Parcial práctico 2

ANÁLISIS Y DISEÑO DE ALGORITMOS

Carlos Alberto Álvarez Henao

Isabella Hernández Posada
Hellen Yanes Doria

INTRODUCCIÓN EJERCICIO 1

El problema del cambio de monedas trata sobre la combinación mínima de monedas que sumen un monto M, dado un conjunto de denominaciones D. Este ejercicio va a comparar dos enfoques para resolverlo el algoritmo codicioso (greedy), que siempre elige la moneda más grande disponible, y el algoritmo de fuerza bruta, que explora todas las combinaciones posibles para encontrar la óptima. El objetivo de este primer ejercicio es identificar cuándo el enfoque greedy nos da la solución mínima y en qué casos falla, poniendo atención en los sistemas no canónicos de monedas.

1. Metodología

Para resolver el problema del cambio de monedas utilizamos dos funciones en C++. La primera es cambio_greedy(M, D), que se basa en un enfoque codicioso. Básicamente, lo que hace esta función es ordenar las monedas de mayor a menor, y en cada paso elige la moneda más grande posible, pero siempre teniendo en cuenta que no se pase del monto que queda por cubrir. Este algoritmo toma decisiones rápidas, intentando llegar a una solución óptima.

La segunda función es cambio_bruteforce(M, D), y en este caso se usa la fuerza bruta. Esta genera todas las combinaciones posibles de monedas que suman exactamente M, sin repetir, y de forma ordenada. Al final, se queda con la combinación que usa la menor cantidad de monedas. Es verdad que es más lento, pero tiene la ventaja de que siempre encuentra la solución mas optima.

En el experimento que se realizó, se trabajó con dos sistemas de monedas: uno canónico y otro no canónico. Esto se hizo para comparar cómo se comportan ambos algoritmos en distintos contextos y ver en qué casos greedy realmente da una solución óptima y cuándo no.

2. Resultados

Sistema canónico

En el sistema canónico vamos a utilizar como ejemplo el experimento dado en el documento, con el conjunto de monedas $D = \{1, 2, 5, 10, 20, 50\}$.

Dando un poco más de contexto, el algoritmo greedy siempre intenta encontrar la solución más óptima tomando decisiones locales. O sea, va paso a paso, eligiendo siempre la moneda de mayor valor posible, sin pasarse del monto que se tiene. La idea es que esas decisiones locales, hechas en cada paso, lleven a una solución globalmente óptima, usando la menor cantidad de monedas posibles.

Ojo, y esto es muy importante no siempre esas decisiones locales llevan a la mejor solución global. Porque el algoritmo greedy toma decisiones rápidas, sin pensar mucho más allá, y no se devuelve. Si eligió mal, ya sigue con eso hasta el final. Así que, aunque parece inteligente, a veces no es tan eficiente como uno espera.

Pero en este caso, con este sistema de monedas, sí funcionó muy bien. Durante las pruebas que se hicieron, se encontró que para todos los valores de M entre 1 y 30, el resultado del algoritmo greedy coincidió exactamente con lo que dio la

búsqueda exhaustiva (también conocida como fuerza bruta), que prueba todas las combinaciones posibles y se queda con la mejor.

Esto confirma que este conjunto de monedas cumple con el objetivo de un sistema canónico, ya que greedy fue capaz de encontrar siempre la mejor solución, sin necesidad de usar métodos más pesados o complejos.

M	Greedy	Fuerza bruta	Greedy	Optimo	¿Coindicen?
1	{1}	{1}	1	1	Si
2	{2}	{2}	1	1	Si
24	{20,2,2}	{2,2,20}	3	3	Si
30	{20,10}	{10,20}	2	2	Si

Sistema no canónico

Ahora, con el segundo ejemplo del experimento, que usa un sistema no canónico de monedas D = {1, 4, 6}, el algoritmo no siempre encuentra la mejor solución óptima. Aunque genera combinaciones correctas que suman el monto deseado, estas combinaciones no siempre utilizan el menor número de monedas posible. Esto pasa porque este conjunto de monedas no está bien diseñado para que el algoritmo greedy funcione adecuadamente.

