REPORTE TAREA 2 y 3
ALGORITMOS Y COMPLEJIDAD

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

Martín Felipe Ibañez Gonzalez

18 de noviembre de 2024

01:10

Resumen

En este informe se revisará el problema de la Distancia de Levenshtein mas una operacion mas: Transposicion. El objetivo sera hacer una comparacion de dos enfoques de programacion para este mismo problema: Fuerza Bruta y Programacion Dinamica. Dentro del informe se estudiará la logica, complejidad temporal y espacial de los algoritmos implementados. De los graficos obtenidos podemos observar que en el caso de Fuerza Bruta existe una tendencia exponencial, mientras que Programacion Dinamica es una tendencia lineal, demostrando su superioridad.

Índice

1. Introducción

En este informe realizado en el curso de Analisis y Diseño de algoritmos en Ciencias de la Computacion se va a analizar el problema de la distancia mínima de edición. Este campo es importante porque es aca donde se trabaja en resolver problemas recurrentes en la vida diaria en un tiempo aceptable y que la solucion sea optima. Entre los problemas que ya se han encontrado solucion esta: Problemas de ordenacion (QuickSort, MergeSort), Problemas de busqueda (Busqueda Binaria), Problemas de Grafos (Algoritmo de Dijkstra), etc.

El problema a resolver es una variacion de la Distancia de Levenshtein. Esta distancia, que obtiene su nombre del cientifico ruso Vladimir Levenshtein, es utilizada con frecuencia en programas que comparan dos cadenas de texto y dentro del mundo de las ciencas de la computacion es bien conocido. Se trata de minimizar el costo final de convertir una cadena de caracteres en otra, originalmente usando solo tres operaciones, pero en el contexto de esta tarea agregamos una cuarta operacion:

- Sustitucion: Sustituye un carácter 'x' por un caracter 'y' en la cadena
- Insercion: Insertar un carácter 'x'en la cadena
- Eliminacion: Elimina el carácter 'x' de la cadena
- Transposicion: Intercambia los carácteres 'x' e 'y' si son adyacentes

Cada una de estas operaciones tiene un costo predeterminado. Para efectos de esta tarea se va a utilizar el abecedario ingles, esto significa que se trabajará con un total de 26 letras ('a'-z') y sin carácteres especiales. Los costos van a ser generados aleatoreamente en valores del 1 hasta el 5, para las operaciones sustitucion y transposicion importa el orden en que apliquen, esto significa que el costo de sustituir 'a' por 'b' no es el mismo que el costo de sustituir 'b' por 'a'. Con esta informacion, es prudente preguntarse, entre todas las posibilidades, ¿Cual es el costo minimo que se puede obtener?

Para resolver este problema, se implementaron dos algoritmos con dos enfoques diferentes: Fuerza Bruta y Programación Dinamica.

Fuerza Bruta se enfoca en resolver todos los casos posibles y al final elegir el caso con el menor costo. Los pros de este enfoque es que es mas facil de implementar que Programacion Dinamica pero su rendimiento es uno de los peores dentro del mundo de algoritmos.

Programacion Dinamica resuelve estos problemas dividiendolos en subproblemas mas pequeños y guardando las respuestas para usarlas en las siguientes iteraciones. Los pros de este enfoque es que su rendimiento es uno de los mejores, pero el problema radica en su elevada dificultad para entender e implementar los algoritmos de este tipo.

Este problema es importante para un curso de pregrado porque en este contexto el estudiante aprende a programar una solucion con programacion dinamica y fuerza bruta, ademas se evidencia la diferencia de rendimiento que distintos enfoques tienen. De esta manera, uno puede tomar decisiones con un abanico mas amplio de opciones, dando la posibilidad a la mejor opcion a ser escogida.

2. Diseño y Análisis de Algoritmos

> *Ambos codigos usaron como referencia a el algoritmo de la Distancia de Levenshtein*
>
> — GeeksforGeeks_community, 2024 [2]

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?"*
>
> — Knuth, 1998 [3]

- Descripcion de la solución diseñada: El algoritmo recibe cuatro parametros:
  - int i: largo de s1
  - int j: largo de s2
  - string s1: cadena de caracteres 1
  - string s2: cadena de caracteres 2

  Primero revisamos si i y j son iguales a 0, si es cierto, significa que no hay nada que comparar, ya que los largos son cero y retornamos 0. Luego el algoritmo sigue con el caso de que si i es igual a 0 pero j distinto de 0, este caso sucede cuando la palabra 1 tiene menor longitud que la palabra a transformar, por lo que llamamos a la funcion 'costo_ins' para insertar las letras faltantes. El siguiente caso es cuando j es igual a 0, esto pasa cuando la palabra 1 es mas larga que la palabra 2, en este caso llamamos a la funcion 'costo_del' para eliminar las palabras sobrantes.

