Monografía - Proyecto Final ADA

Claudia Noche, Mauricio Bernuy, Gonzalo Alfaro

Profesor: Juan Gutierrez

Introducción

En este texto, se presenta nuestra implementación y respuesta de la primera entrega del proyecto del curso de ADA. Nuestra implementación se puede encontrar en: https://github.com/Mauricio-Bernuy/ADA_FINAL_PROJECT

Para esta entrega, se trabajó con los *Tries*, los cuales son un tipo de estructura de datos de tipo árbol. Como cualquier árbol, este presenta nodos o *vértices*, así como aristas que los unen.

Un Trie, bajo este contexto, presenta la característica de ser parejo, es decir, es un tipo de árbol donde todas sus hojas están a la misma profundidad. Para poder describir mejor un Trie, se debe comprender dos funciones clave f y o, las cuales se describirán brevemente.

La función f asigna para cada arista del Trie, un carcter del alfabeto $\sum$. Se busca asegurar que no hayan aristas hermanas, que salgan del mismo nodo, con el mismo carácter.

La función o asigna un orden a los hijos de un determinado vértice. Este orden puede estar definido por el orden alfabético de los caracteres.

Posterior a entender estos conceptos, se tiene que entender *S-ptrie* para poder explicar el siguiente problema. Un S-ptrie es un Trie que se forma en base a S cadenas y mantiene un valor p de permutaciones en las que las cadenas pueden ser leídas. En la siguiente figura, se muestra un ejemplo de un *S-ptrie*.

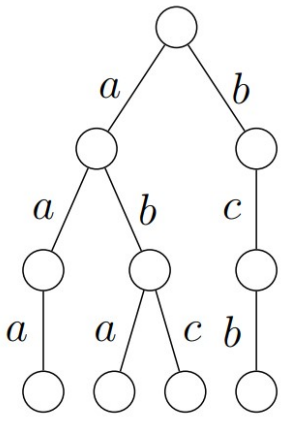


Fig. 1. Ejemplo de un S-ptrie.

S-Ptrie generalizado. Con el fin de poder implementar dicho algoritmo utilizando *programación dinámica*, se hicieron unos cambios a la estructura del trie previamente explicado. Se le aumenta una función que permite poner números a los nodos, los cuales representan los índices en las cadenas de los caracteres. Se tiene como ejemplo la siguiente figura, que utiliza las cadenas S = *(aaa, baa, bac, cbb)*.

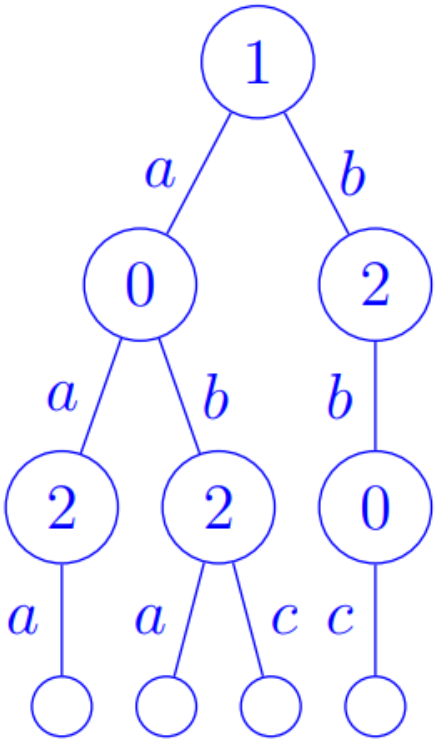


Fig. 2. Ejemplo de un S-Ptrie generalizado.

El problema del **Min-Trie-Generalizado** es similar al problema que se resolverá en la siguiente sección, y se basa en obtener, en base a una serie de cadenas, un S-Ptrie-generalizado óptimo, es decir, con el menor número de aristas posibles.

Pregunta 1 (Heurística Voraz)

Problema Min-Trie. Se recibe una secuencia S de cadenas y se debe devolver un *S-ptrie* con el menor numero de aristas posible, así como la suma de las mismas. En esta sección, buscaremos solucionar este problema con un algoritmo de complejidad en $O(nm+mlgm)$.

Implementación. Para solucionar el problema del Min-Trie, lo primero que se tuvo que implementar fue un Trie sim-

ple. Sobre dicha estructura, se corren una serie de procesos que generan un trie utilizando una heurística voraz. El pseudocódigo para el algoritmo principal es el siguiente:

```
Min_Trie_Heuristic(S,m,n):
    let max_cnt[m] = (0,'')
    let accum_cnt[|Σ|] = 0

    for col = 0 to m:
        let col_instances[|Σ|] = 0
        for row = 0 to n:
            char = S[row][col]
            col_instances[char]++
        largest_char = max(col_instances)
        largest_val = col_instances[largest_char]
        max_cnt[col] = (largest_val, largest_char)
        accum_cnt[largest_char] += largest_val

    let p_insert[m] = {0,1,...,m-1}
    quick_sort(p_insert, using=comparator{max_cnt,accum_cnt})
    let p_read[m] = empty
    for index=0 to m:
        p_read[p_insert[index]] = index

    return SPTRIE[S,p_insert,p_read,m,n]
```

