REPORTE TAREA 2 y 3
ALGORITMOS Y COMPLEJIDAD

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

Rodrigo Ramírez Díaz

18 de noviembre de 2024

01:59

Resumen

En este informe se aborda el problema de la distancia mínima de edición extendida entre cadenas de caracteres, implementando y evaluando dos enfoques algorítmicos: Fuerza Bruta y Programación Dinámica. Se presentan las características principales de cada enfoque, junto con una comparación detallada de sus rendimientos en términos de tiempo de ejecución y uso de memoria, utilizando datasets diseñados específicamente para evaluar casos de transposición, semiordenados y desordenados.

La infraestructura experimental incluye la generación automatizada de datos mediante Python y la evaluación de los algoritmos implementados en C++ bajo un entorno controlado. Los resultados obtenidos destacan la ineficiencia del enfoque de Fuerza Bruta para cadenas de longitud moderada o alta, en contraste el enfoque de Programación Dinámica. Este estudio subraya la importancia de seleccionar algoritmos adecuados para problemas computacionales complejos, proporcionando una base sólida para abordar problemas similares en áreas como la bioinformática y la corrección de texto.

Índice

1. Introducción

En este informe, analizaré el problema de calcular la distancia de edición entre dos cadenas de caracteres, utilizando operaciones con costos variables, como inserción, sustitución, transposición y eliminación, aplicadas una operación a la vez. Este análisis se realizará empleando dos enfoques algorítmicos distintos: Fuerza Bruta y Programación Dinámica. La elección de estos enfoques responde a la importancia de este problema en el campo del *Análisis y Diseño de Algoritmos en Ciencias de la Computación*, ya que el cálculo de la distancia de edición ha permitido avances significativos en aplicaciones como la búsqueda de texto y en motores de búsqueda (por ejemplo, Google), donde es posible restringir resultados a palabras similares dentro de un margen específico de error.

Diversas investigaciones han explorado el concepto de búsqueda aproximada, en la cual se permite cierto grado de 'error' o diferencia entre dos cadenas de caracteres, una herramienta muy utilizada en áreas como la bioinformática. Un ejemplo relevante se menciona en *Un algoritmo de Consenso para la Búsqueda Aproximada de Patrones en Cadenas de Proteínas*, donde se destaca que *üna de las principales herramientas que permiten la localización de características comunes en cadenas de proteínas o ADN de distintas especies es la búsqueda aproximada de cadenas."*. Esta técnica es fundamental para el análisis de secuencias biológicas, donde la distancia de edición se convierte en una métrica clave para medir similitudes y realizar comparaciones entre datos complejos.

El principal objetivo de este informe es resaltar la importancia de aplicar estos algoritmos de forma eficiente, reduciendo de manera significativa la complejidad temporal del problema, para ello se presentará una comparación en términos de velocidad entre los enfoques de Fuerza Bruta y Programación Dinámica. Además, se busca responder una de las preguntas fundamentales en el estudio de los algoritmos: **"¿Se puede hacer mejor?"**.

Con esta premisa en mente, el propósito de este informe es realizar un análisis exhaustivo del algoritmo de distancia de edición, centrándose principalmente en su análisis temporal. Para ello, se utilizarán diversas notaciones de complejidad, como la notación Big O, la más utilizada para evaluar el rendimiento de algoritmos. Este análisis permitirá explorar cómo un enfoque algorítmico distinto y más eficiente puede reducir significativamente el tiempo de ejecución, pasando de una complejidad exponencial a una lineal en ciertos casos. Además, se plantearán variaciones al problema estándar de distancia entre dos cadenas de carácteres, para evaluar si dichas variantes pueden ofrecer beneficios en aplicaciones prácticas actuales.

2. Diseño y Análisis de Algoritmos

En esta sección se diseñan dos algoritmos que buscan resolver el problema de la distancia mínima de edición extendida entre dos cadenas dadas, S1 y S2. Cada algoritmo utiliza las operaciones de inserción, sustitución, eliminación, y la variante de transposición, junto con sus respectivos costos.

El primer enfoque, *"Fuerza Bruta"*, implementa una solución exhaustiva en la que se exploran todos los caminos posibles para transformar S1 en S2. En este enfoque, para hacer coincidir un solo carácter entre ambas cadenas, se generan todas las opciones posibles, lo que crea un árbol de decisiones donde cada nodo representa una operación. A medida que se avanza en cada carácter de las cadenas, el proceso se repite recursivamente para las letras restantes, evaluando todas las combinaciones de operaciones posibles. Al final, cuando se han recorrido todas las ramas del árbol, se selecciona la solución con el menor costo total.

