Monografía Proyecto Final

Integrantes:
Diego Sebastián Ortiz Sánchez 201910248
Jorge Luis Vásquez del Águila 201310292
José Adrián Porres Brugué 201910265

Universidad de Ingeniería y Tecnología
Ciencia de la computación
Análisis y Diseño de Algoritmos - ADA - CS2102
Docente: Juan Gabriel Gutierrez Alva

5 de agosto de 2021

Índice general

1. Introducción **3**

1.1. Definición del problema 3

1.2. Planteamiento de solución 3

1.3. Glosario 3

2. Algoritmos implementados **4**

2.1. Pregunta 1 4

2.1.1. Implementación 4

2.1.2. Pseudocódigo 5

2.1.3. Complejidad 5

2.2. Pregunta 3 5

2.2.1. Implementación 5

2.2.2. Pseudocódigo 7

2.2.3. Complejidad 7

2.3. Pregunta 4 7

2.3.1. Implementación 7

2.3.2. Pseudocódigo 8

2.3.3. Complejidad 9

2.4. Pregunta 5 9

2.4.1. Implementación 9
2.4.2. Pseudocódigo 10
2.4.3. Complejidad 11
2.5. Pregunta 6 11
2.5.1. Implementación 11
2.6. Pregunta 7 11
2.6.1. Implementación 11
2.7. Pregunta 8 12
2.8. Pregunta 9 12
2.9. Pregunta 10 12

3. Conclusiones **14**

4. Anexos **15**

1. Introducción

1.1. Definición del problema

Los S-ptrie y S-ptrie generalizados son de mucha utilidad para implementar motores de base de datos, compiladores, entre otros. Por lo que se busca disminuir la cantidad de aristas, con la finalidad de que las búsquedas o queries sean mucho más rápidas y eficientes.

1.2. Planteamiento de solución

En el presente informe se plantean dos tipos de trie el S-ptrie en la pregunta 1 y el S-ptrie generalizado para las demás. La principal diferencia es que el S-ptrie tiene solo una posible permutación que viene dada por un vector p, mientras que el S-ptrie generalizado puede tener múltiples, porque cada camino desde la raíz hasta la hoja (excepto la hoja misma) forma una permutación desde 0, 1, ..., h-1.

Para resolver el problema del S-ptrie plantearemos una heurística voraz que puede resolver el problema rápidamente, aunque no asegura una solución óptima. Luego, para el S-ptrie generalizado utilizaremos algoritmos recursivos, memoizados y de programación dinámica que nos permitirán encontrar la solución óptima al problema, pero con una complejidad algorítmica superior. Finalmente, pondremos a prueba los algoritmos de Greedy y Programación Dinámica en una experimentación con distintas cantidades de datos que serán leídas mediante un parser, se mostrará una interfaz en consola que permitirá a los usuarios realizar la compilación heurística u óptima según necesiten.

1.3. Glosario

- n: número de cadenas.
- m: longitud de las cadenas.
- sigma: alfabeto que conforma las cadenas.

2. Algoritmos implementados

2.1. Pregunta 1

2.1.1. Implementación

Para esta pregunta desarrollamos un algoritmo voraz que, basándonos en la siguiente heurística:
Se analiza comparando las letras de cada una de las n cadenas desde la posición i = (0, 1, ..., m-1) y se obtienen m pares con la posición i (first) y la máxima frecuencia de letra (second) para dicha posición. Luego, se ordenan todos estos en base al segundo elemento del par y de forma decreciente.
Una vez que ya se encuentran ordenados nuestro maxPerLevel.first es el vector p que se le tiene que pasar al S-ptrie para que a partir de este se cambien de orden las cadenas para juntar los caracteres repetidos e intentado eliminar la mayor cantidad de aristas.

