

REPORTE TAREA 2 y 3

ALGORITMOS Y COMPLEJIDAD

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

Benjamín Camus Valdés

17 de noviembre de 2024

21:16

Resumen

Este informe explora el cálculo de la distancia mínima de edición entre dos cadenas, un problema relevante en áreas como bioinformática y procesamiento de texto. Se analizan dos enfoques: el algoritmo de fuerza bruta, con complejidad exponencial, y con programación dinámica, que optimiza el cálculo mediante subproblemas, logrando una complejidad cuadrática. Se considera adicionalmente la operación de transposición entre dos caracteres adyacentes y costos variables en las operaciones. Los objetivos incluyen evaluar el rendimiento de ambos algoritmos bajo distintas condiciones, como patrones repetitivos, cadenas con gaps, transposiciones, cadenas aleatorias, entre otros. El estudio se llevó a cabo en un entorno controlado utilizando hardware moderno para medir tiempos de ejecución y consumo de memoria. Los resultados muestran que la programación dinámica es significativamente más eficiente, mientras que la fuerza bruta es limitada debido a su alto costo computacional. Este trabajo resalta la importancia de elegir algoritmos adecuados según el problema y los recursos disponibles.

Índice

1. Introducción	2
2. Diseño y Análisis de Algoritmos	4
3. Implementaciones	9
4. Experimentos	10
5. Conclusiones	16
6. Condiciones de entrega	17
A. Apéndice 1	18

1. Introducción

La optimización de algoritmos para el cálculo de similitud entre cadenas es un área fundamental en Ciencias de la Computación, especialmente en el contexto del análisis y diseño de algoritmos. Un problema central en este campo es el cálculo de la distancia mínima de edición, también conocido como **distancia de Levenshtein** *Levenshtein distance* — *Wikipedia, The Free Encyclopedia* [8], que determina el número mínimo de operaciones necesarias para transformar una cadena en otra. Su extensión, la distancia de **Damerau-Levenshtein** *Damerau-Levenshtein distance* — *Wikipedia, The Free Encyclopedia* [4], incorpora la operación de transposición entre caracteres adyacentes, permitiendo modelar con mayor precisión errores comunes de escritura y transcripción. El estudio de estos algoritmos ha sido extensamente abordado en la literatura académica, destacando dos enfoques principales: la aproximación por fuerza bruta y la programación dinámica. Mientras que el enfoque por fuerza bruta garantiza encontrar la solución óptima mediante una exploración exhaustiva con una complejidad temporal de $O(4^{n+m})$, donde $m = |S1|$ y $n = |S2|$, la programación dinámica ofrece una optimización significativa al reutilizar cálculos previos, reduciendo la complejidad a $O(n \times m)$. Esta diferencia sustancial en complejidad plantea interrogantes sobre los escenarios prácticos donde cada enfoque podría resultar más apropiado. La importancia de este problema trasciende el ámbito puramente teórico. En el campo de la bioinformática, por ejemplo, la comparación eficiente de secuencias genéticas de ADN o ARN requiere algoritmos optimizados para procesar grandes volúmenes de datos *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology* [2]. Sin embargo, existe una falta en la comprensión práctica de cómo estos enfoques se comportan bajo diferentes condiciones y patrones de entrada. Este informe aborda un análisis comparativo experimental de ambos enfoques algorítmicos, centrándonos específicamente en la implementación de la distancia de Damerau-Levenshtein, o distancia de edición extendida, en la cual se pueden implementar con costos variables. El objetivo principal es evaluar y contrastar el rendimiento de estos algoritmos bajo diversas condiciones controladas, incluyendo:

- Casos donde las cadenas están vacías.
- Casos con caracteres repetidos.
- Casos con patrones alternados.
- Casos con transposiciones.
- Casos con palíndromos y desorden.
- Casos con caracteres aleatorios.
- Casos con gaps de tamaño variable.

Este análisis experimental complementa los conceptos teóricos estudiados en la asignatura de Análisis y Diseño de Algoritmos, proporcionando evidencia empírica sobre las ventajas y limitaciones de cada enfoque en escenarios prácticos. La investigación se centra particularmente en identificar los puntos de

inflexión donde un enfoque supera al otro, considerando tanto el tiempo de ejecución como el consumo de memoria. Los resultados de este estudio servirán como referencia para comprender mejor las implicaciones prácticas de las decisiones algorítmicas en problemas de similitud de cadenas. Este conocimiento es fundamental para desarrolladores y estudiantes que necesiten implementar soluciones eficientes en sistemas reales, permitiéndoles tomar decisiones informadas sobre qué algoritmo utilizar según sus restricciones específicas de tiempo y memoria. Adicionalmente, este trabajo contribuye a la comprensión de cómo los principios teóricos de complejidad algorítmica se traducen en rendimiento real, considerando factores prácticos como la arquitectura del hardware y consumo de memoria. La estructura del informe sigue un enfoque sistemático, comenzando con una revisión de los fundamentos teóricos, seguida por la descripción detallada de la metodología experimental, la presentación de resultados y su análisis, y finalmente las conclusiones.

2. Diseño y Análisis de Algoritmos

2.1. Fuerza Bruta

El algoritmo de fuerza bruta está diseñado para calcular la distancia mínima de edición extendida entre dos cadenas $S1$ y $S2$ explora exhaustivamente todas las posibles secuencias de operaciones que transforman $S1$ en $S2$. Este enfoque recursivo considera, en cada posición, todas las operaciones permitidas: sustitución, inserción, eliminación y transposición, aplicando los costos variables especificados para cada operación y carácter.

La función principal, `calcular_distancia_fuerza_bruta`, toma como entrada las cadenas $S1$ y $S2$, y los índices actuales i y j . La recursión termina cuando uno de los índices alcanza cero, caso en el cual se calcula el costo de insertar o eliminar los caracteres restantes. En cada paso recursivo, se calculan los costos de las cuatro operaciones posibles (solo si se puede realizar dicha operación) y se retorna el mínimo de estos costos.

Este algoritmo garantiza encontrar la secuencia de operaciones con el costo mínimo, pero debido a que considera todas las combinaciones posibles, su complejidad es exponencial.

Algoritmo 1: Pseudocódigo del algoritmo de cálculo de la distancia de edicion utilizando fuerza bruta, donde i representa la posición en $S1$, y j la posición en $S2$

```

1 Procedure CALCULAR_DISTANCIA_FUERZA_BRUTA( $S1, S2, i, j$ )
2   if  $i = 0$  then
3     return  $j \times \text{COSTO\_INS}(S2[j - 1])$ 
4   else if  $j = 0$  then
5     return  $i \times \text{COSTO\_DEL}(S1[i - 1])$ 
6    $\text{costo\_sustitucion} \leftarrow \text{CALCULAR\_DISTANCIA\_FUERZA\_BRUTA}(S1, S2, i - 1, j - 1) + \text{COSTO\_SUB}(S1[i - 1], S2[j - 1])$ 
7    $\text{costo\_insercion} \leftarrow \text{CALCULAR\_DISTANCIA\_FUERZA\_BRUTA}(S1, S2, i, j - 1) + \text{COSTO\_INS}(S2[j - 1])$ 
8    $\text{costo\_eliminacion} \leftarrow \text{CALCULAR\_DISTANCIA\_FUERZA\_BRUTA}(S1, S2, i - 1, j) + \text{COSTO\_DEL}(S1[i - 1])$ 
9    $\text{costo\_transposicion} \leftarrow \infty$  (INT_MAX)
10  if  $i > 1$  and  $j > 1$  and  $S1[i - 1] = S2[j - 2]$  and  $S1[i - 2] = S2[j - 1]$  then
11     $\text{costo\_transposicion} \leftarrow \text{CALCULAR\_DISTANCIA\_FUERZA\_BRUTA}(S1, S2, i - 2, j - 2) + \text{COSTO\_TRANS}(S1[i - 2], S1[i - 1])$ 
12  return  $\min\{\text{costo\_sustitucion}, \text{costo\_insercion}, \text{costo\_eliminacion}, \text{costo\_transposicion}\}$ 
13 Procedure CALCULAR_DISTANCIA_FUERZA_BRUTA( $S1, S2$ )
14   return  $\text{CALCULAR\_DISTANCIA\_FUERZA\_BRUTA}(S1, S2, \text{length}(S1), \text{length}(S2))$ 

```

Análisis de Complejidad

El algoritmo de fuerza bruta tiene una complejidad temporal exponencial, específicamente $O(4^{n+m})$, donde $m = |S1|$ y $n = |S2|$. Esto se debe a que en cada paso, el algoritmo puede explorar hasta cuatro opciones: sustitución, inserción, eliminación y transposición.

La complejidad espacial es $O(n + m)$, correspondiente a la profundidad máxima de la recursión y al almacenamiento de las operaciones en la pila de llamadas.

Impacto de las Transposiciones y Costos Variables

La inclusión de transposiciones aumenta el número de posibilidades a considerar en cada paso recursivo, incrementando el factor base en la complejidad temporal. Los costos variables también impactan el

proceso de decisión, ya que el algoritmo debe calcular y comparar costos específicos para cada operación y par de caracteres, lo que añade sobrecarga computacional.

