REPORTE TAREA 2 y 3
ALGORITMOS Y COMPLEJIDAD

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

Joaquín Domínguez Bustos

20 de noviembre de 2024

23:42

Resumen

Este informe estudia la distancia de Damerau-Levenshtein utilizando algoritmos de fuerza bruta y programación dinámica, comparando sus eficiencias mediante experimentos en diferentes escenarios, y el impacto que tiene la busqueda de menores costos de edición en el rendimiento. Los objetivos incluyen analizar estas diferencias, optimizar los enfoques y evaluar su aplicabilidad. La infraestructura utilizada incluye hardware con procesador AMD Ryzen 5 4600H y herramientas como Python y C++ en un entorno WSL. Los resultados confirman la ventaja en eficiencia temporal de la programación dinámica, aunque con mayor uso de memoria, mientras que la fuerza bruta es útil en casos simples como cadenas vacías.

Índice

1. Introducción

En el ámbito de las Ciencias de la Computación, el análisis y diseño de algoritmos constituye un área fundamental, ya que establece las bases para resolver problemas complejos de manera eficiente. Uno de los problemas clásicos en este campo es la comparación de cadenas, esencial en aplicaciones como la corrección ortográfica, el reconocimiento de patrones y la bioinformática. La distancia de Damerau-Levenshtein, que mide el costo mínimo para transformar una cadena en otra mediante operaciones de inserción, eliminación, reemplazo y transposición, proporciona una métrica versátil y ampliamente utilizada.

A pesar de su relevancia, este problema plantea desafíos significativos en términos de eficiencia computacional, especialmente cuando se utilizan algoritmos de alta complejidad como la fuerza bruta. Las soluciones modernas, como las basadas en programación dinámica, prometen una mejora considerable, aunque con un costo adicional de memoria. Sin embargo, existe una brecha en la comprensión detallada del impacto de las operaciones individuales y los escenarios específicos sobre el rendimiento de estos algoritmos.

El propósito de este informe es estudiar y comparar ambos enfoques, fuerza bruta y programación dinámica, en el contexto del cálculo de la distancia de Damerau-Levenshtein. A través de un diseño detallado, implementaciones prácticas y experimentos controlados, se busca determinar cómo los costos asociados a cada operación afectan la complejidad temporal y espacial de las soluciones. Además, se evalúan casos particulares, como cadenas vacías o con caracteres repetidos, para identificar patrones de rendimiento y posibles áreas de optimización.

2. Diseño y Análisis de Algoritmos

- La solución para ambos algoritmos (algoritmo 1 y algoritmo 2) está basada en el uso de indices (i, j) con los cuales se recorren las cadenas $S1$ y $S2$ respectivamente para realizar el calculo de todas las posibles operaciones con las cuales se puede transformar la cadena $S1$ en la cadena $S2$ (eliminacion, inserción, reemplazo y transposición).

2.1. Fuerza Bruta

Algoritmo 1: Algoritmo de Fuerza Bruta para la Transformación de Cadenas

```
1  Procedure BF(S1, S2, i, j)
2      if i ≥ longitud(S1) and j ≥ longitud(S2) then
3          return 0
4      if i ≥ longitud(S1) then
5          return COSTO_INS(S2[j]) + BF(S1, S2, i, j+1)
6      if j ≥ longitud(S2) then
7          return COSTO_DEL(S1[i]) + BF(S1, S2, i+1, j)
8      elim ← COSTO_DEL(S1[i]) + BF(S1, S2, i+1, j)
9      ins ← COSTO_INS(S2[j]) + BF(S1, S2, i, j+1)
10     remp ← COSTO_SUB(S1[i], S2[j]) + BF(S1, S2, i+1, j+1)
11     transp ← ∞
12     if i + 1 < longitud(S1) and j + 1 < longitud(S2) and S1[i] = S2[j + 1] and S1[i + 1] = S2[j] then
13         transp ← COSTO_TRANS(S1[i], S1[i+1]) + BF(S1, S2, i+2, j+2)
14     return mín({elim, ins, remp, transp})
```

2.1.1. Ejemplo de ejecución

Consideremos las cadenas de longitud 3:

$$S_1 = \texttt{abc}, \quad S_2 = \texttt{adc}.$$

Supongamos que los costos de las operaciones son:

- costo_ins$(x) = 1$
- costo_del$(x) = 1$
- costo_sub$(x, y) = 1$ si $x \neq y$, 0 en caso contrario
- costo_trans$(x, y) = 1$

Paso a Paso de la Ejecución

1. Llamada inicial: $BF(\texttt{abc}, \texttt{adc}, 0, 0)$.
    - $S_1[0] = \texttt{a}, S_2[0] = \texttt{a}$. Como son iguales, el algoritmo calcula: remp $= 0 + BF(1, 1)$.

