Parcial 2 – Momento 2 (Práctico)

Análisis y Diseño de Algoritmos

Daniela Álvarez Acevedo
Universidad EAFIT

Septiembre 2025

Introducción

Este informe presenta el desarrollo de los cuatro ejercicios prácticos del parcial de Análisis y Diseño de Algoritmos. Cada ejercicio se resolvió mediante implementación en C++, con pruebas de validación, análisis de complejidad y conclusiones. Los resultados experimentales se muestran en tablas para facilitar la comparación.

1. Ejercicio 1: Cambio de Monedas (Greedy vs Fuerza Bruta)

Objetivo

Comparar el algoritmo codicioso para cambio de monedas con una solución óptima por fuerza bruta.

Metodología

- Se implementó `cambio_greedy(M,D)` y `cambio_bruteforce(M,D)`.
- Se probaron montos $M \in [1, 30]$ en:
  - Sistema canónico: $D = \{1, 2, 5, 10, 20, 50\}$.
  - Sistema no canónico: $D = \{1, 4, 6\}$.

Resultados

M	Greedy (comb, #monedas)	Óptimo (comb, #monedas)	Coinciden
8	6+1+1 (3)	4+4 (2)	No
10	10 (1)	10 (1)	Sí
27	20+5+2 (3)	20+5+2 (3)	Sí

Correr Codigo

- Ingresa en una teminal y entrar al directorio: cd Cambio_monedas
- Compilar con: g++ -O2 -std=c++17 cambio_monedas.cpp -o cambio_monedas
- Ejecutar con: .\cambio_monedas
- De esta forma se generará el resultado en tu terminal y un archivo llamado resultados_ej1.csv donde se puede visualizar los resultados

- **NOTA:** Al ejecutar la primera vez se ven lso resultados del sistema no canonico D = {1, 4, 6}, para cambiar a una canonica como D= {1, 2, 5, 10, 20, 50} debes insertarlo en el codigo y volver a compilar y ejecutar. (Especificamente en la función main en el vector D linea 116)

Complejidad

- **Greedy:** El algoritmo recorre las denominaciones en orden descendente y selecciona la mayor posible en cada paso.
  - Complejidad temporal: $O(d + k)$, donde d es el número de denominaciones y k el número de monedas seleccionadas. En el peor caso $k \leq M$, con M el monto.
  - Complejidad espacial: $O(1)$.
- **Fuerza bruta:** Explora todas las combinaciones de monedas.
  - Complejidad temporal: $O(d^M)$ en el peor caso, exponencial.
  - Complejidad espacial: $O(M)$.

Conclusión

- En sistemas canónicos, greedy siempre produce la solución óptima.
- En sistemas no canónicos, greedy puede fallar, como se muestra en $M = 8$.
- Greedy es eficiente ($O(n \log n)$ o mejor) pero no siempre óptimo, mientras que la fuerza bruta es intratable para n grandes.

2. Ejercicio 2: Selección de Actividades (EFT + Verificación)

Objetivo

Validar que el algoritmo por Fin Más Temprano (EFT) siempre encuentra una solución óptima y medir su eficiencia.

Metodología

- Se implementó `seleccion_eft` (greedy) y `seleccion_bruteforce` (óptimo para $n \leq 20$).
- Se usaron 50 instancias pequeñas ($n \leq 18$) y 10 instancias grandes ($n \geq 5000$).

Resultados (pequeñas)

Instancia	n	EFT	Óptimo (BF)	Coinciden
1	14	8	8	OK
2	16	10	10	OK
3	15	8	8	OK

...

Resultados (grandes)

Instancia	n	Cardinalidad	Tiempo (ms)
1	10617	4808	1
3	9609	4379	1.001
8	5274	2493	0

Correr Codigo

- Ingresa en una teminal y entrar al directorio: cd **Eft**
- Compilar **eft** con: g++ -O2 -std=c++17 eft.cpp -o eft
- Compilar **generador_instancias.cpp** con: g++ -O2 -std=c++17 generador_instancias.cpp -o generador
- Ejecutar con .\generador ¿actividades_large.json
- Compilar **generador_small.cpp** con: g++ -O2 -std=c++17 generador_small.cpp -o generador_small
- Ejecutar con .\generador_small ¿actividades_small.json
- Ejecutar **eft** con: .\eft actividades_large.json o .\eft actividades_small.json dependiendo cuales instancias quieras verificar
- De esta forma se generará el resultado en tu terminal.
- **Nota**: Los archivos actividades_large.json y actividades_small.json ya vienen en la carpeta, pero se pueden generar como se ve en los pasos de compilación y ejecución para estos si se desea.

2.1. Complejidad

- **EFT (Fin Más Temprano):**
  - Ordenamiento de actividades: $O(n \log n)$.
  - Selección secuencial: $O(n)$.
  - Complejidad total: $O(n \log n)$.
  - Complejidad espacial: $O(1)$ adicional.
- **Fuerza bruta:**
  - Explora todos los subconjuntos: $O(2^n \cdot n)$.
  - Complejidad espacial: $O(n)$.

