REPORTE TAREA 2 y 3
ALGORITMOS Y COMPLEJIDAD

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

Joaquin Antonio Veliz Carmona

17 de noviembre de 2024

03:46

Resumen

*En este trabajo se analizan dos enfoques para resolver el problema de distancia mínima de edición de cadenas, donde buscamos transformar la cadena S_1 en S_2 utilizando cuatro operaciones posibles. Los enfoques evaluados son **fuerza bruta** y **programación dinámica**, comparando su eficiencia y rendimiento. Para llevar a cabo el análisis, generaremos una amplia variedad de datasets utilizando Python y aplicaremos algoritmos implementados en C++ para resolver el problema. Los datos obtenidos serán procesados y visualizados mediante librerías de Python, presentando los resultados a través de gráficos. Finalmente, se discutirán las conclusiones obtenidas a partir de la investigación.*

Índice

1. Introducción

En este informe se analizará el problema de distancia mínima de edición de cadenas. Problema en el que, dada una tabla de costo respectiva para cada una de las 4 operaciones que poseemos, queremos ver cómo podemos convertir la primera cadena en la segunda "gastando"lo menos posible. Esto es un problema bastante sencillo en la teoría, ya que es algo que cualquiera podría hacer a mano. El problema viene cuando queremos hacerlo con cadenas más grandes, entonces lo podríamos programar... pero ¿cómo? Lo primero que nos podría venir a la cabeza es la metodología o paradigma de **fuerza bruta**, el cual resuelve el problema, pero dependerá del largo que queramos resolver. Si antes decíamos que cualquiera lo podría hacer con lápiz y papel, eso sería hasta unos 5 caracteres en más o menos un minuto... si se lo damos al computador con un programa de fuerza bruta, tardaremos más de 5 minutos si queremos superar los 14 caracteres, al menos en el computador en el que se hicieron las pruebas.

Esto claramente no es una solución si lo que se busca es trabajar con cadenas más grandes, y hoy en día... ¿qué son 14 caracteres? Para eso necesitamos una solución más rápida. Para eso utilizaremos la **programación dinámica**, la cual nos permite dividir nuestro problema en subproblemas y guardar sus soluciones para no tener que recalcular como haríamos con fuerza bruta, lo cual nos permitirá bajar drásticamente la complejidad temporal del problema, ya que, como veremos más adelante, podremos en un quinto del tiempo recorrer cadenas de **2500** caracteres lo cual demuestra a propios y extraños una mejora significativa del rendimiento, con los mismos resultados. Esto nos permitiría trabajar con cadenas de órdenes de magnitud tremendos en comparación, sin tener que dejar el computador tanto tiempo prendido resolviendo nuestro problema, lo cual también es un ahorro energético.

2. Diseño y Análisis de Algoritmos

2.1. Fuerza Bruta

La solución de fuerza bruta explora todas las posibles combinaciones de operaciones para transformar una cadena en otra. El algoritmo intenta aplicar cada operación en cada paso, calculando el costo acumulado para cada ruta posible de transformaciones. Luego, selecciona la secuencia de operaciones con el costo mínimo total. Este enfoque recursivo evalúa todas las rutas posibles y, aunque garantiza encontrar la solución óptima, su complejidad exponencial hace que sea ineficiente para cadenas largas debido al gran número de subproblemas repetidos que resuelve sin aprovechar la memoria para almacenar resultados parciales.

Algoritmo 1: Algoritmo de Fuerza Bruta para la Distancia Mínima de Edición

```
1  Procedure EditDistanceBruteForce(S1, S2, i, j)
2      if i = longitud(S1) then
3          return (longitud(S2) - j) × CostoInsercion(S2[j])
4      else if j = longitud(S2) then
5          return (longitud(S1) - i) × CostoEliminacion(S1[i])
6      minCost ← ∞
       // 1. Sustitución
7      cost_subst ← CostoSustitucion(S1[i], S2[j]) + EditDistanceBruteForce(S1, S2, i + 1, j + 1)
8      minCost ← mín(minCost, cost_subst)
       // 2. Inserción
9      cost_ins ← CostoInsercion(S2[j]) + EditDistanceBruteForce(S1, S2, i, j + 1)
10     minCost ← mín(minCost, cost_ins)
       // 3. Eliminación
11     cost_del ← CostoEliminacion(S1[i]) + EditDistanceBruteForce(S1, S2, i + 1, j)
12     minCost ← mín(minCost, cost_del)
       // 4. Transposición (si aplica)
13     if i + 1 < longitud(S1) and j + 1 < longitud(S2) and S1[i] = S2[j + 1] and S1[i + 1] = S2[j] then
14         cost_trans ← CostoTransposicion(S1[i], S1[i + 1]) + EditDistanceBruteForce(S1, S2, i + 2, j + 2)
15         minCost ← mín(minCost, cost_trans)
16     return minCost
```