- También calcula: elim = $1 + BF(1,0)$ y ins = $1 + BF(0,1)$, mientras que no aplica transposición.

2. Segunda llamada: $BF(\texttt{abc}, \texttt{adc}, 1, 1)$.
   - $S_1[1] = \texttt{b}, S_2[1] = \texttt{d}$. Como son diferentes, el algoritmo calcula: remp = $1 + BF(2,2)$.
   - También calcula: elim = $1 + BF(2,1)$, ins = $1 + BF(1,2)$, mientras que no aplica transposición.
3. Tercera llamada: $BF(\texttt{abc}, \texttt{adc}, 2, 2)$.
   - $S_1[2] = \texttt{c}, S_2[2] = \texttt{c}$. Como son iguales, calcula: remp = $0 + BF(3,3)$.
   - También calcula: elim = $1 + BF(3,2)$ y ins = $1 + BF(2,3)$, mientras que no aplica transposición.
4. Llamada base: $BF(\texttt{abc}, \texttt{adc}, 3, 3)$.
   - Caso base: retorna 0 porque ambas cadenas han sido procesadas completamente.

Resultado

El costo mínimo para transformar S_1 en S_2 es 1. La única operación necesaria es reemplazar `b` por `d`.

2.1.2. Análisis de complejidad espacial y temporal

En cada llamada que se realiza se pueden generar hasta 4 subproblemas nuevos, así tambien es necesario recorrer cada cadena hasta el final de la misma mediante el uso de los indices (i, j), los cuales llegaran a un valor final (n, m) en donde n y m son los largos de $S1$ y $S2$ respectivamente, por lo que el número total de operaciones será aproximadamente $4^{\text{mín}(n,m)}$, dando así que la complejidad temporal pertence a $O(4^{\text{mín}(n,m)})$. Por otra parte, la complejidad espacial asociada a esta implementacion sería $O(1)$ puesto que no se utiliza ninguna estructura de datos adicional al tamaño de la entrada y para las llamadas recursivas se apoya en el stack.

2.2. Programación Dinámica

2.2.1. Descripción de la solución recursiva

Por cada llamada a la función es necesario evaluar los cuatro casos posibles (eliminacion, inserción, reemplazo y transposición) y escoger el de menor costo, si guardamos el costo asociado a esa llamada este nos puede servir para evitar realizar el mismo calculo si se da el caso, de esta manera se asegura que cada llamada que posea los mismos parametros se resuelva una única vez, esto implica que mediante la consulta recurrente de valores podemos disminuir el costo temporal del problema en comparación a la utilizacion de fuerza bruta y obtendremos exactamente la misma respuesta final.

2.2.2. Relación de recurrencia

- *(condicion)* Si el valor de $Cache[i][j]$ es distinto de -1 se retorna el valor de $Cache[i][j]$
- *(caso base)* Si el valor de i es igual al largo de $S1$ y el valor de j igual al largo de $S2$ entonces se retorna 0
- *(condicion)* Si el valor de i iguala el largo de $S1$ entonces solo se realizan inserciones por cada caracter faltante por cubrir de $S2$, devolviendo el costo de estas operaciones.
- *(condicion)* Si el valor de j iguala el largo de $S2$ entonces solo se realizan eliminaciones por cada caracter sobrante de $S1$, devolviendo el costo de estas operaciones.
- *(Recurrencia)*

$$DP(S1,S2,i,j) = \text{Cache}[i][j] = min \left\{ \begin{array}{l} \text{costo\_del}(S1[i]) + DP(S1,S2,i+1,j), \\ \text{costo\_ins}(S2[j]) + DP(S1,S2,i,j+1), \\ \text{costo\_sub}(S1[i],S2[j]) + DP(S1,S2,i+1,j+1), \\ \text{costo\_trans}(S1[i],S1[i+1]) + DP(S1,S2,i+2,j+2) \end{array} \right\}$$

2.2.3. Identificación de subproblemas

Los subproblemas son las combinaciones de las posiciones i y j que representan el costo mínimo para transformar la subcadena de $S1$ desde i a la subcadena de $S2$ desde j.

