Jueves, 4 de diciembre de 2025.

Universidad Autonóma de Aguascalientes (UAA)

ING. EN COMPUTACIÓN INTELIGENTE (ICI)

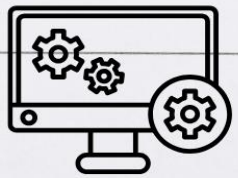
SEMESTRE III

CIENCIAS DE LA COMPUTACIÓN

ESTRUCTURAS COMPUTACIONALES AVANZADAS

Proyecto final

- Alumno: Badajoz Santacruz José Julián.
- Docente: Miguel Ángel Meza de Luna.



Índice:

Integrantes y roles:

Nombre	Rol
➢ José Julián Badajoz Santacruz.	a) **Propietario del proyecto** (definición del alcance y prioridades del proyecto). b) **Scrum Master** (organización del flujo de trabajo y eliminación de bloqueos). c) **Desarrollador** (implementación, pruebas y documentación de los distintos algoritmos)

Metodologías usadas:

La metodología usada para este proyecto es: ***Scrum***, la cual se adapta a un solo integrante, que mantiene todo desempeño fundamental. Para ellos, se hizo el uso de un tablero donde se documentaron las tareas del sprint, fechas y estados.

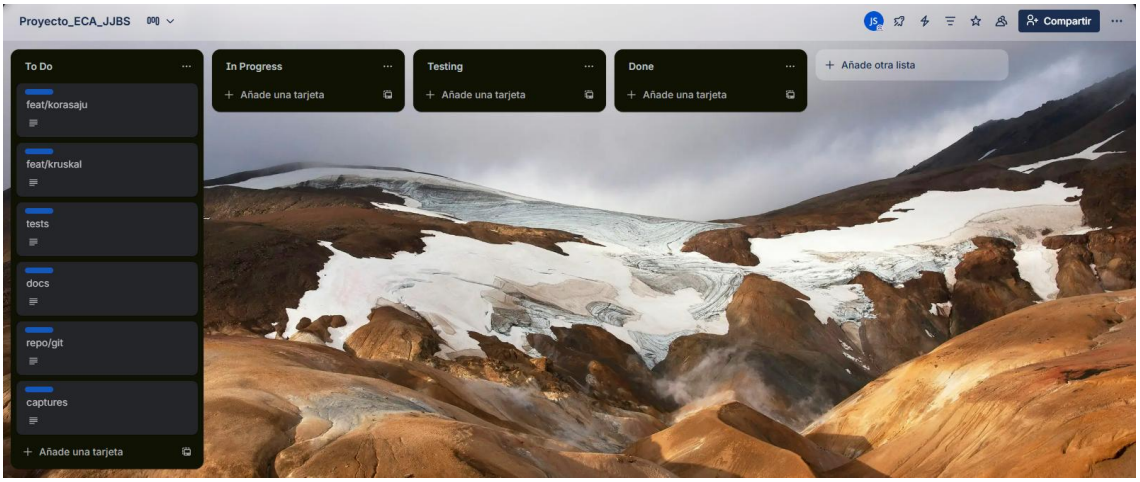
Marco	Herramienta	Control de repositorio
Scrum	Trello	Git y GitHub

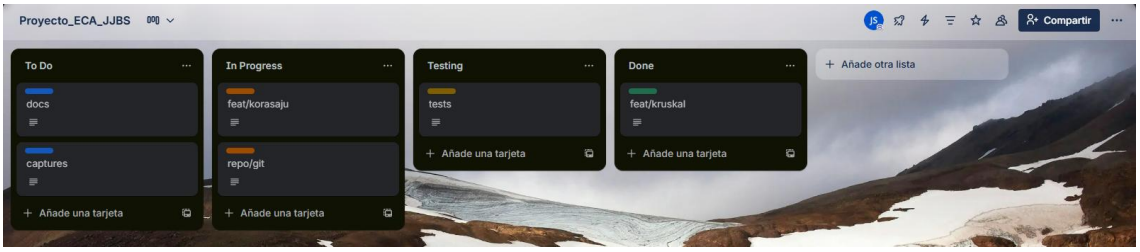
Capturas del tablero:

