REPORTE TAREA 2 y 3
ALGORITMOS Y COMPLEJIDAD

«Explorando la Distancia entre Cadenas, una Operación a la Vez»

Carlos Vera Quezada

17 de noviembre de 2024

19:27

Resumen

En este reporte se explora la distancia de edición extendida, un algoritmo que calcula el número mínimo de operaciones necesarias para transformar una cadena en otra mediante sustituciones, inserciones, eliminaciones y transposiciones. El estudio se centra en comparar dos implementaciones: fuerza bruta y programación dinámica, considerando sus complejidades temporales y espaciales tanto en teoría como en práctica, ademas se busca comparar el uso de costos variables en las operaciones.

Se diseñaron distintos datasets, midiendo el tiempo de ejecución y consumo de memoria de las implementaciones. Los resultados confirman que la programación dinámica supera a la fuerza bruta en eficiencia para entradas grandes, mientras que esta última es mejor para entradas pequeñas. Además, se evidencia que los costos variables afectan la elección de las operaciones óptimas, pero no a la complejidad temporal o espacial.

Índice

1. Introducción

> "La algoritmia es una de las bases fundamentales de ciencias de la computación. Aunque otras areas han ganado terreno últimamente e.g., ciencia de datos, inteligencia artificial y deep learning , la algoritmia sigue siendo fundamental para proveer soluciones eficientes a muchos de los problemas que aparecen en esas areas. Es, de alguna manera, un area transversal a ciencias de la computación."*Algoritmos Discretos: Análisis y Diseño* [2]

En este informe se estudiará la distancia de edición extendida o también conocida como **Optimal String Alignment (OSA)**, este algoritmo nos permite calcular el numero mínimo de operaciones para transformar una cadena de caracteres en otra, sin modificar mas de una vez alguna de sus subcadenas. Las operaciones corresponden a **Sustitución, Inserción, Eliminación y Transposición.** Donde cada una tiene un costo asociado y el cual se busca minimizar.

El algoritmo tiene diversas aplicaciones, dentro de ellas esta la búsqueda sobre documentos mediante escaneo, búsqueda en textos antiguos, búsqueda en bases de datos biológicas y corrección ortográfica. Lo cual hace que este algoritmo sea igual de importante para la ciencia que para la vida cotidiana. Ademas, es importante mencionar que debido a la cantidad de datos a analizar en los distintos campos, la eficiencia del algoritmo es algo que se ha ganado importancia a traves del tiempo. [2]

Con este informe se busca implementar el algoritmo mediante dos metodologías de diseño distintas, fuerza bruta y programación dinámica. Las cuales en la teoría, poseen distintas complejidades temporales y espaciales. Esto se hará con el fin de comparar empíricamente las dos implementaciones y contrastarlo con la teoría, para asi determinar cual de las dos es mejor.

Teóricamente, la implementación del algoritmo mediante fuerza bruta posee una complejidad temporal perteneciente a $O(4^n)$, donde el 4 proviene de la cantidad de operaciones a probar y el n corresponde al tamaño de la cadena mas larga, la complejidad espacial del algoritmo pertenece a $O(m)$ (Que corresponde a la profundidad de la pila de recursion). Por otra parte, la implementación mediante programación dinámica posee una complejidad temporal y espacial perteneciente a $O(n * m)$ donde n y m corresponden al largo de las cadenas.

Sobre el papel, a medida que crece el tamaño de la entrada, el algoritmo bajo programación dinámica debería ser mucho mejor, por otra parte, el algoritmo de fuerza bruta debería utilizar menor cantidad de espacio adicional, por lo cual dependiendo del escenario, uno podría ser mejor que el otro. Es significativo realizar estas comparaciones ya que puede ocurrir que asintóticamente una implementación sea mejor que otra, pero en la practica solo se cumpla para entradas especificas. Por lo cual, nos podemos preguntar ¿Esto se refleja en escenarios reales?

Para realizar las comparaciones se medirá el tiempo de ejecución de cada implementación al igual que el consumo de memoria RAM para las mismas entradas, con el fin de reducir desvíos en las mediciones, se usara el promedio de varias ejecuciones para dar un resultado mas significativo.

2. Diseño y Análisis de Algoritmos

2.1. Fuerza Bruta

1) Solución diseñada:

Se diseñó un algoritmo recursivo de fuerza bruta para calcular la distancia mínima de edición entre dos cadenas, S1 y S2. Este algoritmo utiliza índices i y j para comparar caracteres de las cadenas y una lista para registrar las operaciones óptimas que minimizan el costo de transformación.