2.2.4. Estructura de datos y orden de cálculo

Se utiliza el enfoque top-down una tabla de `Cache` para almacenar los resultados, que son calculados recursivamente, garantizando que cada subproblema solo se evalúe una vez.

2.2.5. Algoritmo utilizando programación dinámica

Algoritmo 2: Algoritmo de Programación Dinámica para la Transformación de Cadenas

Input: Cadenas $S1$, $S2$, índices i, j, y matriz de cache de tamaño (n+1) × (m+1) inicializada en -1

```
Procedure DP(S1, S2, i, j, Cache)
    if Cache[i][j] ≠ −1 then
        return Cache[i][j]
    if i = largo(S1) and j = largo(S2) then
        return Cache[i][j] = 0
    if i = largo(S1) then
        return Cache[i][j] = costo_ins(S2[j]) + DP(S1, S2, i, j + 1, Cache)
    if j = largo(S2) then
        return Cache[i][j] = costo_del(S1[i]) + DP(S1, S2, i + 1, j, Cache)
    elim ← costo_del(S1[i]) + DP(S1, S2, i + 1, j, Cache);
    ins ← costo_ins(S2[j]) + DP(S1, S2, i, j + 1, Cache);
    remp ← costo_sub(S1[i], S2[j]) + DP(S1, S2, i + 1, j + 1, Cache);
    transp ← ∞;
    if i + 1 < largo(S1) and j + 1 < largo(S2) and S1[i] = S2[j + 1] and S1[i + 1] = S2[j] then
        transp ← costo_trans(S1[i], S1[i + 1]) + DP(S1, S2, i + 2, j + 2, Cache)
    return Cache[i][j] = mín(elim, ins, remp, transp)
```

2.2.6. Ejemplo de ejecución

Para ilustrar la ejecución del algoritmo de programación dinámica, consideremos las siguientes cadenas de largo 3:

$$S_1 = \texttt{abc}, \quad S_2 = \texttt{adc}$$

Supongamos que los costos de las operaciones son:

- costo_ins$(x) = 1$
- costo_del$(x) = 1$
- costo_sub$(x, y) = 1$ si $x \neq y$, 0 en caso contrario
- costo_trans$(x, y) = 1$

Inicializamos una matriz de caché de dimensiones 4×4 (longitudes de S_1 y S_2 más 1), con todos sus valores en -1, indicando que aún no se han calculado.

Paso a Paso de la Ejecución:

1. Llamada inicial: $DP(\texttt{abc}, \texttt{adc}, 0, 0, \text{Cache})$.
    - $S_1[0] = \texttt{a}, S_2[0] = \texttt{a}$. Como son iguales, el algoritmo calcula remp $= 0 + DP(1, 1)$.

- También calcula elim = $1 + DP(1,0)$ y ins = $1 + DP(0,1)$, mientras que no aplica transposición porque $S_1[0] \neq S_2[1]$.

2. Segunda llamada: $DP(\texttt{abc}, \texttt{adc}, 1, 1)$.
   - $S_1[1] = \texttt{b}, S_2[1] = \texttt{d}$. Como son diferentes, calcula remp = $1 + DP(2,2)$.
   - También calcula elim = $1 + DP(2,1)$ y ins = $1 + DP(1,2)$, mientras que no aplica transposición.
3. Tercera llamada: $DP(\texttt{abc}, \texttt{adc}, 2, 2)$.
   - $S_1[2] = \texttt{c}, S_2[2] = \texttt{c}$. Como son iguales, calcula remp = $0 + DP(3,3)$.
   - También calcula elim = $1 + DP(3,2)$ y ins = $1 + DP(2,3)$, mientras que no aplica transposición.
4. Llamada base: $DP(\texttt{abc}, \texttt{adc}, 3, 3)$.
   - Caso base: retorna 0 porque ambas cadenas han sido procesadas completamente.

Matriz de Caché al Final:

$$\text{Cache} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 1 & 2 & -1 \\ 2 & 2 & 1 & -1 \\ 3 & -1 & -1 & 0 \end{bmatrix}$$

Resultado

El costo mínimo para transformar S_1 en S_2 es 1. La única operación necesaria es reemplazar `b` por `d`.

