

REPORTE TAREA 2-3

ALGORITMOS Y COMPLEJIDAD

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

Alejandro Vergara

17 de noviembre de 2024

22:37

Resumen

En este informe se presenta un proyecto Informático, el cual consiste en el calculo de distancia entre cadenas extendido con enfoque de programación dinámica y fuerza bruta, se le agrego la función de transponer caracteres que estén seguidos, y también costos para realizar cada función, el informe consiste en una introducción que contextualiza mejor el problema, luego el diseño y análisis de los algoritmos a realizar para resolver el problema, donde se incluye pseudocodigo y complejidades, luego se habla sobre las dos implementaciones, donde se especifica como utilizar los programas, luego algunos experimentos con sus gráficos y su respectivo análisis, para finalizar con una conclusión.

REPOSITORIO

Índice

1. Introducción	2
2. Diseño y Análisis de Algoritmos	3
3. Implementaciones	9
4. Experimentos	10
5. Conclusiones	15

1. Introducción

En este Informe se solucionara el problema del calculo de distancia entre cadenas extendido, para la solución se utilizaran dos enfoques distintos, uno es programación dinámica y el otr fuerza bruta, programación dinámica consiste en probar todos los casos posibles pero ir guardando las soluciones óptimas de los sub-problemas, de esta manera no re-calcula los valores y solo los utiliza, siendo asi mas rápido al momento de ejecutarse, por otro lado tenemos a la fuerza bruta, que también calcula todas las opciones posibles pero sin guardarlas, por ende repetirá trabajo ya realizado, reduciendo el rendimiento, pero ahorrando espacio.

Este problema es muy importante, ya que puede ser utilizado para el reconocer si una palabra esta mal escrita o no, o para auto-correctores, que cuando la distancia sea muy pequeña se corrija, estas son herramientas que se utilizan en el día a día, sin embargo se ignora lo que hacen, solo es una caja negra que resuelve problemas, sin embargo este problema soluciona una parte de eso, este informe buscara que se comprenda mejor este algoritmo, como funciona y que estrategia de programación resulta mejor para el.

Este proyecto es muy significativo ya que ayuda a comprender mejor la diferencia entre programación dinámica y fuerza bruta, y también brinda la oportunidad de comparar sus tiempos de ejecución y espacio utilizado, pero resolviendo el mismo problema.

2. Diseño y Análisis de Algoritmos

Para los dos paradigmas de programación utilizados, se utilizo de estrategia la idea de calcular todos los costos posibles de forma recursiva, para finalmente elegir el mínimo, es decir, se tendrá una llamada recursiva cuando la función sea sustituir, insertar, eliminar o transponer, donde cada una obtendrá su respectivo coste, eligiendo así el mínimo entre estos.

Las diferencia con el problema de mínima distancia entre cadenas es que ahora se podrá ejecutar con distintos costos, y también se le agrega la función de transponer, donde a esta solo se le considero intercambiar con el carácter que tiene adelante solo si los dos caracteres resultantes quedan en la posición que deberían según la cadena 2,

por ejemplo:

palabra: ab

objetivo: ba

en este caso si se podrá ejecutar una transposición,

sin embargo si se tiene algo del tipo:

palabra: acb

objetivo: ba

no se podrá resolver con transposiciones, ya que primero necesitamos eliminar 'c' para que luego sea factible la transposición, sin embargo esto no estará soportado por los algoritmos

Se tomaron otros supuestos, los cuales son, si los dos caracteres a comparar son iguales, no se comparara con ninguna función (sustituir, insertar, eliminar o transponer), ya que como son iguales se considero que el costo sera 0.

La entrada consiste en dos cadenas de caracteres, la primera sera la que buscaremos transformar a la segunda, estas dos las etiquetaremos, la primera sera simplemente **palabra**, y la segunda sera **objetivo**, estas dos serán guardadas como variables globales para las dos implementaciones, también los costos de cada función estarán designados de forma matricial/vectorial que especificara cada costo específico dependiendo de cual/es caracteres estén implicados.