El algoritmo explora todas las posibles combinaciones de operaciones (inserción, eliminación, sustitución, y transposición) y selecciona la que tiene el menor costo, basado en una matriz de costos. La solución diseñada fue una iteración sobre al algoritmo de fuerza bruta recursiva propuesto en el blog AfterAcademy [1] , donde se modifico para agregar la operación de transposición y llevar el registro de operaciones optimas.

2) Pseudocódigo:

```
1  Function FuerzaBruta(s1, s2, i, j):
2      m ← |s1|, n ← |s2|
3      if i = m and j = n then
4          return 0
5      if i = m then
6          return CostoInsertar(s2[j]) + FuerzaBruta(s1, s2, i, j + 1)
7      if j = n then
8          return CostoEliminar(s1[i]) + FuerzaBruta(s1, s2, i + 1, j)
9      if i <m and j <n and s1[i] = s2[j] then
10         return FuerzaBruta(s1, s2, i + 1, j + 1)
11     else
12         ins ← CostoInsertar(s2[j]) + FuerzaBruta(s1, s2, i, j + 1)
13         del ← CostoEliminar(s1[i]) + FuerzaBruta(s1, s2, i + 1, j)
14         sub ← CostoSustituir(s1[i], s2[j]) + FuerzaBruta(s1, s2, i + 1, j + 1)
15         trans ← INT_MAX
16         if i + 1 <m and j + 1 <n and s1[i] = s2[j + 1] and s1[i + 1] = s2[j] then
17             trans ← CostoTransponer(s1[i], s1[i + 1]) + FuerzaBruta(s1, s2, i + 2, j + 2)
18     costo_min ← min({ins, del, sub, trans})
19     return costo_min
```

3) Ejecución del algoritmo

Para ejemplificar el funcionamiento del algoritmo, usaremos las cadenas abba y baba, ademas, todos los valores de las operaciones serán 1, excepto por la operación de transposición que tendrá valor 2, los pasos son los siguientes:

1) Como primer paso, se definen los valores de m y n, ademas se comprueba si alguno de los indices ha llegado al final, como no es el caso se sigue la ejecución.

2) Luego, se comparan S1[0]='a' y S2[0]='b', al ser distintos estos ingresan al bloque ELSE donde se calcularan el costo de todas las operaciones, este calculo se compone de llamar la función de costo respectiva y realizar la llamada recursiva para continuar con el resto de caracteres, probando las 4 operaciones por cada indice.

3) Al volver la llamada recursiva a la llamada raíz, se calcula el mínimo entre las operaciones iniciales, se guarda el valor y se procede a agregar las acciones a la lista. esto ocurre en cada llamada recursiva.

4) Como resultado de la ejecución, el programa determina que la distancia mínima de edición que transforma la cadena 'abba' en 'baba' es de 1, lo cual corresponde a realizar una transición del primer y segundo carácter. En el caso que el valor de transponer fuera 3, el coste mínimo de edición pasaría a ser producido por la combinación de inserciones y eliminaciones.

4) Complejidad temporal y espacial

Si consideramos m y n como el tamaño de las cadenas S1 y S2, respectivamente, tenemos que el algoritmo propuesto posee una complejidad temporal perteneciente a $O\left(4^{\min(n,m)}\right)$, esto se debe a que el algoritmo tiene que calcular todas las posibles combinaciones, donde por cada carácter de alguna de las cadenas, existen 4 opciones factibles, ademas, se usa el mínimo entre los dos tamaños ya que al llegar al final de una de las cadenas, se sigue completando los caracteres restantes con operaciones de inserción o eliminación, lo cual nos lleva a tiempo lineal.

Al ser un algoritmo que utiliza recursion y ademas solo almacena variables de manera constante, la complejidad espacial del algoritmo pertenece a $O(\min(n,m))$ donde este mínimo corresponde a la altura del árbol de recursion.

Ademas, se podría considerar la memoria que se usa para almacenar las operaciones que producen la distancia de edición minima, donde el máximo de operaciones por cada par de cadenas se define como m+n, por lo tanto la memoria necesaria para almacenar las operaciones en el peor caso pertenece a $O(m+n)$

5) Transposiciones y costos variables

En el algoritmo implementado, la complejidad depende directamente de la cantidad de operaciones disponibles, al agregar la operación de transposición, la complejidad aumenta desde $O\left(3^{\min(n,m)}\right)$ a $O\left(4^{\min(n,m)}\right)$, por otra parte, los costos variables modifican las operaciones que producen la distancia de edición minima, pero no aumentan el orden de la complejidad, ya que estos costos están almacenados en memoria y recuperar su valor tiene un costo de $O(1)$.

