Top2Bottom Parser

David Santiago Flórez Alsina* and Laura Sofía Ortiz Arcos†
Universidad del Rosario, Escuela de Ingeniería, Ciencia y Tecnología, Bogotá, Colombia
(Dated: 12/09/2022)

Resumen

En este articulo se muestra el análisis aplicado a nuestra solución para el problema de parsing Top-down. Lo anterior, por medio de algunos algoritmos de busqueda vistos en clase, los cuales fueron: Breadth first search, Depth limited search, Best first search y por último Greedy search. Finalmente, se realizo una comparación de los tiempos que cada algoritmo tomaba para resolver el problema, en donde se pudo observar que los dos mejores algoritmos en la mayoría de los análisis, son el Greedy search y Best first search, aunque si hablamos únicamente de palabras con longitudes pequeñas, el Depth search suele ganarles.

Keywords: *Best first search, Greedy search, Bread first search, Depth limited search, Parsing tree and Parsing top-down.*

I. INTRODUCCIÓN

Una gramatica se define como un conjunto de reglas de sustitución de la forma:

variable → cadena de variables y terminales

Existen diferentes tipos de gramática, pero para nuestro problema usualmente se tiene en cuenta: La gramática independiente del contexto (CFG), la cuál es una cuádrupla $G = (V, \Sigma, S, P)$, donde:

- V = Es el alfabeto de variables o símbolos no terminales.
- Σ = Alfabeto, disyunto de V, cuyos elementos son símbolos terminales.
- S = Es la variable inicial, $S \in V$
- P = Es el conjunto finito de reglas, donde cada regla esta asociando una variable a una cadena de variables y terminales (A → w).

Sea G una gramática independiente libre de contexto y suponga w, tal que $w \in L(G)$, donde L(G) es el lenguaje de la gramática, es decir las palabras que puede generar la gramática. El problema de parsing para w es encontrar un árbol de análisis de acuerdo a las reglas que contiene G [1].

En otras palabras, el problema de parsing se puede ver como una función cuyo input va a ser una palabra (secuencia de símbolos), y el output un árbol de análisis de la palabra.

Existen dos maneras de resolver este problema, pero para nuestro análisis solo vamos a considerar el definido como, Top-down parsing, este tiene como objetivo comenzar con la variable inicial de la gramática e ir reescribiéndolo mediante reglas que permitan generar a w.

A. Ejemplos del problema

Para evaluar nuestra solución hicimos 4 gramáticas, los siguientes ejemplos ilustran el concepto de las gramática:

Ejemplo 1

La siguiente es la gramática 1, que genera palabras de la forma $a^i b^i, i \in \mathbb{Z}^+$.

$$G = \begin{cases} S \mapsto aSb \\ S \mapsto AB \\ A \mapsto a \\ B \mapsto b \end{cases}$$

Nuestro objetivo será partir de S y llegar por ejemplo a $aabb$:

* Correspondence email address: davidsa.florez@urosario.edu.co
† Correspondence email address: lauraso.ortiz@urosario.edu.co

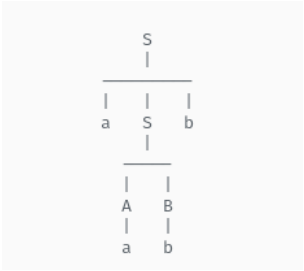

Figure 1. Árbol de parsing para generar *aabb*.

Ejemplo 2

La siguiente es la gramática 2, que genera palabras de la forma $a^i b^j c^j d^i, i, j > 0, i, j \in \mathbb{Z}^+$.

$$G = \begin{cases} S \mapsto aWd \\ W \mapsto aWd \\ W \mapsto V \\ V \mapsto bVc \\ V \mapsto X \\ X \mapsto bc \end{cases}$$

Nuestro objetivo será partir de S y llegar por ejemplo a *aabcdd*:

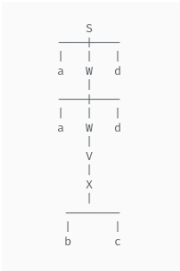

Figure 2. Árbol de parsing para generar *aabcdd*.

