DALGO Proyecto - Problema C

Daniel Zambrano H. - 201914912 , Angel Restrepo - 201914073

12 de diciembre de 2021

Algoritmo de solucion

Para este problema en particular, que se puede resumir en el titulo "hallar la Super-cadena mas corta.º "hallar la Super-cadena minimal"de una lista de cadenas m, para la cual se cumple que $(\forall s : m \mid : ||s|| = k)$, ademas de fijar un alfabeto Σ = ASCII. Sea $P(k)$ todas las cadnas de longitud k sobre Σ. Se trato de utilizar 2 tecnicas de desarrollo de algortimos:

- ◇ Algoritmos Avaros
- ◇ Programacion Dinamica

Sin llegar a ningun resultado favorable con respecto a las entradas de ejemplo provistas en el enunciado del proyecto. Al investigar un poco mas sobre el trasfondo del problema, se descubrio que el problema en cuestion pertenece a la categoria **NP-hard** y es equivalente al problema del conjunto de cobertura de la siguiente manera:

1. Sea S el conjunto de que contiene a las cadenas dadas:

$$S = \{s_1, s_2, ...s_n\}$$

2. El universo para el problema de cojunto de cobertura es S (Necsitamos encontrar una Super-cadena que tiene a todo s_i como subcadena)
3. $\sigma(x, y) : P(k) \times P(k) \to \{0, 1\} \approx$ funcion que determina si un par de cadenas se sobrelapan (1) o no (0)
4. $(\forall(s_i, s_j) \in (S) : \sigma(s_i, s_j) = 1)$ se debe:
   a) Construir una cadena r_{ijk} donde k es el maximo sobrelapamiento entre (s_i, s_j)
   b) Añadir el conjunto R, represetado por r_{ijk} a S $(S \cup R)$. R es el conjunto de todas las cadenas que son subcadenas de él. El costo de el subconjunto sera el tamaño de r_{ijk}

Teniendo en cuenta este contexto sobre el problema, es razonable tratar de desarrollar de aproximacion un algoritmo avaro para encontrar una solucion optima **aproximada** del problema con un factor de aproximacion de 2, es decir el alogirtmo no genera un resultado mayor que 2 veces el peor caso. Luego es factible que el algoritmo de aproximacion avaro que resuelve el problema del conjunto de cobertura pueda resolver tambien este problema por la equivalencia que se plantea.

Y en efecto este enfoque tambien resuelve el problema, aunque decidimos optar por el enfoque que usa el paradigma de "Travelling Salesman Problem"debido a que se nos hizo mas intuitivo de implementar debido a que usa las 2 tecnicas que mas usamos en el curso (DP y algoritmos avaros).

Primero se inicializan todas las variables necesarias para calcular la Super-cadena:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.Stack;

/**
 * @author Daniel Zambrano Huertas
 */
public class ProblemaC {
    private static int N;
    private static String[] m;
    private static int[][] dp;
    private static int[][] G;
    private static int[][] path;
```

Tambien es necesario definir la funcion que calculara si cualquier par de cadenas que se reciben (s, t) se sobrelapan entre si, es decir si t es prefijo de la cadena s y t es a su vez, sufijo de la cadena s. Y si efectivamente se sobrelapana, no indica en que indice en concreto lo hacen:

```
public static int check_suffix_prefix(String s, String t){
        for (int i=1 ; i<s.length() ; i++){
            if (t.startsWith(s.substring(i))){
                //return t.length() - s.length() + i; caso k diferentes
                   para cada m[i]
                return i;
            }
        }
        return t.length();
    }
```

Y po ultimo nos remitimos a la funcion principal , que es la que calcula cual seria la Super-cadena optima en base a los calculos de **check_suffix_prefix**. Cabe resaltar que como vamos a usar TSP para resolver el problema, es tambien necesario crear el grafo $G(V, E)$ que sera recorrido:

```
public static String minimal_super_string(String[] pM){
        N = pM.length;
        m = pM;
        G = new int[N][N];
        for (int l=0; l<N;l++){
            Arrays.fill(G[l],-1);
        }
        // calculate suffix and preffix of all the entries in m and
           build a graph that represents the overlapping of all entries
        for (int i=0; i<N; i++){
            for (int j=0;j<N;j++){
                if(G[i][j] == -1 && G[j][i] == -1 ) {
                    G[i][j] = check_suffix_prefix(m[i], m[j]);
                    G[j][i] = check_suffix_prefix(m[j], m[i]);
                }
            }
        }
```

En el caso del ejemplo se tendria:

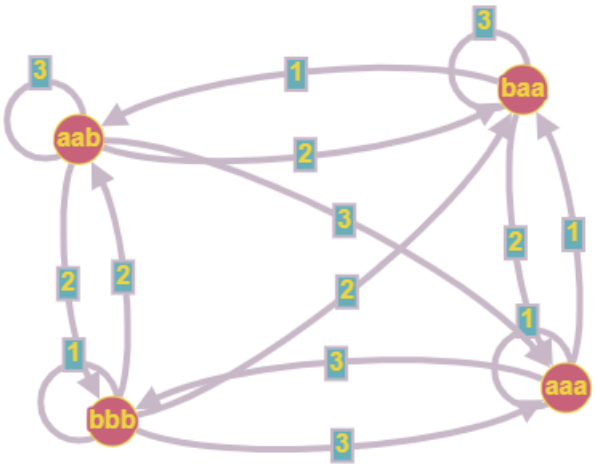

Figura 1: Grafo del ejemplo del proyecto

```
dp = new int[1 << N][N];
path = new int[1 << N][N];
int last = -1;
int min = Integer.MAX_VALUE;
int inf = Integer.MAX_VALUE;

for(int i = 1; i < (1 << N) ;  i++ ){
    Arrays.fill(dp[i], inf);
    for(int j=0; j < N; j++){
        if ((i&(1 << j))>0){
            int previous = i - (1<<j);
            if (previous == 0)
                dp[i][j] = m[j].length();
            else{
                for (int k=0; k<N;k++){
                    if(dp[previous][k]< inf && (dp[previous][k]
                       + G[k][j]< dp[i][j])){
                        dp[i][j] = dp[previous][k] + G[k][j];
                        path[i][j] = k;
                    }
                }
            }

        }
        if (i == (1<<N)-1 && dp[i][j] < min){
            min = dp[i][j];
            last = j;
        }
    }
}
```

Una vez creado G, se incializan las estructuras de datos que nos ayudaran a determinar cual sera el mejor camino a tomar por el grafo dirigido para formar la supercadena (en particular las estructura de DP

se incializa con $+\infty$). cabe resaltar que la operacion $(1\acute{}N)$ se realiza para ahorrar tiempo de computo en las multiplicaciones , debido a que para las estructuras auxiliares a nuestro algoritmo es necesario iterar sobre el conjunto de datos (N), 2^N veces para hallar la respuesta optima al problema. En cuanto a ecuacion de recurrencia para el algoritmo, se sigue que:

$$sup(i,j) = \begin{cases} sup(i,j) = ||m[j]||, i = j \\ sup(i,j) = \text{mín}\{m(i-1,k) + G[k][j], m(i,j)\}, i \neq j \wedge (0 \leq k \leq N) \end{cases}$$

$$p(i,j) = \begin{cases} p(i,j) = 0, i = j \\ p(i,j) = k, i \neq j \wedge (0 \leq k \leq N) \wedge (m(i-1,k) + G[k][j] < m(i,j)) \end{cases}$$

Una vez completada esta parte de la funcion, se tienen los valores aproximadamente optimos para la longitud de la super cadena, en el caso del ejemplo del proyecto, las estructucturas auxilares quedan en el siguiente estado:

dp =

i/j	0	1	2	3
0	∞	∞	∞	∞
1	3	∞	∞	∞
2	∞	3	∞	∞
3	4	5	∞	∞
4	∞	∞	3	∞
5	4	∞	6	∞
6	∞	6	4	∞
7	5	6	6	∞
8	∞	∞	∞	3
9	6	∞	∞	5
10	∞	5	∞	6
11	6	7	∞	6
12	∞	∞	6	6
13	7	∞	8	6
14	∞	8	6	7
15	7	8	9	7

path =

i/j	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	1	0	0	0
4	0	0	0	0
5	2	0	0	0
6	0	2	1	0
7	2	0	1	0
8	0	0	0	0
9	3	0	0	0
10	0	3	0	1
11	1	3	0	0
12	0	0	3	2
13	2	0	3	0
14	0	3	1	2
15	2	3	1	0

Con **dp** y **path** calculados, solo resta construir la cadena con su informacion. Para esto, se crea una pila usando **path** para construir un camino que cree una solucion posible al problema , y seguido a esto, desempilar lo elementos de la solucion posible con el tope en el primer nodo que habra que escoger para empezar a construir la minima supercadena y seguir asi hasta que no haya elementos en la pila. Al desempilar , hay que tneer en cuenta los calculos hechos al principio sobre que cadena se sobrelapaban en que indices sobre otras cadenas para combinarlas de manera que su combinacion a lo sumo , agregue un 1 caracter mas a la supercadena.