2.2. Programación Dinámica

1) Solución diseñada

Como solución diseñada, se opto por implementar un algoritmo de programación dinámica que utiliza el método iterativo bottom up, el cual va construyendo la solución a medida que resuelve

los subproblemas que lo componen. En este caso, el algoritmo recibe como entrada las cadenas S_1 y S_2 a comparar y devuelve el costo de edición mínimo, ademas se implemento una matriz auxiliar la cual va almacenando las operaciones que producen el costo mínimo.

Para calcular la distancia de edición, el algoritmo calcula todos los subproblemas posibles y almacena los que producen el costo mínimo en una matriz, donde al completarla, el resultado estará en la ultima posición.

La solución diseñada fue una iteración sobre al algoritmo de programación dinámica iterativo propuesto en el blog AfterAcademy [1] , donde se modifico para agregar la operación de transposición y llevar el registro de operaciones optimas.

2) Pseudocódigo

```
1  Function PROGRAMACIONDINAMICA(S1, S2):
2      m ← |S1|, n ← |S2|
3      Matriz dp[m][n] = 0; Para todas las casillas de la matriz
4      for i = 1 to m do
5          dp[i][0] = dp[i-1][0] + costo_eliminacion(S1[i-1])
6      for j = 1 to n do
7          dp[0][j] = dp[0][j-1] +costo_insercion(S2[j-1])
8      for i = 1 to m do
9          for j = 1 to n do
10             costoIns, costoDel, costoSub = costo_insercion(S2[j-1]) , costo_eliminacion(S1[i-1])
                costo_sustitucion(S1[i-1], S2[j-1])
11             dp[i][j] = dp[i-1][j-1] + costoSub
12             if dp[i][j-1] + costoIns <dp[i][j] then
13                 dp[i][j] = dp[i][j-1] + costoIns
14             if dp[i-1][j] + costoDel <dp[i][j] then
15                 dp[i][j] = dp[i-1][j] + costoDel
16             if i + 1 <m and j + 1 <n and S1[i] = S2[j + 1] and S1[i + 1] = S2[j] then
17                 if dp[i-2][j-2] + costo_transposicion(S1[i-2], S1[i-1]) <dp[i][j] then
18                     dp[i][j] = dp[i-2][j-2] + costo_transposicion(S1[i-2], S1[i-1])
19     return dp[m][n]
```

3) Ejecución del algoritmo

Para ejemplificar el funcionamiento del algoritmo, usaremos las cadenas abba y baba, ademas, todos los valores de las operaciones serán 1, excepto por la operación de transposición que tendrá valor 2, los pasos son los siguientes:

1) Como primer paso, se definen los valores de m, n y la matriz dp la cual almacena los resultados parciales, esta se inicializa en tamaño [m+1][n+1], para poder manejar casos con cadenas vacías.

2) Luego, se llenan la primera columna y fila de la matriz, con las operaciones de eliminar y insertar para manejar los casos con cadenas vacías.

3) Seguido se entra al bucle principal, donde para $S_1[0]$='a' y $S_2[0]$='b' se calcularan todas las operaciones llamando a las funciones de costos, para luego tomar como base la operación se sustitución con la cual se realizaran las comparaciones para determinar cual de todas las operaciones produce el menor costo, modificando el valor en la matriz dp.

4) Para el ejemplo propuesto, una vez llenada la matriz dp, se retornara la distancia de edición minima, la cual corresponde a transponer los primeros dos caracteres y su costo es 1. En el caso que el valor de transponer fuera 3, el coste mínimo de edición pasaría a ser producido por la combinación de inserciones y eliminaciones.

4) Complejidad temporal y espacial

Si consideramos m y n como el tamaño de las cadenas S_1 y S_2, respectivamente, tenemos que el algoritmo propuesto posee una complejidad temporal perteneciente a $O(m*n)$, esto se debe a que el algoritmo realiza m x n operaciones con costo constante, lo que corresponde a cada combinación de caracteres de las cadenas de entrada, ademas, por su naturaleza de programación dinámica, aprovecha los cálculos previos almacenados en la matriz para evitar recalcular algunos costos.

Al ser un algoritmo de programación dinámica, este almacena los valores parciales en una matriz, la complejidad espacial del algoritmo pertenece a $O(m*n)$ ya que tiene que almacenar m+1 x n+1 resultados, esto ocurre en todos los casos ya que siempre se llenara la matriz al completo.