Aunque las transposiciones y los costos variables incrementan el tiempo de cómputo debido a cálculos adicionales y mayor ramificación, no alteran la naturaleza exponencial del algoritmo. Esto hace que el enfoque de fuerza bruta sea ineficiente para cadenas de longitud moderada o grande, y refuerza la necesidad de utilizar técnicas más eficientes para problemas de mayor escala.

Para un ejemplo específico, podemos considerar las cadenas $S1 = abc$ y $S2 = acb$. Si establecemos el costo de transposición en 1, encontramos que la distancia mínima de edición entre estas dos cadenas es 1, debido a que solo se requiere una transposición para transformar $S1$ en $S2$. Para una explicación completa y detallada de este cálculo, incluyendo los costos de sustitución, inserción, eliminación y transposición en cada paso, consulte el Apéndice [A.1](#).

2.2. Programación Dinámica

2.2.1. Descripción de la solución recursiva

La distancia mínima de edición entre dos cadenas se puede calcular de forma recursiva evaluando todas las posibles operaciones que transforman un prefijo de la primera cadena en un prefijo de la segunda cadena. En cada paso, se consideran las siguientes operaciones:

Sustitución: reemplazar un carácter de la primera cadena por otro de la segunda cadena.

Inserción: insertar un carácter en la primera cadena para que coincida con un carácter de la segunda cadena.

Eliminación: eliminar un carácter de la primera cadena.

Transposición: intercambiar dos caracteres adyacentes en la primera cadena.

La solución recursiva compara los caracteres actuales de ambas cadenas. Si los caracteres son iguales, se avanza al siguiente par de caracteres sin costo adicional. Si son diferentes, se calcula el costo mínimo considerando todas las operaciones posibles (sustitución, inserción, eliminación y transposición si aplica) y se elige la operación con el costo mínimo. Este proceso continúa hasta que se hayan procesado todos los caracteres de ambas cadenas.

2.2.2. Relación de recurrencia

Para describir la relación de recurrencia llamaremos a $D(i, j)$ el costo mínimo (o distancia mínima) para transformar el prefijo de longitud i de la cadena $S1$ en el prefijo de longitud j de la cadena $S2$, i y j son los índices de los strings $S1$ y $S2$ respectivamente, que inicialmente comienzan siendo la longitud de las cadenas al realizar la primera llamada y luego van disminuyendo a medida que se realizan las llamadas recursivas.

Casos base:

1. Ambos Strings vacíos, cuando $i = 0$ y $j = 0$:

Cuando ambos strings están vacíos, no se necesita ninguna operación, por lo que $D(i, j) = 0$.

2. Primera cadena vacía, cuando $i = 0$ y $j > 0$:

Transformar una cadena vacía en un prefijo de la segunda cadena requiere solo inserciones, así que $D(0, j) = D(0, j - 1) + \text{costo_ins}(S2[j - 1])$.

3. Segunda cadena vacía cuando $i > 0$ y $j = 0$:

Transformar un prefijo de la primera cadena en una cadena vacía requiere solo eliminaciones, por lo que $D(i, 0) = D(i - 1, 0) + \text{costo_del}(S1[i - 1])$.

4. Cuando los caracteres son iguales $S1[i - 1] == S2[j - 1]$ (no necesario): Este caso base es cuando ambos caracteres son iguales por lo que $D(i, j) = D(i - 1, j - 1)$, Sin embargo este caso no es necesario incluirlo en la relación de recurrencia debido a que se puede considerar simplemente como un caso particular de sustitución con costo cero. Si los caracteres son iguales, la función de costo de sustitución (costo_sub) debería devolver automáticamente cero, y la expresión general de la recurrencia lo manejará sin necesidad de un caso separado. Pero si por alguna razón se quisiera hacer que cambiar 2 letras iguales, con costo distinto de 0, esto se tendría que ver en la matriz de costos.

La relación de recurrencia se define de la siguiente manera:

$$D(i, j) = \begin{cases} 0, & \text{si } i = 0 \text{ y } j = 0 \\ D(i, j - 1) + \text{costo_ins}(S2[j - 1]), & \text{si } i = 0 \text{ y } j > 0 \\ D(i - 1, j) + \text{costo_del}(S1[i - 1]), & \text{si } i > 0 \text{ y } j = 0 \\ \min \begin{cases} D(i - 1, j) + \text{costo_del}(S1[i - 1]), & \text{eliminación} \quad i, j > 0 \\ D(i, j - 1) + \text{costo_ins}(S2[j - 1]), & \text{inserción} \quad i, j > 0 \\ D(i - 1, j - 1) + \text{costo_sub}(S1[i - 1], S2[j - 1]), & \text{sustitución} \quad i, j > 0 \\ D(i - 2, j - 2) + \text{costo_trans}(S1[i - 2], S1[i - 1]), & \text{transposición} \quad i > 1, j > 1; \quad S1[i - 1] = S2[j - 1] \text{ y } S1[i - 2] = S2[j - 2] \end{cases} \end{cases}$$

2.2.3. Identificación de subproblemas

El problema de la distancia de edición entre dos cadenas se descompone en subproblemas, donde cada subproblema calcula la distancia de edición para transformar un prefijo de la primera cadena $S1$ en un prefijo de la segunda cadena $S2$. Para definir los subproblemas de manera efectiva, necesitamos identificar cada posible prefijo de las cadenas involucradas.

Dado un prefijo de longitud i de $S1$ y un prefijo de longitud j de $S2$, el subproblema $D(i, j)$ representa el costo mínimo necesario para transformar estos prefijos en cada paso posible (inserción, eliminación, sustitución, o transposición si aplica).

Cada subproblema depende de los valores de subproblemas más pequeños que ya se han resuelto previamente:

- $D(i - 1, j)$: Eliminar un carácter del prefijo de $S1$ para convertirlo en el prefijo de $S2$.
- $D(i, j - 1)$: Insertar un carácter en el prefijo de $S1$ para hacerlo coincidir con el prefijo de $S2$.

- $D(i-1, j-1)$: Sustituir un carácter en $S1$ por un carácter en $S2$ si son diferentes, o no hacer nada si son iguales.
- $D(i-2, j-2)$: Transponer dos caracteres adyacentes en $S1$ si se permite la transposición y si los caracteres coinciden en orden transpuesto.

Estos subproblemas se resuelven de manera **repetitiva** para distintos valores de i y j , utilizando los resultados de subproblemas previos para construir la solución óptima de manera acumulativa. Por lo tanto, cada subproblema comparte resultados parciales con otros subproblemas, y estos resultados pueden ser reutilizados para optimizar el cálculo de la distancia de edición.

La naturaleza de estos subproblemas hace que la programación dinámica sea especialmente útil, ya que permite almacenar los resultados intermedios en una tabla o matriz, evitando así cálculos redundantes.

2.2.4. Estructura de datos y orden de cálculo

Para calcular la distancia mínima de edición entre dos cadenas de forma eficiente, utilizamos una estructura de datos de tipo tabla (o matriz) bidimensional, donde cada posición $D(i, j)$ almacena el costo mínimo de transformar el prefijo de longitud i de la cadena $S1$ en el prefijo de longitud j de la cadena $S2$.

Estructura de datos La estructura utilizada es una matriz D de dimensiones $(m+1) \times (n+1)$, donde m es la longitud de $S1$ y n es la longitud de $S2$. Los elementos de esta matriz $D(i, j)$ representan los subproblemas, y cada entrada almacena el costo mínimo para transformar los primeros i caracteres de $S1$ en los primeros j caracteres de $S2$.

- La primera fila de la matriz, $D(0, j)$, representa los costos de insertar caracteres sucesivos para construir el prefijo de $S2$ cuando $S1$ está vacío.
- La primera columna, $D(i, 0)$, representa los costos de eliminar caracteres sucesivos para transformar el prefijo de $S1$ en una cadena vacía.
- Cada entrada $D(i, j)$ se calcula considerando las operaciones posibles (inserción, eliminación, sustitución y transposición) y tomando el mínimo costo entre ellas.

Orden de cálculo La matriz D se llena de manera iterativa. Empezamos con los casos base:

- Inicializamos $D(0, 0) = 0$ porque transformar una cadena vacía en otra vacía no tiene costo.
- La primera fila $D(0, j)$ se llena con los costos de insertar caracteres de $S2$ en una cadena vacía.
- La primera columna $D(i, 0)$ se llena con los costos de eliminar caracteres de $S1$ hasta convertirlo en una cadena vacía.

A continuación, el cálculo de cada $D(i, j)$ se realiza de izquierda a derecha y de arriba hacia abajo en la matriz. En cada posición, consideramos:

- El valor de $D(i-1, j) + \text{costo_del}(S1[i-1])$ para la operación de eliminación.
- El valor de $D(i, j-1) + \text{costo_ins}(S2[j-1])$ para la operación de inserción.
- El valor de $D(i-1, j-1) + \text{costo_sub}(S1[i-1], S2[j-1])$ para la operación de sustitución.
- Opcionalmente, si es aplicable, el valor $D(i-2, j-2) + \text{costo_trans}(S1[i-2], S1[i-1])$ para la operación de transposición.

La matriz D se llena hasta alcanzar la posición $D(m, n)$, que contiene el costo mínimo de transformar la cadena completa $S1$ en $S2$. La complejidad temporal del algoritmo es $O(m \times n)$ y la complejidad espacial también es $O(m \times n)$, lo que resulta en una solución eficiente en comparación con el enfoque de fuerza bruta.

