REPORTE TAREA 2 y 3
ALGORITMOS Y COMPLEJIDAD

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

Alvaro Opazo Saavedra

14 de noviembre de 2024

20:31

Resumen

Este informe analiza y compara dos algoritmos para resolver el problema de la distancia de edición entre cadenas: un enfoque de Fuerza Bruta y otro de Programación Dinámica. El objetivo principal es evaluar el rendimiento de ambos algoritmos en términos de tiempo de ejecución y cantidad de operaciones, considerando varios casos de prueba. Para ello, se implementaron ambos algoritmos en C++ y se realizaron experimentos utilizando un conjunto de cadenas predefinido. Los resultados obtenidos muestran diferencias significativas en tiempos de ejecución para ciertos casos, y se validaron mediante gráficos y tablas generados con Python, asegurando la reproducibilidad de los experimentos. Finalmente, la conclusión destaca cómo los resultados responden a las preguntas planteadas sobre la eficiencia y aplicabilidad de estos algoritmos en escenarios prácticos.

Índice

1. Introducción

El análisis y diseño de algoritmos es un campo fundamental en las Ciencias de la Computación, dado que permite resolver problemas complejos de manera eficiente, optimizando el uso de recursos como el tiempo y la memoria. Dentro de este campo, el cálculo de la distancia de edición (o distancia de Levenshtein) es un problema clásico que se utiliza para medir el grado de similitud entre dos secuencias de caracteres. Este problema tiene aplicaciones directas en diversas áreas, como el procesamiento de lenguaje natural y los sistemas de corrección ortográfica, lo que subraya su relevancia [3].

A lo largo del tiempo, se han propuesto diversos enfoques para resolver el problema de la distancia de edición. El más simple de estos es la Fuerza Bruta, que explora todas las posibles transformaciones entre las dos cadenas, lo que resulta en un algoritmo con una complejidad exponencial [4]. Por otro lado, la Programación Dinámica, introducida en el ámbito de los algoritmos por Bellman [1], ofrece una solución más eficiente en términos de tiempo, reduciendo la complejidad a O(n*m) mediante el uso de una tabla que almacena subproblemas previamente resueltos. A pesar de que la Programación Dinámica es teóricamente más eficiente, en ciertos escenarios prácticos esta diferencia de rendimiento puede no ser tan evidente debido a las características específicas de los datos o implementaciones.

El propósito de este informe es evaluar ambos enfoques —Fuerza Bruta y Programación Dinámica— para el cálculo de la distancia de edición, comparando su rendimiento en términos de tiempo de ejecución y cantidad de operaciones necesarias. Además, se busca analizar si las implementaciones de estos algoritmos cumplen con las expectativas teóricas en cuanto a eficiencia y complejidad, como se detalla en la obra de Cormen et al. [2].

Este informe aborda una pregunta clave: ¿hasta qué punto las optimizaciones teóricas de la Programación Dinámica se traducen en mejoras prácticas frente a la Fuerza Bruta? El análisis de estos algoritmos no solo permitirá contrastar sus desempeños en distintos escenarios, sino también comprender las implicaciones de seleccionar un enfoque algorítmico sobre otro, dependiendo de las características del problema.

A lo largo del desarrollo del informe, se expondrán los detalles de la implementación de ambos algoritmos, los experimentos realizados para evaluar su desempeño y los resultados obtenidos. Se espera que estos análisis proporcionen una comprensión más profunda de los factores que influyen en la eficiencia de los algoritmos de distancia de edición y cómo estos se comportan en la práctica, más allá de sus complejidades teóricas.

2. Diseño y Análisis de Algoritmos

2.1. Fuerza Bruta

> *"Indeed, brute force is a perfectly good technique in many cases; the real question is, can we use brute force in such a way that we avoid the worst-case behavior?"*
>
> — **taocv3**, **taocv3** [**taocv3**]

El algoritmo realiza una búsqueda recursiva exhaustiva, donde en cada paso calcula los resultados de todas las operaciones posibles y selecciona la que tiene el menor costo. Este enfoque es ineficiente, ya que repite los mismos cálculos para las subcadenas muchas veces. La complejidad del algoritmo crece exponencialmente debido a la gran cantidad de posibles combinaciones que se deben considerar.