Por ejemplo, consideremos las palabras S1 = 'amores' y S2 = 'amaras'. Al comparar los primeros caracteres de ambas cadenas, no se genera ningún conflicto; sin embargo, al llegar al tercer carácter de S1 'o' y compararlo con el tercer carácter de S2 'a', el algoritmo calcula los costos de las cuatro posibles operaciones (inserción, eliminación, sustitución y transposición) para hacer que ambos caracteres coincidan. Cada una de estas operaciones genera una llamada recursiva para el resto de la cadena, resultando en cuatro ramificaciones desde este punto en el árbol.

Este proceso continúa a medida que avanzamos en las cadenas, y al comparar el quinto carácter de S1 'e' con el quinto de S2 'a', cada una de las cuatro ramificaciones previas vuelve a generar otras cuatro, ya que cada rama evalúa nuevamente las cuatro operaciones posibles. Esto resulta en un árbol de decisiones con 16 nodos hoja hasta este punto.

Este proceso puede visualizarse más fácilmente en la siguiente fig. 1 del diagrama del árbol resultante, que ilustra el crecimiento exponencial de las llamadas recursivas en el enfoque de Fuerza Bruta:

- **Análisis Temporal**

  El análisis temporal del algoritmo de Fuerza Bruta, en el peor de los casos, considera que se deben calcular los costos para cada carácter de la cadena S1. A medida que se avanza en cada carácter, se generan cuatro ramificaciones (una para cada operación: inserción, eliminación, sustitución y transposición), lo cual expande exponencialmente el árbol de decisiones. Si consideramos que la longitud de S1 es n, entonces el árbol resultante tendrá aproximadamente 4^n nodos en el peor de los casos, dado que cada operación genera cuatro posibles caminos a seguir. Por lo tanto, la complejidad temporal del algoritmo es $O(4^n)$, lo que lo convierte en un método ineficiente para cadenas largas debido a su crecimiento exponencial en tiempo de ejecución.

- **Análisis Espacial**

  En cuanto al análisis espacial, la cantidad de espacio requerida difiere del crecimiento temporal. En el algoritmo de Fuerza Bruta, cada llamada recursiva crea una copia de las cadenas restantes en la pila de ejecución, pero el almacenamiento de datos solo requiere guardar los caracteres de

S1 y S2 en el nivel actual de la recursión. Esto significa que, en el primer nivel (la raíz del árbol de decisiones), se necesita espacio para n caracteres. A medida que se desciende en el árbol y se realizan llamadas recursivas, el espacio utilizado por cada nivel se reduce, ya que las cadenas a comparar disminuyen en tamaño.

Por lo tanto, en el último nivel, correspondiente a los nodos hoja, se almacenan únicamente las operaciones finales sobre caracteres individuales. Esto implica que el espacio máximo en la pila de ejecución está limitado por la profundidad del árbol, que es n, resultando en una complejidad espacial de $O(n)$.