2.2.5. Algoritmo utilizando programación dinámica

Algoritmo 2: Cálculo de la distancia de edición con programación dinámica

```

1 Procedure CALCULAR_DISTANCIA_DP( $S1, S2$ )
2    $m \leftarrow$  longitud de  $S1$ 
3    $n \leftarrow$  longitud de  $S2$ 
4   //Crear matriz  $D$  de dimensiones  $(m+1) \times (n+1)$ 
5    $D[m+1][n+1]$ 
6    $D[0][0] \leftarrow 0$ 
7   for  $i \leftarrow 1$  to  $m$  do
8      $D[i][0] \leftarrow D[i-1][0] + \text{COSTO\_DEL}(S1[i-1])$ 
9   for  $j \leftarrow 1$  to  $n$  do
10     $D[0][j] \leftarrow D[0][j-1] + \text{COSTO\_INS}(S2[j-1])$ 
11  for  $i \leftarrow 1$  to  $m$  do
12    for  $j \leftarrow 1$  to  $n$  do
13       $\text{costoSustitucion} \leftarrow D[i-1][j-1] + \text{COSTO\_SUB}(S1[i-1], S2[j-1])$ 
14       $\text{costoInsercion} \leftarrow D[i][j-1] + \text{COSTO\_INS}(S2[j-1])$ 
15       $\text{costoEliminacion} \leftarrow D[i-1][j] + \text{COSTO\_DEL}(S1[i-1])$ 
16       $D[i][j] \leftarrow \text{mín}(\text{costoSustitucion}, \text{costoInsercion}, \text{costoEliminacion})$ 
17      if  $i > 1$  and  $j > 1$  and  $S1[i-1] = S2[j-2]$  and  $S1[i-2] = S2[j-1]$  then
18         $\text{costoTransposicion} \leftarrow D[i-2][j-2] + \text{COSTO\_TRANS}(S1[i-2], S1[i-1])$ 
19         $D[i][j] \leftarrow \text{mín}(D[i][j], \text{costoTransposicion})$ 
20  return  $D[m][n]$ 

```

Para ilustrar cómo funciona el algoritmo de programación dinámica en el cálculo de la distancia de edición, considere el **Ejemplo Paso a Paso** en el Apéndice A.2. Este ejemplo detalla cómo se rellena la matriz de costos para transformar las cadenas $S1 = "abc"$ y $S2 = "acb"$, aplicando los costos de sustitución, inserción, eliminación y transposición. La solución óptima se encuentra en la última celda de la matriz, donde se determina que la distancia mínima de edición es 1, correspondiente a una transposición.

3. Implementaciones

A continuación se presenta una breve descripción de los archivos de las implementaciones, para información mas detallada consultar el README del repositorio [3].

distancia_edicion_DP.cpp

Código principal para calcular la distancia de edición extendida entre 2 cadenas, con costos de operaciones variables, utilizando programación dinámica. La idea principal de este código está basado en las implementaciones mostradas en *Distancia de Levenshtein — Wikipedia, La enciclopedia libre* [6] y en *Levenshtein distance — Wikipedia, The Free Encyclopedia* [8], también la información de *Edit distance — Wikipedia, The Free Encyclopedia* [7]. Luego para implementar el caso de la operación adicional de la transposición, se utilizaron las ideas de *Distancia de Damerau-Levenshtein — Wikipedia, La enciclopedia libre* [5] y de *Damerau-Levenshtein distance — Wikipedia, The Free Encyclopedia* [4].

distancia_edicion_BE.cpp

Este código implementa el algoritmo de distancia de edición extendida pero utilizando el enfoque de fuerza Bruta, la función para este calculo es **calcular_distancia_fuerza_bruta()** la cual llama a la función recursiva **distancia_fuerza_bruta()**. La idea principal de este código está basada en el **Naive recursive C++ program** de *Edit Distance | Dynamic Programming* [1] y relaciones de recurrencia mostradas en los artículos de Wikipedia anteriores para la Distancia de Damerau-Levenshtein.

distancia_edicion_dp_memory.cpp

Código para calcular la distancia de edición extendida entre 2 cadenas utilizando programación dinámica, a diferencia del código anterior, este es utilizado para medir la memoria y no guarda las operaciones realizadas para realizar el calculo de la distancia de edición, esto debido a que esas operaciones utilizan memoria extra.

distancia_edicion_bf_memory.cpp

Este código implementa el algoritmo de distancia de edición extendida utilizando el enfoque de fuerza Bruta, pero no guarda las operaciones realizadas en el calculo, este código es utilizado para medir el uso de la memoria sin considerar operaciones adicionales.

/datasets/gen_datasets.cpp

Este código se encarga de generar todos los datasets, genera los casos probados y guarda los pares de strings para cada caso correspondiente en archivos .txt dentro de la misma carpeta.

medidor_memoria_DP.cpp

Se encarga de automatizar el proceso de ejecutar el archivo **distancia_edicion_dp_memory.cpp** para medir la memoria utilizada por el programa utilizando la herramienta **Valgrind**, según los casos de los datasets generados. Exporta los datos en la carpeta **mediciones_memoria_dp**.

medidor_memoria_BE.cpp

Realiza lo mismo que el código anterior pero para el código **distancia_edicion_BF_memory.cpp** que utiliza el enfoque de fuerza bruta. Exporta los datos en la carpeta **mediciones_memoria_bf**.

4. Experimentos

Infraestructura Utilizada:

Información del hardware en el que se realizaron las pruebas:

Procesador: AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz

Tarjeta gráfica: NVIDIA GeForce GTX 1650 Ti

RAM instalada: 16,0 GB (15,4 GB utilizable)

Almacenamiento: Micron SSD 512GB 2210 NVMe PCIe Gen3 x4 MTFDHBA512QFD

Información del entorno de Software

Tipo de sistema Sistema operativo de 64 bits, procesador x64

Sistema operativo: Windows 11, pruebas realizadas y programadas en WSL Ubuntu 22.04.4 LTS

Compilador: g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0, gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0

Se utilizó la librería `#include <bits/stdc++.h>` en las implementaciones.

`#include <chrono>` - Para medir tiempos y gestionar intervalos de tiempo.

`#include <sys/stat.h>` - Para crear directorios en sistemas UNIX.

`#include <sys/types.h>` - Para definir tipos de datos usados en el manejo de archivos y directorios.

Valgrind - Para realizar las mediciones del consumo de memoria de los programas

4.1. Dataset (casos de prueba)

1. **Casos donde las cadenas están vacías:** Para este caso se considera que la cadena S1 está vacía mientras que S2 tiene letras generadas de manera aleatoria según las 26 letras del abecedario inglés. Ejemplo: S1="", S2="dyzviyks" y para las mediciones se va aumentando la longitud de de S2.
2. **Casos con caracteres repetidos:** En este caso con caracteres repetidos consideramos que S2 va a ser la cadena que tiene algunos caracteres repetidos y las repeticiones de los caracteres crecen proporcionalmente al tamaño de la cadena, y S1 son los mismos caracteres que S2 pero están desordenados, por lo que se quiere transformar S1 en S2. Ejemplos: S1="zrtrzcwrresp", S2="scrrrrtpwzz"; S1="pvpspkmspyyy", S2="mpkssyypppv".
3. **Casos con patrones alternados:** En este caso, S2 está formada por patrones repetitivos, como "xyz" o "abc" (para las pruebas se utilizó "xyz"), repetidos a lo largo de la longitud de la cadena. La cadena S1 es una versión desordenada de S2, por lo que el objetivo es transformar S1 en S2. Este tipo de caso ayuda a evaluar el algoritmo con cadenas estructuradas y patrones cíclicos. Ejemplos: S1="xzxyzxyzy", S2="xyzxyzzyzx". Podría verse también como un caso particular de caracteres repetidos.
4. **Casos con transposiciones** En este caso, S2 es una cadena en la que los caracteres están ordenados, mientras que S1 se genera aplicando transposiciones adyacentes (intercambio de pares de caracteres consecutivos) en S2. Este tipo de prueba evalúa la capacidad del algoritmo para identificar

y corregir transposiciones en cadenas casi ordenadas. Ejemplos: S1=“vfcizkohn”, S2=“fvickzhos”; S1=“uyeplolnldt”, S2=“yupeolnldt”.

5. **Casos con palíndromos y desorden:** Aquí, S2 es una cadena palíndroma, mientras que S1 es una versión desordenada de S2. Esto prueba el manejo de cadenas simétricas y desordenadas. Ejemplos: S1=“toxloxtl”, S2=“lxottoxl”; S1=“nknrmtrmt”, S2=“mnrtktrnm”.
6. **Casos con caracteres aleatorios:** En este caso, tanto S1 como S2 son cadenas de caracteres generados aleatoriamente sin ningún patrón específico. Este tipo de prueba permite evaluar el algoritmo en condiciones de aleatoriedad. Ejemplos: S1=“bvnrpkykxoh”, S2=“ukngnrwzerhx”; S1=“qhinnmuohjiyt”, S2=“gxgfnncacinvq”.
7. **Casos con gaps de tamaño variable:** En estos casos, S2 representa una cadena original, mientras que S1 es una versión de S2 en la que se han eliminado caracteres en posiciones aleatorias, generando gaps (huecos) de diferentes tamaños (en las pruebas se realizaron cadenas con 3 gaps, los gaps varían de tamaño entre 1 y 3, aunque ambos valores se pueden ajustar en el código de generación de datasets, **gen_datasets.cpp**). Este tipo de prueba evalúa la capacidad del algoritmo para identificar patrones similares entre una cadena completa y sus versiones incompletas o fragmentadas, una función esencial en aplicaciones de alineamiento de secuencias biológicas.