  Ahora, revisamos si los caracteres que estamos comparando son iguales, en caso contrario obtenemos el costo minimo de sustitucion con las letras que se esten revisando. Luego, empezamos a calcular el costo minimo, inicializando la variable en un valor maximo. Empezamos a calcular todos los casos, partiendo por la sustitucion. Igualamos el costo minimo al minimo entre el valor minimo anterior y a un llamado recursivo (avanzando una letra en la comparacion) + el costo de la sustitucion. El segundo caso que se revisará sera el de la insercion, se iguala el costo minimo al minimo entre el mismo y un llamado recursivo (avanzando solo en s2) + el costo de insersion de la letra que se esta revisando en s2. El tercer caso que se revisa es el de la eliminacion, aqui igualamos el costo minimo al minimo entre el mismo y un llamado recursivo (avanzando solo en s1) + costo de eliminacion de la letra que se esta revisando en s1. Y el cuarto y ultimo caso que se revisa es el de la transposicion, donde solo se revisa si solo si la letra que se este revisando en s1 es igual a la letra siguiente en s2 y si la letra siguiente en s1 es igual a la letra que se este revisando en s2. Si se cumple la condicion, el costo minimo se iguala al minimo entre el mismo y la suma de la llamada recursiva (avanzan dos espacios en las dos cadenas) con el costo de transposicion de las letras.

El algoritmo se termina retornando la variable costo minimo, despues de todos los casos revisados.

- Pseudocódigo (línea 1)

- s1: abba, s2: baba y los costos siendo: Sustitucion: 2, Insercion: 1, Eliminacion: 1 y Transposicion: 1. Primero se revisará si las longitudes son 0, en este caso las dos son 4 asi que no entra en el primer if. Tampoco entra en el segundo ni el el tercer if, ya que i y j siguen siendo 4. Luego, entra en el cuarto if, para calcular el valor del costo de la sustitucion, en este caso particular como s1[i-1] y s2[j-1] ambos son 'a' el costo de sustitucion es igual a 0. Ahora, empiezan los casos particulares para calcular el costo minimo, partiendo por la sustitucion, donde se aplica recursividad restandole 1 a i y j para revisar las palabras s1: abb y s2: bab. Despues de la recursividad se actualiza el costo minimo. El siguiente caso a revisar es el de la insercion, donde la recursividad esta vez es con i y j-1, al agregar una letra s2[j-1] a s1 mas el costo de insercion de esa letra. El tercer caso a revisar es el de eliminacion, este siendo parecido al caso anterior pero esta vez se aplica recursion con i-1 y j, al eliminar la letra i-1 de s1. El cuarto y ultimo caso a revisar es el de la transposicion, donde al i y j ser mayores a cero y cuando se llegue al caso de s1: ab y s2: ba, se entrara al if. Dentro del if se actualizara el valor de costo minimo al minimo entre este y otra recursion con i-2 y j-2 mas el valor del costo de la transposicion. En este caso despues de esta recursion se terminaria la palabra y entraria al primer if, retornando cero. Sumando el valor de costo de transposicion, que en este caso es igual a 1, se encontraria el valor minimo de edicion, resultando en 1. La operacion seria una unica transposicion en la recursion de s1: ab y s2: ba

-
  - Complejidad Temporal: $O(4^{(i+j)})$
  - Complejidad Espacial: $O(i+j)$

- La inclusion de la transposicion hace que se tenga que hacer una recursion mas, esto significa que usando el teorema maestro la complejidad cambie de $O(3^{(i+j)})$ a $O(4^{(i+j)})$. Ahora, los costos variables tambien impactan en la complejidad, ya que el tener que buscarlos dentro de una matriz, en vez de tener costo fijo, le pega al rendimiento del programa. Las busquedas dentro de matrices siempre son un tema.

