Informe Trabajo Práctico Final
Analizador Sintáctico

Rassi, Octavio 04/09/2023

1. La estructura

1.1. Definición

La estructura de datos principal en la que se basa el programa es **Trie.** Un Trie es una estructura de datos de tipo árbol que permite la recuperación de información, de allí su nombre re*trie*val. La implementación de Trie que utilizaremos en el programa es:

Definición de la estructura Trie.

```
1 #define ALPHABET_SIZE 26
2
3 typedef struct _Trie {
4   struct _Trie *children[ALPHABET_SIZE];
5   struct _Trie *fail;
6   struct _Trie *valid_suffix;
7   int word_length;
8 } Trie;
```

Esta definición cuenta con algunas diferencias respecto a la definición mas elemental de Trie. Además de contar con un array de punteros a Trie, que representa los 26 posibles hijos de cada nodo, consideramos tres campos extra que serán de utilidad.

El campo **word_length** será utilizado como reemplazo del campo *is_word* que suele estar presente en la implementación de Trie. Este campo almacenará, en los nodos que representan el final de una palabra, la longitud del camino desde ese nodo hasta la raíz. Cuando el nodo no represente una palabra válida en nuestro diccionario, este valor sera 0.

Los campos **fail** y **valid_suffix**, por otro lado, serán clave para el algoritmo propuesto para el programa. Inicialmente podemos pensarlos como enlaces a otros nodos del Trie que nos serán de utilidad para recuperarnos de errores eficientemente. Su construcción se verá en el apartado 2.1.

1.2. Construcción

A partir de un diccionario de palabras dado, la construcción del Trie que lo representa se logra a partir de la inserción de cada una de las palabras del diccionario en un Trie inicialmente vacío.

Inserción en un Trie.

```
1  void trie_insert(Trie * root, char *word) {
2    // Asumimos root != NULL, i. e. insertamos en un trie ya definido.
3    Trie *tmp = root;
4
5    int cPos, i;
6    for (i = 0; word[i] != '\0'; i++) {
7      cPos = (int) tolower(word[i]) - 'a';
8
9      if (tmp->children[cPos] == NULL)
10       tmp->children[cPos] = trie_create();
11
12     tmp = tmp->children[cPos];
13   }
14
15   // Al terminar la cadena, el ultimo nodo es una palabra valida.
16   tmp->word_length = i;
17 }
```

El proceso de inserción consiste en leer la cadena que queremos ingresar caracter por caracter. Partiendo de la raíz, tomamos el primer caracter de la cadena, le asignamos su índice correspondiente (sin distinción de mayúsculas o minúsculas), y verificamos si ya existe un hijo del nodo actual que lo represente.

Si tal nodo no existe, lo creamos con trie_create(). Luego, desplazamos el nodo *tmp* a este nuevo hijo, y continuamos recorriendo la cadena.

Al finalizar el bucle for, habremos generado todo el camino que representa a la palabra en el trie. Solo resta establecer en el último nodo la longitud de la palabra, que es exactamente la cantidad de iteraciones que realizó el bucle.

2. El algoritmo

El programa entregado se basa en el **algoritmo de búsqueda de cadenas de Aho-Corasick** (*Alfred V. Aho* y *Margaret J. Corasick* 2, 1975), que desarrollaremos a continuación.

2.1. Construcción del autómata

Función transform_trie_into_automaton

```
1  void transform_trie_into_automaton(Trie * root) {
2    root->fail = NULL;
3    for (int i = 0; i < ALPHABET_SIZE; i++)
4      if (root->children[i] != NULL)
5        root->children[i]->fail = root;
6
7    // BFS para la construccion de los enlaces sufijos (fail)
8    Queue q = queue_create();
9    for (int i = 0; i < ALPHABET_SIZE; i++)
10     if (root->children[i] != NULL)
11       queue_enqueue(q, root->children[i], id);
12
13   while (!queue_is_empty(q)) {
14     Trie *node = queue_start(q, id);
15     queue_dequeue(q, skip);
16
17     for (int i = 0; i < ALPHABET_SIZE; i++) {
18       if (node->children[i] != NULL) {
19         Trie *fail_node = node->fail;
20
21         while (fail_node != NULL && fail_node->children[i] == NULL)
22           fail_node = fail_node->fail;
23
24         if (fail_node == NULL)
25           node->children[i]->fail = root;
26
27         else {
28           node->children[i]->fail = fail_node->children[i];
29
30           // Construccion de los enlaces sufijos en diccionario
31           Trie* vsuffix_node = node->children[i]->fail;
32           while (vsuffix_node != root && vsuffix_node->word_length == 0)
33             vsuffix_node = vsuffix_node->fail;
34
35           if (vsuffix_node != root)
36             node->children[i]->valid_suffix = vsuffix_node;
37         }
38
39         queue_enqueue(q, node->children[i], id);
40       }
41     }
42   }
43   queue_destroy(q, skip);
44 }
```

El algoritmo inicia con la construcción de un **autómata de estado finito** a partir de un Trie. Este paso se corresponde con la función transform_trie_into_automaton. En ella, utilizamos una cola general para "conectar" los punteros fail y valid_suffix de cada nodo con Breadth-First Search.