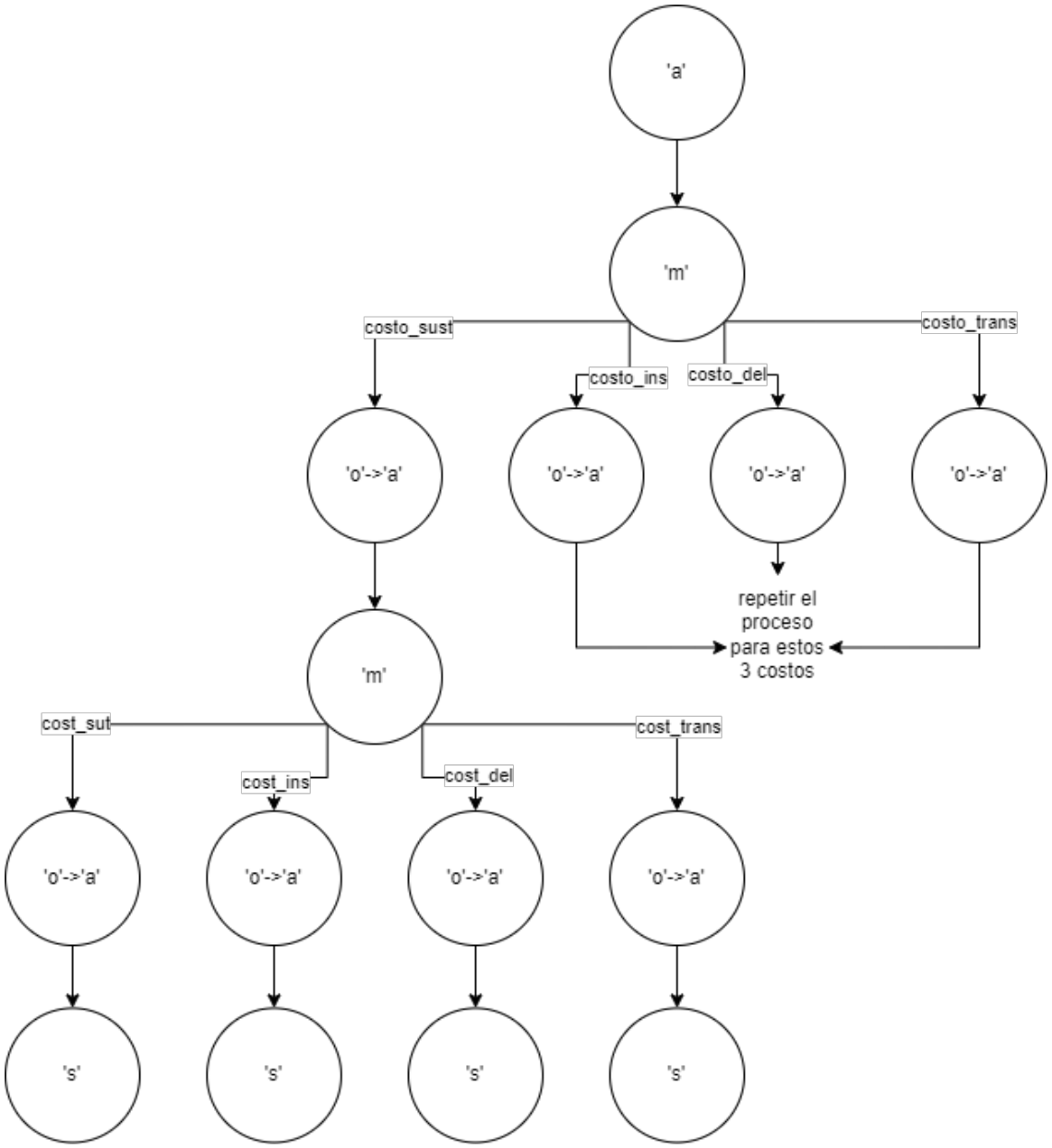

Figura 1: Diagrama del arbol implicito que resulta de la ejecucion del prgrama *brute_fi.py*.

2.1. Fuerza Bruta

Algoritmo 1: Pseudocódigo del algoritmo de fuerza bruta para calcular la distancia mínima entre dos cadenas con costos variables.

```
1  Procedure COSTO_SUB(char a, char b)
2      index1 ← obtener_indice(a)
3      index2 ← obtener_indice(b)
4      costo ← Matriz_Costos_Sustitucion[index1][index2]
5      return costo
6  Procedure COSTO_INS(char b)
7      index ← obtener_indice(b)
8      costo ← Arreglo_Costos_Insercion[index]
9      return costo
10 Procedure COSTO_DEL(char a)
11     index ← obtener_indice(a)
12     costo ← Arreglo_Costos_Eliminacion[index]
13     return costo
14 Procedure COSTO_TRANS(char a, char b)
15     index1 ← obtener_indice(a)
16     index2 ← obtener_indice(b)
17     costo ← Matriz_Costos_Transposicion[index1][index2]
18     return costo
19 Procedure FuerzaBruta(S1, S2)
20     if S1 está vacía then
21         for i desde 0 hasta longitud(S2) do
22             costo_total+ = COSTO_INS(S2[i])
23         return costo_total
24     else if S2 está vacía then
25         for i desde 0 hasta longitud(S1) do
26             costo_total+ = COSTO_DEL(S1[i])
27         return costo_total
28     else if S1[0] = S2[0] then
29         return FUERZA_BRUTA((S1[1:], S2[1:]))
30     else
31         costo_sustitucion ← COSTO_SUB(S1[0], S2[0]) + FUERZA_BRUTA(S1[1 :], S2[1 :])
32         costo_insercion ← COSTO_INS(S2[0]) + FUERZA_BRUTA(S1, S2[1 :])
33         costo_eliminacion ← COSTO_DEL(S1[0]) + FUERZA_BRUTA(S1[1 :], S2)
34         if longitud(S1) >1 y longitud(S2) >1 y S1[0] = S2[1] y S1[1] = S2[0] then
35             costo_transposicion ← COSTO_TRANS(S1[0], S1[1]) + FUERZA_BRUTA(S1[2 :], S2[2 :])
36         else
37             costo_transposicion ← ∞ //Costo muy alto;
38         return mín(costo_sustitucion, costo_insercion, costo_eliminacion, costo_transposicion)
```

2.2. Programación Dinámica

Dynamic programming is not about filling in tables. It's about smart recursion!

Erickson, 2019 [2]

2.2.1. Descripción de la solución recursiva

La solución por programación dinámica se implementa utilizando una estrategia de tabulación (*Bottom-Up*), que calcula de forma iterativa los costos mínimos necesarios para transformar una cadena $S1$ en otra $S2$. Este enfoque optimiza el cálculo al evitar la recalculación de subproblemas, almacenando los resultados parciales en una matriz de costos.