Algoritmo 1: Pseudocodigo, inspirado en la Distancia de Levenshtein

```
Procedure FUERZA_BRUTA(i, j, s1, s2)
    if longitudes de s1 y s2 = 0 then
        return 0
    if longitud de s1 = 0 then
        costo ← 0
        for k ← 0 to j do
            costo ← costo + COSTO_MATRIZ_ins(s2[k])
        return costo
    if longitud de s2 = 0 then
        costo ← 0
        for k ← 0 to i do
            costo ← costo + COSTO_MATRIZ_del(s1[k])
        return costo
    if s1[i-1] = s2[j-1] then
        costo_sustitucion ← 0
    else
        costo_sustitucion ← COSTO_MATRIZ_sub(s1[i − 1], s2[j − 1])
    costo_min ← MAX
    costo_min ← mín(costo_min, FUERZA_BRUTA(i-1, j-1, s1, s2) + costo_sustitucion)
    costo_min ← mín(costo_min, FUERZA_BRUTA(i, j-1, s1, s2) + COSTO_MATRIZ_ins(s2[j − 1]))
    costo_min ← mín(costo_min, FUERZA_BRUTA(i-1, j, s1, s2) + COSTO_MATRIZ_del(s1[i − 1]))
    if i >1 and j >1 and s1[i-1] = s2[j-2] and s1[i-2] = s2[j-1] then
        costo_min ← mín(costo_min, FUERZA_BRUTA(i-2, j-2, s1, s2) + COSTO_MATRIZ_trans(s1[i − 2], s2[j − 2]))
    return costo_min
Procedure COSTO_MATRIZ(a, b)
    return Costo en matriz
```

2.2. Programación Dinámica

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

Erickson, 2019 [1]

2.2.1. Descripción de la solución recursiva

Esta solucion sigue una logica bien parecida a fuerza bruta, pero en este caso usamos una matriz para que se guarden los resultados obtenidos anteriormente, para no calcularlos de nuevo. Este cambio hace que el tiempo total se reduzca considerablemente. La matriz es iniciada llena de -1, para calcular el costo minimo, esto es porque no es posible que el costo minimo sea menor a 0.

2.2.2. Relación de recurrencia

Para los casos bases tenemos:

1) Cuando i es igual a 0: En este caso se retorna la suma del costo de insercion de las letras faltantes

2) Cuando j es igual a 0: Aqui se retorna la suma del costo de eliminacion de las letras sobrantes

3) Si el resultado ya fue calculado anteriormente: Si el valor en la posicion [i][j] es distinto a -1, esto es porque ya fue calculado anteriormente, entonces, se retorna el valor sin hacer ningun calculo. De esta manera, podemos utilizar los valores anteriormente calculados y mejorar el rendimiento.

Ahora para los calculos con recursiones, estos pueden ser escritos de esta manera:

$$DP(i,j) = \text{mín} \begin{cases} DP(i-1,j-1) + \text{costo\_sub}(s1[i-1], s2[j-1]), & \text{si } s1[i-1] \neq s2[j-1], \\ DP(i,j-1) + \text{costo\_ins}(s2[j-1]), \\ DP(i-1,j) + \text{costo\_del}(s1[i-1]), \\ DP(i-2,j-2) + \text{costo\_trans}(s1[i-2], s2[j-2]), & \text{si aplica transposición.} \end{cases}$$

2.2.3. Identificación de subproblemas

Los subproblemas equivalen a las llamadas recursivas de la funcion DP, lo que va cambiando es el valor de i y j, donde estos representan la letra que esta siendo revisada. Existen ixj subproblemas en total que equivalen a los caminos que se pueden tomar para resolver el problema.

2.2.4. Estructura de datos y orden de cálculo

La estructura de datos utilizada es una Matriz de ixj de tamaño. Cada celda de la matriz contiene un valor minimo calculado por el algoritmo, donde el ultimo valor de la matriz es el valor minimo que se busca. El orden de calculo en este caso es de arriba hacia abajo, osea TOP-DOWN. Los subproblemas se descomponen 'avanzado en la cadena' recursivamente pero cambiando los parametros en i-1 y j-1 o i-2 y j-2, en caso de que se ocupe la transposicion.