2.1.2. Pseudocódigo

```
umap := {};
maxPerLevel := [()];
for i:=1 to m do
    max := -1;
    for each palabra in S do
        umap(palabra(i), i)++;
        if umap(palabra(i), i) >max) then
            max = umap(palabra(i), i);
        end
    end
    maxPerLevel.push_back((i, max));
end
sort(maxPerLevel, bySecond);
p = maxPerLevel.first;
SPTrieNode *root(p);
for each palabra in S do
    root→insert(palabra);
end
return (root, aristas);
```

2.1.3. Complejidad

Note que, la complejidad para:

- Llenar el vector maxPerLevel es $\mathcal{O}(nm)$ porque son n cadenas de longitud m y te quedas con la posición que más se repite junto con su frecuencia.
- Ordenar el vector maxPerLevel es $\mathcal{O}(mlgm)$ porque tiene m elementos.
- Llenar el S-ptrie es $\mathcal{O}(nm)$ porque son n cadenas de longitud m.

Por lo tanto, la complejidad del algoritmo sería $\mathcal{O}(nm + mlgm)$.

2.2. Pregunta 3

2.2.1. Implementación

Para las preguntas siguientes es importante definir K_{ij}, R_{ij}, $RsinK_{ij}$ y C_{ijr}.

- K_{ij}, devuelve un arreglo de posiciones donde se cumple que todos los caracteres desde la cadena i hasta la cadena j son iguales y su complejidad es $\mathcal{O}(nm)$.
- R_{ijk}, devuelve un arreglo de las posiciones que están en el Universo = [0, 1, ..., m-1] pero no en k (K_{ij}), es decir una diferencia y como ambos ya están ordenados su complejidad es solo $\mathcal{O}(m)$.
- $RsinK_{ij}$, devuelve el mismo arreglo que R_{ij} solo que sin la necesidad de calcular K_{ij} antes y su complejidad es $\mathcal{O}(nm)$.
- C_{ijr}, devuelve un arreglo de pares ordenados que es la forma en la que puedes agrupar los caracteres repetidos desde la cadena i hasta la cadena j, en la columna r y su complejidad es $\mathcal{O}(n)$.

Para la construcción del S-ptrie generalizado tenemos nuestra función build_trie la cual recibe 3 parámetros: el i, el j y el min_pos. El i y j son las cadenas que se van a leer y también tenemos a min_pos la cual es una matriz que almacena todas las posiciones mínimas para cada uno de los i j posibles. Esta matriz se llena a la hora de ejecutar alguno de los algoritmos (recursivo, memorizado o dinámico).

Primero traemos la posición óptima en donde se realiza la partición localizándola en la tabla min_pos y almacenando en una variable p, todo esto utilizando los valores de i y de j tomados de la entrada. Luego sacamos todos sus pares con C(i,j,p) para iterar en ellos y crear un nodo por cada par. Después, para cada par obtenido, traemos el R'. Todo esto con el fin de obtener la diferencia entre R(i,j) y R'(par.first, par.second) y tener todos los RDIF resultantes. Teniendo esto, iteramos en RDIF y para cada valor diferente al p obtenido al inicio de la función crearemos un nuevo nodo el cual lo unimos con el creado con los pares. Finalmente, luego de ya tener este subárbol, simplemente lo agregamos a las adyacentes del root para así terminar y tener el trie completo.

2.2.2. Pseudocódigo

OPTR(i, j, min_pos)

```
if i = j then
    min_pos := -1;
    return 0;
k := K_ij;
raux := R_ijk;
minimo := ∞;
for r in raux do
    c := C_ijr;
    suma := 0;
    for (a, b) in c do
        suma += OPTR(a, b) + |K(a, b)| - |k|;
    if suma <minimo then
        minimo := suma;
        min_pos(i, j) := r;
return minimo;
```

OPT(i, j)

```
return OPTR(i, j) + |K_ij|;
```

2.2.3. Complejidad

Este algoritmo es de complejidad exponencial y como no almacena el OPTR entonces realiza cálculos innecesario de información que ya tiene. En las siguientes preguntas realizaremos la versión memoizada y de programación dinámica para mejorar la complejidad del algoritmo y que pueda resolverlo en tiempo polinomial.

