Proyecto de Análisis y Diseño de Algoritmos

Loja Zumaeta, Alex Jose; Neira Riveros, Jorge Luis

1. Pregunta 1: Heurística Voraz

Analice, diseñe e implemente una heurística voraz para el problema MIN-TRIE. Su algoritmo no deberá encontrar necesariamente la respuesta óptima.

Entrada del algoritmo: Una secuencia S de n cadenas de longitud m sobre el alfabeto Σ.

Salida del algoritmo: Un S-ptrie y su número de aristas.

Tiempo de ejecución del algoritmo: $O(nm + mlgm)$.

Espacio utilizado por el algoritmo: $O(nm|\Sigma|)$.

Elección voraz: Insertar primero la posición que tiene menos caracteres distintos

Entrada: Un S_ptrie vacío, una secuencia de cadenas $S = (s_1, s_2, \ldots, s_n)$, y el conjunto de posiciones $P = 0, 1, \ldots, m-1$ a insertar.

Salida: Un S-ptrie con una cantidad mínima de aristas.

Pseudocódigo

```
1: function MIN-EDGES-SPTRIE(S-PTRIE, S, P)
2:     if P = ∅ then
3:         return
4:     end if
5:     P_i* = P_i : A(P_i) = min{A(P_j) : P_j ∈ P}
6:     for j = 1 → n do
7:         S-ptrie.insert(s_j, P_i*)    ▷ Inserta el caracter P_i* de s_j
8:     end for
9:     P' = P\{P_i*}
10:    MIN-EDGES-SPTRIE(S-ptrie, S, P')
11: end function
```

Nota 1: El algoritmo debe ser llamado con un S-ptrie vacío, el conjunto de cadenas S y el conjunto de posiciones P.

Nota 2: La función $A(P_i)$ cuenta los caracteres distintos en la posición P_i de las cadenas. Este procesamiento debe hacerse antes del algoritmo.

Lema 1: (Elección voraz). Existe una solución óptima para el problema que inicia la inserción de caracteres de las cadenas de S en la posición P_i en cada iteración de la recursión, tal que $A(P_i) = min\{A(P_j) : P_j \in P\}$.

Prueba: Sea X el S-ptrie óptimo del problema. Este fue generado siguiendo una permutación ordenada de las posiciones en P. Sea esta permutación $p = (p_1 p_2 ...)$, entonces p corresponde con la secuencia creciente $A = (A_{p_1}, A_{p_2}, ...) : p_i \in p \wedge A_{p_i} = A(P_i)$. Demostraremos que la solución óptima se da cuando A es creciente, es decir, no tiene inversiones.

Suponga por contradicción que A tiene por lo menos una inversión. Tome la inversión (i, j) en A tal que $j - i$ es mínimo (i y j son índices p_i de A).

Mostraremos que $j - i = 1$. Suponga por contradicción que $j - i > 1$. Entonces existe un índice k tal que $i < k < j$. Si $A_i > A_k$ entonces (i, k) es una inversión de tamaño menor a la escogida (i, j), lo cual es una contradicción a la elección mínima. Si $A_i < A_k$ entonces, como $A_i > A_j$, tenemos $A_j < A_k$ y (k, j) sería una inversión de tamaño menor a la escogida (i, j), lo cual contradice a la elección mínima.

Del párrafo anterior, entonces sea (i, j) una inversión de tamaño mínimo en A. Sea A' la secuencia que resulta de intercambiar A_i y A_j. Esta nueva secuencia A' genera una nueva solución X'.

Demostraremos que X' es solución al problema. A' genera una permutación de posiciones, la cual si se ejecuta en el algoritmo, generamos un nuevo S-ptrie, no necesariamente con cantidad de aristas mínima. Como X' genera un S-ptrie, entonces es una solución al problema.

Mostraremos que la cantidad de aristas en X' es menor o igual a la cantidad de aristas en X.

Como $j = i + 1$, entonces el intercambio se realiza entre 2 índices consecutivos de A. Entonces las aristas en el nivel p_i de X estarán un nivel arriba a las de p_j, como muestra la siguiente imagen.

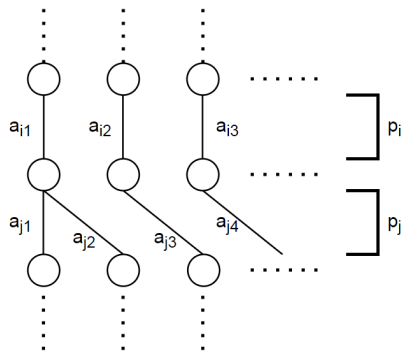

Fig 1.1: Posiciones p_i y p_j de la solución X

Sean a_{i1}, a_{i2}, ... los caracteres únicos en p_i, y a_{j1}, a_{j2}, ... los de p_j. Como $A_i > A_j$, entonces $|p_i| > |p_j|$. Además, se puede representar la secuencia $p_i p_j$ del S-ptrie como un grafo bipartito.

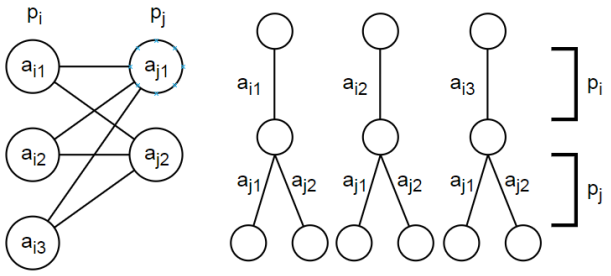

Fig 1.2: Posiciones p_i y p_j de la solución X_1 en un grafo bipartito

La figura 1.2 muestra una relación total entre los caracteres de p_i y p_j, es decir, existen cadenas en S que contienen a cada par de caracteres $a_i a_j$ en las posiciones $p_i p_j$. La figura también muestra al S-ptrie X_1 que generaría el grafo bipartito. Note que p_i tendrá una arista por caracter, y p_j tendrá A_j aristas por cada arista en p_i. La cantidad total de aristas sería $A_i + A_i \times A_j$.

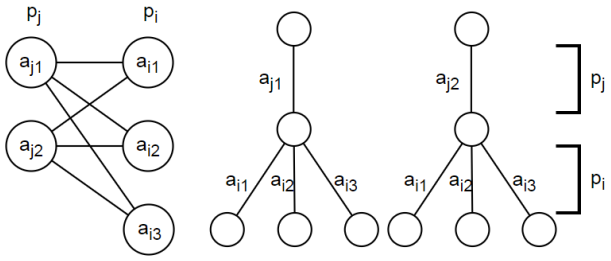

Fig 1.3: Posiciones p_j y p_i de la solución X_2 en un grafo bipartito

Cuando intercambiamos la inversión, obtenemos el grafo bipartito de la figura 1.3, y su correspondiente S-ptrie X_2 que lo genera. Note que la cantidad total de aristas será $A_j + A_i \times A_j$. Como $A_i > A_j$, entonces observamos que la cantidad de aristas disminuye al quitar la inversión, por lo que, X_2 tendrá menos aristas que X_1.

Si el S-ptrie no genera un grafo bipartito completo, note que su grafo puede formarse a partir de la eliminación de aristas de su grafo bipartito completo correspondiente, lo que le quitaría solo 1 arista a ambos S-ptrie X_1 y X_2. Note también que no pueden quedar nodos aislados.

A partir de esta prueba, concluimos que la solución X', la cual se generó por intercambiar las posiciones de una inversión en A de X, es más óptima dado que X es óptimo. Por este motivo, la solución más óptima se dará cuando A no presente inversiones, es decir, sea creciente.

Lema 2: (Subestructura óptima) Sea X la solución óptima para (S, P), entonces $x' = X \backslash \{P_{i*}\}$ es una solución óptima para (S, P').