Por Ultimo tenemos que la palabra en realidad no se vera modificada, puesto que no es necesario para calcular el costo mínimo, solo es necesario saber que si se quiere eliminar un carácter el tamaño de la palabra se reducirá 1, si queremos insertar un carácter, se asumirá que se insertará el correcto, por ende revisaremos el siguiente carácter de objetivo con el actual de palabra, es decir se reducirá 1 a objetivo, si se quiere sustituir, hay que reducir palabra y objetivo en 1 puesto que ya cumplimos con ese carácter, y por ultimo si transponemos (se cumplen las condiciones) se reducirán los dos en 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 [2]

Para el enfoque de fuerza bruta tenemos dos variables globales (strings), una será la palabra y la otra el objetivo. Esta función recibe dos parámetros, **i**, tamaño de la palabra a resolver - 1, y **j**, que es el tamaño del objetivo a resolver - 1, es importante recalcar que la palabra se va subdividiendo, generando así todos los sub-problemas posibles y resolviéndolos con fuerza bruta.

Algoritmo 1: Distancia Mínima de Edición - Fuerza Bruta

```

1 String palabra, objetivo
2 Procedure DISTANCIAEDICIONFB(i, j)
3   if palabra vacia (i < 0) and objetivo vacio (j < 0) then
4     return 0
5   if palabra vacia (i < 0) then
6     return (costo insertar objetivoj) + DISTANCIAEDICIONFB(i, j-1)
7   if objetivo vacio (j < 0) then
8     return (costo eliminar palabrai) + DISTANCIAEDICIONFB(i-1, j)
9   if palabra[i] == objetivo[j] then
10    return DISTANCIAEDICIONFB(i-1, j-1)
11  eliminar ← DISTANCIAEDICIONFB(i-1, j) + (costo eliminar palabrai)
12  insertar ← DISTANCIAEDICIONFB(i, j-1) + (costo insertar objetivoj)
13  sustituir ← DISTANCIAEDICIONFB(i-1, j-1) + (costo sustituir palabrai por objetivoj)
14  transponer ← ∞
15  if i y j son > 0 and palabrai == objetivoj-1 and palabrai-1 == objetivoj then
16    transponer ← DISTANCIAEDICIONFB(i-2, j-2) + (costo transponer palabrai por palabrai-1)
17  return (minimo entre eliminar, insertar, sustituir, transponer)

```

Analizando este algoritmo tenemos que en el peor caso se generan 4 llamadas recursivas, las cuales son:

- 1 Eliminar: DistanciaEdicionFB(*i*-1, *j*)
- 2 Insertar: DistanciaEdicionFB(*i*, *j*-1)
- 3 Sustituir: DistanciaEdicionFB(*i*-1, *j*-1)
- 4 transponer: DistanciaEdicionFB(*i*-2, *j*-2)

por ende podemos generar una relación de recurrencia que se aproxima al problema , esta es:

$$T(i, j) = 4 * T(i - 1, j - 1)$$

ademas tenemos que $T(-1, -1) = 1$, esto ya que en este caso solo retornara 0, dejando que el problema trivial de dos palabras vacías $\in O(1)$, ahora si tenemos una palabra de tamaño n , y un objetivo de tamaño

m, tendremos que $T(n,m)$ ira aumentando en potencia de 4, es decir:

$$T(n, m) = 4 * T(n - 1, m - 1)$$

$$T(n, m) = 4 * 4 * T(n - 2, m - 2)$$

$$T(n, m) = 4 * 4 * \dots * T(-1, -1)$$

Hay que tener en cuenta de que la relación de recurrencia no se cumple siempre, por ejemplo si $i < 0$, tenemos que ya no habrán 4 llamadas recursivas sino 1, por ende como estamos considerando el peor caso tendremos una cota superior para la complejidad temporal, que será:

$$T(n, m) \in O(4^{\min(n,m)})$$

Para la complejidad espacial tenemos que, ignorando la utilización de matrices/vectores de costos, el algoritmo utilizara un espacio extra $O(1)$

Ejemplo de ejecución:

considerando los siguientes costos:

- $\text{costo_sub}(a, b) = 3$ si $a \neq b$ y 0 si $a = b$
- $\text{costo_ins}(b) = 1$ para cualquier carácter b
- $\text{costo_del}(a) = 1$ para cualquier carácter a
- $\text{costo_trans}(a, b) = 1$ para cualquier par de caracteres adyacentes a, b

y los strings:

- palabra = 'zcoal'
- objetivo = 'hola'

Con el propósito de resumir, solo se considerara el camino mas optimo para conseguir que la palabra se transforme en objetivo, comenzaremos llamando a la función con $i = 4$ y $j = 3$.

- DistanciaEdicionFB(4, 3)

Se ejecutaran las llamadas recursivas de eliminar, insertar y sustituir, pero al momento de entrar en el condicional para transponer nos damos cuenta de que si se respeta, logrando transponer la palabra y llamándose de forma recursiva con $i = 2$ y $j = 1$.