Conclusión

- En instancias pequeñas, EFT y fuerza bruta coinciden siempre, demostrando la optimalidad de EFT.
- En instancias grandes, EFT selecciona en milisegundos conjuntos máximos de miles de actividades.
- La selección de actividades con EFT es siempre óptima y eficiente.

3. Ejercicio 3: Codificación de Huffman

Objetivo

Comparar la codificación de Huffman con una codificación de longitud fija.

Metodología

- Se calcularon frecuencias de caracteres en un `corpus.txt` de 147 KB.
- Se construyó un árbol de Huffman y se exportó `tabla_codigos.csv` donde se puede ver la tabla con simbolo,frecuencia,codigo,longitud.

Resultados

- Alfabeto: 72 símbolos.
- Longitud fija: 7 bits/símbolo.
- Longitud media Huffman: 4.85624 bits/símbolo.
- Compresión obtenida: $\approx 31\,\%$.
- (estos se ven cuando se ejecuta el programa en cmd)

3.1. Correr Codigo

- Ingresa en una teminal y entrar al directorio: cd **huffman**
- Compilar **huffman** con: g++ -O2 -std=c++17 huffman.cpp -o huffman
- Ejecutar: ./huffman corpus.txt
- De esta forma se generará el resultado en tu terminal.
- **Nota**: El archivo generado tabla_codigos.csv muestra simbolo, frecuencia, codigo, longitud y en la terminal verás a detalle los resultados especificos.

3.2. Complejidad

- Construcción del árbol con cola de prioridad:
  - Inserción de n símbolos y $n-1$ extracciones.
  - Complejidad temporal: $O(n \log n)$.
  - Complejidad espacial: $O(n)$.
- Comparación:
  - Codificación fija: $\lceil \log_2 n \rceil$ bits por símbolo.
  - Codificación de Huffman: longitud media $\approx H(p)$, donde $H(p)$ es la entropía de la fuente.

Conclusión

- La codificación de Huffman cumple la propiedad de prefijo.
- Reduce el tamaño en bits respecto a una codificación fija.
- Huffman es $O(n \log n)$ y garantiza un código óptimo de prefijo.

4. Ejercicio 4: MST con Kruskal y Prim

Objetivo

Validar que Kruskal y Prim producen el mismo MST y comparar su eficiencia.

Metodología

- Se implementó Kruskal con DSU y Prim con cola de prioridad.
- Se generaron grafos dispersos ($E \approx 3V$) y densos ($E \approx V(V-1)/4$) con $V = \{100, 500, 1000\}$.

Resultados

V	E	CostoKruskal	CostoPrim	tKruskal(ms)	tPrim(ms)	Tipo
100	300	16101	16101	0	0	Disperso
100	2475	2072	2072	0	0	Denso
500	1500	92219	92219	0	2.051	Disperso
...	...	...	...	...	...	...

Correr Codigo

- Ingresa en una teminal y entrar al directorio: cd **mst**
- Compilar **mst** con: g++ -O2 -std=c++17 mst.cpp -o mst
- Ejecutar: ./mst
- De esta forma se generará el resultado en tu terminal.
- **Nota**: El archivo generado resultados_mst.csv muestra V, E, costoKruskal, costoPrim, tKruskal(ms), tPrim(ms), tipo y en la terminal verás a detalle los resultados especificos.

4.1. Complejidad

- **Kruskal:**
  - Ordenamiento de aristas: $O(E \log E)$.
  - Operaciones de Union-Find: $O(E\alpha(V))$, con α la función inversa de Ackermann.
  - Complejidad total: $O(E \log E)$.
  - Espacio: $O(V + E)$.
- **Prim (con heap binario):**
  - Inserciones y extracciones en heap: $O(E \log V)$.
  - Complejidad total: $O(E \log V)$.
  - Con heap de Fibonacci: $O(E + V \log V)$.
  - Espacio: $O(V + E)$.

Conclusión

- Ambos algoritmos generan siempre el mismo costo de MST.
- Kruskal es más rápido en grafos dispersos.
- Prim resulta más eficiente en grafos densos.

Conclusiones Generales

- Los algoritmos codiciosos son óptimos en ciertos contextos (EFT, monedas canónicas), pero pueden fallar en otros.
- La validación por fuerza bruta en instancias pequeñas permitió comprobar la correctitud.
- Huffman ejemplifica cómo la teoría algorítmica se aplica a compresión real.
- La estructura del input (disperso/denso) define cuál algoritmo de MST es más eficiente.
- **NOTA:** Se tomo como apoyo la inteligencia artificial Chatgpt y Claude para la comprensión de los ejercicios y guia cuando hubo erroes. Además como ayuda para el análisis de la complejidad de los ejercicios propuestos.

5. References

1. **CLRS09** T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009. Disponible en:`https://www.cs.mcgill.ca/~akroit/math/compsci/Cormen%20Introduction%20to%20Algorithms.pdf?`
2. **PivkinaHistorical** I. Pivkina, "Discovery of Huffman codes," material histórico, Universidad de Nuevo México.`https://www.cs.nmsu.edu/historical-projects/Projects/18920140825Huffm pdf?`
3. Chatgpt `https://chatgpt.com/`