Prueba: Suponga por contradicción que X' no es óptima. Entonces, existe Y' tal que $aristas(Y') < aristas(X')$. Pero, en este caso $Y = Y' \cup \{P_{i*}\}$ y $aristas(Y) = aristas(Y') + A(P_{i*}) < aristas(X') + A(P_{i*}) = aristas(X)$, lo cual es una contradicción a la optimalidad de X.

Implementación del algoritmo

```
vector<string> cad(n);
vector<unordered_set<int>> charByPosition(m);
for(ll i = 0; i < n; i++){
    cin >> cad[i];
    for(ll j = 0; j < m; j++)
        charByPosition[j].insert(cad[i][j]-'a');
}
```

En primer lugar, creamos un vector de cadenas(cad) en el que almacenaremos las cadenas ingresadas lo cual nos tomará un tiempo de n, ya que esta es la cantidad de inputs que tendremos. Además, un vector de unordered_set, que nos permitirá obtener cuántos caracteres distintos hay en cada posición de las cadenas, y esto a su vez toma un tiempo de $O(1)$ por cada posición. Para poder hacer estas operaciones necesitamos recorrer cada palabra ingresada, esto nos tomará un tiempo de m ya que esta es la longitud de cada input. **Tiempo total:** $O(n * m)$.

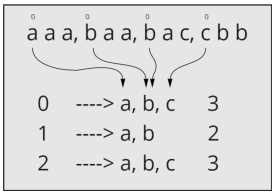

Fig 1.4: Cantidad de caracteres únicos por índice

```
bool sortP(pair<ll, ll> p1, pair<ll, ll> p2){
    if(p1.second == p2.second)  return p1.first < p2.first;
    return p1.second < p2.second;
}
...
vector<pair<ll, ll>> permut;
for(ll i = 0; i < m; i++)
    permut.push_back(make_pair(i, charByPosition[i].size()));
sort(permut.begin(), permut.end(), sortP);
```

Creamos un vector permutación en el que guardaremos la posición y la cantidad de caracteres distintos en él, esto tomará un tiempo de $O(m)$. Después, ordenamos el vector de acuerdo al número de caracteres distintos, lo que tomará $O(mlg(m))$. **Tiempo total:** $O(m + mlg(m))$.

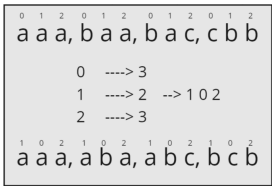

Fig 1.5: Orden óptimo de acuerdo a los índices

```
unordered_set<int> characters;
vector<int> alf;
for(string &s : cad){
  string news = "";
  for(ll i = 0; i < m; i++){
      news += s[permutation[i].first];
      if(!characters.count(s[permutation[i].first] - 'a')){
          characters.insert(s[permutation[i].first] - 'a');
          alf.push_back(s[permutation[i].first] - 'a');
      }
  }
  s = news;
}
```

Luego, reordenaremos los caracteres de las cadenas de acuerdo a la permutación obtenida. En el vector alf guardaremos el orden de caracteres, y usamos un unordered_set (characters) para saber si ya fue insertado en el vector. Como recorreremos las n palabras y por cada palabra sus m caracteres nos tomará un tiempo de n * m. **Tiempo total:** O($n * m$)

```
vector<vector<ll>> sptrie(m * n + 1);
for(ll i = 0; i < m * n + 1; i++)
    sptrie[i].resize(alf.size(), 0);
```

Creamos nuestra matriz S-ptrie. Tendrá como máximo (si todos los caracteres de todas las cadenas son distintas) $m*n+1$ filas (un nodo por fila) y $|\Sigma|$ columnas. **Tiempo total:** O($m * n$).

```
ll parent = 0, node = 0;
for(string s : cad){
    parent = 0;
    for(char c : s){
        if(sptrie[parent][c-'a'] == 0)
            //No existe, la creamos
            sptrie[parent][c-'a'] = ++node;
        parent = sptrie[parent][c-'a'];
    }
}
```

i	a	b	c
0 →	1	7	0
1 →	2	4	0
2 →	3	0	0
3 →	0	0	0
4 →	5	0	6
5 →	0	0	0
6 →	0	0	0
7 →	0	0	8
8 →	0	9	0

Fig 1.6: Matriz del S-ptrie

Finalmente, para cada cadena guardada en cad, la insertamos en el S-ptrie. Como recorremos el vector de palabras(n), y para cada palabra sus caracteres(m) nos tomará n*m **Tiempo total:** O(n * m)

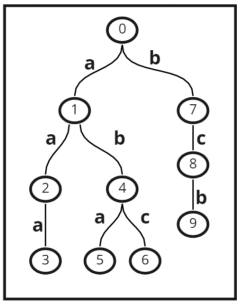

Fig 1.7: S-ptrie con el número mínimo de aristas

Como salida, mostramos el S-ptrie formado, la permutación óptima y la cantidad mínima de aristas.

2. Pregunta 2: Recurrencia

Plantee una recurrencia para $OPT(i, j)$.

Sea $S_{i,j} = (s_i, s_{i+1}, ..., s_j)$. Sea K(ij) el conjunto de posiciones (desde 0 a $m - 1$) en donde todas las cadenas tienen el mismo valor de caracter. Por ejemplo, si S = (aaa, bab, cab) entonces $K(13) = \{1\}, K(12) = \{1\}, K(23) = \{1, 2\}, K(33) = K(22) = K(11) = \{0, 1, 2\}$.

Sea $\overline{S}_{ij} = (\overline{s}_i, \overline{s}_{i+1}, ..., \overline{s}_j)$ el conjunto de cadenas resultantes luego de remover estas posiciones comunes en cada cadena (es decir, cada $\overline{s}_k$ es la cadena s_k truncada). En el ejemplo, $\overline{S}_{13} = (aa, bb, cb), \overline{S}_{12} = (aa, bb, cb), \overline{S}_{23} = (b, c), \overline{S}_{33} = \overline{S}_{22} = \overline{S}_{11} = \emptyset$, donde $\emptyset$ es la cadena vacía.

Sea $\overline{OPT}(i, j)$ el número de aristas de una solución óptima para el subproblema que considera a las cadenas $\overline{s}_i, \ldots, \overline{s}_j$.

Lema 2.1. $OPT(i, j) = \overline{OPT}(i, j) + |K(i, j)|$

La demostración del lema anterior es un tanto técnica debido a la notación usada, pero está basada intuitivamente en lo siguiente: siempre existe un trie óptimo que pone a las posiciones en K(ij) al inicio.

Para esto necesitamos un poco más de notación. Sea R(ij) las posiciones (de 0 a m-1), en donde no todas las cadenas tienen el mismo valor de caracter. En otras palabras, $R(ij) = \{0, 1, ..., m\text{-}1\} \backslash K(ij)$. En el ejemplo, $R(13) = \{0, 2\}, R(12) = \{0, 2\}, R(23) = \{0\}, R(33) = R(22) = R(11) = \{\}$

Note que cada índice en r particiona la secuencia Sij en subsecuencias contiguas, cada una de las cuales está definida por dos índices (i,j). Para cada $r \in R(ij)$, definimos C(i, j, r) como el conjunto de pares de índices que definen tales subsecuencias. En el ejemplo, C(1, 3, 0) = {(1, 1),(2, 2),(3, 3)}, C(1, 3, 2) = {(1, 1),(2, 3)}, C(2, 3, 0) = {(2, 2),(3, 3)}

Lema 2.2.

$$\overline{OPT}(i, j) = \begin{cases} 0 & \text{si i=j} \\ min_{i,j} \in R(i,j)\{\sum_{i'j' \in C(i,jr)} \overline{OPT}(i'j') + |K(i'j')| - |K(i,j)|\} & \text{si i <j} \end{cases} \quad (1)$$

Nomenclatura de variables a utilizar

1) $n =$ número de cadenas en la entrada.
2) $m =$ número de caracteres de cada cadena.
3) $cad =$ vector que guarda las cadenas, irá de 1 a n.
4) $K =$ función mostrada en la demostración.
5) $Ksize =$ función mostrada en la demostración.
6) $path =$ matriz que guarda la posición que divide a la secuencia de cadenas de i a j. Será utilizada para generar el S-ptrie.
7) $opt =$ función recursiva que calcula el valor de $\overline{\text{OPT}}$.
8) $dp =$ matriz utilizada para la programación dinámica.
9) $resiz =$ función que inicializa todas las variables y contenedores a utilizar.
10) $fillC =$ función que calcula C.