2.2.7. Análisis de complejidad espacial y temporal

La implementación de programación dinámica utiliza una tabla de caché para almacenar el costo mínimo de edición para cada par (i, j), permitiendo así que cada estado (eliminación, inserción, sustitución y transposición) realice un número constante de operaciones implicando un costo temporal $O(1)$ ya que solo consideramos la consulta al costo, aunque, en el peor caso, será necesario llenar por completo la tabla de caché, la cual tiene dimensiones $(n+1) \times (m+1)$, donde n y m son los largos de $S1$ y $S2$, respectivamente. Esto significa que hay $O(n \times m)$ estados posibles, y cada uno se evaluará a lo sumo una vez, por lo que la complejidad temporal es $O(n \times m)$. En cuanto al análisis espacial, el uso de la tabla de caché la que utiliza mayor espacio, lo que implica que la complejidad espacial también es $O(n \times m)$.

3. Implementaciones

A continuacion se detallan los archivos utilizados y el contenido de cada uno, además se puede encontrar el link al repositorio de Github el cual puede ser revisado en el apendice A.

- **Archivos .cpp y .hpp**
  - **DP.cpp:** Algoritmo de programacion dinamica.
  - **BF.cpp:** Algoritmo de fuerza bruta.
  - **utils.hpp:** Funcion de lectura de archivos para los costos, funciones de costos, funcion de tamaño espacial, funcion de reconstruccion.
- **Generadores**
  - **GeneradorData.py:** Genera los datasets a utilizar.
  - **GenerarCostos.py:** Genera los costos de las operaciones de insercion, eliminacion, reemplazo y transposicion.
  - **Graficos.py:** Genera los graficos con los resultados almacenados en BF.txt y DP.txt.
- **Archivos .txt**
  - **cost_operacion.txt:** estos archivos contienen las matrices de costos generadas.
  - **DP.txt y BF.txt:** contienen los datos de salida de cada ejecucion
  - **(datasets).txt:** estos archvos contienen los casos de prueba utilizados.
- **Carpetas**
  - **Codigos:** Se tienen los .cpp junto con los costos y los datasets en archivos .txt
  - **Generacion:** Contiene los generadores

4. Experimentos

Todos los experimentos se desarrollaron en el siguiente ambiente computacional:

- **Hardware**
  - **Procesador:** AMD Ryzen 5 4600H 3.0GHz
  - **RAM:** 16GB Dual Channel 3200 MT/s DDR4 SODIMM
  - **Almacenamiento:** 500GB SSD NVMe
- **Software**
  - **SO:** Windows 11 v10.0.22631.4460
  - **Entorno:** WSL v2.3.26.0 con Ubuntu 14.2.0 y g++ 14.2.0
  - **Python:** v3.12.3 con liberías pandas, matplotlib.pyplot, numpy, pathlib, random.
- **Consideraciones**
  - Se utilizó el siguiente comando para la compilacion: g++ -O3 (.cpp) -o (ejecutable).
  - Los datasets fueron ingresados mediante entrada estandar.
  - Los archivos de resultados finales que se utilizaron para la creacion de los graficos son una recopilacion manual de los archivos generados por los algoritmos.

4.1. Dataset (casos de prueba)

- **1. IgualLargo.txt**
  - Este archivo contiene pares de cadenas de igual longitud. Cada par está identificado por un número que indica la longitud de las cadenas, seguido de las cadenas mismas.
  - La utilizacion de este dataset se utiliza a modo de estudiar como varia el tiempo de ejecucion de ambos algoritmos sobre casos aleatorios.
- **2. LetrasRepetidas.txt**
  - Este archivo incluye pares de cadenas donde los caracteres de cada cadena son iguales y repetidos.
  - Este dataset se utiliza para estudiar el peor caso posible teoricamente, en el cual se pueden realizar todas las operaciones en cada llamada
- **3. Transposiciones.txt**
  - Este archivo contiene pares de cadenas que difieren únicamente en una o más transposiciones de caracteres consecutivos.

- Este dataset se utiliza a modo de estudiar la complejidad adicional que proporcionan las transposiciones y la optimización del costo.

- **4. VacioS1.txt y VacioS2.txt**

  - Estos archivos incluyen cadenas donde una de las cadenas está vacía, mientras que la otra tiene diferentes longitudes.
  - Permite estudiar casos donde todas las operaciones son solamente inserciones o eliminaciones.

