**Informe Trabajo Practico:**

**Suite de Algoritmos de Grafos en C++**

1. **¿Cuándo Conviene utilizar cada algoritmo incorporado en el programa?**
   1. **BFS:**

BFS garantiza encontrar la ruta más corta en grafos no ponderados (es decir, donde todas las aristas tienen el mismo “peso” o costo). Este explora el grafo nivel por nivel, lo que lo hace ideal para encontrar todos los nodos a una distancia especifica desde un nodo origen o determinar si un nodo es alcanzable desde otro en un número limitado de pasos.

Sin embargo, si el grafo tiene pesos distintos en las aristas, BFS no garantiza la ruta de menor costo.

* 1. **Dijkstra:**

Dijkstra está diseñado para grafos ponderados con pesos no negativos, tal que este encuentra la ruta de menor costo total desde un nodo origen a todos los demás nodos. Por lo que, a diferencia de BFS, que minimiza el número de aristas, Dijkstra la mínima suma de los pesos, lo que lo hace útil no solo para encontrar una ruta optima entre dos nodos, sino que puede calcular el camino más barato a todos los nodos del grafo.

Sin embargo, si el grafo tiene pesos negativos, Dijkstra puede fallar. En ese caso, se recomienda usar Bellman-Ford.

* 1. **Bellman-Ford:**

Bellman-Ford permite el ingreso de aristas con pesos positivos y negativos, por lo que es útil para calcular las distancias mínimas desde un nodo fuente, pero sin restricciones sobre los pesos y lo hace crucial para identificar si hay un ciclo de peso negativo en el grafo.

Sin embargo, si el grafo no tiene pesos negativos, Dijkstra es más rápido y eficiente.

* 1. **Floyd-Warshall:**

Floyd-Warshall permite calcular la distancia mínima entre todos los pares de nodos, por lo que funciona bien en grafos dirigidos ponderados con muchas aristas y un numero de nodos no demasiado grande. Este lo hace comparando todos los posibles caminos a través del grafo entre cada par de vértices.

Sin embargo, si solo necesitas rutas desde un nodo origen, Dijkstra o Bellman-Ford son más eficientes.

* 1. **A\*:**

A\* está diseñado para encontrar el camino optimo entre un nodo origen y un nodo destino, sin explorar innecesariamente todo el grafo. Para usarlo, el grafo debe tener una estructura que permita definir una heurística útil y las aristas deben tener pesos no negativos.

Sin embargo, si el grafo no tiene una buena heurística, el algoritmo se comporta como Dijkstra, sin ventajas reales.

* 1. **Kruskal/Prim:**

Kruskal/Prim se utilizan para construir un árbol de expansión minima(MST) buscando conectar todos los nodos del grafo con el menor costo total posible, sin formar ciclos. Para ello, se requiere que el grafo tenga aristas con pesos, y este debe ser no dirigido (ya que se busca una conexión bidireccional entre nodos.

1. **Complejidades y decisiones de representación**

En el estudio de grafos, las estructuras de datos como listas o matrices de adyacencia cumplen un papel fundamental para representar las conexiones entre vértices y permitir que los algoritmos de recorrido operen de forma eficiente. Estas representaciones sirven como la base sobre la que se construyen procesos como la exploración de nodos, el seguimiento de rutas o el análisis de relaciones dentro de una red.

* 1. **BFS:**

Cuando implementamos BFS, la mejor elección fue utilizar una lista de adyacencia como estructura principal, sobre todo pensando en la eficiencia.

BFS necesita explorar todos los vértices adyacentes al nodo actual. Con una lista de adyacencia, esa búsqueda es directa: simplemente recorres la lista asociada a ese nodo, sin iterar sobre nodos que no están conectados.

Además, el BFS tiene una complejidad de O (V + E) usando listas de adyacencia (V = vértices, E = aristas). Con matrices de adyacencia, la exploración sube a O(V2), lo que es mucho menos eficiente para grafos grandes y poco densos.

