Estudio comparativo del rendimiento de analizadores sintácticos para gramáticas de adjunción de árboles*

Carlos Gómez-Rodríguez y **Miguel A. Alonso**
Departamento de Computación
Universidade da Coruña
Campus de Elviña, s/n
15071 A Coruña, Spain
{cgomezr, alonso}@udc.es

Manuel Vilares
E. S. de Ingeniería Informática
Universidad de Vigo
Campus As Lagoas, s/n
32004 Ourense, Spain
vilares@uvigo.es

Resumen: En este trabajo se estudia el comportamiento de los algoritmos de análisis sintáctico más utilizados en el tratamiento de las Gramáticas de Adjunción de Árboles (TAG). Para ello se aplica una técnica de compilación que permite la transformación automática de esquemas de análisis sintáctico en implementaciones eficientes de los algoritmos que describen, lo que nos permite comparar el rendimiento de diferentes analizadores en un entorno homogéneo. En nuestro estudio, analizamos los diferentes algoritmos tanto mediante la utilización de gramáticas generadas artificialmente, que nos permiten conocer la influencia del tamaño de la gramática en el rendimiento; como mediante la aplicación de las implementaciones generadas a una gramática real de amplia cobertura (la XTAG).
Palabras clave: Análisis sintáctico, esquemas de análisis sintáctico, gramáticas de adjunción de árboles

Abstract: In this paper, we study the behavior of some of the most popular parsing algorithms for Tree Adjoining Grammars (TAG). To this end, we apply a compilation technique allowing automatic transformation of parsing schemata into efficient implementations of their corresponding algorithms; which allows us to compare the performance of different parsers in an homogeneous environment. In our study, we analyze the different algorithms both by using artificially generated grammars, which allow us to estimate the influence of grammar size in performance, and by applying the generated implementations to a real-life, wide coverage grammar (the XTAG).
Keywords: Parsing, parsing schemata, tree adjoining grammars

1. Introducción

Las Gramáticas de Adjunción de Árboles (TAG) son una extensión de las gramáticas independientes del contexto en la que la estructura primaria de representación es el árbol, lo que las hace más adecuadas para la descripción de los fenómenos sintácticos presentes en los lenguajes naturales o lenguas.

En las últimas décadas se han desarrollado un gran número de analizadores sintácticos para este formalismo. Entre los más utilizados están el algoritmo basado en CYK descrito en (Vijay-Shanker y Joshi, 1985), un algoritmo basado en Earley sin la propiedad del prefijo válido (VPP) — (Schabes, 1994) y (Alonso et al., 1999) —, un algoritmo basado en Earley con la propiedad del prefijo válido (Alonso et al., 1999) y el algoritmo de Nederhof (Nederhof, 1999).

Es curioso constatar la práctica ausencia de trabajos en los que se realice un análisis comparativo entre varios de estos algoritmos. Los pocos que existen, como (Díaz y Alonso, 2000), trabajan con una gramática fija y pequeña, con lo cual no permiten apreciar cómo cambia el comportamiento de los algoritmos al aumentar el tamaño de la gramática y de la cadena de entrada.

* Parcialmente financiado por Ministerio de Educación y Ciencia y FEDER (TIN2004-07246-C03-01, TIN2004-07246-C03-02), Xunta de Galicia (PGIDIT05PXIC30501PN, PGIDIT05PXIC10501PN y PGIDIT05SIN044E), y Programa de becas FPU (Ministerio de Educación y Ciencia).

El objeto de este trabajo es precisamente cubrir este vacío. Para ello realizaremos un estudio del comportamiento de los analizadores antes citados, tanto sobre gramáticas artificiales como sobre una gramática real de amplia cobertura como es la XTAG (XTAG, 2001). Nótese que la ventaja de usar gramáticas artificialmente generadas es que podemos ver fácilmente la influencia del tamaño de la gramática en el rendimiento, dado que el número de árboles elementales de la gramática es un parámetro que podemos hacer variar libremente, cosa imposible si se utiliza una gramática de un corpus real de lenguaje natural. Así, la comparación de algoritmos mediante gramáticas artificiales complementa a la que más tarde se hará con la gramática de uso práctico XTAG.

Para que los resultados sean comparables, se ha dispuesto un entorno homogéneo basado en los esquemas de análisis sintáctico que describen el comportamiento de cada algoritmo. Mediante la técnica de compilación descrita en (Gómez-Rodríguez et al., 2006a; Gómez-Rodríguez et al., 2005), estos esquemas son transformados automáticamente en implementaciones eficientes de los algoritmos que describen.