Consideraciones de la implementación

1) Para implementar la función K, utilizamos un vector de vectores, donde cada posición guardará un vector de booleanos de tamaño m, uno para cada posición. Si pertenece a K entonces es $True$, sino, pertenece a R y sería $False$. De esta forma implementamos K y R.
2) La función $fillC$ genera el vector de posiciones de acuerdo a un i, j y una posición p. En lugar de guardar pares, guarda consecutivamente todas las posiciones. El primer elemento de cada par está en una posición par, y el segundo, en una posición impar.
3) En el cálculo de OPT se obtiene, además de la menor cantidad de aristas para cada par de posiciones, la posición que debería dividirla. Esta información se guarda en $path$ para generar el S-ptrie.

Generación del S-ptrie

1) El S-ptrie es una matriz de $m * n + 1$ filas (una por cada nodo generado, como máximo pueden ser $m*n+1$ nodos) y $|\Sigma|$ columnas. También tiene un vector pos que representa la posición a la que pertenece cada nodo.
2) Esta matriz se lee: "Desde el nodo i, con el caracter j llegamos al nodo $sptrie$". Para esto, debemos tener una codificación desde 0 para cada caracter del alfabeto.
3) En la generación se utilizará un recorrido por **BFS**, de esta forma, tenemos $edges + 1$ nodos y los nodos hojas que están en el último nivel no guardarán información relevante, esto reduce la cantidad de filas de la matriz.
4) Para generar la matriz, utilizamos un $queue$ que guarda el valor del nodo (el nodo raíz es 0), y las posiciones i y j (al inicio serán 1 y n). Para cada nodo, debemos calcular C con la posición p guardada en $path$ y agregamos al $queue$ a todos los intervalos en los que se divide $i - j$.
5) Para llenar la matriz tenemos 3 casos posibles en el generalizado. Considerar el uso de una matriz auxiliar para cada posición de caracter de cada cadena que guarda si ya se agregó o no.
   - **Caso 1:** Si $K[i][j]$ tiene posiciones de caracteres aun no añadidas, entonces las añadimos primero al $queue$, uno por uno (es decir, procesamos uno y continuamos con la siguiente iteración), y creamos el nodo. A la vez, las marcamos como usadas.
   - **Caso 2:** Si no hay partición del intervalo, entonces agregamos las posiciones de caracteres que aun han sido usadas. Si todas han sido usadas, llegamos a un nodo hoja.
   - **Caso 3:** Si hay partición del intervalo, crear un nodo para cada subintervalo, agregar al $queue$ y marcar como leído.
6) Cuando creamos un nodo, si no es hoja, entonces asignamos la posición que representa en pos.
7) Al final, tenemos la matriz que representa el S-ptrie y el vector de posiciones que representa cada nodo.
8) La complejidad en notación O de la generación del S-ptrie es $O(n^3 * m^2)$, sin embargo, esta cota es muy superior a la complejidad real, ya que la complejidad $O(n^2 * m)$ de la transición de un nodo a otro no sieempre es así, ya que, dependiendo del caso, podría tener complejidad $O(n + m)$ o $O(n * m)$.
9) Es importante resaltar que el orden de las cadenas en el S-ptrie está representado por el número de nodo, y no la posición del nodo en la fila de la matriz. En el ejemplo, en la fila del nodo 2, el nodo 4 figura antes que el nodo 3, sin embargo, el orden de aristas corresponde al orden de nodos, es decir la arista l con nodo 3 está a la izquierda de la arista a con nodo 4.

SPTRIE:

	a	c	d	i	l	m	p	u	v	y
0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	2
2	4	0	0	0	3	0	0	0	0	0
3	0	0	5	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	6	0	0
5	0	0	7	0	0	0	0	0	0	0
6	0	0	8	0	0	0	0	0	0	9
7	0	0	0	0	0	0	10	11	0	0
8	0	0	0	0	0	12	0	0	0	0
9	0	0	0	13	0	0	0	0	14	0

Fig 2.1: Ejemplo de salida de S-ptrie

```
set<char> alf;
for(ll i=1; i<=n; i++){                          //O(n*m)
    for(ll j=0; j<m; j++){
        alf.insert(cad[i][j]);
    }
}
map<char, ll> posAlf;
map<ll, char> alfPos;
ll posAlfab = 0;
for(char c : alf){                               //O(alf)
    posAlf[c] = posAlfab;
    alfPos[posAlfab++] = c;
}
vector<vector<bool>> posUsed(n+1);
for(ll i=1; i<=n; i++)  posUsed[i].resize(m, 0);//O(
    n)
vector<ll> pos; //pos[nodo] = posicion de 0 a m-1
vector<vector<ll>> sptrie(m*n+1);
for(ll i=0; i<m*n+1; i++)                        //O(m*n)
    sptrie[i].resize(alf.size());
queue<array<ll, 3>> q;  //nodo, pinicial, pfinal
ll nodo = 0;
q.push({nodo++, 1, n});
while(!q.empty()){      //BFS: O((V+E)*Costo de
    transicion) = O(m*n * (n^2*m)) = O(n^3*m^2)
    auto cur = q.front();
    q.pop();
    ll partition = path[ cur[1] ][ cur[2] ];
    ll nodePar = cur[0];

    bool continua = false;
```

```
    //Asegurar que tenga K's pendientes, si hay, y
        no estan marcados, los pongo primero
    for(ll p=0; p<m; p++){
        //O(m*n)
        //Si los caracteres no estan marcados, y
            pertenece a K[][] entonces entra primero
        if(!posUsed[ cur[1] ][p] && K[cur[1]][cur
            [2]][p]){
            pos.push_back( p );
            for(ll k=cur[1]; k<=cur[2]; k++)
                //O(n)
                posUsed[k][p] = 1;
            sptrie[nodePar][ posAlf[cad[cur[1]][p]]
                ] = nodo;
            q.push({nodo++, cur[1], cur[2]});
            continua = true;
            break;
        }
    }
    if(continua) continue;
    if(partition == -1){                    //O(n^2*
        m)
        bool faltan = false;
        for(ll i=cur[1]; i <= cur[2]; i++){ //O(n^2*
            m)
            for(ll p=0; p<m; p++){          //O(n*m)
                if(!posUsed[i][p]){
                    pos.push_back( p );
                    for(ll k=cur[1]; k<=cur[2]; k++)
                        //O(n)
                        posUsed[k][p] = 1;

                    sptrie[nodePar][ posAlf[cad[i][p
                        ]] ] = nodo;
                    q.push({nodo++, cur[1], cur[2]})
                        ;
                    faltan = true;
                    break;
                }
            }
            if(faltan)  break;
        }
    }else{                                  //O(n+m)
        for(ll i=cur[1]; i<=cur[2]; i++)    //O(n)
            posUsed[i][ partition ] = 1;
        pos.push_back( partition );

        vector<ll> C = fillC(cur[1], cur[2],
            partition);//O(n)
        for(ll p = 0; p < C.size(); p += 2){    //O(
            m)
            sptrie[ nodePar ][ posAlf[   cad[ C[p]
                ][ partition ]   ] ] = nodo;
            q.push({nodo++, C[p], C[p+1]} );
        }
    }
}
nodo = pos.size()-1;
```

3. Pregunta 3: Recursivo

Analice, diseñe e implemente un algoritmo recursivo con complejidad exponencial para el problema Min-Trie-Gen. Su algoritmo deberá encontrar la solución óptima.