Algoritmo 1: Algoritmo con un enfoque de fuerza bruta basado en recursión

```
1  Procedure EDITDISTANCE(S1, S2)
2      if LENGTH(S1) es vacía then
3          return LENGTH(S2)
4      else if LENGTH(S2) es vacía then
5          return LENGTH(S1)
6      return CASES(S1, S2, 0, 0)
7  Procedure CASES(S1, S2, i, j)
8      if LENGTH(S1) es igual a i then
9          return LENGTH(S2) - j
10     else if LENGTH(S2) es igual a j then
11         return LENGTH(S1) - i
12     costo ← 0
13     if S1[i] == S2[j] then
14         costo ← CASES(S1, S2, i + 1, j + 1)
15     else
16         inserción ← CASES(S1, S2, i, j + 1)
17         eliminación ← CASES(S1, S2, i + 1, j)
18         sustitución ← CASES(S1, S2, i + 1, j + 1)
19         transposición ← ∞
20         if i + 1 < LENGTH(S1) and j + 1 < LENGTH(S2) and S1[i] == S2[j + 1] and S1[i + 1] == S2[j] then
21             transposición ← CASES(S1, S2, i + 2, j + 2)
22         costo ← mín(inserción, eliminación, sustitución, transposición) + 1
23     return costo
```

2.2. Programación Dinámica

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

algorithms_erickson, **algorithms_erickson** [**algorithms_erickson**]

1) La solución recursiva consiste en comparar los caracteres de las dos cadenas de izquierda a derecha y realizar una de las 4 operaciones posibles. La función recursiva evalúa todas estas opciones en cada paso y elige la que tiene el menor costo. Si las cadenas ya coinciden en la posición actual, simplemente avanza a la siguiente posición sin realizar ninguna operación.

2) Para este problema, la recurrencia es:

$$dp[i][j] = \text{mín} \begin{cases} dp[i][j-1]+1 & \text{(inserción)} \\ dp[i-1][j]+1 & \text{(eliminación)} \\ dp[i-1][j-1]+2 & \text{(sustitución)} \\ dp[i-2][j-2]+1 & \text{(transposición, si aplica)} \end{cases}$$

Casos base:

- Si una de las cadenas está vacía, el costo es simplemente la longitud de la otra cadena, ya que todas las operaciones serán inserciones o eliminaciones.
- Si los caracteres actuales de ambas cadenas son iguales, no se realiza ninguna operación y el costo es el mismo que el de la subcadena anterior.

3)
- $dp[i][j]$ representa el costo de convertir los primeros i caracteres de $S1$ en los primeros j caracteres de $S2$.
- Cada cálculo de $dp[i][j]$ depende de los resultados previos almacenados en $dp[i-1][j]$, $dp[i][j-1]$, $dp[i-1][j-1]$, o $dp[i-2][j-2]$ (en caso de transposición).

En lugar de recalcular estos valores repetidamente, los almacenamos en la tabla dp, de manera que cada subproblema se resuelve solo una vez.

4) La estructura de datos que usamos es una **matriz bidimensional** ($dp[][]$), donde cada celda $dp[i][j]$ contiene el costo mínimo para convertir la subcadena $S1[0...i]$ en la subcadena $S2[0...j]$.

El orden de cálculo es **de abajo hacia arriba**:

- Primero, se llenan los casos base donde una de las cadenas es vacía.
- Luego, para cada par de índices i y j, se calculan las operaciones posibles y se toma la de menor costo.
- Finalmente, el valor de $dp[m][n]$ nos da la distancia de edición final.

Esto garantiza que cada subproblema se resuelva en el orden correcto, ya que cada celda $dp[i][j]$ depende de otras celdas que ya han sido calculadas.

Algoritmo 2: Algoritmo de distancia de ediciónusando programación dinámica

```
1  Procedure EDITDISTANCEDP(S1, S2, m, n)
2      dp ← matriz 2D de tamaño (m + 1, n + 1)
3      for i ← 0 to m do
4          for j ← 0 to n do
5              if i == 0 then
6                  dp[i][j] ← j
7              else if j == 0 then
8                  dp[i][j] ← i
9              else if S1[i - 1] == S2[j - 1] then
10                 dp[i][j] ← dp[i − 1][j − 1]
11             else
12                 inserción ← dp[i][j − 1]
13                 eliminación ← dp[i − 1][j]
14                 sustitución ← dp[i − 1][j − 1]
15                 transposición ← ∞
16                 if i >1 and j >1 and S1[i - 1] == S2[j - 2] and S1[i - 2] == S2[j - 1] then
17                     transposición ← dp[i − 2][j − 2]
18                 dp[i][j] ← 1 + mín(inserción, eliminación, sustitución, transposición)
19     return dp[m][n]
```

Ejemplo de ejecución

Supongamos las cadenas $S1$ = "ACB" y $S2$ = "ABC", sabiendo que los costos para cada operación son los siguientes:

- costo_sub(a, b) = 2 si $a \neq b$, y 0 si $a = b$
- costo_ins(b) = 1 para cualquier carácter b.
- costo_del(a) = 1 para cualquier carácter a.
- costo_trans(a, b) = 1 para transponer los caracteres adyacentes a y b.