La matriz de costos tiene dimensiones $N \times M$, donde N es la longitud de $S1$ y M es la longitud de $S2$. Cada celda $dp[i][j]$ en la matriz representa el costo mínimo para transformar el prefijo o substring $S1[0..i-1]$ en el prefijo $S2[0..j-1]$. Este costo se calcula considerando cuatro operaciones posibles: *Inserción, Eliminación, Sustitución, Transposición* (si es que es aplicable).

Inicialización Se definen los valores iniciales para los casos base:

- $dp[i][0]$: El costo de transformar los primeros i caracteres de $S1$ en una cadena vacía es simplemente el costo acumulado de eliminar esos i caracteres.
- $dp[0][j]$: El costo de transformar una cadena vacía en los primeros j caracteres de $S2$ es el costo acumulado de insertar esos j caracteres.

Relleno de la Matriz La matriz se rellena iterativamente siguiendo el orden $dp[i][j]$, donde cada celda considera las cuatro operaciones posibles:

1. **Inserción:** El costo de insertar $S2[j-1]$ en $S1[0..i-1]$ y sumar el costo mínimo previo $dp[i][j-1]$.
2. **Eliminación:** El costo de eliminar $S1[i-1]$ y sumar el costo mínimo previo $dp[i-1][j]$.
3. **Sustitución:** Si $S1[i-1] \neq S2[j-1]$, el costo de sustituir $S1[i-1]$ por $S2[j-1]$, más el costo previo $dp[i-1][j-1]$. Si los caracteres son iguales, no hay costo adicional.
4. **Transposición:** Si los caracteres adyacentes entre $S1$ y $S2$ están intercambiados ($S1[i-1] = S2[j-2]$ y $S1[i-2] = S2[j-1]$), se considera el costo de transponerlos, sumado al costo previo $dp[i-2][j-2]$.

Propagación de Costos El cálculo en cada celda asegura que el valor almacenado en $dp[i][j]$ sea el mínimo entre los costos de las operaciones disponibles. Este proceso garantiza que:

- Cada subproblema (transformar un prefijo de $S1$ en un prefijo de $S2$) se resuelve óptimamente antes de ser usado en cálculos posteriores.
- Al finalizar, el valor $dp[N][M]$ en la esquina inferior derecha de la matriz contiene el costo mínimo total para transformar $S1$ en $S2$.

Ejemplo Supongamos $S1$ = `'amores'` y $S2$ = `'amaras'`:

- Se inicializan los casos base: costos acumulados para transformar una cadena en vacío.
- En cada celda, se consideran las operaciones mencionadas y se selecciona el mínimo costo.
- La matriz resultante tiene el costo total mínimo $dp[6][6]$, que es la distancia de edición extendida entre las dos cadenas.

Este enfoque aprovecha la naturaleza recursiva del problema mientras reduce la redundancia mediante almacenamiento en la matriz, logrando una complejidad temporal de $O(N \cdot M)$ y una complejidad espacial de $O(N \cdot M)$.

2.2.2. Relación de recurrencia

La relación de recurrencia establece el costo mínimo para transformar un prefijo de `S1` en un prefijo de `S2`. Se define como:

$$D(i,j) = \text{mín} \begin{cases} D(i-1,j) + \text{COSTO\_DEL}(\texttt{S1}[i-1]) & \text{(eliminación)} \\ D(i,j-1) + \text{COSTO\_INS}(\texttt{S2}[j-1]) & \text{(inserción)} \\ D(i-1,j-1) + \text{COSTO\_SUB}(\texttt{S1}[i-1], \texttt{S2}[j-1]) & \text{(sustitución)} \\ D(i-2,j-2) + \text{COSTO\_TRANS}(\texttt{S1}[i-2], \texttt{S1}[i-1]) & \text{(transposición)} \end{cases}$$

con los casos base:

$$D(i,0) = i \cdot \text{COSTO\_DEL}(\texttt{S1}[i-1])$$

$$D(0,j) = j \cdot \text{COSTO\_INS}(\texttt{S2}[j-1])$$

2.2.3. Identificación de subproblemas

Cada subproblema consiste en calcular el costo mínimo de transformar un prefijo de `S1` en un prefijo de `S2`. Esto se traduce en encontrar la mínima cantidad de operaciones para cada prefijo de longitud i y j de `S1` y `S2`, respectivamente. Al resolver estos subproblemas y almacenarlos en la matriz, se evita la recalculación redundante y se optimiza el cálculo de la solución final.