Ejemplo Resuelto Paso a Paso

Dadas las cadenas $s1 =$ ‘‘abcd’’ y $s2 =$ ‘‘acbd’’, calcularemos la distancia mínima de edición utilizando fuerza bruta. Comparamos primero los caracteres iniciales $s1[0]$ y $s2[0]$, ambos iguales a “a”, por lo que avanzamos sin costo. Luego, comparamos $s1[1] =$ ‘‘b’’ y $s2[1] =$ ‘‘c’’, que son distintos, por lo que consideramos cuatro operaciones: (1) **Sustitución** de “b” por “c” con costo 2, avanzando en ambas cadenas; (2) **Inserción** de “c” en $s1$ con costo 1, avanzando solo en $s2$; (3) **Eliminación** de “b” de $s1$ con costo 1, avanzando solo en $s1$; y (4) **Transposición** de “b” y “c” en $s1$ con costo 1, avanzando dos posiciones en ambas cadenas. Evaluando todas las opciones, encontramos que la transposición da el menor costo total de 1, ya que permite que los caracteres finales coincidan sin operaciones adicionales. Por lo tanto,

la mejor ruta es transformar “abcd” en “acbd” mediante una transposición de “b” y “c”.

Análisis de Complejidad de la Solución de Fuerza Bruta

En cada posición, el algoritmo puede optar por realizar una de las cuatro operaciones, lo que da lugar a múltiples ramas en cada llamada recursiva. La complejidad temporal de esta solución es aproximadamente $O(4^{\text{máx}(n,m)})$, donde n y m son las longitudes de las cadenas, ya que en el peor caso, existen cuatro opciones posibles en cada paso. Este crecimiento exponencial hace que el algoritmo sea ineficiente y poco práctico para cadenas largas. En cuanto a la complejidad espacial, el algoritmo utiliza un espacio de pila proporcional a la profundidad máxima de la recursión, que es $\text{máx}(n,m)$, resultando en una complejidad espacial de $O(\text{máx}(n,m))$. A diferencia de enfoques como la programación dinámica, la solución de fuerza bruta no almacena los resultados de subproblemas ya calculados, lo que contribuye a su alta ineficiencia.

La inclusión de transposiciones y costos variables incrementa la complejidad de la solución de fuerza bruta, ya que agrega opciones adicionales en cada paso y modifica el cálculo de los costos. Las transposiciones introducen una cuarta operación posible en cada posición, aumentando el número de llamadas recursivas y, por lo tanto, el tiempo de ejecución. Además, los costos variables hacen que el algoritmo deba calcular y comparar diferentes valores para cada operación, lo cual incrementa la carga computacional. Como resultado, la complejidad temporal del algoritmo se vuelve aún más alta, dificultando su uso en problemas de mayor tamaño.

2.2. Programación Dinámica

La implementación con programación dinámica (DP) utiliza una matriz dp de tamaño $(n+1)\times(m+1)$, donde n y m son las longitudes de las cadenas $s1$ y $s2$. Cada celda $dp[i][j]$ almacena el costo mínimo para transformar los primeros i caracteres de $s1$ en los primeros j caracteres de $s2$. Inicializamos la primera fila y columna para representar los casos en que una cadena está vacía, con costos acumulados de inserciones y eliminaciones. Luego, llenamos la matriz considerando, en cada posición, los costos de las operaciones posibles: sustitución, inserción, eliminación y transposición. Cada celda toma el valor mínimo entre estas opciones, acumulando el costo óptimo hasta ese punto. Al final, $dp[n][m]$ contiene la distancia mínima de edición entre ambas cadenas. Esta solución tiene una complejidad temporal y espacial de $O(n\times m)$, ya que calcula cada subproblema solo una vez y reutiliza los resultados almacenados en la matriz.