2. Generación de los analizadores

El formalismo de los esquemas de análisis sintáctico (Sikkel, 1997) permite describir analizadores sintácticos de una manera sencilla y declarativa, facilitando así su diseño, análisis y comparación. Un esquema de análisis sintáctico es una representación de un algoritmo de análisis sintáctico como una función que, dada una gramática de una clase determinada, le hace corresponder un conjunto de reglas de inferencia (llamadas pasos deductivos) que se utilizan para llevar a cabo deducciones sobre resultados intermedios denominados ítems. Dichos ítems representan conjuntos de árboles de análisis incompletos que el algoritmo puede generar. Dada una cadena de entrada que se quiere analizar, independientemente del algoritmo o esquema concreto que se utilice, se puede definir un conjunto de ítems iniciales o hipótesis a partir de las palabras de la cadena. Adicionalmente, cada esquema de análisis sintáctico debe definir un criterio para determinar qué ítems son finales, es decir, corresponden a un análisis completo de la cadena de entrada. Dado este criterio, si es posible obtener algún ítem final a partir del conjunto de ítems iniciales utilizando los pasos deductivos, la cadena de entrada pertenece al lenguaje definido por la gramática. En este caso, los ítems intermedios que se han utilizado para deducir el ítem final nos proporcionan una traza del análisis de la que se puede obtener fácilmente el bosque de análisis, tal y como se describe en (Billot y Lang, 1989).

Su sencillez y abstracción de los detalles de bajo nivel hace que los esquemas de análisis sintáctico resulten muy útiles, permitiéndonos definir algoritmos de análisis de manera sencilla y haciendo más fácil compararlos o considerar aspectos como su corrección, completitud y complejidad computacional.

Sin embargo, el problema de los esquemas de análisis sintáctico es que, aunque son muy útiles cuando se trabaja sobre el papel, no pueden ejecutarse directamente en un computador. Para ejecutar los analizadores y comprobar sus resultados y rendimiento no queda más remedio que implementarlos en un lenguaje de programación, abandonando el alto nivel de abstracción de los esquemas.

Para salvar esta dificultad, hemos diseñado e implementado una técnica de compilación que permite transformar automáticamente los esquemas de análisis sintáctico en implementaciones ejecutables en lenguaje Java de los algoritmos correspondientes. La entrada al sistema es una representación simple y declarativa de un esquema de análisis, que prácticamente coincide con la notación formal de (Sikkel, 1997). Por ejemplo, el paso del algoritmo de tipo CYK que se define

$$\frac{[O_1^\gamma, i, j', p, q, adj1] \quad [O_2^\gamma, j', j, p', q', adj2]}{[M^\gamma, i, j, p \cup p', q \cup q', false]}\ M^\gamma \rightarrow O_1^\gamma O_2^\gamma \in P(G)$$

se podría representar así:

```
@step CYKBinary
[ N1 , i , j' , p , q , adj1 ]
[ N2 , j' , j , p' , q' , adj2 ]
---------------------------------------- N3 -> N1 N2
[ N3 , i , j , Union(p;p') , Union(q;q') , false ]
```

La técnica de compilación que permite compilar estos esquemas se basa en las siguientes ideas fundamentales (Gómez-Rodríguez et al., 2006a; Gómez-Rodríguez et al., 2005):

- Cada paso deductivo se compila a una clase Java que contiene código para emparejar y buscar ítems del antecedente y generar las conclusiones correspondientes a partir del consecuente.

- Las clases correspondientes a los pasos son coordinadas por una máquina deductiva de análisis sintáctico como la que se describe en (Shieber, Schabes, y Pereira, 1995). Este algoritmo garantiza un proceso deductivo correcto y completo, garantizando que se obtendrán todos los ítems que sea posible generar a partir de los ítems iniciales.
- Para garantizar la eficiencia de las implementaciones generadas, se lleva a cabo un análisis automático de los esquemas de entrada para crear índices de acceso rápido a los ítems. Como cada esquema de análisis necesita llevar a cabo búsquedas distintas para encontrar ítems que emparejen con los antecedentes, las estructuras de índices generadas son específicas para cada esquema. De este modo, garantizamos acceso en tiempo constante a los ítems, de manera que la complejidad computacional de las implementaciones generadas nunca estará por encima de la complejidad teórica de los algoritmos correspondientes.
- Dado que la notación de los esquemas de análisis es abierta, pudiendo aparecer en sus ítems cualquier objeto matemático, el sistema incluye un mecanismo de extensibilidad que se puede utilizar para definir nuevos tipos de objetos a utilizar en los esquemas. El generador de código puede manejar estos objetos definidos por el usuario siempre que éstos se especifiquen siguiendo unas pautas determinadas.

