REPORTE TAREA 2 y 3
ALGORITMOS Y COMPLEJIDAD

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

Luis Zegarra Stuardo

15 de noviembre de 2024

23:51

Resumen

Este estudio compara dos algoritmos, Fuerza Bruta y Programación Dinámica, utilizando como referencia el algoritmo de Distancia de Levenshtein, para calcular la distancia mínima de edición entre dos cadenas de texto, con costos variables en operaciones como inserción, eliminación, sustitución y transposición. Ambos algoritmos fueron implementados en C++ y se realizaron pruebas con cadenas de diferentes tamaños. Los resultados muestran que la Programación Dinámica es más eficiente en tiempo de ejecución que la Fuerza Bruta, aunque los datos sobre el uso de memoria fueron inconsistentes debido a problemas en la medición. En conclusión, el algoritmo de Programación Dinámica demuestra ser más efectivo, pero la implementación de la medición de memoria y el registro de operaciones requiere mejoras para futuros estudios.

Índice

1. Introducción **2**

2. Diseño y Análisis de Algoritmos **3**

3. Implementaciones **8**

4. Experimentos **9**

5. Conclusiones **15**

6. Condiciones de entrega **16**

A. Apéndice 1 **17**

1. Introducción

Hoy en día, en la era de la tecnología, la **calidad y eficiencia de los programas computacionales** ha crecido enormemente. Con el avance tecnológico, el incremento en la cantidad de datos y la complejidad de los procesos, es fundamental contar con métodos para consultar, organizar y manejar esta información de manera eficiente. En este contexto, el campo de Análisis y Diseño de Algoritmos en Ciencias de la Computación cobra especial relevancia, ya que permite abordar la resolución de problemas complejos y la creación de programas optimizados que puedan ejecutarse en tiempos razonables y con un consumo de memoria reducido.

> *"... However, Generalized Levenshtein Distance is not suitable for certain applications such as recognizing noisy subsequences and skeletal images since it lacks an appropriate normalization with respect to the lengths of the compared strings."*
>
> — Yujian y Bo, 2007 [5]

Un problema recurrente en este campo es el cálculo de la **distancia mínima de edición**, ampliamente utilizado en aplicaciones como el **procesamiento de lenguaje natural, la recuperación de información, la biología computacional y la inteligencia artificial**. En estos casos, la necesidad de comparar y procesar cadenas de texto de manera precisa y eficiente es clave para obtener resultados de calidad. El cálculo de la distancia mínima de edición, además de ser una tarea común, es esencial en estos ámbitos, ya que permite medir la similitud entre dos secuencias mediante la transformación de una en otra [5, 2].

En este documento se **presentará la implementación de dos algoritmos para calcular la distancia mínima de edición entre dos cadenas con costos variables dependiendo de los caracteres**, aplicando dos enfoques: **fuerza bruta** y **programación dinámica**. Ambos algoritmos incorporan operaciones de **inserción, eliminación, sustitución y transposición** con costos variables, lo cual aumenta la complejidad y utilidad del cálculo, se utilizará la implementacion de distancia de Levenshtein como guía.

El algoritmo de **fuerza bruta** explora todas las posibles transformaciones de manera exhaustiva, y aunque es un enfoque simple, su tiempo de ejecución crece exponencialmente conforme aumenta el tamaño de las cadenas, lo que lo convierte en una solución viable solo para casos de pequeña escala. Por otro lado, el algoritmo de **programación dinámica** optimiza la búsqueda de soluciones almacenando los resultados de subproblemas ya resueltos, lo cual reduce la cantidad de operaciones necesarias para obtener el resultado final. Sin embargo, esta optimización implica un mayor consumo de memoria, dado que es necesario almacenar los subproblemas en una estructura de datos.

Este estudio se realiza para **evaluar la eficiencia de ambos métodos**; por tanto, se analizarán el **tiempo de ejecución** y el **uso de memoria** en diferentes pares de palabras con diversas características. Se documentarán las implementaciones realizadas y se presentarán los pros y contras de ambos algoritmos de manera gráfica. De esta forma, se busca ayudar al lector a comprender las características de cada algoritmo en función de la entrada que reciben, así como la cantidad de operaciones necesarias para obtener una solución.

2. Diseño y Análisis de Algoritmos

Los algoritmos de **fuerza bruta** y **programación dinámica** utilizarán las siguientes funciones para calcular los costos, dependiendo de la letra involucrada. Según el tipo de operación, se consultará el archivo correspondiente. En el caso de **inserción** o **eliminación**, se accederá a archivos que contienen una fila de 26 números, cada uno representando el costo de operación para una letra específica, determinado por su índice. Para las operaciones de **sustitución** o **transposición**, se empleará una matriz simétrica de 26x26, en la cual se registra el costo de cambiar una letra por otra.

Algoritmo 1: Funciones de costos para las operaciones de edición

```
1 Function COSTO_INSERT(b)
2     return costosInsert[b - 'a']
3 Function COSTO_DELETE(a)
4     return costosDelete[a - 'a']
5 Function COSTO_SUB(a, b)
6     return costosReplace[a - 'a'][b - 'a']
7 Function COSTO_TRANSPOSE(a, b)
8     return costosTranspose[a - 'a'][b - 'a']
```

Se utilizó como guía los algoritmos y las explicaciones presentadas en Wikipedia sobre la Distancia de Levenshtein [4], que busca el número mínimo de operaciones requeridas para transformar una cadena de caracteres en otra, y el algoritmo Smith-Waterman [3], una estrategia para realizar alineamiento local de secuencias basado en programación dinámica. También se utilizó la explicación en GeeksforGeeks sobre la distancia de Levenshtein [6] para la confección de los algoritmos necesarios para esta experiencia.

