Proyecto - ADA

Andrés Lostaunau

Junio 2021

1 Problema del Trie eficiente

1.1 Introducción

El Trie nos presenta una estructura de datos parecida a la de un árbol parejo, cada arista tiene una valor de un caracter que pertenece a un alfabeto Σ y además cumple con un regla específica (en este caso que todos los aristas de un mismo nodo padre están en orden alfabético). Un ptrie cumple con los mismos requerimientos solo que todas las hojas de la misma altura pueden cambiar de nivel en cualquier nivel al igual que una permutación p=h!. Y un S-ptrie además de lo indicado recibe un conjunto de cadenas S para construir la estructura. El problema requerido es ver cual de las permutaciones posibles del S-ptrie tiene la menor cantidad de aristas.
Para resolver el problema primero es necesario identificar el causante. En este caso el enunciado es muy claro, son las posibles permutaciones. Dejando de lado este problema debemos tener bien definido el resto de fragmentos del algoritmo implementado.

1.2 Implementación General

Lo primero a definir es que el código ha sido implementado en C++. La estructura tomada del S-ptrie se interpretó como una matriz de caracteres del tamaño $m*n$, donde n es igual número de cadenas de S y m es igual a la longitud de cada cadena. Cada uno de los caracteres representa a un arista del S-ptrie siendo que todos los caracteres en el mismo eje horizontal se encuentran en la misma altura. Los caracteres @ representan a la ausencia de arista. Si un arista es precedido por un arista ausente, entonces el arista perteneces al predecesor del arista anterior. El número de aristas es igual a $n*m$ - r, donde r es igual al número de aristas ausentes (Figure 1).

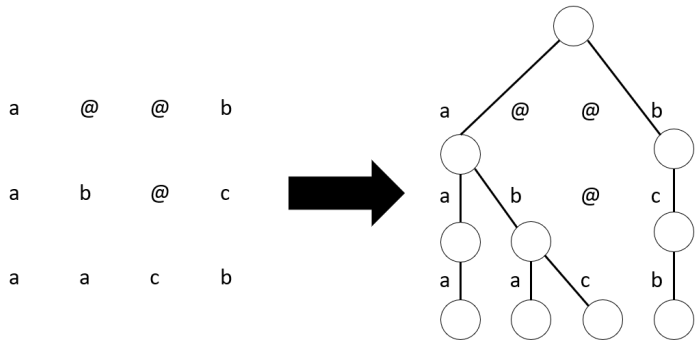

Figure 1: Representación del S-ptrie

1.2.1 Transpose Algorithm

El input original es recibido como un vector de strings con las dimensiones *n*m*, inverso a las dimensiones necesitadas para construir un S-ptrie, para esto se implementa la función *transpose(input)* que sirve para obtener la matriz transpuesta del input.

```
vector<string> transpose(vector<string> input){
    vector<string> t_matrix;
    for(int i = 0; i < input[0].size(); i++){
        t_matrix.emplace_back("");
        for(int j = 0; j < input.size(); j++){
            t_matrix[i].push_back(input[j][i]);
        }
    }
    return t_matrix;
}
```

El valor de *m* está representado como *input.size()*, al igual que n está representado por *input[0].size()*, esto se da múltiples veces en el código. El algoritmo son dos iteraciones anidadas, por ende el tiempo de ejecución del algoritmo es de *O(nm)*.

1.2.2 Build Algorithm

```
int build_strie(vector<string>* input){
    int removed_edge_counter = 0;
    char current_char = ' ';
    for(int i = 0; i < input->size(); i++){
        for(int j = 0; j < input[0].size(); j++){
            if((*input)[i][j] != current_char){
                current_char = (*input)[i][j];
```

```
            }else if(i - 1 < 0) {
                (*input)[i][j] = '@';
                removed_edge_counter++;
            }else if((*input)[i - 1][j] == '@') {
                (*input)[i][j] = '@';
                removed_edge_counter++;
            }
        }
        current_char=' ';
    }
    return removed_edge_counter;
}
```

El algoritmo de construcción del S-ptrie es algo interesante porque se optó por una estructura de datos más liviana a la propuesta por el enunciado, pero igual de funcional. Al usar una matriz como estructura de datos el espacio que toma se reduce de $O(mn\Sigma)$ a $O(mn)$ y el algoritmo de construcción como se ve aquí también comparte el mismo tiempo de ejecución. Ya se explicó el como es que se colocan los aristas ausentes. El algoritmo además devuelve el número de aristas ausentes para calcular posteriormente el número de aristas sin necesidad de volver a iterar.