2.2.1. Descripción de la solución recursiva

La solución combina la recursión con el almacenamiento de resultados intermedios en una estructura matriz, lo que evita el cálculo redundante de subproblemas. Cada llamada recursiva evalúa las operaciones posibles y almacena el costo mínimo para transformar los primeros i caracteres de $s1$ en los

primeros j caracteres de $s2$ en una tabla $dp[i][j]$. Si una subsolución ya ha sido calculada, el algoritmo simplemente la reutiliza, en lugar de recalcularla. Esto permite reducir la complejidad temporal a $O(n \times m)$, donde n y m son las longitudes de las cadenas, al resolver cada subproblema solo una vez. Al final, $dp[n][m]$ contendrá el costo mínimo de edición para transformar una cadena en la otra, evitando la ineficiencia de la solución de fuerza bruta.

2.2.2. Relación de recurrencia

$$dp[i][j] = \begin{cases} i \cdot \text{CostoEliminacion} \rightarrow \text{si j = 0} \\ j \cdot \text{CostoInsercion} \rightarrow \text{si i = 0} \\ dp[i-1][j-1] \rightarrow \text{si s1[i-1] = s2[j-1]} \\ \min \begin{cases} dp[i-1][j-1] + \text{CostoSustitucion}(s1[i-1], s2[j-1]) \\ dp[i][j-1] + \text{CostoInsercion}(s2[j-1]) \\ dp[i-1][j] + \text{CostoEliminacion}(s1[i-1]) \\ dp[i-2][j-2] + \text{CostoTransposicion}(s1[i-2], s1[i-1]) \\ \rightarrow \text{si i >1, j >1, s1[i-1] = s2[j-2] y s1[i-2] = s2[j-1]} \end{cases} \end{cases}$$

2.2.3. Identificación de subproblemas

Cada subproblema se define como el costo mínimo para transformar los primeros i caracteres de la cadena $s1$ en los primeros j caracteres de la cadena $s2$. Así, el subproblema $dp[i][j]$ representa la distancia de edición entre estas dos subcadenas parciales. Para resolver $dp[i][j]$, se consideran tres operaciones básicas: sustitución, inserción, eliminación. Además, se evalúa una transposición si es aplicable, en caso de que los caracteres actuales de ambas cadenas sean intercambiables. Al resolver cada subproblema $dp[i][j]$ de manera independiente y reutilizar los resultados almacenados en la matriz dp, evitamos cálculos redundantes y logramos una solución eficiente para el problema general de transformar $s1$ en $s2$.

2.2.4. Estructura de datos y orden de cálculo

utilizamos matriz dp de tamaño $(n+1) \times (m+1)$, donde n y m son las longitudes de las cadenas $s1$ y $s2$, respectivamente. Cada celda $dp[i][j]$ en la matriz almacena el costo mínimo para transformar los primeros i caracteres de $s1$ en los primeros j caracteres de $s2$. La matriz se inicializa llenando la primera fila y la primera columna con los costos de inserciones y eliminaciones, correspondientes a los casos en los que una de las cadenas está vacía.

El orden de cálculo procede de izquierda a derecha y de arriba hacia abajo en la matriz dp, llenando cada celda $dp[i][j]$ en función de los valores previamente calculados en $dp[i-1][j]$, $dp[i][j-1]$, $dp[i-1][j-1]$, y, en el caso de transposición, $dp[i-2][j-2]$. Este orden asegura que cada subproblema

esté resuelto antes de ser utilizado en cálculos posteriores, permitiendo construir la solución de manera acumulativa y eficiente. Al finalizar, el valor en $dp[n][m]$ contiene la distancia mínima de edición entre las cadenas completas $s1$ y $s2$.