2.3. Pregunta 4

2.3.1. Implementación

La versión memoizada es bastante similar a la recursiva, solo que ahora utilizamos dos mapas (mapa de int con mapa int int), preferimos utilizar mapas para no gastar memoria innecesaria porque solo nos interesa la matriz triangular superior. Primero, llamamos a la función fillMap que nos va a servir para llenar el umapOPT con 0s en la diagonal (casos base), -1s en otras casillas (casos que

faltan calcular) y al mismo tiempo el umapK que calcula el K en $\mathcal{O}(nm)$ para cada i, j $\mathcal{O}(n^2)$ válido (triangular superior) que como el K se calcula dentro de ese doble for quedaría como $\mathcal{O}(n^3m)$, pero para cumplir con la complejidad espacial no podemos guardar los elementos de K así que decidimos solo almacenar su cardinalidad. Luego, como umapK ya ha sido precalculado podemos acceder a este en $\mathcal{O}(1)$ y si umapOPT es diferente de -1 quiere decir que ya ha sido calculado anteriormente, caso contrario va a llamar recursivamente a memoizado y haciendo una suma acumulativa en suma.

2.3.2. Pseudocódigo

Memoizado(i, j, umapOPT, umapK, min_pos)

```
if i = j then
    min_pos(i, j) := -1;
    return 0;
k := umapK(i, j);
raux := RsinK_ij;
minimo := ∞;
for r in raux do
    c := C_ijr;
    suma := 0;
    for (a, b) in c do
        suma += |K(a, b)| - k;
        if umapOPT(a, b) ≠ -1 then
            return umapOPT(a, b);
        else
            suma += Memoizado(a, b, umapOPT, umapK, min_pos) +
             umapK(a, b) - k;
    if suma <minimo then
        min_pos(i, j) := r;
        minimo := suma;
```

LlamarMemoizado(i, j)

```
umapOPT = {{}};
umapK = {{}};
fillMap(umapOPT, umapK); // O(n^3 m)
return Memoizado(i, j, umapOPT, umapK) + |umapK_ij|;
```

2.3.3. Complejidad

- El fillMap es $\mathcal{O}(n^3m)$.
- En algoritmos memoizados el llamado recursivo se considera como $\mathcal{O}(1)$ porque se considera que en sí tiene n^2 estados.
- $RsinK_{ij}$ es $\mathcal{O}(nm)$.
- El for de raux se ejecuta como máximo m iteraciones $\mathcal{O}(m)$, la complejidad de C_{ijr} es $\mathcal{O}(n)$ y el for sobre c también es $\mathcal{O}(n)$. Note que los $\mathcal{O}(n)$ están en paralelo, así que se pueden sumar $\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$ y luego este se ejecuta m veces, teniendo así un $\mathcal{O}(nm)$.
- Finalmente, se tiene que la complejidad de un estado es $\mathcal{O}(nm) + \mathcal{O}(nm) = \mathcal{O}(nm)$, pero como son n^2 estados, entonces la complejidad final sería de $\mathcal{O}(n^3m)$ (sin considerar aún la del build trie).

2.4. Pregunta 5

2.4.1. Implementación

La idea es bastante similar a la memoizada utilizando un fillmap para umapOPT y umapK, solo que ahora empleamos un nuevo mapa que es umapAgrupa que como su nombre indica nos va a servir para agrupar caracteres repetidos y para saber donde particionar.

2.4.2. Pseudocódigo

ProgramacionDinamica(start, end)

```
–start;
–end;
fillMap(umapOPT, umapK); // O(n^3 m)
// O(nm)
for i:=0 to m-1 do
    umapAgrupa(i, end) := end;
    for j:=end-1 to 0 do
        umapAgrupa(i, j) := (S(j, i) == S(j+1, i)) ? umapAgrupa(i, j+1) :
         j;
// O(n^3 m)
for len:=1 to n do
    for i:=0 to n-len do
        j := i+len-1;
        if umapK(i+1, j+1) ≠ m then
            minimo = ∞;
            for k:=0 to m-1 do
                if umapAgrupa(k, i) <j then
                    suma := 0;
                    for l:=i to j l:=umapAgrupa(k, l)+1 do
                        if j <umapAgrupa(k, l) then
                            suma += umapK(l+1, j+1) + umapOPT(l, j);
                        else
                            suma += umapK(l+1, umapAgrupa(k, l)+1) +
                             umapOPT(l, umapAgrupa(k, l));
                        suma -= umapK(i+1, j+1);
                    if (suma <minimo) minimo = suma;
            umapOPT(i, j) = minimo;
        else
            umapOPT(i, j) = 0;
// O(m)
for k:=0 to m-1 do
    if umapAgrupa(k, start) = end then
        umapOPT(start, end)++;
return umapOPT(start, end);
```

2.4.3. Complejidad

- El fillMap es $\mathcal{O}(n^3m)$.
- Los cuatro for anidados es $\mathcal{O}(n^3m)$.
- Note que $\mathcal{O}(n^3m)$ + $\mathcal{O}(n^3m)$ = $\mathcal{O}(n^3m)$ (sin considerar aún la del build trie).