1.2.3 Permutación Óptima

Por último es necesario saber cual de las tantas permutaciones posibles es la óptima para resolver el problema. El principal asunto es que este paso no es sencillo o al menos es muy costoso. Para evaluar todos los casos en condiciones normales tendría un costo inalcanzable de $O(n!)$, por esta razón más adelante veremos diferentes algoritmos que buscan poder realizar esta tarea con una facilidad mucho mayor.

2 Heurística Voraz

Crear un algoritmo voraz usualmente es una tarea sencilla y encima los algoritmos tienden a ser poco costosos. Observando el problema detenidamente puedes notar un pequeño patrón con respecto al número de aristas y el número de caracteres diferentes en los primeros niveles. Mientras menos caracteres diferentes se encuentren dentro de los primeros niveles el número de aristas será menor. Aquí una pequeñas demostración con un S=*aaa,aab,abc,adb* donde se tomarán dos instancias donde se puede apreciar este efecto más. (Figure 2)

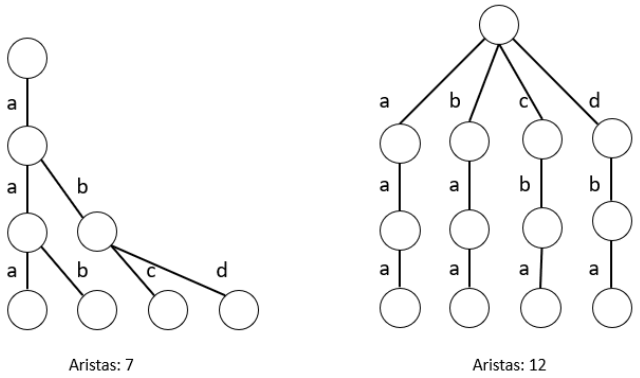

Figure 2: Ejemplo del impacto del orden de los niveles

Bajo este razonamiento es posible crear un algoritmo que ordene la matriz priorizando los niveles con menor repetición primero. Para este algoritmo se han empleado dos partes, la creación de un vector de *score* y realizar un sort.

2.1 Implementación

2.1.1 Score Vector

```
vector<int> get_score(vector<string> t_matrix){
    int score;
    char actual_char;
    vector<int> score_vec;
    for(int i = 0; i < t_matrix.size(); i++){
        actual_char = ' ';
        score = 0;
        for(int j = 0; j < t_matrix[0].size(); j++){
            if(t_matrix[i][j] != actual_char){
                score++;
                actual_char=t_matrix[i][j];
            }
        }
        score_vec.emplace_back(score);
    }
    return score_vec;
}
```

La finalidad del vector de score es usarlo como apoyo para el algoritmo de sort. El tiempo de ejecución es de *O(mn)*

2.1.2 Sorting Algorithm

El algoritmo de sort a elección fue un MergeSort personalizado que recibía tanto el vector de score y la matriz, de modo que el algoritmo ordenaba el vector de score y la matriz imitaba los cambios que se realizaban en el vector. Para no llenar de mucho código solo compartiré el algoritmo de Merge.

```
void merge(vector<int>* score_vec, vector<string>* input, int l, int
    mid, int r){
    vector<int> lArray, rArray, orderedArray;
    vector<string> lMatrixArray, rMatrixArray, orderedMatrixArray;
    for(int i = l; i < mid; i++){
        lArray.push_back((*score_vec)[i]);
        lMatrixArray.push_back((*input)[i]);
    }
    for(int i = mid; i < r; i++){
        rArray.push_back((*score_vec)[i]);
        rMatrixArray.push_back((*input)[i]);
    }
    int rIndex = 0, lIndex = 0;
    while(lIndex != lArray.size() || rIndex != rArray.size()){
        if(lIndex < lArray.size() && (rIndex == rArray.size() ||
            (lArray[lIndex] < rArray[rIndex]))){
            orderedArray.emplace_back(lArray[lIndex]);
            orderedMatrixArray.emplace_back(lMatrixArray[lIndex]);
            lIndex++;
        }else{
            orderedArray.emplace_back(rArray[rIndex]);
            orderedMatrixArray.emplace_back(rMatrixArray[rIndex]);
            rIndex++;
        }
    }
    for(int i = l; i < r; i++){
        (*score_vec)[i] = orderedArray[i-l];
        (*input)[i] = orderedMatrixArray[i-l];
    }
}
```

