Universidad de Concepción

Universidad de Concepción

(501404-1)

Análisis de algoritmos

Profesor Jérémy Barbay

Ayudante Leonardo Lovera

Tarea 2: Delete Insert Edit Distance

Autores:
Daniel Soto Salgado - 2021750673
Alex Jara Muñoz - 2021456147
Brendan Rubilar Vivanco - 2021750657

Concepción, 2025

1. Casos de Prueba

Liderado por: Alex Jara
Cadenas seleccionadas: `Guaren`, `ColoColo`, `Chinchilla` y `Quirquincho`.
La matriz de distancias entre estas cadenas es simétrica, ya que la distancia de edición entre dos cadenas es la misma independientemente del orden (es decir, $d(S,T) = d(T,S)$). A continuación se muestran todas las distancias calculadas con el código:

	`Guaren`	`ColoColo`	`Chinchilla`	`Quirquincho`
`Guaren`	0	14	14	11
`ColoColo`	14	0	12	17
`Chinchilla`	14	12	0	13
`Quirquincho`	11	17	13	0

Justificación de las distancias entre cadenas distintas:

- La distancia entre `Guaren` y `ColoColo` es 14, ya que ambas cadenas no comparten letras en la misma posición. Por ello, se deben eliminar todas las letras de la cadena origen y luego insertar todas las letras de la cadena destino.
- De forma similar, la distancia entre `Guaren` y `Chinchilla` es también 14, dado que son cadenas con pocas letras en común ('a','n') y en posiciones diferentes, lo que implica muchas eliminaciones e inserciones.
- La distancia entre `Guaren` y `Quirquincho` es la menor con 11, porque aunque son diferentes, comparten algunas letras importantes como 'u', 'r' y 'n', lo que reduce la cantidad total de operaciones necesarias.
- Entre `ColoColo` y `Chinchilla`, la distancia es 12, ya que, aunque diferentes, tienen algunas letras comunes como 'c', 'o' y 'l' que permiten disminuir la cantidad de inserciones y eliminaciones.
- La distancia entre `ColoColo` y `Quirquincho` es 17 porque estas cadenas son bastante distintas en longitud y composición,dado que solo comparten 1 'o', lo que implica muchas operaciones para transformar una en la otra.
- Finalmente, la distancia entre `Chinchilla` y `Quirquincho` es 13, debido a que comparten varias letras ('c', 'h', 'i', 'n'), lo que reduce las operaciones necesarias.
- Las distancias entre cadenas iguales son cero, ya que no es necesario realizar ninguna operación para transformarlas en sí mismas.

2. Fórmula Recursiva

Liderado por: Daniel Soto
Sea `str1` y `str2` cadenas cualesquiera, y $M = |str1|$, $N = |str2|$.
La **fórmula recursiva para la Edit Distance** es:

- **Caso 1 (m=0)**: Si la primera cadena está vacía, se necesitan n inserciones para convertirla en la segunda cadena.
- **Caso 2 (n=0)**: Si la segunda cadena está vacía, se necesitan m eliminaciones para convertir la primera cadena en vacía.
- **Caso 3 (str1[m] = str2[n])**: Si el último carácter de ambas cadenas es el mismo, no se requiere ninguna operación, y el resultado es el mismo que para los prefijos de tamaño $m-1$ y $n-1$.
- **Caso 4 (str1[m] != str2[n])**: Si los últimos caracteres son diferentes, se debe elegir entre:
  - Insertar un carácter al final de `str1` para que coincida con `str2`.
  - Eliminar el último carácter de `str1`.

  En ambos casos, se suma 1 operación y se toma el mínimo entre ambas posibilidades.