> *"Brute-force algorithm, which is also called the "naïve" is the simplest algorithm that can be used inpattern searching. It is probably the first algorithm we might think of for solving the pattern searching problem. It requires no preprocessing of the pattern or the text"*
>
> — Mohammad, Saleh y Abdeen, 2006 [1]

2.1. Fuerza Bruta

Algoritmo 2: Algoritmo de distancia de edición con fuerza bruta optimizado para registrar solo las operaciones seleccionadas.

```
1  Procedure DISTANCIAEDICIONFUERZABRUTA(S1, S2, i, j, operaciones)
2      if i es igual a la longitud de S1 then
3          costo ← 0
4          for k desde j hasta la longitud de S2 - 1 do
5              Escribir "inserción" + S2[k] en operaciones
6              costo ← costo + costo_insert(S2[k])
7          return costo
8      else if j es igual a la longitud de S2 then
9          costo ← 0
10         for k desde i hasta la longitud de S1 - 1 do
11             Escribir "eliminación" + S1[k] en operaciones
12             costo ← costo + costo_delete(S1[k])
13         return costo
14     costoSustitucion ← costo_sub(S1[i], S2[j]) + DISTANCIAEDICIONFUERZABRUTA(S1, S2, i + 1, j + 1, operaciones)
15     costoInsercion ← costo_insert(S2[j]) + DISTANCIAEDICIONFUERZABRUTA(S1, S2, i, j + 1, operaciones)
16     costoEliminacion ← costo_delete(S1[i]) + DISTANCIAEDICIONFUERZABRUTA(S1, S2, i + 1, j, operaciones)
17     costoTransposicion ← INT_MAX
18     if i + 1 < longitud de S1 y j + 1 < longitud de S2 y S1[i] = S2[j + 1] y S1[i + 1] = S2[j] then
19         costoTransposicion ← costo_transpose(S1[i], S1[i + 1]) + DISTANCIAEDICIONFUERZABRUTA(S1, S2, i + 2, j + 2,
             operaciones)
20     costoMinimo ← mínimo entre {costoSustitucion, costoInsercion, costoEliminacion, costoTransposicion}
21     if costoMinimo es igual a costoSustitucion then
22         Escribir "sustitución" + S1[i] + "->" + S2[j] en operaciones
23     else if costoMinimo es igual a costoInsercion then
24         Escribir "inserción" + S2[j] en operaciones
25     else if costoMinimo es igual a costoEliminacion then
26         Escribir "eliminación" + S1[i] en operaciones
27     else if costoMinimo es igual a costoTransposicion then
28         Escribir "transposición" + S1[i] + S1[i + 1] en operaciones
29     return costoMinimo
```

2.1.1. Ejemplo Paso a Paso

La función `distanciaEdicionFuerzaBruta` calcula la distancia mínima de edición entre dos cadenas, $S1$ y $S2$, usando un enfoque de fuerza bruta. A continuación se describen los pasos clave:

- **Condiciones de Fin:** Si se alcanza el final de $S1$ (índice i), inserta todos los caracteres restantes de $S2$ y suma sus costos. Si se alcanza el final de $S2$ (índice j), elimina los caracteres restantes de $S1$ y acumula el costo de cada eliminación.
- **Cálculo de Costos de Operaciones:**

- **Sustitución:** Calcula el costo de reemplazar $S1[i]$ por $S2[j]$, avanzando ambos índices.
- **Inserción:** Calcula el costo de insertar $S2[j]$ en $S1$, avanzando solo j.
- **Eliminación:** Calcula el costo de eliminar $S1[i]$, avanzando solo i.

- **Costo de Transposición:** Si los caracteres $S1[i]$ y $S1[i+1]$ están invertidos con respecto a $S2[j]$ y $S2[j+1]$, calcula el costo de transponer estos caracteres y avanza dos posiciones en ambas cadenas.
- **Selección y Registro de Operación de Menor Costo:** Una vez calculados los costos, se identifica el mínimo y se registra únicamente la operación correspondiente en un archivo de salida.
- **Retorno del Costo Mínimo:** Devuelve el menor costo calculado, representando el costo total mínimo para transformar $S1$ en $S2$ desde las posiciones actuales.

2.1.2. Análisis de Complejidad

La complejidad temporal del algoritmo de fuerza bruta es exponencial, $O(4^{\text{máx}(m,n)})$, donde m y n son las longitudes de $S1$ y $S2$, respectivamente. La falta de almacenamiento de subproblemas resueltos obliga a recalcular cada transformación posible, resultando en una alta demanda de tiempo de ejecución para cadenas largas. La complejidad espacial es $O(\text{máx}(m,n))$ debido a la profundidad máxima de la pila de recursión.

La inclusión de transposiciones y costos variables aumenta la cantidad de posibles operaciones a evaluar en cada paso, incrementando la carga computacional en comparación con la versión estándar del algoritmo de distancia de edición.