Hay que tener presente que los sistemas no canónicos no garantizan que greedy encuentre la óptima, pero hay ocasiones en las que sí funciona. Eso sucede cuando no hay combinaciones eficientes; básicamente, cuando no existe mejor opción, greedy acierta, así sea no canónico. Un ejemplo de esto se observa en el caso del monto 1 y 2, como se muestra en la tabla.

M	Greedy	Fuerza bruta	Greedy	Optimo	¿Coindicen?
1	{1}	{1}	1	1	Si
2	{2}	{2}	1	1	Si
8	{6,1,1}	{4,4}	3	2	No
20	{6,6,6,1,1}	{6,6,4,4}	5	4	No

3. Conclusiones

¿Por qué greedy funciona en sistemas canónicos?

Greedy toma las decisiones paso a paso, eligiendo en cada momento la moneda de mayor denominación posible con el objetivo de llegar a la solución globalmente óptima. Como se dijo anteriormente, no siempre se llega a esa solución global óptima porque el algoritmo toma la mejor decisión que encuentra en el momento, sin la posibilidad de devolverse.

Entonces, lo que sucede es que a veces no escoge la opción más óptima para todo el problema. Esto es válido cuando el sistema de monedas es canónico. Un sistema de monedas canónico significa que las denominaciones están bien diseñadas, de forma que el algoritmo greedy siempre va a escoger el número mínimo de monedas para cualquier monto, ya que las monedas están distribuidas de manera proporcional.

En estos sistemas no existen combinaciones de monedas pequeñas que superen la eficiencia de las combinaciones con monedas grandes. Por eso, cada monto puede construirse sin necesidad de retroceder o reconsiderar decisiones anteriores para mostrar la mejor solución. Esto no quiere decir que el algoritmo pueda retroceder, pero sí es un ejemplo claro de por qué greedy funciona bien en sistemas canónicos.

Otras conclusiones

El algoritmo greedy es eficiente, sobre todo en sistemas canónicos donde las denominaciones están bien diseñadas. En sistemas no canónicos, greedy puede fallar porque no considera combinaciones más eficientes. La fuerza bruta siempre garantiza la solución óptima, pero es más lenta y costosa en tiempo de ejecución.

En eficiencia algorítmica, greedy funciona rápido, con **O(n)** siendo n la cantidad de monedas, porque toma decisiones locales rápidas paso a paso. En cambio, la fuerza bruta es más lenta, con un tiempo que crece exponencialmente, **O(k^n)**, porque prueba todas las combinaciones posibles.

Introducción ejercicio 2

El problema de selección de actividades consiste en elegir la mayor cantidad de actividades posibles en determinado tiempo, teniendo en cuenta que ninguna debe cruzarse con otra. En este ejercicio se comparan dos enfoques: el algoritmo de Selección por Fin Más Temprano (EFT), que siempre elige la actividad que termina primero, y la fuerza bruta, que explora todas las posibles combinaciones de las actividades para encontrar la solución óptima. El objetivo es verificar que EFT encuentra la solución óptima en instancias pequeñas y ver cómo es su eficiencia en instancias grandes.

1. Metodología

Para resolver este problema se desarrolló un código en C++ con tres partes importantes. La primera es la implementación de la función *seleccion_eft*, que ordena todas las actividades por su hora de fin y luego selecciona de manera iterativa las que no se crucen con la última escogida.
La segunda es la función *seleccion_fuerza_bruta*, que genera los subconjuntos posibles de actividades y revisa cuáles son compatibles.
La tercera es el generador de actividades aleatorias donde se usaron los archivos JSON para guardar y leer instancias. En las pruebas pequeñas se creó y leyó el archivo actividades_small.json. En las pruebas grandes se generaron 10.000 actividades con inicios y fines aleatorios, asegurando que siempre el fin fuera mayor al inicio, guardadas en actividades_large.json.

2. Resultados

Para verificar que el algoritmo de Selección por Fin Más Temprano (EFT) funciona correctamente, se realizaron pruebas con un conjunto pequeño de actividades, los resultados de EFT coincidieron siempre con los de Fuerza Bruta, lo que confirma que EFT obtiene soluciones óptimas en este problema.