3. Implementación

Liderado por: Alex Jara

3.1. Edit Distance Recursivo

3.1.1. Funcionamiento

La función es recursiva y toma como parámetros las cadenas `S` y `T`, junto con sus longitudes actuales `m` y `n`. Los casos base son: si `m == 0`, el costo es `n`, ya que se deben insertar todos los caracteres restantes de `T`; y si `n == 0`, el costo es `m`, ya que se deben eliminar todos los caracteres restantes de `S`. Para cada llamada recursiva, si los caracteres actuales coinciden (`S[m-1] == T[n-1]`), no hay costo adicional, y se realiza una llamada recursiva para las subcadenas más cortas (`m-1, n-1`). Si los caracteres no coinciden, se calcula el costo mínimo entre eliminar un carácter de `S` (`m-1, n`) o insertar un carácter en `S` (`m, n-1`). A este mínimo se le suma 1 para reflejar la operación realizada. El resultado final se obtiene al completar todas las llamadas recursivas.

```cpp
int editDistanceRecursive(string S, string T, int m, int n) {
    // Caso base: si S esta vacio, se requieren n inserciones
    if (m == 0) return n;
    // Caso base: si T esta vacio, se requieren m eliminaciones
    if (n == 0) return m;

    if (S[m-1] == T[n-1]) {
        // Si los caracteres coinciden, no hay costo adicional
        return editDistanceRecursive(S, T, m-1, n-1);
    } else {
        // Minimo entre eliminar (m-1,n) o insertar (m,n-1)
        return 1 + min(editDistanceRecursive(S, T, m-1, n),
                       editDistanceRecursive(S, T, m, n-1));
    }
}
```

Implementación recursiva

3.2. Edit Distance Memoización

3.2.1. Funcionamiento

Se utiliza una matriz bidimensional `memo` de tamaño $(m+1) \times (n+1)$, donde `m` es la longitud de `S` y `n` es la longitud de `T`. Cada celda `memo[m][n]` almacena el resultado previamente calculado para evitar recalcularlo. Los casos base son: si `m == 0`, el costo es `n`, ya que se deben insertar todos los caracteres de `T`; y si `n == 0`, el costo es `m`, ya que se deben eliminar todos los caracteres de `S`. Para cada celda `memo[m][n]`, se calcula el costo mínimo basado en lo siguiente: si los caracteres coinciden (`S[m-1] == T[n-1]`), el costo es igual al de la celda diagonal superior izquierda (`memo[m-1][n-1]`). Si no coinciden, se toma el mínimo entre el costo de eliminar un carácter (`memo[m-1][n]`) y el costo de insertar un carácter (`memo[m][n-1]`). A este mínimo se le suma 1 para reflejar la operación realizada. La función auxiliar `editDistanceMemoHelper` realiza los cálculos recursivos y utiliza la matriz `memo` para almacenar los resultados. El resultado final se encuentra en `memo[m][n]`.

```cpp
int editDistanceMemoHelper(string &S, string &T, int m, int n, vector<vector<int>> &memo) {
    if (m == 0) return n; // Caso base: insertar todos los caracteres de T
    if (n == 0) return m; // Caso base: eliminar todos los caracteres de S
    if (memo[m][n] != -1) return memo[m][n]; // Resultado ya calculado
    if (S[m-1] == T[n-1]) {
        memo[m][n] = editDistanceMemoHelper(S, T, m-1, n-1, memo); // No se necesita operacion
    } else {
        memo[m][n] = 1 + min(editDistanceMemoHelper(S, T, m-1, n, memo), // Eliminar
                             editDistanceMemoHelper(S, T, m, n-1, memo)); // Insertar}
    return memo[m][n];}
int editDistanceMemo(string S, string T) {
    int m = S.length();
    int n = T.length();
    vector<vector<int>> memo(m+1, vector<int>(n+1, -1)); // Inicializar memo con -1
    return editDistanceMemoHelper(S, T, m, n, memo);}
```

Implementación con Memoizacion

3.3. Edit Distance DP

3.3.1. Funcionamiento

Se inicializa una matriz bidimensional `matriz` de tamaño $(filas+1) \times (columnas+1)$, donde `filas` es la longitud de `str1` y `columnas` es la longitud de `str2`. Los casos base se configuran de la siguiente manera: `matriz[i][0]` se inicializa con i, representando el costo de eliminar todos los caracteres de `str1`; y `matriz[0][j]` se inicializa con j, representando el costo de insertar todos los caracteres de `str2`. Para cada celda `matriz[i][j]`, se calcula el costo mínimo basado en lo siguiente: si los caracteres coinciden (`str1[i-1] == str2[j-1]`), el costo es igual al de la celda diagonal superior izquierda (`matriz[i-1][j-1]`). Si no coinciden, se toma el mínimo entre el costo de eliminar un carácter (`matriz[i-1][j]`) y el costo de insertar un carácter (`matriz[i][j-1]`). A este mínimo se le suma 1 para reflejar la operación realizada.

```cpp
    for (int i = 1; i <= filas; i++) {
    for (int j = 1; j <= columnas; j++) {
        if (str1[i-1] == str2[j-1]) {
            matriz[i][j] = matriz[i-1][j-1];
        } else {
            matriz[i][j] = 1 + min(matriz[i-1][j], matriz[i][j-1]);
        }
```

