Reporte de resultados
Proyecto Investigación de Operaciones

Kevin Ledezma Jiménez
Ingeniería en Computación
Instituto Tecnológico de Costa Rica
Heredia, Costa Rica
kledezma204@gmail.com

Abstract*—Este documento está enfocado en analizar los resultados obtenidos durante este proyecto, en el cual implementamos el algoritmo de la *"mochila"* en el lenguaje *python*, tanto de manera *fuerza bruta* así como *implementando programación dinámica*. También se implementó el análisis de hileras por medio del método *alineamiento* utilizando el algoritmo *Needleman-Wunsch

I. Introducción

El proyecto consta de dos partes: la primera es el programa computacional que resuelve los problemas de contenedor (mochila) y el problema del alineamiento de secuencias. Para esto, nos enfocaremos en dos técnicas para el problema del contenedor: algoritmo de fuerza bruta y algoritmo utilizando programación dinámica. Además, para el problema del alineamiento, también se mostrará el algoritmo *Needleman-Wunsch* que utiliza programación dinámica para resolver y dar puntajes de comparación a secuencias de hileras.

II. Problema del contenedor

El problema del contenedor es bastante "simple" pero no fácil. Se trata de simular tener un contenedor (o una mochila) y nos dan un listado de objetos, los cuales tienen un valor (beneficio) y un peso dentro del contenedor. Nuestro objetivo es conseguir la mayor cantidad de objetos maxificando nuestro valor final (ganancia). Para lograr esto, existen dos enfoques: Fuerza bruta y Algoritmo aprovechando recursos de programación dinámica. Para lograr una ejecución de este ejercicio, se necesitarán los siguientes datos: peso máximo que la mochila puede cargar, y la información de cada objeto, tal que: sepamos cuanto pesa cada objeto, cuanto beneficio nos otorga ese objeto además de saber si estamos bajo el escenario donde existen muchas copias de este objeto (es decir, tomar 3 celulares de idénticos).

A. Algoritmo de Fuerza Bruta aplicado a Contenedor

Primero, podemos observar la simpleza que ofrece un algoritmo de fuerza bruta para resolver este ejercicio.

$$V[k,w]=\begin{cases} V[k-1,w] & si\ w_i > W \\ max\{V[k-1,w], V[k-1,w-w_k]+b_k\} & sino \end{cases}$$

Imagen tomada del curso Investigación de Operaciones, ITCR 2020

En pocas palabras, tenemos dos verificaciones: llevar o no llevar el objeto. Si no es posible llevarlo, lo omitimos y seguimos al siguiente. Sin embargo, si es posible llevarlo, haremos un *max* de ambos escenarios: tomando el objeto y no llevándolo para analizar más casos.

No obstante, esta solución tiene un complejo costo computacional, ya realiza dos llamadas recursivas por cada elemento que analiza en la mochila, esto provoca una complejidad computacional de $O(n*k)$[1], donde k es la cantidad de elementos a analizar y su complejidad se vuelve muy complicada entre más objetos deseemos analizar.

B. Algoritmo de Programación Dinámica aplicado a Contenedor

Cuando analizamos este mismo problema, haciendo uso de la programación dinámica *el famoso memoize*, notamos una gran mejoría tanto en rendimiento y hasta en implementación para comprender el funcionamiento de esta solución.

```
for i=1 to n
  for w=0 to W
  if w_i>w
    V[i,w]=V[i-1,w]
  else
    if  b_i+V[i-1,w-w_i] > V[i-1,w]
     V[i,w]=b_i+V[i-1,w-w_i]
    else
     V[i,w]=V[i-1,w]
```

Imagen tomada del curso Investigación de Operaciones, ITCR 2020

Aquí podemos notar una gran diferencia en implementación, empezando por que no hacemos llamadas recursivas, sino que optamos por hacer ciclos iterativos *for* sobre una matriz que nos ayudará a recordar información importante como los objetos que hemos decidido y los que no. Además, nos otorga una mejor duración de ejecución, como podemos observar en las siguientes imágenes, tomadas utilizando el código creado para este proyecto como base de la ejecución.

```
Fuerza Bruta
Resultado:  260
¡Programa Finalizado!
Segundos transcurridos 0.0014951229095458984
```

Imagen tomada como resultado de la implementación de fuerza bruta

```
Ejecutando Programacion Dinamica para Mochila
Resultado:  260
¡Programa Finalizado!
Segundos transcurridos 0.0009868144989013672
```

Imagen tomada como resultado de la implementación de programación dinámica

Tal vez la diferencia en la duración es muy pequeña *también debido a que nuestro computador es bastante robusto en términos de componentes*, pero prestando atención, podemos ver como el tiempo de ejecución del algoritmo de programación dinámica es menor en aproximadamente la mitad del tiempo incluso siendo fracciones de segundo. Esto connota una gran eficiencia computacional, ya que conforme el algoritmo de fuerza bruta consuma recursos, podremos ahorrar recursos y tiempo de manera exponencial con el algoritmo de programación dinámica.