- DistanciaEdicionFB(2, 1)

Se comparan los dos caracteres, y notamos que son los mismos, por lo tanto solo se revisaran los siguientes caracteres $i = 1$ y $j = 0$.

- DistanciaEdicionFB(1, 0)

En este momento en palabra tendremos c y en objetivo h, por lo cual llama de forma recursiva a cada una de las funciones, pero no a transponer porque no cumple con las condiciones, aca se generan varios caminos pero se optara por que lo mejor sera eliminar, esto es porque ahora estamos buscando que 'zc' se transforme en 'h', con lo cual se podría simplemente sustituir c por h pero esto tiene un costo de 3, sale mas barato eliminar caracteres y luego insertarlos, lo cual tendría un costo de 2, por ende la llamada recursiva sera con $i = 0$ y $j = 0$.

- DistanciaEdicionFB(0, 0)

siguiendo lo expuesto anteriormente también eliminaremos este carácter, por ende la llamada recursiva sera con $i = -1$ y $j = 0$.

- DistanciaEdicionFB(-1, 0)

Ahora i pasa a ser negativo, esto indica que el sub-problema se limita a solo agregar los caracteres que faltan (if de la linea 5 pseudocodigo), como solo nos queda esta opción la llamada recursiva sera con $i = -1$ y $j = -1$.

- DistanciaEdicionFB(-1, -1)

Como los dos indices son negativos retorna 0, indicando que ha terminado.

Recopilando los costos efectuados tenemos un total de 4, que corresponden a:

terminar(0) + insertar(1) + eliminar(1) + eliminar(1) + transponer(1)

2.2. Programación Dinámica

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

Erickson, 2019 [1]

Para este enfoque se utilizó una tabla que indica los costos de cada sub-problema, haciendo que el programa realice menos llamadas recursivas, logrando un mejor rendimiento.

En esta implementación a palabra y a objetivo se les agrega un carácter al inicio, el cual es '-', esto es para simplificar el acceso a la matriz de sub-problemas, también tendremos la matriz que estará implementada con un vector de 2 dimensiones, esta estará inicializada con valores -1 en todos los puntos menos en el (0,0), en este punto valdrá 0, que es el caso base, la solución al problema trivial de palabra=" y objetivo = ".

Algoritmo 2: Distancia Minima de Edición - Programación Dinamica

```

1 String palabra, objetivo
2 Matriz tabla
3 Procedure DISTANCIAEDICIONPD(i, j)
4   if tablaij  $\neq$  -1 then
5     return tablaij
6   if palabra vacia (i == 0) then
7     tablaij  $\leftarrow$  (costo insertar objetivoj) + DISTANCIAEDICIONPD(i, j-1)
8     return tablaij
9   if objetivo vacio (j == 0) then
10    tablaij  $\leftarrow$  (costo eliminar palabrai) + DISTANCIAEDICIONPD(i-1, j)
11    return tablaij
12  if palabra[i] == objetivo[j] then
13    tablaij  $\leftarrow$  DISTANCIAEDICIONPD(i-1, j-1) return tablaij
14  eliminar  $\leftarrow$  DISTANCIAEDICIONPD(i-1, j) + (costo eliminar palabrai)
15  insertar  $\leftarrow$  DISTANCIAEDICIONPD(i, j-1) + (costo insertar objetivoj)
16  sustituir  $\leftarrow$  DISTANCIAEDICIONPD(i-1, j-1) + (costo sustituir palabrai por objetivoj)
17  transponer  $\leftarrow$   $\infty$ 
18  if i y j son >0 and palabrai == objetivoj-1 and palabrai-1 == objetivoj then
19    transponer  $\leftarrow$  DISTANCIAEDICIONPD(i-2, j-2) + (costo transponer palabrai por palabrai-1)
20  tablaij  $\leftarrow$  (minimo entre eliminar, insertar, sustituir, transponer)
21  return tablaij
22 Procedure SET()
23   ...
24   palabra  $\leftarrow$  " " + leertxt
25   objetivo  $\leftarrow$  " " + leertxt
26   setear tabla con valores -1
27   tabla[0][0]  $\leftarrow$  0
28   ...