Ejemplo 3

La siguiente es la gramática que genera palabras de la forma $a^i b^j, i > j, i, j \in \mathbb{Z}^+$.

$$G = \begin{cases} S \mapsto aR \\ R \mapsto aRb \\ R \mapsto aR \\ R \mapsto aW \\ W \mapsto b \end{cases}$$

Nuestro objetivo será partir de S y llegar por ejemplo a *aaab*:

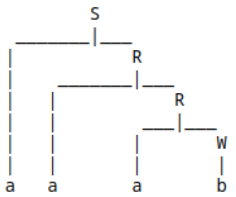

Figure 3. Árbol de parsing para generar *aaab*.

Ejemplo 4

Por último, se creo una gramática que genera palabras palíndromos, como por ejemplo $10101, 01010, 10001$ o 01110, la cuál no se utilizó para realizar pruebas de rendimiento, ya que debido a su cantidad de reglas el tiempo de ejecución de los algoritmos es muy alto (más adelante se podrá observar una gráfica de su tiempo de ejecución unicamente en un intervalo pequeño para la longitud de la palabra a solucionar).

En nuestro problema se tienen las siguientes propiedades del entorno:

Opción 1	Opción 2
✓ Completamente observable	Parcialmente observable
✓ Agente único	Multiagente
✓ Determinista	Estocástico
Episódico	✓ Secuencial
✓ Estático	Dinámico
✓ Discreto	Continuo
✓ Conocido	Desconocido

II. MÉTODOS

La definicion formal del problema de parsing, puede ser descrita como:

- Estado inicial: Variable inicial de la gramática (S).
- Posibles acciones: Las reglas definidas en cada gramática.
- Función de transiciones: Cada una de las reglas que llevan de un estado a otro.
- Prueba de satisfacción del objetivo: Que se logre generar el árbol de anaĺisis de w *(y así se llegue a la palabra w deseada)*.
- Función de costo: En el caso general, se asume que la función de costo es constante para todas las transiciones. Sin embargo en las heurísticas que implementamos el costo depende del largo de la palabra, y en otros casos también de la cantidad de errores. *(esto lo explicaremos a más detalle más adelante)*.

Los métodos empleados para resolver este problema, fueron por medio de los algoritmos de búsqueda mencionados anteriormente. En los algoritmos Best first, Depth limited y Greedy, fue necesario implementarles heurísticas para mejorar su rendimiento. A continuación, una explicación de cada heurística.

A. Heurísticas

1. *Función del largo de la transición*

Nuestra primera Heurística consiste en tener en cuenta el largo de la transición como función de costos, la intención detrás de esto es ayudar al algoritmo a llegar más rápido al estado deseado, un ejemplo de su uso es:

Ejemplo 3.1

Siguiendo el ejemplo 3, suponga que está en un estado aR y quiere generar la palabra ab, el recorrido de acciones hasta aR habría sido: $S \to aR$, en este estado hay tres opciones:

$$\begin{cases} aRb, & len(aRb) = 3 \\ aR, & len(aR) = 2 \\ aW, & len(aW) = 2 \end{cases}$$

donde len significa lenght

Dado que en todo momento nuestro algoritmo *(Best first search y Greedy search)* trata de minimizar el costo camino, se decantará por probar aR o aW primero, ayudándolo así a acercarse más rápido a la solución.

Sin embargo esta Heurística tiene sus desventajas, si la palabra es larga el algoritmo tratará de escoger la regla más corta en vano, un ejemplo particular es que si hubiesemos querido generar *aaaabbb*, nos habría tomado más intentos dado que la heuristica tiene preferencia por las reglas más cortas que llevan a aR o aW, y que no aprovechan las bs, como lo hace aRb.