Ademas, se podría considerar la memoria que se usa para almacenar las operaciones que producen la distancia de edición minima, la cual funciona igual a la matriz dp, por lo tanto se necesaria $O(m*n)$ espacio adicional, por lo cual la complejidad espacial del algoritmo se mantiene igual.

5) Transposiciones y costos variables

La inclusion de la operación de transposición no afecta a la complejidad, ya que esta depende solamente del tamaño de la entrada, por otro lado, los costos variables tampoco afectan a la complejidad, ya que al estar almacenados en un vector, el costo de consultar los valores es constante.

2.2.1. Descripción de la solución recursiva

El problema de distancia de edición extendida se puede resolver mediante la resolución y composición de subproblemas, ya que al resolver estos subproblemas obtenemos su optimo local, lo cual al volver a componer la solución general nos entrega una solución la cual también es optima, En especifico, si tenemos S_1[0:i] y S_2[0:j], para resolver el problema necesitamos conocer como se transforma S_1[0:i-1] a S_2[0:j-1], asi sucesivamente.

Esta solución se demuestra mediante el principio de optimalidad de Bellman [3], el cual describe lo mencionado.

2.2.2. Relación de recurrencia

Dada dos cadenas S_1 y S_2, donde $S_1 = S_1[1..m]$ y $S_2 = S_2[1..n]$, la distancia de edición extendida DE(i, j), para (i,j) hasta (m,n), se define como:

$$\text{DE}(i,j) = \begin{cases} 0 & \text{si } i = 0 \text{ y } j = 0, \\ j \cdot \text{costo\_insercion}(S_2[j-1]) & \text{si } i = 0, \\ i \cdot \text{costo\_eliminacion}(S_1[i-1]) & \text{si } j = 0, \\ \min\{\text{DE}(i-1, j-1) + \text{costo\_replace}(S_1[i-1], S_2[j-1]), & \\ \quad \text{DE}(i, j-1) + \text{costo\_insert}(S_2[j-1]), & \\ \quad \text{DE}(i-1, j) + \text{costo\_delete}(S_1[i-1]), & \\ \quad \text{DE}(i-2, j-2) + \text{costo\_trans}(S_1[i-2], S_1[i-1])\} & \text{si } i > 0 \text{ y } j > 0. \end{cases}$$

2.2.3. Identificación de subproblemas

Los subproblemas los podemos categorizar con las 4 operaciones, entonces:

1. Inserción: Para resolver el costo de inserción con (i,j), se necesita calcular la distancia entre los primeros i caracteres de S_1 y los primeros j-1 caracteres de S_2 para luego sumarle el costo de insertar el carácter S_2[j-1].

2. Eliminación: Para resolver el costo de eliminación con (i,j), se necesita calcular la distancia entre los primeros i-1 caracteres de S_1 y los primeros j caracteres de S_2 para luego sumarle el costo de eliminar el carácter S_1[i-1].

3. Sustitución: Para resolver el costo de sustitución con (i,j), se necesita calcular la distancia entre los primeros i-1 caracteres de S_1 y los primeros j-1 caracteres de S_2 para luego sumarle el costo de sustituir el carácter S_1[i-1] por el carácter S_2[j-1].

4. Transposición: Para resolver el costo de transposición con (i,j), se necesita calcular la distancia entre los primeros i-2 caracteres de S_1 y los primeros j-2 caracteres de S_2 para luego sumarle el costo de transponer los caracteres S_1[i-2] y S_1[i-1].

Con esto, podemos ver que los subproblemas se solapan, lo que nos da la oportunidad de optimizar el algoritmo evitando recalcular ciertos resultados.

2.2.4. Estructura de datos y orden de cálculo

Como estructura de datos se utilizo una matriz para almacenar los resultados parciales óptimos, los cuales compondrán el resultado final, ademas esta matriz es llenada desde izquierda a derecha y arriba hacia abajo, esto, ya que como definimos en los subproblemas, se necesitan los resultados anteriores o con menor indice para componer la solución general.

3. Implementaciones

Los algoritmos fueron implementados en el lenguaje de programación c++, los archivos son los siguientes:

1. bf.cpp: archivo que contiene el algoritmo de fuerza bruta, ademas de toda la lógica para la ejecución del algoritmo, como la lectura de las matrices de costos, funciones de costos,función principal y almacenamiento de la salida en el archivo output_bf.txt. La salida por consola del programa muestra las cadenas las cuales fueron comparadas, la distancia minima de edición, el tiempo de ejecución y si el resultado coincide con la distancia de edición esperada. El código y sus instrucciones de ejecución se pueden encontrar aquí.