2.2.5. Algoritmo utilizando programación dinámica

Algoritmo 2: Algoritmo de Programación Dinámica para la Distancia Mínima de Edición

```
1  Procedure EditDistanceDynamic(S1, S2)
2      n ← longitud(S1)
3      m ← longitud(S2)
4      Crear matriz dp de tamaño (n+1) × (m+1) con valores iniciales ∞
5      dp[0][0] ← 0
6      for i ← 1 to n do
7          dp[i][0] ← dp[i−1][0] + CostoEliminacion(S1[i−1])
8      for j ← 1 to m do
9          dp[0][j] ← dp[0][j−1] + CostoInsercion(S2[j−1])
10     for i ← 1 to n do
11         for j ← 1 to m do
               // 1. Costo de Eliminación
12             dp[i][j] ← mín(dp[i][j], dp[i−1][j] + CostoEliminacion(S1[i−1]))
               // 2. Costo de Inserción
13             dp[i][j] ← mín(dp[i][j], dp[i][j−1] + CostoInsercion(S2[j−1]))
               // 3. Costo de Sustitución
14             cost ← CostoSustitucion(S1[i−1], S2[j−1])
15             dp[i][j] ← mín(dp[i][j], dp[i−1][j−1] + cost)
               // 4. Costo de Transposición (si aplica)
16             if i > 1 and j > 1 and S1[i−1] = S2[j−2] and S1[i−2] = S2[j−1] then
17                 dp[i][j] ← mín(dp[i][j], dp[i−2][j−2] + CostoTransposicion(S1[i−2], S1[i−1]))
18     return dp[n][m]
```

Ejemplo de Paso a Paso en Programación Dinámica

Consideremos las cadenas $s1 =$ ‘‘`cat`’’ y $s2 =$ ‘‘`cut`’’. Inicializamos una matriz dp de tamaño 4×4 (ya que ambas cadenas tienen longitud 3, y sumamos 1 para incluir el caso de las subcadenas vacías). Primero, llenamos la primera fila y columna con los costos de insertar y eliminar caracteres, de modo que $dp[i][0] = i$ y $dp[0][j] = j$. A continuación, calculamos cada celda $dp[i][j]$ basándonos en los valores anteriores. Para $dp[1][1]$, como $s1[0] =$ ‘‘`c`’’ y $s2[0] =$ ‘‘`c`’’, copiamos el valor de $dp[0][0]$ sin costo adicional, así que $dp[1][1] = 0$. Luego, en $dp[2][2]$, $s1[1] =$ ‘‘`a`’’ y $s2[1] =$ ‘‘`u`’’ son distintos, por lo que tomamos el mínimo entre sustitución ($dp[1][1] + 1$), inserción ($dp[2][1] + 1$) y eliminación ($dp[1][2] + 1$), resultando en $dp[2][2] = 1$. Continuamos este proceso hasta completar la matriz. Al final, $dp[3][3]$ contiene el valor 1, que es la distancia mínima de edición entre las cadenas.

Análisis de Complejidad Temporal y Espacial

La complejidad temporal es $O(n \times m)$, donde n y m son las longitudes de las cadenas $s1$ y $s2$. Esto se debe a que llenamos una matriz dp de tamaño $(n+1) \times (m+1)$, calculando cada entrada en función de un número constante de operaciones (sustitución, inserción, eliminación y, opcionalmente, transposición). La inclusión de la operación de transposición no cambia la complejidad temporal asintótica, pero agrega una verificación adicional en cada celda, lo que puede aumentar ligeramente el tiempo de ejecución en la práctica. En cuanto a la complejidad espacial, es también $O(n \times m)$ debido al almacenamiento de la matriz dp. Los costos variables de cada operación afectan el valor final de cada celda, pero no modifican la complejidad asintótica, ya que solo influyen en el cálculo de cada operación, sin aumentar el número total de operaciones.

3. Implementaciones

3.1. Sobre el Código

Para ejecutar el programa, primero se debe correr `Ejecutar.py`, este programa corre todos los programas en orden y presenta finalmente los graficos. Esto se realiza con el comando:

```
python .\Ejecutar.py
```

El código genera dos datasets, uno para FB y DP y otro solo para DP, ya que FB no procesa más de 13 caracteres. Si se quieren correr cadenas en especifico se debe correr el programa c++ por separado y seleccionar la opcion e ingresar las cadenas, habiendo corrido antes el codigo de generacion, ya que ahi se encuentran las matrices de costos.

```
python .\generar.py
g++ .\fuerzaDinamica.cpp -o ./fuerzaDinamica
./fuerzaDinamica.exe
```

3.2. Tipos de Cadenas

- **Idéntica**: Las cadenas son iguales; la distancia de edición es cero. Es útil como referencia.
- **Sustitución**: Las cadenas son casi idénticas, pero requieren algunas sustituciones.
- **Inserción**: Una cadena tiene caracteres adicionales que se deben agregar.
- **Eliminación**: Una cadena tiene caracteres adicionales que deben eliminarse.
- **Completamente diferente**: Las cadenas no comparten similitudes, necesitando múltiples operaciones.
- **Repetitiva**: Cadenas con patrones repetitivos.
- **Desplazada**: Una cadena es una versión desplazada de la otra, útil para evaluar la transposición.