C. Comparativa de Algoritmos

Es muy curioso, que al realizar pruebas de ejecución en estas implementaciones, al ser un ejercicio de pocas entradas (es decir, pocos elementos de donde decidir) parece ser que el algoritmo de programación dinámica no es nada eficaz y hasta menos eficiente que fuerza bruta.

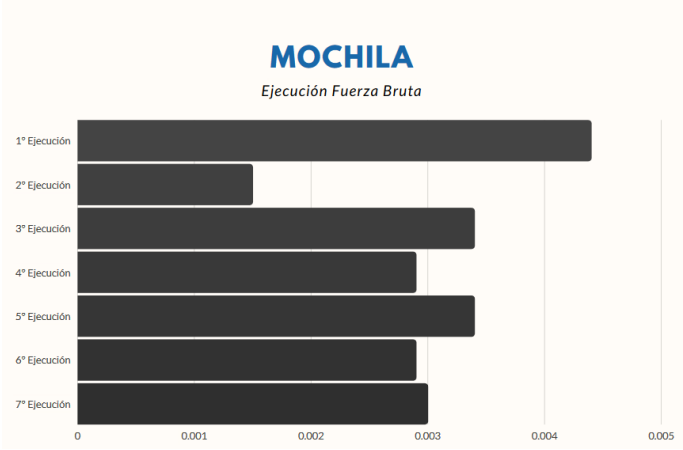

Imagen creada como resultado de siete ejecuciones usando fuerza bruta en un ejercicio de pocos elementos

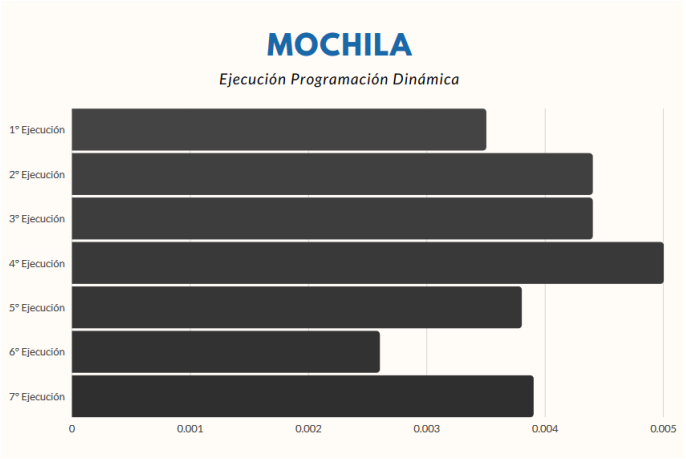

Imagen creada como resultado de siete ejecuciones usando programación dinámica en un ejercicio de pocos elementos Nótese que las escalas de la barra horizontal inferior son las mismas y vemos una mayor duración en general por parte del algoritmo de programación dinámica, sin embargo, analicemos que sucede cuando ejecutamos ejercicios con grandes cantidades de elementos.

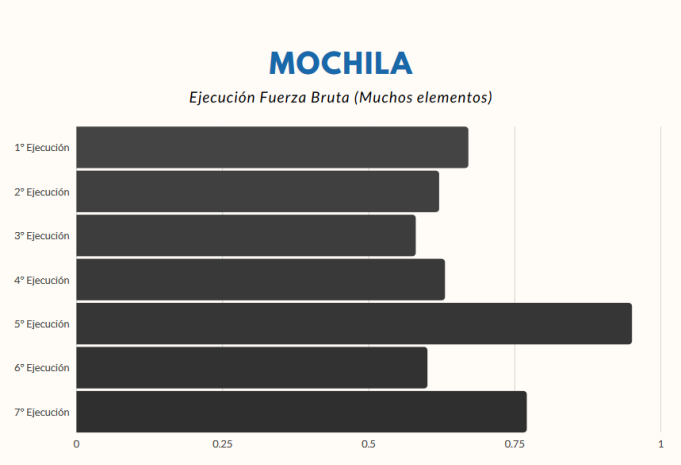

Imagen tomada como resultado de siete ejecuciones usando fuerza bruta en un ejercicio de 40 elementos