2.2.5. Algoritmo utilizando programación dinámica

Algoritmo 2: Pseudocodigo, inspirado en la Distancia de Levenshtein

```
1  Procedure DP(i, j, s1, s2)
2      if s1 esta vacia then
3          costo_total ← 0
4          for k ← 0 to j − 1 do
5              costo_total ← costo_total + COSTO_MATRIZ_ins(s2[k])
6          return costo_total
7      if s2 esta vacia then
8          costo_total ← 0
9          for k ← 0 to i − 1 do
10             costo_total ← costo_total + COSTO_MATRIZ_del(s1[k])
11         return costo_total
12     if valor en matriz ya calculado then
13         return dp_1[i][j]
14     costo_min ← MAX_VALOR
15     if los caracteres que se estan revisando con distintos then
16         costo_min ← mín(costo_min, DP(i-1, j-1, s1, s2) + COSTO_MATRIZ_sub(s1[i − 1], s2[j − 1])
17     else
18         costo_min ← mín(costo_min, DP(i-1, j-1, s1, s2))
19     costo_min ← mín(costo_min, DP(i, j-1, s1, s2) + COSTO_MATRIZ_ins(s2[j − 1])
20     costo_min ← mín(costo_min, DP(i-1, j, s1, s2) + COSTO_MATRIZ_del(s1[i − 1])
21     if i > 1 ∧ j > 1 ∧ s1[i − 1] = s2[j − 2] ∧ s1[i − 2] = s2[j − 1] then
22         costo_min ← mín(costo_min, DP(i-2, j-2, s1, s2) + COSTO_MATRIZ_trans(s1[i − 1], s2[j − 1])
23     dp_1[i][j] ← costo_min
24     return dp_1[i][j]
25 Procedure COSTO_MATRIZ(a, b)
26     return Costo en matriz
```

3. Implementaciones

Los archivos implementados son los siguientes:

- operaciones.cpp: En este .cpp estan las funciones para buscar los costos, estas son: costo_sub, costo_ins, costo_del, costo_trans. Algoritmo DP para encontrar el valor minimo de edicion y el Algoritmo de fuerza bruta para encontrar el valor minimo de edicion.
- Carpeta output:
  - costo_del.txt: Archivo txt donde se guardan los costo para eliminar las letras especificas
  - costo_ins.txt: Archivo txt donde se guardan los costos para insertar una letra en especifico
  - costo_sub.txt: Archivo donde se guarda la matriz de costos para sustituir una letra con otra
  - costo_trans.txt: Archivo donde se guarda la matriz de costos para hacer una transposicion de una letra con otra
  - palabras.txt: Archivo que contiene los casos de prueba, estos incluyen la palabra original y junto a ella esta la palabra revuelta para ser comparada.
- generacion_dataset.cpp: Este archivo lo que hace es crear los txt con los costos, estos tienen valores aleatorios del 1 al 5.
- graficos.py: Este archivo de python crea los graficos en base a los resultados obtenidos en los experimentos.
- resultados_dp.txt: Este archivo de texto contiene los tiempos que se demoro el algoritmo de Programacion dinamica en obtener el costo minimo de edicion con sus respectivas cantidades de letras.
- resultados_fb.txt: Este archivo de texto contiene los tiempos que se demoro el algoritmo de Fuerza Bruta en obtener el costo minimo de edicion con sus respectivas cantidades de letras.

4. Experimentos

> "*Non-reproducible single occurrences are of no significance to science.*"
>
> —Popper,2005 [4]

Los experimentos fueron realizados con un procesador 11th Gen Intel Core i5-1155G7 2.50 GHz (8 CPUs), 16 GB RAM, almacenamiento SSD 250 GB. Con un WSL Ubuntu 20.04 y g++ 11.3.0.

4.1. Dataset (casos de prueba)

Los costos utilizados en este experimento fueron generados aleatoriamente y estan guardados en los archivos txt que vienen junto a este informe.