2.2. Programación Dinámica

2.2.1. Descripción de la Solución Recursiva

En programación dinámica, el problema se descompone en subproblemas, almacenando cada resultado dentro de una matriz a memdida que se revisan los strings, para evitar cálculos repetitivos. Esto permite calcular la distancia de edición entre $S1$ y $S2$ de manera más eficiente.

2.2.2. Relación de Recurrencia

La relación de recurrencia es la siguiente:

$$\text{DP}[i][j] = \text{mín} \begin{cases} \text{DP}[i-1][j] + \text{costo\_eliminación} & \\ \text{DP}[i][j-1] + \text{costo\_inserción} & \\ \text{DP}[i-1][j-1] + \text{costo\_sustitución} & \text{si } S1[i] \neq S2[j] \\ \text{DP}[i-2][j-2] + \text{costo\_transposición} & \text{si hay transposición} \end{cases}$$

Los casos base son:

$$\text{DP}[0][j] = j \times \text{costo\_inserción} \quad \text{y} \quad \text{DP}[i][0] = i \times \text{costo\_eliminación}$$

2.2.3. Identificación de Subproblemas

Cada subproblema DP[i][j] representa la distancia mínima de edición para transformar los primeros i caracteres de $S1$ en los primeros j caracteres de $S2$.

Algoritmo 3: Algoritmo de distancia de edición con programación dinámica.

```
1  Procedure DISTANCIAEDICIONPROGDINAMICA(S1, S2, operaciones)
2      m ← longitud de S1
3      n ← longitud de S2
4      Crear matriz dp de dimensiones (m + 1) × (n + 1) inicializada en 0
5      for i desde 1 hasta m do
6          dp[i][0] ← dp[i - 1][0] + costo_delete(S1[i - 1])
7          Escribir "eliminación" + S1[i - 1] en operaciones
8      for j desde 1 hasta n do
9          dp[0][j] ← dp[0][j - 1] + costo_insert(S2[j - 1])
10         Escribir "inserción" + S2[j - 1] en operaciones
11     for i desde 1 hasta m do
12         for j desde 1 hasta n do
13             costoSustitucion ← dp[i - 1][j - 1] + costo_sub(S1[i - 1], S2[j - 1])
14             costoInsercion ← dp[i][j - 1] + costo_insert(S2[j - 1])
15             costoEliminacion ← dp[i - 1][j] + costo_delete(S1[i - 1])
16             dp[i][j] ← mínimo entre {costoSustitucion, costoInsercion, costoEliminacion}
17             if dp[i][j] es costoSustitucion then
18                 Escribir "sustitución" + S1[i - 1] + "->" + S2[j - 1] en operaciones
19             else if dp[i][j] es costoInsercion then
20                 Escribir "inserción" + S2[j - 1] en operaciones
21             else if dp[i][j] es costoEliminacion then
22                 Escribir "eliminación" + S1[i - 1] en operaciones
23             if i > 1 y j > 1 y S1[i - 1] = S2[j - 2] y S1[i - 2] = S2[j - 1] then
24                 costoTransposicion ← dp[i - 2][j - 2] + costo_transpose(S1[i - 2], S1[i - 1])
25                 if dp[i][j] > costoTransposicion then
26                     dp[i][j] ← costoTransposicion
27                     Escribir "transposición" + S1[i - 2] + S1[i - 1] en operaciones
28     return dp[m][n]
```

2.2.4. Ejemplo Paso a Paso

La función `distanciaEdicionProgDinamica` calcula la distancia mínima de edición entre dos cadenas, $S1$ y $S2$, mediante un enfoque de programación dinámica, usando una matriz de costos (`dp`) para almacenar soluciones de subproblemas. A continuación, se presenta un resumen de los pasos clave:

- **Inicialización de Casos Base:**
  - Rellena la primera columna de `dp` para reflejar el costo de eliminar cada carácter en $S1$ cuando $S2$ está vacío.

- Rellena la primera fila de `dp` para reflejar el costo de insertar cada carácter de $S2$ cuando $S1$ está vacío.

- **Cálculo de Costos de Operaciones:**
  - Para cada posición (i, j) en la matriz, calcula:
    - **Sustitución:** El costo de reemplazar $S1[i-1]$ por $S2[j-1]$.
    - **Inserción:** El costo de insertar $S2[j-1]$.
    - **Eliminación:** El costo de eliminar $S1[i-1]$.

- **Registro de Operaciones:**
  - Se escribe en el archivo de operaciones el tipo de operación seleccionada en cada paso (sustitución, inserción, o eliminación) basada en el menor costo.

- **Verificación de Transposición (Opcional):**
  - Si $S1[i-1]$ y $S1[i-2]$ están invertidos respecto a $S2[j-1]$ y $S2[j-2]$, calcula el costo de transponerlos y actualiza `dp[i][j]` si este costo es menor que los otros.
  - Registra la operación de transposición en el archivo de operaciones si es elegida.

- **Resultado:** Al finalizar, `dp[m][n]` contiene el costo mínimo para transformar $S1$ en $S2$.

2.2.5. Análisis de Complejidad

La complejidad temporal de la versión dinámica es $O(m \times n)$, ya que se calcula cada celda de la matriz dp una sola vez. La complejidad espacial también es $O(m \times n)$ debido a la matriz utilizada para almacenar los subproblemas. La inclusión de transposiciones y costos variables agrega un costo adicional en el cálculo de cada celda de dp, pero no cambia la complejidad asintótica.