2. dp.cpp: archivo que contiene el algoritmo de fuerza bruta, ademas de toda la lógica para la ejecución del algoritmo, como la lectura de las matrices de costos, funciones de costos, función principal y almacenamiento de la salida en el archivos output_dp.txt. La salida por consola del programa muestra las cadenas las cuales fueron comparadas, la distancia minima de edición, el tiempo de ejecución y si el resultado coincide con la distancia de edición esperada. El código y sus instrucciones de ejecución se pueden encontrar aquí.

4. Experimentos

Para los experimentos, se utilizo el siguiente entorno:

1. Hardware
   - Procesador Ryzen 7 5700x3D, 96MB Cache L3, Base clock 3.0GHz , Boost clock 4.05GHz
   - 16GB Ram DDR4 3200MT/s Dual-Channel
   - SSD NVMe PCIe-3.0
2. Software
   - Subsistema de Windows para Linux basado en Ubuntu 22.04.5 LTS
   - 16 Núcleos Virtuales asignados
   - 8GB de RAM asignados
   - 2GB de memoria Swap
   - g++ 11.4.0
   - valgrind 3.18.1

4.1. Dataset (casos de prueba)

La extensión máxima para esta sección es de 2 páginas.

Se diseñaron 10 Datasets con casos de prueba, cada dataset contiene 20 pares de cadenas y su distancia de edición esperada, la cual fue calculada con la librería de python weighted-levenshtein, que permite trabajar con el calculo de OSA y costos personalizados

Cuando se habla de matrices de costo estándar, esto hace referencia a que los costos de las operaciones son los siguientes:

- Inserción: 1, Para cualquier carácter
- Eliminación: 1, Para cualquier carácter
- Sustitución: 2, Para cualquier par de caracteres distintos
- Transposición: 1, Para cualquier par de caracteres distintos

1. Datasets
   - Transposiciones con matrices de costo estándar: este dataset cuenta con cadenas de largo 8 , donde son necesarias las transposiciones, con matrices de costo estándar.
   - Transposiciones con matrices de costo con valores modificados para la transposición: este dataset cuenta con cadenas de largo 8, donde son necesarias las transposiciones, con matrices de costo con valores mas elevados para la operación de transposición.

- Cadenas vacías: dataset con una cadena vacía y otra de largo entre 5 y 8, con matrices de costo estándar.
- Caracteres repetidos: dataset con par de cadenas con caracteres repetidos hasta 4 veces, de largo 8, con matrices de costo estándar.
- Palabras aleatorias con matrices de costo estándar: diversos datasets con cadenas de tamaño menor a 2, 3 a 4, 7, 8 y 12, con matrices de costo estándar, estas palabras pertenecen al diccionario ingles.
- Palabras aleatorias con matrices de costo modificado: dataset con cadenas de largo 8, estas palabras pertenecen al diccionario ingles, donde las operaciones tienen los costos: Inserción=2, Eliminación=2, Sustitución=3, Transposición=2.

La documentación y resultados de los casos de prueba se encuentran en el siguiente recurso.

4.2. Resultados

Los resultados de los casos de prueba se encuentran organizados en un recurso dividido por carpetas, donde se almacenan los archivos de salida de los programas **bf.cpp**,**dp.cpp** y el dataset correspondiente. Estos archivos contienen las cadenas comparadas, la distancia calculada, la distancia esperada correcta, un indicador con la correctitud del resultado, el tiempo de ejecución, y las operaciones que producen la distancia mínima de edición junto con sus costos. Además, al final de cada archivo, se muestra el tiempo promedio de ejecución del dataset procesado.

Para obtener los resultados a continuación, es necesario ejecutar los programas mencionados. El tiempo fue medido utilizando la biblioteca **chrono** de C++. También se considera el tiempo promedio de ejecución como medida representativa, salvo que se indique lo contrario.

El uso de memoria RAM se determinó utilizando la herramienta **Massif** de Valgrind, que toma muestras del uso de memoria durante la ejecución del programa y genera un gráfico con estas métricas.

Las instrucciones detalladas para la ejecución de los programas se encuentran aquí.