3. *Estudio con gramáticas artificialmente generadas*

Dado un entero $k > 0$, definimos la gramática de adjunción de árboles $G_k = (V_T, V_N, S, I, A)$ [1] con $V_T = \{a_j | 0 \leq j \leq k\}$, $V_N = \{S, B\}$, y

$I = \{S(B(a_0))\}$[2],

$A = \{B(B(B^* \ \ a_j)) | 1 \leq j \leq k\}$.

Por lo tanto, para un k dado, G_k es una gramática con un árbol inicial y k árboles auxiliares, que define un lenguaje sobre un alfabeto con $k+1$ símbolos terminales. El lenguaje definido por G_k es concretamente el lenguaje regular $L_k = a_0(a_1|a_2|..|a_k)^*$[3]. Cabe destacar que, aunque los lenguajes L_k sean triviales, las gramáticas G_k están construidas de tal modo que cualquiera de sus árboles auxiliares puede adjuntarse en cualquier otro, cosa que las hace adecuadas para llevar a cabo un análisis empírico de la complejidad en el peor caso.

Las siguientes tablas muestran los tiempos de ejecución en segundos[4] de los cuatro analizadores, para distintos valores de la longitud de la cadena de entrada (n) y el tamaño de la gramática (el parámetro k de las gramáticas G_k que hemos definido). El algoritmo con mejor resultado para cada caso se muestra en negrita.

Tiempos en s: tipo CYK					
n=	Tamaño de la Gramática (k)				
	1	8	64	512	4096
2	∼0	∼0	0,02	1,34	125,75
4	∼0	∼0	0,06	4,11	290,19
8	0,02	0,03	0,23	15,89	777,97
16	**0,02**	**0,06**	0,78	44,19	2,247,16
32	**0,09**	**0,31**	3,78	170,61	
64	**0,27**	**2,06**	25,09	550,02	
128	**1,19**	14,52	269,05		
256	**6,78**	108,30			
512	**52,00**				

Tiempos en s: tipo Earley sin VPP					
n=	Tamaño de la Gramática (k)				
	1	8	64	512	4096
2	∼0	0,02	0,01	**1,16**	**109,84**
4	∼0	0,03	0,06	**2,58**	**256,09**
8	0,02	0,03	**0,17**	**6,89**	**589,58**
16	0,03	0,17	**0,62**	**18,73**	**1.508,61**
32	0,11	0,61	**3,22**	**69,41**	
64	0,48	2,95	**22,45**	**289,98**	
128	2,03	**13,87**	**234,59**		
256	10,00	**101,22**			
512	61,27				

Tiempos en s: tipo Earley con VPP					
n=	Tamaño de la Gramática (k)				
	1	8	64	512	4096
2	∼0	∼0	0,03	1,94	194,05
4	∼0	0,02	0,08	4,08	453,20
8	0,01	0,03	0,23	10,92	781,14
16	0,03	0,19	0,87	27,12	1,787,14
32	0,12	0,75	4,14	98,83	
64	0,58	3,55	28,64	350,22	
128	2,45	20,77	264,50		
256	12,19	122,80			
512	74,05				

[1] Donde V_T denota el conjunto de símbolos terminales, V_N el conjunto de símbolos no terminales, S el axioma, I el conjunto de árboles iniciales y A el conjunto de árboles auxiliares.

[2] Donde escribimos los árboles con notación de paréntesis, y * denota el nodo pie.

[3] Además, se puede demostrar fácilmente que la gramática G_k es una de las gramáticas de adjunción de árboles mínimas (en términos de número de árboles) cuyo lenguaje asociado es L_k. Nótese que necesitamos al menos un árbol que contenga a_0 como único terminal para poder analizar la frase a_0, y para cada $1 \leq i \leq k$, necesitamos al menos un árbol que contenga a_i y ningún otro a_j $(j > 0)$ para poder analizar la frase $a_0 a_i$. Por lo tanto, cualquier TAG para el lenguaje L_k debe tener al menos $k+1$ árboles elementales.

[4] La máquina utilizada para todas las pruebas fue un Intel Pentium 4 a 3.40 GHz, con 1 GB de RAM y la máquina virtual Java Sun Hotspot (version 1.4.2_01-b06) sobre Windows XP.

