Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Tarea 2: Arreglo y Árbol de Sufijos

Diseño y Análisis de Algoritmos - CC4102-1

Integrantes: Pablo Gutiérrez
María Antonia Hernández
Sebastián Salinas
Profesor: Gonzalo Navarro
Auxiliar: Antonia Labarca
Ayudantes: Claudio Zamorano
Javier Oliva
Matías Rojas
Nicolás Canales

Fecha de entrega: 13 de diciembre de 2021

Índice de Contenidos

Índice de Figuras

1. Introducción

En esta tarea se estudian e implementan árboles y arreglos de sufijos, con el objetivo de encontrar ocurrencias de palabras en distintos textos. Para ello, a partir del nombre de un archivo de texto ingresado por el usuario se debe conseguir un arreglo de sufijos para todo el contenido que se menciona en el documento.

Luego, con un programa interactivo en el cual se le pide al usuario una cadena de texto a buscar, el programa busca las ocurrencias de la cadena en el archivo ingresado y reporta la cantidad de ocurrencias de éste, imprimiendo para cada una de ellas la línea de texto en donde se encontró.

Luego, con el arreglo de sufijos ya creado, se implementa un árbol Patricia donde se almacenan todos los sufijos de dicho texto, para ello primero se implementa una estructura para representar al árbol y luego se implementan métodos de inserción, búsqueda y creación de árbol.

Estas dos estructuras antes mencionadas (arreglo y árbol de sufijos) son de interés para la última implementación, la cual es un auto completado de textos: Al momento de escribir una frase el programa te sugiere 3 posibles continuaciones para la palabra a buscar, mientras más frases escribes, las sugerencias irán cambiando.

2. Implementación

Para la realización de esta tarea se utilizó el lenguaje de programación *Python* y se crearon diferentes scripts, funciones y clases para modelar el problema y darle una solución.

2.1. Arreglo de Sufijos

En la implementación del arreglo de sufijos se implementó una función *suffix_array* que recibe un string y crea un arreglo de sufijos de acuerdo al string ingresado de la siguiente manera:

- Dado el string, se crea un diccionario y se itera con un valor 'i' que recorre todos los índices del largo de la lista (es decir, desde 0 hasta len(palabra)).
- En cada iteración se aumenta el tamaño del diccionario, agregando una llave y un valor asociado, donde la llave es el índice donde inicia la palabra (es decir, nuestro i) y el valor es el string desde dicho índice hasta el final.

  De esta manera, al ingresar el string 'hola' a la función, tendremos un diccionario:

  $$\{0\text{:'hola'}, 1\text{,'ola'}, 2\text{:'la}, 3\text{:'a'}\}.$$

- Se ordenan las tuplas en una lista de acuerdo a el valor que contenían, es decir, las palabras. esto se hace obteniendo los ítems de un diccionario como tuplas.

  En nuestro ejemplo, el diccionario {0:'hola', 1,'ola',2:'la, 3:'a'} pasa a ser la lista de tuplas:

  $$[(0,\text{'hola'}),(1,\text{'ola'}),(2,\text{'la'}),(3,\text{'a'})]$$

  y al ordenarse esta lista, tendríamos:

  $$[(3,\text{'a'}),(0,\text{'hola'}),(2,\text{'la'}),(1,\text{'ola'})].$$

- Finalmente, con las palabras ya ordenadas, solamente se guardan los índices en una lista que sería nuestro arreglo de sufijos, siendo en nuestro ejemplo, el arreglo de sufijos [3,0,2,1].

Luego, con la función ya hecha, se abre el archivo de texto en modo lectura y se genera el arreglo de sufijos a partir de dicho texto leído. Con esto tenemos la primera parte de nuestro programa, logrando hacer el arreglo de sufijos a partir de un texto.

2.2. Programa de búsqueda de cadenas

Primero se consigue el arreglo de sufijos con la función de la primera parte. Luego, se inicia un loop para preguntarle al usuario sobre la cadena con un input.

Después de preguntarle una cadena al usuario, el programa realizará una búsqueda binaria en la cual se asignan punteros al inicio y al final del arreglo (es decir puntero1= 0 y puntero2 = len(arreglo_sufijos)), estos punteros se ajustarán hasta lograr que entre estos índices tengamos índices de palabras que contengan nuestra palabra buscada.