3.3. Descripción de los Costos

- **Eliminar caracteres**: `cost_delete.txt`, costo constante de 1.
- **Insertar caracteres**: `cost_insert.txt`, costo de 2 en posiciones pares y 1 en impares.
- **Reemplazar caracteres**: `cost_replace.txt`, costo aleatorio entre 1 y 2.
- **Transponer caracteres**: `cost_transpose.txt`, costo aleatorio entre 1 y 2.

3.4. Finalización

El programa genera tres archivos `.txt` con los resultados de los experimentos y muestra gráficos en pantallas emergentes.

4. Experimentos

Para correr los programas utilizaremos un pc de escritorio con las siguientes especificaciones tecnicas:

- Procesador: Amd Ryzen 5600x 6 core 4.6 ghz
- Ram: 16 GB DDR4 3200 mhz
- Almacenamiento: SSD NMVe
- SO: Windows 10 pro
- Compilador: g++.exe (MinGW.org GCC-6.3.0-1) 6.3.0

4.1. Dataset (casos de prueba)

- $S_1 = \epsilon$ $S_2 = \epsilon$
  Resultados:
  Longitud maxima: 0 caracteres
  Programacion Dinamica:
  Costo: 0
  Tiempo: 0.000 ms
  Fuerza Bruta:
  Costo: 0
  Tiempo: 0.000 ms
  sin costo...

- $S_1 = \epsilon$ $S_2 = abcde$
  Longitud maxima: 5 caracteres
  Programacion Dinamica:
  Costo: 8
  Tiempo: 0.000 ms
  Fuerza Bruta:
  Costo: 8
  Tiempo: 0.000 ms
  insertar: 'a' = 2, 'b' = 1, 'c'= 2, 'd' = 1, 'e' = 2

- $S_1 = abcde$ $S_2 = \epsilon$
  Longitud maxima: 5 caracteres
  Programacion Dinamica:
  Costo: 5
  Tiempo: 0.000 ms

Fuerza Bruta:
Costo: 5
Tiempo: 0.000 ms
costo de insertar cualquier caracter es 1, $1 * 5 = 5$

- $S_1 = aaaaaa$ $S_2 = bbbbbb$
  Longitud maxima: 6 caracteres
  Programacion Dinamica: Costo: 6
  Tiempo: 0.000 ms
  Fuerza Bruta:
  Costo: 6 Tiempo: 0.000 ms
  costo, se genera de manera aleatoria, en este caso cuesta 1 cambiar a por b, por lo cual $1 * 6 = 6$, sale mejor que eliminar e insertar

- $S_1 = abcd$ $S_2 = bacd$
  Longitud maxima: 4 caracteres
  Programacion Dinamica:
  Costo: 1
  Tiempo: 0.000 ms
  Fuerza Bruta:
  Costo: 1
  Tiempo: 0.000 ms
  girar a y b en esta seed tiene costo de 1. imposible obtener un valor menor...

Luego para muchos mas casos de prueba tenemos los entregados en la carpeta, tiene libertad de revisarlos en los txt, los resultados seran presentados en graficos posteriores. los enviados son con los que se elaboro el informe, por lo que se pueden generar otros, pero pueden diferir minimamente en algunos valores de costos por su naturaleza aleatoria.

Al momento de correr el programa la carga del dispositivo estuvo en un 14 % en la cpu y 50 % en ram