Esta implementación recibe una lista N de cadenas S, con longitud m, y devuelve una estructura SPTRIE, la cual está conformada por la inserción de n cadenas de tamaño m contenidas en S, insertadas en el orden definido por la lista de índices p_ins, y leídas de vuelta por los índices de p_rd. La construcción de esta estructura devolvería también el numero de aristas del trie minimo.
Se puede observar de manera como el running time de este algoritmo es O(nm + mlgm), nm dado por el for anidado y mlgm por el uso de quick_sort para reordenar los índices de inserción. Los resultados devueltos a través de SPTRIE logran armar un S-PTrie de manera directa, con un running time de O(nm) que se considera dentro de la complejidad original. El costo espacial de O(nm|Σ|) estaría ligado al almacenamiento requerido por el S-PTrie construido.

```
comparator(i,j){max_cnt, acc_cnt}:
    if max_cnt[i].counter == max_cnt[j].counter:
        return acc_cnt[max_cnt[i].character] > acc_cnt[max_cnt[j].character]
    else:
        return max_cnt[i].counter > max_cnt[j].counter
```

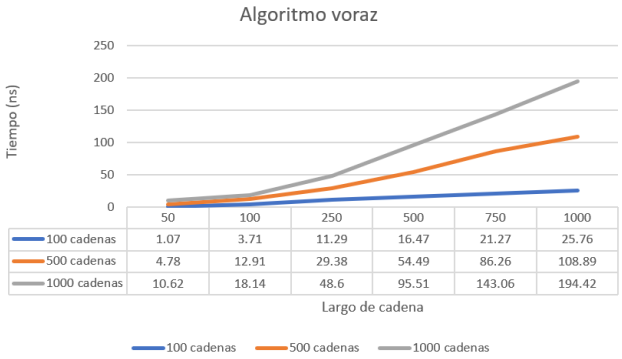
Se utiliza un comparador especializado para el algoritmo de quick_sort, que recibe i,j, y devuelve un valor booleano en base a la lógica definida, evaluada sobre el arreglo de contador por caracter max_cnt y el arreglo de conteo acumulado por carácter acc_cnt.
Estructura de un Node de un Trie. Se basa en la estructura de un *map* que almacena el carácter y el Nodo actual. La función de *add* busca añadir no solo un nuevo child con el carácter nuevo, sino también añade el carácter al final de la cadena que almacena el nodo actualmente.

```
class Trie
{
private:
  Node head;
public:
  Trie() : head() {}

  string find(const string& word)
  {
    Node* node = &head;
    for (auto&& ch : word) {
      if (node->child.find(ch) == node->child.end()) return "";
      node = &node->child[ch];
    }
    return node->value;
  }

  void insert(const string& word)
  {
    Node* node = &head;
    for (auto&& ch : word) {
      if (node->child.find(ch) == node->child.end()) node->add(ch);
      node = &node->child[ch];
    }
    node->is_leaf = true;
  }
};
```

La implementación del Trie se ve como se muestra en la figura previa. Se tiene un nodo principal, el cual es la cabeza o *root*, y se tienen dos funciones auxiliares, *find* y *insert*.

La función de *find(word)* va a buscar una cadena que se le pase. Hace esto iterando a través de los caracteres de la cadena provista. Si encuentra el carácter deseado, continua iterando hasta que encuentre que el último carácter es un nodo hoja, significando que encontró la palabra, o si no encuentra un carácter entre los hijos de un nodo, rompe el loop y finaliza.

La función *add(word)* utiliza la función *find* para ver si es que en el Trie se encuentra parte de la palabra que se desea insertar. Una vez encuentra un nodo que no tiene un hijo con un carácter deseado, utiliza la función *add* para añadir el carácter a un *child* nuevo.



	50	100	250	500	750	1000
100 cadenas	1.07	3.71	11.29	16.47	21.27	25.76
500 cadenas	4.78	12.91	29.38	54.49	86.26	108.89
1000 cadenas	10.62	18.14	48.6	95.51	143.06	194.42

Fig. 3. Ejemplo de un S-Ptrie generalizado.