2.2.4. Estructura de datos y orden de cálculo

La estructura de datos utilizada es una matriz bidimensional $D[i][j]$, donde $D[i][j]$ almacena el costo mínimo para transformar el prefijo de longitud i de `S1` en el prefijo de longitud j de `S2`. La matriz se rellena secuencialmente desde la esquina superior izquierda hasta la esquina inferior derecha, asegurando que cada valor $D[i][j]$ dependa únicamente de valores previamente calculados.

2.2.5. Algoritmo utilizando programación dinámica

Algoritmo 2: Algoritmo de Programación Dinámica para la distancia mínima de edición entre dos cadenas.

```
1  Procedure DistanciaMinima(S1, S2)
2      n ← longitud de S1
3      m ← longitud de S2
4      Crear matriz D de tamaño (n+1) × (m+1)
5      for i ← 0 to n do
6          D[i][0] ← i × COSTO_DEL(S1[i−1])
7      for j ← 0 to m do
8          D[0][j] ← j × COSTO_INS(S2[j−1])
9      for i ← 1 to n do
10         for j ← 1 to m do
11             costo_del ← D[i−1][j] + COSTO_DEL(S1[i−1])
12             costo_ins ← D[i][j−1] + COSTO_INS(S2[j−1])
13             costo_sub ← D[i−1][j−1] + COSTO_SUB(S1[i−1], S2[j−1])
14             if i > 1 ∧ j > 1 ∧ S1[i−1] = S2[j−2] ∧ S1[i−2] = S2[j−1] then
15                 costo_trans ← D[i−2][j−2] + COSTO_TRANS(S1[i−2], S1[i−1])
16             else
17                 costo_trans ← ∞
18             D[i][j] ← mín(costo_del, costo_ins, costo_sub, costo_trans)
19     return D[n][m]
```

3. Implementaciones

Para la implementación del programa se ha agregado una carpeta llamada, *Ïmplementacion"*, en la cual irán los respectivos archivos README.md, costos e implementacion de codigos utilizando el lenguaje de programacion C++, de todas formas, adjunto el link al repositorio, ubicandolo exactamente en la carpeta de Implementacion para evitar posible errores o confusiones.

Puedes encontrar mi repositorio de GitHub en el siguiente enlace: Mi repositorio de GitHub, seccion: Implementacion.

4. Experimentos

> *"Non-reproducible single occurrences are of no significance to science."*
>
> —Popper, 2005 [3]

En esta sección de Experimentos, analizaremos y compararemos los tiempos de ejecución y la memoria utilizada por los algoritmos implementados, utilizando gráficos que nos ayuden a visualizar las diferencias significativas entre los enfoques de *Fuerza Bruta* y *Programación Dinámica*.

4.0.1. Configuración de los Experimentos

La recopilación de datos se realiza desde el momento en que se llaman las funciones implementadas en los archivos correspondientes. Estas funciones reciben como parámetros dos cadenas, $S1$ y $S2$, que representan los datos de entrada. Una vez que la función finaliza su ejecución, se obtienen los siguientes datos de salida: - La distancia mínima necesaria para transformar $S1$ en $S2$. - El tiempo de ejecución medido en segundos y microsegundos. - La cantidad de memoria gestionada en bytes, recopilada mediante la herramienta **Valgrind** en el campo *bytes allocated*.

