunto S_1, al trabajar con filtros para la gramática SRTAS con la cadena de entrada `1b2a 2b`.

- Algoritmo `EarleyRDMGIGRecognizer`:

```
pos = 1
       AP1 -> 1bw+ . AP1 | push {0}
             multistacks:
              ms= ((2a, #),)
       AP1 -> . 1bw+ | push {0}
             multistacks:
              ms= ((2a, #),)
       AP1 -> . 1bw+ AP1 | push {0}
             multistacks:
              ms= ((2a, #),)
       AP1 -> 1bw+ . | push {0}
             multistacks:
              ms= ((2a, #),)
       A1 -> AP1 .
             multistacks:
              ms= ((2a, #),)
       B1 -> 1bw* . B1
             multistacks:
              ms= ()
       B1 -> . 1bw*
             multistacks:
              ms= ()
       AR1 -> A1 .
             multistacks:
              ms= ((2a, #),)
       B1 -> 1bw* .
             multistacks:
              ms= ()
       B1 -> . 1bw* B1
             multistacks:
              ms= ()
       S1 -> AR1 . S2
             multistacks:
              ms= ((2a, #),)
       BP1 -> B1 .
             multistacks:
              ms= ()
       S2 -> . BR2
             multistacks:
```

```
        ms= ((2a, #),)
    BR1 -> BP1 .
          multistacks:
           ms= ()
    S1 -> BR1 . S2
          multistacks:
           ms= ()
    S2 -> . BR2
          multistacks:
           ms= ()
    BR2 -> . B2
          multistacks:
           ms= ()
    B2 -> . 2bw* B2
          multistacks:
           ms= ()
    B2 -> . 2bw*
          multistacks:
           ms= ()
Set 1 size: items=18, total multistacks=19
```

A.7.2. Gráfico de estructura global de stacks

Con la opción `outputStackTrieToFile`, se puede especificar un archivo a donde volcar la representación del Trie de stacks generado por el algoritmo.

Inicialmente trabajamos con las versiones mencionadas en A.8, que exploraban la idea de unificación a nivel de Stack, generando entonces un DAG de Stacks en lugar de un árbol. Especialmente durante esta etapa, la generación de gráficos de estos DAGs fue útil para estudiar el funcionamiento del algoritmo.

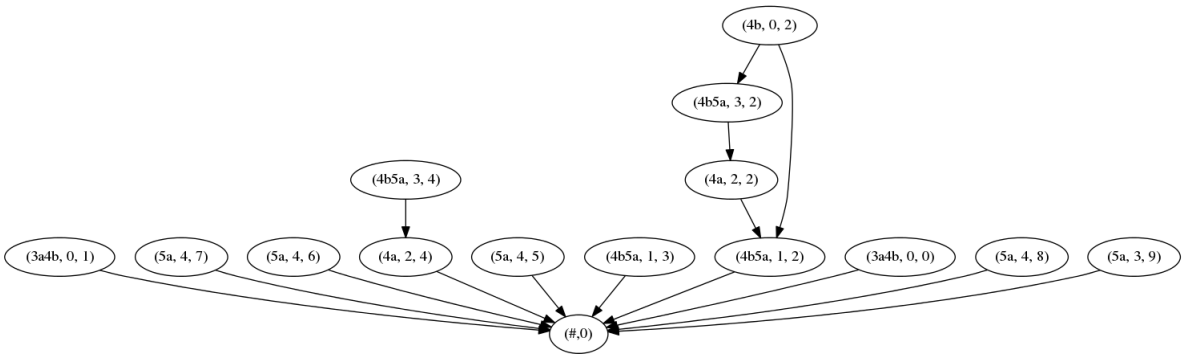


Fig. A.1: Ejemplo de gráfico generado para las versiones previas, cuando utilizábamos unificación a nivel de stack. En particular, notar que las Stacks se estructuran en un DAG, no un árbol.

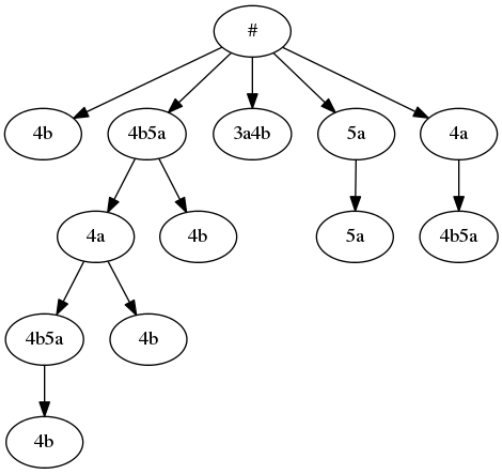


Fig. A.2: Ejemplo de gráfico del Trie de stacks. Notar que en esta versión de la estructura, dibujamos las flechas en el sentido contrario a la versión anterior.

El código correspondiente a la generación de estos gráficos puede encontrarse en `msgraph.py`. Para esta tarea utilizamos las librerías `python-graph`, `Graphviz` y `pydot`.

A.8. Historia de versiones consideradas

Nuestra intención inicial era tratar de aprovechar la idea de *unificación de estados*, para procesar todos juntos (en una única operación) todos aquellos ítems de Earley que serían procesados de igual manera por el algoritmo general (al tener multistacks con un mismo tope), y solo volver a *separar* estos ítems cuando una operación de pop así lo requiera. Esta idea es completamente análoga a la introducida por el autor de [46], en el cual la utiliza para dar un algoritmo de Parsing-GLR con resultados de igual generalidad que el algoritmo de Earley, pero basados en un autómata clásico LR.

A continuación mostramos las distintas versiones de unificación consideradas. En todas las versiones se trabaja con una **estructura global** de multistack, que será un DAG (en la última versión como veremos terminaremos usando simplemente tries, es decir DAGs con forma de árbol) que generaliza a la representación natural de pilas mediante tries como consecuencia de la unificación realizada. Un ítem de Earley constará entonces de una lista ordenada de punteros a nodos de este DAG global. Ante un pop, se deberá cambiar el puntero de la correspondiente pila por un puntero a un antecesor del nodo, lo cual puede dar lugar a más de una posibilidad cuando exista más de un ancestro (debido a la unificación).

Una propiedad interesante para destacar es que todas las versiones que siguen, a pesar de no dar un reconocedor correcto, carecen por completo de falsos negativos: Es decir, si cualquiera de estas versiones indica que una cadena no pertenece al lenguaje, entonces podemos estar seguros de que eso es cierto y la cadena efectivamente no pertenece al lenguaje. Esto es porque en todas ellas, la única diferencia con el algoritmo de Earley-MGIG para determinadas a derechas sobre el cual ya hemos demostrado la correctitud, es que se generan ítems "adicionales" con multistacks erróneas (como resultado de la unificación en el DAG global que representa a las stacks), pero en el algoritmo nunca faltarán los representaciones de los ítems que son correctos en el sentido de la propiedad fundamental. Por lo tanto, si una cadena de entrada está en el lenguaje, necesariamente se producirá un ítem

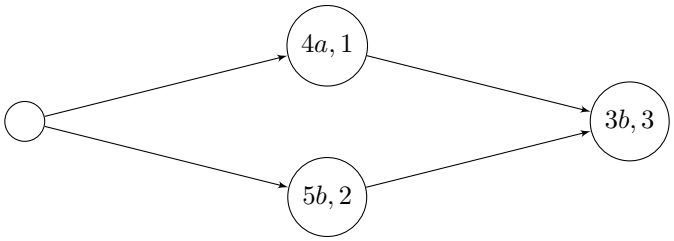
para $S' \to S\bullet$ con multistack vacía, y por lo tanto la cadena será aceptada. Por lo tanto, el algoritmo no tendrá nunca falsos negativos, aunque como explicamos a continuación las versiones anteriores a la final tienen falsos positivos.