Entrada del algoritmo: Un conjunto S de n cadenas de longitud m.
Salida del algoritmo: Un S-ptrie generalizado óptimo y su número de aristas.
Tiempo de ejecución del algoritmo: exponencial.

Consideraciones de la implementación

1) La función recursiva *opt* calcula el número de aristas óptimos desde la cadena i a la j.
2) Tiene complejidad exponencial debido a las llamadas recursivas.

```
ll opt(ll i, ll j){
    if(i == j)  return 0;
    ll mn = LLONG_MAX;
    for(ll r = 0; r < m; r++){
        if(K[i][j][r])  continue;
        ll loc = 0;
        auto C = fillC(i, j, r);
        for(ll k = 0; k < C.size(); k+=2)
            loc += opt(C[k], C[k+1]) + KSize[ C[k]
                ][ C[k+1] ] - KSize[i][j];    //
                Llamada recursiva
        if(loc < mn){
            mn = loc;
            path[i][j] = r;
        }
    }
    return mn;
}

int main(){
    cin >> n >> m;
    //Resize
    resiz();
    for(ll i = 1; i <= n; i++)  //O(n*m)
        cin >> cad[i];

    for(ll i = 1; i < n; i++){  //O(n*m)
        for(ll p = 0; p < m; p++){  //O(m)
            if(cad[i][p] == cad[i+1][p]){
                if(!K[i][i+1][p]) KSize[i][i+1]++;
                K[i][i+1][p] = 1;
            }
        }
    }
    for(ll l = 2; l <= n-1; l++){   //O(n^2*m)
        for(ll i = 1; i <= n - l; i++){
            ll j = i + l;
            for(ll p = 0; p < m; p++){
                if(K[i][i+1][p] && K[i+1][j][p]){
                    if(!K[i][j][p]) KSize[i][j]++;
                    K[i][j][p] = 1;
                }
            }
        }
    }
    ll edges = opt(1, n) + KSize[1][n];
    //Generacion del SPtrie
    return 0;
}

vector<ll> fillC(ll i, ll j, ll r){ //O(n)
    vector<ll> ans;
    ll idx = i;
    char c = cad[i][r];
    for(ll it = i+1; it <= j; it++){    //O(n)
        if(c != cad[it][r]){
            ans.push_back(idx);
            ans.push_back(it-1);
            idx = it;
            c = cad[it][r];
        }
    }
    ans.push_back(idx);
    ans.push_back(j);
    return ans;
}

void resiz(){
    //Inicializacion de valores
}
```

4. Pregunta 4: Memoizado

Analice, diseñe e implemente un algoritmo recursivo con complejidad exponencial para el problema Min-Trie-Gen. Su algoritmo deberá encontrar la solución óptima.

Entrada del algoritmo: Un conjunto S de n cadenas de longitud m.
Salida del algoritmo: Un S-ptrie generalizado óptimo y su número de aristas.
Tiempo de ejecución del algoritmo: $O(n^2m(n+m))$
Tiempo de ejecución del algoritmo: $O(n^2 + nm|\sum|)$

Consideraciones de la implementación

1) La función recursiva *opt* ahora verifica si el valor ya fue calculado ($\neq -1$) antes entrar a la recurrencia.
2) La complejidad se reduce considerablemente a $O(n^3 * m)$, la complejidad de la llamada recursiva es $O(n^2)$ ya que, en el peor caso, llamará a cada par i j.

```
ll opt(ll i, ll j){                    //O(n^3*m)
    if(i == j)  return 0;
    if(dp[i][j] != -1)  return dp[i][j];//O(1)
    ll mn = LLONG_MAX;
    for(ll r = 0; r < m; r++){         //O(m)
        if(K[i][j][r])  continue;
        ll loc = 0;
        auto C = fillC(i, j, r);       //O(n)
        for(ll k = 0; k < C.size(); k+=2)//O(n^3)
            loc += opt(C[k], C[k+1]) + KSize[ C[k]
                ][ C[k+1] ] - KSize[i][j];         //
                O(n^2)
        if(loc < mn){                  //O(1)
            mn = loc;                  //O(1)
            path[i][j] = r;            //O(1)
        }
    }
    dp[i][j] = mn;                     //O(1)
    return dp[i][j];                   //O(1)
}

int main(){
    cin >> n >> m;
    //Resize
    resiz();
    for(ll i = 1; i <= n; i++)         //O(n*m)
        cin >> cad[i];

    for(ll i = 1; i < n; i++){         //O(n*m)
        for(ll p = 0; p < m; p++){     //O(m)
            if(cad[i][p] == cad[i+1][p]){
                if(!K[i][i+1][p]) KSize[i][i+1]++;
                K[i][i+1][p] = 1;
            }
        }
    }
    for(ll l = 2; l <= n-1; l++){      //O(n^2*m)
        for(ll i = 1; i <= n - l; i++){ //O(n*m)
            ll j = i + l;
            for(ll p = 0; p < m; p++){  //O(m)
                if(K[i][i+1][p] && K[i+1][j][p]){
                    if(!K[i][j][p]) KSize[i][j]++;
                    K[i][j][p] = 1;
                }
            }
        }
    }
    ll edges = opt(1, n) + KSize[1][n];
    //Generacion del SPtrie O(n^3*m^2)
    return 0;
}
```

```
vector<ll> fillC(ll i, ll j, ll r){      //O(n)
    vector<ll> ans;
    ll idx = i;
    char c = cad[i][r];
    for(ll it = i+1; it <= j; it++){     //O(n)
        if(c != cad[it][r]){             //O(1)
            ans.push_back(idx);          //O(1)
            ans.push_back(it-1);         //O(1)
            idx = it;                    //O(1)
            c = cad[it][r];              //O(1)
        }
    }
    ans.push_back(idx);                  //O(1)
    ans.push_back(j);                    //O(1)
    return ans;
}
```

5. Pregunta 5: Programación Dinámica

Analice, diseñe e implemente un algoritmo recursivo con complejidad exponencial para el problema Min-Trie-Gen. Su algoritmo deberá encontrar la solución óptima.

Entrada del algoritmo: Un conjunto S de n cadenas de longitud m.
Salida del algoritmo: Un S-ptrie generalizado óptimo y su número de aristas.
Tiempo de ejecución del algoritmo: $O(n^2m(n+m))$
Tiempo de ejecución del algoritmo: $O(n^2 + nm|\sum|)$