- cost_del.txt: Matriz de 1x26 con los costos de eliminacion de la letra a hasta la z
- cost_ins.txt: Matriz de 1x26 con los costos de insercion de la letra a hasta la z
- cost_sub.txt: Matriz de 26x26 con los costos de substitucion de la letra a hasta la z
- cost_trans.txt: Matriz de 26x26 con los costos de transposicion de la letra a hasta la z

Las palabras utilizadas van a variar en la cantidad de letras que contengan. Luego, se va a revolver s1 para que en cada iteracion quede en una forma distinta a la anterior. Van a ser 11 palabras a probar de 4 hasta 15 letras. Las palabras son:

hola, aloh: Costo: 4
estas, staes: Costo: 5
prueba, ebrapu: Costo: 12
salidas, adsilas: Costo 5:
palabras, arsbalpa: Costo: 12
sencillos, nliossecs: Costo: 11
resultados, dtreasosul: Costo: 13
fotografias, otfasrigfoa: Costo: 17
profesionales, aeslonspoeirf: Costo: 22
skibiditoilets, ttilsioikeibds: Costo: 15
manifestaciones, nfcmaoisetainse: Costo: 17
", " Costo: 0

Tambien esta su respectivo costo minimo que los algoritmos calcularon.

Estas palabras fueron revueltas aleatoreamente para los experimentos y calculadas mas de una vez. Las palabras originales con su comparacion estan en un txt llamado palabras.txt.

4.2. Resultados

Las palabras usadas para los casos de prueba fueron palabras al azar que fueron reordenadas aleatoriamente, luego fueron procesados y graficados por un archivo python usando la libreria matplotlib.

Los resultados son los esperados, la fuerza bruta crece exponencialmente y programacion dinamica pareciera que crece con una tendencia lineal, a continuacion se mostraran los graficos correspondientes.

$Cantidad\ de\ Letras$	$Tiempo[ns]$
0	12752
4	25200
5	48624
6	198206
7	1024360
8	5381835
9	30039713
10	161451058
11	865655638
12	28526771974
13	160535398672
14	940537928386

Cuadro 1: Tabla de valores de la cantidad de letras con su respectivo tiempo en ns en el enfoque Fuerza Bruta

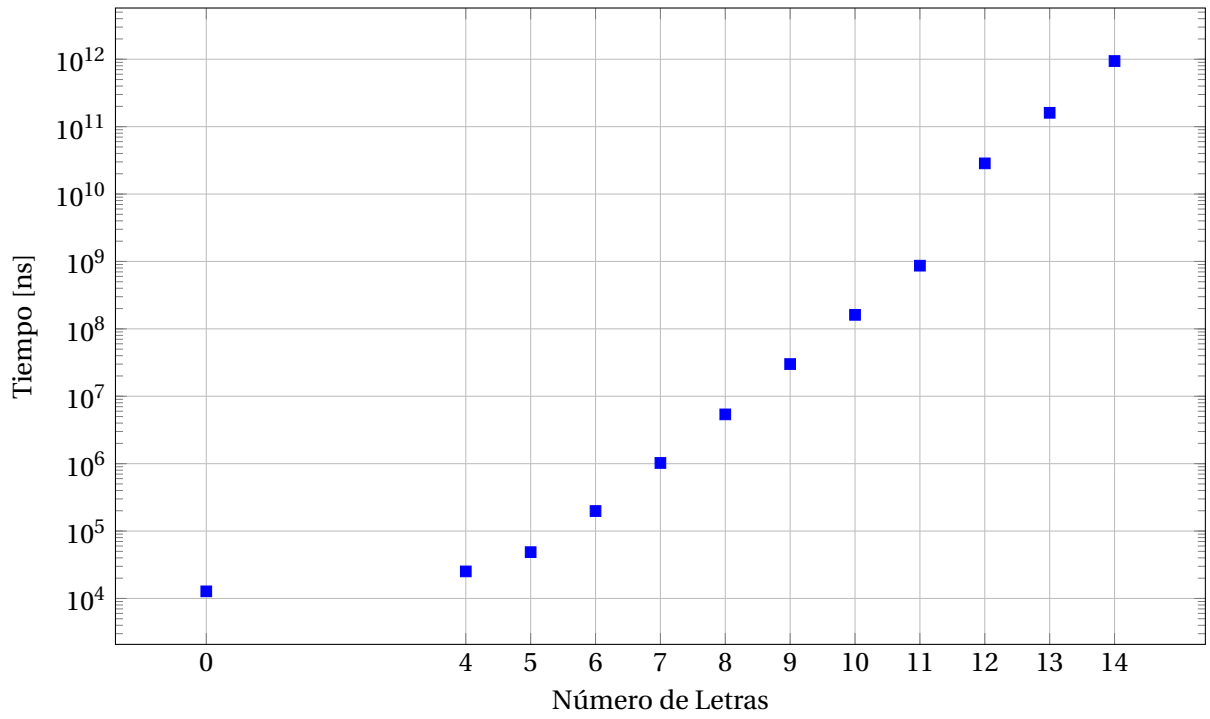

Figura 1: Grafico de Fuerza Bruta

Con la tabla y el grafico podemos destacar la velocidad con la que aumenta el tiempo si aumentamos la cantidad de letras. Esto es porque el enfoque de Fuerza Bruta ocupa mucha memoria y hace calculos rebundantes, afectando el rendimiento