```
        StringBuilder msg = new StringBuilder();
        int current = (1 << N) - 1;

        Stack<Integer> stack = new Stack<>();

        while(current>0){
            stack.push(last);
            int temp = current;
            current -= (1<<last);
            last = path[temp][last];
        }

        int i = stack.pop();
        msg.append(m[i]);

        while (!stack.isEmpty()){
            int j = stack.pop();
            msg.append(m[j].substring(m[j].length() - G[i][j]));
            i = j;
        }
        return msg.toString();

    }
```

Todo esto formalmente:
Entiendase $(\cdot \sqsubset \cdot, \cdot \sqsupset \cdot)$ como prefijo y sufujo de $\cdot$, respectivamente:

check_suffix_prefix()

E/S	Tipo	Nombre	Descripcion
E	**String**	s	primera cadena de la pareja a verificar desde donde empieza a ser sufijo
E	**String**	t	segunda cadena de la pareja a verificar desde donde empieza a ser prefijo
S	**int**	a	indice en donde la cadena s es sufijo de t y la cadena t es prefijo de t , o solo el tamaño de la cadena a concatenar en caso de que ninguna sea prefijo o sufijo de la otra

$$Pre : \{(s,t) \in (S \times S)\}$$
$$Post : \{(\exists i : \mathbb{N} \mid 0 \leq i \leq k \ : \ s_i \sqsupset t \wedge (s_i \sqsubset t))\}$$

minimal_super_string()

E/S	Tipo	Nombre	Descripcion
E	**String[]**	m	lista de cadenas que haran parte y solo apareceran 1 vez en la supercadena final
E	**int**	N	Tamaño de los datos a analizar
S	**String**	msg	Supercadena final para la cual $(\forall s_i \mid s_i \in (m \subseteq P(k) : s_i \in \textbf{msg})$

$Pre : \{\texttt{true}\}$
$Post : \{\neg(\exists super : String \mid (\forall s_i \mid s_i \in (m \subseteq P(k) : s_i \in \textbf{super}) \ : \ 4||super|| < ||msg|| \vee 2||super|| < ||msg||)\}$

*Se toma en consideracion que el algoritmo es un algoritmo de aproximacion avaro de factor 4, conjeturado hasta factor 2

Analisis de Complejidad

$$\begin{aligned}\mathbf{T}(G) &= (N-1)^2 \\ \mathbf{T}(DP) &= (N-1)^2 \cdot (2^{(N-1)}) \\ \mathbf{T}(Emp) &= N \\ \mathbf{T}(DesEmp) &= N\end{aligned}$$

La Complejidad Temporal del algoritmo se deduce de que para el calculo de los datos que se usan para llegar a una posible solucion en el peor de los casos se haran al menos

$$\mathbf{T}(G) + \mathbf{T}(DP) + \mathbf{T}(Emp) + \mathbf{T}(DesEmp)$$

$$(N-1)^2 + (N-1) \cdot (2^{(N-1)}) + 2N$$

Operaciones, y por ley de sumas, se tiene que

$$T(N) = O((N-1)^2 \cdot (2^{(N-1)}))$$

Para la complejidad espacial es suficiente saber que se usan $\{G, dp, path, stack\}$ como estructuras de datos auxiliares para el calculo de la solucion, luego

$$\mathbf{S}(G) + \mathbf{S}(DP) + \mathbf{S}(path) + \mathbf{T}(stack)$$

$$N^2 + N \cdot 2^N + N \cdot 2^N + N$$

Y por Ley de Sumas se obtiene:

$$S(N) = O(N \cdot 2^N)$$

Comentarios Finales

Se realizaron pruebas con listas largas de cadenas de un k moderado y el algortimo propone soluciones que tienden a ser 2 veces peores que el resultado optimo, como se esperaba de su equivalencia con el problema NP-Hard de conjunto de cobertura y debido a su complejidad temporal, como era de esperarse, se toma su tiempo en calcular la super cadena. Para problemas con entradas pequeñas depende de si se esta lidiando con un caso promedio o con el peor caso, cuando ocurre el primero el algoritmo tiende a calcular la solucion optima en un 90 % de las veces, si ocurre lo segundo el algoritmo tiene a calcular soluciones con un factor de aproximacion mas grande que 2, que generalmente es un coeficiente que cumple $2 \leq \frac{a}{b} \leq 4$.