Existe el caso en el cual algunas palabras que busquemos no estén en el intervalo definido por ambos punteros. Para este caso iremos expandiendo el intervalo incluyendo las palabras que nos faltan, esto lo haremos moviendo los punteros hasta la derecha o izquierda para obtener el mayor rango de palabras posibles. Con esto ya tenemos todos los índices y podemos seleccionar todos los strings del texto original que contienen la cadena, siendo estos cualquier índice denotado por el intervalo obtenido.

Luego, se muestra la línea entera donde la palabra ingresada por el usuario aparece, para esto se actúa de forma similar: A partir del índice de la palabra se crean dos punteros, uno irá hacia la izquierda de saltos de a 1 hasta llegar al inicio o a un salto de línea. El otro hará lo mismo hacia la derecha, encontrando un santo de línea o el fin del texto.

Por ejemplo, si tenemos el string '/nchaobholaa/n' podemos setear dos punteros en la posición de la 'b' y de ellos, ir a la izquierda y derecha encontrando el salto de línea o el fin de la palabra.

Finalmente repetimos para cada índice que coincide con nuestra cadena a buscar y la mostramos en pantalla, obteniendo lo solicitado.

2.3. Árbol de Sufijos

Para construir el árbol de sufijos, se procede a hacer una nueva clase que denota a un nodo, este tiene 8 elementos internos:

- *index_or_char* que nos indica el índice del inicio de la palabra o la letra que este tiene.
- *common_len* que indica cuantas palabras en común tienen los hijos después de *index_or_char*
- *children*, una lista con nodos que representan a los hijos de dicho nodo.
- *weight* que representa la cantidad de hojas que tiene un nodo
- *accumulated_len* que almacena la cantidad acumulada de la cantidad de letras.
- *depth* que indica la profundidad en la que se encuentra el nodo respecto al árbol.
- *is_leaf* que indica si un nodo es hoja.
- *is_root* que indica si un nodo es *root*.

Con todas estas variables tenemos una clase que puede cumplir el rol de cualquier nodo, tanto sea *root*, *leaf* o algún nodo interno. A continuación, se muestra un ejemplo del árbol resultante con la frase inventada 'banana babnana'.

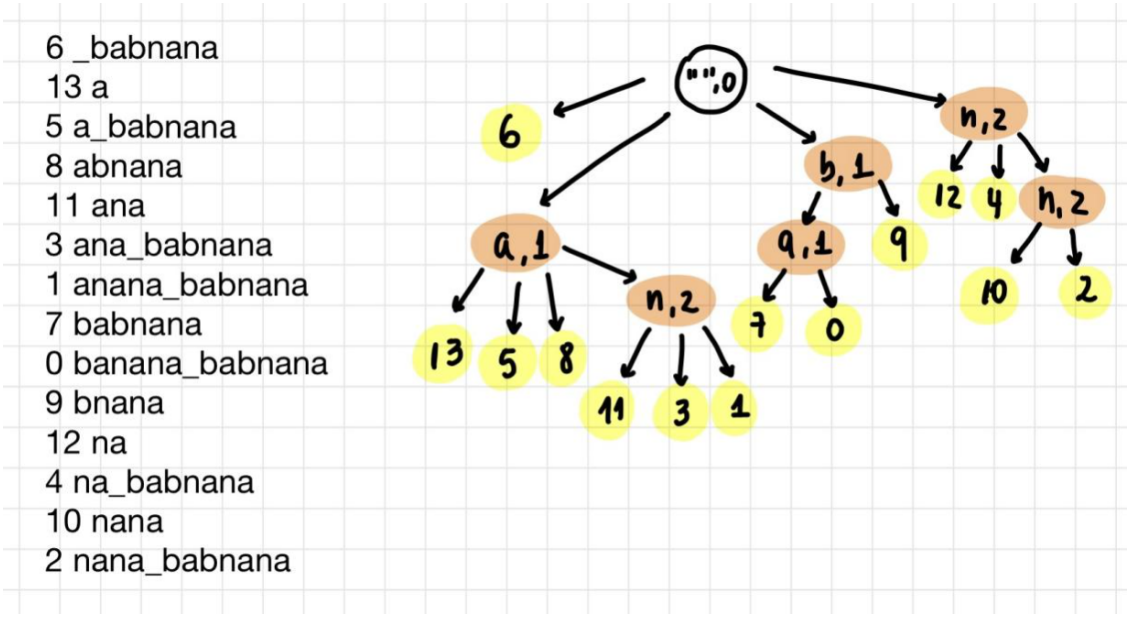

Figura 1: Árbol de sufijos para "banana babnana"

Para la búsqueda se crea una función que retorna el nodo que contiene a la palabra a buscar. En el caso de que la palabra no se encuentre, retorna el nodo más cercano de la aparición de este, esto será útil para la implementación de la inserción.