Fuerza bruta:

1. Comparamos los primeros caracteres: "A" y "A", son iguales, por lo que avanzamos sin costo adicional.
2. Comparamos los segundos caracteres: "C" y "B". Aquí se pueden hacer varias operaciones:
   - Insertar: Insertar "B" en S1 después de "A", luego tenemos que comparar "CB" con "BC". Costo total hasta este punto: 1 (inserción).
   - Eliminar: Eliminar "C" de S1, y comparar "A" con "BC". Costo total hasta este punto: 1 (eliminación).

- Sustituir: Sustituir "C" por "B", luego comparar "B" con "C". Costo total hasta este punto: 2 (sustitución).
- Transponer: Intercambiar "C" y "B" en S1, de modo que las cadenas ahora sean iguales ("ABC" y "ABC"). Costo total hasta este punto: 1 (transposición).

La operación de transposición es la que tiene el menor costo en este caso.

3. El costo total mínimo para convertir "ACB" en "ABC" es 1, usando la transposición.

Programación dináica:

1. Creamos una matriz de tamaño $(3+1)x(3+1)$ (una fila y columna adicional para el caso de cadenas vacías).
2. Inicializamos los casos base, si una de las cadenas es vacía, el costo es igual a la longitud de la otra cadena (inserciones o eliminaciones).
3. Rellenamos la matriz utilizando las operaciones posibles (inserción, eliminación, sustitución y transposición) en cada paso, calculando el costo mínimo en cada celda.
4. En la celda dp[3][3], que corresponde a la conversión completa de "ACB" en "ABC", el valor es 1, que indica el costo mínimo utilizando transposición.

Complejidad de los algoritmos

Fuerza bruta:

- Temporal: $O(4^n)$, donde n es la longitud de la cadena más larga. Esto se debe a que, en cada posición, hay cuatro opciones (inserción, eliminación, sustitución, transposición), y cada rama se evalúa recursivamente.
- Espacial: $O(n)$, ya que el algoritmo de fuerza bruta utiliza la pila de recursión para almacenar las llamadas recursivas. En el peor caso, la profundidad de la pila es proporcional a la longitud de la cadena.

Programación dináica:

- Termporal: $O(m*n)$, donde m y n son las longitudes de las cadenas S1 y S2. Esto se debe a que llenamos una tabla de m+1 filas y n+1 columnas, evaluando cada celda una sola vez.
- Espacial: $O(m*n)$, ya que necesitamos almacenar una matriz de tamaño (m+1) x (n+1) para guardar los resultados intermedios.

Impacto de las transposiciones y los costos variables en la complejidad

La inclusión de transposiciones y costos variables no afecta la complejidad temporal y espacial de forma significativa en el enfoque de programación dinámica. Las transposiciones simplemente añaden una nueva condición a evaluar en cada paso, pero no aumentan el número de operaciones de forma exponencial.
Sin embargo, en el enfoque de fuerza bruta, agregar transposiciones incrementa el número de posibilidades en cada paso, lo que puede aumentar el número de ramas en el árbol de recursión. Esto puede hacer que el rendimiento del algoritmo de fuerza bruta empeore aún más.

3. Implementaciones

Fuerza bruta:

1. Estructura general
   - El programa define una función *editDistanceBrute* que implementa la lógica recursiva para calcular la distancia mínima de edición entre dos cadenas, utilizando las operaciones de inserción, eliminación, sustitución y transposición.
   - La función *main()* lee las cadenas de entrada, llama a *editDistanceBrute()* y muestra el resultado en la consola.
2. Funcionamiento de la recursión
   - La función editDistanceBrute(S1, S2, i, j) compara las subcadenas de S1 y S2 empezando en los índices i y j. Si las cadenas son iguales en las posiciones actuales, la función avanza a la siguiente posición sin costo adicional.
   - Si los caracteres difieren, la función explora cuatro operaciones posibles, inserción, eliminación, sustitución o transposición.

