REPORTE TAREA 2 y 3
ALGORITMOS Y COMPLEJIDAD

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

Geraldine Cornejo Valenzuela

18 de noviembre de 2024

00:24

Resumen

Estudiamos el problema de la distancia de edición extendida entre cadenas, un problema interesante con aplicaciones en el procesamiento de texto y la bioinformática. Comparamos la fuerza bruta con la programación dinámica con respecto al costo mínimo de transformación. La fuerza bruta tiene una complejidad exponencial que resulta de una combinación a vista general de todas las posibles operaciones de secuencia, mientras que la programación dinámica almacena soluciones de subproblemas. El método de fuerza bruta es impráctico para entradas grandes.

Índice

1. Introducción

En el campo de las Ciencias de la Computación, el Análisis y Diseño de Algoritmos es una disciplina fundamental que proporciona las herramientas teóricas y prácticas para abordar problemas complejos de manera eficiente. Uno de los problemas más estudiados en dicho campo es el cálculo de distancia de edición, consiste en un indicador que mide la similitud entre dos secuencias de carácteres, la cuál ha tenido un impacto importante en aplicaciones como:

- Procesamiento de texto: Ayudando a corregir errores tipográficos y comparar similitudes entre palabras o frases.
- Bioinformática: Se analiza la similitud entre secuencias de ADN o proteínas para determinar relaciones evolutivas.
- Aprendizaje automático: Se evalúa la calidad de predicciones en tareas de generación de texto o traducción automática.
- Recuperación de información: Mejora la búsqueda al encontrar documentos relevantes incluso si contienen errores tipográficos o variaciones.

El problema tradicional es presentado con un entrada de dos secuencias de carácteres S1 y S2, y el algoritmo ejecuta tres operaciones:

- Inserción: Se inserta un carácter en S1 para acercarse a S2.
- Eliminación: Se elimina un carácter de S1 para acercarse a S2.
- Sustitución: Se sustituye un carácter S1 por el carácter comparado en S2.

En esta instancia los costos de cada operación son fijos, es decir, independiente del carácter a alterar, se tiene el mismo costo. Este varía solo dependiendo de la operación.

En esta entrega se introduce una variación al problema, donde se tiene que los costos pueden ser variables dependiendo del caracter, por ejemplo, el costo de insertar una 'a' o insertar una 'b' puede no ser el mismo. Además, se agrega una nueva operación:

- Transposición: Se intercambian carácteres adyacentes de S1, de modo que al realizar la operación, ambos caráceres coincidan con S2

Esta extensión vuelve el problema más realista en los casos de usos explicados anteriormente, pero al mismo tiempo aumenta la complejidad del cálculo. Esto será lo estudiado a lo largo del informe mediante dos enfoques.

La implementación del problema mendiante el método de **fuerza bruta** explora todas las posibles combinaciones de operaciones, retornando la mínima distancia de edición de las dos secuencias. Este acercamiento garantiza encontrar una solución óptima, pero debido a la exploración exhaustiva que

realiza, su ejecución no es óptima, ya que su costo computacional crece de manera exponencial y no es práctico para cadenas de mayor extensión.

Por otro lado, el método de **programación dinámica** divide el problema en subproblemas de menor tamaño, utilizando una tabla para almacenar soluciones parciales que vuelven a ser visitadas posteriormente y de esta forma evitar cálculos rebundantes. Con este enfoque se reduce la complejidad de exponencial a O(n*m), lo que lo hace mucho más eficiente.

El propósito de este informe es evalúar experimentalmente el rendimiento de cada enfoque, identificando ventajas y limitaciones de cada uno. Para esto se interpretará y compararán los algoritmos de **Buerza Bruta y Programación Dinámica** para calcular la distancia de edición extendida. De esta forma, se busca evaluar su eficiencia en términos de *complejidad temporal, complejidad espacial y manejo de costos variables y transposiciones.*

La importacia de llevar a cabo este análisis es poder demostrar cómo las decisiones algorítmicas afectan la resolución de problemas.