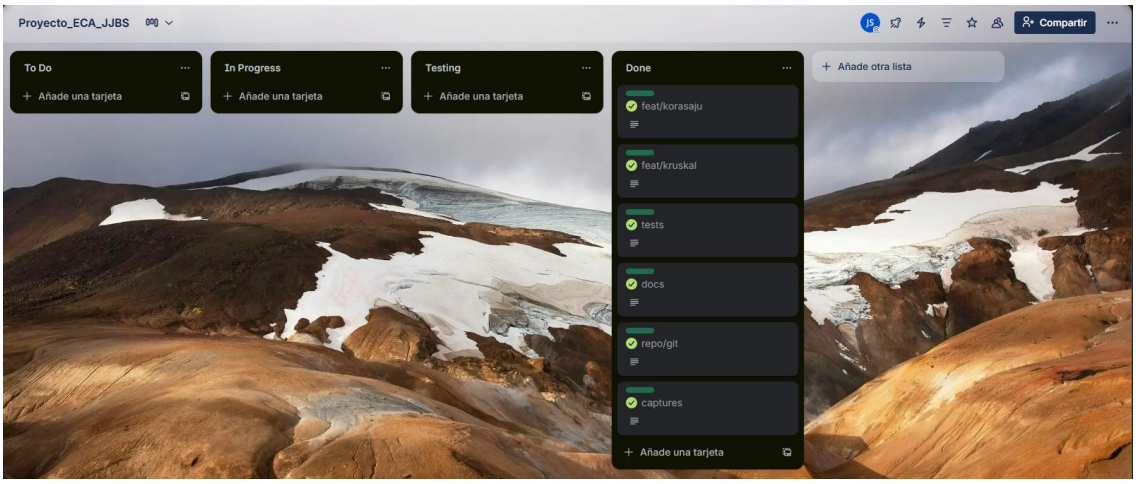
A continuación, se muestran las capturas del tablero de la metodología Scrum realizadas estos últimos días:

Enlace:

https://trello.com/invite/b/6930f2f7c162ffc19ee5f790/ATTIb88ad77acd9f03fca9240053990aae69D9421A36/proyectoecajjbs





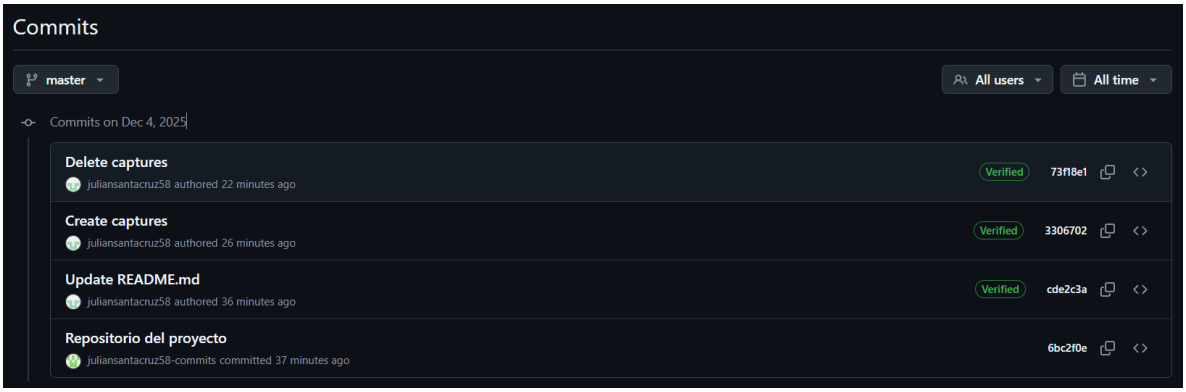


Capturas/Links del repositorio:

https://github.com/juliansantacruz58/Proyecto_ECA.git

https://github.com/juliansantacruz58/Proyecto_ECA/tree/master/src

https://github.com/juliansantacruz58/Proyecto_ECA/tree/master/tests



README

Proyecto de Algoritmos con Grafos

Universidad Autónoma de Aguascalientes Alumno: José Julián Badajoz Santacruz Materia: Estructuras Computacionales Avanzadas

Repositorio listo para entrega: Kosaraju (SCC) y Kruskal (MST) en C++.