1. Transposiciones con matrices de costo estándar y costo modificado

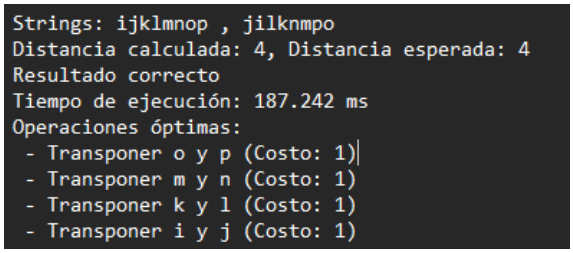
Tipo/Data	Transposiciones costo estándar	Transposiciones costo modificado
Fuerza Bruta (ms)	188.489	193.375
Programación Dinámica (ms)	0.0956651	0.0860588
Memoria FB (KB)	103.2	105.5
Memoria PD (KB)	134.4	142.1

Cuadro 1: Tabla con valores de prueba para transposiciones

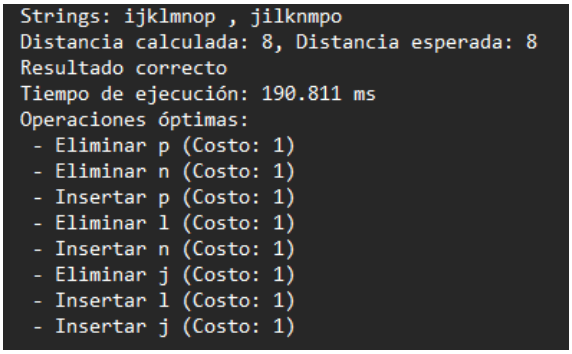
Para las pruebas de transposiciones con costo estándar, podemos observar que al algoritmo de fuerza bruta le toma mucho mas tiempo en resolver la consulta en comparación al algoritmo de programación dinámica, ademas el uso de memoria es mayor para el algoritmo de programación dinámica, pero esta diferencia en uso de memoria no es proporcional a la diferencia en tiempo de ejecución.

Para las pruebas de transposiciones con costo modificado, se sigue la misma tendencia en términos de tiempo de ejecución y uso de memoria que para las transposiciones con costo estándar.

No obstante, lo que si cambió fueron las operaciones las cuales producen la distancia de edición minima, para las transposiciones con costo estándar, siempre se selecciono la operación de transposición (Esto ocurre para ambos algoritmos). Por otro lado, para las transposiciones con costo modificado, al aumentar el costo de realizar una transposición, el algoritmo determinó que se podia llegar a la distancia de edición minima mediante otras operaciones. Ejemplo:



(a) Transposiciones con costo estándar.



(b) Transposiciones con costo modificado.

Figura 1: Ejemplo ejecución de transposiciones.

Por lo tanto, al modificar los costos de las operaciones, no se modifica en gran medida el tiempo de ejecución de los algoritmos al igual con el uso de memoria, pero si las operaciones las cuales producen la distancia de edición minima. Por lo tanto, se evidencia que los costos de de las operaciones no afectan a la complejidad temporal y espacial.

2. Cadenas vacías y caracteres repetidos

Tipo/Data	**Cadenas vacías**	**Caracteres repetidos**
Fuerza Bruta (ms)	0.0033952	183.942
Programación Dinámica (ms)	0.0099436	0.0952444
Memoria FB (KB)	97.15	106.5
Memoria PD (KB)	100.7	143.9

Cuadro 2: Tabla con valores de prueba para cadenas vacías y caracteres repetidos

Para el dataset de cadenas vacías podemos observar que al algoritmo de fuerza bruta le toma menos tiempo en resolver la consulta con respecto al de programación dinámica. Por otro lado, el uso de memoria es similar en los dos casos.

Estos resultados se deben a las complejidades temporales y espaciales de cada algoritmo, para el algoritmo de fuerza bruta, su tiempo de ejecución es exponencial con respecto al tamaño mas pequeño de la entrada, al tener una entrada con largo 0, el algoritmo resuelve el problema muy eficientemente, podríamos decir que le toma tiempo $4^0 = 1$.

Por el lado del algoritmo de programación dinámica, su tiempo de ejecución se modela según los tamaños de las dos entradas mas 1, por lo tanto, podríamos decir que le toma tiempo $1 * 9 = 9$, lo cual es mayor que el algoritmo de fuerza bruta.

Ademas, los usos de memoria son similares ya que el algoritmo de programación dinámica basa su uso de memoria en los tamaños de las dos entradas, al tener una de las entradas pequeñas, el uso de memoria disminuye.

Sobre el dataset de caracteres repetidos, podemos observar que el algoritmo de fuerza bruta es mas ineficiente con respecto al de programación dinámica, no obstante, el uso de memoria es mayor en este ultimo.