2. *Función de distancia en longitud y diferencia en caracteres*

Teniendo en cuenta los errores de la anterior Heurística y para explotar aún mejor toda la información que se tiene respecto a la palabra que se quiere obtener, y la palabra que se ha generado hasta el momento planteamos la siguiente función de costos que aprovecha ambas:

$$l\_\,dif(w, obj) = \Big\{ (l(obj) - l(w))^2 \quad \forall w, obj \in L(G) \quad (1)$$

$$c\_\,dif(w, obj) = \begin{cases} l(obj)^2, si\ l(w) \neq l(obj) \\ (\#letras\ diferentes)^2, si\ l(w) = l(obj) \end{cases} \quad (2)$$

Donde l significa *lenght*, y $(w,\ obj)$ son la palabra en la cual se encuentra el algoritmo y la palabra que se desea construir respectivamente.

$$l\_\,c\_\,difference = \Big\{ l\_\,diff(w, obj) + c\_\,diff(w, obj) \quad (3)$$

Como nota para el lector los nombres l_dif y c_dif, vienen de *lenght difference* y *character difference*.

Esta función tiene la ventaja de que castiga si el largo de la palabra es muy alejado de aquel que tiene la palabra obj, incentivando a expandir más la palabra o previniendo de expandirla si es necesario, adicionalmente no se detiene allí, sino que una vez alcanza la longitud de la palabra deseada la penalización por diferencias de longitudes es cero, y entra entonces la información particular que tenemos sobre la palabra solución a la que queremos llegar y cómo se relaciona con la palabra que se ha formado hasta el momento.

3. *Terminación de ejecución*

No son estrictamente unas heurísitcas pero cabe mencionar que para los algoritmos de Best first search y Greedy search optamos por limitar aún más el espacio de búsqueda bajo un básico principio:

Una vez sobrepasada la longitud de la palabra objetivo, se ignora y se pasa a estudiar otro caso que no genere una palabra más larga que la que quiero crear.

De esta manera logramos quedarnos con las exploraciones que sí podrían ser pertinentes a nuestro objeto de búsqueda, al menos en lo que respecta a longitud de palabra.

Por otro lado, para el Depth limited search, escogimos como métrica limitante en profundidad al largo de la palabra que se quiere generar, ya que en el peor de los casos este sería la cantidad de iteraciones requeridas en profundidad.

III. RESULTAODS Y DISCUSIÓN

Una vez comprendido el entorno y el problema de parsing, además de implementados los algoritmos de busqueda utilizados, y sus heurísticas para la solución del problema, se procedió a analizar los tiempos de ejecución de cada uno, para determinar cual algoritmo tiene

mejor rendimiento resolviendo el problema. Para lograr esto usamos las 4 gramáticas definidas, el proceso se resume como:

1. Escoger aleatoriamente entre 5 y 10 gramaticas de las 4 creadas, *(pueden haber repeticiones)* y para cada gramática seleccionada se crea una palabra del estilo de la gramática *(dicha palabra tiene una longitud variable, por ejemplo el largo de una palabra para la gramática 1 podría estar entre 1 y 5, particularmente "aabb" está en este rango).*
2. Para cada problema creado *(gramatica y palabra a solucionar)* medíamos tiempos de ejecución, en esta parte se ejecutaba n veces para cada problema *(generalmente 10 veces).*

Los resultados fueron los siguientes:

A. Palabras de longitud (1, 5) generadas por las gramáticas 1, 2 y 3

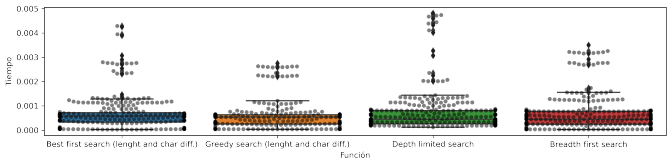

Figure 4. Boxplot de resultados de los tiempos de ejecución para problemas de un rango de longitud de palabra (1, 5).