En el ámbito biológico, los gaps son importantes porque permiten representar mutaciones complejas, como inserciones o eliminaciones de segmentos de ADN o ARN, que pueden variar considerablemente en tamaño. Este tipo de mutación ocurre en procesos como duplicaciones génicas, inserciones de elementos transponibles (“jumping genes”) y otros mecanismos evolutivos que afectan segmentos enteros. Al incluir gaps en el alineamiento, el algoritmo no solo detecta sustituciones puntuales, sino que también captura eventos de mutación a mayor escala, simulando así la historia evolutiva de las secuencias. Esto permite inferir relaciones genéticas profundas y entender las trayectorias de mutación en genes o proteínas *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology* [2]. Por ejemplo: S1 = “ytqirmfmrxo”, S2 = “ybitq-sixdrmfmrxo”; donde los gaps S1=“y__tq__i__rmfmrxo”.

Para generar los casos de prueba se utilizó el código **gen_datasets.cpp**, en donde para una primera prueba (prueba 1) la longitud máxima para una cadena (variable longitudMaxima=20) se estableció en 20, y se fue incrementando de a 1 carácter o letra (variable incremento=1), por lo que con esto se generaron 20 pares de cadenas para cada uno de los 7 casos descritos anteriormente. Sin embargo, debido al alto tiempo de ejecución del algoritmo de fuerza bruta, se descartaron casos superiores, dejando a cada cadena con una longitud máxima de unos 13 caracteres aproximadamente (aprox. debido al caso de los gaps). Mientras que para una segunda prueba (prueba 2) con el objetivo de analizar principalmente el algoritmo que utiliza programación dinámica, se estableció longitudMaxima=5000, e incremento=250, dando un total de 20 pares de cadenas para cada caso, que van incrementando en 250 caracteres hasta llegar a una cadena de 5000. En la prueba 2 se utilizaron numGaps = 6 y tamanoMaxGap = 4.

4.2. Resultados

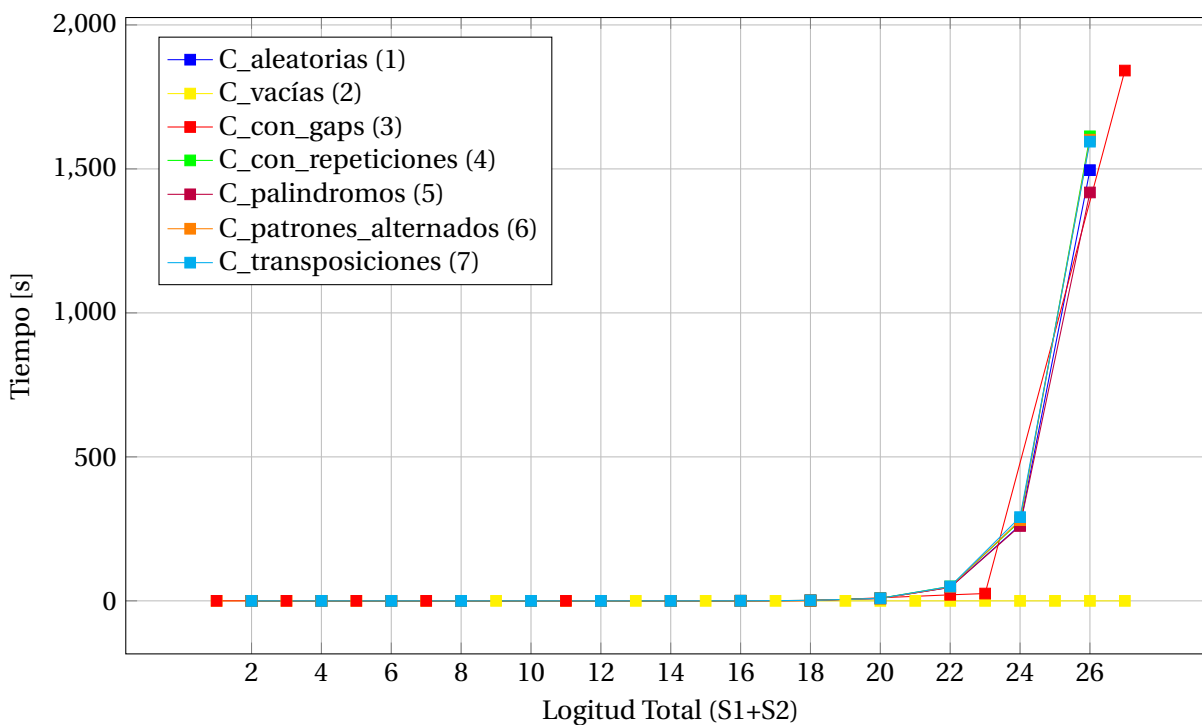


Figura 1: Tiempos de Ejecución Algoritmo Fuerza Bruta (Prueba 1)

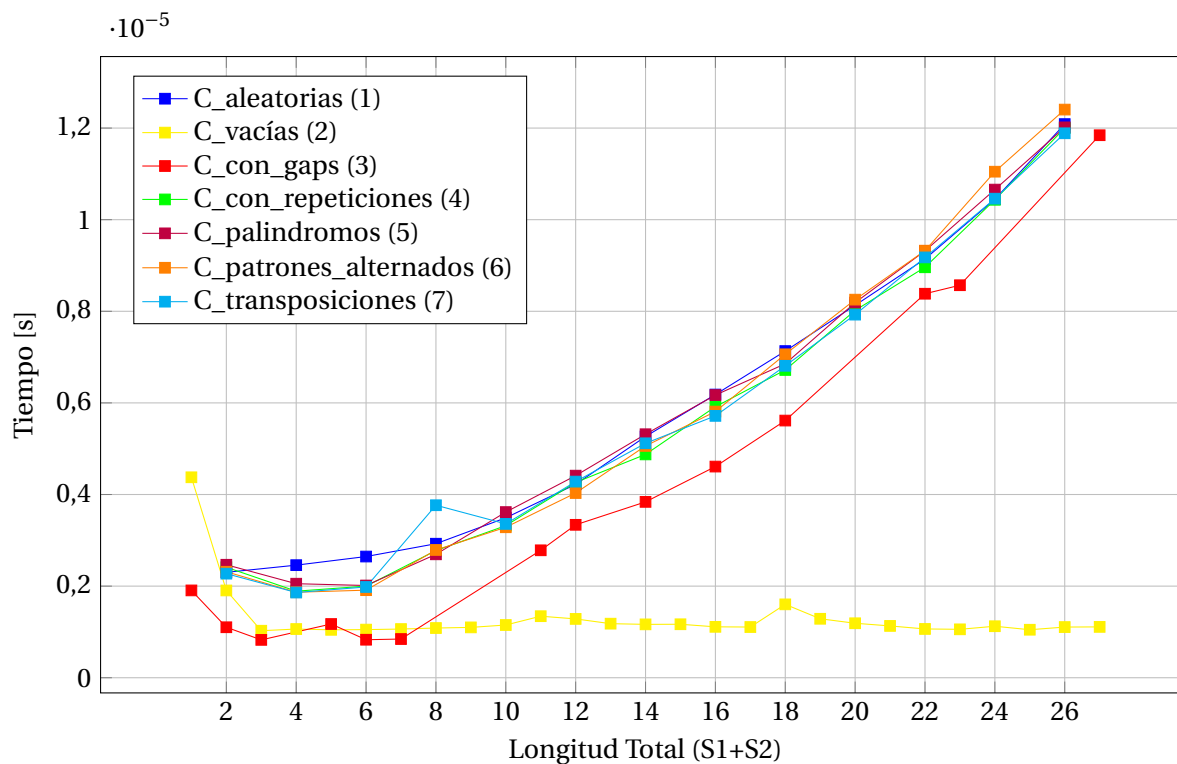


Figura 2: Tiempos de Ejecución Algoritmo Programación Dinámica (Prueba 1)