En la implementación del método, primero se analizan los casos básicos de hoja y *root*: para la hoja, retornamos la hoja misma, ya que no existen nodos que puedas ser más específicos para el elemento buscado. Para el caso del *root* nos guiamos por el resultado del parámetro *index_or_char*

para tomar una decisión, si encontramos un índice retornamos este, si no es el caso, retornamos la búsqueda del nodo que comparta la primera letra, si no existe solamente retornamos el *root*.

Para el caso de un nodo interno, vemos si tiene hijos o no, el caso trivial es que, si no tiene hijos, el mismo nodo es el de último acceso. Después, analizamos el largo de las letras que tienen en común en ese nodo, de ahí existen 3 posibilidades, una donde el largo del nodo es mayor, otro donde los largos son iguales, para eso buscamos una hoja en ese nodo, para entregar esa palabra, y finalmente, si no existe buscaremos en los hijos la letra que sigue por buscar en nuestra palabra, verificando que las letras que estemos saltando cumplan con sus propiedades.

Para la inserción, se crea el método *insert* el cual recibe como parámetro el índice a insertar.

Lo primero que hace es buscar el nodo donde se debe insertar, esto se hace con un llamado al método *search*. Luego el nodo entregado puede tener 4 formas:

- Ser un nodo con su lista de hijos vacía, este caso solo se utiliza 1 vez, al ingresar la primera palabra al árbol. En este caso se crea una hoja con el índice ingresado y se modifican los demás parámetros de *Node* para que sean consistentes. Esta hoja se agrega a la lista de hijos de *root*.
- Puede ser un nodo interno, y que ninguno de sus hijos siga con la misma letra que la de la palabra ingresada. En este caso se crea una hoja con el índice ingresado y se modifican los demás parámetros de *Node* para que sean consistentes. Esta hoja se agrega a la lista de hijos del nodo.
- Que alguno de sus hijos sea una hoja, la cual coincide en una secuencia de caracteres con la de la palabra ingresada. En este caso se calcula cuantas caracteres en común tienen. Luego se modifica la hoja, haciéndola nodo interno, donde se coloca la primera letra en común y el largo de la secuencia de caracteres en común encontrada. Éste tiene como hijos 2 nodos, uno con el valor anterior y el otro el índice ingresado. Además, se modifican los demás parámetros de los nodos para que el árbol sea consistente.
- Que la palabra coincida en una menor cantidad de caracteres que las de *common_len*, en este caso se hace un *split* del nodo. Los hijos que tenía el nodo encontrado se agregan como hijos a un nuevo nodo. Al nodo que se tenía se le modifica el *common_len*, se le agrega como hijo un nodo con el índice a insertar y el nuevo nodo creado el cual contiene la siguiente letra correspondiente en común y los hijos del nodo inicial.

2.4. Auto completado

Para la implementación del auto completado se utiliza un atributo *weight*. Para su cálculo se realizó un preprocesamiento al momento de realizar las inserciones de los nodos. De esta forma se logra que cada nodo interno tenga información de la cantidad de hojas que tiene debajo. Con este atributo es posible identificar a partir de un nodo interno los tres hijos con más descendientes que nacen desde él, y así recomendar mediante el auto completado las palabras más útiles para el usuario.

Las opciones que finalmente se recomiendan son las primeras 3 ocurrencias encontradas entre los hijos con más descendientes del nodo interno en el cual se está buscando.
Cabe destacar que según la estructura del árbol se tienen varias combinaciones posibles:

- El nodo interno, desde el cual se está recomendando, tiene 3 hojas. En este caso se recomiendan esas 3 palabras.
- El nodo interno, desde el cual se está recomendando, tiene menos de 3 hojas como hijas, pero hay nodos internos que nacen desde él. En este caso, si es que hay, se recomiendan las opciones representadas por las hojas existentes y se seleccionan los nodos internos con más descendientes para buscar recursivamente las opciones restantes.
- El nodo desde el cual se está recomendando tiene menos de 3 hojas y no hay nodos internos como hijos de él. En este caso no es posible recomendar 3 opciones y solo se recomiendan las opciones equivalentes a la cantidad de hojas existentes.
- Solo existen 2 nodos internos como hijos del nodo desde el cual se está recomendando. En este caso se selecciona el nodo interno con más descendientes entre los 2 y se le piden recursivamente 2 opciones para recomendar y al otro 1 opción.