Daremos los ejemplos que producen falsos positivos que encontramos inicialmente para cada versión, motivando a la siguiente versión. Solamente estudiaremos en cierto detalle el ejemplo final, que resulta relativamente simple y sirve de contraejemplo a todas las versiones previas a la versión final (es decir, la versión actual sin unificación a nivel de stack), pero que fue el último que encontramos.

A.8.1. MGIG-REC-TOKENPOS (MRT)

Esta es la primera versión que consideramos. La idea de esta versión es que en cada nodo del multistack se indique el **token** correspondiente a ese tope de pila, junto a la **posición** en la cadena de entrada en la cual fue pusheado ese índice. Notar que esta posición se mantiene ante una operación de pop-push. Podemos llamar *clave* de un nodo a este par formado por el token y su posición en el input, y lo que exigimos durante el algoritmo es que no existan en el multistack global dos nodos con una misma clave. Más precisamente, cuando se crea un nodo nuevo como consecuencia de una operación de push (o pop-push), si existe ya un nodo con la correspondiente clave en el multistack se utiliza directamente en lugar de crear un nodo nuevo, lo cual provoca que un nodo pueda tener varios antecesores si fue agregado desde múltiples nodos distintos.

Por ejemplo, el siguiente podría ser un DAG resultante de procesar `1a4a 1b5b 2b3b`



De esta forma, el nodo etiquetado con $(3b, 3)$ representaría dos stacks: `3b 4a` y `3b 5b`.

Esta versión no resulta correcta. Notar que para la correctitud de esta versión (y las siguientes) es necesario que multistacks distintas que son representadas de idéntica manera (es decir, con una misma lista ordenada de punteros a nodos de este DAG global mostrado) no den lugar a derivaciones posteriores fundamentalmente diferentes, ya que al unificarlas a una misma representación, ya no es posible distinguirlas entre sí.

Esto puede pasar: el primer contraejemplo que encontramos para este algoritmo fue el siguiente:

`2a7a9a 2b7a9a 2a7b9a 2b7b9a 2b7b9b 2a7b9b 7a9b`

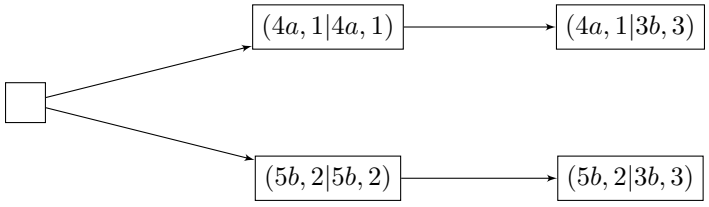
El siguiente algoritmo considerado (TOKENPOS-BASE) funciona correctamente en este caso.

A.8.2. MGIG-REC-TOKENPOS-BASE (MRTB)

Para evitar el problema generado en el ejemplo anterior consideramos agregar a la clave de los nodos el token y posición en el input de la *base de pila* correspondiente a la pila representada por el nodo. De esta forma solo se pueden unificar dos nodos cuando además de coincidir sus topes de pila, coinciden también sus bases de pila. Esto es esencialmente

equivalente a pensar en que tenemos un DAG diferente por cada posible clave de la base de pila de un stack, ya que dos nodos con diferente clave para la base de pila únicamente podrán relacionarse a través del nodo raíz.

Para el ejemplo de `1a4a 1b5b 2b3b` tendríamos ahora el siguiente DAG (para cada nodo, se indica (base de pila|tope de stack)):



Este algoritmo pese a resolver el problema del caso que encontramos antes, también produce falsos positivos. El primer ejemplo de esto que encontramos fue para

`1a2b5a 1a2a5a 1a2a5b 1b2b5a 1b2a5a 1b2a5b 2b6b 3b4b 4b 5b 6b 7b8a 8b`

El siguiente algoritmo de unificación considerado (TOKENPOS-STACKID) funciona correctamente en este caso.

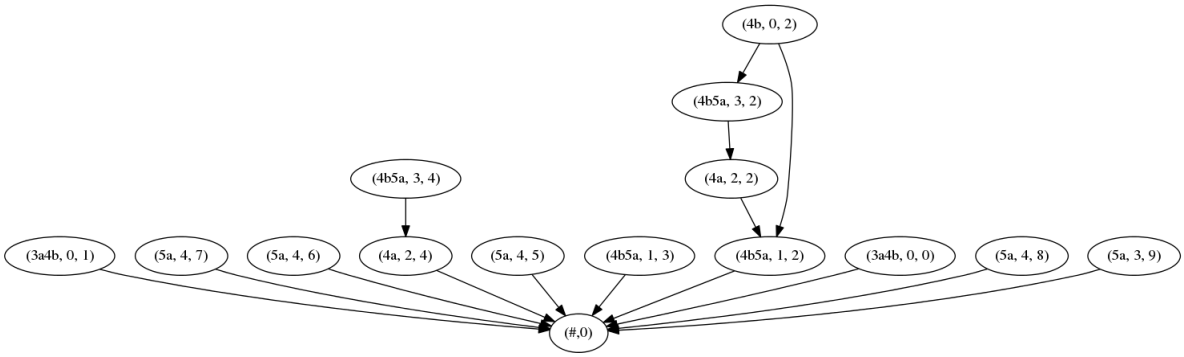
A.8.3. MGIG-REC-TOKENPOS-STACKID (MRTS)

Para solucionar el problema anterior introducimos el concepto de STACK-ID, que es simplemente un identificador único para cualquier posible stack generada durante la ejecución del algoritmo. Para esto simplemente mantendremos un contador global (inicializado por ejemplo a cero) que se incrementa cada vez que se hace push al nodo base de pila (que sirve de raíz del DAG), en cuyo caso se crea un **nuevo nodo** con el id de pila indicado por el contador global. La clave será entonces el token y posición del tope de pila, junto con el id de pila correspondiente. Esto evita unificar dos pilas "distintas" en el algoritmo, entendidas como dos pilas que no fueron creadas en una misma operación.

Esto aún es insuficiente para evitar falsos positivos, como muestra el ejemplo que estudiamos a continuación:

`1b3a4b 1b4b5a 2a4a 2a4b5a 2b5a 3b 4b 5b`

Esta es una instancia falsa para SAT: las variables 3,4 y 5 están determinadas en b, con lo cual la cláusula $2a4a$ obliga a que la variable 2 tenga el valor a, pero eso se contradice con la $2b5a$. Se puede corroborar que el algoritmo producirá el siguiente DAG de stacks:



Para cada nodo, se indica $(\text{token}, \text{posición}, \text{id-de-pila})$. La parte interesante del gráfico es la situación de la pila con identificador 2: veamos que la unificación que allí se produce lleva a un falso positivo en este algoritmo.