Estructura:

- src/ -> código fuente (.cpp)
- tests/ -> archivos de entrada de ejemplo (.txt)
- captures/ -> evidencias (pon aquí capturas)
- docs/ -> documento final (PDF/placeholder)

Algoritmos implementados

En el laboratorio se escogió el desarrollo de dos algoritmos fundamentales en teoría de grafos, los cuales tienen el objetivo de cubrir dos áreas distintas, como lo son: componentes en grafos dirigidos y construcción de árboles con expansión mínima en los grafos no dirigidos. Los algoritmos implementados son:

Algoritmo Kosaraju:

- **Descripción:**

  El algoritmo de **Kosaraju** sirve para encontrar todas las **Componentes Fuertemente Conexas (SCC)** de un grafo dirigido. Una SCC es simplemente un grupo de nodos en el que cada uno puede alcanzar a cualquier otro siguiendo las direcciones del grafo.

  La lógica detrás del algoritmo es bastante sencilla:

  1. **Realizar un DFS normal** y anotar el orden en que cada nodo termina su recorrido.
  2. **Invertir el grafo**, es decir, cambiar el sentido de todas las aristas.
  3. **Hacer un nuevo DFS en el grafo invertido**, siguiendo el orden obtenido en el paso 1. Ahora, cada recorrido completo del DFS corresponde exactamente a una SCC.

- **Objetivo:**
  Identificar las Componentes Fuertes Conexas (SCC) dentro de un grafo dirigido. Cada SCC representa un grupo de nodos en el que todos se pueden alcanzar entre sí, al seguir las direcciones del grafo; permitiendo la detección de ciclos, subestructuras que son interdependientes y de conectividad completa.

- **Recorrido:**

Se utilizan dos distintos recorridos DFS, los cuales se diferencian de:

1. El primero para determinar el orden de salida de cada nodo, para posteriormente almacenarlo en la pila.
2. El segundo para aplicarlo a un grafo transpuesto, que sigue el orden de la pila.

- **Tipo de grafo:**
  1. Grafo dirigido.
  2. No necesita de ponderación.
  3. Puede incluir ciclos, tanto externos como internos.

- **Representación:**

  1. Head[] representa el primer enlace de cada lista distinta.
  2. To[] representa cada nodo de destino.
  3. nextEdge[] representa al siguiente borde.

  Cabe recalcar también que cada lista de adyacencia del grafo transpuesto se genera por default.

- **Uso:**
  1. Análisis de dependencias en software.
  2. Identificación de ciclos de referencias.
  3. Clasificación de clusters en grafos dirigidos.
  4. Optimización de ordenamiento topológico.
  5. Motores de búsqueda.

- **Pseudocódigo:**

1. Inicializar todos los nodos como NO_VISITADOS.
2. Crear una PILA vacía para guardar el orden de finalización.

3. Para cada nodo u en G:
   Si u NO_VISITADO:
   DFS_LlenarPila(u)

4. Construir el grafo transpuesto GT invirtiendo todas las aristas.

5. Volver a marcar todos los nodos como NO_VISITADOS.

6. Mientras la PILA no esté vacía:
    u = PILA.pop()
    Si u NO_VISITADO:
        Crear lista COMPONENTE
        DFS_EncontrarSCC(GT, u, COMPONENTE)
        Imprimir COMPONENTE

```
SUBRUTINA DFS_LlenarPila(u):
    Marcar u como VISITADO
    Para cada vecino v de u:
        Si v NO_VISITADO:
            DFS_LlenarPila(v)
    Push(u) en la PILA

SUBRUTINA DFS_EncontrarSCC(GT, u, COMPONENTE):
    Marcar u como VISITADO
    Agregar u a la lista COMPONENTE
    Para cada vecino v de u en GT:
        Si v NO_VISITADO:
            DFS_EncontrarSCC(GT, v, COMPONENTE)
```

Algoritmo Kruskal:

- **Descripción:**

El algoritmo de Kruskal se utiliza para encontrar el Árbol de Expansión Mínima (MST) de un grafo no dirigido y con pesos en sus aristas. El MST es un conjunto de aristas que conecta todos los nodos con el menor costo total posible y sin crear ciclos.