También se realizaron pruebas con un conjunto más grande de actividades, en estas se aplicó únicamente EFT, ya que la Fuerza Bruta resulta computacionalmente inviable en estos casos, El tiempo de ejecución registrado fue de segundos, mostrando que EFT escala eficientemente incluso cuando el número de actividades es muy grande.

Tipo de prueba	Número de actividades	Método	Actividades seleccionadas	Tiempo de ejecución
Pequeña	18	EFT	5	Instantáneo (bajo)
Pequeña	18	Fuerza bruta	5	Alto (manejable)
Grande	10.000	EFT	789	0.000923 s
Grande	10.000	Fuerza bruta	No aplica	Imposible calcular

3. Conclusiones

El algoritmo EFT demostró ser óptimo en instancias pequeñas, ya que el resultado de este y el de fuerza bruta siempre coincidieron, se comprobó también que su tiempo de ejecución es eficiente para instancias grandes. La razón por la que EFT es óptimo es porque siempre escoge la actividad que termina más temprano, lo que garantiza una mayor cantidad de espacio disponible para nuevas actividades.

La fuerza bruta no es viable en instancias grandes ya que para n actividades se deben evaluar 2^n subconjuntos posibles, en el caso del ejercicio con 18 actividades se puede manejar la fuerza bruta, pero con 10.000 se convierte en una cantidad imposible de procesar en un tiempo razonable y se haría un consumo de memoria alto.

Introducción ejercicio 3

El objetivo del tercer ejercicio es construir un árbol de Huffman a partir de las frecuencias de caracteres de un archivo de texto corpus.txt y comparar la longitud media del código resultante con la codificación de longitud fija. La idea es evaluar la eficiencia de la compresión mediante códigos prefijo, que aseguran una codificación correcta y sin ambigüedades.

1. Metodología.

Se utilizó un archivo de texto llamado *corpus.txt* con un tamaño superior a 50 KB. El enlace de donde proviene el archivo es:
https://www.gutenberg.org/cache/epub/16109/pg16109.txt

El proceso consiste básicamente en calcular la frecuencia de aparición de cada símbolo (carácter) en el texto. Una vez construido el árbol de Huffman, se recorre para asignar a cada símbolo un código binario de longitud variable que garantiza que los códigos cumplen la propiedad de que ningún código es prefijo de otro.

Se generó una tabla con los campos de símbolo, frecuencia, código binario y longitud del código.

Luego, se evaluó la longitud media de codificación y se comparó con la longitud fija teórica, además de comparar el tamaño total del archivo antes y después de aplicar la codificación de Huffman.

2. Resultados

A partir del archivo de texto corpus.txt, se calcularon las frecuencias de cada símbolo y se construyó el árbol de Huffman correspondiente. El recorrido del árbol permitió generar códigos binarios únicos para cada símbolo, cumpliendo con la propiedad de que ningún código es prefijo de otro.

Además, se obtuvo una longitud media de codificación, lo cual representa una mejora significativa frente a la codificación de longitud fija. También se comparó el tamaño total del archivo antes y después de aplicar la codificación, incluyendo:

- Tamaño original (en bits)
- Tamaño codificado con Huffman
- Reducción obtenida en porcentaje

Estos resultados muestran cómo la codificación de Huffman permite una compresión más eficiente, especialmente cuando los símbolos tienen frecuencias desiguales.

Simbolo	Frecuencia	Codigo Hufmman	Longitud del codigo
Ö	6	1101011010110011	16
ÿ	1	11010110101100010110	19
%	1	11010110101100010111	19
Ø	11	110101101011100	15

3. Conclusiones

El algoritmo de Huffman es una técnica muy útil para reducir la longitud promedio en la codificación, sobre todo cuando los símbolos no aparecen con la misma frecuencia. A diferencia de la codificación fija, Huffman permite comprimir mejor los datos, lo que lo hace bastante práctico en muchas situaciones reales. Además, como se implementa usando una cola de prioridad mínima, se puede construir el árbol de forma eficiente, logrando un buen equilibrio entre velocidad y compresión. En resumen, es un algoritmo que aprovecha bien las frecuencias y logra una codificación más corta sin perder calidad.