En efecto, podemos considerar las siguientes dos derivaciones parciales:

- $S_1 \Rightarrow A_1S_2 \Rightarrow$ `3a4b #1b3a4b` $A_1S_2 \Rightarrow$ `3a4b #4b5a #1b3a4b 1b4b5a` $S_2 \Rightarrow$
  `3a4b #4b5a #1b3a4b 1b4b5a` $B_2S_3 \Rightarrow$
  `3a4b #4a 4b5a #1b3a4b 1b4b5a 2a4a` $B_2S_3 \Rightarrow$
  `3a4b #4b5a 4a 4b5a #1b3a4b 1b4b5a 2a4a 2a4b5a` $B_2S_3 \Rightarrow$
  `3a4b #4b5a 4a 4b5a #1b3a4b 1b4b5a 2a4a 2a4b5a 2b5a` $S_3 \Rightarrow$
  `3a4b #4b5a 4a 4b5a #1b3a4b 1b4b5a 2a4a 2a4b5a 2b5a` $B_3S_4 \Rightarrow$
  `4b 4b5a 4a 4b5a #1b3a4b 1b4b5a 2a4a 2a4b5a 2b5a` B_3S_4
- $S_1 \Rightarrow A_1S_2 \Rightarrow$ `3a4b #1b3a4b` $A_1S_2 \Rightarrow$ `3a4b #4b5a #1b3a4b 1b4b5a` $S_2 \Rightarrow$
  `3a4b #4b5a #1b3a4b 1b4b5a` $A_2S_3 \Rightarrow$
  `3a4b #4b5a #1b3a4b 1b4b5a 2a4a` $A_2S_3 \Rightarrow$
  `3a4b #4b5a #1b3a4b 1b4b5a 2a4a 2a4b5a` $A_2S_3 \Rightarrow$
  `3a4b #4b5a #5a #1b3a4b 1b4b5a 2a4a 2a4b5a 2b5a` $S_3 \Rightarrow$
  `3a4b #4b5a #5a #1b3a4b 1b4b5a 2a4a 2a4b5a 2b5a` $B_3S_4 \Rightarrow$
  `4b 4b5a #5a #1b3a4b 1b4b5a 2a4a 2a4b5a 2b5a` B_3S_4

Sabemos que la cadena no pertenece al lenguaje, así que no estas derivaciones parciales no pueden extenderse a una derivación exitosa. En el primer caso, la derivación claramente no puede terminar con multistack vacía, porque tenemos en el único stack, tanto un `4a` como un `4b`. En el segundo caso, el multistack contiene un `5a`, incompatible con la cadena de entrada.

Los pasos mostrados en la primera línea coinciden para ambas derivaciones, y en ellos se crea la pila que comienza con $4b5a$, que en el DAG resulta tener identificar 2. Notemos que para ambas derivaciones, la pila en cuestión tiene el mismo identificador pues este depende únicamente del momento en que dicha pila es creada, y las derivaciones son idénticas hasta ese punto, con lo cual difieren recién después de creada esta pila. En virtud de esto, el DAG para la pila tiene la forma mostrada, como resultado de que se hagan en esta pila push hasta llegar a `4b 4b5a 4a 4b5a` en una derivación, y a `4b 4b5a` en la otra. Además, el `4b` del cual se hace push al final de la derivación es exactamente el mismo en ambas, pues en ambas se produce como resultado de un pop-push sobre exactamente la misma pila que ya existe en los primeros pasos, y por lo tanto este `4b` tiene la misma posición en el input 0 en ambas derivaciones, correspondiéndole así exactamente el mismo nodo en el DAG. Es decir, las dos stacks son unificadas en este caso.

Ahora bien, notemos que como resultado de la unificación, cuando el algoritmo realice pop sobre este nodo del DAG, se obtendrán ambas stacks posibles `4b5a` y `4b5a 4a 4b5a`. Esto hace que, si al continuar una de estas derivaciones con el correspondiente stack de la otra se puede obtener finalmente una derivación exitosa, el algoritmo así lo haga y entonces se reportará un falso positivo. Tal derivación en efecto existe, por ejemplo completando la primera derivación con el stack `4b 4b5a` de la segunda:

- $\cdots \Rightarrow$
  `4b 4b5a #1b3a4b 1b4b5a 2a4a 2a4b5a 2b5a` $B_3S_4 \Rightarrow$
  `4b 4b5a #1b3a4b 1b4b5a 2a4a 2a4b5a 2b5a 3b` $S_4 \Rightarrow$
  `4b 4b5a #1b3a4b 1b4b5a 2a4a 2a4b5a 2b5a 3b` $B_4S_5 \Rightarrow$
  `4b5a #1b3a4b 1b4b5a 2a4a 2a4b5a 2b5a 3b` $B_4S_5 \Rightarrow$
  `1b3a4b 1b4b5a 2a4a 2a4b5a 2b5a 3b` $B_4S_5 \Rightarrow$

1b3a4b 1b4b5a 2a4a 2a4b5a 2b5a 3b 4b S_5 $\Rightarrow$
1b3a4b 1b4b5a 2a4a 2a4b5a 2b5a 3b 4b B_5 $\Rightarrow$
1b3a4b 1b4b5a 2a4a 2a4b5a 2b5a 3b 4b 5b

Esto prueba entonces que ante este caso, el algoritmo reportará un falso positivo. Este mismo contraejemplo, además, produce un falso positivo en las versiones anteriores, por exactamente los mismos motivos, ya que dichas versiones utilizan menos información de pila para la unificación, y por lo tanto unifican cualquier cosa que esta versión unifique. Este ejemplo captura el problema esencial que trae intentar unificar **a nivel de stacks**.

A.8.4. MGIG-REC-TOKENPOS-NODEID (MRTN)

Esta versión final correcta es esencialmente equivalente al algoritmo de Earley-MGIG ya explicado. En esta cada multistack está representado por completo como una lista de (punteros) a stacks independientes entre sí, que nunca se unifican. No unificar se corresponde con unificar por "nodo", utilizando un id único distinto cada vez que se crea un nodo nuevo (lo cual impide cualquier posible unificación), y de ahí el nombre.

Como esta versión no producirá nunca ninguna unificación, el DAG será de hecho un árbol. Por este motivo, cada nodo representará unívocamente una única stack, en forma completa. Es posible en este caso colapsar nodos que tengan exactamente el mismo camino a la raíz, y que representan entonces idéntica stack, de forma que haya exactamente un nodo por stack (y no múltiples nodos equivalentes, como los nodos $(5a, 4, 8)$ y $(5a, 3, 9$ del ejemplo). De esta forma, llegamos a la implementación con un Trie de stacks explicada en 4.3.1.

Una vez que establecimos la correctitud de esta versión, exploramos la posibilidad de **unificar a nivel de multistack** en lugar de a nivel de cada stack individual. En efecto, la idea central de la unificación introducida en [46] es la de unificar las pilas que tienen idéntico "estado visible", **con respecto a las operaciones push**. Es decir, aquellas que serán procesadas de idéntica manera por cualquier secuencia de push posterior.

Teniendo eso en cuenta, planteamos una manera diferente de unificar en base a *lista de topes de stacks*, unificando de esta forma a nivel de multistacks completos, y no a nivel de nodos individuales, que es lo que se observa del ejemplo anterior como un planteo esencialmente inválido. Esta versión es la descrita en 4.3.2.

B. DETALLE COMPLETO DE LAS MEDICIONES REALIZADAS

Todas las mediciones que aquí se muestran son de versiones con filtros (A.3.4), ya que no utilizar los filtros genera una performance completamente inaceptable en tiempo y memoria, que no resulta posible ni interesante medir en el hardware en el que se realizaron las pruebas.

Además, en todas las mediciones se evalúan los tres algoritmos, (Earley-MGIG, Earley-MGIG para determinadas a derecha, y la versión de este último con unificación por lista de topes de multistack). En todas también se evalúan las 3 posibles estrategias de ordenamiento de variables, *max*, *min* y *johnson*; en un archivo en particular se evalúan también los ordenamientos inversos. Lo que varía fundamentalmente entre distintos archivos de datos es entonces la gramática concreta utilizada, y el tipo de instancias de SAT sobre las que se evalúa el algoritmo.