2. Diseño y Análisis de Algoritmos

A continuación se explican los métodos de Fuerza Bruta y Programación Dinámica, como fueron diseñados y analizando su complejidad tanto espacial como temporal, además se reflexiona como la inclusión de la operación transposición y los costo variables afectan la complejidad de los algoritmos.

2.1. Fuerza Bruta

2.1.1. Descripción de la solución diseñada

El enfoque de fuerza brua explora todas las posibles combinaciones de operaciones para transformar una cadena en otra, considerando las operaciones de inserción, eliminación, sustitución y transposición.

El algoritmo recursivo evalúa cada una de las operaciones a medida que avanza en las cadenas, eligiendo la que minimice el costo acumulado. El algoritmo avanza en la recursión hasta que las cadenas sean iguales o una de ellas queda vacía.

Descripción del flujo de ejecución:

1. Si las cadenas son iguales ($S1 == S2$), la distancia de edición es cero.
2. i una de las cadenas está vacía, el costo es la suma de los costos de insertar o eliminar todos los carácteres de la otra cadena.
3. Si las dos cadenas no son iguales, se calcula el costo de las cuatro operaciones recursivamente y se selecciona la operación con el costo mínimo.

2.1.2. Impacto de las Transposiciones y Costos Variables

La inclusión de **transposiciones** permite reducir el costo en situaciones en las que dos caracteres adyacentes en una cadena se encuentran en el orden inverso en la otra cadena, lo que permite optimizar la distancia de edición. Esto es especialmente útil en casos donde, de no haber transposición, tendríamos que realizar dos operaciones (una sustitución y una inserción/eliminación). Con esta operación se agrega un camino adicional de búsqueda recursiva. Si bien, su inclusión en el problema co aumenta significativamente la complejidad en términos de crecimiento exponencial, sí puede afectar el número de llamadas recursivas evaluadas. Esto puede aumentar ligeramente el tiempo de ejecución comparado con el algoritmo que solo considere el problema tradicional.

Los **costos variables** para las operaciones hacen que el algoritmo sea más flexible, ya que no todos los cambios entre caracteres tienen el mismo costo. En el código, estos costos son proporcionados mediante archivos externos, lo que permite ajustar el algoritmo a diferentes escenarios o dominios. La inclusión de costos variables permite que el algoritmo sea más preciso en cuanto a la asignación de recursos para cada operación, pero también significa que el algoritmo debe realizar más cálculos para determinar el costo exacto de cada operación en lugar de asumir costos constantes. Sin embargo, los costos variables no afectan la complejidad temporal en términos de crecimiento exponencial, ya que solo agregan una operación de suma y comparación adicional en cada paso recursivo.

2.1.3. Complejidad Temporal

El algoritmo tiene una complejidad exponencial en términos de la longitud de las cadenas de entrada. Esto debido a que, en cada llamada recursiva, el algoritmo evalúa 4 posibles operaciones y vuelve a realizar una llamada recursiva para cada una de estas operaciones.

En el peor caso, el número total de llamadas recursivas es de $4^{min(m,n)}$, donde m es la longitud de la cadena **S1** y n es la longitud de la cadena **S2**. Esto hace que el algoritmo sea ineficiente para cadenas largas, ya que su tiempo de ejecución crece rápidamente con el tamaño de las cadenas.

- **Complejidad Temporal:** $T(m,n) \in O(4^{min(m,n)})$

2.1.4. Complejidad Espacial

La complejidad espacial está determinada principalmente por el uso de la pila de recursión. Dado que cada llamada recursiva genera una nueva instancia de la función, el tamaño máximo de la pila de recursión es $O(min(m,n))$.