Tiempos en s: algoritmo de Nederhof					
n=	Tamaño de la Gramática (k)				
	1	8	64	512	4096
2	∼0	∼0	0,05	1,87	151,53
4	∼0	0,01	0,19	4,56	390,47
8	0,01	0,03	0,47	12,53	998,59
16	0,05	0,19	1,50	40,09	2,579,58
32	0,22	0,95	6,23	157,06	
64	1,08	4,73	35,86	620,05	
128	5,70	25,70	302,77		
256	37,12	159,61			
512	291,14				

A partir de estos resultados, podemos observar que ambos factores (longitud de la cadena y tamaño de la gramática) tienen influencia en el tiempo de ejecución, y que interaccionan entre ellos: las tasas de crecimiento con respecto a uno de los factores se ven influidas por el otro factor, así que resulta difícil dar estimaciones precisas de la complejidad computacional empírica. Sin embargo, podemos obtener estimaciones aproximadas si nos centramos en los casos donde uno de los factores toma valores altos y el otro valores bajos (dado que en estos casos los factores constantes que afectan a la complejidad serán menores) y comprobarlas estudiando si la serie $T(n,k)/f(n)$ parece converger a una constante positiva por cada valor fijo de k (si $f(n)$ es nuestra estimación de complejidad con respecto a n) o si $T(n,k)/f(k)$ parece converger a una constante positiva para cada valor fijo de n (si $f(k)$ es una estimación de complejidad con respecto a k).

Aplicando estas ideas, vemos que la complejidad temporal empírica con respecto a la longitud de la cadena de entrada está en el rango entre $O(n^{2,8})$ y $O(n^3)$ para los algoritmos tipo CYK y Nederhof, y entre $O(n^{2,6})$ y $O(n^3)$ para los basados en Earley con y sin la propiedad del prefijo válido (VPP). Por lo tanto, la complejidad temporal práctica que obtenemos está muy por debajo de las cotas teóricas en el peor caso para estos algoritmos, que son de $O(n^6)$ (excepto para el algoritmo tipo Earley con la VPP, que es $O(n^7)$).

Aunque por motivos de espacio no incluimos tablas con el número de ítems generados en cada caso, nuestros resultados muestran que la complejidad espacial empírica con respecto a la longitud de la cadena de entrada es aproximadamente $O(n^2)$ para todos los algoritmos, también muy por debajo de las cotas para el peor caso ($O(n^4)$ y $O(n^5)$).

Con respecto al tamaño de la gramática, obtenemos una complejidad temporal de aproximadamente $O(|I \cup A|^2)$ para todos los algoritmos. Esto se corresponde con la cota teórica del peor caso, que es $O(|I \cup A|^2)$ debido a los pasos de adjunción, que trabajan con pares de árboles. En el caso de nuestras gramáticas artificialmente generadas, cualquier árbol auxiliar puede ser adjuntado en cualquier otro, así que es lógico que nuestros tiempos crezcan de forma cuadrática. Nótese, en todo caso, que gramáticas reales como la XTAG (XTAG, 2001) tienen pocos símbolos no terminales diferentes en relación con su número de árboles, así que muchos pares de árboles son susceptibles de adjunción y no podemos esperar que su comportamiento sea mucho mejor que el que vemos aquí.

La complejidad espacial con respecto al tamaño de la gramática es aproximadamente $O(|I \cup A|)$. Éste es un resultado esperado, dado que cada ítem que se genera está asociado a un nodo de un árbol dado.

Las aplicaciones prácticas de las TAG en procesamiento del lenguaje natural suelen caer en el rango de valores para n y k que hemos cubierto en los experimentos (se usan gramáticas con cientos o unos pocos miles de árboles para analizar frases de unas pocas docenas de palabras). En estos rangos, tanto la longitud de la cadena como el tamaño de la gramática toman valores significativos y tienen mucha influencia en los tiempos de ejecución, como podemos ver en los resultados de las tablas. Esto nos lleva a destacar que el análisis tradicional basado en la complejidad con respecto a un solo factor (longitud de la cadena o tamaño de la gramática) puede resultar insuficiente en aplicaciones prácticas, llevándonos a una idea incompleta de la complejidad real. Por ejemplo, si trabajamos con una gramática de miles de árboles, el tamaño de la gramática es el factor más influyente, y el uso de técnicas de filtrado (Schabes y Joshi, 1991) para reducir la cantidad de árboles utilizada en el análisis es esencial para conseguir un buen rendimiento. Por otra parte, la influencia de la cadena de entrada en estos casos se ve suavizada por los grandes factores constantes relativos al tamaño de la gramática. Por ejemplo, en los tiempos para la gramática G_{4096} podemos ver que los tiempos se multiplican por un factor menor que 3 cuando se duplica la longitud de la cadena, aunque el resto de los resultados nos han llevado a concluir que la complejidad asintótica práctica con respecto a este parámetro es de al menos $O(n^{2,6})$. Es-