```

El algoritmo con el enfoque de programación dinámica, específicamente utilizando el enfoque top down, es muy similar a fuerza bruta, de hecho de igual forma se calculan todos los caminos posibles, pero en este con este enfoque no se volverá a re-calcular, sino que los resultados parciales (resultados del sub-problema) se irán guardando, ahorrando así llamadas recursivas y mejorando el rendimiento. Este algoritmo divide el problema en varios sub-problemas, dependiendo de la última función que se ejecute a la palabra, la manera más fácil de verlo es que se fija el último carácter y se decide hacer una operación, en donde si se agrega, o se sustituye un carácter, siempre se considerará que es el que beneficia al problema, por ende tendremos una fracción del problema resuelto, faltaría el resto, ese sería el primer sub-problema.

Para la complejidad temporal tenemos que lo que buscaremos realizar es llenar la matriz con los sub-problemas, esta tendrá un tamaño de $n * m$ donde n es el tamaño de palabra y m es el tamaño de objetivo, en conclusión su complejidad será:

$$T(n, m) \in O(n * m)$$

Por otro lado para la complejidad espacial tendremos que de espacio extra (ignorando espacio utilizado

para los costos) tendremos la matriz de sub-problemas, la cual consta de n filas y m columnas, por ende el espacio estará definido por:

$$E(n, m) \in O(n * m)$$

Ejemplo de ejecución: considerando los siguientes costos:

- $\text{costo_sub}(a, b) = 3$ si $a \neq b$ y 0 si $a = b$
- $\text{costo_ins}(b) = 1$ para cualquier carácter b
- $\text{costo_del}(a) = 1$ para cualquier carácter a
- $\text{costo_trans}(a, b) = 1$ para cualquier par de caracteres adyacentes a, b

y los strings:

- palabra = 'abc'
- objetivo = 'bad'

como el tamaño de palabra y de objetivo es 3, sin embargo le agregaremos el '-' antes, que como ya fue explicado anteriormente es para acceder de manera mas sencilla a la matriz, sabiendo esto los dos strings quedaran de largo 4, por ende comenzaremos llamando a la función con $i = 3$ y $j = 3$.

Para este ejemplo el algoritmo se comportara igual que el de fuerza bruta e ira llenando la matriz de a poco, en este problema es visible que la menor distancia es eliminar 'c', agregar 'd', transponer 'ab', con un costo total de 3, el algoritmo generara la siguiente matriz:

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ 1 & 2 & 1 & 2 \\ 2 & 1 & 1 & 2 \\ 3 & 2 & 2 & 3 \end{array} \quad (1)$$

de la cual podemos apreciar que el algoritmo efectivamente revisa todos los i y los j posibles, y que el resultado será 3 ($\text{DistanciaEdicionPD}(2,3) + 1$), como ya sabemos el resultado, podemos seguir el camino de regreso, como la primera operación fue 'eliminar' tenemos que ir a (2, 3), que guarda el valor 2, esto nos indica que $\text{DistanciaEdicionPD}(2,3) = 2$, luego la siguiente operación fue 'agregar' en consecuencia visitaremos la matriz en (2,2), que es 1, entonces $\text{DistanciaEdicionPD}(2,2) = 1$, por ultimo la operación que viene sera transponer, la cual tiene un costo de 1, que viene de $\text{DistanciaEdicionPD}(0,0) + 1$, pero como $i=0$ y $j=0$, el resultado es 0

3. Implementaciones

Ahora se especificara mas acerca de las implementaciones de estos programas, estas se encuentran en el siguiente [repositorio](#), en este se encuentra la información de todo lo relacionado con el informe.

La implementación tiene 4 archivos de gran importancia, estos son:

- `costos.cpp`:

En este sub-programa se tendrán las implementaciones de los costos de las funciones.

- `entrada.txt`:

Aquí se ingresara en la primera linea la palabra, y en la segunda el objetivo para resolver el problema.

- `FuerzaBruta/distanciaDeEdicion.cpp`

En este archivo se guarda la implementación con fuerza bruta del problema, esta posee dos funciones, las cuales son `distanciaEdicion(i,j)`, y `set()`, la primera resuelve el problema con fuerza bruta revisando todos los casos posibles, la segunda lee los strings del archivo `entrada.txt`.

- `prg_dinamica/distanciaDeEdicion.cpp`

En este archivo se guarda la implementación con programación dinámica del problema, esta posee dos funciones, las cuales son `distanciaEdicion(i,j)`, y `set()`, la primera resuelve el problema con programación dinámica revisando todos los casos posibles pero sin re-calcularlos, y la segunda lee los strings del archivo `entrada.txt` y a estas se les agrega un '-' al inicio, esto es solo para facilitar el acceso a la matriz, no perjudica a la entrada.