- **Complejidad Espacial:** $O(min(m,n))$

2.1.5. Algoritmo utilizando fuerza bruta

Algoritmo 1: Distancia de edición mínima mediante fuerza bruta.

```
Procedure DistanciaFuerzaBruta(S1, S2)
    if S1 es igual a S2 then
        return 0
    if S1 está vacía then
        costo ← 0
        for cada carácter c en S2 do
            costo ← costo + costo_ins(c)
        return costo
    else if S2 está vacía then
        costo ← 0
        for cada carácter c en S1 do
            costo ← costo + costo_del(c)
        return costo
    else
        minCosto ← ∞
        insertar ← DistanciaFuerzaBruta(S1, Subcadena(S2, 1, Longitud(S2))) + costo_ins(PrimerCarácter(S2))
        minCosto ← mín(minCosto, insertar)
        eliminar ← DistanciaFuerzaBruta(Subcadena(S1, 1, Longitud(S1)), S2) + costo_del(PrimerCarácter(S1))
        minCosto ← mín(minCosto, eliminar)
        sustituir ← DistanciaFuerzaBruta(Subcadena(S1, 1, Longitud(S1)), Subcadena(S2, 1, Longitud(S2))) +
          costo_sub(PrimerCarácter(S1), PrimerCarácter(S2))
        minCosto ← mín(minCosto, sustituir)
        if Longitud(S1) >1 y Longitud(S2) >1 y (SegundoCarácter(S1) = PrimerCarácter(S2) y PrimerCarácter(S1) =
          SegundoCarácter(S2)) then
            transponer ←
              DistanciaFuerzaBruta(Subcadena(S1, 2, Longitud(S1)), Subcadena(S2, 2, Longitud(S2))) +
              costo_trans(PrimerCarácter(S1), SegundoCarácter(S1))
            minCosto ← mín(minCosto, transponer)
        return minCosto
```

2.2. Programación Dinámica

2.2.1. Descripción de la solución recursiva

La solución recursiva calcula la distancia de edición mínima entre dos cadenas **S1** y **S2** considerando los siguientes casos:

1. **Casos base:**
   - S1 está vacía, distancia de edición es el costo de insertar todos los caracteres restantes de S2.
   - S2 está vacía, distancia de edición es el costo de eliminar todos los caracteres restantes de S1.
2. **Caso general:**

- Si el carácter actual de S1 coincide con el carácter actual de S2, se procede al siguiente par de caracteres sin costo adicional.
- Si no coinciden, se consideran las siguientes operaciones y sus costos:
  - Sustitución: Cambiar el carácter actual de S1 para que coincida con el carácter actual de S2.
  - Inserción: Agregar el carácter actual de S2 a S1.
  - Eliminación: Eliminar el carácter actual de S1.
- Además, se considera una operación especial de transposición si los caracteres adyacentes en ambas cadenas están invertidos.

2.2.2. Relación de recurrencia

La relación de recurrencia del algoritmo de distancia de edición es una forma de resolver el problema recursivamente utilizando los casos base y las operaciones de transformación de las cadenas. En términos más simples: si los caracteres coinciden, no se hace ninguna operación adicional. Si no coinciden, se evalúan tres operaciones posibles: sustitución, inserción y eliminación, y se elige la de menor costo. Además, si se cumplen las condiciones, también se evalúa la transposición de caracteres adyacentes.

La relación de recurrencia aprovecha los subproblemas para evitar el recalculo repetido, lo que hace que el algoritmo sea mucho más eficiente que el enfoque de fuerza bruta.

2.2.3. Identificación de subproblemas

Los subproblemas consisten en calcular la distancia de edición mínima para diferentes pares de prefijos de las cadenas **S1** y **S2**. Especificamente:

- **Dist(i,j):** La distancia de edición entre las primeras *i* letras de *S1* y las primeras *j* letras de *S2*.
- Los subproblemas se solapan debido a que cada cálculo de *Dist(i,j)* depende de *Dist(i-1, j-1)* (caso de sustitución), *Dist(i, j-1)* (caso de insertar), *Dist(i-1, j)* (caso de eliminación) y *Dist(i-2, j-2)* (caso de transposición).