tas interacciones entre ambos factores se deben tener en cuenta al analizar el rendimiento en términos de complejidad computacional. Un análisis más detallado puede encontrarse en (Gómez-Rodríguez et al., 2006b).

Los algoritmos basados en Earley consiguen mejores tiempos de ejecución que el basado en CYK para gramáticas grandes, aunque son peores para gramáticas pequeñas. Nótese que en gramáticas independientes del contexto pasa lo contrario (Gómez-Rodríguez et al., 2005): CYK funciona mejor para gramáticas grandes porque es lineal con respecto al tamaño de la gramática. En el caso de las TAG, sin embargo, ambos son cuadráticos y el algoritmo CYK pierde esta ventaja.

El algoritmo basado en CYK genera menos ítems que los basados en Earley para gramáticas grandes y cadenas cortas, pero genera más para gramáticas pequeñas y cadenas largas. El algoritmo tipo Earley con VPP genera la misma cantidad de ítems que el que no tiene esta propiedad, y tiene peores tiempos de ejecución. El motivo es que en el caso particular de esta gramática no se generan análisis parciales que violen la propiedad del prefijo válido, así que garantizar esta propiedad no evita la generación de ningún ítem. Por lo tanto, el hecho de que la variante sin VPP funcione mejor en este caso concreto no puede ser extrapolado a otras gramáticas. No obstante, las diferencias en tiempo entre estos dos algoritmos ilustran la sobrecarga causada por las comprobaciones adicionales necesarias para garantizar la VPP en un caso particularmente malo.

El algoritmo de Nederhof presenta tiempos de ejecución más lentos que las otras variantes de Earley. A pesar de que el algoritmo de Nederhof es una mejora sobre el algoritmo de Earley con VPP en términos de complejidad computacional, los pasos deductivos adicionales que contiene lo hacen más lento en la práctica.

4. Estudio con la XTAG

La XTAG (XTAG, 2001) es una gramática de adjunción de árboles lexicalizada de amplia cobertura para el idioma inglés; que cuenta con más de mil árboles elementales y es una FBTAG (Feature Based Tree Adjoining Grammar), con lo cual sus nodos están decorados con estructuras de rasgos.

Si extendemos los esquemas de análisis sintáctico que aparecen en (Alonso et al., 1999; Nederhof, 1999) con soporte para unificación de estructuras de rasgos, los analizadores generados pueden usarse con la XTAG (Gómez-Rodríguez et al., 2006c), aunque algunos otros añadidos, como el filtrado de árboles, son muy convenientes si queremos obtener un rendimiento aceptable con esta gramática.

En esta sección describimos cómo hemos aplicado a la XTAG nuestra técnica de compilación genérica, y proporcionamos una comparación de los cuatro algoritmos vistos con anterioridad en su aplicación al caso práctico de la XTAG. Cabe destacar que ya se han hecho comparaciones similares sobre gramáticas más pequeñas, como subconjuntos simplificados de la XTAG, pero no con la XTAG completa con todos sus árboles y estructuras de rasgos. Hemos visto en la sección anterior que el tamaño de la gramática tiene una gran influencia en el rendimiento, y que no se puede esperar que el comportamiento observado con gramáticas pequeñas sea extrapolable a gramáticas grandes; por lo tanto, resulta interesante poder comparar los distintos analizadores directamente sobre una gramática real como la XTAG.

4.1. Soporte de unificación

Los árboles de la XTAG incluyen estructuras de rasgos asociados a cada uno de sus nodos. Las estructuras de rasgos contienen información sobre cómo los nodos pueden interactuar entre sí, y pueden imponer restricciones sobre las operaciones de adjunción y substitución, dado que los rasgos de los nodos que intervienen deben unificar.