3. Implementaciones

El programa se puede encontrar en "este enlace", que lo redirigirá a un repositorio en GitHub donde están los códigos para ejecutar la solución al problema. La carpeta que contiene la solución se llama `codigos`, y tiene la siguiente estructura:

- `casos_prueba.txt`: aquí se encuentran los casos de prueba a evaluar. Cada caso debe ingresarse en el formato "palabra1 palabra2". Si se desea especificar una cadena vacía, se representa mediante comillas dobles (""). Las entradas pueden ser palabras o secuencias de letras en minúsculas del abecedario en inglés.
- `costos_delete.txt` y `costos_insert.txt`: archivos que representan los costos de eliminación e inserción respectivamente, para cada letra (cada valor corresponde a una letra desde la "a" hasta la "z").
- `costos_replace.txt` y `costos_transpose.txt`:archivos que contienen una matriz que representa los costos de reemplazo y transposición de una letra por otra; es una matriz de tamaño 26x26 (cada columna y fila corresponden a una letra).
- `costos.cpp`: archivo en C+, genera los archivos **costos**, estableciendo un valor de 1 para todas las operaciones, lo cual permite verificar que los algoritmos creados funcionen correctamente al entregar soluciones exactas para cada caso.
- `random-costos.cpp`: archivo en C++, genera los archivos **costos** con valores aleatorios entre 1 y 10 para los costos de las operaciones. Esta configuración es la válida para cumplir con el propósito de la implementación.
- `graficos.py`: archivo en Python que genera gráficos en función de los resultados registrados en `resultados.txt`.
- `resultados.txt`: archivo que contiene los resultados de cada caso probado. Incluye información sobre la distancia de edición, el tiempo en microsegundos y la memoria utilizada para la implementación de Fuerza Bruta y Programación Dinámica.
- `operaciones.txt`: archivo en el que se registran las operaciones realizadas por los diferentes algoritmos. Su tamaño puede variar considerablemente según la cantidad de casos y la longitud de las palabras probadas.
- `main.cpp`: archivo principal, implementa la solución completa; Debe ejecutarse para obtener todos los resultados. Lee el set de pruebas en `casos_prueba.txt`.
- `Makefile`: archivo Makefile que contiene los comandos necesarios para ejecutar cada uno de los archivos.

4. Experimentos

Para asegurar la reproducibilidad de los resultados, se detallaran a continuación la infraestructura utilizada en la ejecución del experimento, especificando tanto el hardware como el entorno software.

Hardware

- **Procesador:** Apple M2
- **RAM:** 16 GB (con 3.2 GB en uso durante los experimentos)
- **Almacenamiento:** SSD NVMe

Software

- **Sistema Operativo:** macOS 15.0.1
- **Compilador:** g++ (versión por defecto para macOS 16.0.0, ajustado para el soporte ARM64 en M2) y se utilizó un entorno de python con la versión 3.13
- **Entorno de Ejecución:** Terminal WezTerm, Shell ZSH 5.9
- **Entorno Gráfico:** Aqua DE con Quartz Compositor

Esta configuración garantiza que los experimentos sean replicables bajo condiciones similares. Se utilizó el comando `neofetch` para obtener los detalles de la infraestructura, que se incluyen como referencia en la Figura 1.

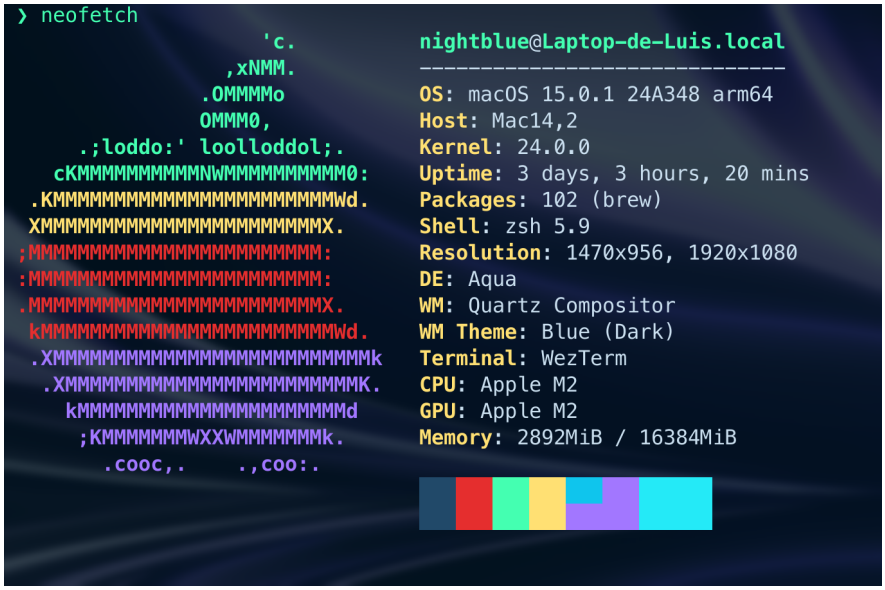

Figura 1: Detalle del sistema usando el comando `neofetch`.