Un ejemplo de ejecución se puede observar en la fig. 2:

```
yoyo@DESKTOP-U0ARGTD:/mnt/c/Users/yoyo/OneDrive - Universidad Técnica Federico Santa María/Algoritmo y complejidad/Tareas/Tarea_2&3/Implementation$ make ejecutar_brute_force
valgrind --leak-check=yes ./brute_force
==47484== Memcheck, a memory error detector
==47484== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==47484== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==47484== Command: ./brute_force
==47484==
intention
execution
Distancia mínima: 8

EL tiempo de ejecucion en segundos es: 0.730533

El tiempo de ejecucion en microsegundos es: 730533

==47484==
==47484== HEAP SUMMARY:
==47484==     in use at exit: 0 bytes in 0 blocks
==47484==   total heap usage: 71 allocs, 71 frees, 116,498 bytes allocated
==47484==
==47484== All heap blocks were freed -- no leaks are possible
==47484==
==47484== For lists of detected and suppressed errors, rerun with: -s
==47484== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 2: Ejemplo de la ejecución del programa con sus respectivas entradas y salidas.

4.0.2. Entorno de Pruebas

Para llevar a cabo los experimentos, se utilizó el siguiente hardware y software:

Hardware: - Procesador: Intel Core i5-5200U, 2.20 GHz. - Memoria RAM: 12 GB. - Almacenamiento: Sin SSD.

Software: - Sistema Operativo: Windows 10. - Terminal: Ubuntu 22.04.3 LTS (en entorno WSL). - Compilador: `g++ 11.4.0`. - Librerías utilizadas: `<bits/stdc++.h>`, `<sys/time.h>`

Esta configuración asegura una ejecución reproducible de los experimentos, permitiendo contrastar los resultados en términos de tiempos de ejecución y uso de memoria bajo condiciones controladas.

4.1. Dataset (casos de prueba)

Para los casos de prueba o dataset, se hace uso del lenguaje de programación Python para facilitar la generación automática de datos en archivos `.txt`, los cuales se utilizan como datos de entrada para los programas `brute_force.cpp` y `dynamic_programming.cpp`.

El programa GENERATE_DATASETS.PY genera automáticamente 5 archivos `.txt` para cada una de las tres categorías de casos de prueba definidas:

4.1.1. Transpose Datasets

En esta categoría se generan 5 archivos con el prefijo `input_transpose`, enumerados del 1 al 5. Los strings en estos archivos se ordenan de menor a mayor longitud, y todos los problemas planteados pueden resolverse exclusivamente mediante operaciones de transposición. Sin embargo, no se garantiza que la solución óptima obtenida por los programas implique únicamente transposiciones, ya que esto dependerá de los costos asociados a cada operación y de si estas resultan más convenientes.

4.1.2. Semi-Ordered Datasets

En esta categoría, la mitad del string $S1$ será idéntica a la otra mitad del string $S2$. Esto permite observar cómo se comportan los enfoques algorítmicos de *Fuerza Bruta* y *Programación Dinámica* en casos parcialmente ordenados. Los archivos generados llevan el prefijo `input_semiordered`.