Se pueden utilizar dos estrategias a la hora de tener en cuenta la unificación en el análisis sintáctico: se puede unificar las estructuras de rasgos después del análisis o durante el análisis. La primera estrategia consiste en analizar la frase igual que en una gramática sin rasgos, sin hacer unificaciones, y después recuperar el bosque de análisis y llevar a cabo la unificación sólo sobre los árboles sintácticos finales, eliminando aquéllos que violen las restricciones de unificación. La segunda estrategia consiste en llevar a cabo unificaciones como parte del proceso de análisis, de modo que las estructuras de rasgos son unificadas en los pasos deductivos que implementan operaciones como la adjunción o sustitución.

A priori, cada una de las estrategias tiene

ventajas e inconvenientes: la unificación durante el análisis considera lo antes posible las restricciones impuestas por los rasgos, evitando la generación de árboles innecesarios que violen dichas restricciones. Sin embargo, esta estrategia lleva a cabo unificaciones sobre todos los ítems que genera, mientras que si unificamos al terminar el análisis sólo necesitamos considerar los ítems finales y los que conducen a ellos en el proceso deductivo.

La conveniencia de uno u otro método dependerá del modo en que las estructuras de rasgos se usen en la gramática. En el caso particular de la XTAG, al comparar las dos estrategias en la práctica (véase figura 1), llegamos a la conclusión general de que la unificación durante el análisis funciona mejor para la gran mayoría de las frases, aunque sus tiempos de ejecución tienen mayor varianza y son mucho peores para algunos casos particulares. Para la comparación entre algoritmos hemos utilizado esta estrategia de unificación durante el análisis.

Para implementar la unificación durante el análisis en nuestro sistema basado en esquemas de análisis sintáctico, debemos extender los esquemas para que lleven a cabo las comprobaciones de unificación. Esto implica extender los ítems para que almacenen estructuras de rasgos, definir operaciones para unificar y para eliminar información innecesaria de las estructuras de rasgos en los ítems, y utilizar dichas operaciones en los pasos deductivos. La operación de unificación se debe utilizar siempre que un paso deductivo maneje varias estructuras de rasgos que se refieran al mismo nodo, o a nodos que se van a unir en uno solo en el árbol de análisis al intervenir en una sustitución o adjunción. En estos últimos casos, hay que tener en cuenta el diferente rol que juegan las estructuras de rasgos "top" y "bottom" en estas operaciones, tal y como se describe en (XTAG, 2001). La operación de eliminación de información innecesaria se utiliza para que los ítems generados por un paso deductivo sólo propaguen información sobre las variables del árbol al que hacen referencia, y no otra información que se haya utilizado en la unificación.

Al extender los algoritmos tipo Earley con el soporte de unificación, nos encontramos con dos posibles opciones para los pasos deductivos predictivos: podemos tener predictores que propaguen las estructuras de rasgos o que las descarten. Cada una de las dos estrategias tiene ventajas e inconvenientes (la primera puede detectar antes el incumplimiento de algunas restricciones y descartar caminos inútiles del análisis, pero también puede generar más ítems en algunos casos). En la práctica y en el caso concreto de la XTAG, nuestros experimentos muestran que funciona más rápido la estrategia que descarta las estructuras de rasgos en los pasos predictivos, por lo que será la que utilicemos en la comparación.

4.2. Filtrado de árboles

La gramática XTAG completa contiene más de mil árboles elementales, así que los tiempos de ejecución obtenidos no serán buenos si utilizamos la gramática completa para analizar cada frase (véanse los resultados para gramáticas de ese orden en la sección 3). Los filtros de selección de árboles (Schabes y Joshi, 1991) permiten seleccionar un subconjunto de la gramática, descartando los árboles que se sabe que no resultarán útiles a la vista de las palabras de la cadena de entrada.

Para emular esta funcionalidad en nuestro sistema basado en esquemas, añadimos a los esquemas un paso de filtrado que genera ítems que indican que un árbol ha sido seleccionado, y estos ítems se utilizan como antecedentes en todos los pasos del esquema que introduzcan un árbol nuevo en el análisis.

4.3. Comparación de diferentes algoritmos sobre la XTAG

En esta sección hacemos una comparación de los algoritmos de análisis estudiados sobre la gramática XTAG, utilizando nuestro sistema y las ideas que hemos explicado. Los esquemas para estos algoritmos sin soporte de unificación, que pueden encontrarse en (Alonso et al., 1999), fueron extendidos como se ha descrito en las subsecciones anteriores, y utilizados como entrada para nuestro sistema que generó los analizadores correspondientes. Estos analizadores fueron ejecutados más tarde sobre las frases de prueba de la figura 2, obteniéndose las medidas de rendimiento (tiempo de ejecución y cantidad de ítems generados) que se pueden ver en la figura 3. Nótese que las frases están ordenadas por tiempo de ejecución mínimo.