Condiciones de Entrada y Parámetros

Cada experimento fue ejecutado bajo las siguientes condiciones de entrada y parámetros específicos:

- **Cadenas de Entrada:** Se utilizaron cadenas de prueba de diferentes longitudes y combinaciones de caracteres para evaluar el rendimiento de los algoritmos de distancia mínima de edición (alfabeto inglés en letras minúsculas).
- **Parámetros de Algoritmos:** Los costos para inserción, eliminación, sustitución y transposición fueron establecidos de acuerdo a las especificaciones de los algoritmos.
- **Orden de ejecución de archivos:** Primero se deben crear los archivos de costos (en caso de que no estén creados o se quieran cambiar); luego se debe ejecutar el `main.cpp`, este sobreescribirá los archivos de los resultados y operaciones; por último, ejecutar el archivo Python si se quieren obtener las gráficas.

Es importante mecnionar, que esta implementación utiliza la biblioteca `sys/resource.h` de C++, la cual permite la lectura del consumo de memoria en las operaciones, pero solo está disponible en entornos Linux y Unix. En caso de utilizar un entorno diferente, será necesario eliminar esta funcionalidad en el archivo **main.cpp**. Esto implicará realizar modificaciones tales como eliminar la función de lectura de memoria del `main` y omitir la escritura de este dato en el archivo de resultados.

4.1. Dataset (casos de prueba)

La extensión máxima para esta sección es de 2 páginas.

Para los casos de prueba, se decidió no generarlos de manera automática, con el objetivo de mantener control sobre los casos a evaluar y verificar que los algoritmos ejecuten correctamente los pasos necesarios para alcanzar la solución. Los casos de prueba fueron diseñados tomando en cuenta los siguientes escenarios:

- Casos donde las cadenas están vacías: Ejemplos incluyen S1 = "" y S2 = "", S1 = "aaa" y S2 = "", y S1 = "" y S2 = "xyz".
- Casos con caracteres repetidos: Ejemplos incluyen S1 = "aabb" y S2 = "ab", y S1 = "abc" y S2 = "abc".
- Casos donde las transposiciones son necesarias: Ejemplos incluyen S1 = "abcd" y S2 = "abdc", y S1 = "ab" y S2 = "ba".
- Casos variando el largo de las cadenas.

Los casos de prueba están almacenados en el archivo `casos_prueba.txt`. En este archivo se pueden agregar las líneas que se deseen probar, utilizando el formato "`palabra1 palabra2`". Para representar una cadena vacía, se debe emplear el símbolo de comillas dobles (`""`).

4.2. Documentación de Casos de Prueba

A continuación se presentan los casos de prueba utilizados (Figura 4) para evaluar los algoritmos de distancia de edición. En cada uno se especifican las entradas $S1$ y $S2$, las operaciones esperadas, el costo total estimado y la justificación de la salida.

Caso de Prueba	**Entrada** $S1$	**Entrada** $S2$	**Operaciones Esperadas**	**Costo Total Esperado**
1	"abc"	"abc"	Ninguna operación.	0
2	"ab"	"ba"	Transposición de 'a' y 'b'.	Costo transposición
3	"abc"	"acb"	Transposición de 'b' y 'c'.	Costo transposición
4	"abcde"	"abcde"	Ninguna operación.	0
5	"abc"	"a"	Eliminación de 'b', Eliminación de 'c'.	2 * Costo eliminación
6	"abc"	"def"	Sustitución de 'a' → 'd', 'b' → 'e', 'c' → 'f'.	3 * Costo sustitución
7	"abcd"	"abdc"	Transposición de 'c' y 'd'.	Costo transposición
8	"aaa"	""	Eliminación de 'a', 'a', 'a'.	3 * Costo eliminación
9	""	"xyz"	Inserción de 'x', 'y', 'z'.	3 * Costo inserción
10	""	""	Ninguna operación.	0
11	"cuadrado"	"cuaresma"	Sustitución de 'd' → 'r', 'r' → 'e', 'a' → 's', 'd' → 'm', 'o' → 'a'.	5 * Costo sustitución
12	"rodilla"	"paella"	Sustitución de 'r' → 'p', 'o' → 'a', 'd' → 'e'. Eliminación de 'i'	3 * Costo sustitución + 1 * Costo Eliminación
13	"amanda"	"ada"	Eliminación de 'm', 'a', 'n'.	3 * Costo eliminación

Cuadro 1: Algunos casos de prueba para demostrar operaciones y costo esperado.

4.2.1. Explicación de Ejecución

Cada caso ha sido diseñado para cubrir un escenario específico de edición, como inserciones, eliminaciones, sustituciones y transposiciones. A continuación se presenta la justificación por tipo de caso:

- **Igualdad Directa (Casos 1 y 4)**: No requieren cambios, ya que ambas cadenas son idénticas.
- **Transposiciones Necesarias (Casos 2, 3, 7)**: Requieren una transposición para igualar $S1$ y $S2$, minimizando las operaciones de edición.
- **Inserciones y Eliminaciones (Casos 5, 8, 9, 13)**: Estos casos requieren insertar o eliminar caracteres adicionales en $S1$ o $S2$ para hacerlas equivalentes.
- **Sustituciones (Casos 6, 11, 12)**: Exigen reemplazar uno o varios caracteres para obtener la cadena deseada.

Es importante tener en cuenta que esto es solo una predicción de lo que debería ocurrir, pero pueden existir variaciones. Esto se debe a que la implementación está orientada a encontrar la distancia de edición con costos variables que se generan de manera aleatoria. Por lo tanto, la selección de las operaciones dependerá directamente de los valores con los que se generen los archivos de costos.