4.2. Resultados

A continuacion se discuten los resultados obtenidos por cada dataset

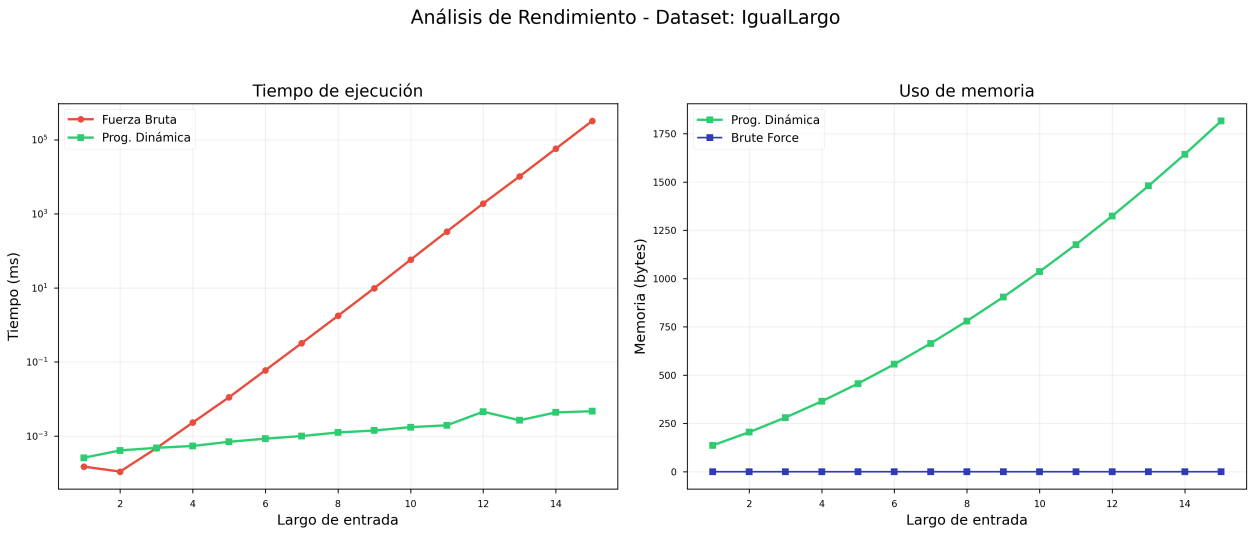

Figura 1: Cadenas aleatorias de largo creciente

En la figura 1, en el grafico de la iquierda se puede observar la gran diferencia en tiempo de ejecucion de ambos algoritmos a medida que el tamaño de la entrada crece, aunque la menor complejidad temporal que maneja programacion dinamica es a costa de una mayor complejidad espacial. El estudio de estos datos es importante puesto que es necesario comprender eperimentalmete que tan sensibles son los algoritmos implementados al tamaño de la entrada