Programación dinámica:

1. Estructura general
   - La función principal es *editDistanceDP(string S1, string S2)*, que calcula la distancia mínima de edición entre dos cadenas S1 y S2 utilizando una matriz para almacenar los resultados intermedios.
   - El programa inicializa una matriz bidimensional dp de tamaño (m + 1) x (n + 1), donde m y n son las longitudes de las cadenas S1 y S2, respectivamente.
2. Inicialización de los casos base
   - Si una de las cadenas está vacía, el costo de convertirla en la otra cadena es simplemente el número de inserciones o eliminaciones necesarias. Esto se refleja en la primera fila y columna de la matriz dp.
3. Evaluación de las operaciones
   - Para cada par de caracteres en las cadenas S1 y S2, se calculan las cuatro posibles operaciones, inserción, eliminación, sustitución o transposición.

4. Experimentos

Todos los experimentos fueron ejecutados en un sistema con las siguientes especificaciones:

- **Hardware:**
  - **Procesador:** Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, 6 núcleos, 12 hilos.
  - **RAM:** 16 GB DDR4.
  - **Tarjeta Gráfica:** NVIDIA GTX 1050 (notebook).
  - **Almacenamiento:** SSD 500GB.
- **Software:**
  - **Sistema Operativo:** Windows 11 utilizando WSL con Ubuntu 22.04.3 LTS.
  - **Compilador:** gcc 11.4.0 para la compilación del código en C++.
  - **Librerías:**
    - `climits`, `vector`, `string`, `fstream`, `sstream` para en C++.
    - Librerías de Python: `time`, `subprocess`, `pandas`, `matplotlib`, `collections`

4.1. Dataset (casos de prueba)

Caso 1: Cadenas Vacías

Cadenas de entrada: (“", “")

Secuencia de operaciones esperada: Ninguna operación necesaria.

Costo total esperado: 0

Caso 2: Una Cadena Vacía

Cadenas de entrada: (“", “$abcdefg$")

Secuencia de operaciones esperada: Insertar los caracteres de “$abcdefg$" en la primera cadena.

Costo total esperado: 7 (inserciones)

Caso 3: Caracteres Repetidos

Cadenas de entrada: (“$aaaaaaaaaa$", “$aaaaaaaa$")

Secuencia de operaciones esperada: Eliminar dos caracteres de la primera cadena.

Costo total esperado: 2 (eliminaciones)

Caso 4: Necesidad de Transposición

Cadenas de entrada: (“$abba$", “$baba$")

Secuencia de operaciones esperada: Transponer los caracteres en las posiciones 1 y 2.

Costo total esperado: 2 (1 transposición)

Caso 5: Sustituciones

Cadenas de entrada: ("$kitten$","$sitting$")

Secuencia de operaciones esperada: Sustituir "k" por "s", "e" por "i", insertar "g" al final.

Costo total esperado: 3 (2 sustituciones y 1 inserción)

Caso 6: Ningún Carácter en Común

Cadenas de entrada: ("$abcdefgh$","$ijklmnop$")

Secuencia de operaciones esperada: Sustituir cada carácter de la primera cadena por el correspondiente en la segunda.

Costo total esperado: 8 (sustituciones)

Caso 7: Transposición Parcial

Cadenas de entrada: ("$abcdef$","$abcfed$")

Secuencia de operaciones esperada: Sustituir la "d" de la primera cadena por "f" y luego sustituir la última "f" por "d".

Costo total esperado: 2 (sustituciones)

Caso 8: Posibles Transposiciones Múltiples

Cadenas de entrada: ("$longstringtest$","$lontgsritenst$")

Secuencia de operaciones esperada: En la primera cadena, inserta una "t" entre la "g" y "n", luego elimina la siguiente "t", "n" y "g" e inserta una "n" entre la "e" y "s".

Costo total esperado: 5 (2 inserciones y 3 eliminaciones)

Caso 9: Cadenas Idénticas

Cadenas de entrada: ("$abcdefghijklmnop$","$abcdefghijklmnop$")

Secuencia de operaciones esperada: Ninguna operación necesaria.

Costo total esperado: 0

Caso 10: Subcadena

Cadenas de entrada: ("$abcdefghijklmnop$","$efghijk$")

Secuencia de operaciones esperada: Eliminar los caracteres iniciales y finales de la primera cadena.