Los datos que aquí se muestran son un volcado directo de los resultados de tiempos que se pueden encontrar en la versión digital bajo el directorio `resultados-corridas`. Concretamente allí encontramos los siguientes archivos:

- `resultados-random-6ordenamientos-super-restricted-con-filtros`:

  Gramática SRTAS, ejecutada sobre las instancias *random* con los 6 posibles criterios de ordenamiento de variables.

```
Run for General-MGIG , variable_ordering=johnson
 ******* Finished uf20-0105.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.48 s
 ******* Finished uf20-0135.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.79 s
 ******* Finished uf20-0231.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.93 s
 ******* Finished uf20-0235.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.52 s
 ******* Finished uf20-0635.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.78 s
 ******* Finished uf20-0735.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.50 s
 ******* Finished uf20-0811.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.46 s
 ******* Finished uf20-0812.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.59 s
 ******* Finished uf20-0905.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.94 s
 ******* Finished uf20-0935.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.07 s
    Time (seconds): 0.705766701698 +- 0.223464344837
Run for Right-D-MGIG , variable_ordering=johnson
 ******* Finished uf20-0105.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.34 s
 ******* Finished uf20-0135.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.60 s
 ******* Finished uf20-0231.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.63 s
 ******* Finished uf20-0235.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.35 s
 ******* Finished uf20-0635.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.54 s
 ******* Finished uf20-0735.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.36 s
 ******* Finished uf20-0811.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.34 s
 ******* Finished uf20-0812.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.39 s
 ******* Finished uf20-0905.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.75 s
 ******* Finished uf20-0935.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.89 s
    Time (seconds): 0.518976044655 +- 0.196651915783
Run for Unif-RD-MGIG , variable_ordering=johnson
 ******* Finished uf20-0105.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.53 s
 ******* Finished uf20-0135.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.84 s
 ******* Finished uf20-0231.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.90 s
 ******* Finished uf20-0235.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.48 s
 ******* Finished uf20-0635.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.71 s
```

```
 ******* Finished uf20-0735.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.49 s
 ******* Finished uf20-0811.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.44 s
 ******* Finished uf20-0812.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.59 s
 ******* Finished uf20-0905.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.93 s
 ******* Finished uf20-0935.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.02 s
    Time (seconds): 0.693758106232 +- 0.214320021648
Run for General-MGIG , variable_ordering=anti-johnson
 ******* Finished uf20-0105.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 5.95 s
 ******* Finished uf20-0135.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 5.33 s
 ******* Finished uf20-0231.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 2.60 s
 ******* Finished uf20-0235.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 6.22 s
 ******* Finished uf20-0635.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 4.72 s
 ******* Finished uf20-0735.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 6.11 s
 ******* Finished uf20-0811.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 7.62 s
 ******* Finished uf20-0812.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 2.81 s
 ******* Finished uf20-0905.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 4.08 s
 ******* Finished uf20-0935.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 3.06 s
    Time (seconds): 4.8496035099 +- 1.68668316272
Run for Right-D-MGIG , variable_ordering=anti-johnson
 ******* Finished uf20-0105.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 4.03 s
 ******* Finished uf20-0135.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 3.75 s
 ******* Finished uf20-0231.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.95 s
 ******* Finished uf20-0235.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 4.52 s
 ******* Finished uf20-0635.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 3.98 s
 ******* Finished uf20-0735.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 4.96 s
 ******* Finished uf20-0811.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 5.58 s
 ******* Finished uf20-0812.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 2.11 s
 ******* Finished uf20-0905.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 2.99 s
 ******* Finished uf20-0935.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 2.21 s
    Time (seconds): 3.61023397446 +- 1.25531390785
Run for Unif-RD-MGIG , variable_ordering=anti-johnson
 ******* Finished uf20-0105.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 5.35 s
 ******* Finished uf20-0135.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 4.97 s
 ******* Finished uf20-0231.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 2.92 s
 ******* Finished uf20-0235.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 5.51 s
 ******* Finished uf20-0635.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 4.76 s
 ******* Finished uf20-0735.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 5.88 s
 ******* Finished uf20-0811.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 7.92 s
 ******* Finished uf20-0812.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 2.81 s
 ******* Finished uf20-0905.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 4.13 s
 ******* Finished uf20-0935.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 3.21 s
    Time (seconds): 4.74640872478 +- 1.57093214336
Run for General-MGIG , variable_ordering=max
 ******* Finished uf20-0105.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.54 s
 ******* Finished uf20-0135.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.85 s
 ******* Finished uf20-0231.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.81 s
 ******* Finished uf20-0235.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.51 s
 ******* Finished uf20-0635.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.74 s
 ******* Finished uf20-0735.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.55 s
 ******* Finished uf20-0811.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.65 s
 ******* Finished uf20-0812.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.60 s
 ******* Finished uf20-0905.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.91 s
 ******* Finished uf20-0935.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.06 s
    Time (seconds): 0.721858763695 +- 0.183754977273
Run for Right-D-MGIG , variable_ordering=max
 ******* Finished uf20-0105.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.37 s
 ******* Finished uf20-0135.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.60 s
 ******* Finished uf20-0231.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.56 s
```

```
 ******* Finished uf20-0235.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.36 s
 ******* Finished uf20-0635.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.51 s
 ******* Finished uf20-0735.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.32 s
 ******* Finished uf20-0811.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.35 s
 ******* Finished uf20-0812.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.36 s
 ******* Finished uf20-0905.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.62 s
 ******* Finished uf20-0935.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.75 s
    Time (seconds): 0.478983235359 +- 0.147515479178
Run for Unif-RD-MGIG , variable_ordering=max
 ******* Finished uf20-0105.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.54 s
 ******* Finished uf20-0135.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.88 s
 ******* Finished uf20-0231.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.86 s
 ******* Finished uf20-0235.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.49 s
 ******* Finished uf20-0635.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.71 s
 ******* Finished uf20-0735.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.47 s
 ******* Finished uf20-0811.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.51 s
 ******* Finished uf20-0812.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.53 s
 ******* Finished uf20-0905.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.84 s
 ******* Finished uf20-0935.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.05 s
    Time (seconds): 0.689763903618 +- 0.206386929394
Run for General-MGIG , variable_ordering=anti-max
 ******* Finished uf20-0105.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 5.22 s
 ******* Finished uf20-0135.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 5.30 s
 ******* Finished uf20-0231.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 2.85 s
 ******* Finished uf20-0235.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 5.61 s
 ******* Finished uf20-0635.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 4.40 s
 ******* Finished uf20-0735.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 5.36 s
 ******* Finished uf20-0811.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 7.47 s
 ******* Finished uf20-0812.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 3.13 s
 ******* Finished uf20-0905.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 4.23 s
 ******* Finished uf20-0935.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 3.76 s
    Time (seconds): 4.73396017551 +- 1.35859666371
Run for Right-D-MGIG , variable_ordering=anti-max
 ******* Finished uf20-0105.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 3.85 s
 ******* Finished uf20-0135.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 4.03 s
 ******* Finished uf20-0231.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 2.15 s
 ******* Finished uf20-0235.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 4.11 s
 ******* Finished uf20-0635.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 3.24 s
 ******* Finished uf20-0735.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 4.05 s
 ******* Finished uf20-0811.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 5.48 s
 ******* Finished uf20-0812.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 2.35 s
 ******* Finished uf20-0905.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 2.78 s
 ******* Finished uf20-0935.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 2.63 s
    Time (seconds): 3.46699943542 +- 1.02686930374
Run for Unif-RD-MGIG , variable_ordering=anti-max
 ******* Finished uf20-0105.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 4.67 s
 ******* Finished uf20-0135.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 4.89 s
 ******* Finished uf20-0231.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 3.02 s
 ******* Finished uf20-0235.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 5.15 s
 ******* Finished uf20-0635.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 4.53 s
 ******* Finished uf20-0735.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 4.83 s
 ******* Finished uf20-0811.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 5.90 s
 ******* Finished uf20-0812.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 2.94 s
 ******* Finished uf20-0905.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 3.91 s
 ******* Finished uf20-0935.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 3.29 s
    Time (seconds): 4.31216249466 +- 0.987509765483
Run for General-MGIG , variable_ordering=min
 ******* Finished uf20-0105.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.55 s
```

```
 ******* Finished uf20-0135.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.60 s
 ******* Finished uf20-0231.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.96 s
 ******* Finished uf20-0235.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.60 s
 ******* Finished uf20-0635.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.76 s
 ******* Finished uf20-0735.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.51 s
 ******* Finished uf20-0811.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.42 s
 ******* Finished uf20-0812.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.55 s
 ******* Finished uf20-0905.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.06 s
 ******* Finished uf20-0935.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.13 s
    Time (seconds): 0.712306213379 +- 0.249857434414
Run for Right-D-MGIG , variable_ordering=min
 ******* Finished uf20-0105.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.39 s
 ******* Finished uf20-0135.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.40 s
 ******* Finished uf20-0231.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.68 s
 ******* Finished uf20-0235.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.41 s
 ******* Finished uf20-0635.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.53 s
 ******* Finished uf20-0735.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.33 s
 ******* Finished uf20-0811.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.28 s
 ******* Finished uf20-0812.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.40 s
 ******* Finished uf20-0905.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.75 s
 ******* Finished uf20-0935.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.78 s
    Time (seconds): 0.495309925079 +- 0.180772810468
Run for Unif-RD-MGIG , variable_ordering=min
 ******* Finished uf20-0105.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.54 s
 ******* Finished uf20-0135.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.60 s
 ******* Finished uf20-0231.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.96 s
 ******* Finished uf20-0235.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.59 s
 ******* Finished uf20-0635.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.75 s
 ******* Finished uf20-0735.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.50 s
 ******* Finished uf20-0811.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.41 s
 ******* Finished uf20-0812.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.54 s
 ******* Finished uf20-0905.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.09 s
 ******* Finished uf20-0935.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.08 s
    Time (seconds): 0.705439996719 +- 0.248031663552
Run for General-MGIG , variable_ordering=anti-min
 ******* Finished uf20-0105.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 5.84 s
 ******* Finished uf20-0135.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 4.84 s
 ******* Finished uf20-0231.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 2.17 s
 ******* Finished uf20-0235.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 6.12 s
 ******* Finished uf20-0635.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 5.24 s
 ******* Finished uf20-0735.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 6.99 s
 ******* Finished uf20-0811.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 7.36 s
 ******* Finished uf20-0812.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 2.97 s
 ******* Finished uf20-0905.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 4.79 s
 ******* Finished uf20-0935.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 3.74 s
    Time (seconds): 5.00666801929 +- 1.67972467192
Run for Right-D-MGIG , variable_ordering=anti-min
 ******* Finished uf20-0105.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 4.11 s
 ******* Finished uf20-0135.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 3.54 s
 ******* Finished uf20-0231.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.62 s
 ******* Finished uf20-0235.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 4.39 s
 ******* Finished uf20-0635.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 3.91 s
 ******* Finished uf20-0735.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 5.57 s
 ******* Finished uf20-0811.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 5.53 s
 ******* Finished uf20-0812.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 2.19 s
 ******* Finished uf20-0905.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 3.53 s
 ******* Finished uf20-0935.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 2.75 s
    Time (seconds): 3.7136371851 +- 1.29524968838
```

```
Run for Unif-RD-MGIG , variable_ordering=anti-min
 ******* Finished uf20-0105.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 5.47 s
 ******* Finished uf20-0135.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 5.12 s
 ******* Finished uf20-0231.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 2.49 s
 ******* Finished uf20-0235.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 5.22 s
 ******* Finished uf20-0635.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 5.31 s
 ******* Finished uf20-0735.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 6.41 s
 ******* Finished uf20-0811.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 7.40 s
 ******* Finished uf20-0812.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 2.79 s
 ******* Finished uf20-0905.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 3.82 s
 ******* Finished uf20-0935.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 3.24 s
    Time (seconds): 4.72653009892 +- 1.60019504678
```