Pregunta 2 (Recurrencia)

Algoritmo Óptimo Min-Trie-Gen.
Se tiene un conjunto K(ij) que representa índices en las cadenas donde éstas comparten el mismo carácter. Los valores de i y j representan el rango de cadenas que se ha revisado (desde 0 hasta n). Se arma un conjunto de cadenas removiendo las posiciones de caracteres repetidos, $\overline{S}$. El primer lema de OPT(i,j) sale de esto; y es lo siguiente:

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

Este lema se basa en la idea de que siempre hay un Trie óptimo que pone las posiciones K*(i,j)* al inicio; es decir, pone los caracteres más repetidos más arriba en el Trie. Entonces queda calcular $\overline{OPT}(i,j)$, para lo cual requerimos un conjunto R(ij), donde no todas las cadenas tienen el mismo valor de carácter, es decir, el complemento del conjunto K. Para cada r ∈ R(ij) definimos C(ijr), como el conjunto que representan el conjunto de pares de índices que definen tales subsecuencias.
Definiendo $\overline{OPT}$(i,j):

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

Fig. 4. Lema 2.2: Definición de $\overline{OPT}$(i,j).

Pregunta 3 (Recursivo)

En base a esta recurrencia hallada, se implementó un algoritmo para hallar esta cantidad mínima de aristas, además de construir un Min-Trie-Gen en el proceso, inicialmente con un costo computacional exponencial.

Implementación. El pseudocódigo para el algoritmo principal es el siguiente: El algoritmo requiere unas funciones auxiliares adicionales, las cuales se presentan como pseudocódigo:

recibe: dos índices del conjunto de cadenas S.
devuelve: un arreglo de valores booleanos de tamaño m, indicando si dicha columna tiene todos sus caracteres iguales para este rango de índices.

```
function K_GEN(i,j):
    check[m] = { true }
    last_S = S[i-1]
    for x = i to j-1:
        for c = 0 to m-1:
            if (S[x][c] != last_S[c]):
                check[c] = false

    return check
```

Complexity = O(nm)

recibe: dos índices del conjunto de cadenas S.
devuelve: una lista de índices de columnas con todos sus caracteres iguales para este rango de índices.

```
function K(i,j):
    kgen = K_GEN(i,j)
    k_list = empty_list
    cnt = 0
    for k in kgen:
        if (k == true):
            k_list.push(cnt)
        cnt = cnt + 1

    return k_list
```

Complexity = O(nm)

recibe: dos índices del conjunto de cadenas S.
devuelve: una lista de índices de columnas con todos sus caracteres no iguales para este rango de índices.

```
function R(i,j):
    kgen = K_GEN(i,j)
    r_list = empty_list
    cnt = 0
    for k in kgen:
        if (k == false):
            r_list.push(cnt)
        cnt = cnt + 1

    return r_list
```

Complexity = O(nm)

recibe: dos índices del conjunto de cadenas S y un valor r R(i,j) para estos índices.
devuelve: una lista de pares indicando las subsecuencias contiguas para este rango de índices y la columna r.

```
function C(i,j,r):
    c_list = empty_list
    beginpos = endpos = i

    while endpos <= j and beginpos <= j:
        if S[beginpos-1][r] == S[endpos-1][r]:
            endpos = endpos + 1
            if endpos > j:
                c_list.push( (beginpos, endpos-1) )
        else:
            c_list.push( (beginpos, endpos-1)
            beginpos = endpos

    return c_list
```

Complexity = O(n)

almacena: dos indices, i y j, el valor del nodo target_r, y una lista de elementos trie_node de sus hijos.

```
structure trie_node { i, j, target_r, childs}
```

Con estas funciones y estructuras auxiliares, ahora podemos definir el algoritmo recursivo principal, en pseudocódigo:

recibe: dos índices del conjunto de cadenas S.
devuelve: un par de elementos: número de aristas parcial y una lista de nodos del trie parcial.

```
function OPT_line(i,j):
    if (i == j):
        return ( 0, empty_list )

    target_min = +inf
    target_trie_list = empty_list
    target_r = 0

    for r in R(i,j):
        accumulated_sum = 0
        accumulated_trie_list = empty_list

        for (i',j') in C(i,j,r):
            (edges', trie_list') = OPT_line(i',j')
            accumulated_sum = accumulated_sum + edges + |K(i',j')| - |K(i,j)|

            if (|K(i',j')| > 1):
                for k' in K(i',j'):
                    if (k' != r && k' not in K(i,j)):
                        tmp = trie_list'
                        trie_list' = empty_list
                        trie_list'.push( trie_node{ i', j', k', tmp} )


            accumulated_trie_list.push( trie_node{ i', j', r, trie_list'} )

        if (accumulated_sum < target_min):
            target_min = accumulated_sum
            target_trie_list = accumulated_trie_list
```

```
            target_r = r

    return (target_min, target_trie_list)
```

Complexity = Exponential

El ultimo paso sería el obtener el OPT completo en base a el OPT_line ya definido, en pseudocódigo:

recibe: dos índices del conjunto de cadenas S.
devuelve: un par de elementos: número de aristas final y una lista de nodos del trie final.

```
function OPT(i,j):
    (edges_line, trie_list_line) = OPT_line(i,j)

    for k in K(i,j):
        tmp = trie_list_line
        trie_list_line = empty_list
        trie_list_line.push( trie_node{ i, j, k, tmp } )

    return (edges_line + |K(i,j)|, trie_list_line)
```