También se necesitan 4 archivos txt para los costos, estos tendrán un tamaño donde $n = 26$ (alfabeto inglés, solo con letras minúsculas) estos son; `cost_insert.txt`, `cost_delete.txt`, `cost_replace.txt`, `cost_transpose.txt`, estos estarán guardados en la carpeta `costos`, y cada uno guardara n (insert, delete) o $n * n$ (replace, transpose) números enteros, de igual manera en el repositorio hay un generador de costos aleatorio.

Para correr los programas, solo hay que modificar `entrada.txt`, ingresando la palabra y el objetivo deseados, luego desde la carpeta **tarea2-3** compilar con:

```
g++ -o out fuerzabruta/distanciadeedicion.cpp -Wall
```

si se quiere por fuerza bruta, o

```
g++ -o out prg_dinamica/distanciadeedicion.cpp -Wall
```

si se quiere por programación dinámica, luego para ejecutar tan solo ingresar el comando:

```
.\out
```

4. Experimentos

“Non-reproducible single occurrences are of no significance to science.”

—Popper, 2005 [3]

Para los experimentos realizados, se utilizaron costos fijos, es decir, para todos los casos se utilizaron las mismas matrices y vectores. Estos fueron generados aleatoriamente con el archivo ‘costos.cpp’ y están guardados en el [repositorio](#).

Si se quiere utilizar ‘costos.cpp’, hay que tener en cuenta que este funciona con rangos de valores: se tiene que ingresar el valor mínimo y el valor máximo. Además, su salida se guarda en una carpeta llamada **costos**, que contiene los archivos de texto con los valores.

También, en algunos casos, se crearon palabras aleatorias con el archivo ‘entrada_generator.py’. En este programa se necesita ingresar el string objetivo y generará el string palabra, el cual contiene los caracteres generados según los parámetros dados.

hardware con las siguientes especificaciones:

- Procesador Intel Core i3-12100F
- RAM 16 GB DDR4
- Almacenamiento ssd SATA
- SO windows 10 pro, pero los algoritmos fueron ejecutados en la maquina virtual de ubuntu 18.04.6
- Librerías utilizadas: iostream, string, fstream, random, vector, algorithm, climits, chrono, sstream
- Los algoritmos fueron compilados utilizando g++ 7.5.0

4.1. Dataset (casos de prueba)

Los experimentos realizados fueron muy variados, cada uno tiene un enfoque particular, esto debido a que el problema tiene muchas variables que son importantes de analizar, los enfoques utilizados fueron:

- 1 objetivo fijo, este fue "paralelepipedo" de tamaño 14, y la palabra varió con el proposito de que se ejecutara solo una función, es decir eliminar, insertar, sustituir o transponer, las palabras de entrada fueron "paralelepipedoa", "paralelepiped", "paralelepipedwo", "paralelepiped".
- 2 objetivo fijo, este fue "paralelepipedo" de tamaño 14, y la palabra se definio aleatoriamente (con el programa de python), de un tamaño ± 2 con el proposito de analizar como se comporta en casos mas aleatorios.
- 3 objetivo fijo, este fue ".^bcabcbca" de tamaño 10, y ahora se fueron eligiendo palabras aleatorias (con el programa de python), y se comenzo con que estas tuvieran un tamaño de 5 y luego se fue

aumentando de 5 en 5 hasta llegar a tamaño 20, esto para ver como se comporta el tiempo de ejecucion en base al largo de la entrada.

- 4 este enfoque es muy parecido al anterior, sin embargo se utiliza un string objetivo que es `."abcdefghij"`, esto genera mas variedad en los caracteres que se utilizaran en la palabra, y se tiene el proposito de ver como aumenta el tiempo de ejecucion con strings con caracteres mas variados.
- 5 este enfoque busca hacer crecer la palabra y el objetivo al mismo tiempo, y para estos strings se utilizaron los caracteres `'a', 'b', 'c'`, esto para ver que tanto afecta que los dos strings aumenten.
- 6 este enfoque busca hacer hacer lo mismo que el anterior pero con mas varianza en los caracteres de los strings, objetivo va aumentando en orden alfabetico.

Por ultimo, en el [repositorio](#) hay un archivo excel que contiene todos los datos medidos, la palabra utilizada, el tiempo medido, el resultado de distancia de edicion, el espacio utilizado y los graficos hechos.