Consideraciones de la implementación

1) Utilizamos la matriz auxiliar dp y recorremos los intervalos desde distancia 1 hasta n-1. Es el equivalente iterativo al algoritmo memoizado.
2) La complejidad se mantiene en $O(n^3 * m)$.

```
int main(){
    cin >> n >> m;
    //Resize
    resiz();
    for(ll i = 1; i <= n; i++)           //O(n*m)
        cin >> cad[i];

    for(ll i = 1; i < n; i++){           //O(n*m)
        for(ll p = 0; p < m; p++){       //O(m)
            if(cad[i][p] == cad[i+1][p]){
                if(!K[i][i+1][p]) KSize[i][i+1]++;
                K[i][i+1][p] = 1;
            }
        }
    }
    for(ll l = 2; l <= n-1; l++){        //O(n^2*m)
        for(ll i = 1; i <= n - l; i++){
            ll j = i + l;
            for(ll p = 0; p < m; p++){
                if(K[i][i+1][p] && K[i+1][j][p]){
                    if(!K[i][j][p]) KSize[i][j]++;
                    K[i][j][p] = 1;
                }
            }
        }
    }

    for(ll i = 1; i <= n; i++)  dp[i][i] = 0;//O(n)
    for(ll l = 2; l <= n; l++){              //O(n^3*
        m)
        for(ll i = 1; i <= n - l + 1; i++){ //O(n*m*
            n)
```

```
            ll j = i + l - 1;
            ll mn = LLONG_MAX;
            for(ll r = 0; r < m; r++){      //O(m*n)
                if(K[i][j][r])  continue;
                ll loc = 0;
                auto C = fillC(i, j, r);    //O(n)
                for(ll k = 0; k < C.size(); k+=2)
                    //O(n)
                    loc += dp[ C[k] ][ C[k+1] ] +
                        KSize[ C[k] ][ C[k+1] ] -
                        KSize[i][j];
                if(loc < mn){
                    mn = loc;
                    path[i][j] = r;
                }
            }
            dp[i][j] = mn;
        }
    }
    ll edges = dp[1][n] + KSize[1][n];
    //Generacion de Sptrie      //O(n^3*m^2)
    return 0;
}
```

Ejemplo

Sean las cadenas aaa, bab, cab, cbb, dcb, dcc. Los resultados de los 3 programas son los mismos, mostramos la ejecución de la programación dinámica.

```
6 3
aaa
bab
cab
cbb
dcb
dcc

SPTRIE:
       a       b       c       d
0      1       2       3       0
1      4       5       0       0
2      0       0       6       0
3      0       0       0       7
4      8       0       0       0
5      0       9       10      0
6      0       11      0       0
7      0       12      13      0

node -> pos
0 -> 1
1 -> 2
2 -> 0
3 -> 0
4 -> 0
5 -> 0
6 -> 2
7 -> 2
Edges: 13
```

Fig 5.1: Ejemplo

Casos que se presentan en la generación del S-ptrie

En el caso 1, el BFS encuentra un nodo que pertenece a un intervalo que tiene posiciones en K no usadas. Para el ejemplo, el nodo 3, que pertenece al intervalo de 5 a 6, tiene las siguientes características:

- Tiene al caracter d en la posición 0 que pertenece a $K[5][6]$.
- También, para dicho intervalo, la matriz path determina que el OPT escogió a la posición 2 para dividirlo.

Debido a esas 2 características, debemos escoger primero las posiciones en K, es decir, a la posición 0 y caracter d.

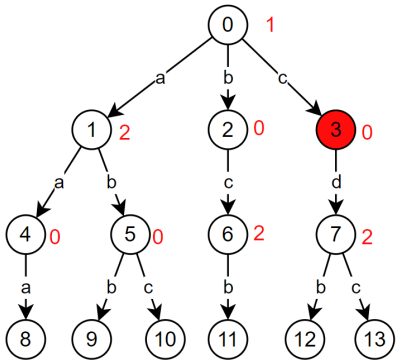

Fig 5.2: Caso 1

En el caso 2, tenemos al nodo 2, cuya posición en la matriz path es -1, lo cual nos puede indicar que ya alcanzamos un nodo hoja. Sin embargo, aún hay posiciones no visitadas. En el ejemplo, las posiciones 0 y 2 aun no fueron visitadas, entonces se visitan.

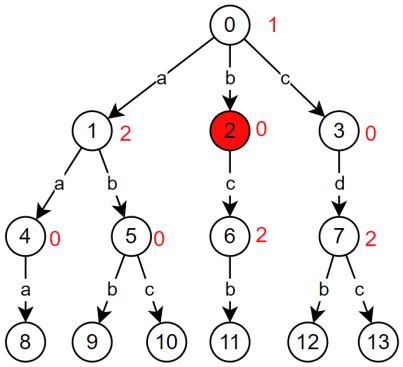

Fig 5.3: Caso 2

En el caso 3, el nodo 1, que corresponde al intervalo 1 a 3, debe dividirse en 2 subintervalos, y continuar con el recorrido en BFS.

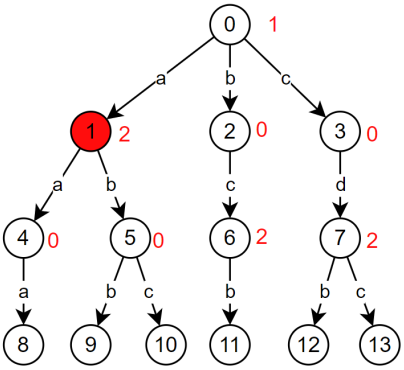

Fig 5.4: Caso3

Si alcanzamos un nodo hoja, no debemos asignarle un valor de posición, pues ya no hay caracteres.

Minicompilador de miniProlog

Un programa en miniProlog consiste en una serie de reglas de la forma $r(s)$, donde r es el nombre de la regla, y s una secuencia de caracteres. Para este caso, utilizaremos los caracteres del alfabeto español en minúsculas. Por ejemplo, sea el siguiente conjunto de reglas:

$$duenho\ ab$$
$$duenho\ ac$$
$$duenho\ ad$$
$$duenho\ ba$$
$$duenho\ ca$$
$$hermanos\ abcd$$
$$hermanos\ acda$$

Con las reglas definidas, podemos realizar consultas donde uno de los caracteres es la incógnita. La consulta busca encontrar caracteres para la incógnita que hagan matching en el orden establecido por las reglas. Por ejemplo

$$duenho\ aX$$
$$duenho\ Xa$$

Deberá devolver

$$X = b, X = c, X = d.$$
$$X = b, X = c.$$

La implementación del minicompilador miniProlog será realizado por partes, en cada una se explica la idea y la implementación.

6. Pregunta 6: Compilación heurística

Entrada del algoritmo: Un programa escrito en miniProlog (archivo de texto).
Salida del algoritmo: Un archivo codificado con tries que permitan responder queries posteriormente. Debe usar como subrutina al algoritmo implementado en la Pregunta 1.

Consideraciones de la implementación

1) El programa lee un archivo con reglas y guarda el conjunto de cadenas en un $unordered_map$ con clave igual al nombre de la regla.
2) Utiliza una clase $trie.h$ implementada con el código del problema 1, es decir, realiza la implementación heurística.
3) Agregamos una función que codifica el S-ptrie y genera un archivo de salida. Esta función utiliza un $priority_queue$ para codificar en forma ordenada a los números de los nodos.
4) La codificación separa los datos con &, donde x son las posiciones que representan los nodos, y y son las aristas del S-ptrie donde y_{ia}, y_{ib}, y_{ic} corresponden al nodo de origen, nodo de destino y caracter respectivamente.

Codificación

$$Rulename\&n\&m\&x\&x_1\&x_2\&...$$
$$\&y\&y_{1a}\&y_{1b}\&y_{1c}\&y_{2a}\&y_{2b}\&y_{2c}\&...$$

```
vector<string> rules;
set<string> rulesNames;
vector<string> chars;
unordered_map<string, vector<string>> mapa;

string getRuleName(string &rule); //Obtiene el
    nombre de la regla
string getCharsByRule(string &rule); //Obtiene los
    caracteres de la regla sin espacios
void read(fstream &file, string &inputName); //Lee
    el archivo de reglas
int main()
{
  fstream file;
  ofstream offile;
  string inputFile = "input.txt";
  string outputFile = "output.txt";
  file.open(inputFile, ios::in);
  offile.open(outputFile);
  read(file, inputFile);
  for(auto y: mapa){
    trie tri(y.first, y.second.size(), y.second[0].
        size(), y.second);
    tri.write(offile, outputFile);
  }
  file.close();
  offile.close();
  return 0;
}
class trie{
    template< typename F, typename S >
    struct PairComparator {
    bool operator()( const pair<F, S>& p1, const
        pair<F, S>& p2 ) const {
            return p1.first > p2.first;
        }
    };
    void write(ofstream &file, string &outputName){
        char charBase = '&';
        file << name << charBase << n << charBase <<
             m << charBase << pos.size() << charBase
            ;
        for(ll i=0; i<pos.size(); i++){
            file << pos[i];
            if(i != pos.size()-1)   file << charBase
                ;
            else file << charBase;
        }
        file << edges << charBase;
        for(ll i=0; i<node; i++){
            //Ordenar de acuerdo al numero
            priority_queue< pair<ll, char> , vector<
                 pair<ll, char> >, PairComparator<ll
                , char> > pq;

            for(ll j=0; j<alf.size(); j++)
                if(sptrie[i][j] != 0)
                    pq.push(make_pair(sptrie[i][j],
                        char(alf[j]+'a')));
            while(!pq.empty()){
                file << i << charBase << pq.top().
                    first << charBase << pq.top().
                    second << charBase;
                pq.pop();
            }
        }
        file << "\n";
    }
};

#endif
```