En estos primeros boxplots, podemos notar que los cuatro algoritmos tienen un tiempo de ejecución similar, pero el Greedy search y el Best first search, son un poco más rápidos. También cabe mencionar que en todos los boxplot se encuentran bastantes datos atípicos, indicandonos que los tiempos de ejecución también son muy variables.

B. Palabras de longitud (5, 10) generadas por las gramáticas 1, 2 y 3

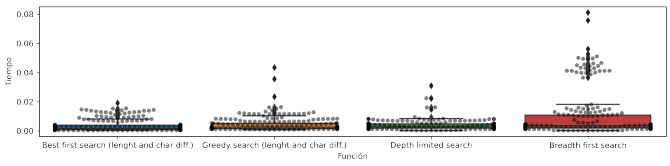

Figure 5. Boxplot de resultados de los tiempos de ejecución para problemas de un rango de longitud de palabra (5, 10).

En este caso, se puede observar que los cuatro algoritmos son rápidos en un rango pequeño, pero que el Breadth first search es el más lento entre todos. Además, el Breadth junto al Best first son los que más datos atípicos tienen.

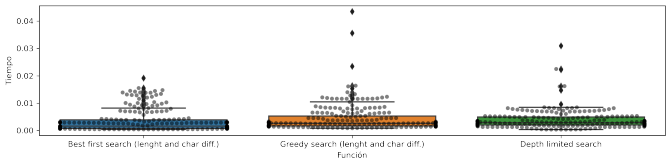

Figure 6. Boxplot de resultados de los tiempos de ejecución para problemas de un rango de longitud de palabra (5, 10).

Viendo mejor los algoritmos más rápidos, el Depth limited rinde un poco mejor, y a comparación de los tros dos no tiene casi datos atípicos en este caso.

C. Palabras de longitud (10, 15) generadas por las gramáticas 1, 2 y 3

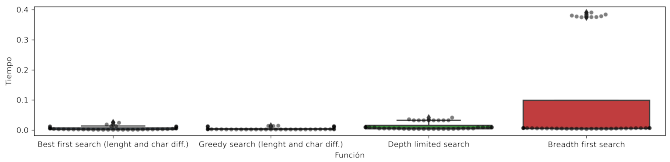

Figure 7. Boxplot de resultados de los tiempos de ejecución para problemas de un rango de longitud de palabra (10, 15).

En este rango se puede seguir observando que el Breadth search sigue siendo mas demorado.

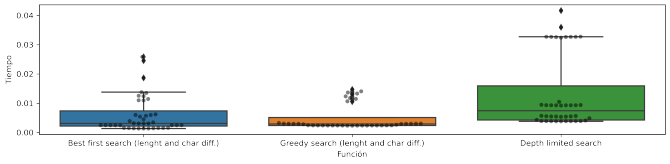

Figure 8. Boxplot de resultados de los tiempos de ejecución para problemas de un rango de longitud de palabra (10, 15).

Y al hacer una comparación entre los tres más rápidos, se puede ver que el Greedy search es el que mejor tiempo tiene entre todos.

D. Palabras de longitud (15, 20) generadas por las gramáticas 1, 2 y 3

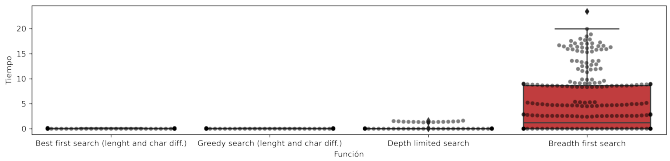

Figure 9. Boxplot de resultados de los tiempos de ejecución para problemas de un rango de longitud de palabra (15, 20).

Para un rango un poco mas grande, el Breadth search incrementa su tiempo de ejecución mucho más.