El algoritmo más eficiente en términos de utilización de memoria es el algoritmo basado en CYK. Por otra parte, si comparamos tiempos de ejecución, no hay un algorit-

Strategy	Media	Media R. 10 %	Media R. 20 %	1er Cuart.	Mediana	3er Cuart.	Desv. Típ.	Wilcoxon
Durante	108,270	12,164	7,812	1,585	4,424	9,671	388,010	0,4545
Después	412,793	10,710	10,019	2,123	9,043	19,073	14,235	

Figura 1: Tiempos de ejecución en segundos de un analizador basado en Earley con dos estrategias distintas de unificación: unificación durante y después del análisis. Se muestran: medias, medias recortadas (10 y 20 %), cuartiles, desviación típica, y p-valor para el test de Wilcoxon (el valor de 0.4545 indica que no se ha encontrado diferencia estadísticamente significativa entre las medianas).

1. He was a cow	9. He wanted to go to the city
2. He loved himself	10. That woman in the city contributed to this article
3. Go to your room	11. That people are not really amateurs at intelectual duelling
4. He is a real man	12. The index is intended to measure future economic performance
5. He was a real man	13. They expect him to cut costs throughout the organization
6. Who was at the door	14. He will continue to place a huge burden on the city workers
7. He loved all cows	15. He could have been simply being a jerk
8. He called up her	16. A few fast food outlets are giving it a try

Figura 2: Frases de prueba obtenidas de la distribución de la XTAG.

mo claramente mejor, dado que los resultados dependen del tamaño y complejidad de las frases. El algoritmo basado en Earley con la VPP es el más rápido para las frases más sencillas, mientras que CYK proporciona los mejores resultados para las más complejas. Nótese que en este caso, al contrario que en las gramáticas artificiales, el algoritmo con la VPP obtiene mejores resultados que el que no tiene esta propiedad en muchas de las frases. Esto se debe a que las comprobaciones asociadas a la propiedad del prefijo válido pueden evitar la generación de algunos ítems, cosa que como comentamos no sucedía en el caso particular de las gramáticas artificiales estudiadas anteriormente.

El algoritmo de Nederhof tiene tiempos de ejecución más lentos que los demás, a pesar de ser una mejora del algoritmo basado en Earley con la VPP que reduce la complejidad temporal. La diferencia en tiempo es en este caso aún mayor que con las gramáticas artificiales debido al impacto de la unificación: la extensión de este algoritmo para soportar la unificación durante el análisis necesita llevar a cabo más operaciones de unificación que los otros algoritmos tipo Earley. Probablemente este algoritmo concreto se vería beneficiado si se utilizase la estrategia que sólo lleva a cabo unificaciones al final del análisis.

En general, la variabilidad en los tiempos para las distintas frases es grande, y no se da una correlación clara entre tiempos de ejecución y longitud de la entrada. Esto se debe al efecto del filtrado, que hace que en cada frase trabajemos con un subconjunto distinto de la gramática, y la gran influencia del tamaño de la gramática (en este caso, de dichos subconjuntos) en el rendimiento, que vimos en la comparación con las gramáticas artificiales.

5. Conclusiones

La utilización de un sistema que transforma esquemas de análisis sintáctico en implementaciones ejecutables de los algoritmos que describen nos ha permitido comparar el rendimiento de varios algoritmos de análisis para gramáticas de adjunción de árboles (TAG).

Esta comparación se ha llevado a cabo en dos planos: por una parte, hemos utilizado gramáticas artificiales para ponderar la influencia de la longitud de la cadena y el tamaño de la gramática en el tiempo de ejecución, llegando a la conclusión de que ambos factores son significativos en la práctica e interaccionan entre sí. Por otra parte, hemos comparado los algoritmos en el caso práctico de la XTAG, constatando que la elección del mejor algoritmo se debe hacer en función del tamaño de las frases que se quieran analizar, y que la variabilidad de los tiempos de ejecución es grande debido principalmente al uso de técnicas de filtrado.