Costo total esperado: 9 (eliminaciones)

Caso 11: Ejemplo Clásico

Cadenas de entrada: ("$intention$","$execution$")

Secuencia de operaciones esperada: Sustituir "i" por "e", "n" por "x", "t" por "c", "t" por "u", y agregar "o".

Costo total esperado: 5 (4 sustituciones y 1 inserción)

4.2. Resultados

Los resultados presentados en esta sección se obtuvieron mediante un análisis experimental automatizado. Para ello, se implementaron dos versiones del algoritmo de distancia de edición: una utilizando fuerza bruta y otra mediante programación dinámica. Los tiempos de ejecución y las operaciones fueron registrados para varios conjuntos de pruebas.

Todos los gráficos y tablas que se muestran a continuación fueron generados utilizando un programa en Python. Este programa automatiza el análisis de datos, permitiendo la ejecución repetible de los experimentos y la visualización de los resultados en forma de gráficos comparativos y tablas de operaciones. Los archivos utilizados para la generación de estos resultados están disponibles, lo que asegura la reproducibilidad de los análisis al ejecutar los scripts provistos.

En la fig. 1 se observa la comparación entre los resultados de Fuerza Bruta y Programación Dinámica en términos de cantidad de operaciones. Se puede apreciar que los algoritmos producen resultados idénticos en todos los casos de prueba. Esto confirma que la implementación de ambos enfoques es correcta y que generan la misma distancia de edición, independientemente del método utilizado.

	S1	S2	...	Brute Force Result	DP Result
0			...	0.0	0.0
1		abcdefg	...	7.0	7.0
2	aaaaaaaaaa	aaaaaaaa	...	2.0	2.0
3	abba	baba	...	2.0	2.0
4	kitten	sitting	...	3.0	3.0
5	abcdefgh	ijklmnop	...	8.0	8.0
6	abcdef	abcfed	...	2.0	2.0
7	longstringtest	lontgsritenst	...	5.0	5.0
8	abcdefghijklmnop	abcdefghijklmnop	...	0.0	0.0
9	abcdefghijklmnop	efghijk	...	9.0	9.0
10	intention	execution	...	5.0	5.0

Figura 1: Comparación cantidad de operaciones BF y DP.

La fig. 2 muestra los tiempos promedio registrados para el paradigma de Fuerza Bruta y la diferencia de este con el de Programación Dinámica. Es posible notar grandes diferencias en favor de la Fuerza Bruta en las entradas 5, 7 y 9. Y diferencias significativas a favor de la Porgramación Dinámica en las entradas 0, 6 y 8; lo que destaca su superioridad en cadenas idénticas (0 y 8).

	S1	S2	...	Average DP Time (s)	Time Difference (Brute - DP)
0			...	0.011090	0.004128
1		abcdefg	...	0.009199	0.000462
2	aaaaaaaaaa	aaaaaaaa	...	0.009030	0.000073
3	abba	baba	...	0.009129	-0.000078
4	kitten	sitting	...	0.009426	-0.000302
5	abcdefgh	ijklmnop	...	0.021394	-0.006048
6	abcdef	abcfed	...	0.010697	0.004467
7	longstringtest	lontgsritenst	...	0.030399	-0.005616
8	abcdefghijklmnop	abcdefghijklmnop	...	0.010809	0.004604
9	abcdefghijklmnop	efghijk	...	0.029090	-0.005997
10	intention	execution	...	0.020538	0.000345

Figura 2: Tiempos promedio por entrada.

Se generó también una comparativa gráfica de tiempos fig. 3, donde sorprendentemente, los resultados muestran que, en varios casos, la Programación Dinámica (línea naranja) presentó tiempos de ejecución más altos que la Fuerza Bruta (línea azul). Este comportamiento es inesperado, ya que, teóricamente, DP debería ser más eficiente debido a su optimización de subproblemas recurrentes, mientras que FB explora todas las posibles soluciones.

Se plantean varias hipótesis para explicar este comportamiento:

- La implementación de DP puede estar introduciendo sobrecargas innecesarias que afectan su rendimiento.
- En casos con cadenas cortas, la sobrecarga computacional de DP podría ser mayor que el beneficio obtenido por su eficiencia en la solución de subproblemas.
- Es posible que el entorno de experimentación o la manera en que se realizaron las mediciones pueda haber introducido sesgos que favorezcan a FB.