Ahora revisaremos la tabla y el grafico correspondiente al enfoque de Programacion Dinamica:

$Cantidad\ de\ Letras$	$Tiempo[ns]$
0	1175
4	15259
5	16697
6	18495
7	20119
8	26385
9	21226
10	23265
11	23720
12	27026
13	29308
14	31102

Cuadro 2: Tabla de valores de la cantidad de letras con su respectivo tiempo en ns del enfoque Programacion Dinamica

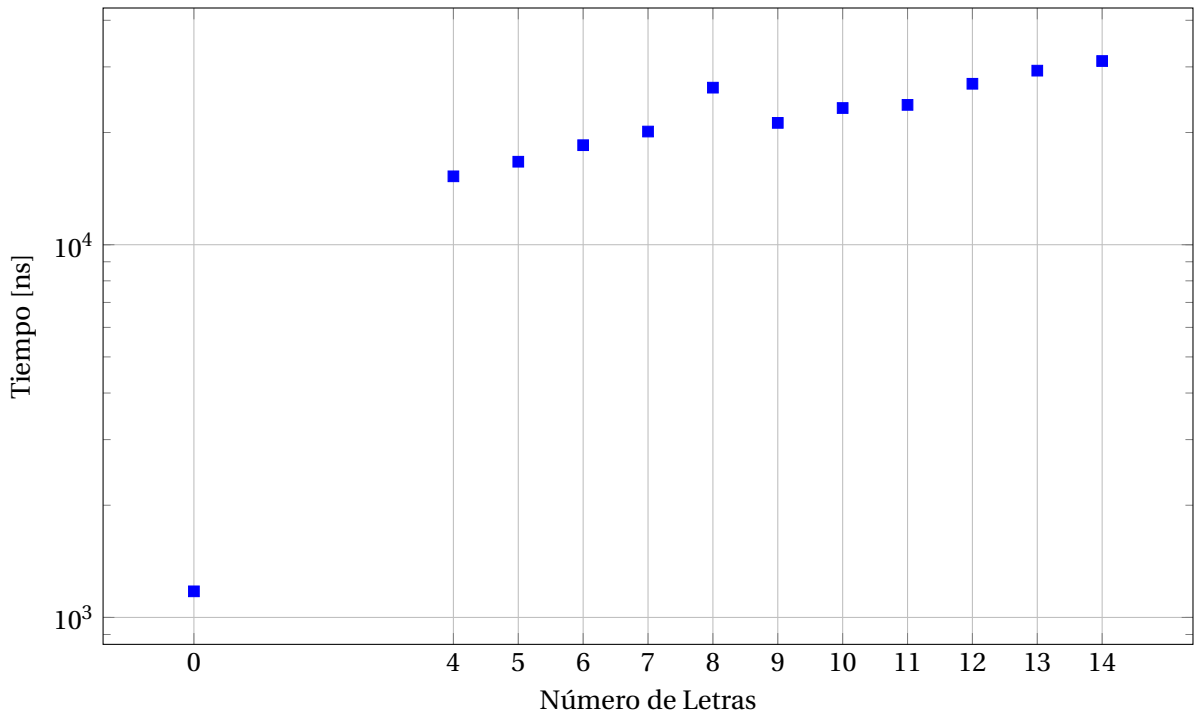

Figura 2: Grafico de Programacion Dinamica

La tabla y el grafico nos muestra una variacion muy pequeña en comparacion al enfoque estudiado anteriormente, aumentando en pequeñas cantidades dandole poca importancia a la cantidad de letras que se esten comparando.