El número total de subproblemas es *(n+1)x(m+1)*, donde *n* y *m* son las longitudes de las cadenas.

2.2.4. Estructura de datos y orden de cálculo

Se define una matriz bidimensional de tamaño *(m+1)x(n+1)* donde *dp[i][j]* almacena el valor de *Dist(i,j)*.
El orden de cálculo está dado de la siguiente manera:

1. Inicializar las celdas correspondientes a los casos base:
   - **dp[0][j]**: costos de insertar carácteres en S1, dado que S1 es una cadena vacía.
   - **dp[i][0]**: costos de eliminar carácteres de S1, dado que S2 es una cadena vacía.

2. Llenar la matriz **dp** en orden creciente de tamaño de prefijos:
   - Procesar las filas (i de 1 a m).
   - Para cada fila, procesar las columnas (j de 1 a n).
3. Calcular los valores en función de la relación de recurrencia para cada celda.
4. El valor final *dp[m][n]* contiene la distancia de edición mínima.

2.3. Complejidad temporal

Cada celda de la matriz dp se calcula una vez, por lo que el tiempo total es **O(m x n)**, donde m y n son las longitudes de las cadenas.

2.4. Complejidad espacial

La matriz ocupa **O(m x n)** espacio en memoria, debido al tamaño de la tabla dp.

2.4.1. Algoritmo utilizando programación dinámica

Algoritmo 2: Cálculo de la distancia de edición mínima utilizando programación dinámica.

Procedure DISTANCIADP($S1$, $S2$)
- $n \leftarrow$ longitud de $S1$
- $m \leftarrow$ longitud de $S2$
- Crear una matriz dp de tamaño $(m+1) \times (n+1)$ con todos los valores inicializados a ∞
- **for** $i \leftarrow 0$ **to** m **do**
  - $dp[i][0] \leftarrow \begin{cases} 0 & \text{si } i = 0 \\ dp[i-1][0] + \_ins(S2[i-1]) & \text{si } i > 0 \end{cases}$
- **for** $j \leftarrow 0$ **to** n **do**
  - $dp[0][j] \leftarrow \begin{cases} 0 & \text{si } j = 0 \\ dp[0][j-1] + \_del(S1[j-1]) & \text{si } j > 0 \end{cases}$
- **for** $row \leftarrow 1$ **to** m **do**
  - **for** $col \leftarrow 1$ **to** n **do**
    - **if** $S1[col-1] = S2[row-1]$ **then**
      - $dp[row][col] \leftarrow dp[row-1][col-1]$
    - **else**
      - $sustituir \leftarrow dp[row-1][col-1] + \_sub(S1[col-1], S2[row-1])$
      - $insertar \leftarrow dp[row][col-1] + \_ins(S2[row-1])$
      - $eliminar \leftarrow dp[row-1][col] + \_del(S1[col-1])$
      - $dp[row][col] \leftarrow$ mín($\{sustituir, insertar, eliminar\}$)
    - **if** $row > 1$ **y** $col > 1$ **y** $S1[col-1] = S2[row-2]$ **y** $S1[col-2] = S2[row-1]$ **then**
      - $transponer \leftarrow dp[row-2][col-2] + \_trans(S1[col-2], S1[col-1])$
      - $dp[row][col] \leftarrow$ mín($dp[row][col], transponer$)
- **return** $dp[m][n]$

3. Implementaciones

El siguiente enlace redirige al repositorio con las implementaciones de los algoritmos en C++: Repositorio Distancia de Edición

En dicho repositorio se busca resolver el problema de cálculo de distancia de edición mínima entre dos cadenas, mediante **programación dinámica** y **fuerza bruta**. Ambos algoritms consideran las operaciones: *inserción, eliminación, sustitución y transposición* de carácteres, con costos específicos que se cargan desde archivos de texto y son procesados mediante funciones para cada operación.