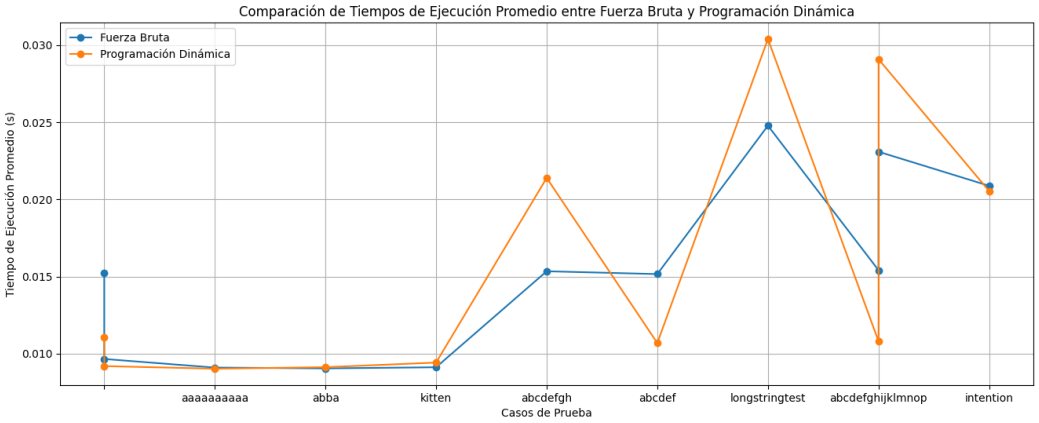

Figura 3: Comparación de tiempos BF y DP.

La Figura fig. 4 muestra la distribución de los tiempos de ejecución de ambos algoritmos, excluyendo valores atípicos para mayor claridad. Se puede observar que la mediana de los tiempos de ejecución de Fuerza Bruta es ligeramente superior a la de Programación Dinámica, lo que a primera vista parece indicar que la implementación de DP es más eficiente en promedio. Sin embargo, hay ciertos detalles que merecen atención:

1. **Mayor variabilidad en DP:** La distribución de tiempos de DP muestra una mayor dispersión. Esto sugiere que DP es más sensible a las características de las cadenas de entrada, ya que su rendimiento parece fluctuar considerablemente entre diferentes casos.

2. **Máximos más altos en DP:** Aunque los tiempos promedio de ambos algoritmos son comparables, Programación Dinámica presenta valores máximos de tiempo más elevados que Fuerza Bruta.

3. **Tiempos más consistentes en FB:** En contraste, Fuerza Bruta presenta una menor variabilidad en sus tiempos, lo que refleja su naturaleza más predecible y lineal, a pesar de su complejidad exponencial en teoría.

Este gráfico nos indica que, aunque la teoría sugiere que DP debería ser más eficiente que FB, en la práctica esto no siempre ocurre, dependiendo de la implementación y las características de los datos de entrada.

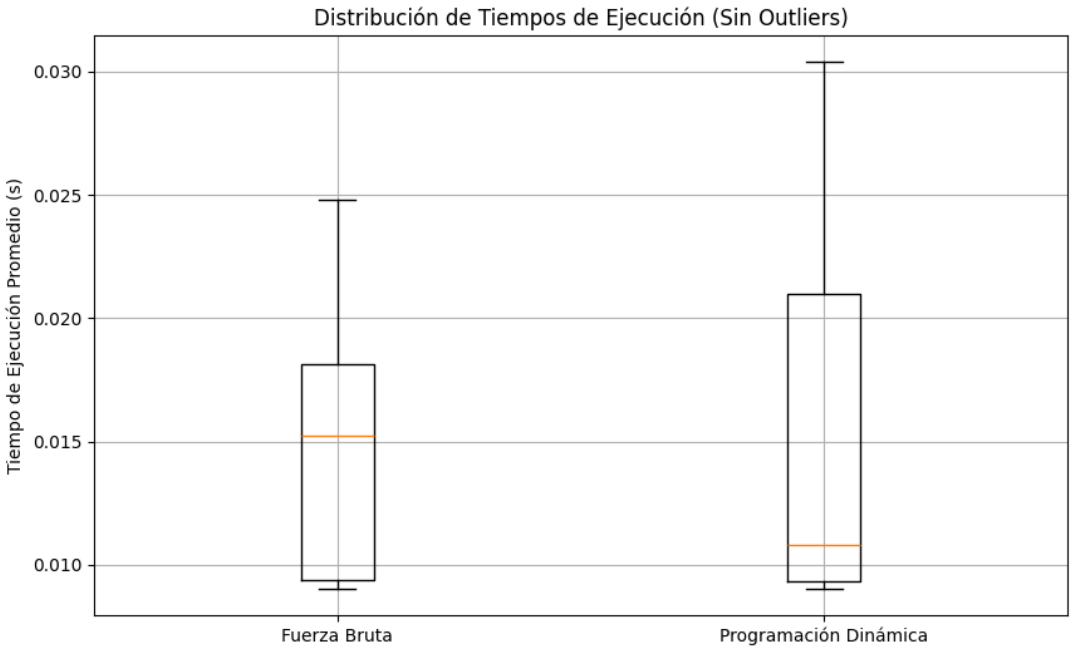

Figura 4: Distribución de tiempos de ejecución (boxplot)