- `resultados-random-tas-confiltros`:

  Gramática TAS, ejecutada sobre las instancias *random* con los 3 criterios de ordenamiento de variables. En este caso, ejecutamos únicamente 3 de los 10 archivos, puesto que el procesamiento de esta gramática resulta insumir un tiempo mucho mayor que las demás.

```
Run for General-MGIG , variable_ordering=johnson
 ******* Finished uf20-0105.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 154.42 s
 ******* Finished uf20-0135.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 147.22 s
 ******* Finished uf20-0231.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 251.68 s
    Time (seconds): 184.443422318 +- 58.343330606
Run for Right-D-MGIG , variable_ordering=johnson
 ******* Finished uf20-0105.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 68.58 s
 ******* Finished uf20-0135.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 55.37 s
 ******* Finished uf20-0231.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 79.43 s
    Time (seconds): 67.7953673204 +- 12.0486802927
Run for Unif-RD-MGIG , variable_ordering=johnson
 ******* Finished uf20-0105.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 42.16 s
 ******* Finished uf20-0135.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 42.97 s
 ******* Finished uf20-0231.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 87.55 s
    Time (seconds): 57.5579609871 +- 25.9779731693
Run for General-MGIG , variable_ordering=max
 ******* Finished uf20-0105.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 138.09 s
 ******* Finished uf20-0135.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 456.97 s
 ******* Finished uf20-0231.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 342.06 s
    Time (seconds): 312.375478029 +- 161.498985796
Run for Right-D-MGIG , variable_ordering=max
 ******* Finished uf20-0105.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 47.89 s
 ******* Finished uf20-0135.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 111.68 s
 ******* Finished uf20-0231.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 98.56 s
    Time (seconds): 86.0411825975 +- 33.6892177382
Run for Unif-RD-MGIG , variable_ordering=max
 ******* Finished uf20-0105.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 49.05 s
 ******* Finished uf20-0135.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 113.74 s
 ******* Finished uf20-0231.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 101.70 s
    Time (seconds): 88.1644239426 +- 34.4061220842
Run for General-MGIG , variable_ordering=min
 ******* Finished uf20-0105.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 244.48 s
 ******* Finished uf20-0135.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 121.91 s
 ******* Finished uf20-0231.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 343.12 s
    Time (seconds): 236.501945019 +- 110.819651952
Run for Right-D-MGIG , variable_ordering=min
 ******* Finished uf20-0105.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 74.04 s
 ******* Finished uf20-0135.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 35.76 s
 ******* Finished uf20-0231.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 101.77 s
```

```
   Time (seconds): 70.5209976037 +- 33.1438250397
Run for Unif-RD-MGIG , variable_ordering=min
 ******* Finished uf20-0105.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 74.08 s
 ******* Finished uf20-0135.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 35.49 s
 ******* Finished uf20-0231.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 109.82 s
   Time (seconds): 73.1327747504 +- 37.1759272629
```