4.3. Resultados

La extensión máxima para esta sección es de 4 páginas.

4.3.1. Comprobación de algoritmos

Primero se detallará el desarrollo para verificar que los algoritmos están correctamente diseñados, proporcionando así resultados correctos. Para esto, se ha utilizado la página web PLANETCALC, la cual cuenta con un apartado para calcular la Distancia de Levenshtein con costos iguales para todas las operaciones. En esta página solo se entrega el resultado, por lo que se comparará este con el resultado obtenido por los algoritmos de Fuerza Bruta y Programación Dinámica.

Caso de Prueba	**Entrada** $S1$	**Entrada** $S2$	**PLANETCALC**	**Fuerza Bruta**	**Programación Dinámica**
1	"abc"	"abc"	0	0	0
2	"ab"	"ba"	1	1	1
3	"abc"	"acb"	2	1	1
4	"abcde"	"abcde"	0	0	0
5	"abc"	"a"	2	2	2
6	"abc"	"def"	3	3	3
7	"abcd"	"abdc"	1	1	1
8	"aaa"	""	3	3	3
9	""	"xyz"	3	3	3
10	""	""	0	0	0
11	"cuadrado"	"cuaresma"	5	5	5
12	"rodilla"	"paella"	4	4	4
13	"amanda"	"ada"	3	3	3

Cuadro 2: Comprobación de la efectividad de los algoritmos con algunos casos de prueba

Este paso es solo para verificar que los algoritmos están correctamente diseñados o, al menos, que arrojan el resultado correcto. Para esto, se utiliza el archivo `costos.cpp`, el cual genera los archivos `costos` con valor 1 para todas las operaciones, de la siguiente manera:

- Contenido de los archivos `cost_delete.txt` y `cost_insert.txt` so una fila de 26 unos, cada uno representa una letra.
- El contenido de los archivos `cost_replace.txt` y `cost_transpose.txt` son matrices simétrics de 26x26 que representan el valor de cambiar una letra por otra con un 1.

Se puede observar en el Cuadro 2 que los valores obtenidos son bastante similares, pero esto puede resultar engañoso, ya que la página PLANETCALC utiliza el cálculo original de la Distancia de Levenshtein, donde solo se consideran las operaciones de inserción, eliminación y reemplazo. Es por esto que estas medidas solo se tomaron como referencia para verificar los algoritmos.

4.3.2. Soluciones encontradas

Para generar los costos variables, los cuales son requisitos dentro del problema a resolver, se creó el archivo `random-costos.cpp`, el cual genera los costos asignando un valor aleatorio de 1 a 10 para cada operación. Los costos utilizados pueden verse en el repositorio de Github, aquí están los enlaces:

- delete
- insert
- replace
- transpose

Al ejecutar el `main.cpp` con estos costos, los resultados obtenidos se pueden evidenciar en el archivo `resultados.txt`, en el cual se encuentra que:

Caso de Prueba	Entrada $S1$	Entrada $S2$	Distancia (FB)	Distancia (PD)
1	"abc"	"abc"	0	0
2	"ab"	"ba"	3	3
3	"abc"	"acb"	2	2
4	"abcde"	"abcde"	0	0
5	"abc"	"a"	4	4
6	"abc"	"def"	14	14
7	"abcd"	"abdc"	2	2
8	"aaa"	""	6	6
9	""	"xyz"	15	15
10	""	""	0	0
11	"cuadrado"	"cuaresma"	15	15
12	"rodilla"	"paella"	20	20
13	"amanda"	"ada"	13	13

Cuadro 3: Comprobación

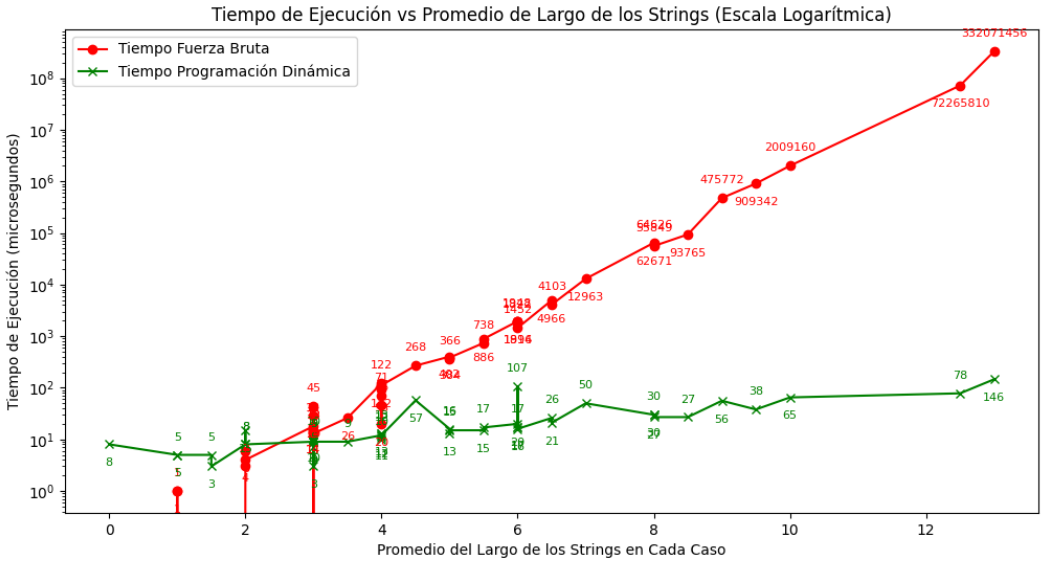

Figura 2: Gráfica obtenida con los resultados de las pruebas.