Con esto, podemos evidenciar en la practica que las complejidades temporales y espaciales para cada algoritmo se cumplen.

3. Palabras aleatorias con matrices de costo estándar y modificado

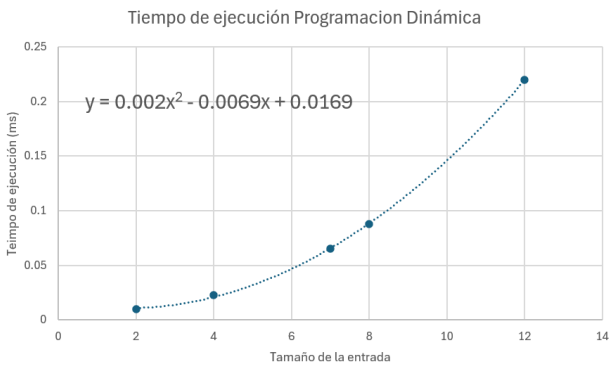
Data/Tipo	Fuerza Bruta (ms)	Programación Dinámica (ms)	Memoria FB (KB)	Memoria PD (KB)
Menor a 2	0.00890745	0.0101063	97.55	98.02
3 a 4	0.15021	0.0225789	99.37	104
7	33.0569	0.0653561	105.2	131.1
8	180.548	0.0875144	108.2	145.6
8 Modificado	181.627	0.0992815	105.5	145.2
12	172570	0.219571	No medido	252.5

Cuadro 3: Tabla con valores de prueba para cadenas aleatorias

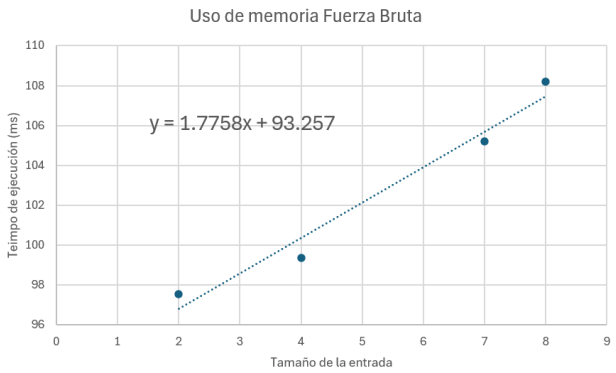
El uso de memoria para las palabras de largo 12 con el algoritmo de fuerza bruta no pudo ser medido ya que se excedía el tiempo de ejecución con la utilidad **massif**.

Para los datasets de palabras aleatorias podemos ver que nuevamente, el algoritmo de fuerza bruta es mas ineficiente frente al algoritmo de programación dinámica, exceptuando cuando la entrada es muy pequeña. Ademas podemos ver mas claramente el comportamiento a medida que crece el tamaño de la entrada:

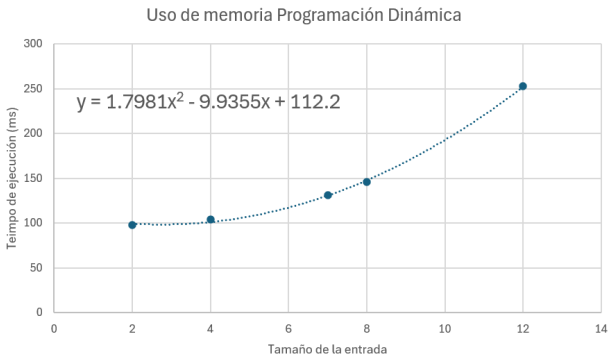
(a) Tiempo de ejecución Fuerza Bruta.



(b) Tiempo de ejecución Programación Dinámica.



(c) Uso de memoria Fuerza Bruta.



(d) Uso de memoria Programación Dinámica.

Los gráficos representan el tiempo de ejecución de los algoritmos a medida que crece la entrada, al igual que el uso de memoria, ademas se les agrego una linea de tendencia en conjunto con la ecuación del gráfico.

Con esto, podemos determinar que a medida que crece la entrada, el tiempo de ejecución del algoritmo de fuerza bruta se comporta de manera exponencial y su memoria se comporta de manera lineal.

Por otra parte, el tiempo de ejecución del algoritmo de programación dinámica se comporta de manera cuadrática, al igual que su uso de memoria.

Ademas, con el dataset con costos modificados, no se observo mayores diferencias con respecto al resto de las pruebas, exceptuando la secuencia de operaciones que producen la distancia de edición optima, la cual cambio con respecto a las matrices de costo estándar.