Ya que es un MergeSort el tiempo de ejecución del algoritmo es de O(mlg(m)).

2.2 Ejecución

```
struct Answer{
    vector<string> sptrie;
    int edges;
};

Answer greedy_algorithm(vector<string> const input){
    vector<string> t_matrix = transpose(input);
```

```
    sort(&t_matrix);
    int removed_edges = build_strie(&t_matrix);

    Answer ans;
    ans.edges = (t_matrix.size()*t_matrix[0].size()) - removed_edges;
    ans.sptrie = t_matrix;
    return ans;
}
```

Esta es la ejecución del algoritmo voraz que devuelve tanto el S-ptrie como el número de aristas. El tiempo de ejecución del algoritmo es igual a la suma de los tres fragmentos mencionados previamente de *O(mn)*, *O(mlg(m))* y *O(mn)* dejando al algoritmo con un tiempo de ejecución de *O(mn + mlg(m))* Se realizó una prueba con un *S=aaa,baa,bac,cbb*, siendo el resultado el de Figure 3.

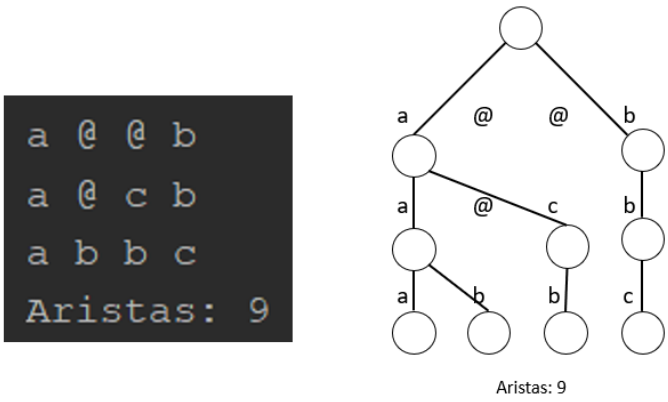

Figure 3: Ejemplo del impacto del orden de los niveles

El resultado es el mínimo entre las permutaciones posibles, por ende es correcto para este caso.

2.3 Demostración

El algoritmo que determinaba la permutación adecuada fue presentado como un algoritmo voraz correcto, sin embargo para poder determinar su exactitud es necesario demostrar la propiedad de la elección voraz y la propiedad de la subestructura óptima.
Antes de entrar a las demostraciones específicas es necesario explicar la ley del score. Si tenemos un vector de score V con n niveles y un vector de aristas A con n niveles que representan el número de aristas entre el nivel n-1 y n en el S-ptrie podemos decir que siempre se cumplirá que para todo $v_i \epsilon V$ tiene un $a_i \epsilon A$ donde $v_i \leq a_i$, $a_{i-1} \leq a_i$ y $a_i \leq n$ a excepción de $a_1 = v_1$.

2.3.1 Elección Voraz

Lema: Si existe un vector de score ordenado V donde V=$\{v_1, v_2, ..., v_n\}$ y $v_i \epsilon V, v_{i-1} < v_i$, debe ser el óptimo.
Prueba: Si existe un vector X donde con n valores diferentes y con el orden $x_i \epsilon x, 1 \leq x_i \leq n, x_{i-1} < x_i$ será óptimo.
Sabiendo que $a_1 = x_1 = 1$ y que $a_{i-1} \leq a_i$ es posible determinar que $x_1 \leq a_2$ y considerando que $v_i \leq a_i$ también que $x_2 \leq a_2$ pero como sabemos que $x_1 = 1$ entonces obtenemos que $x_1 < x_2$ y lo mismo se puede aplicar hasta x_n llegando al final a la forma óptima donde $x_1 < x_2 < ... < x_n$