4.2. Resultados

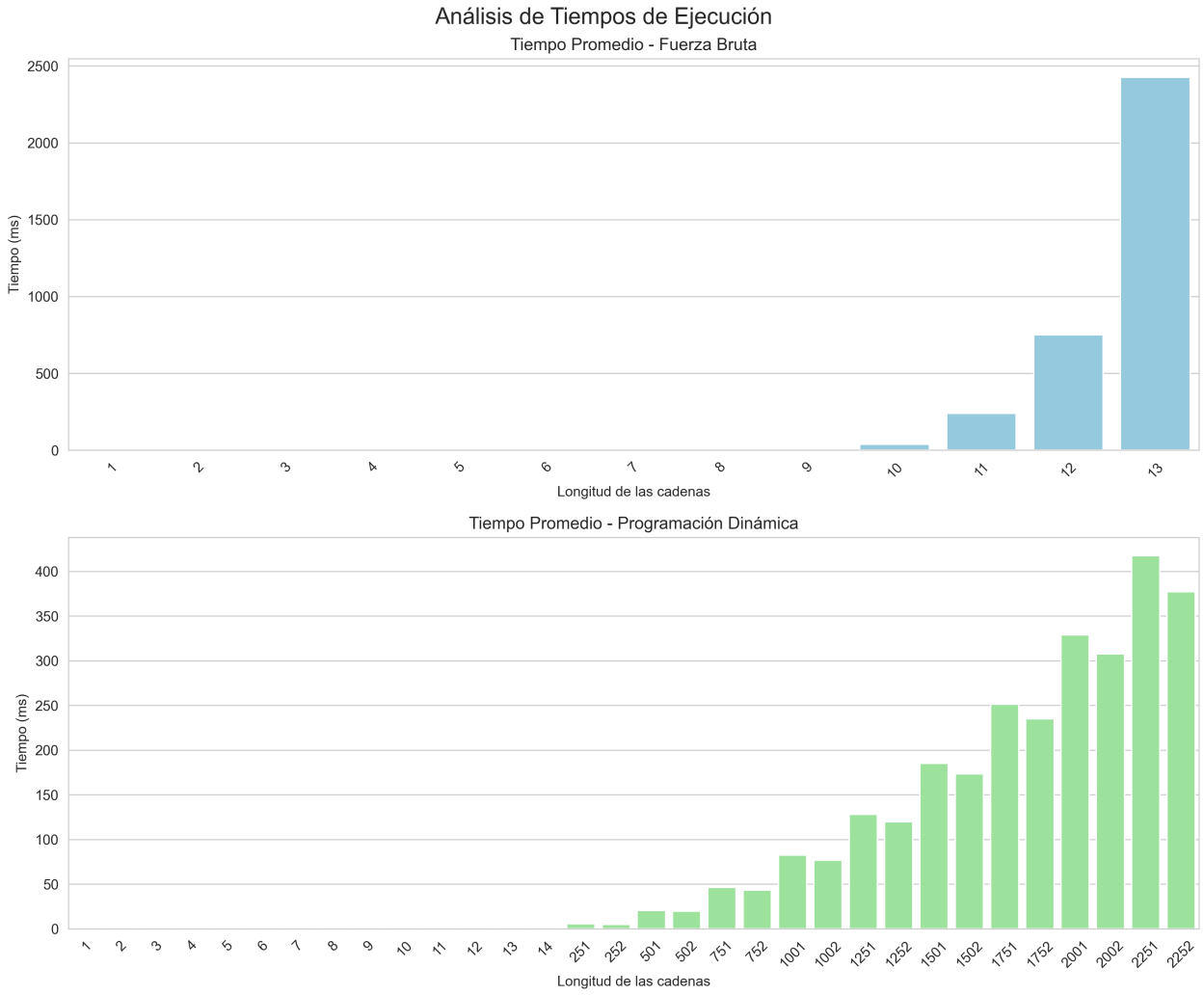

Figura 1: histograma de tiempos vs aumento de caracteres

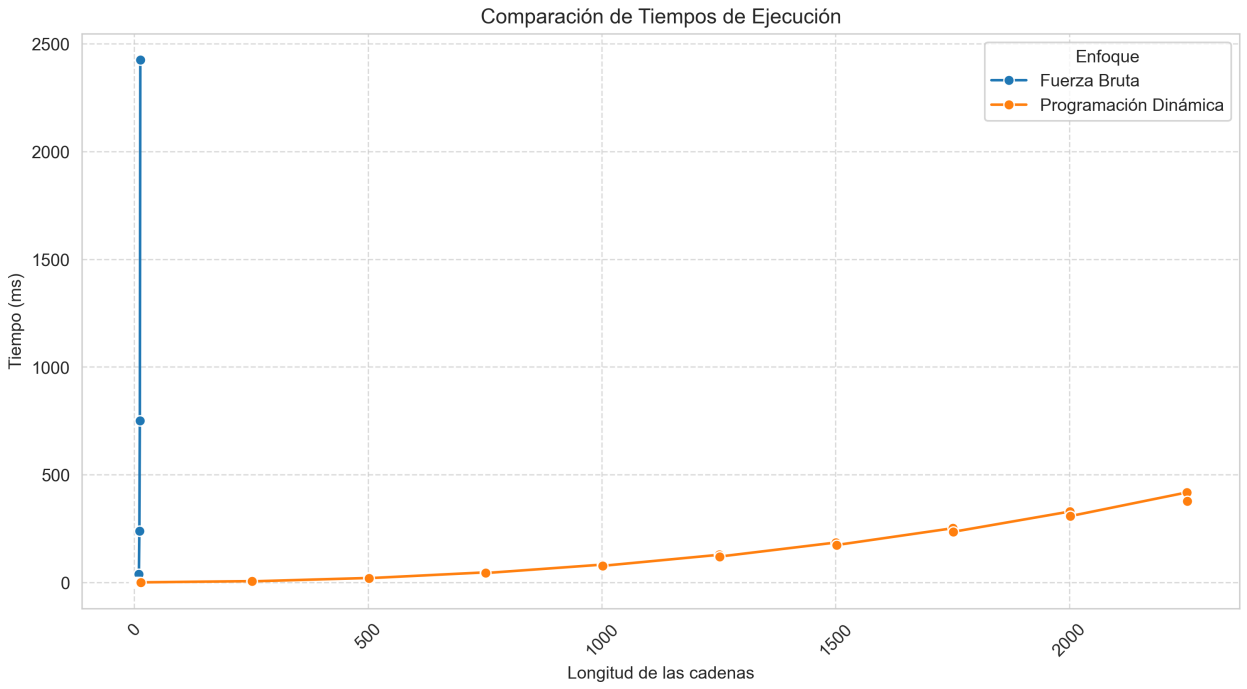

Figura 2: comparacion de tiempos

Puede ver los resultados específicos de la última ejecución del programa en la carpeta entregada o en el link de GitHub provisto (ver Apéndice .1). Los resultados se encuentran en el archivo **resultados.txt** con el siguiente formato:

```
----------------------------------------
Tipo: Cadenas desplazadas
Longitud: 13 caracteres
S1: eimcouefgdljb
S2: beimcouefgdlj
Costo FB: 2
Tiempo FB: 7507.304 ms
Costo DP: 2
Tiempo DP: 0.000 ms
----------------------------------------
```

Son demasiados resultados como para ponerlos todos en este informe, pero los gráficos son un buen resumen de lo que ahí se encuentra. Podemos apreciar, sobre todo en la **Figura 2**, cómo es la diferencia entre ambos enfoques y lo absurdamente superior que es la programación dinámica contra la fuerza bruta. Esto se debe a que la primera posee una complejidad temporal lineal, mientras que la segunda tiene una complejidad exponencial.

5. Conclusiones

Podemos apreciar la abismal diferencia entre los enfoques de programación expuestos en este informe, cómo con un quinto del tiempo somos capaces de analizar y transformar cadenas de grandes dimensiones. Esto demuestra una mejora significativa a la hora de resolver este tipo de problemas, ya que, en este informe, solo nos quedamos hasta los 2500 caracteres.