Complexity = Exponential

El algoritmo sigue estrechamente el funcionamiento de la recurrencia propuesta, pero con unas modificaciones con el fin de obtener la estructura del Min-Trie-Gen. Esto se hace al mantener una lista de los nodos candidatos visitados en cada iteración del segundo for loop, utilizando la búsqueda del valor mínimo de aristas para también quedarnos con la permutación de aristas hijas mas óptima. Un caso extra se debe considerar también, en el cual existe mas de un K(i',j') en algún paso de la obtención de hijos, caso en el cual se debe realizar una inserción continua hasta que todos los indices k' se hayan insertado correctamente.
Se puede observar la naturaleza exponencial que presenta este algoritmo, dadas las llamadas recursivas anidadas dentro de dos for loops. Además, cada llamada a las funciones K y R tambien pueden implicar un costo elevado debido a su frecuencia y repetitividad. La ejecución del algoritmo es bastante costosa en término computacional, bastante más que la propuesta Heurística, pero esta logra obtener el Min-Trie-Gen más óptimo. Veremos ahora unas optimizaciones para mejorar el running time del algoritmo.

Pregunta 4 (Memoizado)

Se aplicó la técnica de memoización en las funciones de K_GEN y OPT_line con el fin de aliviar las llamadas repetitivas a los mismos índices. Se aplicó de la siguiente manera, mostrado en pseudocódigo:
recibe: dos índices del conjunto de cadenas S. *devuelve*: un arreglo de valores booleanos de tamaño m, indicando si dicha columna tiene todos sus caracteres iguales para este rango de índices. De haber sido llamado previamente, retorna el valor previamente memoizado.

```
function K_GEN_MEMO(i,j):
    if K_MEM[i][j] not empty:
        return K_MEM[i][j]

    check[m] = { true }
    last_S = S[i-1]
    for x = i to j-1:
        for c = 0 to m-1:
            if (S[x][c] != last_S[c]):
                check[c] = false

    K_MEM[i][j] = check
    return check
```

Complexity = O(nm) (O(1) en HIT)

recibe: dos índices del conjunto de cadenas S.
devuelve: un par de elementos: número de aristas parcial y una lista de nodos del trie parcial. De haber sido llamado previamente, retorna el valor previamente memoizado.

```
function OPT_line(i,j):
    if OPT_LINE_M[i][j] not empty:
        return OPT_LINE_M[i][j]
    }

    if (i == j):
        return ( 0, empty_list )

    target_min = +inf
    target_trie_list = empty_list
    target_r = 0

    for r in R(i,j):
        accumulated_sum = 0
        accumulated_trie_list = empty_list

        for (i',j') in C(i,j,r):
            (edges', trie_list') = OPT_line(i',j')
            accumulated_sum = accumulated_sum + edges + |K(i',j')| - |K(i,j)|

            if (|K(i',j')| > 1):
                for k' in K(i',j'):
                    if (k' != r && k' not in K(i,j)):
                        tmp = trie_list'
                        trie_list' = empty_list
                        trie_list'.push( trie_node{ i', j', k', tmp} )


            accumulated_trie_list.push( trie_node{ i', j', r, trie_list'} )

        if (accumulated_sum < target_min):
            target_min = accumulated_sum
            target_trie_list = accumulated_trie_list
            target_r = r

    OPT_LINE_M[i][j] = (target_min, target_trie_list)
    return (target_min, target_trie_list)
```

Complexity = A Evaluar más adelante

Tambien se añade la siguiente rutina al comienzo de OPT (casos base):

```
    ...
    for f = 0 to j:
        OPT_LINE_M[f][f] = (0, empty_list);
        K_MEM[f][f] = {true,true, ..., m-1};
    ...
```

Pregunta 5 (Programación Dinámica)

En base a la pregunta anterior, solo basta con eliminar la recursividad aplicando una estrategia bottom-up, la cual se presenta de esta manera:
Se realizaron estas ultimas modificaciones a la estructura Memoizada (en el OPT final):

```
    ...

    lim = j;
    for j_ = i to j:
                i = lim
                jj = j_
                for ii = 1 to lim:
            OPT_line_MEMO(ii,jj);
            jj++;
                lim--;
```