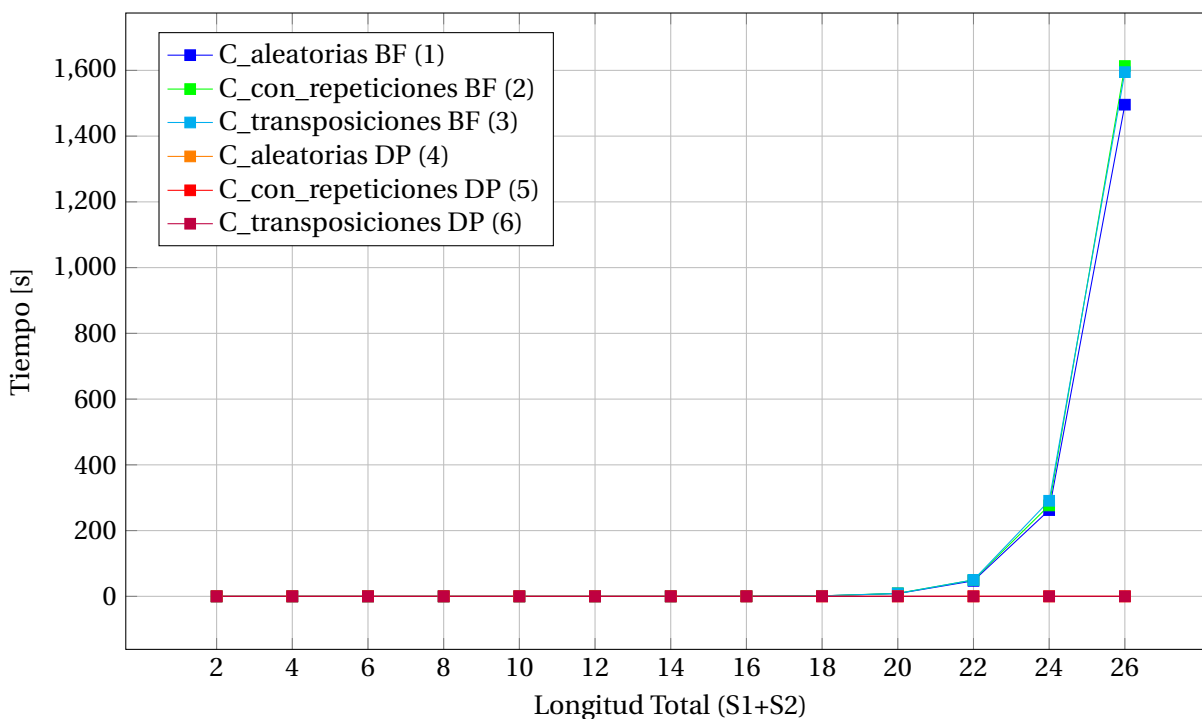


Figura 3: Comparación Tiempos de ejecución Algoritmo DP vs BF (Prueba 1)

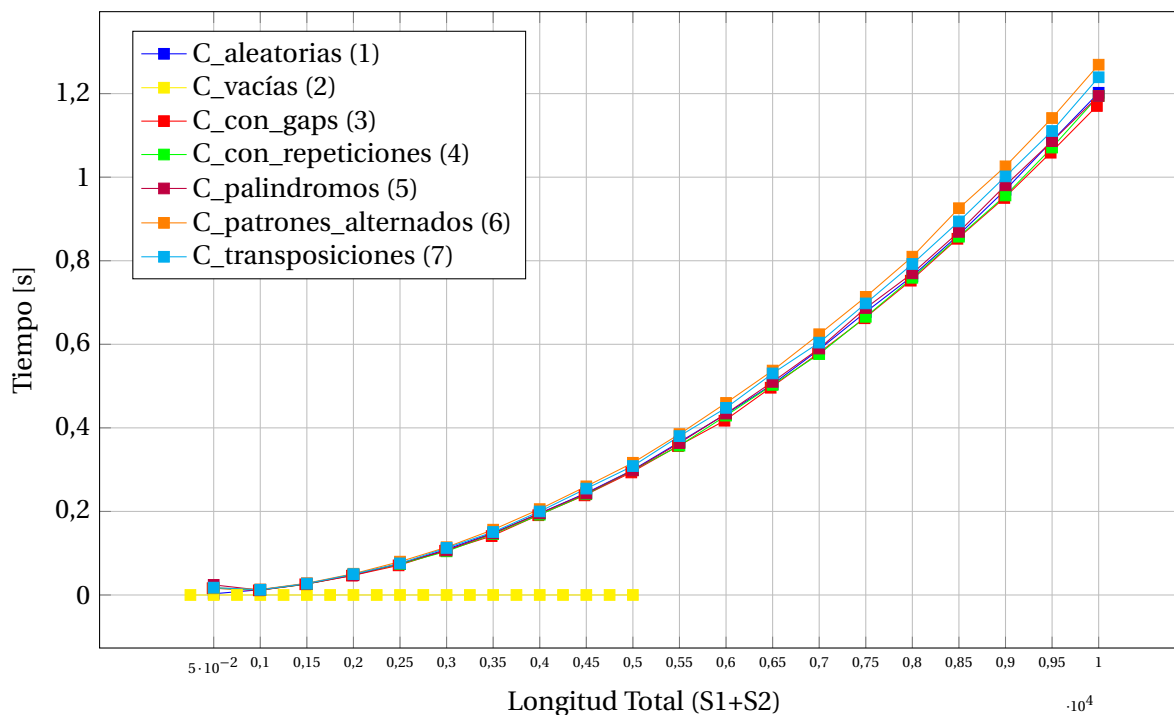


Figura 4: Tiempos de Ejecución Algoritmo Programación Dinámica, longitud máxima por palabra: 5000 letras (Prueba 2)

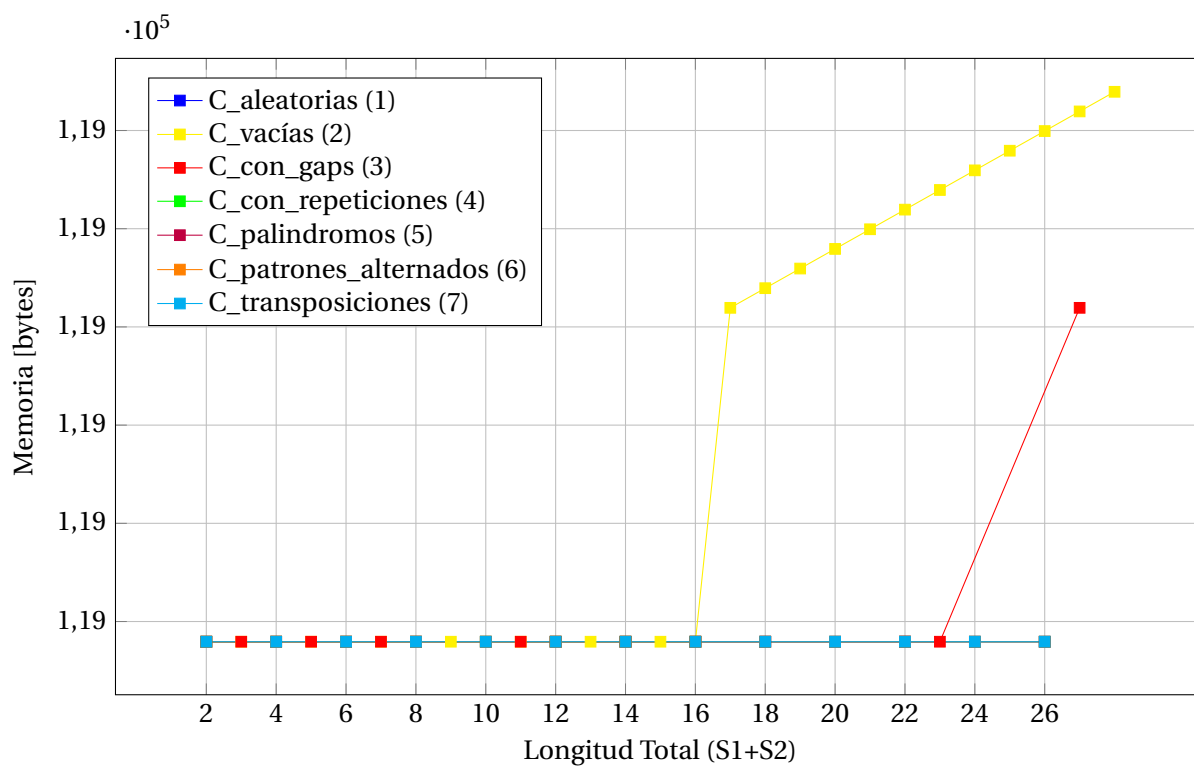


Figura 5: Memoria Utilizada Algoritmo de fuerza bruta (Prueba 1)