2.3.2 Subestructura Óptima

Lema: Un vector V que contiene dos o más elementos $v_i = v_j$ no cumple con la elección voraz.
Proof: Tenemos un vector X que contiene a un par de elementos $x_i = x_j$
Si asumimos que j = i+1 tendremos que $x_i = x_i + 1, x_i \leq a_i, x_{i+1} \leq a_{i+1}$ y $a_i \leq a_{i+1}$ lo que nos dejaría en $x_i \leq a_i \leq a_{i+1}$. Como no es posible identificar si $a_i < a_{i+1}$ o $a_i = a_{i+1}$ y eso influye en el valor óptimo, entonces X no es pertenece al conjunto de respuestas cuando existe un $x_i = x_j$.

3 Recurrencia

El problema del algoritmo presentado anteriormente es que al momento del ordenamiento este no distingue de niveles con el mismo score por ende si el orden de niveles con el mismo score fuese relevante el algoritmo no aseguraría su respuesta. Es muy poco probable que todos los niveles tengan el mismo puntaje así que no se apostar como un algoritmo confiable para resolver este problema. Observando nuevamente el problema es posible darle un nuevo enfoque, un enfoque recurrente.
Analizando nuevamente la matriz transpuesta es posible notar otro comportamiento, cada vez que una cadena no se relaciona con otra cadena en un nivel se da que por el resto de niveles la cadena no va a relacionarse, por ende siempre va a generar una arista. Entonces si es que es posible separar a todas las cadenas que ni bien deje de relacionarse. Mientras que una cadena se siga relacionando con otra sigue siendo tomada como candidata y se sigue tomando en consideración. (Figure 4)

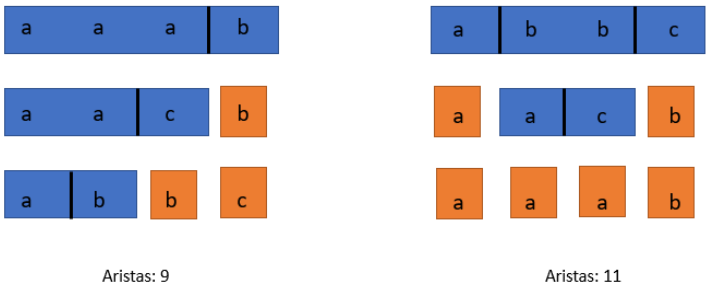

Figure 4: Ejemplo del conteo de aristas

Entonces con esta interpretación podemos definir un subproblema de la siguiente forma:

$$OPT(m,i,j) = \begin{cases} 0 & ,\text{m} = 0 \\ m & ,\text{i} = \text{j} \\ (OPT(m-1,i,a)+1)+ & \\ (OPT(m-1,a+1,b)+1) & \\ +...+ & \\ (OPT(m-1,z+1,j)+1) & ,\text{cc} \end{cases} \tag{1}$$

Para interpretarlo con más facilidad la variable m representa la altura dentro del S-ptrie, la variable i y t son el índice inferior y superior respectivamente y las variables a,b,..,z representan los índices en donde ocurre una división. Con esta estructura podemos crear un grafo acíclico dirigido que permita una mejor visualización de lo que pasa.

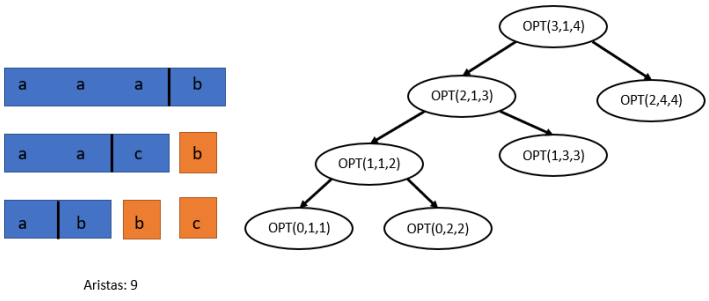

Figure 5: Ejemplo con S={aaa,aab,acb,bbc}

Aquí se puede apreciar como el problema se descompone subproblema por subproblema de modo que devuelve el resultado esperado.