El método para hallar los enlaces fail es simple. Los enlaces fail de los hijos de la raíz son la misma raíz. Ahora, sea P un nodo con su enlace fail ya construído, tal que posee un hijo H asociado a un caracter *c*. Entonces, el enlace fail de H es el hijo asociado al caracter *c* del primer nodo en el camino de los enlaces fail de P tal que ese hijo existe. Si tal nodo no existiera, definiremos al enlace fail de H como la raíz.

Finalmente, el enlace valid_suffix de H es el primer nodo en el camino de los enlaces fail de H que es una palabra válida, si es que existe.

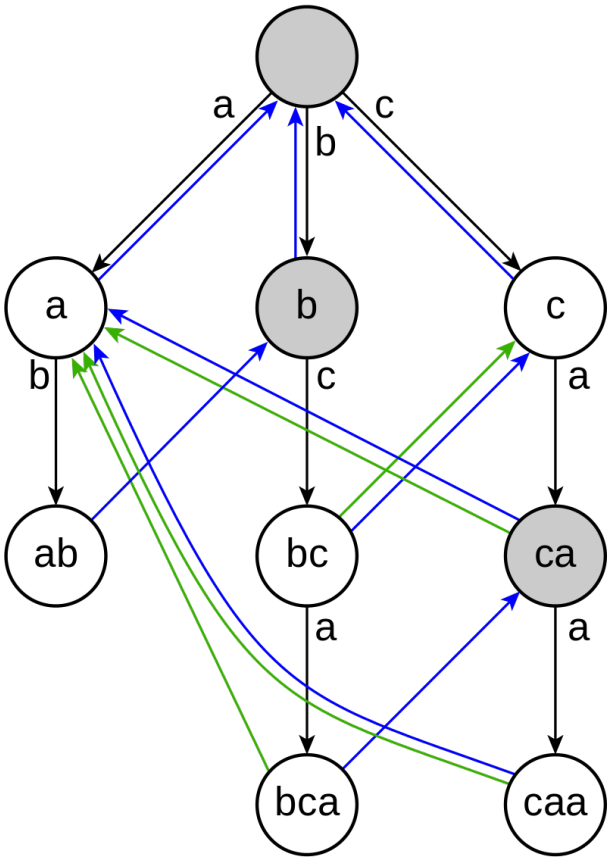

Figura 1: Trie transformado en Autómata

En la figura 1 podemos ver representado un Trie con los enlaces mencionados ya construídos. Este Trie reconoce al diccionario **{a, c, ab, bc, bca, caa}**, cuyos nodos podemos ver en blanco. Los nodos grises representan simplemente prefijos de estas palabras válidas. Podemos observar arcos de tres colores en cada nodo:

- En negro, se grafícan los arcos dirigidos de cada nodo a aquel que puede obtenerse añadiendose un caracter. Son los arcos *hijos*, que conectan dos nodos a través de un

caracter *c*.

- En azul, se grafícan los arcos que van de cada nodo hacia su sufijo propio mas largo en el trie. Los llamaremos enlaces *sufijos*, estos se corresponden con el campo *fail*.
- En verde, se grafícan los arcos que denominaremos *sufijos en diccionario*. Estos enlaces van de cada nodo al siguiente nodo que es una palabra válida y al cual se puede llegar siguiendo un camino de arcos azules. Rapidamente podemos notar que no todo nodo posee un arco verde, pues no todo camino de arcos azules necesariamente pasa por una palabra válida. Estos se corresponden con el campo *valid_suffix*.

En nuestro algoritmo de construcción, hasta ahora, hemos creado los enlaces hijos con la inserción de todas las palabras en el Trie, y los enlaces sufijos y sufijos en diccionario con la transformación del Trie en autómata.

2.2. Parseo

La utilidad del autómata generado en el paso anterior recae en la habilidad para recuperarse de errores que le brinda al programa. Estos enlaces adicionales que hemos generado se utilizan para saltar a través del Trie cuando no se encuentra una coincidencia durante la búsqueda de patrones.