Los archivos se organizan de la siguiente forma:

- **DPdistancia.cpp:** Implementa el algoritmo de programación dinámica para calcular la distancia de edición mínima. Contiene la función *DistanciaDP*, que toma dos cadenas y calcula la distancia utilizando una matriz de costos acumulados.
- **FBdistancia.cpp:** Implementa el algoritmo de fuerza bruta para calcular la distancia de edición. La función principal es *DistanciaFuerzaBruta*, que recursivamente evalúa todas las combinaciones posibles de operaciones para encontrar la distancia mínima.
- **funciones.h:** Archivo de cabecera que contiene las declaraciones de funciones para cargar los costos y obtener el valor de los costos de cada operación (inserción, eliminación, sustitución, transposición).
- **funciones.cpp:** Implementa las funciones definidas en *funciones.h*, que cargan los costos desde los archivos correspondientes y proporcionan las funciones para calcular los costos de las operaciones entre caracteres.
- **Archivos de Costos:** Contienen los valores de los costos de inserción, eliminación, sustitución y transposición entre caracteres, siendo estos cargados por las funciones correspondientes.

En esta implementación se busca separar la lógica de cálculo de las funciones auxiliares (gestionar costos y carga de datos). La interacción principal entre los archivos es la siguiente:

- Los archivos *DPdistancia.cpp* y *FBdistancia.cpp* llaman a las funciones definidas en *funciones.cpp* para acceder a los costos de las operaciones.
- Los archivos de costos son cargados por las funciones correspondientes que almacenan los valores en un matriz de vectores.

El proyecto utiliza bibliotecas estándar de C++ para manejar operaciones como la correcta carga de datos desde archivos de texto, manipulación eficiente de las cadenas y el cálculo de la distancia de edición.

4. Experimentos

En la sección de Experimentos, se detallará la infraestructura utilizada para asegurar la reproducibilidad de los resultados. Hardware utilizado: procesador Intel(R) Core(TM) i3-10100 CPU @ 3.60GHz 3.60 GHz, 16 GB RAM DDR4. Entorno software: sistema operativo Ubuntu 22.04 LTS, compilador g++ 11.4.0.

4.1. Dataset (casos de prueba)

Costos de cada operación:

- costosub(a, b)= 2 si a=b,y 0 si a=b.
- costoins(b) = 1 para cualquier carácter b.
- costodel(a) = 1 para cualquier carácter a.
- costotrans(a, b)=1 para transponer los caracteres adyacentes a y b.

Caso 1: Cadena Vacía Entradas:

- S1 = ""
- S2 = "$example$"

Salida esperada y costo total:

- La distancia mínima de edición es: 7

Operaciones:

1. Insertar todos lo carácteres "" $\rightarrow$ "$example$"
2. Costo total: 1 + 1 + 1 + 1 + 1 + 1 + 1 = 7

Caso 2: Transposiciones necesarias Entradas:

- S1 = "$laptop$"
- S2 = "$altpop$"

Salida esperada:

- La distancia mínima de edición es: 2

Operaciones:

1. l $\rightarrow$ a: Sustituir los carácteres ("$alptop$")
2. p $\rightarrow$ t: Sustituir los carácteres ("$altpop$")
3. Costo total: 1 + 1 = 2

Caso 3: Cadenas Distintas Entradas:

- S1 = "$contradiction$"
- S2 = "$distortion$"

Salida esperada:

- La distancia mínima de edición es: 11

Operaciones:

1. Sustituir los carácteres "con" $\rightarrow$ "dis" ("$distradiction$")
2. o: Insertar el carácter ("$distoradiction$")
3. Eliminar los carácteres "$adic$" (""$distortion$)
4. Costo total = 2 + 2 + 2 + 1 + 1 + 1 + 1 + 1 = 11

Caso 4: Cadena con carácteres repetidos Entradas:

- S1 = "$calle$"
- S2 = "$casa$"

Salida esperada:

- La distancia mínima de edición es: 5

Operaciones:

1. l $\rightarrow$ s: Sustituir los carácteres ("$casle$")
2. l $\rightarrow$ a: Sustituir los carácteres ("$casae$")
3. e: Eliminar el caracter ("$casa$")
4. Costo total = 2 + 2 + 1 = 5

Caso 5: Cadena con carácteres repetidos./ Entradas:

- S1 = "$banana$"
- S2 = "$nabana$"

Salida esperada:

- La distancia mínima de edición es: 3

Operaciones:

1. b: Eliminar el caracter ("$anana$")
2. a $\rightarrow$ n: Transponer los carácteres ("$naana$")
3. b: Insertar el caracter ("$nabana$")
4. Costo total: 1 + 1 + 1 = 3

4.2. Resultados

Los resultados obtenidos se presentan en dos cuadros, el primero relacionado a los tiempos de ejecución y el segundo con el consumo de memoria, comparando los enfoques de fuerza bruta (FB) y programación dinámica (DP) para resolver el problema de la distancia mínima de edición extendida. Se analizaron cinco casos de prueba con diferentes tamaños de entrada (S1 y S2) para evaluar el desempeño de los algoritmos bajo diferentes escenarios.

Cuadro 1: Resumen de resultados de los experimentos - Tiempo de ejecución

Caso de Prueba	Tamaño S1	Tamaño S2	Tiempo de Ejecución FB (ms)	Tiempo de Ejecución DP (ms)
Caso 1	0	7	4	11,5
Caso 2	6	6	124,7	29,7
Caso 3	13	10	362.890	30,8
Caso 4	5	4	40,1	24,3
Caso 5	6	6	103,2	26,8

Cuadro 2: Resumen de resultados de los experimentos - Memoria ocupada

Caso de Prueba	Tamaño S1	Tamaño S2	Memoria Utilizada FB (bytes)	Memoria Utilizada DP (bytes)
Caso 1	0	7	116,516	116,744
Caso 2	6	6	116,516	116,908
Caso 3	13	10	116,516	117,452
Caso 4	5	4	116,516	116,780
Caso 5	6	6	116,516	116,908

4.2.1. Resultados observados

1. **Tiempo de ejecución**

   - El algoritmo de fuerza bruta muestra un incremento exponencial en el tiempo de ejecución conforme aumenta el tamaño de las cadenas de entrada. Este comportamiento es esperable debido a que este enfoque analiza todas las posibles combinaciones, generando una alta complejidad computacional.
   - El algoritmo de programación dinámica, en contraste, mantiene tiempos de ejecución considerablemente más bajos en todos los casos, evidenciando una eficiencia mucho mayor. Esto se debe al almacenamiento de soluciones intermedias y la reutilización de estas, evitando el cálculo redundante.

2. **Memoria utilizada**

- Ambos algoritmos presentan un consumo de memoria relativamente similar en la mayoría de los casos, siendo el enfoque de programación dinámica ligeramente más exigente en algunos casos debido a la tabla de almacenamiento utilizada.
- En el caso de fuerza bruta, el consumo de memoria permanece constante, ya que no se utilizan estructuras auxiliares significativas, lo que contrasta con el enfoque de programación dinámica, donde la memoria depende de las dimensiones de la tabla.

4.2.2. Interpretación de los resultados

Los resultados obtenidos muestran una clara diferencia en el desempeño de los algoritmos de fuerza bruta y programación dinámica para el problema de la distancia mínima de edición extendida. En términos de eficiencia temporal, el enfoque de fuerza bruta exhibe un crecimiento exponencial en el tiempo de ejecución conforme aumentan los tamaños de las cadenas de entrada. Por ejemplo, en el caso 3, con cadenas de tamaños 13 y 10, el tiempo de ejecución alcanza 362,890 ms, lo que evidencia su alto costo computacional. Por el contrario, el algoritmo de programación dinámica, gracias a su estructura basada en la resolución de subproblemas y almacenamiento intermedio, reduce significativamente los tiempos, con un máximo de 30,8 ms en el mismo caso, destacándose como una opción mucho más eficiente cuando las cadenas son más grandes o complejas.