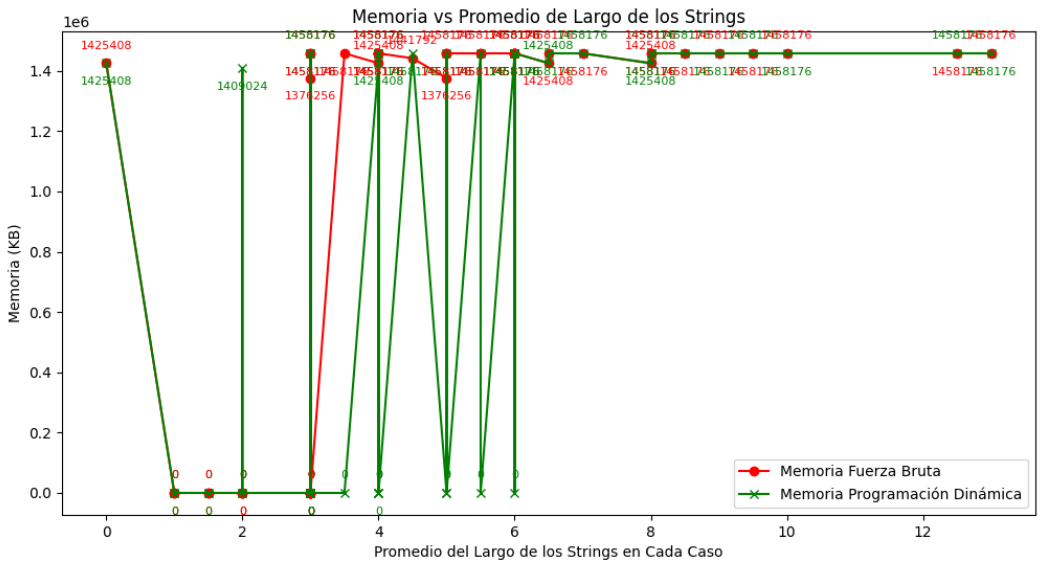

Figura 3: Gráfica obtenida con resultados de las pruebas.

La Figura 2 se genera a partir de los resultados de los casos de prueba registrados en el archivo resultados.txt. En ella, se observa el comportamiento de los algoritmos en función del tiempo que tardan según el promecio de longitud de las cadenas de entrada, además de cómo influye la estructura de dichas cadenas. A medida que aumenta el promedio de la suma de las longitudes de las cadenas, el tiempo de ejecución incrementa de manera más pronunciada en el algoritmo de Fuerza Bruta, mientras que el algoritmo de Programación Dinámica muestra un cambio más controlado, corroborando el análisis aisntótico realizado.

También se intentó registrar la memoria utilizada por cada función para resolver el problema, pero la implementación no fue correcta, ya que registraba todas las ejecuciones de los algoritmos, incluidas aquellas que no afectaban el resultado final. Esto se realizó mediante un hilo que se ejecutaba en paralelo con el archivo `main.cpp`, consultando la memoria utilizada cada 10 milisegundos y almacenándola en una variable global. Durante estas pruebas, no se ejecutó ningún otro programa para evitar interferencias en los resultados. La memoria utilizada se presenta en la Figura 3. Sin embargo, debido a que no se obtuvo un registro correcto de las operaciones realizadas por los algoritmos, el archivo `operaciones.txt` registraba una gran cantidad de líneas de operaciones, lo que resultaba en un archivo extremadamente extenso y pesado para ser cargado en GitHub. Por esta razón, el archivo disponible en el repositorio corresponde a una ejecución con menos casos de prueba, lo cual permite evidenciar de manera más clara los datos que se generan.

5. Conclusiones

Los resultados obtenidos a partir de las simulaciones demuestran de manera clara la superioridad del algoritmo de Programación Dinámica frente al de Fuerza Bruta en cuanto al tiempo de ejecución. A medida que el tamaño de las cadenas de entrada aumenta, el algoritmo de Programación Dinámica muestra una eficiencia notablemente mayor en términos de tiempo, lo que valida su eficacia para resolver el problema planteado.

En cuanto al análisis de la memoria, los resultados no son concluyentes debido a la deficiencia en la implementación de la medición de memoria. Las variaciones observadas en el uso de memoria pueden atribuirse tanto a limitaciones del compilador, que optimiza el uso de memoria en ciertos casos, como a posibles fallos en la implementación de los registros dentro del archivo `operaciones.txt`, lo cual limitó la revición sobre los pasos que siguiieron cada algoritmo para encontrar la solución, ya que, se registraron gran parte de las reviciones realizadas y no solos las que aportaban en el cálculo de la solución final.

A pesar de estas dificultades, el análisis realizado confirma que ambos algoritmos son adecuados para resolver el problema de calcular la menor distancia de edición con costos variables, aunque la obtención de los pasos intermedios de cada algoritmo no fue posible debido a los problemas con la implementación del manejo de registros. Estos resultados contribuyen a una mejor comprensión de las ventajas y limitaciones de los algoritmos en el contexto de la distancia de edición con costos variables, destacando la importancia de una implementación precisa para obtener métricas confiables.