Implementación DP

3.4. Edit Distance DP-Optimized

3.4.1. Funcionamiento

Se utilizan dos vectores unidimensionales, `prev` y `curr`, para almacenar los costos de la fila anterior y la fila actual de la matriz, respectivamente. Esto reduce el uso de memoria de $O(m \times n)$ a $O(n)$, donde m y n son las longitudes de `str1` y `str2`. Los casos base se configuran de la siguiente manera: `prev[j]` se inicializa con j, representando el costo de insertar todos los caracteres de `str2`; y `curr[0]` se inicializa con i, representando el costo de eliminar todos los caracteres de `str1`. Para cada posición `curr[j]`, se calcula el costo mínimo basado en lo siguiente: si los caracteres coinciden (`str1[i-1] == str2[j-1]`), el costo es igual al de la celda diagonal superior izquierda (`prev[j-1]`). Si no coinciden, se toma el mínimo entre el costo de eliminar un carácter (`prev[j]`) y el costo de insertar un carácter (`curr[j-1]`). A este mínimo se le suma 1 para reflejar la operación realizada.

```
// Inicializacion de vectores
vector<int> prev(n + 1, 0);
vector<int> curr(n + 1, 0);
for (int j = 0; j <= n; j++) prev[j] = j; // Caso base: insertar

// Llenado de los vectores
for (int i = 1; i <= m; i++) {
    curr[0] = i; // Caso base: eliminar
    for (int j = 1; j <= n; j++) {
        if (str1[i - 1] == str2[j - 1]) {
            curr[j] = prev[j - 1]; // No se necesita operacion
        } else {
            curr[j] = 1 + min(prev[j], curr[j - 1]); // Eliminar o insertar
        }
    }
    prev = curr; // Actualizar fila anterior
}
return prev[n];
```

Implementación DP-Optimized

1. Al final de cada iteración, `prev` se actualiza con los valores de `curr`.
2. El resultado final se encuentra en `prev[n]`.

4. Complejidad

Liderado por: Daniel Soto

4.1. Edit Distance Recursivo

4.1.1. Complejidad Temporal

La complejidad temporal de esta implementación recursiva es $O(2^{\text{máx}(m,n)})$ en el peor caso, donde m y n son las longitudes de las cadenas S y T, respectivamente. Esto se debe a que el algoritmo recalcula los mismos subproblemas múltiples veces, generando una cantidad exponencial de llamadas recursivas. Cada llamada recursiva genera dos nuevas llamadas (una para eliminar y otra para insertar).

4.1.2. Complejidad Espacial

La complejidad espacial del algoritmo es $O(\text{máx}(m, n))$, ya que la profundidad máxima de la pila de llamadas recursivas está limitada por la suma de las longitudes de las cadenas. Esto ocurre porque cada llamada recursiva reduce la longitud de una de las cadenas en uno, hasta que se alcanza un caso base.

4.2. Edit Distance Memoización

4.2.1. Complejidad Temporal

La complejidad temporal del algoritmo es $O(m \times n)$. Esto se debe a que cada subproblema, representado por una celda en la matriz `memo`, se calcula una sola vez. En el peor de los casos, se realizan $m \times n$ operaciones para llenar la matriz, y cada operación tiene un costo constante.

4.2.2. Complejidad Espacial

La complejidad espacial del algoritmo $O(m \times n)$, debido al uso de una matriz que guarda los resultados de los subproblemas resueltos.

4.3. Edit Distance DP

4.3.1. Complejidad Temporal

La complejidad temporal del algoritmo es $O(m \times n)$. Esto se debe a que se recorren todas las celdas de la matriz, y cada celda se calcula en tiempo constante mediante una comparación y una operación como mínimo. En el peor de los casos, se realizan $m \times n$ operaciones.

4.3.2. Complejidad Espacial

La complejidad espacial del algoritmo es $O(m \times n)$, ya que se utiliza una matriz bidimensional para almacenar los resultados de los subproblemas. Cada celda de la matriz ocupa memoria proporcional al producto de las longitudes de las cadenas.

4.4. Edit Distance DP-Optimized

4.4.1. Complejidad Temporal

La complejidad temporal del algoritmo es $O(m \times n)$, donde m y n son las longitudes de las cadenas $str1$ y $str2$, respectivamente. Esto se debe a que el algoritmo recorre cada carácter de ambas cadenas y realiza operaciones constantes ($O(1)$) para calcular el costo mínimo en cada posición. En el peor de los casos, se realizan $m \times n$ operaciones, lo que garantiza que todos los subproblemas sean resueltos de manera eficiente.