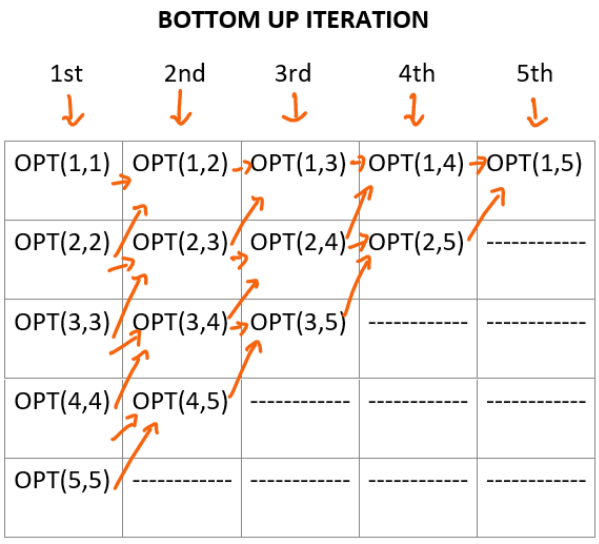
```
...
```

Bottom up OPT_line_MEMO

```
...

for j_ = i to j:
    for i_ = i to j_:
        K_GEN_MEMO(i_,j_)

    ...
```

Bottom up K_GEN_MEMO

Lo mas importante de observar es el orden de iteracion del for loop anidado de OPT_line_MEMO. Para poder aplicar bottom-up de manera óptima, es necesario seguir un orden de iteración bien específico. La siguiente gráfica lo puede explicar mejor:



Por pruebas experimentales, encontramos que cada llamada a OPT_line con indices i,j necesita llamar recursivamente como máximo a los OPT_line con indices desde i,i hasta j,j. Por ende, una reconstruccion optima bottom-up deberá haber evaluado este rango de índices previamente y haberlo guardado en memoria, con el fin de mejorar drásticamente el tiempo de ejecución y eliminando la recursividad.

Nuestro DP itera de esta manera, como se puede ver a continuación:

```
OPT_line_MEMO(1,1): {
OPT_line_MEMO(2,2): {
OPT_line_MEMO(3,3): {
OPT_line_MEMO(4,4): {
OPT_line_MEMO(5,5): {
OPT_line_MEMO(1,2): {
OPT_line_MEMO(2,3): {
OPT_line_MEMO(3,4): {
OPT_line_MEMO(4,5): {
OPT_line_MEMO(1,3): {
OPT_line_MEMO(2,4): {
OPT_line_MEMO(3,5): {
OPT_line_MEMO(1,4): {
OPT_line_MEMO(2,5): {
OPT_line_MEMO(1,5): {
OPT_DP runtime: 11.308ms
```

Entonces, evaluando esta nueva complejidad, podríamos verla ligada por dos for anidados, ejecutando la siguiente función:

$$\text{mín}_{r \in R(ij)}\{\sum_{(i',j') \in C(i,j,r)} \text{OPT'\_MEMO[i][j]} + |K(i'j')| - |K(ij)|\}$$

Entonces, podriamos decir que este procedimiento se encuentra ligado por O(n_2 * extra_costs), donde los costos extra vendrian del calculo de las K_GEN y los casos de multiple push al trie. El almacenamiento de la matriz también seria O(n_2), sumado al costo de O(n_2) del almacenamiento de K_GEN y O(nm|Σ|) por el almacenamiento del trie. Este análisis resultará válido tanto para DP y para la versión memoizada.

Pregunta 6 (Compilación heurística) & Pregunta 7 (Compilación óptima)

Una vez implementados los algoritmos, se implementó un mini complilador con miniProlog, donde se pueda crear Sptries con estos algoritmos. Debería poderse escoger el algoritmo al momento de compilación. Se nos pidió leer de un archivo de texto, y en base a este, crear Tries distintos y válidos.

Para esta lectura del archivo, sin importar qué algoritmo se desee utilizar, la misma función fue la que se encargó de *parsear* el archivo de entrada. Este tiene el formato de *R1 cadena*. El algoritmo acá lee linea por linea dicho archivo y revisa primero el nombre del trie. Si este ya lo ha reconocido, solo continua con la cadena, la cual es leida e insertada a un vector de strings del correspondiente trie. Si no reconoce el trie por nombre, crea un nuevo par, dándole el nombre recien leido, y se repite el proceso con la cadena.

```
void filler( string &filler, string line_to_read, int n){
    if (line_to_read[n] == ' '){
        int j = n+1;
        int len = line_to_read.length();
        for (j; j < len; j++){
            filler.push_back(line_to_read[j]);
```

```
        }
    }
}
pair<string,string> parseline(string line){
    stringstream line_s(line);
    string temp;

    getline(line_s, temp, ' ');
    string rule = temp;
    getline(line_s, temp);
    string lin = temp;

    return make_pair(rule,lin);

}

vector<pair<string,vector<string>>> readFile(string path){
    vector<pair<string,vector<string>>> output;
    ifstream file;
    file.open(path);
    while(!file.eof() && file.is_open()){
        vector<string> rulez;
        int cnt = 0;
        string line;
        while(getline(file,line)){
            string trie;
            string cadena;
            int cnt_cad = 0;
            //while(line[cnt_cad] != '('){ // ' '
            while(line[cnt_cad] != ' '){
                trie.push_back(line[cnt_cad]);
                cnt_cad++;
            }

            filler(cadena, line, cnt_cad);

            if (output.empty()){
                pair<string,vector<string>> neu;
                neu.first = trie;
                neu.second.push_back(cadena);
                output.push_back(neu);
            } else if (output[cnt].first == trie){
                output[cnt].second.push_back(cadena);
            }else if (output[cnt].first != trie){
                cnt++;
                pair<string,vector<string>> neu;
                neu.first = trie;
                neu.second.push_back(cadena);
                output.push_back(neu);
            }
        }
    }
    file.close();
    return output;
}
```