Entrando un poco más en detalle, para poder obtener las opciones de autocompletado a partir de un nodo se creó el método *get_words*, el cual se encarga de añadir a una lista las mejores opciones según lo indicado anteriormente y en el caso en que aún no existan 3 recomendaciones, entrar recursivamente a buscar opciones más en profundidad en el árbol.
A continuación, se presenta un ejemplo de recomendaciones de autocompletados con un extracto de un poema de *Shakespeare*

```
Leyendo el archivo: .\Archivos_de_prueba\Shakespeare.txt
Ingrese una cadena a buscar ("Fin para terminar la ejecución"): bro
Podrías estar buscando estas palabras:
-> brother, on his
 blessing...
-> bred better; for, besides...
-> brother Jaques he keeps a...
Ingrese una cadena a buscar ("Fin para terminar la ejecución"): bro
```

Figura 2: Ejemplo de autocompletado

3. Explicación de la utilización de los programas

Para crear el arreglo de sufijos de un archivo, se decide hacer un archivo que solo implemente esta funcionalidad, para eso, se crea el documento *'suffix_array_for_part_1'* el cual se debe ejecutar en una terminal de la siguiente manera:

python suffix_array_for_part_1.py textfile.txt

donde textfile.txt es el archivo de texto que se quiere leer. El programa imprime el arreglo de sufijos en pantalla.

Importante: El archivo importa un documento llamado *'suffix_array.py'* el cual se adjunta en este informe, solo contiene la función y se utilizará para lo que resta de la tarea.

Para el segundo punto de la sección, se crea el programa interactivo, el cual hace un loop para encontrar las ocurrencias de una palabra. Los datos del loop se encuentran en el archivo *suffix_array_program*, para ejecutar el archivo se debe escribir en la terminal:

python suffix_array_program.py textfile.txt

donde textfile.txt es el archivo de texto a leer. Para salir del programa se puede escribir 'Fin'.

Para el tercer y cuarto punto, se decide hacerlo en un solo archivo, siendo este llamado *'mariTree.py'*, el cual se ejecuta de la siguiente manera:

python mariTree.py textfile.txt

donde 'textfile.txt' es el nombre del archivo que se va a leer, el archivo crea el arreglo de sufijos y luego el árbol patricia con los contenidos de este, para finalmente terminar con el programa interactivo de auto completado.

4. Discusión y Conclusión

Se concluye que el arreglo de sufijos es una estructura útil al momento de tener textos de gran tamaño donde se quiere hacer búsqueda entre sus palabras. Dado que el arreglo está ordenado lexicográficamente, se puede implementar búsqueda binaria y así obtener una respuesta de manera mucho más rápida que haciendo una búsqueda lineal. Además, la creación de este arreglo se pudo hacer de manera más eficiente en otros lenguajes de programación como C dada la facilidad de sus punteros y así hacer menos uno de memoria, o en C++ dado que la búsqueda dentro de un diccionario se hace de forma más eficiente.

Además, el árbol de sufijos es una buena forma de obtener predicciones ya que los nodos pueden almacenar la cantidad de hojas que hay debajo de él y de esta forma se recomienda la palabra más probable.

Con respecto al archivo *'aaa.txt'*, al crear su árbol de sufijos se encuentra que, dada forma del contenido de éste, el árbol resultante está totalmente desbalanceado. En nuestro caso, al usar el lenguaje *Python* y métodos recursivos, el resultado final fue que se alcanzó el límite de recursión con facilidad, de esta manera concluimos para este caso el lenguaje seleccionado no óptimo para textos de esta manera, ya que el árbol en este caso se podría completar, pero las limitaciones del lenguaje lo impiden.

Las limitaciones del lenguaje no solo jugaron un rol en contra en el caso del archivo antes mencionado, sino también en archivos de tamaños mayores a 500kb.

A pesar de todo se logró comprender la estructura de datos y por qué es bueno utilizarla para problemas de la vida real, se logra optimizar espacio debido al ahorro al momento de almacenar solo un carácter y aun así servir para la búsqueda y auto completado de prefijos de un texto.