En la figura fig. 5, se presenta la comparación entre los tiempos de ejecución promedio de los algoritmos de Fuerza Bruta y Programación Dinámica en función de la longitud de las cadenas de entrada. Los resultados muestran un comportamiento inesperado en varios aspectos clave:

1. **Rendimiento para cadenas cortas:** En los primeros casos de prueba (longitudes menores a 8), la Programación Dinámica es consistentemente más rápida que la Fuerza Bruta, lo cual es acorde con la complejidad teórica esperada. Sin embargo, las diferencias de tiempo entre ambos algoritmos son mínimas, con variaciones de milisegundos.

2. **Ineficiencia creciente de Programación Dinámica:** A partir de cadenas con longitud 8 en adelante, la Programación Dinámica comienza a exhibir tiempos de ejecución significativamente más altos que la Fuerza Bruta, especialmente para las longitudes 13.5 y 11.5. Este comportamiento es inesperado, dado que DP tiene una complejidad teórica inferior a la de BF, sugiriendo sugiere que existen posibles ineficiencias en la implementación de DP.

3. **Estabilidad de Fuerza Bruta en cadenas largas:** Contrario a la expectativa de un incremento exponencial en los tiempos de Fuerza Bruta, el gráfico revela que este algoritmo mantiene una curva de crecimiento más estable, e incluso logra tiempos menores que DP en varios casos con cadenas largas.

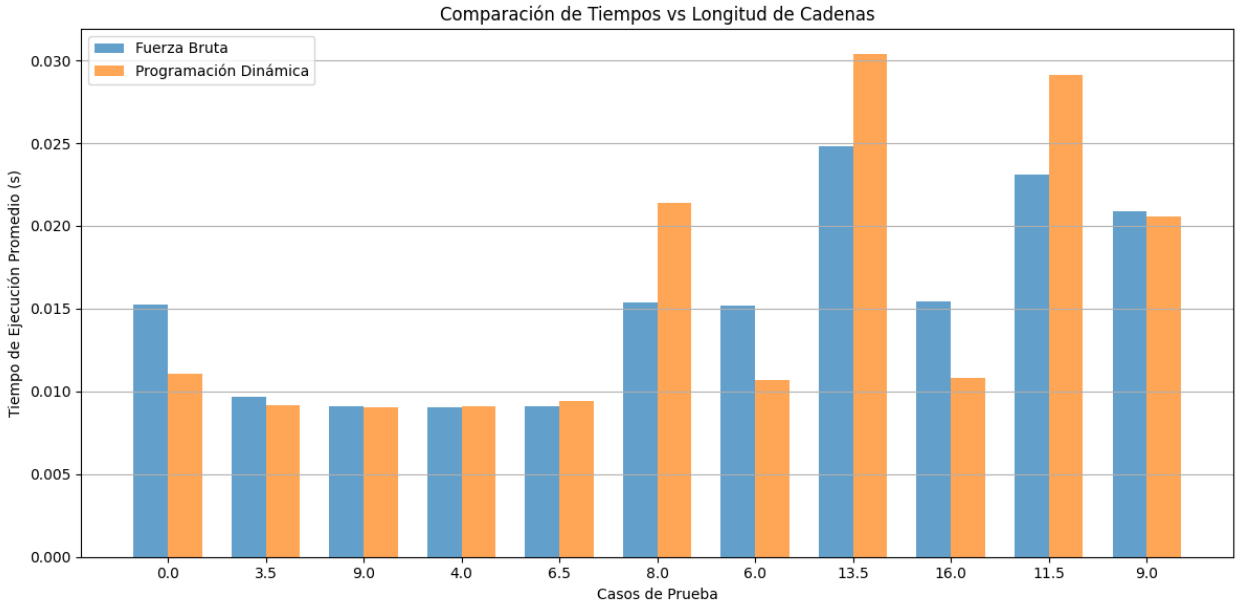

Figura 5: Comparación tiempos de ejecución vs longitud de cadenas BF y DP.