7. Pregunta 7: Compilación óptima

Entrada del algoritmo: Un programa escrito en miniProlog (archivo de texto).
Salida del algoritmo: Un archivo codificado con tries que permitan responder queries posteriormente. Debe usar como subrutina al algoritmo implementado en la Pregunta 5.

Consideraciones de la implementación

1) El programa lee un archivo con reglas y guarda el conjunto de cadenas en un $unordered_map$ con clave igual al nombre de la regla.
2) Utiliza una clase $trie.h$ implementada con el código del problema 4, es decir, realiza la implementación óptima.
3) Agregamos la misma función que codifica el heurístico. La codificación es la misma. En esta codificación tenemos menos caracteres ya que hay menos aristas y menos nodos.

```
vector<string> rules;
set<string> rulesNames;
vector<string> chars;
unordered_map<string, vector<string>> mapa;
string getRuleName(string &rule); //Obtiene el
    nombre de la regla
string getCharsByRule(string &rule); //Obtiene los
    caracteres de la regla sin espacios
void read(fstream &file, string &inputName); //Lee
    el archivo de reglas
int main()
{
  fstream file;
  ofstream offile;
  string inputFile = "input.txt";
  string outputFile = "output.txt";
  file.open(inputFile, ios::in);
  offile.open(outputFile);
  read(file, inputFile);
  for(auto y: mapa){
    trie tri(y.first, y.second.size(), y.second[0].
        size(), y.second);
    tri.write(offile, outputFile);
  }
  cout << "\nArhivo generado\n";
  file.close();
  offile.close();
  return 0;
}
class trie {
    template< typename F, typename S >
    struct PairComparator {
    bool operator()( const pair<F, S>& p1, const
        pair<F, S>& p2 ) const {
            return p1.first > p2.first;
        }
    };
  void write(ofstream &file, string &outputName){
    char charBase = '&';
        file << name << charBase << n << charBase <<
             m << charBase << pos.size() << charBase
            ;
        for(ll i=0; i<pos.size(); i++){
            file << pos[i];
            if(i != pos.size()-1)   file << charBase
                ;
            else file << charBase;
        }
        file << edges << charBase;
        for(ll i=0; i<=nodo; i++){
            //Ordenar de acuerdo al numero
            priority_queue< pair<ll, char> , vector<
                 pair<ll, char> >, PairComparator<ll
                , char> > pq;

            for(auto c : alf){
                if(sptrie[i][posAlf[c]] != 0)
                    pq.push(make_pair(sptrie[i][
                        posAlf[c]], c));
            }
            while(!pq.empty()){
                file << i << charBase << pq.top().
                    first << charBase << pq.top().
                    second << charBase;
                pq.pop();
            }
        }
        file << "\n";
  }
};

#endif
```

8. Pregunta 8: Parser

Entrada del algoritmo: Un archivo codificado con tries que permita ejecutar queries
Salida del algoritmo: Respuesta a los queries.

Consideraciones de la implementación

1) Lee el archivo codificado, tiene funciones para decodificar y generar los S-ptries de las reglas. Cada S-ptries se representa como una lista de adyacencia donde los vértices son los nodos no hojas y las aristas son pares de nodo y caracter.
2) Lee el archivo de las consultas. Si tenemos la regla, entonces realiza un **DFS**. Mientras existan los caracteres de las aristas en la posición correspondiente en la consulta seguirá avanzando. Cuando encuentre X, guarda las aristas como posibles candidatos y continúa recorriendo. Si llega a un nodo hoja, entonces el posible candidato se agrega a un vector de caracteres res. Al terminar el algoritmo, tenemos las respuestas de las consultas en res.
3) El **DFS** tiene una complejidad $O(m * n)$ ya que tiene $O(m * n)$ nodos y $O(m * n)$ aristas.

```
Parser(){
    cout << "\nParser creado\n";
}

void parse(bool opt){
    fstream trieF;
    string trieFile;
    if(opt)
        trieFile = "opt.trie";
    else
        trieFile = "heu.trie";

    trieF.open(trieFile, ios::in);
    readTrie(trieF);
    trieF.close();
    cout << "\nParser terminado\n";
}

void queryExecuter(string inputFile){
    fstream file;
    file.open(inputFile, ios::in);
```

```
    if(file.is_open()){
        string temporal;
        while(getline(file, temporal)){
            string rule = getRuleConsulta(temporal);
            if(rulesNames.count(rule) == 0){
                cout << "No existe regla " << rule
                    << " para la consulta: " <<
                    temporal << "\n";
                continue;
            }
            string consulta = getCharsByRuleConsulta
                (temporal);
            if(consulta.size() != rulesM[rule]){
                cout << "La consulta tiene muchos
                    caracteres\n";
                continue;
            }
            vector<char> res;
            //SPTrie: tries[rule] = vector<vector<
                pair<ll, char>>>;
            dfs(tries[rule], triesPos[rule], res, 0,
                 triesPos[rule].size(), consulta);

            cout << temporal << "\n\t";
            if(res.size() == 0) cout << "Consulta no
                 encontrada\n";

            for(ll i = 0; i < res.size(); i++){
                cout << "X = " << res[i];
                if(i != res.size() - 1)
                    cout << ", ";
                else
                    cout << ".\n";
            }
        }
    }else{
        cout << "\nParser. Archivo de consultas, no
            se abrio\n";
    }
    file.close();
    cout << "\nConsultas finalizadas\n";
}

void readTrie(fstream &file){
    if(file.is_open()){
        cout << "\nInicia parseo\n";

        string temporal;
        unordered_set<string> ruleNew;
        while(getline(file, temporal)){
            //Primera regla
            string rule = getRuleName(temporal);
            if(rulesNames.count(rule) == 1){
                rulesNames.erase(rule);
                rulesN.erase(rule);
                rulesM.erase(rule);
                tries.erase(rule);
                triesPos.erase(rule);
            }
            if(ruleNew.count(rule)){
                cout << "\nRegla repetida\n";
                continue;
            }
            ruleNew.insert(rule);
            rulesNames.insert(rule);
            rulesN[rule] = getNumber(temporal);
            rulesM[rule] = getNumber(temporal);
            ll posSize = getNumber(temporal);
            vector<ll> pos;
            for(ll i=0; i<posSize; i++){
                pos.push_back(getNumber(temporal));
            }
            ll edges = getNumber(temporal);
            vvplc adj(edges+1);//Hay edges + 1 nodos
            for(ll i=0; i<edges; i++){
                if(temporal.size()==0) break;
                ll nodeA = getNumber(temporal);
                ll nodeB = getNumber(temporal);
                char caracter = getCar(temporal);
                adj[nodeA].push_back({nodeB,
                    caracter});
            }
            tries[rule] = adj;
            triesPos[rule] = pos;
        }
    }else{
        cout << "\nParser. Archivo no leido\n";
    }
}

void dfs(vector<vector<pair<ll,char>>> &adj, vector<
    ll> &pos, vector<char> &res, ll node, ll limit,
    string consulta, char possible = ' '){
    if(node >= limit){
        if(possible != ' ')
            res.push_back(possible);
        return;
    }
    for(auto v : adj[node]){
        if(possible != ' '){
            if(consulta[pos[node]] == v.second){
                dfs(adj, pos, res, v.first, limit,
                    consulta, possible);
            }
        }else{
            if(consulta[pos[node]] == 'X'){
                dfs(adj, pos, res, v.first, limit,
                    consulta, v.second);
            }else{
                if(consulta[pos[node]] == v.second){
                    dfs(adj, pos, res, v.first,
                        limit, consulta, possible);
                }
            }
        }
    }
}

void showRules(){
    cout << "Reglas en memoria:\n";
    for(string s : rulesNames)
        cout << "\t" << s << "\n";
    cout << "\n";
}
void eraseRule(string rule){
    if(rulesNames.count(rule) > 0){
        rulesNames.erase(rule);
        rulesN.erase(rule);
        rulesM.erase(rule);
        tries.erase(rule);
        triesPos.erase(rule);
        cout << "\nRegla " << rule << " eliminada\n"
            ;
    }else
        cout << "\nNo existe la regla " << rule << "
            \n";
}
void showRule(string rule){
    if(rulesNames.count(rule) > 0){
        cout << "TODO: Imprimir cadenas de la regla"
            ;
    }else
        cout << "\nNo existe la regla " << rule << "
            \n";
}
```