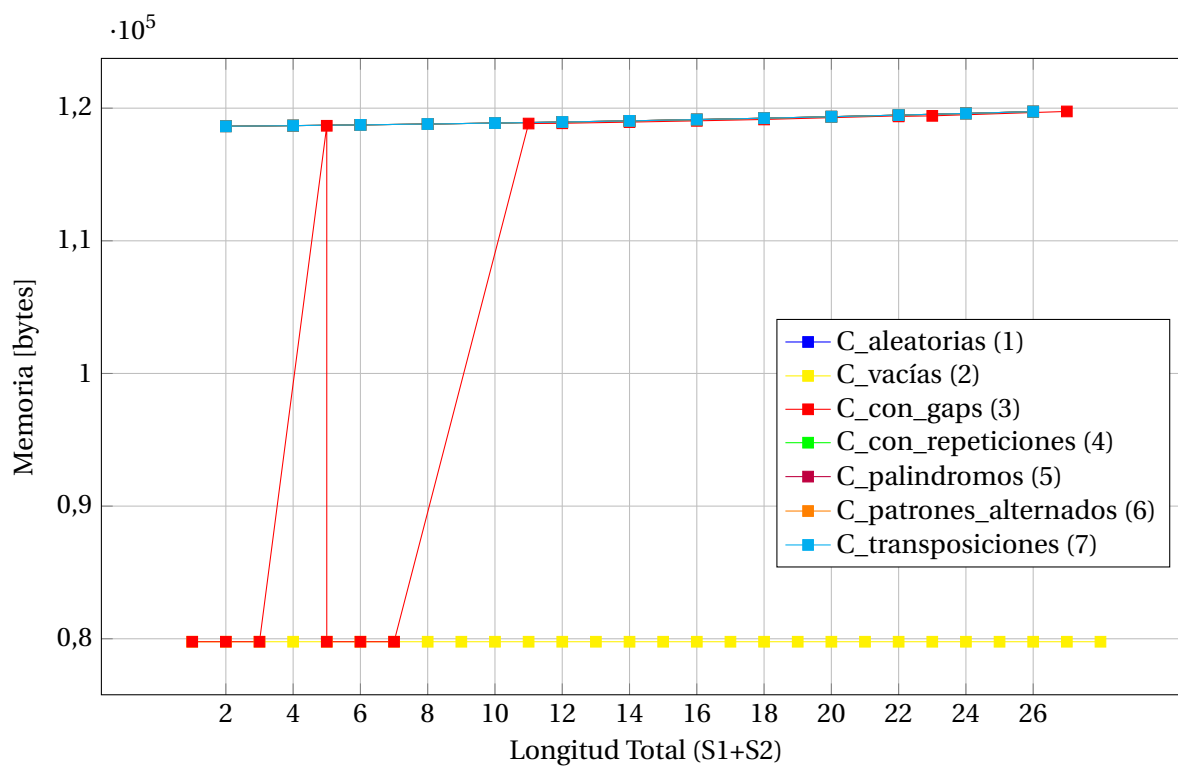


Figura 6: Memoria Utilizada Algoritmo de Programación Dinámica (Prueba 1)

Para obtener los resultados de la mediciones de tiempo, se utilizaron los códigos **distancia_edicion_BF.cpp** y **distancia_edicion_DP.cpp** con la variable `usar_salida_estandar = false` (cambiar a `true` para escribir 2 cadenas por la terminal y probar de manera individual) para leer los archivos .txt con los pares de cadenas para cada caso generado en los datasets. Cada vez que se ejecuta uno de estos códigos, se crea una carpeta llamada “ejecucion_n”, en donde se guardan todos los resultados obtenidos de las mediciones realizadas y sus respectivas secuencias de operaciones de la ejecución número “n”, en un archivo de resultados .txt según el caso de prueba, al final de cada uno estos archivos se guarda una tupla con 2 valores (longitud S1+S2, tiempo ejecución[s]). Estas ejecuciones se guardan en las carpetas **carpeta_pruebas_bf** y **carpeta_pruebas_dp**, luego hay un código llamado **calcular_promedios.cpp** dentro de cada una de estas carpetas que se encarga de calcular el promedio de los tiempos para cada ejecución realizada, exportando el resultado en el archivo `promedios.txt` en donde se pueden encontrar las tuplas con los valores promediados para cada caso, estos valores fueron los que se utilizaron en los gráficos. Para las mediciones de memoria están los archivos **medidor_memoria_BF.cpp** y **medidor_memoria_DP.cpp** que se encargan de procesar cada par de cadenas para cada caso generado en los datasets, y luego realizar las mediciones de memoria utilizando **Valgrind**, cabe aclarar que estos códigos ejecutan los algoritmos de **distancia_edicion_BF_memory.cpp** y **distancia_edicion_dp_memory.cpp** respectivamente, códigos los cuales no calculan la secuencia de las operaciones ya que estarían midiendo memoria adicional. Los resultados obtenidos de las mediciones se guardan en las carpetas `mediciones_memoria_bf` y `mediciones_memoria_dp` donde se puede encontrar una carpeta con las ejecuciones y sus archivos de resultados con la tupla de los valores utilizados en los gráficos. Al comparar los resultados obtenidos en la Figura 1, 2 y 3, podemos ver que el algoritmo que utiliza DP es mucho mejor, mientras que el algoritmo con el enfoque de fuerza bruta es bastante peor y su crecimiento es exponencial, por ejemplo en el caso de caracteres repetidos con un tamaño total de 26 caracteres se demora más de 1500[s], mientras que el algoritmo con DP se demora menos de 2[s]. En general los casos analizados se comportan bastante similar, menos para el caso con gaps y de cadenas vacías, el de gaps debido principalmente a que la suma total de sus caracteres es menor o mayor según el caso, a pesar de esto es el que mejor rendimiento tiene al menos en el algoritmo de DP (Figura 2). Mientras que para el caso de cadenas vacías su tiempo de ejecución es menor en la mayoría de los casos debido a que tiene menor cantidad de caracteres. Debido a que el crecimiento del algoritmo que utiliza DP no es muy claro, se realizó la prueba 2 (Figura 4), donde se puede ver de mejor manera que tiene aproximadamente un comportamiento cuadrático ($O(m*n)$). Para las mediciones de memoria (Figuras 5 y 6), el comportamiento es bastante peculiar, esto debido principalmente a como C++ asigna memoria a las variables, en donde para strings tan pequeños predefine un espacio de memoria fijo hasta alcanzar una cierta cantidad de caracteres (hasta 16), luego empieza a asignar mas memoria a medida que crece el strings, ver prueba realizada en A.3. Por esta razón se ve aproximadamente lineal a partir de un punto en el caso de cadenas vacías (lo que coincide con la complejidad espacial $O(m+n)$), debido a que solo aumenta una cadena. Sin embargo se realizó una medición de memoria para la prueba 2 para el algoritmo DP, en donde se obtuvo un análisis cuadrático, para más detalles se puede consultar A.4.

5. Conclusiones

El análisis realizado demuestra que la programación dinámica es una solución eficiente para el cálculo de la distancia mínima de edición, con una complejidad temporal de $O(m \times n)$ y espacial de $O(m \times n)$, en comparación con el enfoque de fuerza bruta, cuya complejidad temporal es exponencial $O(4^{n+m})$. Los resultados obtenidos validan que la programación dinámica no solo es más escalable, sino también más adecuada para aplicaciones prácticas, incluso cuando se introducen variaciones como transposiciones. Los códigos realizados con ambos enfoques entregan los mismos resultados para los pares de cadena en los casos probados, obteniendo las mismas secuencias de operaciones. por lo tanto son correctos en su ejecución.

Los gráficos generados a partir de las mediciones de tiempo y memoria confirmaron las complejidades teóricas de los algoritmos analizados. El tiempo de ejecución del algoritmo de programación dinámica mostró un comportamiento cuadrático, mientras que el crecimiento exponencial del algoritmo de fuerza bruta fue evidente, limitando su viabilidad a cadenas de tamaño reducido. En cuanto a las mediciones de memoria, se observó que para cadenas pequeñas (hasta 15 caracteres) la asignación se mantuvo constante debido a la gestión interna de C++ para *strings*, complicando la observación de patrones claros. Sin embargo, para cadenas más grandes en el algoritmo de programación dinámica los gráficos reflejaron un crecimiento cuadrático, alineado con las predicciones teóricas para el algoritmo de programación dinámica el cual nos permite escalar el tamaño de los strings para hacer mejores análisis. Para las mediciones de memoria del algoritmo de fuerza bruta se observó un comportamiento lineal pero solamente para el caso de las cadenas vacías a partir de un cierto punto, esto debido a que S1 es la cadena que se mantiene vacía y S2 va aumentando de tamaño, lo cual permitió ver el comportamiento lineal para un segmento de pruebas, debido al alto tiempo de ejecución para cadenas mas grandes se hace difícil poder analizar su uso de memoria mas detalladamente.

La prueba de los *gaps*, en la que una cadena se deriva de otra mediante la eliminación de caracteres en algunas posiciones, resaltó la capacidad del algoritmo para modelar problemas en bioingeniería, como el análisis de secuencias de ADN o ARN. Los *gaps* representan eventos de mutaciones genéticas como inserciones o deleciones de segmentos, esenciales para comprender procesos evolutivos y relaciones genéticas. El éxito del algoritmo de programación dinámica en estos casos demuestra su aplicabilidad en el alineamiento de secuencias biológicas.

En conjunto, este informe evidencia cómo los resultados experimentales confirman las complejidades teóricas y destacan la programación dinámica como un enfoque ideal para resolver problemas complejos de edición de cadenas. Además, se subraya su relevancia en dominios interdisciplinarios como la bioingeniería, donde los problemas prácticos demandan soluciones escalables y precisas.

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 $\LaTeX 2_{\epsilon}$ (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 código fuente 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

A.1. Ejemplo Paso a paso de Algoritmo Fuerza Bruta

Consideremos las cadenas $S1 = "abc"$ y $S2 = "acb"$, y supongamos que tenemos los siguientes costos variables:

- **Costos de sustitución** (costo_sub):

$$\text{costo_sub}('a', 'a') = 0$$

$$\text{costo_sub}('b', 'c') = 2$$

$$\text{costo_sub}('b', 'b') = 0$$

$$\text{costo_sub}('c', 'b') = 2$$

$$\text{costo_sub} \text{ para otros pares de caracteres} = 1$$

- **Costos de inserción** (costo_ins) y **eliminación** (costo_del):

- $\text{costo_ins}(c) = 1$ para cualquier carácter c

- $\text{costo_del}(c) = 1$ para cualquier carácter c

- **Costos de transposición** (costo_trans):

- $\text{costo_trans}('b', 'c') = 1$

- costo_trans para otros pares de caracteres = 2

Procedemos a calcular la distancia mínima de edición extendida entre $S1$ y $S2$ utilizando el algoritmo de fuerza bruta.