5. Conclusiones

El análisis e implementación del algoritmo de distancia de edición extendida, mediante las técnicas de fuerza bruta y programación dinámica, permitieron validar empíricamente lo descrito por la teoría en base a sus complejidades temporales y espaciales. Los resultados demuestran que, mientras el algoritmo basado en programación dinámica sobresale en términos de eficiencia temporal, el uso de memoria aumenta a medida que crece la entrada. Por otra lado, el algoritmo basado en fuerza bruta presenta un desempeño adecuado solo en entradas de tamaño reducido, a medida que crece la entrada, la eficiencia de este algoritmo empeora, pero su uso de memoria se comporta de manera lineal.

Este reporte no solo confirma que la programación dinámica es más adecuada para entradas más grandes, sino que también evidencia cómo la elección del algoritmo depende del contexto, ya que el bajo uso de memoria del algoritmo de fuerza bruta podría ser un factor clave. Por otro lado, los costos variables sobre la secuencia de operaciones óptimas no represento un cambio en los tiempos de ejecución o uso de memoria, reafirmando que estos parámetros no afectan a las complejidades, pero sí al comportamiento del algoritmo.

En resumen, el trabajo logra comparar exitosamente las distintas técnicas de diseño de algoritmos, comprobando empíricamente la teoría, asi, se puede seguir desarrollando en base a lo propuesto para mejorar estas técnicas.

6. Condiciones de entrega

- La tarea se realizará individualmente (esto es grupos de una persona), sin excepciones.
- La entrega debe realizarse vía `http://aula.usm.cl` en un tarball en el área designada al efecto, en el formato `tarea-2 y 3-rol.tar.gz` (`rol` con dígito verificador y sin guión).

  Dicho tarball debe contener las fuentes en LaTeX 2ε (al menos `tarea-2 y 3.tex`) de la parte escrita de su entrega, además de un archivo `tarea-2 y 3.pdf`, correspondiente a la compilación de esas fuentes.
- Si se utiliza algún código, idea, o contenido extraído de otra fuente, este **debe** ser citado en el lugar exacto donde se utilice, en lugar de mencionarlo al final del informe.
- Asegúrese que todas sus entregas tengan sus datos completos: número de la tarea, ramo, semestre, nombre y rol. Puede incluirlas como comentarios en sus fuentes LaTeX (en TeX comentarios son desde % hasta el final de la línea) o en posibles programas. Anótese como autor de los textos.
- Si usa material adicional al discutido en clases, detállelo. Agregue información suficiente para ubicar ese material (en caso de no tratarse de discusiones con compañeros de curso u otras personas).
- No modifique `preamble.tex`, `tarea_main.tex`, `condiciones.tex`, estructura de directorios, nombres de archivos, configuración del documento, etc. Sólo agregue texto, imágenes, tablas, código, etc. En el códigos funte de su informe, no agregue paquetes, ni archivos .tex (a excepción de que agregue archivos en `/tikz`, donde puede agregar archivos .tex con las fuentes de gráficos en `TikZ`).
- La fecha límite de entrega es el día 10 de noviembre de 2024.

NO SE ACEPTARÁN TAREAS FUERA DE PLAZO.

- Nos reservamos el derecho de llamar a interrogación sobre algunas de las tareas entregadas. En tal caso, la nota de la tarea será la obtenida en la interrogación.

NO PRESENTARSE A UN LLAMADO A INTERROGACIÓN SIN JUSTIFICACIÓN PREVIA SIGNIFICA AUTOMÁTICAMENTE NOTA 0.

A. Apéndice 1

Aquí puede agregar tablas, figuras u otro material que no se incluyó en el cuerpo principal del documento, ya que no constituyen elementos centrales de la tarea. Si desea agregar material adicional que apoye o complemente el análisis realizado, puede hacerlo en esta sección.

> Esta sección es solo para material adicional. El contenido aquí no será evaluado directamente, pero puede ser útil si incluye material que será referenciado en el cuerpo del documento. Por lo tanto, asegúrese de que cualquier elemento incluido esté correctamente referenciado y justificado en el informe principal.

Referencias

[1] AfterAcademy. *Edit Distance*. https://afteracademy.com/blog/edit-distance-problem/. Abr. de 2022.

[2] Diego Arroyuelo. *Algoritmos Discretos: Análisis y Diseño*. Mar. de 2022.

[3] Rossel Keila. *Optimización con programación dinámica*. Ene. de 2022.