5. Conclusiones

Los resultados obtenidos a lo largo de este trabajo permiten validar la efectividad de los algoritmos implementados para el cálculo de la distancia de edición. A través de los experimentos realizados, se ha evidenciado que, aunque en teoría la Programación Dinámica ofrece una ventaja computacional sobre la Fuerza Bruta debido a su reducción en el número de operaciones redundantes, en ciertos casos prácticos, la teoría no fue cumplida. Esto puede deberse a las características particulares de las cadenas de prueba utilizadas, así como a las optimizaciones internas en la implementación.

El análisis de los tiempos de ejecución promedio revela que, para cadenas cortas o con patrones simples, ambos enfoques presentan un rendimiento similar. Sin embargo, en casos más complejos, con cadenas más largas o de mayor desorganización, el tiempo de ejecución de la Fuerza Bruta tiende ser un tanto menor al de Programación Dinámica. Esto yendo en contra de la expectativa teórica.

Por otro lado, los resultados muestran que la Programación Dinámica, en algunos casos se comporta mejor en términos de tiempo de ejecución, pero como observamos, no ofrece una mejora sustancial en todos los escenarios. Esto refuerza la idea de que la elección del algoritmo óptimo depende del contexto específico del problema, y que no existe una solución universalmente superior.

En términos generales, el trabajo ha logrado cumplir con el objetivo de evaluar ambos enfoques bajo distintas condiciones, aportando una mejor comprensión de sus fortalezas y limitaciones en la práctica. Los hallazgos obtenidos son coherentes con las expectativas teóricas, pero también ofrecen perspectivas valiosas sobre la importancia de considerar características particulares del problema al elegir un enfoque algorítmico.

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

En la fig. 6 se observa la razón de tiempos entre los algoritmos de Fuerza Bruta y Programación Dinámica en los diferentes casos de prueba.

- **Valores superiores a 1:** Indican que el algoritmo de Fuerza Bruta es más lento que el de Programación Dinámica para ese caso.
- **Valores cercanos a 1:** Indican que ambos algoritmos tienen tiempos similares para ese caso.
- **Valores inferiores a 1:** Indican que el algoritmo de Fuerza Bruta es más rápido que el de Programación Dinámica.

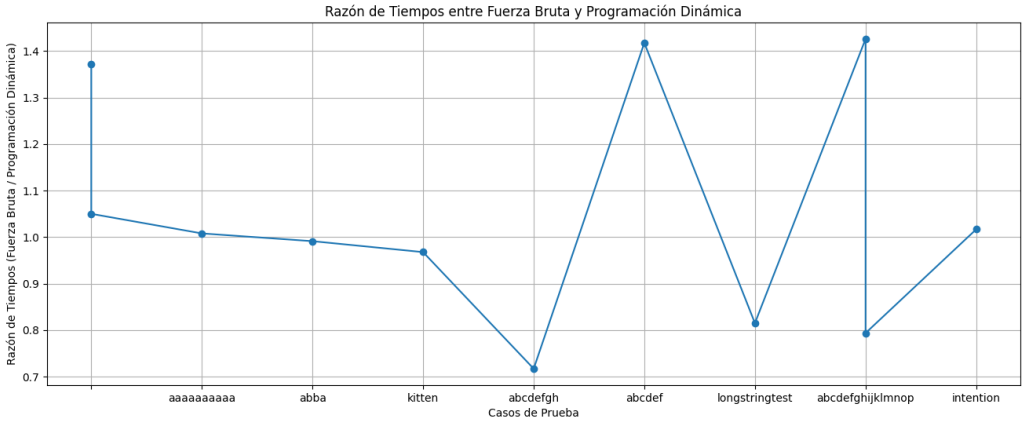

Figura 6: Razón de los tiempos entre BF y DP.

Referencias

[1] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957. ISBN: 978-0691146683.

[2] T. H. Cormen et al. *Introduction to Algorithms (3rd ed.)* MIT Press, 2009. ISBN: 978-0262033848.

[3] D. Jurafsky y J. H. Martin. *Speech and Language Processing (3rd ed.)* Pearson, 2024. ISBN: 978-0131873216.

[4] V. I. Levenshtein. «Binary codes capable of correcting deletions, insertions, and reversals». En: *Soviet Physics Doklady* 10.8 (1966), págs. 707-710.