2.5. Pregunta 6

2.5.1. Implementación

- Ambos tipos de compilación leen dos archivos de texto entradaMiniProlog.txt y consultaMiniProlog.txt, donde el formato es regla cadena y la incógnita a encontrar es X.
- Primero se parsean los archivos de entrada y consulta.
- Después llama a pregunta1Greedy que recordemos que nos retorna un par de la root que forma el S-ptrie y la cantidad de aristas.
- Luego en un mapa con la regla como key y de valor la raíz que forma dicho S-ptrie.
- Finalmente se itera por sobre las consultas y se llama a executeQuery que es una función recursiva donde calculará el valor de X. Cabe recalcar que cuando se lee un X, entonces se deben probar todos los caminos de los hijos del nodo en el que se encuentra y el valor asignado temporalmente solo se añadirá al vector resultado cuando llegue a la hoja; esto para validar que cumpla con la query en su totalidad.

2.6. Pregunta 7

2.6.1. Implementación

- Primero se parsean los archivos de entrada y consulta.
- Después llama a ProgramacionDinamica que nos retorna la cantidad óptima de aristas y se le pasa min_pos por referencia para que lo llene.
- Luego a partir de este min_pos se va a crear el S-ptrie generalizado y guardamos su raíz como valor en el mapa con llave de la regla correspondiente.

- Finalmente se itera por sobre las consultas y se llama a executeQuery que es una función recursiva donde calculará el valor de X.

2.7. Pregunta 8

Para el parser estamos considerando que hay dos archivos uno que es el de entrada (regla cadena) y otro de query (regla cadena y dentro de la cadena se encuentra la incógnita X). Realizamos un split separándolo por espacio para tokenizarlo y llenamos en dos mapas (mapa de input y query) respectivamente las palabras o querys asociadas a cada regla.

2.8. Pregunta 9

Para esta pregunta realizamos una interfaz de usuario en consola, que está dentro de un bucle *while* donde si el usuario puede ingresar las siguientes opciones:

- 1. Greedy (compilación heurística).
- 2. Programación dinámica (compilación óptima).
- 3. Salir.

Una vez decidida la opción, se ejecutarán las consultas asignadas respectivamente para cada trie.

2.9. Pregunta 10

```
r1:
mXicwcidnjkqlrgozhnhzwqhr -> x
Tiempo de Consulta:9445
r2:
yxdgrnmjdukhjsdogykpscXscxkukzrcfikcydbygikelihgjx -> q
Tiempo de Consulta:13718
r3:
mxicwcgdhjkqlrgXzhnhzwqhalnpgzuuwikcqdnyngkbahxgisddrixgbvowhspoqzwfjvnqrcirduqjaswxhjjfjkmbghbtvjun -> o
Tiempo de Consulta:25210
```

Figura 2.1: Resultados de Greedy

```
r1:
mXicwcidnjkqlrgozhnhzwqhr -> x
Tiempo de Consulta:17240
r2:
yxdgrnmjdukhjsdogykpscXscxkukzrcfikcydbygikelihgjx -> q
Tiempo de Consulta:27281
r3:
mxicwcgdhjkqlrgXzhnhzwqhalnpgzuuwikcqdnyngkbahxgisddrixgbvowhspoqzwfjvnqrcirduqjaswxhjjfjkmbghbtvjun -> o
Tiempo de Consulta:52237
```

Figura 2.2: Resultados de Programación Dinámica

En el presente gráfico se puede apreciar como claramente la compilación heurística demora menos tiempo en ejecutarse porque su complejidad algorítmica es notablemente menor, pero a costa de no siempre encontrar la respuesta óptima. Por otro lado, la compilación óptima demora un poco más, pero es el precio a pagar por tener la respuesta óptima y a la larga si se quieren realizar múltiples queries sería lo ideal.