Lector de un archivo y parser de este mismo. Código en: read_prolog.cpp

Con el fin de escoger el tipo de algoritmo con el cual se va a trabajar, se especifica en la línea de compilación. Se trabaja con una enumeración, a la cual se accede luego de leer el argumento en el cual se especifica qué algoritmo se quiere usar. Esta enumeración luego es utilizada en un switch, que crea el Trie en base al algoritmo elegido.

Aparte del algoritmo, en la línea de compilación se puede poner, opcionalmente, el archivo de input y el archivo al cual se quiere mandar como output. Si no se colocan, los nombres son puestos automáticamente por el sistema como "input" para input y "trie" para output. Este último va a ser el punto donde se van a crear nuestros tries, con una extensión .obj.

```
enum WORKMODE {HEU,REC,MEMO,DP};

//#define PRINTING

// COMPILER
// structure -> ./compile.x --WORKMODE i-input o-trie
int main(int argc, char** argv){
    WORKMODE WM;

    vector<string> arguments;
    for (int i = 1; i < argc; i++)
        arguments.push_back(argv[i]);

    bool setflag,setinfile,setoutfile;
    string infile,outfile;
    for (auto s : arguments){
        if (s[0] == '-'){
            if (s == "--heuristic"){
                WM = HEU;
                setflag = true;
            }
            else if (s == "--recursive"){
                WM = REC;
                setflag = true;
            }
            else if (s == "--memoized"){
                WM = MEMO;
                setflag = true;
            }
            else if (s == "--dynprog"){
                WM = DP;
                setflag = true;
            }
            else{
                cerr << "ERROR: invalid flag\n"<<
                "\tflags: --heuristic --recursive --memoized --dynprog";
                return -1;
            }
        }
        else if (s[0] == 'i' && s[1] == '-'){ //input
            s = s.substr(2,s.length());
            infile = s;
            setinfile = true;
        }
        else if (s[0] == 'o' && s[1] == '-'){ //output
            s = s.substr(2,s.length());
            outfile = s;
            setoutfile = true;
        }
        else{
            cerr << "ERROR: wrong argument\n";
            return -1;
        }
    }

    if (!setflag){
        cerr << "ERROR: missing flag\n" <<
        "\tflags: --heuristic --recursive --memoized --dynprog";
        return -1;
    }

    if (!setinfile) infile = "input";
    if (!setoutfile) outfile = "trie";


    auto in = readFile(infile + ".prolog");
    ofstream names;

    names.open(outfile + ".aux");
    print_CLEAR(outfile +".obj");

    for (auto prolog : in){
        names << prolog.first << ":";
        int edges;
        tuple<int, std::vector<targ>> T;
        switch(WM){
            case HEU:
                edges = build_and_output(Min_Trie_Heuristic
                                (prolog.second),outfile + ".obj");
            break;
            case REC:
                T = OPT_REC(1,prolog.second.size(),prolog.second);
                edges = get<0>(T);
                printres_GEN_file(get<1>(T), outfile + ".obj");
            break;
            case MEMO:
                T = OPT_MEMO(1,prolog.second.size(),prolog.second);
                edges = get<0>(T);
                printres_GEN_file(get<1>(T), outfile + ".obj");
            break;
            case DP:
                T = OPT_DP(1,prolog.second.size(),prolog.second);
                edges = get<0>(T);
                printres_GEN_file(get<1>(T), outfile + ".obj");
            break;
        }
        names << edges << "\n";
    }
    names.close();

    return 0;

}
```

Lector de argumentos de la linea de compilación. Código en:

trie_generator.cpp

Luego de leer el argumento, llama a la función del algoritmo requerido y escribe el Trie en un .obj en una dirección indicada. Aparte de manejar la generación de el trie correcto, creamos un archivo con extención .aux que tiene tanto el nombre de los tries con los que estamos trabajando como su número de aristas.