* 1. **Dijkstra:**

En el algoritmo de Dijkstra, la prioridad es recorrer de forma eficiente los vecinos de cada vértice para actualizar las distancias mínimas. Usar una lista de adyacencia como estructura principal facilita esta tarea, ya que almacena para cada nodo únicamente las aristas que realmente existen, junto con el peso asociado a cada una. Es permite que, al procesar un vértice, el algoritmo acceda directamente a sus conexiones relevantes sin tener que revisar todos los posibles nodos del grafo, optimizando el tiempo de ejecución.

* 1. **Bellman-Ford:**

En el caso de Bellman-Ford, está diseñado precisamente para partir de un único vértice origen y calcular la distancia mínima hacia todos los demás vértices de un grafo dirigido con pesos (incluso si esos pesos son negativos). Por eso, una lista de adyacencia resulta especialmente conveniente. En esta se puede iterar únicamente sobre las aristas que realmente existen, evitando inspeccionar pares de vértices sin conexión como ocurría en una matriz de adyacencia. Además, si el grafo es poco denso (pocas aristas en comparación con el número de vértices), la lista almacena solo las conexiones necesarias, ahorrando memoria frente a la matriz que siempre es de tamaño VxV.

* 1. **Floyd-Warshall:**

En Floyd-Warshall, el objetivo es calcular las distancias mínimas entre todos los pares de vértices del grafo, lo que implica trabajar constantemente con la información de cada posible conexión. Usar una matriz de adyacencia como estructura principal resulta ser la opción óptima, porque permite acceder y actualizar de forma directa el peso entre cualquier par de nodos en tiempo constante, algo fundamental dado que el algoritmo recorre y modifica repetidamente estos valores durante su ejecución. Al tener toda la información centralizada en una matriz, las operaciones de comparación y actualización se vuelven más simples y eficientes, especialmente en grafos densos donde existen muchas aristas.

* 1. **A\*:**

Cuando implementamos A\*, la representación más adecuada suele ser una grilla basada en una matriz. Esto se debe a que A\* no solo explora nodos adyacentes, sino que también necesita calcular constantemente las distancias (g-cost) y estimaciones heurísticas (h-cost) hacia el objetivo. En una grilla, los movimientos posibles (arriba, abajo, izquierda, derecha, o incluso diagonales) están claramente definidos y se pueden obtener en tiempo constante O (1) accediendo a las posiciones de la matriz. Así, el control de visitados, los cálculos de vecinos y las actualizaciones de costes se simplifican mucho.

* 1. **Kruskal/Prim:**

Cuando implementamos algoritmos de árbol recubridor mínimo como Kruskal o Prim, la mejor elección suele ser representar el grafo con una lista de aristas o una lista de adyacencia, dependiendo del algoritmo. Kruskal trabaja directamente con las aristas, por lo que resulta natural almacenarlas en una lista y ordenarlas por peso; su complejidad es O (E log E) debido a la ordenación y al uso de estructuras de unión-búsqueda para detectar ciclos. En cambio, Prim se beneficia más de una lista de adyacencia junto a una cola de prioridad, ya que en cada paso necesita elegir la arista de menor peso que conecta el árbol parcial con un nuevo vértice; implementado de esta forma, su complejidad es O (E log V). Usar matrices de adyacencia para cualquiera de los dos algoritmos resulta menos eficiente, ya que obliga a recorrer todos los vértices incluso cuando no están conectados, lo que eleva la complejidad hasta O(V²) y lo hace poco práctico para grafos grandes y dispersos.

1. **Limitaciones Conocidas:**
   1. **BFS:**

El algoritmo BFS es óptimo para encontrar el camino más corto en términos de número de aristas en grafos no ponderados, pero presenta limitaciones importantes en otros contextos. No considera los pesos de las aristas, por lo que en grafos ponderados puede encontrar rutas más largas en costo, aunque tengan menos saltos. Ejemplo: si para llegar desde A hasta C esta la opción de A-C que cuesta 5 y la opción A-B-C que cuesta 3, BFS elegirá A-C, aunque sea más caro.