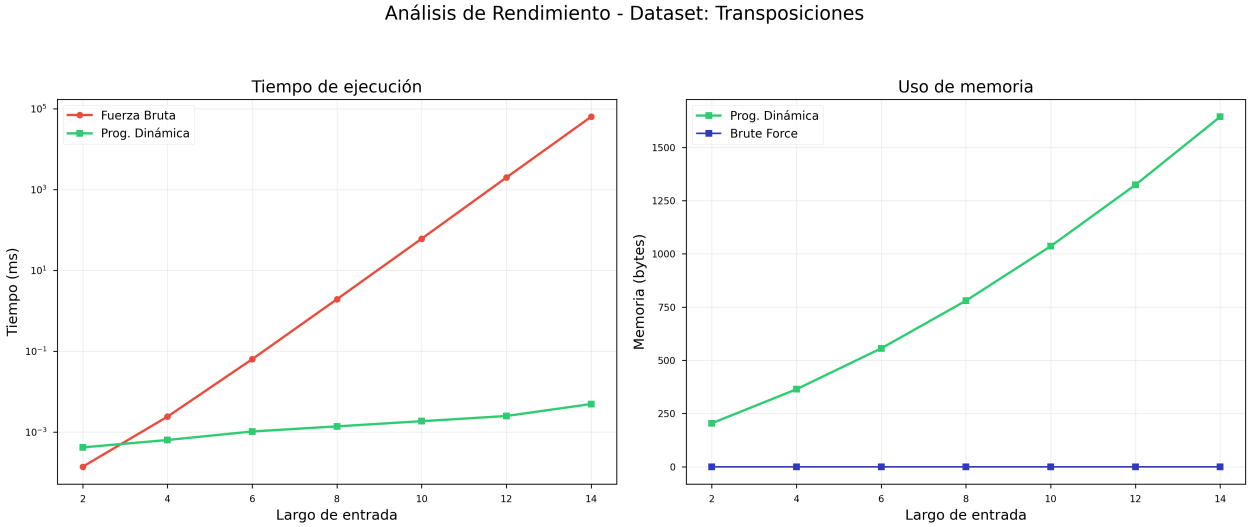

Figura 2: Cadenas de pares transpuestos

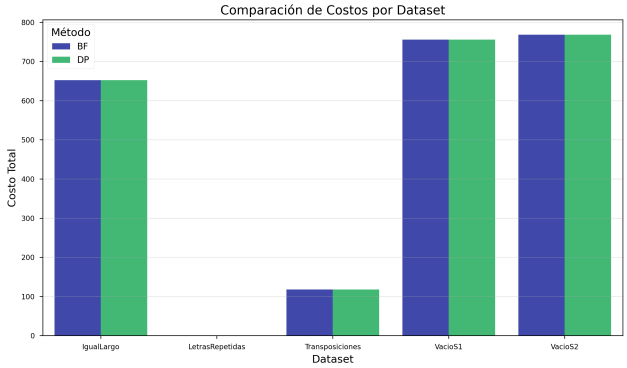

Figura 3: Costos de edicion

Analizando las figuras 2 y 3 se puede observar que las transposiciones juegan un papel relevante dentro de este estudio, puesto que si se busca disminuir el costo que toma editar 2 cadenas estas son una buena opcion a considerar, sin embargo, como ya se ha dicho en la explicacion teorica, estas transposiciones incrementan la complejidad temporal del problema.

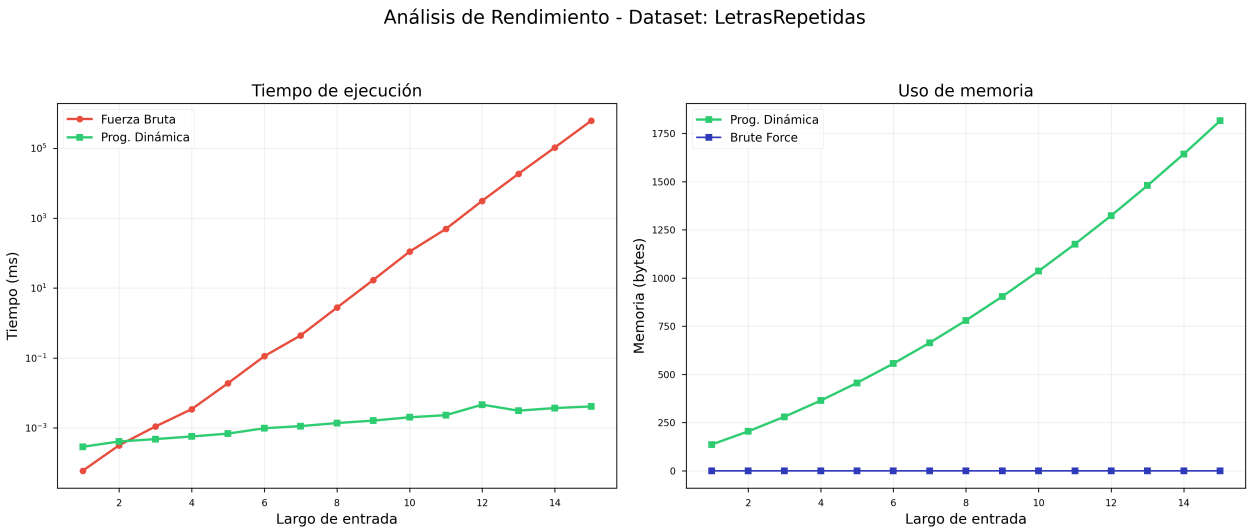

Figura 4: Cadenas de letras repetidas de largo creciente

La figura 4 se asemeja mucho a la figura 1, mas no son lo mismo, en la figura 4 se observa el peor caso posible para el caso de fuerza bruta, puesto que todas las operaciones son realizables en todas las llamadas recursivas, por lo que la recursion alcanza una profundida mucho mayor que en cualquier otro caso. Esto no solo afecta a fuerza bruta, puesto que en el enfoque de programacion dinamica tambien se debe relizar una mayor cantidad de operaciones a comparacion de otros casos, pero debido a que cada operacion se ejecuta solo una vez para el par (i, j) el tiempo de resolucion sigue siendo bajo