4.2. Resultados

En los resultados tenemos que, para el primer enfoque, los tiempos con programación dinámica son muy bajos comparados con fuerza bruta, y del gráfico podemos ver que la eliminación fue el que mas se tardo, esto se atribuye a que para que solo sea una eliminación, hay que tener 1 carácter de mas en palabra, lo cual hace que su longitud sea mas larga, teniendo asi mas casos de prueba, por otro lado tenemos que sustituir es el que menos tarda, y esto se debe a que como las palabras son muy similares, después de sustituir, todo sera igual, no haciendo asi, mas llamadas recursivas.

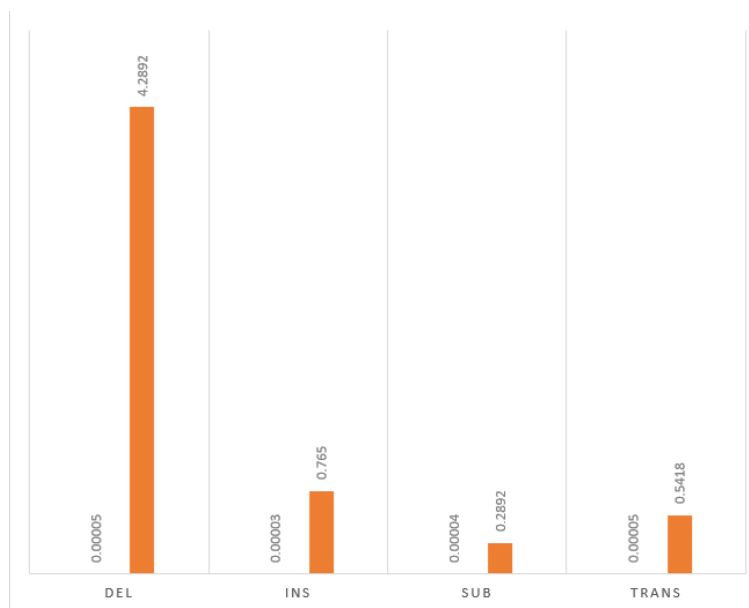


Figura 1: Gráfico para funciones específicas

Con el segundo enfoque se obtiene que al igual que en enfoque anterior, pero en este se puede pre-senciar de mejor manera como crece exponencialmente el enfoque de fuerza bruta, y como programa-

ción dinámica crece mínimamente, por otro lado el espacio se mantiene constante, en general fuerza bruta utilizo menos espacio que programación dinámica, que es lo que se esperaba en un inicio, sin embargo a veces resultaba dar distintos espacios para una misma entrada, esto se debe a que el sistema operativo busca la mejor manera de manejar la memoria dinámica (Heap)

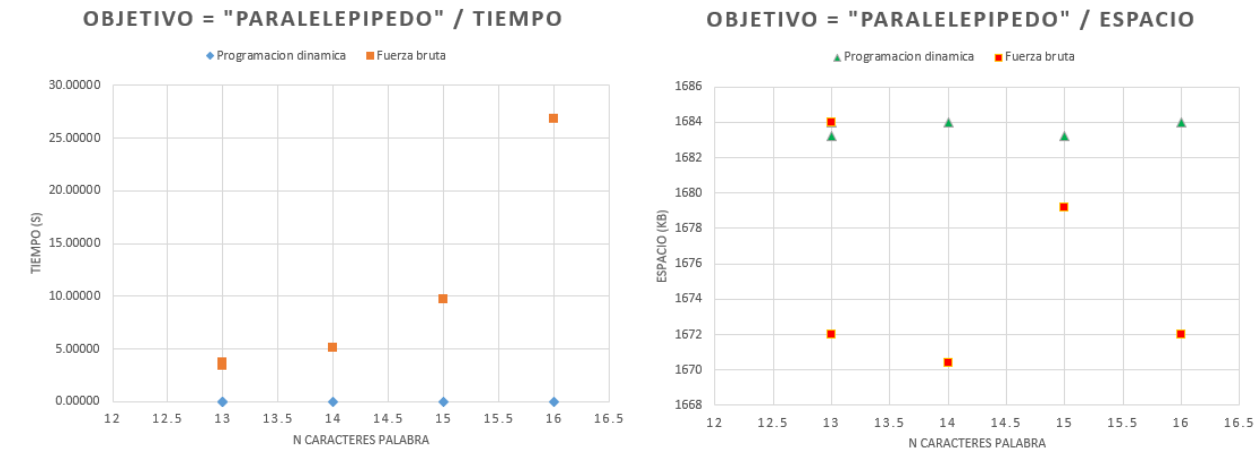


Figura 2: Gráficos con palabras aleatorias para objetivo paralelepipedo

Ahora para los siguientes gráficos se combinaran los enfoques para comparar de la forma mas rigurosa posible, con los enfoques donde la palabra crece se genero el siguiente gráfico para programación dinámica y fuerza bruta:

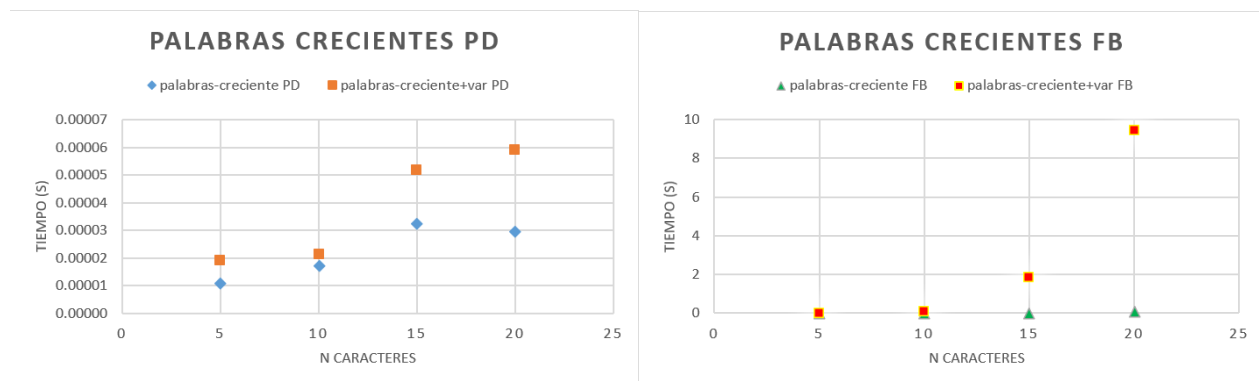


Figura 3: Gráficos para palabras de largo creciente con objetivo fijo

Estos gráficos se separaron para que sea mas fácil ver como crece programación dinámica, que antes parecía ser constante, aquí se revela que no lo es, se comporta de forma mas lineal, aunque el gráfico no es muy exacto porque las palabras para ir aumentando tienen que cambiar.

también se observa que si los caracteres son poco variados, el programa funcionara mejor con cualquiera de los enfoques, de hecho para fuerza bruta, si los caracteres son variados se comporta mucho mas exponencial que si no lo son, esto se debe a que es mas probable que en algún momento se encuentre algún carácter igual, reduciendo las llamadas recursivas, y por tanto reduciendo los tiempos.

Por otro lado se ve la gran diferencia que hay entre fuerza bruta y programación dinámica, esta esta a una escala mucho menor, sus mayor valor no alcanzan a ser 0.001 segundos, mientras que con fuerza bruta, para palabras de 15 caracteres ya tarda 2 segundos.

Los siguientes gráficos muestran como se comportan los tiempos cuando palabra y objetivo tienen el mismo largo y este va variando, aquí ya se pierde mucho la exactitud porque palabra y objetivo tienen que ir cambiando para aumentar su tamaño, lo cual hace que los problemas sean distintos.

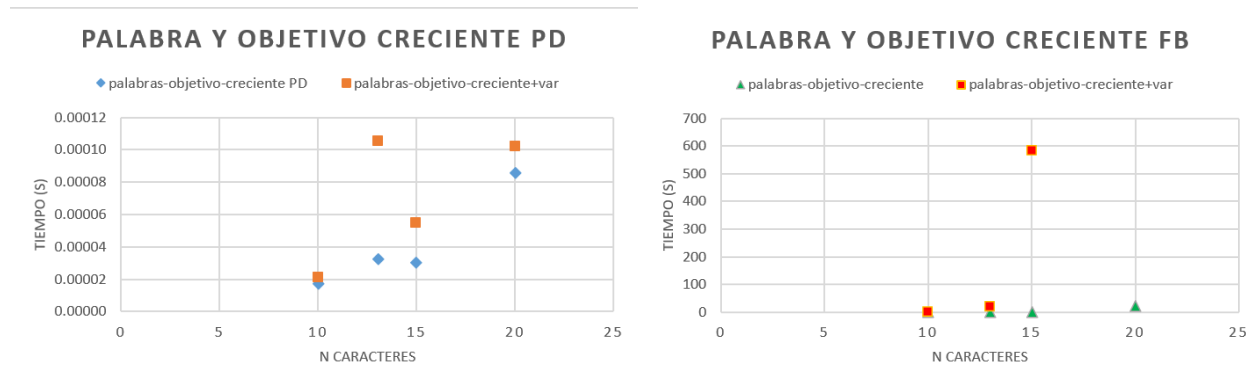


Figura 4: Gráficos para palabras y objetivos de largo creciente

de estos se vuelve a observar que los strings con caracteres mas variados tardan mas, de hecho para fuerza bruta con 20 caracteres variados no termino en 3 horas, lo cual hace que fuerza bruta sea un algoritmo muy deficiente, en comparación con programación dinámica.