Además, a la hora de ingresar grafos dirigidos, si se aplica BFS sin respetar la dirección de las aristas, se obtienen rutas inválidas.

* 1. **Dijkstra:**

El algoritmo de Dijkstra es muy eficiente para encontrar el camino más corto en grafos ponderados con pesos no negativos, pero presenta varias limitaciones importantes según la estructura del grafo. Este no funciona correctamente en grafos con aristas de peso negativo, ya que su lógica de expansión se basa en asumir que una vez que se ha encontrado el camino más corto a un nodo, no puede mejorarse. Esto falla si existen ciclos negativos o caminos más baratos que se descubren después. Además, en grafos muy densos o con muchos nodos, su rendimiento puede verse afectado si no se implementa con estructuras eficientes como colas de prioridad (por ejemplo, un heap binario o Fibonacci heap).

* 1. **Bellman-Ford:**

El algoritmo Bellman-Ford, aunque robusto frente a pesos negativos y capaz de detectar ciclos negativos, presenta varias limitaciones importantes. Su complejidad temporal de O(V⋅E) lo vuelve poco eficiente en grafos grandes o densos, especialmente comparado con alternativas como Dijkstra o A\*. Además, si el destino está involucrado en un ciclo negativo, Bellman-Ford no puede proporcionar una distancia válida, ya que considera que el costo puede disminuir indefinidamente.

* 1. **Floyd-Warshall:**

El algoritmo Floyd-Warshall permite calcular caminos mínimos entre todos los pares de nodos en un grafo, pero presenta limitaciones importantes según el contexto. Su complejidad temporal de O(V3) lo vuelve ineficiente en grafos grandes, y su uso intensivo de memoria —al mantener una matriz de distancias de tamaño V×V puede ser prohibitivo en entornos con recursos limitados. Aunque detecta ciclos negativos, no puede calcular distancias válidas si estos afectan las rutas entre pares, lo que invalida sus resultados.

* 1. **A\*:**

El algoritmo A\* es óptimo y completo siempre que se utilice una heurística admisible (que nunca sobreestime el costo real), pero también presenta limitaciones importantes. Su principal desventaja es el alto consumo de memoria, ya que mantiene en la cola de prioridad una gran cantidad de nodos abiertos, lo que puede volverlo impráctico en grafos muy grandes o con espacios de búsqueda extensos. Además, la calidad de la heurística es determinante: si la heurística es poco informativa (por ejemplo, siempre vale 0), A\* se degrada al comportamiento de Dijkstra, explorando prácticamente todos los nodos alcanzables. En el caso contrario, si la heurística sobreestima el costo, el algoritmo deja de ser óptimo y puede entregar rutas incorrectas.

* 1. **Kruskal/Prim:**

Los algoritmos de Kruskal y Prim son óptimos para construir un árbol recubridor mínimo en grafos no dirigidos y ponderados, pero también presentan limitaciones. En primer lugar, no son aplicables directamente a **grafos dirigidos**, ya que el concepto de MST se define únicamente en grafos no dirigidos. Además, si el grafo no es conexo, no pueden generar un único árbol recubridor mínimo, sino que producen un **bosque recubridor mínimo** en cada componente conexa, lo cual puede no ser deseado según el problema. Otra limitación es que estos algoritmos **no garantizan caminos mínimos entre pares de vértices**, ya que el MST minimiza el costo total de conexión y no las distancias individuales. Finalmente, en grafos muy densos, el uso de estructuras como listas de aristas ordenadas (Kruskal) o colas de prioridad (Prim) puede implicar un alto consumo de memoria y tiempo, lo que limita su eficiencia práctica en comparación con algoritmos especializados para grafos muy grandes.