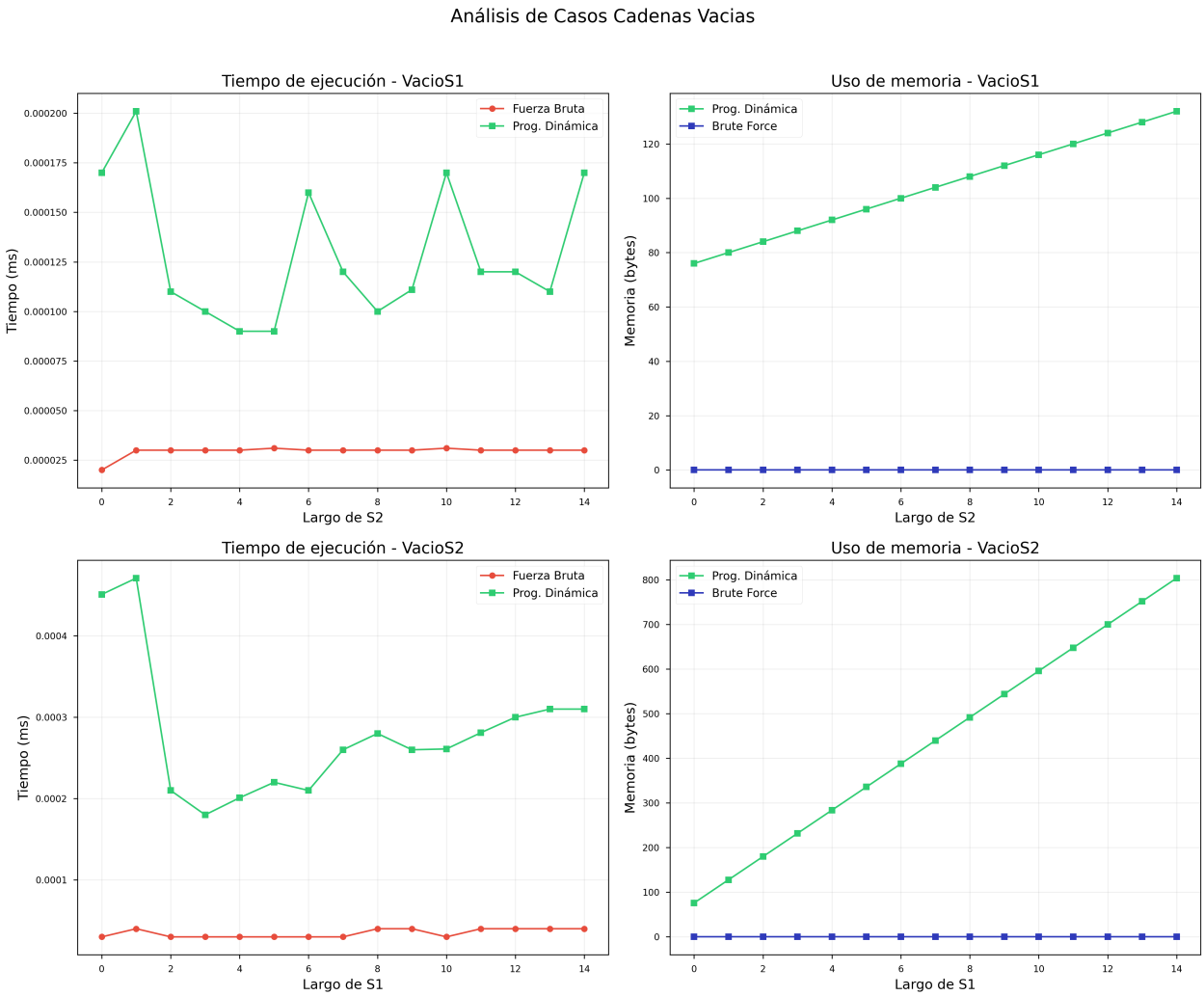

Figura 5: Casos Cadenas Vacias

En la figura 5, en contraste a la figura 4, se puede observar el mejor caso posible para fuerza bruta, puesto que no todas las operaciones son relizables, sino solamente inserciones o eliminaciones constantes y no llega nunca a ser exponencial, en contraste tenemos al enfoque de programacion dinamica, el cual se demora más debido al uso de una matriz de cache, la cual se debe inicializar y posteriormente resolver el problema.