A continuacion, se mostrara un grafico comparativo de los dos enfoques:

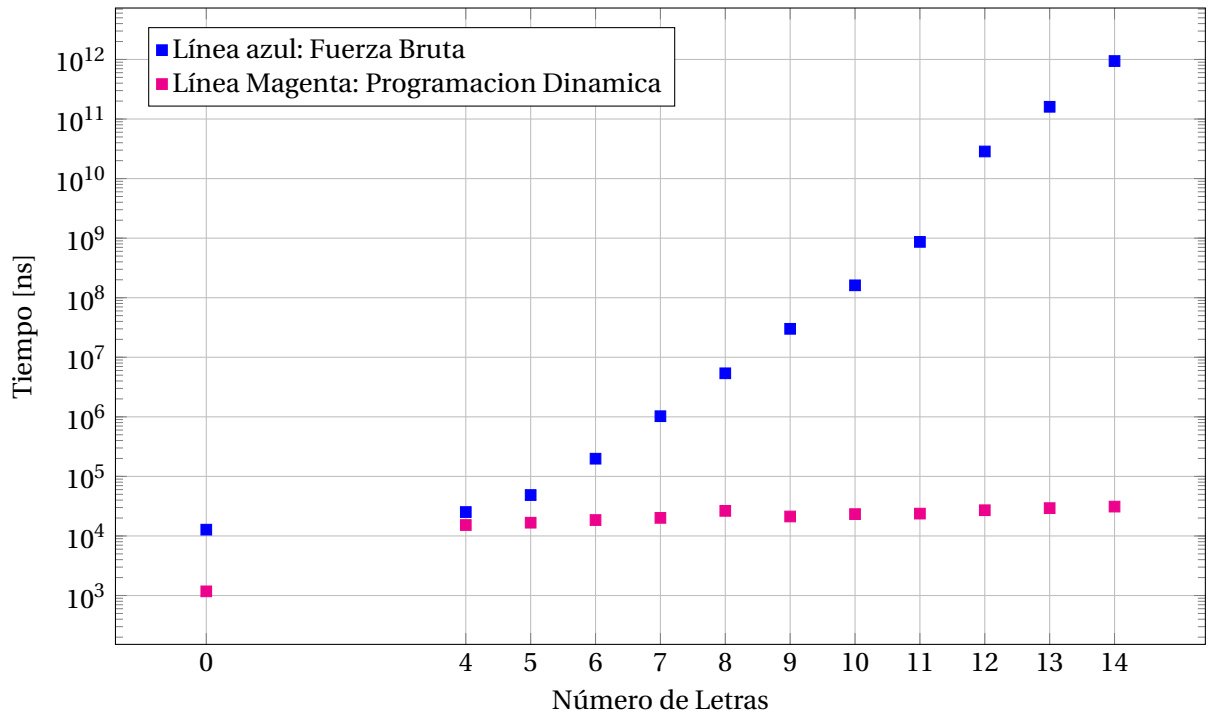

Figura 3: Grafico comparativo

Como podemos ver en el grafico comparativo, es clara la diferencia en el rendimiento de los dos enfoques que se estudiaron en este informe. Si bien fuerza bruta funciona, es el que peor tiempo da y crece de manera exponencial al aumentar la cantidad de letras en la palabra. En cambio, Programacion dinamica, su variacion del tiempo en funcion a la cantidad de letras en la palabra es muy poca.

5. Conclusiones

En este informe, se logró completar el objetivo planteado inicialmente, el cual era poder comparar dos enfoques de diseño de algoritmos. Podemos confirmar la diferencia en el rendimiento de los algoritmos gracias a los graficos obtenidos en la seccion de resultados.

Es posible analizar que es por la cantidad de memoria usada y las llamadas recursivas que fuerza bruta tiene un bajo rendimiento es cualquier caso. En cambio, programacion dinamica logra reducir estos costos por la matriz donde se guardan los datos, y asi evitar hacer calculos que ya se han hecho antes.

En conclusion, Programacion Dinamica demuestra que es un enfoque superior en problemas con calculos repetidos y rebundantes, ademas de almacenar calculos intermedios y trabajar con cadenas largas. Este estudio ayuda al estudiante a diseñar y optimizar algoritmos para asi obtener el mejor rendimiento dentro de lo que es posible.

6. Condiciones de entrega

A. Apéndice 1

Referencias

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

[2] GeeksforGeeks_community. *Introduction to Levenshtein distance*. Accessed: 2024-11-17. 2024. URL: https://www.geeksforgeeks.org/introduction-to-levenshtein-distance/.

[3] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201896850.

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