Introducción ejercicio 4

El problema del árbol de expansión mínima (MST) consiste en encontrar un conjunto de aristas que una todos los nodos de un grafo con el menor costo posible y sin formar ciclos. Para resolverlo se pueden usar distintos algoritmos, pero en este caso se trabajó con Kruskal y Prim. La idea principal fue comprobar que ambos siempre dan el mismo costo total y además comparar qué tan rápido funcionan en grafos dispersos y en grafos densos, dependiendo del número de nodos y aristas.

1. Metodología

Para resolver el problema del Árbol de Expansión Mínima se desarrolló un programa en C++ que implementa los algoritmos de Kruskal y Prim. La idea es comparar su funcionamiento y rendimiento en distintos tipos de grafos.

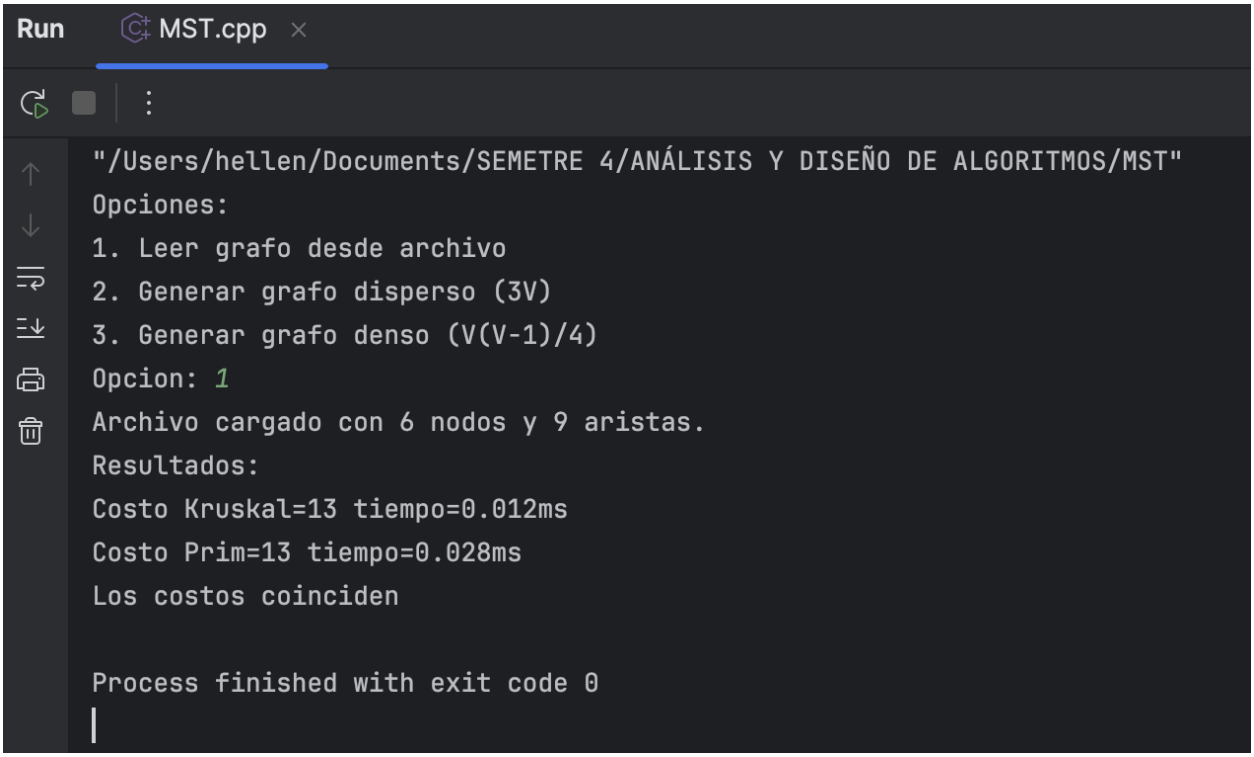
El algoritmo de Kruskal se apoyó en la estructura UnionFind, que permite manejar la unión y búsqueda de conjuntos de nodos para evitar ciclos. Lo que hace Kruskal es ordenar todas las aristas del grafo por peso y luego ir agregándolas siempre que no formen un ciclo, hasta que se conectan todos los nodos; Prim utiliza una cola de prioridad. Empieza desde un nodo y siempre selecciona la arista más barata que conecta un nodo ya incluido en el árbol con otro que todavía no lo está. De esa forma, va creciendo el árbol de expansión mínima paso a paso.