También se nota la reducción de exactitud, por ejemplo, en el punto donde hay 13 caracteres en programación dinámica los dos puntos (con menos y mas varianza) se alejan demasiado entre si en comparación con las otras medidas, de igual manera todos los puntos de mayor varianza tardaron mas que los de menor varianza.

Por ultimo tenemos un gráfico para comparar los espacios utilizados por los distintos algoritmos y enfoques:

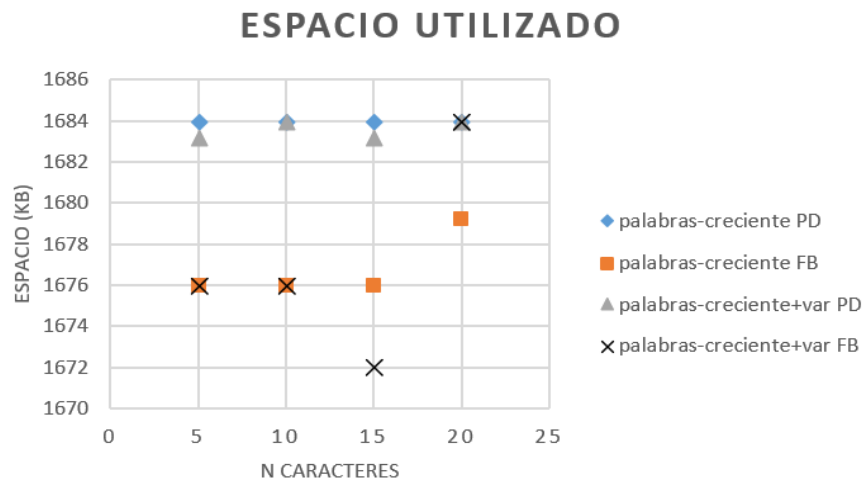


Figura 5: Gráfico para comparar espacios utilizados

De este gráfico se observa que en general programación dinámica utiliza 8 KB mas que fuerza bruta, además parecieran ser constantes, sin embargo teóricamente esto no es así, pero esto pasa por como el sistema operativo reparte los espacios en la memoria RAM para la ejecución del programa, logrando que no avance de manera continua sino de manera discreta por bloques, como las pruebas fueron realizadas con strings relativamente pequeños esto no se puede visualizar.

5. Conclusiones

Como conclusión se tiene que programación dinámica es muy parecida a fuerza bruta, sin embargo, es mucho mas eficiente en lo que a tiempo respecta, si nos enfocamos en el espacio que se utiliza, es mejor fuerza bruta, pero sabiendo esto ¿cual es el mejor?, la respuesta mas rigurosa es que depende, depende del uso y de la memoria que tiene el dispositivo que ejecutara el problema, pues si un dispositivo no tiene mucha memoria, el hardware limitara a el algoritmo haciéndolo mas deficiente, ademas si se utiliza fuerza bruta lo esperable es que se tenga que esperar un poco mas, pero si se sabe que los strings de entrada no pasan los 10 caracteres, sabremos que fuerza bruta y programación dinámica tardan tiempos muy parecidos.

Sabiendo esto podremos concluir que programación dinámica es mejor si los strings son largos, pero si son cortos lo mejor sería fuerza bruta, ya que de esta forma no se perdería espacio, ni tampoco se estaría sacrificando tanto tiempo.

Por ultimo también se concluye que no solo es importante como se implemente un programa, sino que como sera utilizado, en este caso vimos que si los strings eran variados los programas tardaban mas, y si no lo eran, resultaban ser mas rápidos

Referencias

- [1] Jeff Erickson. *Algorithms*. Jun. de 2019. ISBN: 978-1-792-64483-2.
- [2] 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.
- [3] K. Popper. *The Logic of Scientific Discovery*. Routledge Classics. Taylor & Francis, 2005. ISBN: 9781134470020.
URL: <https://books.google.cl/books?id=LWSBAAQBAJ>.