4.1.3. Disordered Datasets

Por último, esta categoría contiene datasets donde $S1$ y $S2$ son completamente desordenados, generados de manera aleatoria. Los archivos generados llevan el prefijo `input_disordered`. En este caso, es probable que el enfoque de *Fuerza Bruta* tarde significativamente más tiempo en procesar strings con una longitud superior a 16 caracteres, debido al crecimiento exponencial de su árbol implícito de llamadas recursivas y la necesidad de explorarlo en profundidad.

Ver la siguiente fig. 3

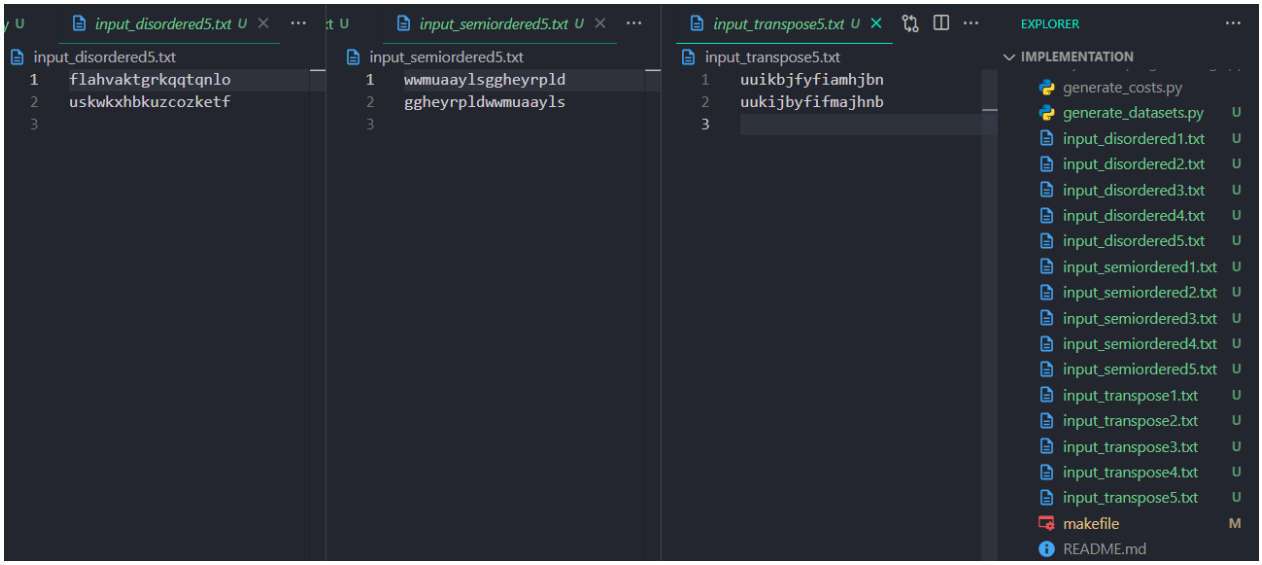

Figura 3: Ejemplo de los archivos .txt generados por el archivo *generate_datasets.py*.

4.2. Resultados

En esta sección se presentan los resultados de ambos enfoques algorítmicos mediante gráficos de dispersión, considerando los costos y archivos de entrada disponibles en el repositorio de GitHub referenciado en la sección de Implementación. Cabe destacar que estos archivos no han sido ni serán modificados, lo que permite verificar la reproducibilidad de los resultados.

Para replicar los experimentos, basta con ejecutar los comandos `make` mencionados anteriormente, ingresando manualmente los datos de entrada que se encuentran en los archivos `.txt`. Aunque sería posible automatizar todos los casos mediante `make`, esto haría el proceso innecesariamente complejo y menos flexible para el usuario, limitando la posibilidad de probar casos específicos de forma manual. Para detalles sobre la compilación y ejecución de los programas `.cpp`, puede utilizar el comando `make help` en la terminal. Es importante asegurarse de que el sistema tenga un compilador compatible con archivos `.cpp` y soporte para `make`, de lo contrario, los comandos no funcionarán correctamente.

En la fig. 4, se presentan los resultados obtenidos al ejecutar los cinco casos del dataset *transpose*. La gráfica muestra un claro crecimiento exponencial en el tiempo de ejecución del algoritmo basado en *Fuerza Bruta* (puntos en color azul), mientras que el enfoque de *Programación Dinámica* (puntos en color naranja) mantiene un crecimiento lineal. Se observan casos anómalos, por ejmplo cuando la longitud de $S1$ es 5 utilizando el enfoque de fuerza bruta, los cuales podrían ser explicados por variaciones en la gestión de memoria o costos específicos.

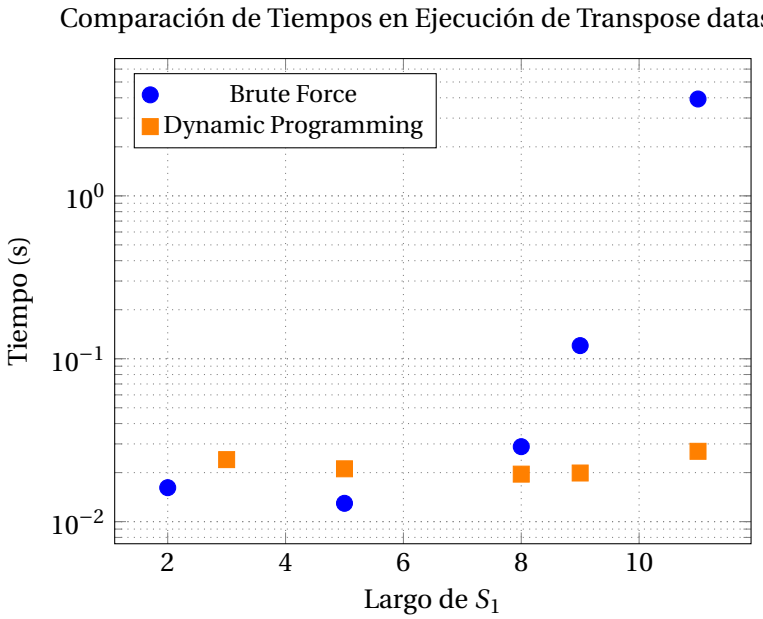

Figura 4: Gráfica de dispersión para *transpose dataset*, contrastando el uso del enfoque de fuerza bruta y el de programación dinámica.

En las siguientes figuras (fig. 5 y fig. 6), se muestran los resultados obtenidos para los datasets *semiordered* y *disordered*. En ambos casos, se observa un comportamiento similar al descrito anteriormente: los tiempos de ejecución dependen principalmente de la longitud de $S1$. Esto es consistente independientemente de si las cadenas están parcialmente ordenadas o completamente desordenadas.