Gracias a los resultados, podemos encontrar mejoras significativas en el rendimiento de otros problemas que se pueden reducir a este. Como por ejemplo, correctores de ortografía, los cuales podrían detectar que al escribir "tecldo"queremos escribir "teclado", ya que calcularían la palabra con menor distancia de edición en el diccionario según la palabra escrita por nosotros. Lo cual quizás no se condice con los experimentos de este informe, ya que en este tenemos una pequeña aleatoriedad de los costos, la cual fue en realidad puesta ahí sin ningún objetivo en mente. Antes había evaluado otros costos, pero por lo general eran demasiado altos en comparación y salía más económico borrar toda la cadena e insertar los caracteres, lo cual no tiene ninguna gracia. Pero al final, para la mayoría de las aplicaciones que se podrían pensar para este problema, tendremos costos equitativos o puestos con un motivo más que el azar.

En concreto, el enfoque de programación dinámica podría servir para algoritmos de detección de plagio en documentos. Si dejáramos de ver las palabras separadas por un espacio, podríamos calcular la distancia mínima de edición entre dos textos completos y determinar qué tan iguales son.

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

.1. Repositorio del Proyecto

El código fuente, datos y scripts utilizados en este informe están disponibles en el siguiente repositorio de GitHub:

https://github.com/juaquo15/Tarea02-INF221.git

Este repositorio contiene los archivos necesarios para ejecutar los experimentos descritos, así como documentación adicional para facilitar la reproducción de los resultados presentados.