Pregunta 8 (Parser)

En esta sección del código leemos los archivos que hemos generado de tries, los parseamos y en base a nuestra codificación, podemos realizar queries en ellos. La siguiente función se encarga simplemente de leer el .obj que fue creado en el paso previo. Separa la información de tal forma que puede ser utilizada más adelante por el programa para realizar las queries.

```
vector<tuple<string, int, vector<targ_ *>>> trie_reader(string FILE){
    auto read = readres_ANY_file(FILE + ".obj");
    ifstream names;
    names.open(FILE + ".aux");
    vector<tuple<string, int, vector<targ_ *>>> result;
    int r = 0;
    for (auto x : read){
        string info;
        getline(names,info);
        stringstream splitter(info);
        getline(splitter, info, ':');
        string name = info;
        getline(splitter, info, ':');
        int edges = stoi(info);
        #ifdef PRINTING
        cout<<"Trie: " << name <<", edges: " << edges << "\n";
        printres_ANY_console(x);
        #endif
        result.push_back(make_tuple(name,edges,read[r]));
        r++;
    }

    names.close();
    return result;
}
```

Código en: read_and_exec_prolog.cpp.

La siguiente sección del código se va a encargar de dos cosas en particular. La primera, es en revisar si en la linea de compilación hay argumentos válidos de input y output. Acá es donde se validan y se ponen dentro del código si es que hubieran. No son los previos input y output mencionados; en este caso, el input se trata de el archivo codificado de los tries, y el output es el .out de respuestas.

```
// EXECUTER - QUERY PROCESSING
// structure -> ./execute.x i-trie o-response < queries.prolog
int main(int argc, char** argv){

    vector<string> arguments;
    for (int i = 1; i < argc; i++)
        arguments.push_back(argv[i]);

    bool setinfile,setoutfile;
    string infile,outfile;
    for (auto s : arguments){
        if (s[0] == 'i' && s[1] == '-'){ //input
            s = s.substr(2,s.length());
            infile = s;
            setinfile = true;
        }
        else if (s[0] == 'o' && s[1] == '-'){ //output
            s = s.substr(2,s.length());
            outfile = s;
            setoutfile = true;
        }
        else{
            cerr << "ERROR: wrong argument\n";
            return -1;
        }
    }
    if (!setinfile) infile = "trie";
    if (!setoutfile) outfile = "response";

    auto loaded_tries = trie_reader(infile);
    ofstream output;
    output.open(outfile + ".out");
    output.close();

    string ln;
    while (getline(cin,ln)){
        auto query = parseline(ln);
        int position = query.second.find('X');
        #ifdef PRINTING
        cout << "rule: " <<  query.first << ", string: "<< query.second <<", Xpos: " << positio
        #endif

        vector<char> result;
        for (auto set : loaded_tries){
            auto rule = get<0>(set);
            if (rule == query.first){
                result = search_string(query.second,position,get<2>(set));
                break;
            }
        }
        bool comma = false;
        ofstream output;
        output.open(outfile + ".out", ios::app);
        if (result.empty()){
            cout << "X = {}";
            output << "X = {}";
        }
        else{
            for (auto c : result){
                if (!comma)
                    comma = true;
                else{
                    cout <<", ";
                    output <<", ";
                }
                cout << "X = " << c;
                output << "X = " << c;
            }
        }
        cout << "\n";
        output << "\n";
        output.close();
    }
}
```

De lo siguiente que se encarga esta sección del código es de ejecutar las queries. Va a leer los tries con la función previamente explicada y los va a guardar en una estructura fácil de manejar. De ahí, lee los inputs de la consola. Este input es parseado para reconocer el nombre del trie, y la cadena que le sigue, la cual tiene una X como incógnita.

Realizar la búsqueda requiere que a la función *search_string* se le pase la cadena donde se encuentra la X, la posición de este caractér, y el trie, en forma de vector. Esta función será explicada más adelante, pero lo que se debe saber es que devuelve un vector de caracteres, los cuales representan las opciones que X puede tomar. Lo que prosigue en el código es imprimir dichos resultados con la codificación requerida no solo en la consola, sino también en un archivo .out de resultados.

```
bool valid_path(string chars, vector<targ_*> data){
    vector <targ_*> * iter = &data;
    vector <targ_*> * next;
    while (true){
        bool found = false;
        if (iter->empty()){
            break;
        }
        for (auto t : *iter){
            auto r = t->targetr;
            if (t->c == chars[r]){
                next = &t->sons;
                found = true;
                break;
            }
        }
        if (!found){
            return false;
        }
```

```
        iter = next;

    }
    return true;
}

vector<char> search_string(string chars, int ind_X, vector<targ_*> data){
    vector<bool> found(chars.length(), false);
    vector<char> empty;
    if (data.empty()  || chars.empty())
        return empty;

    vector<char> result;

    vector <targ_*> * iter = &data;
    vector <targ_*> * next;
    bool finished = false;
    while (!finished){
        bool found = false;
        for (auto t : *iter){
            auto r = t->targetr;
            if (r == ind_X){
                found = true;
                for (auto t_ : *iter){
                    auto success = valid_path(chars,t_->sons);
                    if (success){
                        result.push_back(t_->c);
                    }
                }
                finished = true;
                break;
            }
            else if (t->c == chars[r]){
                next = &t->sons;
                found = true;
                break;
            }
        }
        if (!found){
            finished = true;
        }

        iter = next;
    }

    return result;
}
```

La función recorre iterativamente el Trie hasta encontrar la posición de X, la cual fue pasada como parámetro a la función. Si la encuentra, recorre los *sons* del nodo al que se llegó, y los caminos que lleguen a un nodo hoja son considerados válidos. Luego de encontrar un camino válido, se añade la arista con el carácter inicial al vector de resultados de X.