4.4.2. Complejidad Espacial

La complejidad espacial del algoritmo es $O(n)$, ya que utiliza dos vectores unidimensionales de tamaño $n + 1$ para almacenar los resultados de la fila actual y la fila anterior de la matriz. Esto representa una mejora significativa en comparación con las implementaciones tradicionales de programación dinámica, que requieren $O(m \times n)$ espacio para almacenar toda la matriz bidimensional.

5. Protocolo Experimental

Liderado por: Brendan Rubilar

Basado en el código experimental Uhr (provisto por Leonardo Lovera), se desarrolló una versión modificada que permite ejecutar múltiples instancias de varios algoritmos de forma secuencial. Esta versión genera un archivo CSV en el que se registra el tiempo promedio de ejecución (en milisegundos) y el peak de memoria utilizado (en kilobytes), esto haciendo uso de psapi (librería de Windows) para cada algoritmo, para mejorar la precisión de esta ultima, se midió usando otro hilo del procesador.

Para las pruebas, se utilizaron cuatro extractos de textos provenientes de diferentes libros, lo que permite realizar combinaciones de pares de textos. Los resultados obtenidos en los archivos CSV pueden ser representados mediante gráficos de barras, facilitando la comparación visual del rendimiento de los algoritmos en términos de tiempo y uso de memoria.

6. Análisis Experimental

Liderado por: Brendan Rubilar

Los textos escogidos provienen de los libros:

1. https://www.gutenberg.org/cache/epub/2000/pg2000.txt
2. https://www.gutenberg.org/cache/epub/84/pg84.txt
3. https://www.gutenberg.org/cache/epub/1342/pg1342.txt

Los textos corresponden a extractos de los siguientes capítulos:

- `text1`: Capítulo II Don Quijote (890 bytes).
- `text2`: Letter 1 Frankenstein (625 bytes).
- `text3`: Pride and Prejudice Chapter I (801 bytes).
- `text4`: Alice's Adventures in Wonderland CHAPTER I (602 bytes).

En esta sección se realizaron diferentes experimentos; en primer lugar, se compararon los 4 algoritmos. Dada la complejidad exponencial del algoritmo recursivo, se utilizó el servidor *Chome* al cual tenemos acceso y nos limitamos a usar solo 15 caracteres de los 4 textos, con 32 repeticiones por cada iteración (para calcular desviación estándar y tiempos promedios).
Las características de este equipo son: Intel(R) Xeon(R) Gold 5320T CPU @ 2.30GHz y 264 GB de RAM aprox.

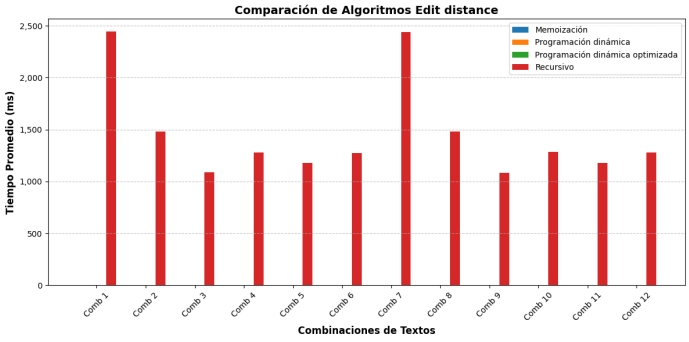

Figura 1: Tiempos de ejecución: Recursivo, Memoización, Dp, Dp-Otimized.

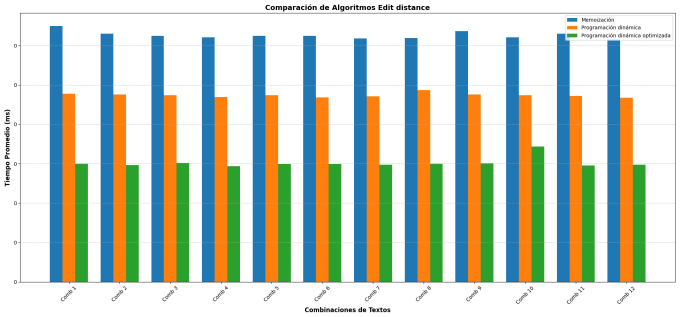

Figura 2: Uso de memoria: Memoización, Dp, Dp-Otimized.