9. Pregunta 9: Minicompilador miniProlog

En esta pregunta deberá implementar un minicompilador de miniProlog usando las preguntas 6, 7 y 8. Este minicompilador recibirá archivos de texto escritos en miniProlog. Para

compilar, el usuario tendrá dos opciones. La primera usará el preprocesamiento heurístico y la segunda el óptimo. En este paso es donde se usan las preguntas 6 y 7, respectivamente. Para ejecutar, el compilador recibirá un archivo de texto de consultas, el cual deberá procesar con el archivo generado en el paso anterior.

Estructura del minicompilador MiniProlog

- **Headers:**
  - **Compiler**: Clase que recibe el texto de input, lo procesa y obtiene las reglas y los caracteres para despues crear el S-ptrie dependiendo de la opción indicada en el main y codificarlos en ".trie". Si es heurístico genera "heu.trie", y si es óptimo, "opt.trie".
  - **Parser**: Clase que decodifica los archivos ".trie", los procesa y ejecuta las consultas. Para las consultas se realiza un DFS.
  - **lib**: Funciones útiles que se usarán en diversas clases.
  - **trieHeu**: Clase que crea una instancia de un sptrie usando la heuristica voraz.
  - **trieOpt**: Clase que crea una instancia de un S-ptrie usando programación dinámica.
- **Cpp:**
  - **main**: Programa principal donde se inicia el menú.
  - **generadorReglas**: Genera reglas aleatorias que se usarán para el análisis experimental.

Compiler

```
Compiler(){
    cout << "\nCompilador creado\n";
}

void compile(string inputFile, bool opt){
    rules.clear();
    rulesNames.clear();
    mapa.clear();
    fstream file;
    ofstream offile;

    string outputFile;
    if(opt)
        outputFile = "opt.trie";
    else
        outputFile = "heu.trie";

    file.open(inputFile, ios::in);
    offile.open(outputFile);
    read(file);
    for(auto y: mapa){
        if(opt){
            trieOpt tri(y.first, y.second.size(), y.
                second[0].size(), y.second);
            tri.write(offile, outputFile);
        }else{
            trieHeu tri(y.first, y.second.size(), y.
                second[0].size(), y.second);
            tri.write(offile, outputFile);
        }
    }
    cout << "\nCompilacion terminada\n";
    file.close();
    offile.close();
}
```

main

```
Compiler compiler;
Parser parser;

int main()
{
    srand (time(NULL));
    do{
        flag = 1;
        solve();
    }while(flag);
    cout << "\n\nPrograma finalizado exitosamente\n\
        n";
    return 0;
}

void solve(){
    int opt1 = menu1();
    if(opt1 == 6){
        flag = 0;
        return;
    }

    switch (opt1){
        case 1:{    //Compilacion heuristica
            t0 = clock();
            compiler.compile("rules.txt", false);
            parser.parse(false);
            t1 = clock();
            tiempo = timeCalculate(t0, t1);
            cout << "\nExecution Time: " << tiempo
                << "\n";
            break;
        }
        case 2:{    //Compilacion optima
            t0 = clock();
            compiler.compile("rules.txt", true);
            parser.parse(true);
            t1 = clock();
            tiempo = timeCalculate(t0, t1);
            cout << "\nExecution Time: " << tiempo
                << "\n";
            break;
        }
        case 3:{    //Ejecucion de consultas
            t0 = clock();
            parser.queryExecuter("queries.txt");
            t1 = clock();
            tiempo = timeCalculate(t0, t1);
            cout << "\nExecution Time: " << tiempo
                << "\n";
            break;
        }
        case 4:{    //Mostrar reglas en memoria
            parser.showRules();
            break;
        }
        case 5:{    //Elimina regla
            cout << "Inserte regla a eliminar: ";
            string r;
            cin >> r;
            parser.eraseRule(r);
            break;
        }
    }
}

int menu1(){
    int opt = 0, optMax = 6;
    do{
        cout << "\n\n--------------------Proyecto de
             Analisis y Disenho de Algoritmos
             --------------------\n";
```

```
        cout << "1. Compilacion heuristica.\n";
        cout << "2. Compilacion optima.\n";
        cout << "3. Ejecutar consultas.\n";
        cout << "4. Mostrar reglas en memoria.\n";
        cout << "5. Borrar regla.\n";
        cout << "6. Salir del programa.\n";
        cout << "Ingrese una opcion: ";
        cin >> opt;
    }while(!isdigit(opt) && (opt < 1 || opt > optMax
        ) );
    return opt;
}
```

En el main instanciamos las clases Compiler y Parser de manera global.

Ejemplo:

Input:

$$rule\ aaa$$
$$rule\ bab$$
$$rule\ cab$$
$$rule\ cbb$$
$$rule\ dcb$$
$$rule\ dcc$$

Cadenas: (aaa, bab, cab, cbb, dcb, dcc)

SPtrie Generado:

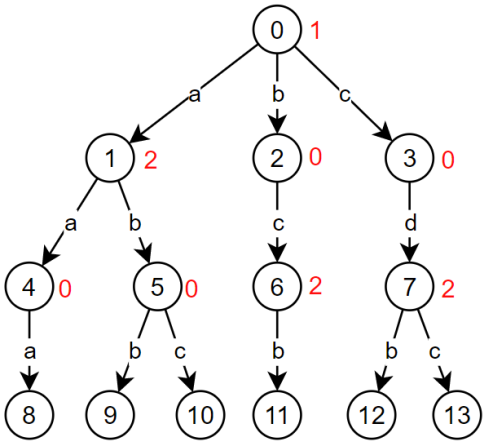

Fig 9.1: Visualización gráfica del Sptrie generado después de la compilación

Los índices de color rojo corresponden al vector posición.

El límite corresponde al valor del primer nodo hoja, para este caso sería 8.