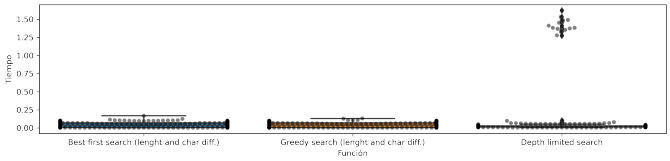

Figure 10. Boxplot de resultados de los tiempos de ejecución para problemas de un rango de longitud de palabra (15, 20).

Y también se puede observar que para este rango el Depth limited search fue el más rápido, igual cabe recalcar que este algoritmo tiene bastantes datos atípicpos.

E. Palabras de longitud (1, 100) y (100, 200) generadas por las gramáticas 1 y 2

Teniendo en cuenta los anteriores resultados, nos dimos cuenta que en rangos un poco más grandes la gramática 3 tomaba mucho tiempo a comparación de los rangos pequeños. Por esto, se decidió comparar el tiempo de ejecución de los algoritmos, con las dos primera gramáticas en un rango más grande.

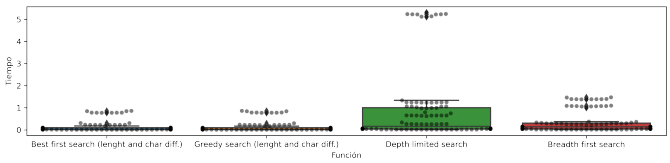

Figure 11. Boxplot de resultados de los tiempos de ejecución para problemas de un rango de longitud de palabra (1, 100).

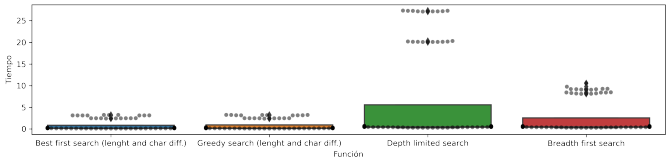

Figure 12. Boxplot de resultados de los tiempos de ejecución para problemas de un rango de longitud de palabra (100, 200).

Se puede observar que en rangos grandes, el algoritmo más lento Depth limited search.

F. Palabras de longitud (1, 5) generadas por la gramática 4

Por otra parte, como se menciono en el ejemplo 4, esta gramática tomaba demasiado tiempo al correrla con los algoritmos. Uno de los posibles motivos era el hecho de que tenía el doble de reglas que las demás gramáticas.

Por lo anterior, y realizando varias pruebas se logró correr la gramática en el rango más pequeño, es decir palabras de longitud (1, 5). El resultado fue el siguiente:

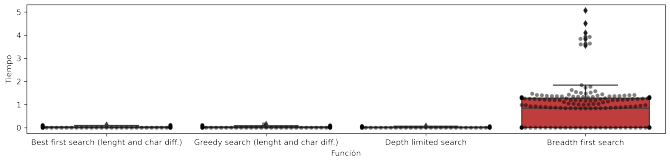

Figure 13. Boxplot de resultados de los tiempos de ejecución para problemas de un rango de longitud de palabra (1, 5).

Con este resultado, se puede seguir observando que el algorimo que le toma más tiempo y que tiene más valores atípicos, es el Breadth first search. El mal rendimiento del Breadth first search podría tener como causa la gran cantidad de reglas que tiene la gramática 4 frente a las demás, dando siempre muchas más opciones para explorar en anchura.

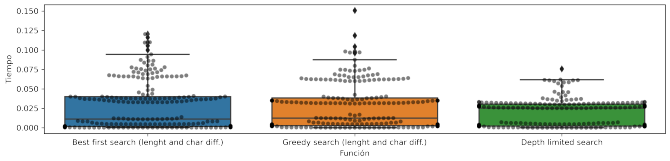

Figure 14. Boxplot de resultados de los tiempos de ejecución para problemas de un rango de longitud de palabra (1, 5).

Y entre los otros tres algoritmos, el mejor fue el Depth limited search. También se puede notar que para la gramática 4, el algoritmo Greedy search tiene unos pocos datos atípicos.