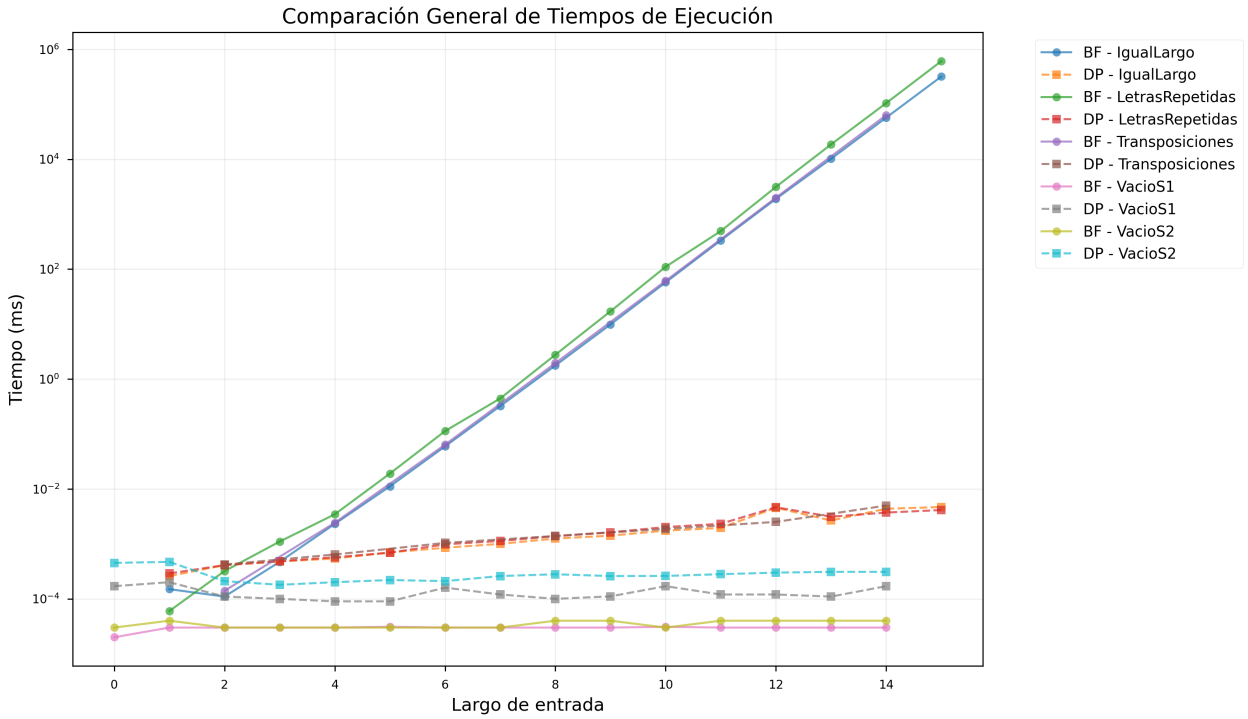

Figura 6: Resumen general de resultados

En la figura 6 se observa un resumen general de todos los casos anteriores, del cual se extrae que si queremos resolver el problema de la menor distancia es importante tener en cuenta que el enfoque de fuerza bruta para cadenas vacias puede llegar a ser util, sin embargo el enfoque de programacion dinamica es una buena opcion general.

5. Conclusiones

Los resultados obtenidos destacan la relevancia de seleccionar el enfoque algorítmico adecuado según las características del problema planteado. La programación dinámica, al reducir cálculos redundantes mediante el almacenamiento de resultados intermedios, demuestra ser eficaz para escenarios con grandes volúmenes de datos o configuraciones complejas. Por otro lado, la fuerza bruta, pese a su alta complejidad, se mantiene como una herramienta competitiva en casos simplificados donde las restricciones de memoria son críticas. Así tambien discernir la inclusion de transposiciones depende expclusivamente de lo que se busque, si es prioritario minimizar los costos de edicion son una buena opcion, sin embargo si solo se buscan soluciones factibles su inclusion es presindible.

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

Github