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

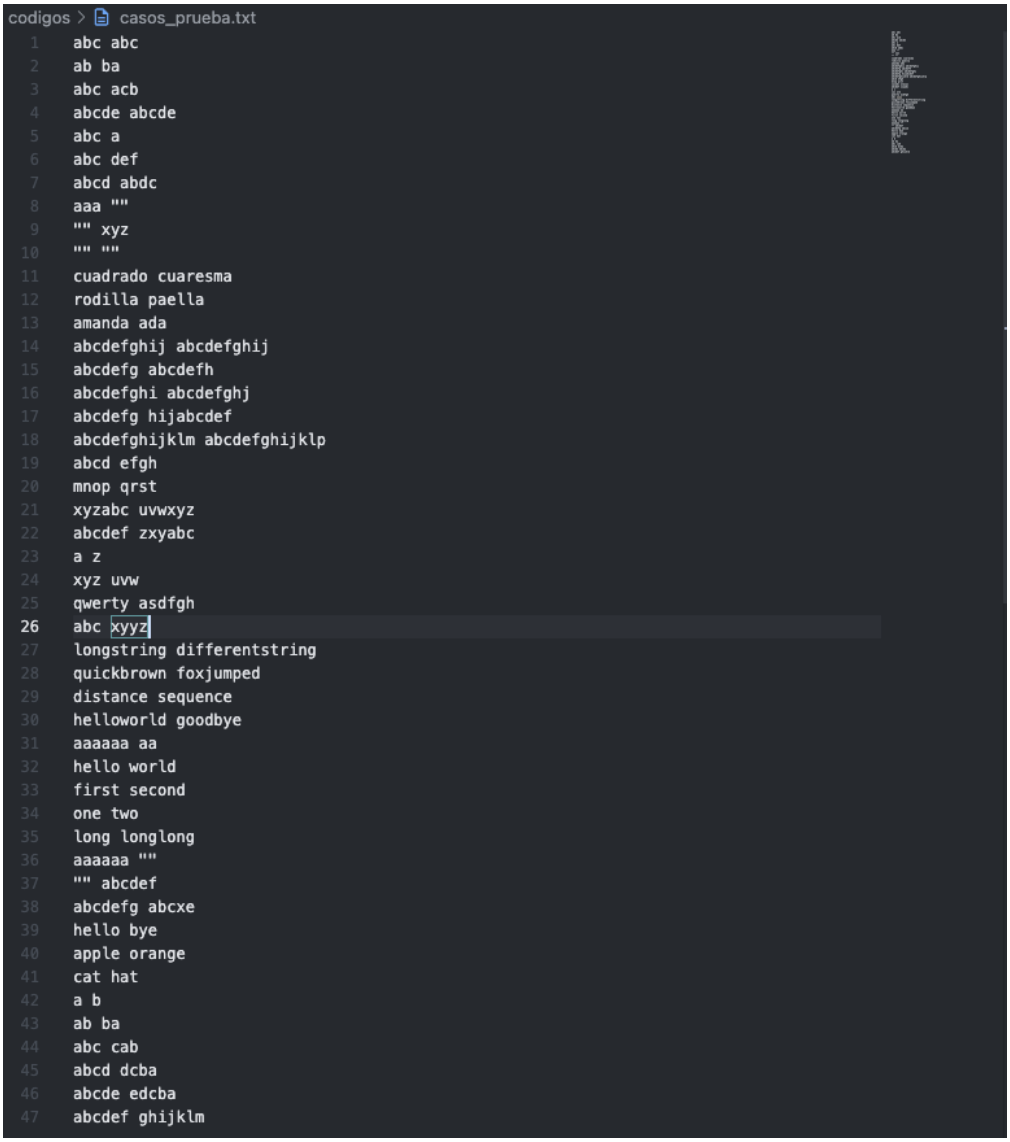

Figura 4: Casos de prueba

Referencias

[1] Ababneh Mohammad, Oqeili Saleh y Rawan A Abdeen. «Occurrences algorithm for string searching based on brute-force algorithm». En: *Journal of Computer Science* 2.1 (2006), págs. 82-85.

[2] Edgar Moyotl-Hernández. «Método de corrección ortográfica basado en la frecuencia de las letras.» En: *Res. Comput. Sci.* 124 (2016), págs. 145-156.

[3] Wikipedia. *Algoritmo Smith-Waterman — Wikipedia, La enciclopedia libre*. [Internet; descargado 25-junio-2024]. 2024. URL: %5Curl%7Bhttps://es.wikipedia.org/w/index.php?title=Algoritmo_Smith-Waterman&oldid=160948078%7D.

[4] Wikipedia. *Distancia de Levenshtein — Wikipedia, La enciclopedia libre*. [Internet; descargado 28-enero-2024]. 2024. URL: %5Curl%7Bhttps://es.wikipedia.org/w/index.php?title=Distancia_de_Levenshtein&oldid=157799255%7D.

[5] Li Yujian y Liu Bo. «A normalized Levenshtein distance metric». En: *IEEE transactions on pattern analysis and machine intelligence* 29.6 (2007), págs. 1091-1095.

[6] zaidkhan15. *Introduction to Levenshtein Distance*. Accessed: 2024-01-31. 2024. URL: %5Curl%7Bhttps://www.geeksforgeeks.org/introduction-to-levenshtein-distance/%7D.