El rendimiento del algoritmo mejora significativamente en casos donde:

1. $S1$ y $S2$ comparten una gran cantidad de caracteres en las mismas posiciones.
2. Uno de los strings está vacío o es considerablemente más corto que el otro.

Estos escenarios representan casos interesantes para probar ambos enfoques, y se invita al lector a explorar estas posibilidades para comprender mejor las fortalezas y limitaciones de cada algoritmo.

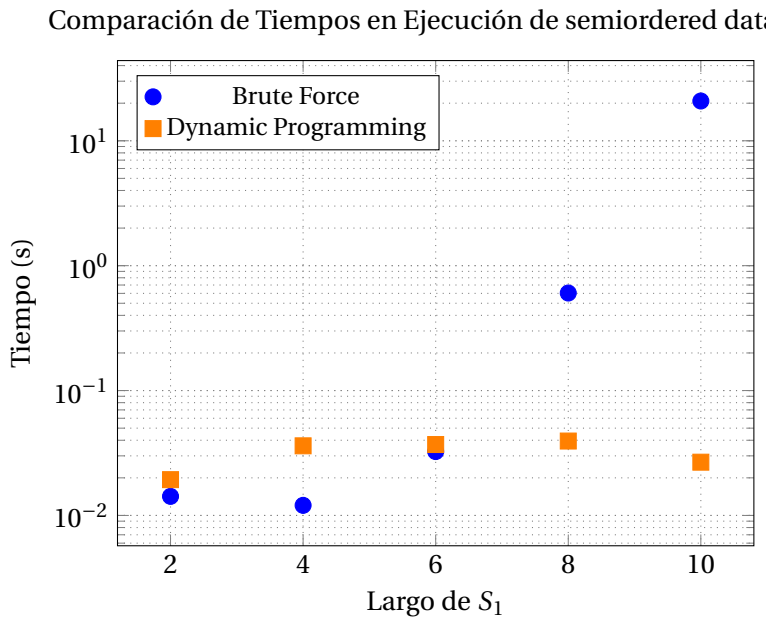

Figura 5: Gráfica de dispersión para *semiordered dataset*, contrastando el uso del enfoque de fuerza bruta y el de programación dinámica.

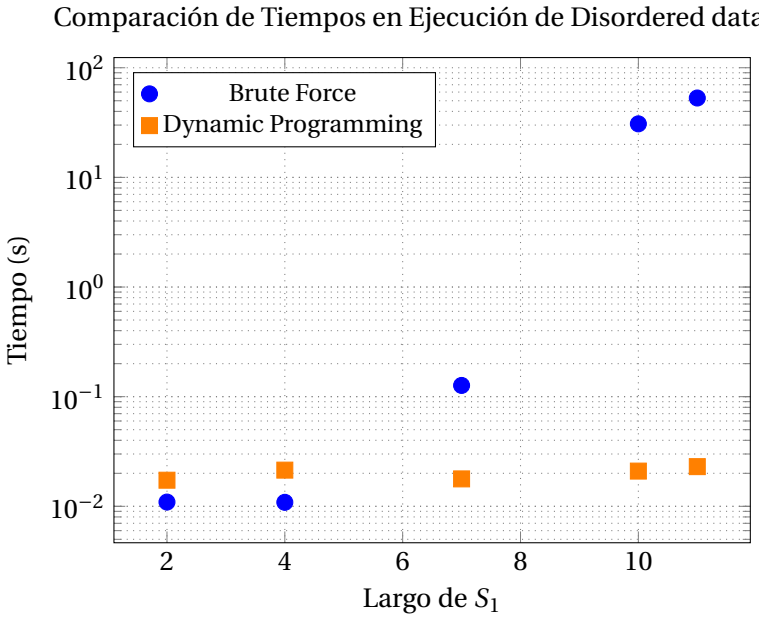

Figura 6: Gráfica de dispersión para *disordered dataset*, contrastando el uso del enfoque de fuerza bruta y el de programación dinámica.

Finalmente, para replicar las gráficas presentadas, puede modificar las tuplas en la sección *tikz* del tarball proporcionado, o bien, utilizar herramientas como Excel para generar automáticamente gráficos de dispersión ingresando los mismos datos. Esta última opción puede ser más conveniente si no está trabajando estrictamente con LaTeX.

5. Conclusiones

Los resultados obtenidos evidencian de manera clara las diferencias de rendimiento entre los enfoques de *Fuerza Bruta* y *Programación Dinámica* al resolver el problema de la distancia mínima de edición. El enfoque de *Fuerza Bruta* muestra un crecimiento exponencial en el tiempo de ejecución a medida que aumenta la longitud de S1, alcanzando tiempos inviables para tamaños moderadamente grandes. Por otro lado, *Programación Dinámica* mantiene un rendimiento altamente eficiente, con tiempos de ejecución que permanecen casi constantes, incluso para cadenas largas.

Estos resultados confirman que *Programación Dinámica* no solo es significativamente más rápida, sino que además proporciona una solución escalable para problemas similares en contextos prácticos. Esto refuerza la importancia de elegir algoritmos óptimos al enfrentar problemas que implican un alto costo computacional, ya que un enfoque eficiente puede marcar la diferencia entre la viabilidad o inviabilidad de una solución.

A. Apéndice 1

Referencias

[1] M. Rubio-Rincón A. Alba. *Un algoritmo de Consenso para la Búsqueda Aproximada de Patrones en Cadenas de Proteínas*. https://www.medigraphic.com/pdfs/inge/ib-2012/ib122c.pdf. Accessed: 2024-11-16. Dic. de 2012.

[2] Jeff Erickson. *Algorithms*. Jun. de 2019. ISBN: 978-1-792-64483-2.

[3] K. Popper. *The Logic of Scientific Discovery*. Routledge Classics. Taylor & Francis, 2005. ISBN: 9781134470020. URL: https://books.google.cl/books?id=LWSBAgAAQBAJ.