En lugar de volver a la raíz del Trie y comenzar de nuevo, el algoritmo de Aho-Corasick (2)utiliza los punteros fail para continuar la búsqueda desde el nodo más largo que es un sufijo propio del prefijo actual.


Figura 2: *Alfred V. Aho y Margaret J. Corasick*

Es importante recalcar que el algoritmo original busca todas las ocurrencias de cada palabra del diccionario en el texto en tiempo lineal (respecto a la suma de las longitudes del diccionario y la longitud del texto). Esto no es exactamente lo que nuestro programa requiere por lo que, como veremos a continuación, propondremos una leve modificación a esta idea.

Una vez construído el autómata, el programa lee el archivo de entrada línea por línea, invocando a la función *parse_string* a medida que avanza.

Función parse_string

```
1  void parse_string(Trie * root, char *text, FILE * fp_out) {
2    int n = strlen(text);
3    int longest_word[n];
4    memset(longest_word, 0, sizeof(longest_word));
5
6    Trie *node = root;
7
8    for (int i = 0; i < n; i++) {
9      int index = char_to_index(text[i]);
10
11     while (node != NULL && node->children[index] == NULL)
12       node = node->fail;
13
14     if (node == NULL)
15       node = root;
16
17     else {
18       node = node->children[index];
19       Trie *suffix_node = node;
20
21       while (suffix_node) {
22         update_longest_word(longest_word, suffix_node, i);
23         suffix_node = suffix_node->valid_suffix;
24       }
25     }
26   }
27
28   char errors[n];
29   int skippedCharsCount =
30       print_parsed_text_into_file(longest_word, text, errors, n, fp_out);
31
32   print_error_message_to_file(errors, skippedCharsCount, fp_out, n);
33 }
```

La función parse_string recibe la raíz del Trie, que ya ha sido convertido a un autómata, una linea a parsear text, y un puntero a archivo donde imprimiremos el resultado del parseo.

Su ejecución comienza hallando el largo de la línea, para luego crear un array de ints, *longest_word*, de igual tamaño. Luego, recorre una vez la línea caracter a caracter, con el objetivo de llenar el array longest_word de manera que en la posición i-ésima se encuentre la máxima longitud posible de una palabra válida en el diccionario que comience en la i-ésima posición de la línea leída.

El algoritmo que seguiremos para encontrar estas longitudes máximas comienza con la lectura de un caracter del texto. Posteriormente, intentamos avanzar por un enlace hijo con este caracter. En caso de no existir tal arco, retrocederemos un paso por el arco sufijo o fail del nodo, y volveremos a intentar avanzar por un enlace hijo desde este nuevo nodo de la misma manera.

Este proceso, representado por el primer bucle *while*, finaliza tras encontrar un nodo que efectivamente tenga un enlace hijo con el caracter actual, que pasará a ser el nodo actual, o tras llegar a un nodo NULL. Esta última situación implica que hemos retrocedido por los enlaces sufijos hasta la raíz, y no hemos encontrado ningún enlace hijo en el camino.

En caso de haber hallado un enlace hijo, procederemos a almacenar todos las palabras a las que se puede llegar por un camino de enlaces sufijos en diccionario (es decir, por enlaces verdes) desde el nodo actual. Esto lo lograremos retrocediendo por los enlaces sufijos en diccionario e invocando a la función *update_longest_word* a medida que avanzamos. Esta función se encarga de actualizar la posición correspondiente del array de longitudes máximas si se verifíca que la palabra hallada efectivamente es de mayor longitud que la previamente almacenada (si la hubiera).

Es interesante destacar que podría haberse omitido el campo *valid_suffix* en la estructura, pues este deriva directamente del campo *fail*. En tal caso, recorreriamos el camino de arcos *fail*, chequeando en cada paso si el nodo actual es una palabra válida. Esto daría lugar a una constante repetición de operaciones, pues siendo el diccionario un elemento fijo, siempre recorreríamos el mismo camino. Es por esta razón que decidimos incorporar un campo extra a la estructura en el cual almacenar esta información, cuyo cómputo se realiza una única vez con la construcción del autómata.

Una vez que hemos completado las posiciones necesarias del array longest_word, solo resta recorrerlo una última vez para obtener el parseo buscado. A medida que avanzamos por el array, tomaremos las palabras asociadas a las longitudes máximas de cada posicion y las imprimiremos en el archivo de salida.

De esta manera, si al recorrer el array nos encontramos con una longitud n en la posición i, escribiremos la palabra comprendida entre las posiciones i e $i + n$ del texto en el archivo, y avanzaremos n posiciones en el array.