- `resultados-random-deterministic-confiltros`: Gramática DTAS, ejecutada sobre las instancias *random* con los 3 criterios de ordenamiento de variables.

```
Run for General-MGIG , variable_ordering=johnson
 ******* Finished uf20-0105.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.70 s
 ******* Finished uf20-0135.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.14 s
 ******* Finished uf20-0231.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.18 s
 ******* Finished uf20-0235.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.69 s
 ******* Finished uf20-0635.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.15 s
 ******* Finished uf20-0735.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.74 s
 ******* Finished uf20-0811.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.72 s
 ******* Finished uf20-0812.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.77 s
 ******* Finished uf20-0905.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.29 s
 ******* Finished uf20-0935.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.59 s
   Time (seconds): 0.997438836098 +- 0.315218432469
Run for Right-D-MGIG , variable_ordering=johnson
 ******* Finished uf20-0105.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.41 s
 ******* Finished uf20-0135.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.70 s
 ******* Finished uf20-0231.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.86 s
 ******* Finished uf20-0235.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.44 s
 ******* Finished uf20-0635.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.61 s
 ******* Finished uf20-0735.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.43 s
 ******* Finished uf20-0811.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.38 s
 ******* Finished uf20-0812.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.45 s
 ******* Finished uf20-0905.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.75 s
 ******* Finished uf20-0935.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.88 s
   Time (seconds): 0.592133522034 +- 0.193487652777
Run for Unif-RD-MGIG , variable_ordering=johnson
 ******* Finished uf20-0105.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.53 s
 ******* Finished uf20-0135.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.89 s
 ******* Finished uf20-0231.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.07 s
 ******* Finished uf20-0235.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.54 s
 ******* Finished uf20-0635.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.79 s
 ******* Finished uf20-0735.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.54 s
 ******* Finished uf20-0811.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.50 s
 ******* Finished uf20-0812.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.76 s
 ******* Finished uf20-0905.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.04 s
 ******* Finished uf20-0935.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.16 s
   Time (seconds): 0.781489276886 +- 0.251510786236
Run for General-MGIG , variable_ordering=max
 ******* Finished uf20-0105.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.72 s
 ******* Finished uf20-0135.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.16 s
 ******* Finished uf20-0231.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.06 s
 ******* Finished uf20-0235.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.72 s
 ******* Finished uf20-0635.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.99 s
 ******* Finished uf20-0735.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.65 s
 ******* Finished uf20-0811.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.67 s
 ******* Finished uf20-0812.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.69 s
 ******* Finished uf20-0905.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.18 s
 ******* Finished uf20-0935.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.48 s
   Time (seconds): 0.932075166702 +- 0.283976484212
```

```
Run for Right-D-MGIG , variable_ordering=max
 ******* Finished uf20-0105.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.42 s
 ******* Finished uf20-0135.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.67 s
 ******* Finished uf20-0231.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.63 s
 ******* Finished uf20-0235.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.40 s
 ******* Finished uf20-0635.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.57 s
 ******* Finished uf20-0735.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.36 s
 ******* Finished uf20-0811.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.39 s
 ******* Finished uf20-0812.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.43 s
 ******* Finished uf20-0905.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.95 s
 ******* Finished uf20-0935.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.76 s
    Time (seconds): 0.658854794502 +- 0.427905297649
Run for Unif-RD-MGIG , variable_ordering=max
 ******* Finished uf20-0105.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.84 s
 ******* Finished uf20-0135.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.34 s
 ******* Finished uf20-0231.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.48 s
 ******* Finished uf20-0235.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.65 s
 ******* Finished uf20-0635.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.84 s
 ******* Finished uf20-0735.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.53 s
 ******* Finished uf20-0811.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.60 s
 ******* Finished uf20-0812.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.60 s
 ******* Finished uf20-0905.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.46 s
 ******* Finished uf20-0935.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.44 s
    Time (seconds): 0.979300117493 +- 0.404324770025
Run for General-MGIG , variable_ordering=min
 ******* Finished uf20-0105.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.90 s
 ******* Finished uf20-0135.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.96 s
 ******* Finished uf20-0231.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.29 s
 ******* Finished uf20-0235.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.01 s
 ******* Finished uf20-0635.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.14 s
 ******* Finished uf20-0735.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.68 s
 ******* Finished uf20-0811.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.66 s
 ******* Finished uf20-0812.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.79 s
 ******* Finished uf20-0905.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.41 s
 ******* Finished uf20-0935.cnf recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.54 s
    Time (seconds): 1.03878481388 +- 0.302046371377
Run for Right-D-MGIG , variable_ordering=min
 ******* Finished uf20-0105.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.48 s
 ******* Finished uf20-0135.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.48 s
 ******* Finished uf20-0231.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.81 s
 ******* Finished uf20-0235.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.47 s
 ******* Finished uf20-0635.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.67 s
 ******* Finished uf20-0735.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.38 s
 ******* Finished uf20-0811.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.32 s
 ******* Finished uf20-0812.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.43 s
 ******* Finished uf20-0905.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.97 s
 ******* Finished uf20-0935.cnf recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.72 s
    Time (seconds): 0.672686219215 +- 0.421563962463
Run for Unif-RD-MGIG , variable_ordering=min
 ******* Finished uf20-0105.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.78 s
 ******* Finished uf20-0135.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.72 s
 ******* Finished uf20-0231.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.11 s
 ******* Finished uf20-0235.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.65 s
 ******* Finished uf20-0635.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.83 s
 ******* Finished uf20-0735.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.56 s
 ******* Finished uf20-0811.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.45 s
 ******* Finished uf20-0812.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.59 s
 ******* Finished uf20-0905.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.22 s
```

```
 ******* Finished uf20-0935.cnf recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.19 s
    Time (seconds): 0.811169266701 +- 0.275506698238
```

- `resultados-factoring-super-restricted-confiltros`:

  Gramática SRTAS, ejecutada sobre las instancias *factoring* con los 3 criterios de ordenamiento de variables.

```
Run for General-MGIG , variable_ordering=johnson
 ******* Finished f004.dimacs recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.06 s
 ******* Finished f005.dimacs recognizer=General-MGIG. Correct: 1 / 1    Total time: 2.38 s
 ******* Finished f006.dimacs recognizer=General-MGIG. Correct: 1 / 1    Total time: 2.36 s
 ******* Finished f007.dimacs recognizer=General-MGIG. Correct: 1 / 1    Total time: 2.39 s
 ******* Finished f008.dimacs recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.68 s
    Time (seconds): 1.77403783798 +- 1.00703709941
Run for Right-D-MGIG , variable_ordering=johnson
 ******* Finished f004.dimacs recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.04 s
 ******* Finished f005.dimacs recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.76 s
 ******* Finished f006.dimacs recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.76 s
 ******* Finished f007.dimacs recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.74 s
 ******* Finished f008.dimacs recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.24 s
    Time (seconds): 1.30691156387 +- 0.744814036504
Run for Unif-RD-MGIG , variable_ordering=johnson
 ******* Finished f004.dimacs recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.05 s
 ******* Finished f005.dimacs recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.79 s
 ******* Finished f006.dimacs recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.80 s
 ******* Finished f007.dimacs recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.77 s
 ******* Finished f008.dimacs recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.30 s
    Time (seconds): 1.34056248665 +- 0.753381106561
Run for General-MGIG , variable_ordering=max
 ******* Finished f004.dimacs recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.05 s
 ******* Finished f005.dimacs recognizer=General-MGIG. Correct: 1 / 1    Total time: 2.33 s
 ******* Finished f006.dimacs recognizer=General-MGIG. Correct: 1 / 1    Total time: 2.34 s
 ******* Finished f007.dimacs recognizer=General-MGIG. Correct: 1 / 1    Total time: 2.30 s
 ******* Finished f008.dimacs recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.65 s
    Time (seconds): 1.73420934677 +- 0.986852110223
Run for Right-D-MGIG , variable_ordering=max
 ******* Finished f004.dimacs recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.03 s
 ******* Finished f005.dimacs recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.74 s
 ******* Finished f006.dimacs recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.80 s
 ******* Finished f007.dimacs recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.91 s
 ******* Finished f008.dimacs recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.21 s
    Time (seconds): 1.33798236847 +- 0.77781980825
Run for Unif-RD-MGIG , variable_ordering=max
 ******* Finished f004.dimacs recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.04 s
 ******* Finished f005.dimacs recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.53 s
 ******* Finished f006.dimacs recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.51 s
 ******* Finished f007.dimacs recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.48 s
 ******* Finished f008.dimacs recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.17 s
    Time (seconds): 1.14637351036 +- 0.633550527644
Run for General-MGIG , variable_ordering=min
 ******* Finished f004.dimacs recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.05 s
 ******* Finished f005.dimacs recognizer=General-MGIG. Correct: 1 / 1    Total time: 2.39 s
 ******* Finished f006.dimacs recognizer=General-MGIG. Correct: 1 / 1    Total time: 2.40 s
 ******* Finished f007.dimacs recognizer=General-MGIG. Correct: 1 / 1    Total time: 2.40 s
 ******* Finished f008.dimacs recognizer=General-MGIG. Correct: 1 / 1    Total time: 1.66 s
    Time (seconds): 1.78001852036 +- 1.01694097834
Run for Right-D-MGIG , variable_ordering=min
```

```
 ******* Finished f004.dimacs recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.03 s
 ******* Finished f005.dimacs recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.73 s
 ******* Finished f006.dimacs recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.79 s
 ******* Finished f007.dimacs recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.72 s
 ******* Finished f008.dimacs recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 1.22 s
    Time (seconds): 1.29866714478 +- 0.742515601215
Run for Unif-RD-MGIG , variable_ordering=min
 ******* Finished f004.dimacs recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.05 s
 ******* Finished f005.dimacs recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.91 s
 ******* Finished f006.dimacs recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.95 s
 ******* Finished f007.dimacs recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.89 s
 ******* Finished f008.dimacs recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.44 s
    Time (seconds): 1.44777059555 +- 0.809915728319
```

- `resultados-pigeonhole-super-restricted-confiltros`

  Gramática SRTAS, ejecutada sobre las instancias *pigeonhole* con los 3 criterios de ordenamiento de variables.

```
Run for General-MGIG , variable_ordering=johnson
 ******* Finished ph02 recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.01 s
 ******* Finished ph03 recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.05 s
 ******* Finished ph04 recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.38 s
 ******* Finished ph05 recognizer=General-MGIG. Correct: 1 / 1    Total time: 3.40 s
 ******* Finished ph06 recognizer=General-MGIG. Correct: 1 / 1    Total time: 33.37 s
    Time (seconds): 7.44026875496 +- 14.5620559876
Run for Right-D-MGIG , variable_ordering=johnson
 ******* Finished ph02 recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.00 s
 ******* Finished ph03 recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.04 s
 ******* Finished ph04 recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.30 s
 ******* Finished ph05 recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 2.59 s
 ******* Finished ph06 recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 25.96 s
    Time (seconds): 5.77854342461 +- 11.3322185327
Run for Unif-RD-MGIG , variable_ordering=johnson
 ******* Finished ph02 recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.01 s
 ******* Finished ph03 recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.04 s
 ******* Finished ph04 recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.26 s
 ******* Finished ph05 recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.36 s
 ******* Finished ph06 recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 6.86 s
    Time (seconds): 1.7064286232 +- 2.93227377701
Run for General-MGIG , variable_ordering=max
 ******* Finished ph02 recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.01 s
 ******* Finished ph03 recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.08 s
 ******* Finished ph04 recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.38 s
 ******* Finished ph05 recognizer=General-MGIG. Correct: 1 / 1    Total time: 3.35 s
 ******* Finished ph06 recognizer=General-MGIG. Correct: 1 / 1    Total time: 33.39 s
    Time (seconds): 7.44225940704 +- 14.5742403606
Run for Right-D-MGIG , variable_ordering=max
 ******* Finished ph02 recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.01 s
 ******* Finished ph03 recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.04 s
 ******* Finished ph04 recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.30 s
 ******* Finished ph05 recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 2.58 s
 ******* Finished ph06 recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 25.42 s
    Time (seconds): 5.66917123795 +- 11.0953506343
Run for Unif-RD-MGIG , variable_ordering=max
 ******* Finished ph02 recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.01 s
 ******* Finished ph03 recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.04 s
 ******* Finished ph04 recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.25 s
```

```
 ******* Finished ph05 recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.30 s
 ******* Finished ph06 recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 6.78 s
    Time (seconds): 1.675507164 +- 2.89972699938
Run for General-MGIG , variable_ordering=min
 ******* Finished ph02 recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.01 s
 ******* Finished ph03 recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.05 s
 ******* Finished ph04 recognizer=General-MGIG. Correct: 1 / 1    Total time: 0.37 s
 ******* Finished ph05 recognizer=General-MGIG. Correct: 1 / 1    Total time: 3.35 s
 ******* Finished ph06 recognizer=General-MGIG. Correct: 1 / 1    Total time: 33.60 s
    Time (seconds): 7.47564167976 +- 14.6715219171
Run for Right-D-MGIG , variable_ordering=min
 ******* Finished ph02 recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.00 s
 ******* Finished ph03 recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.04 s
 ******* Finished ph04 recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 0.30 s
 ******* Finished ph05 recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 2.61 s
 ******* Finished ph06 recognizer=Right-D-MGIG. Correct: 1 / 1    Total time: 25.69 s
    Time (seconds): 5.72887578011 +- 11.2134997371
Run for Unif-RD-MGIG , variable_ordering=min
 ******* Finished ph02 recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.01 s
 ******* Finished ph03 recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.04 s
 ******* Finished ph04 recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 0.25 s
 ******* Finished ph05 recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 1.32 s
 ******* Finished ph06 recognizer=Unif-RD-MGIG. Correct: 1 / 1    Total time: 6.75 s
    Time (seconds): 1.67509756088 +- 2.88952799384
```

Bibliografía

[1] Satisfiability Suggested Format. In *DIMACS Challenge*, 1993.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, techniques, and tools (2nd edition). pages 222–226. Addison Wesley, August 2006.

[3] John Aycock. Compiling little languages in python. In *Proceedings of the 7th International Python Conference*, 1998.

[4] G. E. Barton. Computational complexity in two-level morphology. In *Proc. of the 24th ACL*, pages 53–59, New York, 1986.

[5] T. Becker, A. Joshi, and O. Rambow. Long-distance scrambling and Tree Adjoining Grammars. In *Proceedings of the 5th Conference of the European Chapter of the ACL*, pages 21–26. Berlin, 1991.