En cuanto al uso de memoria, ambos algoritmos tienen un consumo similar en la mayoría de los casos. Sin embargo, el algoritmo de programación dinámica requiere un leve incremento debido a la construcción de la tabla que almacena los resultados intermedios. Esto se puede observar en el caso 3, donde el consumo de memoria pasa de 116,516 bytes (fuerza bruta) a 117,452 bytes (programación dinámica). A pesar de este aumento, el impacto en la memoria es marginal y está ampliamente compensado por la notable mejora en los tiempos de ejecución, confirmando que el costo adicional en recursos es manejable dentro de contextos prácticos.

La escalabilidad del algoritmo de programación dinámica también se destaca frente a la fuerza bruta. Mientras que este último se vuelve impráctico para cadenas de mayor tamaño debido a su crecimiento exponencial en tiempo de ejecución, el enfoque dinámico mantiene un rendimiento estable y eficiente, siendo ideal para entradas de mayor complejidad. Esto lo convierte en una solución adecuada para escenarios donde se requiere procesar cadenas largas sin comprometer la viabilidad del cálculo.

En escenarios más simples o con entradas pequeñas, como el caso 4 (tamaños 5 y 4), las diferencias entre ambos enfoques son menos evidentes, ya que ambos algoritmos presentan tiempos de ejecución cercanos. Sin embargo, a medida que las cadenas se vuelven más largas o presentan características más complejas, como transposiciones o caracteres repetidos, las ventajas del enfoque dinámico se vuelven mucho más pronunciadas, reafirmando su utilidad en aplicaciones más exigentes.

Por último, el balance entre tiempo y memoria logrado por la programación dinámica resulta ser su mayor fortaleza. A pesar del ligero aumento en el uso de memoria, la drástica reducción en el tiempo de ejecución lo posiciona como un enfoque práctico y eficiente, especialmente en contextos donde el rendimiento y la escalabilidad son prioritarios. Estos resultados no solo validan teóricamente las ventajas

de la programación dinámica, sino que también las confirman empíricamente, destacándola como la elección preferida para resolver problemas de edición de cadenas en aplicaciones reales.

5. Conclusiones

Los resultados obtenidos en este trabajo reflejan la esencia de las diferencias fundamentales entre la fuerza bruta y la programación dinámica como enfoques para resolver problemas computacionales complejos, específicamente la distancia mínima de edición extendida. La programación dinámica, gracias a su naturaleza estructurada y eficiente, demuestra ser un método altamente favorable para abordar problemas de este tipo, especialmente cuando las cadenas involucradas son grandes o presentan patrones más complejos.

La relevancia de este análisis radica en cómo los resultados confirman que los enfoques algorítmicos bien diseñados no solo optimizan los recursos computacionales, sino que también hacen viable la resolución de problemas que, de otro modo, serían inabordables en escenarios prácticos. Esto tiene implicaciones significativas no solo en el ámbito académico, sino también en aplicaciones reales, donde la eficiencia en tiempo y memoria puede marcar una diferencia crítica.

En términos de los objetivos planteados, este análisis no solo los cumple, sino que aporta un entendimiento más profundo sobre la importancia de seleccionar el enfoque algorítmico adecuado según las necesidades del problema. Además, se establecen fundamentos claros para futuras optimizaciones y extensiones del algoritmo de programación dinámica, especialmente en entornos con limitaciones de recursos o demandas específicas de escalabilidad.

Por último, los hallazgos de este trabajo destacan el valor de la evaluación experimental rigurosa en la informática, ya que no solo proporciona evidencia empírica sobre el rendimiento de los algoritmos, sino que también permite interpretar con precisión sus límites y alcances en contextos diversos. Con esto, se logra no solo abordar la necesidad inicial planteada, sino también contribuir de manera significativa al entendimiento general del diseño y análisis de algoritmos.

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