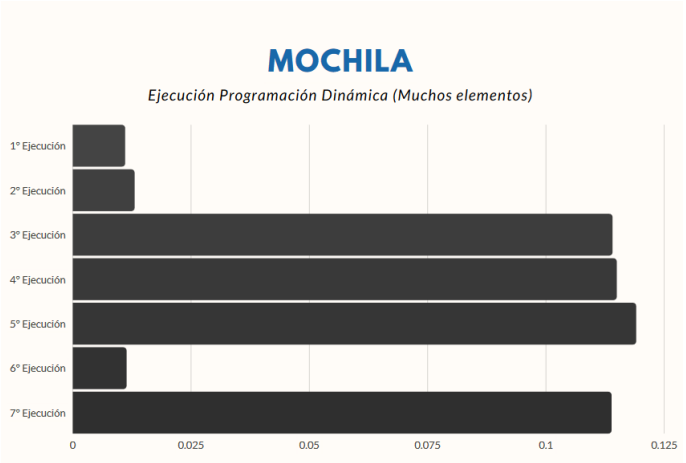

Imagen tomada como resultado de siete ejecuciones usando programación dinámica en un ejercicio de 40 elementos

Aunque las barras de duración parezcan similares, pongamos especial atención a la *barra horizontal inferior* donde vemos que los tiempos de ejecución son verdaderamente inferiores utilizando programación dinámica y podemos afirmar que ese es el verdadero beneficio que otorga este algoritmo.

III. Problema del Alineamiento

Para este ejercicio, tomamos como base la *bio-informática* aplicada a hileras similares a los nucleótidos del ADN *(ACTG)* con la idea de encontrar la similitud entre cadenas conformadas por estos datos. Sin embargo, existen dos maneras de acercarse a un problema de esta magnitud con algoritmos de fuerza bruta o algoritmos que utilizan programación dinámica. En este proyecto, realizaremos el algoritmo de fuerza bruta así como un algoritmo de programación dinámica: *Algoritmo Saul Needleman y Christian Wunsch*. Utilizando como parámetros, dos simples líneas indicando las hileras a ser procesadas, además, se utilizará un *Scoring* de *MATCH: 1 — MISSMATCH: -1 — GAP_PENALTY: -2* al realizar las valoraciones de las hileras. Una vez explicados e implementados, demostraremos sus principales diferencias por medio de tiempos de ejecución así como una captura de los resultados según nuestra consola de ejecución.

A. Algoritmo de Fuerza Bruta aplicado a Alineamiento

Para este algoritmo, nos enfocaremos en "encontrar la secuencia común de mayor tamaño entre dos secuencias X e Y de longitudes *m* y *n*, respectivamente. Para ello se consideran todas las subsecuencias posibles de X (2^m) y se comparan con todas las subsecuencias posibles de Y (2^n)"[2]

De esta manera, podemos ver la increíble duración que posee este algoritmo, al punto de que, con hileras medianamente grandes *unos 10 caracteres* ya se hace imposible su ejecución.

```
Caso alineamiento fuerza bruta
Listas ingresadas:
ATTGTGATTC**********
TTGCATCGGC**********
Las hileras son demasiado grandes y ya me quedé sin memoria!!
MEMORY ERROR
¡Programa Finalizado!
Segundos transcurridos 29.93369174003601
```

Imagen tomada como resultado de una ejecución con hileras de 10 elementos

Además, nótese el tiempo de duración según el reloj interno de al computadora, aproximadamente 30 segundos antes de que la ejecución ejecutara la excepción *Memory Error.*

Incluso, hemos sido óptimos utilizando un código de implementación, que evita la duplicación de hileras a la hora de realizar permutaciones con la fe de optimizar en la medida de lo posible la ejecución aunque siempre habrá un punto de quiebre para el programa.