1. Llamamos a $\text{calcular_distancia_fuerza_bruta}("abc", "acb", 3, 3)$.

2. Como $i \neq 0$ y $j \neq 0$, calculamos los costos de las operaciones:

- **Sustitución:**

- Comparamos $S1[2] = 'c'$ con $S2[2] = 'b'$. Como son diferentes, el costo de sustitución es $\text{costo_sub}('c', 'b') = 2$.

- Llamamos recursivamente a $\text{calcular_distancia_fuerza_bruta}("abc", "acb", 2, 2)$.

- **Inserción:**

- El costo de insertar $S2[2] = 'b'$ es $\text{costo_ins}('b') = 1$.

- Llamamos recursivamente a $\text{calcular_distancia_fuerza_bruta}("abc", "acb", 3, 2)$.

- **Eliminación:**

- El costo de eliminar $S1[2] = 'c'$ es $\text{costo_del}('c') = 1$.

- Llamamos recursivamente a `calcular_distancia_fuerza_bruta("abc", "acb", 2, 3)`.
- **Transposición:**
 - Verificamos si $i > 1$, $j > 1$, $S1[2] = S2[1]$ y $S1[1] = S2[2]$.
 - $S1[2] = 'c'$, $S2[1] = 'c'$, $S1[1] = 'b'$, $S2[2] = 'b'$. Se cumplen las condiciones.
 - El costo de transponer $S1[1]$ y $S1[2]$ es $costo_trans('b', 'c') = 1$.
 - Llamamos recursivamente a `calcular_distancia_fuerza_bruta("abc", "acb", 1, 1)`.
- 3. Calculamos recursivamente los costos de cada operación y sus llamadas subsecuentes. Finalmente, encontramos los siguientes costos totales:
 - **Costo total por sustitución:** $costo = 2$ (sustitución) + costo de subproblema = $2 + 0 = 2$.
 - **Costo total por inserción:** $costo = 1$ (inserción) + costo de subproblema = $1 + 2 = 3$.
 - **Costo total por eliminación:** $costo = 1$ (eliminación) + costo de subproblema = $1 + 2 = 3$.
 - **Costo total por transposición:** $costo = 1$ (transposición) + costo de subproblema = $1 + 0 = 1$.
- 4. Seleccionamos el mínimo costo, que es 1, correspondiente a la transposición de 'b' y 'c' en S1 para obtener S2.

Por lo tanto, la distancia mínima de edición extendida entre "abc" y "acb" es 1, y la operación óptima es una transposición.

A.2. Ejemplo Paso a Paso para la Distancia de Edición utilizando Programación Dinámica

Consideremos las cadenas $S1 = "abc"$ y $S2 = "acb"$, con los siguientes costos de operación:

- **Costos de sustitución ($costo_sub$):**
 - $costo_sub('a', 'a') = 0$
 - $costo_sub('b', 'c') = 2$
 - $costo_sub('b', 'b') = 0$
 - $costo_sub('c', 'b') = 2$
 - $costo_sub$ para otros pares de caracteres = 1
- **Costos de inserción ($costo_ins$) y eliminación ($costo_del$):**
 - $costo_ins(c) = 1$ para cualquier carácter c
 - $costo_del(c) = 1$ para cualquier carácter c
- **Costos de transposición ($costo_trans$):**

- $\text{costo_trans}('b', 'c') = 1$
- costo_trans para otros pares de caracteres = 2

Procedemos a calcular la distancia mínima de edición extendida entre $S1$ y $S2$ utilizando programación dinámica.

1. Inicializamos la matriz D de tamaño (4×4) (para manejar índices 0 a 3):

	0	1	2	3
0				
1				
2				
3				

Cada valor inicial en la primera fila y columna corresponde a costos acumulados de inserciones y eliminaciones.

2. Rellenamos la matriz calculando el valor mínimo de las operaciones posibles en cada paso.

- Para $D(1,1)$: Comparamos $S1[0] = 'a'$ y $S2[0] = 'a'$. Son iguales, por lo que copiamos $D(0,0)$: $D(1,1) = 0$.
- Para $D(1,2)$: Comparamos $S1[0] = 'a'$ y $S2[1] = 'c'$. Costos:
 - Sustitución: $D(0,1) + \text{costo_sub}('a', 'c') = 1 + 1 = 2$
 - Inserción: $D(1,1) + \text{costo_ins}('c') = 0 + 1 = 1$
 - Eliminación: $D(0,2) + \text{costo_del}('a') = 2 + 1 = 3$

Mínimo costo es 1 (inserción): $D(1,2) = 1$.

- Para $D(1,3)$: Comparamos $S1[0] = 'a'$ y $S2[2] = 'b'$. Costos:
 - Sustitución: $D(0,2) + \text{costo_sub}('a', 'b') = 2 + 1 = 3$
 - Inserción: $D(1,2) + \text{costo_ins}('b') = 1 + 1 = 2$
 - Eliminación: $D(0,3) + \text{costo_del}('a') = 3 + 1 = 4$

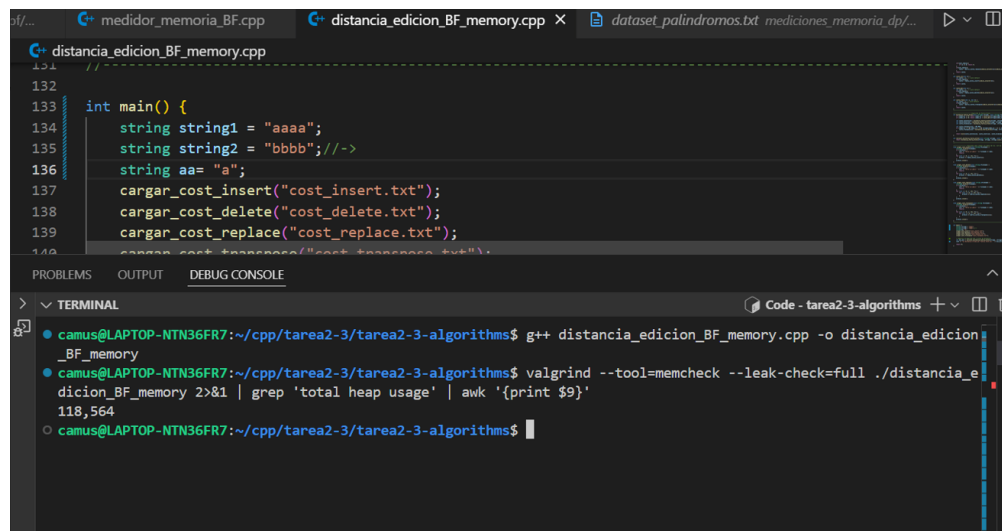
Mínimo costo es 2 (inserción): $D(1,3) = 2$.

- Continuamos este proceso para cada celda en la matriz D , considerando los casos de sustitución, inserción, eliminación, y transposición cuando es aplicable.
- En $D(3,3)$ (la esquina inferior derecha de la matriz), encontramos que el costo mínimo para transformar $S1$ en $S2$ es 1, correspondiente a la transposición de $'b'$ y $'c'$.

Por lo tanto, la distancia mínima de edición extendida entre $S1 = "abc"$ y $S2 = "acb"$ es 1, y la operación óptima es una transposición de $'b'$ y $'c'$.

A.3. Observación de asignación de memoria en C++

Al realizar una prueba de memoria de uno de los algoritmos utilizando Valgrind, podemos ver que luego de que una cadena supere los 15 caracteres, se empieza a asignar memoria adicional, pero antes de ello se mantiene constante, eso por esta razón de que el comportamiento del gráfico de la prueba 1 sea tan raro y constante para algunos casos. Como la suma máxima del tamaño de las cadenas en la prueba 1 es 26, esto quiere decir que como cadena individual ambas son de 13 caracteres aproximadamente, lo cual no supera los 15 donde C++ empieza a asignar memoria, para el único caso en el que empezó a asignar memoria fue para el de cadenas vacías, esto dividido a que para este caso se generó un dataset un poco mas grande, para que la cadena individual llegara hasta los 40 caracteres, pero se tuvo que descartar algunos pares de cadenas para hacer una buena comparación con los demás casos, por que la cadena individualmente llega hasta unos 26 caracteres aproximadamente, como esta cadena individualmente supera los 15 caracteres C++ le empieza a asignar mas memoria y es lo que se puede ver en el gráfico de la Figura 5. Lo mismo pasa para el caso de los gaps, debido a que solo una de las cadenas tiene menos caracteres, la otra cadena puede llegar a tener mas, de modo que la sumatoria llegue a 26 y pueda resolverse en un tiempo razonable para poder asignar en el gráfico. A continuación se muestran unas capturas de una prueba realizada en donde empieza a aumentar la memoria luego de los 15 caracteres en un string:



The screenshot shows a code editor with a C++ file named `distancia_edicion_BF_memory.cpp`. The code defines a `main` function that initializes two strings, `string1 = "aaaa"` and `string2 = "bbbb"`, and a character `aa = 'a'`. It then calls functions `cargar_cost_insert`, `cargar_cost_delete`, and `cargar_cost_replace` with file names. Below the code, the terminal shows the compilation and execution of the program using Valgrind. The output indicates that the total heap usage is 118,564 bytes.

```
131 //-----
132
133 int main() {
134     string string1 = "aaaa";
135     string string2 = "bbbb"; // ->
136     string aa = "a";
137     cargar_cost_insert("cost_insert.txt");
138     cargar_cost_delete("cost_delete.txt");
139     cargar_cost_replace("cost_replace.txt");
140     cargar_cost_replace("cost_replace.txt");
141 }

PROBLEMS OUTPUT DEBUG CONSOLE

> TERMINAL
camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$ g++ distancia_edicion_BF_memory.cpp -o distancia_edicion_BF_memory
camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$ valgrind --tool=memcheck --leak-check=full ./distancia_edicion_BF_memory 2>&1 | grep 'total heap usage' | awk '{print $9}'
118,564
camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$
```

Figura 7: Cadena con 1 carácter, memoria: 118564 bytes

The screenshot shows a VS Code editor with a C++ file named `distancia_edicion_BF_memory.cpp`. The code defines a `main` function that performs string operations and file I/O. The terminal window shows the compilation and execution of the program, with Valgrind output indicating a memory leak of 118,564 bytes.

```

131 //
132
133 int main() {
134     string string1 = "aaaa";
135     string string2 = "bbbb"; //->
136     string aa = "aaaaaaaaaaaaaaaa";
137     cargar_cost_insert("cost_insert.txt");
138     cargar_cost_delete("cost_delete.txt");
139     cargar_cost_replace("cost_replace.txt");
140     //cost_insert("cost_insert.txt");
141 }

```

```

camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$ g++ distancia_edicion_BF_memory.cpp -o distancia_edicion_BF_memory
camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$ valgrind --tool=memcheck --leak-check=full ./distancia_edicion_BF_memory 2>&1 | grep 'total heap usage' | awk '{print $9}'
118,564
camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$ g++ distancia_edicion_BF_memory.cpp -o distancia_edicion_BF_memory
camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$ valgrind --tool=memcheck --leak-check=full ./distancia_edicion_BF_memory 2>&1 | grep 'total heap usage' | awk '{print $9}'
118,564
camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$

```

Figura 8: Cadena con 15 caracteres, memoria: 118564 bytes

[illegible]

Figura 9: Cadena con 16 caracteres, memoria: 118581 bytes

```

132
133 int main() {
134     string string1 = "aaaa";
135     string string2 = "bbbb";//->
136     string aa = "";
137     cargar_cost_insert("cost_insert.txt");
138     cargar_cost_delete("cost_delete.txt");
139     cargar_cost_replace("cost_replace.txt");
140     cargar_cost_replace("cost_replace.txt");
141 }

```

```

camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$ valgrind --tool=memcheck --leak-check=full ./distancia_e
dicion_BF_memory 2>&1 | grep 'total heap usage' | awk '{print $9}'
118,564
camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$ g++ distancia_edicion_BF_memory.cpp -o distancia_edicion
_BF_memory
camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$ valgrind --tool=memcheck --leak-check=full ./distancia_e
dicion_BF_memory 2>&1 | grep 'total heap usage' | awk '{print $9}'
118,564
camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$ g++ distancia_edicion_BF_memory.cpp -o distancia_edicion
_BF_memory
camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$ valgrind --tool=memcheck --leak-check=full ./distancia_e
dicion_BF_memory 2>&1 | grep 'total heap usage' | awk '{print $9}'
118,581
camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$ g++ distancia_edicion_BF_memory.cpp -o distancia_edicion
_BF_memory
camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$ valgrind --tool=memcheck --leak-check=full ./distancia_e
dicion_BF_memory 2>&1 | grep 'total heap usage' | awk '{print $9}'
118,564
camus@LAPTOP-NTN36FR7:~/cpp/tarea2-3/tarea2-3-algorithms$

```

Figura 10: Cadena con 0 caracteres, memoria: 118564 bytes

A.4. Gráfico del consumo de memoria para el algoritmo de DP, (prueba 2, hasta 5000 caracteres por cadena)

Acá se puede ver claramente un crecimiento cuadrático, este crecimiento del consumo de memoria se explica principalmente debido a que el algoritmo de programación dinámica para calcular la distancia de edición extendida utiliza una matriz para ir guardando el valor de los costos para cada una de las operaciones. La matriz creada es de tamaño $(m+1)$ y $(n+1)$, donde m es el tamaño de uno de los strings o cadenas y n es el tamaño de la otra cadena. Por ello la complejidad espacial es del tipo $O(m+n)$, lo cual se puede evidenciar en el gráfico.

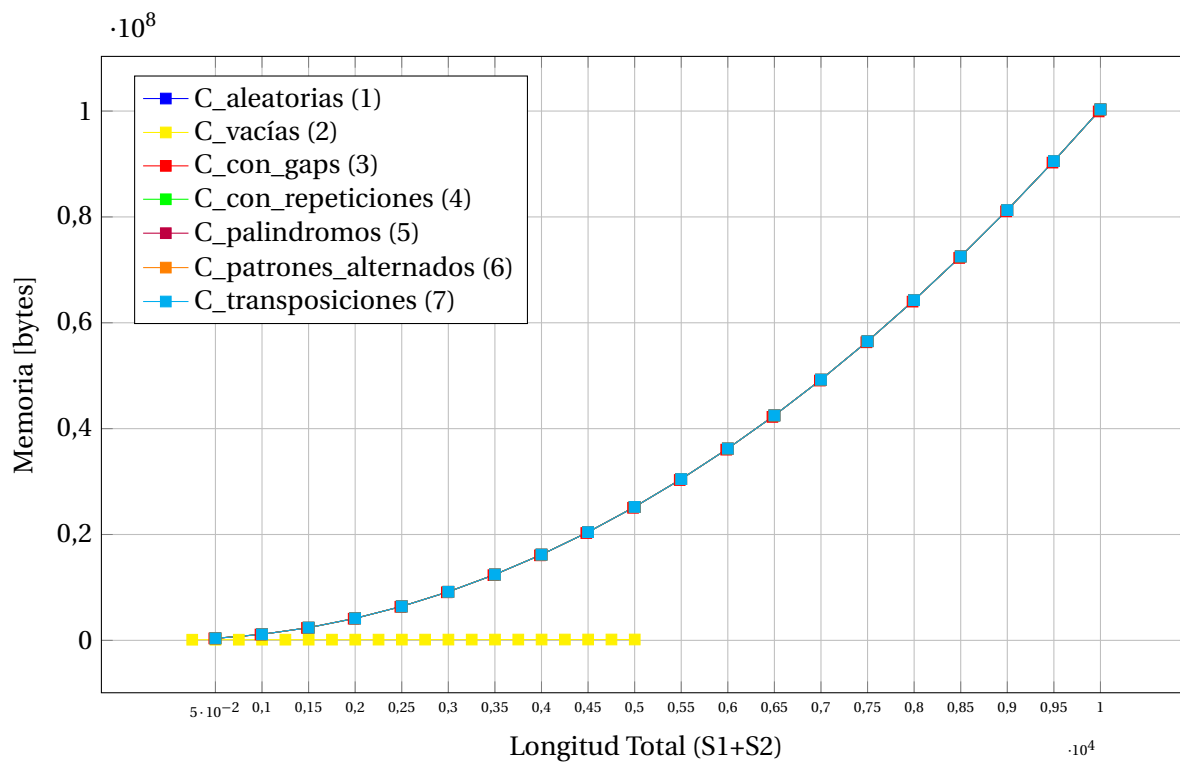


Figura 11: Memoria Utilizada Algoritmo de Programación Dinámica (Prueba 2)

Referencias

- [1] GeeksforGeeks contributors. *Edit Distance | Dynamic Programming*. Accessed: 2024-10-31. 2024. URL: <https://www.geeksforgeeks.org/edit-distance-dp-5/>.
- [2] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge, UK: Cambridge University Press, 1997. ISBN: 978-0-521-58519-4.
- [3] Kamus1. *Tarea 2-3: Algorithms*. Accedido: 2024-11-10. 2024. URL: <https://github.com/kamus1/tarea2-3-algorithms>.
- [4] Wikipedia contributors. *Damerau–Levenshtein distance* — *Wikipedia, The Free Encyclopedia*. Accessed: 2024-10-31. 2024. URL: https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance.
- [5] Wikipedia contributors. *Distancia de Damerau-Levenshtein* — *Wikipedia, La enciclopedia libre*. Accedido: 2024-10-31. 2024. URL: https://es.wikipedia.org/wiki/Distancia_de_Damerau-Levenshtein.
- [6] Wikipedia contributors. *Distancia de Levenshtein* — *Wikipedia, La enciclopedia libre*. Accedido: 2024-10-31. 2024. URL: https://es.wikipedia.org/wiki/Distancia_de_Levenshtein.
- [7] Wikipedia contributors. *Edit distance* — *Wikipedia, The Free Encyclopedia*. Accessed: 2024-10-31. 2024. URL: https://en.wikipedia.org/wiki/Edit_distance.

- [8] Wikipedia contributors. *Levenshtein distance* — *Wikipedia, The Free Encyclopedia*. Accessed: 2024-10-31, 2024. URL: https://en.wikipedia.org/wiki/Levenshtein_distance.