Bibliografía

Alonso, M. A., D. Cabrero, E. de la Clergerie, y

Frase	Tiempos en ms				Ítems generados			
	Algoritmo				Algoritmo			
	CYK	Ear. s/ VPP	Ear. VPP	Neder.	CYK	Ear. s/ VPP	Ear. VPP	Neder.
1	2985	**750**	**750**	2719	1341	1463	**1162**	1249
2	3109	1562	**1219**	6421	**1834**	2917	2183	2183
3	4078	1547	**1406**	6828	**2149**	2893	2298	2304
4	4266	1563	**1407**	4703	1864	1979	**1534**	2085
5	4234	1921	**1421**	4766	1855	1979	**1534**	2085
6	4485	1813	**1562**	7782	**2581**	3587	2734	2742
7	5469	2359	**2344**	11469	**2658**	3937	3311	3409
8	7828	4906	**3563**	15532	**4128**	8058	4711	4716
9	10047	4422	**4016**	18969	**4931**	6968	5259	5279
10	13641	**6515**	7172	31828	**6087**	8828	7734	8344
11	16500	**7781**	15235	56265	**7246**	12068	13221	13376
12	16875	17109	**9985**	39132	**7123**	10428	9810	10019
13	25859	**12000**	20828	63641	**10408**	12852	15417	15094
14	54578	**35829**	57422	178875	**20760**	31278	40248	47570
15	**62157**	113532	109062	133515	**22115**	37377	38824	59603
16	**269187**	3122860	3315359		**68778**	152430	173128	

Figura 3: Tiempos de ejecución y cantidad de ítems generados por diferentes analizadores para la XTAG sobre varias frases. La máquina usada para los tests es un Intel Pentium 4 a 3.40 GHz, con 1 GB de RAM y la máquina virtual Sun Java Hotspot (version 1.4.2_01-b06) sobre Windows XP. Los mejores resultados para cada frase se resaltan en negrita.

M. Vilares. 1999. Tabular algorithms for TAG parsing. En *Proc. of EACL'99*, pp. 150–157, Bergen, Norway. ACL.

Billot, S. y B.Lang. 1989. The structure of shared forest in ambiguous parsing. En *Proc. of ACL'89*, pp. 143–151, Vancouver, Canada.

Díaz, V. J. y M. A. Alonso. 2000. Comparing tabular parsers for tree adjoining grammars. En David S. Warren M. Vilares L. Rodríguez Liñares, y M.A. Alonso, editores, *Proc. of Tabulation in Parsing and Deduction (TAPD 2000)*, pp. 91–100, Vigo, Spain.

Gómez-Rodríguez, C., J. Vilares, y M.A. Alonso. 2005. Generación automática de analizadores sintácticos a partir de esquemas de análisis. *Procesamiento del Lenguaje Natural*, 35:401–408, 2005.

Gómez-Rodríguez, C., J. Vilares y M. A. Alonso. 2006a. Automatic Generation of Natural Language Parsers from Declarative Specifications. in *Third European Starting AI Researcher Symposium (STAIRS 2006)*, Riva del Garda, Italy, August 28-29, 2006. IOS Press, Amsterdam, 2006.

Gómez-Rodríguez, C., M. A. Alonso y M. Vilares. 2006b. On Theoretical and Practical Complexity of TAG Parsers. in *Proc. of The 11th conference on Formal Grammar (FG 2006)*, Malaga, Spain, 2006.

Gómez-Rodríguez, C., M. A. Alonso y M. Vilares.2006c. Generating XTAG Parsers from Algebraic Specifications. in em Proc. of The Eighth International Workshop on Tree Adjoining Grammar and Related Formalisms (TAG+8), Sydney, Australia, 2006.

Nederhof, M.-J. 1999. The computational complexity of the correct-prefix property for TAGs. *Computational Linguistics*, 25(3):345–360.

Schabes, Y. 1994. Left to right parsing of lexicalized tree-adjoining grammars. *Computational Intelligence*, 10(4):506–515.

Schabes, Y. y A.K. Joshi. 1991. Parsing with lexicalized tree adjoining grammar. En M. Tomita, editor, *Current Issues in Parsing Technologies*. Kluwer Academic Publishers, Norwell, MA, cap. 3, pp. 25–47.

Shieber, S.M., Y. Schabes, y F. C. N. Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36, July-August.

Sikkel, K. 1997. *Parsing Schemata — A Framework for Specification and Analysis of Parsing Algorithms*. Springer-Verlag, Berlin/Heidelberg/NY.

Vijay-Shanker, K. y A.K. Joshi. 1985. Some computational properties of tree adjoining grammars. En *23rd Annual Meeting of the ACL*, pp. 82–93, Chicago, IL, Julio. ACL.

XTAG Research Group. 2001. A lexicalized tree adjoining grammar for English. Informe Técnico IRCS-01-03, IRCS, Univ. of Pennsylvania.