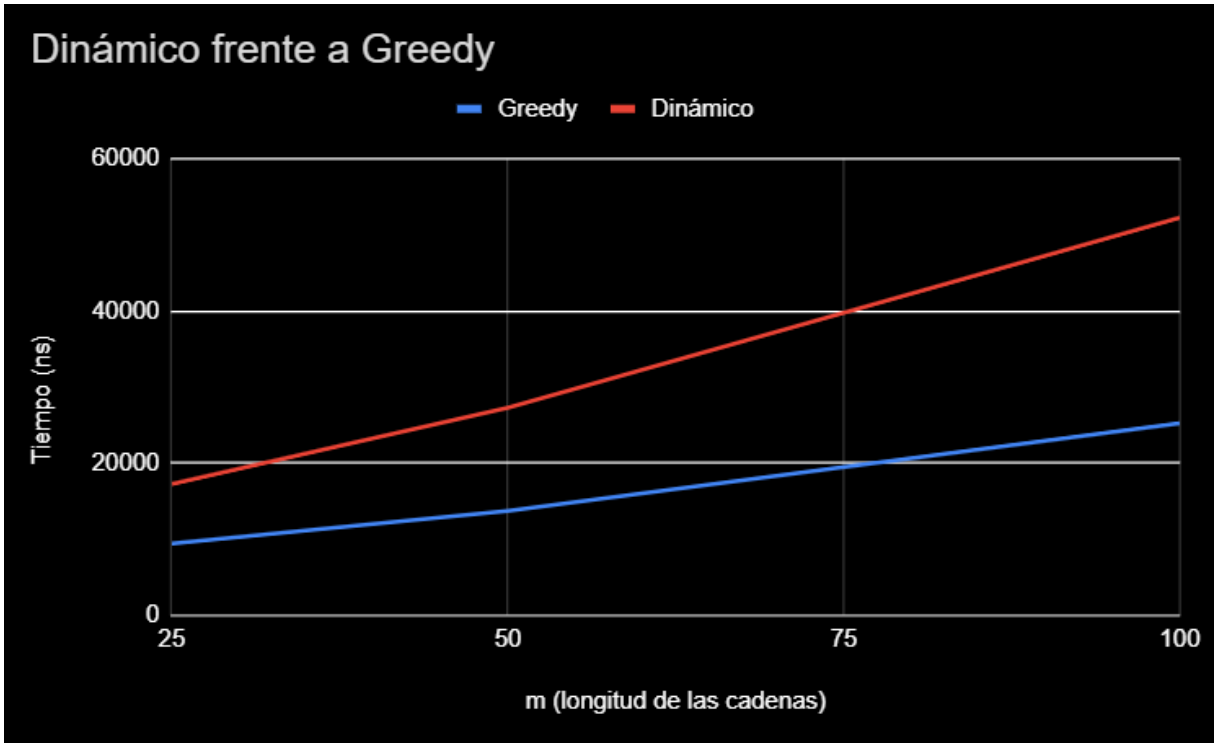

Figura 2.3: Gráfico comparativo de tiempos de ejecución

3. Conclusiones

a) Es importante conocer a profundidad un problema para reconocer que se puede resolver de distintas maneras y escoger el algoritmo que más convenga.

b) Para una gran cantidad de datos a veces es preferible tener una heurística o elección voraz para encontrar una respuesta rápida, aunque no necesariamente sea la óptima.

c) En otras circunstancias, como base de datos o compiladores se suele preferir la solución óptima para que en este caso particular los queries sean lo más eficientes posibles, ya que se realizarán en múltiples ocasiones.

d) Muchas veces la primera idea de algoritmo no será tan rápida con respecto a complejidad algorítmica, pero hay que ir iterando y aprendiendo de este para mejorarlo.

e) A pesar de que los algoritmos implementados de programación dinámica y memoización tengan la misma complejidad algorítmica, muchas veces se suele optar por el dinámico porque no está limitado por una cantidad máxima de llamados recursivos.

4. Anexos

https://github.com/DiegoOrtizS/proyecto-ada