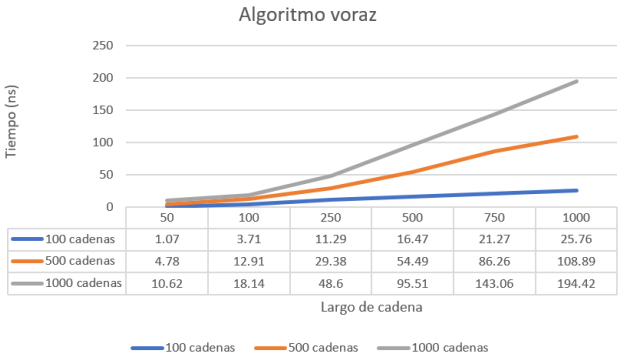
Pregunta 9 (minicompilador miniProlog)

Esta pregunta resume mucho de lo que ya ha sido explicado previamente. Ya se ha visto que el progrma recibe archivos y los procesa a objetos trie, y que en el momento de compilación se escoge qué algoritmo se debe usar.
Para hacer queries múltiples, se puede pasar un archivo por pipe al programa para que se corran las consultas.

El siguiente código muestra cómo correr efectivamente el programa.

```
# build

g++ -D -TIMINGS -g P6\trie_generator.cpp P1\SPTRIE.cpp
    P3\SPTRIEGEN.cpp P6\read_prolog.cpp -o compile.plg

g++ -g P6\read_and_exec_prolog.cpp P1\SPTRIE.cpp
    P3\SPTRIEGEN.cpp P6\read_prolog.cpp -o execute.plg

# execute

./compile.plg --WORKMODE i-input o-trie
              --heuristic --recursive --memoized --dynprog

./execute.plg i-trie o-response < queries.prolog

# execute on one line

compile.plg --dynprog i-input o-trie && execute.plg i-trie o-response < queries.prolog
```

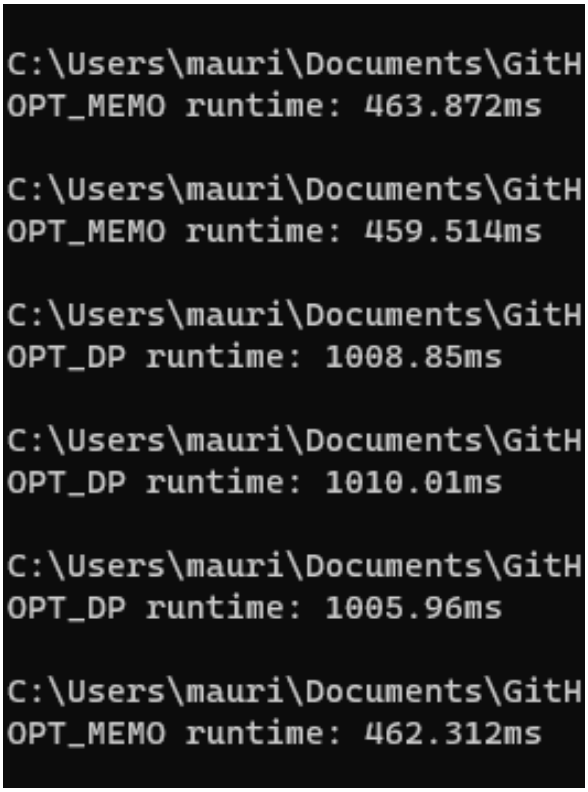
Pregunta 10 (Análisis experimental)



	50	100	250	500	750	1000
100 cadenas	1.07	3.71	11.29	16.47	21.27	25.76
500 cadenas	4.78	12.91	29.38	54.49	86.26	108.89
1000 cadenas	10.62	18.14	48.6	95.51	143.06	194.42

Fig. 5. Ejemplo de un S-Ptrie generalizado.

A diferencia de los otros dos algoritmos, el voraz es sumamente eficiente y puede manejar cadenas mucho mas grandes que los recursivos. Además, el gasto de espacio de memoria también es mucho menor. Las pruebas que realizamos lograron mostrar que las versiones memoizada y DP corren peor que la heuristica, y ademas que la diferencia en el tamaño de aristas suele ser muy pequeña y muchas veces nula.



Cabe resaltar tambien que nuestra implementacion de DP, debido a su elevado uso de memoria, resulta ser menos eficiente que la version directamente memoizada. Eso es de esperarse, pues en el analisis asintotico no se suele considerar los costos de acceso y escritura en memoria, lo cual ponen la version DP en una grave desventaja. Para esto, concluimos que la versión memoizada es superior.