El proceso puede explicarse de forma sencilla:

1. Reunir todas las aristas del grafo.
2. Ordenarlas por peso, de la más barata a la más cara.
3. Ir tomando aristas en ese orden, siempre que no formen un ciclo (esto se verifica de forma eficiente con la estructura *Union-Find*).
4. Repetir hasta tener exactamente n − 1 aristas, donde *n* es el número de nodos.

- **Objetivo:**
  Construir un árbol de expansión mínima (MST), el cual va a garantizar la conexión de todos los nodos del grafo con el menor costo posible sin la necesidad de crear algún otro ciclo.

- **Recorrido:**
  No hace ni de DFS ni mucho menos de BFS, se realizan 3 procesos:
  1. Ordenamiento de toda arista de acuerdo a su peso.
  2. Recorrido lineal en orden.
  3. Union-Find para la verificación de agregar una arista o crear ciclo.

- **Tipo de grafo:**
  1. Grafo no dirigido.
  2. Ponderados.
  3. Densos y dispersos.
  4. Conexos.

- **Representación:**
  Se utilizan diversas listas que guardan cada arista, mientras que por otra parte, la estructura Union-Find se compone de dos estructuras:
  1. Padre[].
  2. Rango[].

- **Uso:**
  1. Construcción en redes eléctricas.
  2. Infraestructura de carreteras o cableado.
  3. Topologías eficientes en telecomunicaciones.
  4. Diseño de rutas de mínima distancia y/o costos.
  5. Optimización de recursos conectados.

- **Pseudocódigo:**

1. Inicializar el conjunto MST vacío.
2. Leer todas las aristas del grafo.
3. Ordenar las aristas por peso ascendente.

4. Inicializar la estructura UNION-FIND:
   Para cada nodo v:
      padre[v] = v
      tamaño[v] = 1

5. Para cada arista (u, v, peso) en el orden ya ordenado:
   Si FIND(u) != FIND(v): // No forman ciclo
      UNION(u, v)
      Agregar arista (u, v, peso) al MST

6. Terminar cuando el MST tenga (n − 1) aristas.
7. Devolver o imprimir el MST.

```
SUBRUTINA FIND(x):
    Mientras x != padre[x]:
        padre[x] = padre[padre[x]]   // compresión de caminos
        x = padre[x]
    Retornar x

SUBRUTINA UNION(x, y):
    rx = FIND(x)
    ry = FIND(y)
    Si rx == ry: retornar

    // unir por tamaño (o por rango)
    Si tamaño[rx] < tamaño[ry]:
        padre[rx] = ry
        tamaño[ry] += tamaño[rx]
    En otro caso:
        padre[ry] = rx
        tamaño[rx] += tamaño[ry]
```

Explicación Big-O por algoritmo

Algoritmo Kosaraju

- **Complejidad en tiempo:** $O(V + E)$
- **Complejidad en espacio:** $O(V + E)$

El algoritmo Kosaraju realiza *dos recorridos DFS* y la construcción del grafo transpuesto. Cada DFS recorre todos los vértices y aristas una sola vez, por lo que su complejidad es lineal respecto al tamaño del grafo. Según GeeksforGeeks (2024), "Kosaraju opera en tiempo $O(V + E)$ debido a que cada arista se procesa exactamente dos veces". El uso de la pila, el vector de visitados y la representación del grafo justifican un espacio total también lineal.