3. Librerías auxiliares

Dentro del directorio del programa puede encontrarse la implementacion de dos estructuras de datos auxiliares para el programa. En esta sección se explicará brevemente sus definiciones y utilidad.

3.1. Listas generales

La implementación de listas generales viene dada por los archivos *GLIST.c* y *GLIST.h*. Se trata de listas simplemente enlazadas de tipos de datos void*, que serán de utilidad para la implementación de colas generales.

Definición de las estructuras GNodo y GList.

```
1 typedef struct _GNode {
2   void *data;
3   struct _GNode *next;
4 } GNode;
5
6 typedef struct _GList {
7   GNode *first, *last;
8 } GList;
```

Es importante destacar que, para poder realizar la inserción y eliminación de elementos de la cola en tiempo constante, hemos definido a las GList con punteros al inicio y fin de la lista.

Incluye las operaciones is_empty, add_start, add_end, remove_start, traverse y destroy.

3.2. Colas generales

Como mencionamos en la subsección anterior, hemos implementado listas generales con el fin de definir una estructura de cola general. Esta estructura fue utilizada para llevar a cabo la construcción del autómata con BFS.

Definición de Queue

```
1 typedef GList *Queue;
```

Como puede observarse, una cola general es simplemente un puntero a una GList. La implementación propuesta incluye las operaciones is_empty, enqueue, start, dequeue, y destroy.

4. Decisiones

4.1. La estructura

Al comenzar a trabajar en el programa, inicialmente dediqué unos dias a analizar las posibles ideas para su implementación.

Al investigar la estructura Trie, encontré una idea relativamente simple y que parecía ajustarse a las necesidades del programa. En ese momento consideré también la implementación de estructuras similares como CTrie, que hace un uso mas eficiente de la memoria, pero elegí quedarme con la estructura mas basica y posteriormente, con el trabajo ya terminado, considerar modificarla por una mejor.

Sin embargo, mas adelante, al decantarme por el algoritmo de Aho-Corasick, conservé la estructura inicial por simplicidad, entendiendo que utilizar otras alternativas de igual manera implicaría un mayor costo en términos de tiempo de ejecución en contraposición con la reducción en memoria.

4.2. El algoritmo de parseo

La elección del algoritmo de parseo fue la principal problemática del trabajo práctico. En un principio, realicé toda la implementacion de una solución basada exclusivamente en Tries, donde el algoritmo de parseo era mas simple. Este consistía en recorrer el texto caracter a caracter, de igual manera que la solución propuesta, pero con una diferencia en la forma de recuperarse de errores.

En el caso de estar recorriendo el Trie y encontrarnos con que el próximo caracter del texto no daba lugar a un prefijo válido en el árbol, simplemente reiniciaríamos las variables del bucle for al punto donde comenzamos a leer esa cadena, y avanzaríamos una posición para marcar el salteo de un caracter. Siguiendo con esta idea, reiniciariamos el nodo con el que recorremos el Trie a la raíz, y volveríamos a intentar formar una palabra válida desde el caracter siguiente.

Esta idea funcionaba, y significaba una solución mucho mas simple al problema, ya que el programa se reducía a la construcción del Trie y el posterior parseo. El motivo que me hizo replantearme la solución fue la complejidad temporal del parseo, ya que con un diccionario y un listado de frases particularmente desfavorables, la cantidad de operaciones sería cuadrática. Incluso en casos no tan desfavorables, este reset de los parámetros no parecía eficiente.

Al finalizar con la implementación de esta idea, y buscando maneras de volver mas eficiente al programa, encontré el algoritmo de Aho-Corasick.

Inicialmente lo descarté, pues no era una solución explícita al problema planteado en el trabajo práctico, ya que el algoritmo original resuelve un problema mas complejo. Luego, retomé la idea y noté que realizando pocos cambios a la implementación con la que ya contaba podía llegar a un algoritmo que resolviera el problema.

Finalmente, teniendo las dos versiones terminadas, realicé testeos con diferentes diccionarios y listados de frases en ambos programas. En ese momento fue cuando me decidí por el programa entregado, pues los tiempos de ejecución fueron menores y el consumo de memoria era similar.

Esta reducción en la complejidad temporal es considerablemente mas significativa cuando el largo del archivo a parsear incrementa, pues el costo extra de construír el autómata se ve amortizado. En casos de prueba mas simples, los tiempos de ejecucion fueron similares, aunque el programa actual obtuvo mejores resultados en todos los casos considerados.