B. Algoritmo Saul Needleman y Christian Wunsch aplicado a Alineamiento

De la misma manera, el algoritmo de Saul Needleman y Christian Wunsch consta de "garantizar la obtención del mejor alineamiento sin tener que realizar todos los alineamientos posibles entre las dos secuencias. Para saber cuál es el mejor alineamiento de dos secuencias (X e Y) de longitud m y n, respectivamente, es necesario definir un sistema de puntuación que favorezca las coincidencias y penalice las diferencias y los huecos. "[2]

¿Qué piensa que será el resultado de ejecutar las mismas hileras de 10 caracteres bajo esta implementación? Veamos!

```
Caso alineamiento programacion dinamica
Listas ingresadas:
attgtgattc
ttgcatcggc
Scoring Obtenido:
-4
¡Programa Finalizado!
Segundos transcurridos 0.0014657974243164062
```

Imagen tomada como resultado de una ejecución con hileras de 10 elementos

Para nuestra fortuna, el algoritmo funciona maravillosamente, logramos obtener un *Scoring* bastante rápido y directo, de una complejidad bastante lineal ya que solo debemos componer una matriz y empezar a recorrerla, evitando así quedarnos sin memoria a la hora de ejecutar nuestro programa.

C. Comparativa en tiempos de ejecución

Primero, hemos realizado siete ejecuciones con las siguientes dos hileras: *CTGCATT* y *CGTCAGG*, ahora, analicemos el comportamiento en los tiempos de ejecución por parte del algoritmo de fuerza bruta.

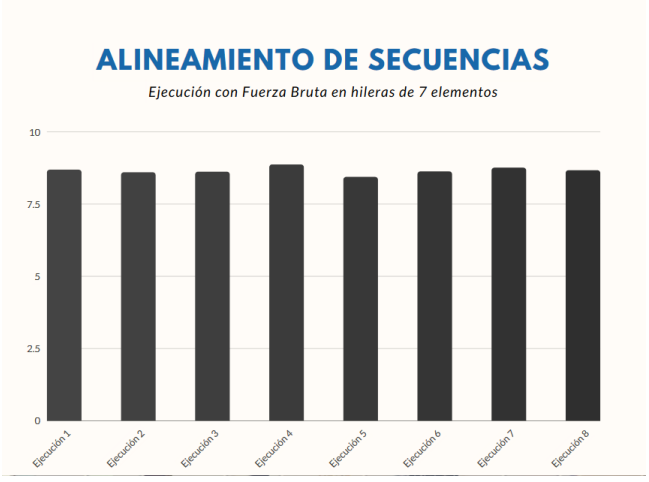

Imagen creada como resultado de ocho ejecuciones sobre el algoritmo de fuerza bruta

Podemos notar con facilidad, que los tiempos de ejecución tuvieron una media de *8.3 segundos*, demasiado alto para estar siendo ejecutado en una máquina de gran poder. Además, se sabe que aumentando los elementos de la hilera a 10, obtenemos un *Memory Error.*

Ahora, es turno de analizar el contexto de esta ejecución sobre el algoritmo de Programación Dinámica

Imagen creada como resultado de cinco ejecuciones sobre el algoritmo de programación dinámica *(Programación dinámica ejecuciones del 1 al 4, Fuerza bruta ejecuciones del 5 al 8)*

Aunque no se hayan realizado ocho ejecuciones, con tan solo 5 de ellas podemos ver la enorme diferencia entre los tiempos de ejecución.

IV. Conclusiones

Es demasiado evidente, lo necesario que es contar con algoritmos de programación dinámica para mejorar el alcande de la ejecución ante estos procesos tan complejos en costo computacional.

Asimismo, también hemos visto que estos algoritmos se pueden mejorar tanto en nuevas implementaciones como en obtener información al respecto, como saber los elementos que han sido guardados en el contenedor así como las posibles rutas que conforman hileras óptimas de nucleótidos.

De la misma manera, aconsejo a cualquier estudiante o persona que esté estudiando una carrera afín a la programación, familiarizarse con los conceptos que ofrece la programación dinámica, no solo por los inmensos aportes que ha otorgado a la Investigación de Operaciones, sino que también te ayudará a ser un mejor profesional y tener una gran apreciación por realizar código óptimo y eficiente.

References

[1] Velasco, "NP-Completeness", *Elisa.dyndns-web.com*, 2014. [Online]. Available: https://elisa.dyndns-web.com/teaching/opt/comb/knapsackOC.pdf. [Accessed: 14- Dec- 2020].

[2] ma, "ALINEAMIENTO DE DOS SECUENCIAS", Ehu.eus. [Online]. Available: *http://www.ehu.eus/biofisica/juanma/bioinf/pdf/1_pairwise.pdf*. [$Accessed: 15 - Dec - 2020$].