2. Resultados

En las pruebas realizadas se comprobó que tanto el algoritmo de Kruskal como el de Prim obtuvieron siempre el mismo costo total del árbol de expansión mínima, lo que confirma que ambas implementaciones son correctas.

Las diferencias se observaron únicamente en los tiempos de ejecución, Kruskal fue más eficiente en grafos dispersos, ya que el número de aristas es bajo y el ordenamiento no se vuelve un problema; Prim mostró un mejor desempeño en grafos densos grandes, donde Kruskal tarda más debido al ordenamiento de todas las aristas.

Tipo de prueba	Nodos (V)	Aristas (E)	Costo Kruskal	Costo Prim	tKruskal	tPrim
Archivo txt	6	9	13	13	0,007 ms	0,012 ms
Disperso	100	300	399	399	0,082 ms	0,400 ms
Denso	100	2.475	105	105	0,484 ms	1,106 ms
Disperso	500	1.500	2.329	2.329	0,412 ms	1,561 ms
Denso	500	62.375	499	499	20,163 ms	20,691 ms
Disperso	1.000	3.000	4.394	4.394	0,895 ms	3,555 ms
Denso	1.000	249.750	999	999	99,970 ms	94,307 ms



3. Conclusiones

A partir de las pruebas se puede concluir que ambos algoritmos son correctos, ya que siempre obtienen el mismo costo para el árbol de expansión mínima. Kruskal es más recomendable en grafos dispersos, porque el número de aristas es pequeño y su ordenamiento resulta rápido, mientras que Prim se comporta mejor en grafos densos grandes, ya que aprovecha la estructura de adyacencia y evita tener que ordenar tantas aristas. El tiempo de ejecución de ambos algoritmos crece con el tamaño del grafo, pero se mantiene dentro de rangos razonables; la elección del algoritmo depende de la densidad del grafo: Kruskal para grafos con pocas aristas y Prim para grafos muy densos.

Uso de IA:

La IA en este trabajo fue de mucha ayuda para entender mejor la teoría y los conceptos importantes de los problemas. También sirvió para aclarar dudas y tener explicaciones más claras, lo que hizo más fácil desarrollar los códigos y aprender en el proceso.

En el ejercicio 1 utilizamos inteligencia artificial como herramienta para lograr una entrega óptima. En el primer ejercicio, por ejemplo, vimos que ya estaba bien estructurado; lo único que hacía que fuera lento era la función **cambio_bruteforce**, porque usa **bruteforce_min_monedas** de forma muy repetida y además como hacer la tabla.

En el ejercicio 2 (Selección de actividades) la IA ayudó en funciones como *generar*, porque no teníamos muy claro cómo crear actividades aleatorias, y también en el manejo de archivos, ya que no sabíamos bien cómo guardar y leer datos en formato JSON para usarlos después en los algoritmos.

En el ejercicio 3 estábamos algo confundidas y necesitábamos más claridad y un mejor manejo y sobre todo investigar sobre el corpus.txt . En ese punto, la inteligencia artificial nos sirvió como apoyo para aprender y aclarar conceptos.

En el ejercicio 4 (MST) la IA fue útil en la función *generarGrafo*, porque no teníamos claro cómo generar las aristas ni cómo controlar la densidad del grafo. También sirvió de apoyo en la parte de medir el tiempo de ejecución de Kruskal y Prim, ya que no recordabamos cómo se hacía con clock() ni la conversión a milisegundos. Además, me ayudó a entender mejor la función *leerGrafo*.

En conclusión, la IA fue usada como apoyo en partes específicas que resultaban más complicadas, pero los algoritmos principales y la lógica general fueron trabajados de manera personal, utilizando la herramienta de forma ética y complementaria en el aprendizaje.