G. Gráfica curva de tiempos de ejecución

Finalamente, luego de realizar varios análsis de las gramáticas, con palabras de diferente longitud, podemos observar que los dos mejores algoritmos en la mayoría de casos son el Greedy search y Best first search, a los cuales se les crearon heurísticas.

Para estos dos algoritmos se decidió realizar una curva de sus tiempos de ejecución según la longitud de la palabra, los resultados fueron los siguientes:

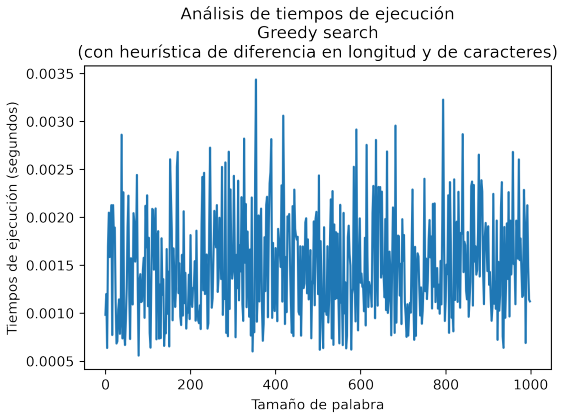

Figure 15. Curva tiempos de ejecución del algoritmo Greedy search.

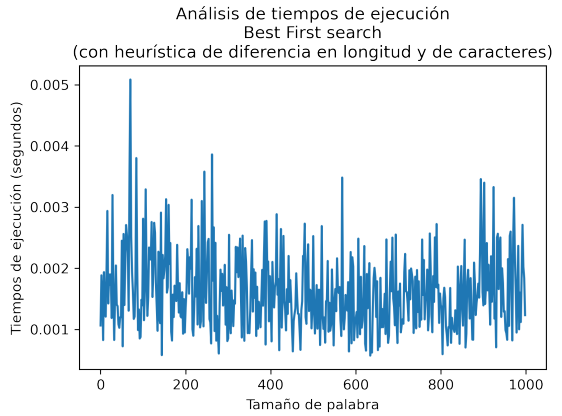

Figure 16. Curva tiempos de ejecución del algoritmo Best first search.

Note, que ambas gráficas tienen mucho ruido, esto es debido a que, como en cada rango se corren diferentes palabras creadas aleatoriamente por las 3 primeras gramáticas, los datos van a ser muy dispersos.

H. Siguientes pasos

Nuestros mejores resultados se dieron con la heurística que combinaba la información que se tenía tanto del largo, como de la diferencia en caracteres que había. Sin embargo aún hace falta explotar la información completa que tenemos sobre la gramática, se podría utilizar un grafo que relacione reglas y que de un peso adicional en función de cuántas reglas *'desbloqueo'* si uso una u otra regla.

También el poder crear un más amplio set de gramáticas de testeo es otro paso, para con ello poder tener información del comportamiento del algoritmo en condiciones más ricas, y conforme la complejidad de la gramática aumenta.

IV. CONCLUSIONES

Podemos concluir que los mejores algoritmos en general *(teniendo en cuenta todos los tamaños de problema)* son Best First Search y Greedy Search, aunque si se busca el mejor rendimiento para tamaños pequeños de problema, el Depth Limited debería ser la elección, sin embargo cabe notar que aunque es rápido su rendimiento también es muy variable luego se corre también el riesgo de esperar mucho más que lo que se esperaría con los dos primeros mencionados.

REFERENCES

[1] Rodrigo De Castro. *Teoría de la computación: lenguajes, autómatas, gramáticas*. Facultad de ciencias de la Universidad Nacional, 2004. URL: `https://repositorio.unal.edu.co/bitstream/handle/unal/53477/Notasdeclaseteoriadelacomputacion.PDF?sequence=1&isAllowed=y`.