Query: rule cbX

Recorrido DFS:

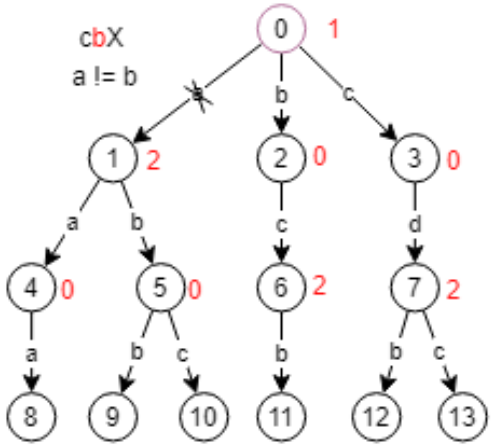

Fig 9.2: DFS sobre el trie generado

El algoritmo recorre el árbol en forma de DFS y verifica que el caracter de la consulta en la posición del nodo actual sea el caracter en la arista de su vecino. Si son equivalentes, entonces continúa su recorrido, caso contrario, termina y no hace nada. En este caso al nodo 0 le corresponde la posición uno y, como el primer vecino no tiene el caracter igual al de la posición 1 de la query(**b**), desiste de seguir la búsqueda en esa rama.

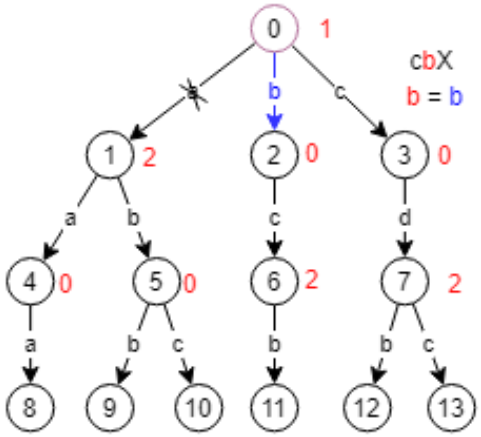

Fig 9.3: DFS sobre el trie generado

Como el caracter asignado a ese nodo en la lista de adyacencia es igual al de la query, la búsqueda por esa rama puede continuar.

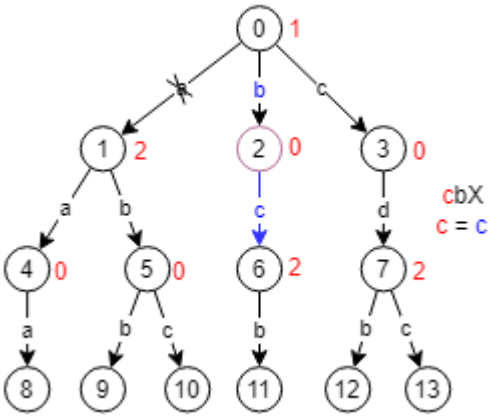

Fig 9.4: DFS sobre el trie generado

La búsqueda continúa ya que se cumplen las condiciones, que los caracteres sean iguales y que el siguiente nodo no sea mayor al límite.

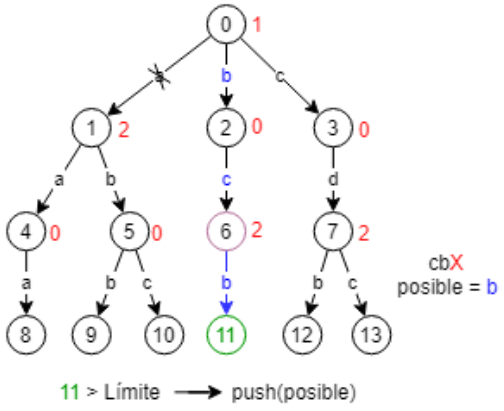

Fig 9.5: DFS sobre el trie generado

En esta instancia se busca el caracter en la posición 2 de la entrada y como encuentra X, añade el caracter que tiene guardado ese nodo como una posibilidad de respuesta (lo manda como parámetro en el DFS). Luego continúa su recorrido hasta llegar a una hoja.

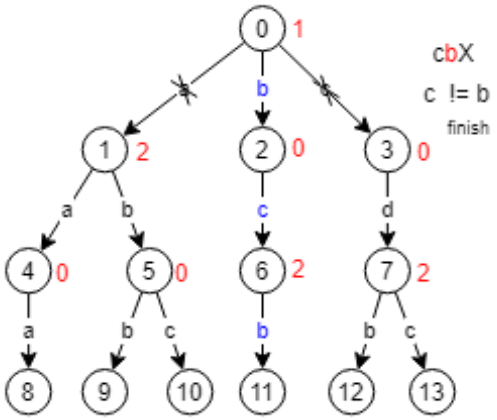

Fig 9.6: DFS sobre el trie generado

Llegar a la hoja significa que la cadena hizo matching, entonces agrega el caracter al vector resultado. Por último, recorre el tercer vecino del nodo inicial, pero, como no tiene una c en la posición 1, termina el recorrido.

Salida: X = b

10. Pregunta 10: Análisis experimental

Realize un análisis experimental entre las dos opciones de compilación. Deberá verse la diferencia en memoria entre una u otra opción de compilación. Su análisis experimental deberá incluir casos de prueba aleatorios e ir incrementando los tamaños de las muestras para notar una diferencia asintótica.

Generador de datos

1) El generador de datos recibe las entradas n y m.
2) Para crear datos válidos, el programa llena aleatoriamente las cadenas columna por columna. Además, debe asegurarse que en una columna los caracteres distintos estén en filas juntas, es decir, por bloques.
3) Para esto, genera números cuya suma sea n, máximo 26 números, y se le asigna una letra distinta a cada número. De esta forma, llenamos primero la columna 0, luego la 1, y así sucesivamente.
4) Las cadenas generadas se guardan en un archivo de texto con el formato ”Regla” + ”(” + ”caracteres separados por comas” + ”)”.

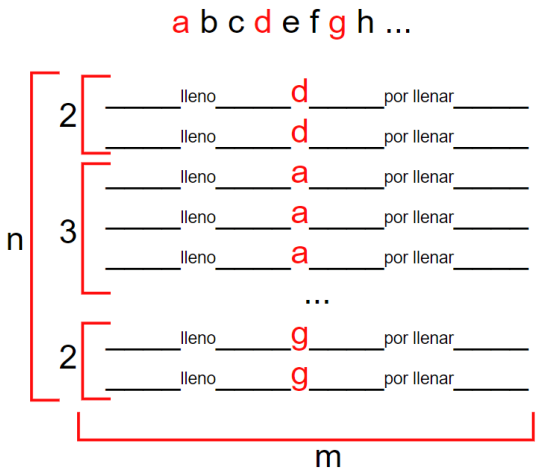

Fig 10.1: Ejemplo de generación aleatoria de reglas

Experimentación

1) Generamos cadenas aleatorias para n y m desde 50 a 200.
2) Aplicamos la compilación heurística y óptima del mini-compilador.
3) Anotamos los tamaños de los archivos heu.trie y opt.trie generados en KB.
4) En la comparación de los tamaños encontrados, observamos que el algoritmo heurístico, si bien es más rápido en tiempo, genera un archivo más pesado que el algoritmo óptimo. Esta diferencia se da por la cantidad de nodos creados por cada algoritmo.

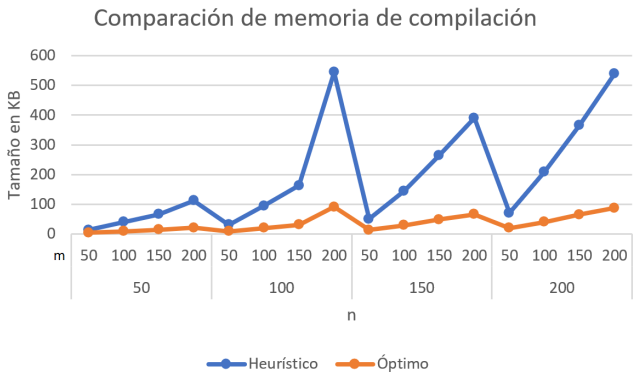

Fig 10.2: Comparación de memoria utilizada por los algoritmos

Repositorio en Github: https://github.com/jneirar/ProyectoADA.git