Debido a la marcada diferencia de complejidades del algoritmo recursivo sobre el resto, nos centramos en estudiar los algoritmos con memoización, programación dinámica y su versión optimizada. Para esto se realizaron pruebas con extractos de los textos completos (tamaño mencionado anteriormente en esta sección) y con 64 repeticiones por cada iteración. Todo esto en un equipo local con las siguientes características: CPU Intel(R) I5 11400H @ 2.70GHz y 16GB de RAM.

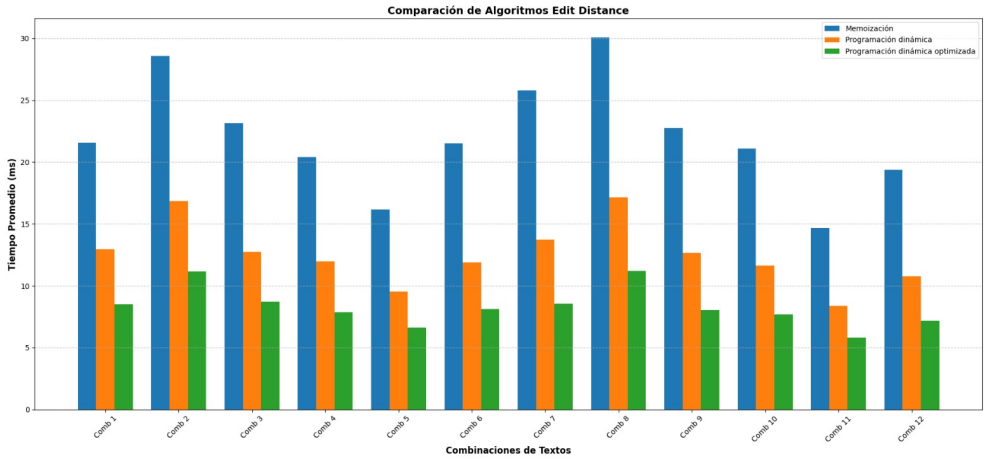

Figura 3: Comparación de tiempos de ejecución: Memoización, Dp, Dp-Otimized.

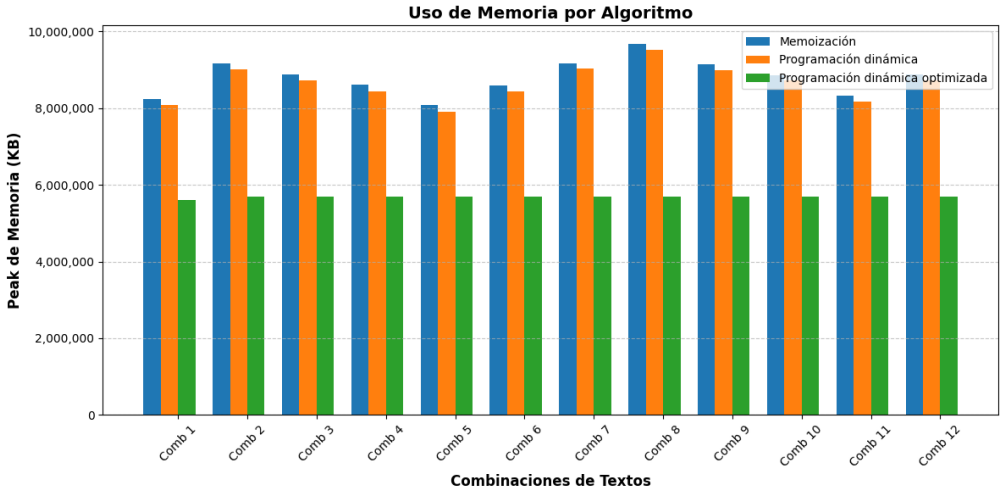

Figura 4: Comparación de uso de Memoria: Memoización, Dp, Dp-Otimized.

7. Conclusión

Liderado por: Daniel Soto

Las complejidades temporales y espaciales teóricas analizadas en esta tarea se ven confirmadas empíricamente mediante los experimentos realizados. La implementación recursiva, aunque conceptualmente sencilla, resultó ser altamente ineficiente para cadenas largas debido a su crecimiento exponencial en tiempo de ejecución. Por el contrario, las versiones con memoización y programación dinámica demostraron ser significativamente más eficientes, tanto en tiempo como en memoria, especialmente la versión optimizada con vectores unidimensionales, que logró mantener el rendimiento reduciendo considerablemente el uso de espacio. Estas observaciones refuerzan la importancia de elegir el enfoque algorítmico adecuado según los recursos disponibles y la naturaleza del problema. En general, este estudio permitió comprender de forma profunda cómo diferentes técnicas de optimización afectan el desempeño de un algoritmo clásico como Edit Distance.