4.3. La incorporación del campo *valid_suffix*

Una de las últimas modificaciones previo a la entrega del programa se trató de la incorporación del campo *valid_suffix* a la estructura Trie. Como se menciona brevemente en el apartado *Parseo*, este campo deriva directamente del campo *fail*. Esto es, se puede omitir en la implementación y calcular este enlace reiteradas veces al realizar el parseo, simplemente siguiendo el camino de enlaces *fail* y añadiendo un condicional, como podemos ver debajo:

VERSIÓN PRE-SUFFIX

```
1 while (temp_node != NULL && temp_node != root) {
2       if (trie_is_word(temp_node))
3         update_longest_word(longest_word, temp_node, i);
4       temp_node = temp_node->fail;
5 }
```

Los resultados obtenidos tras incorporar este preprocesamiento al autómata fueron variados. Como podríamos esperar, la complejidad temporal mejoró en diccionarios como *1000english* o *english* (5.2), que presentaban mayor complejidad, llegando a obtener tiempos incluso 9% inferiores. Sin embargo, al utilizarse diccionarios mas simples como *duhalde_dict*, el costo extra del preprocesamiento no se vió amortizado, incrementando ligeramente los tiempos de ejecución.

5. Extras

5.1. Generador de casos de prueba

Adicionalmente al programa principal, en la carpeta del proyecto se encuentra un programa simple que utilicé a lo largo del desarrollo del trabajo práctico. El mismo puede ser compilado con *make* siguiendo las indicaciones del archivo readme.

Este programa recibe por consola el nombre de un diccionario, una cantidad k de líneas, y la longitud promedio l de las mismas. Luego, genera un archivo txt en la carpeta phrases que consiste de k líneas de texto de longitud $l \pm l/4$ creadas aleatoriamente a partir de prefijos del diccionario ingresado.

El generador de casos de prueba por defecto acepta un diccionario con un máximo de 1000 palabras, de otra manera se producirá un error.

5.2. Casos de prueba

En la carpeta dicts se pueden encontrar los siguientes diccionarios:

1. *bee_movie_dict.txt*, que cuenta con un diccionario extraído del guión de la película Bee Movie (en inglés).
2. *english.txt*, un diccionario inglés completo.
3. *1000english.txt*, un diccionario con las 1000 palabras mas comunes del inglés.
4. *duhalde_dict.txt*, el diccionario del ejemplo encontrado en el enunciado.

En la carpeta phrases se pueden encontrar los siguientes archivos de frases:

1. *bee_movie_script.txt*, el guión de la película Bee Movie (en inglés).
2. *duhalde_300000_375-625.txt* un caso de prueba generado a partir del generador de casos de prueba, que cuenta con 300000 líneas de entre 375 y 625 caracteres de longitud.

5.3. Algunos resultados

Utilizando el generador de casos de prueba creamos una serie de archivos de test. En todos los casos hemos considerado una longitud aleatoria de las frases de entre 375 y 625 caracteres, y los resultados obtenidos son el promedio de 10 casos iguales.

Algunos resultados con el diccionario *duhalde_dict* fueron:

DICCIONARIO *DUHALDE_DICT* SOBRE FRASES DE *DUHALDE_DICT*

líneas	construcción del autómata (s)	parseo (s)	total (s)
30.000	0,0000762	0.287	0,287
300.000		2.816	2,816
3.000.000		27.658	27.658

Por otro lado, utilizando el diccionario *1000english*, obtuvimos los siguientes resultados:

DICCIONARIO *1000ENGLISH* SOBRE FRASES DE *1000ENGLISH*

líneas	construcción del autómata (s)	parseo (s)	total (s)
30.000	0,0014954	0.495	0,49645
300.000		4.830	4,83149
3.000.000		47.624	47.625495

Por último, estos fueron los resultados obtenidos al generar los archivos de frases a partir de un diccionario y parsearlos con uno diferente:

RESULTADOS CON *DUHALDE_DICT* SOBRE FRASES DE *1000ENGLISH*

líneas	construcción del autómata (s)	parseo (s)	total (s)
300.000	0,0000762	2,043	2,043
3.000.000		20,461	20,461

RESULTADOS CON *1000ENGLISH/ENGLISH* SOBRE FRASES DE *DUHALDE_DICT*

líneas	diccionario	construcción del autómata (s)	parseo (s)	total (s)
300.000	1000english	0,0014954	3,7699	3,77139
3.000.000			36,750	36,75149
300.000	english	0,881	5.8824	6,7634
3.000.000			59.164	60,045

Estas (limitadas) pruebas sugieren que el tiempo de ejecución del programa es lineal respecto al tamaño de la entrada una vez hemos fijado un diccionario, lo que coincide con la complejidad esperada del algoritmo de Aho-Corasick.