Algoritmo Kruskal

- **Complejidad en tiempo:** $O(E \log E)$
- **Complejidad en espacio:** $O(V)$

Kruskal comienza ordenando todas las aristas del grafo por peso, operación que domina la complejidad con $O(E \log E)$. Luego aplica la estructura Union-Find para evitar ciclos en tiempo casi constante por operación ($\alpha(V)$). GeeksforGeeks (2023) explica que "el costo principal de Kruskal proviene del ordenamiento de aristas". El espacio requerido corresponde básicamente al arreglo de padres y rangos para Union-Find.

Análisis y discusión

En esta investigación se exploraron dos algoritmos clásicos de la teoría de grafos —Kosaraju y Kruskal— con la intención de entender no solo cómo funcionan internamente, sino también qué tipo de problemas ayudan a resolver. Aunque cada uno aborda ámbitos distintos, comparten un mismo espíritu: fueron diseñados para trabajar de manera eficiente sobre estructuras de grafo y aprovechar al máximo sus propiedades esenciales.

En el caso de **Kosaraju**, su diseño se basa en el uso de recorridos profundos (DFS) y en la relación entre un grafo y su versión transpuesta. Esta estrategia permite descubrir subgrupos fuertemente conectados dentro de grafos dirigidos, revelando ciclos, dependencias y regiones de alta interacción entre nodos. La investigación permitió observar que, aunque su implementación requiere dos recorridos completos y la construcción del grafo invertido, su comportamiento sigue siendo lineal respecto al tamaño del grafo, lo cual lo vuelve muy eficiente incluso en grafos grandes.

Por otro lado, **Kruskal** está orientado a resolver un problema totalmente distinto: construir un Árbol de Expansión Mínima. Su diseño es un claro ejemplo del enfoque voraz, en el que las decisiones locales —elegir siempre la arista más barata que no forme un ciclo— construyen una solución óptima global. Durante el análisis se observó que el uso de la estructura Union-Find es esencial para mantener la eficiencia del algoritmo, ya que permite verificar la formación de ciclos prácticamente en tiempo constante. Esto, sumado al ordenamiento inicial de aristas, hace que Kruskal sea especialmente útil en grafos dispersos y en aplicaciones reales como diseño de redes o rutas de coste mínimo.

De manera general, la investigación muestra que ambos algoritmos resuelven problemas distintos pero complementarios dentro de la teoría de grafos: Kosaraju revela la estructura interna del grafo, mientras que Kruskal construye una estructura óptima que lo conecta. En conjunto, proporcionan una visión más completa de cómo los grafos pueden analizarse y optimizarse. Asimismo, se observa que ambos destacan por su eficiencia teórica y por su claridad conceptual, lo que los convierte en herramientas fundamentales tanto en el estudio académico como en aplicaciones prácticas.

Conclusión

Estudiar los algoritmos Kosaraju's algorithm y Kruskal's algorithm me hizo valorar lo elegante que puede ser la solución a problemas complejos con ideas simples: uno revela la estructura interna de un grafo dirigido, otro construye conexiones óptimas en grafos no dirigidos. Ambos demuestran que con los principios adecuados (recorridos inteligentes, ordenamientos, estructuras eficaces) se puede resolver de forma eficiente tareas que a primera vista se ven complicadas. Esa claridad conceptual y eficiencia los convierte en herramientas fundamentales —no sólo teóricas, sino útiles en aplicaciones reales.

Referencias bibliográficas

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). MIT Press. Información y capítulos. Recuperado de: https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/
2. Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2008). *Algorithms*. McGraw-Hill. Recuperado de: https://cseweb.ucsd.edu/~dasgupta/book/index.html
3. GeeksforGeeks. (s.f.). *Kruskal's Minimum Spanning Tree (MST) Algorithm*. Recuperado de: https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-mst-algorithm/
4. Kramirez. (s.f.). *Teoría de Grafos: Material docente* [Apuntes]. Recuperado de: https://kramirez.github.io
5. Kuisch, H. (2020). *Greedy Algorithms and Minimum Spanning Trees: Prim & Kruskal*. Delft University of Technology. Recuperado de: https://ocw.tudelft.nl/courses/greedy-algorithms/
6. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley. Recuperado de: https://algs4.cs.princeton.edu/home/