[6] M. Bodirsky, M. Kuhlmann, and M. Möhl. Well-nested drawings as models of syntactic structure (extended version). Technical report, Programming Systems Lab, Saarland University, Saarbrücken, Germany, 2005.

[7] R. Book. Confluent and other types of thue systems. *J. Assoc. Comput. Mach.*, 29:171–182, 1982.

[8] J. Castaño. A parsing approach to sat. In Ana L.C. Bazzan and Karim Pichara, editors, *Advances in Artificial Intelligence – IBERAMIA 2014*, volume 8864 of *Lecture Notes in Computer Science*, pages 3–14. Springer International Publishing, 2014.

[9] J. M. Castaño. *Global Index Languages.* PhD thesis, Brandeis University, Dept of Computer Science, Waltham, MA, 2004.

[10] J. Castaño. Gigs: Restricted context-sensitive descriptive power in bounded polynomial-time. In *Proc. of Cicling 2003, Mexico City, February 16-22.*

[11] J. Castaño. Global index grammars and descriptive power. *Journal of Logic, Language and Information; 13:403–419*, 2004.

[12] J. Castaño. Two views on crossing dependencies, language, biology and satisfiability. In *1st International Work-Conference on Linguistics, Biology and Computer Science: Interplays*, 2011.

[13] J. Castaño. Crossing dependencies multi-pushdown languages and multiplanarity (en preparación). 2016.

[14] J. Castaño and R. Castaño. A finite state intersection approach to propositional satisfiability. *Theoretical Computer Science,Volume 450 Pages 92–108. Elsevier*, 2012.

[15] A. Cherubini, L. Breveglieri, C. Citrini, and S. Reghizzi. Multipushdown languages and grammars. *International Journal of Foundations of Computer Science*, 7(3):253–292, 1996.

[16] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[17] Rene de la Briandais. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference*, page 295–298, 1959.

[18] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM, 13:94–102*, 1970.

[19] Michael R. Garey and David S. Johnson. Computers and intractability; a guide to the theory of np-completeness. pages 175–176. W H Freeman & Co, 1979.

[20] Michael R. Garey and David S. Johnson. Computers and intractability; a guide to the theory of np-completeness. pages 27–32. W H Freeman & Co, 1979.

[21] S. Ginsburg and M. Harrison. One-way nondeterministic real-time list-storage. *Journal of the Association for Computing Machinery*, 15, No. 3:428–446, 1968.

[22] C. Gómez-Rodríguez, M. Kuhlmann, and G. Satta. Efficient parsing of well-nested linear context-free rewriting systems. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, pages 276–284, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[23] C. Gómez-Rodríguez and J. Nivre. A transition-based parser for 2-planar dependency structures. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pages 1492–1501, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[24] Pierre Hansen and Brigitte Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44(4):279–303, 1990.

[25] Holger H. Hoos and Thomas Stützle. Satlib: An online resource for research on sat. pages 283–292. IOS Press, 2000.

[26] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation - (2. ed.). Addison-Wesley series in computer science, pages 348–351. Addison-Wesley-Longman, 2001.

[27] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation - (2. ed.). Addison-Wesley series in computer science, pages 277–280. Addison-Wesley-Longman, 2001.

[28] A. Joshi. Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural description? In D. Dowty, L. Karttunen, and A. Zwicky, editors, *Natural language processing: psycholinguistic, computational and theoretical perspectives*, pages 206–250. Chicago University Press, New York, 1985.

[29] L. Kallmeyer. *Parsing Beyond Context-Free Grammars*. Springer, 2010.

[30] M. Kanazawa. The pumping lemma for well-nested multiple context-free languages. In Volker Diekert and Dirk Nowotka, editors, *Developments in Language Theory*, volume 5583 of *Lecture Notes in Computer Science*, pages 312–325. Springer Berlin Heidelberg, 2009.

[31] M. Kanazawa, J. Michaelis, S. Salvati, and R. Yoshinaka. Well-nestedness properly subsumes strict derivational minimalism. In S. Pogodalla and J.-P. Prost, editors, *LACL*, volume 6736, pages 112–128. Springer, 2011.

[32] M. Kanazawa and S. Salvati. The copying power of well-nested multiple context-free grammars. In A. Dediu, H. Fernau, and C. Martín-Vide, editors, *Language and Automata Theory and Applications*, volume 6031, pages 344–355. Lecture Notes in Computer Science, 2010.

[33] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, page 85–103. 1972.

[34] N. A. Khabbaz. A geometric hierarchy of languages. *Journal of Computer and System Sciences*, 8(2):142–157, 1974.

[35] S. Kübler, R. McDonald, and J. Nivre. *Dependency parsing*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2009.

[36] M. Kuhlmann. Mildly non-projective dependency grammar. *Comput. Linguist.*, 39(2):355–387, June 2013.

[37] M. Kuhlmann and J. Nivre. Mildly non-projective dependency structures. In *Proceedings of the COLING/ACL on Main Conference Poster Sessions*, COLING-ACL '06, pages 507–514, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics.

[38] P. Neuhaus and N. Broker. The complexity of recognition of linguistically adequate dependency grammars. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pages 337–343, Madrid, Spain, July 1997. Association for Computational Linguistics.

[39] E. S. Ristad. Computational complexity of current GPSG theory. In *Proc. of the 24th ACL*, pages 30–39, New York, 1986.

[40] G. Satta. Recognition of Linear Context-Free Rewriting Systems. In *ACL*, pages 89–95, 1992.

[41] G. Satta. Some computational complexity results for synchronous context-free grammars. In *In Proceedings of HLT/EMNLP-05*, pages 803–810, 2005.

[42] D.B. Searls. Formal language theory and biological macromolecules. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, page 117ff, 1999.

[43] H. Seki, T. Matsumura, M. Fujii, and T. Kasami. On multiple context-free grammars. *Theoretical Computer. Science*, pages 191–229, 1991.

[44] Carsten Sinz and Armin Biere. *Extended Resolution Proofs for Conjoining BDDs*, pages 600–611. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[45] E. P. Stabler. Varieties of crossing dependencies: structure dependence and mild context sensitivity. *Cognitive Science*, 28(5):699 – 720, 2004.

[46] M. Tomita. An efficient augmented-context-free parsing algorithm. *Computational linguistics, 13:31–46*, 1987.

[47] S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *In LICS*, pages 161–170. IEEE Computer Society, 2007.

[48] H. Vogler. Iterated linear control and iterated oneturn pushdowns. *Mathematical Systems Theory*, 19:117–133, 1986.

[49] D. A. Walters. Deterministic context-sensitive languages: Part II. *Information and Control*, 17:41–61, 1970.

[50] C. Wartena. Grammars with composite storages. In *Proceedings of the Conference on Logical Aspects of Computational Linguistics (LACL '98)*, pages 11–14, Grenoble, 1998.

[51] C. Wartena. On the concatenation of one-turn pushdowns. *Grammars*, pages 259–269, 1999.

[52] Christian Wartena. Storage products and linear control of derivations. *Theory Comput. Syst.*, 42(2):157–186, 2008.

[53] D. Weir. A geometric hierarchy beyond context-free languages. *Theoretical Computer Science*, 104(2):235–261, 1992.

[54] Anssi Yli-jyrä and Matti Nykänen. A hierarchy of mildly context-sensitive dependency grammars, 2004.

[55] H. Yuen and J. Bebel. Tough SAT Project. `http://toughsat.appspot.com/`, 2011. [Online; accessed 4-November-2016].