Algoritmo de dijkstra y grafos bipartidos

Xavier A. Benavides, Gabriel A. Calvache, Marjorie J. Cedeño.

Departamento de Ciencias de la computación, Universidad de las Fuerzas Armadas

28436: Estructura de datos

Ing. Washington Loza

10 de diciembre de 2025

Índice

Índice de Figuras

Introducción

Tanto el estudio de los algoritmos de Dijkstra y como de los grafos bipartitos son puntos valiosos dentro del análisis de estructuras de datos y la teoría de grafos, ya que permiten comprender, modelar y resolver una gran cantidad de problemas reales relacionados con rutas óptimas, redes, particiones y relaciones entre conjuntos. El algoritmo de Dijkstra es considerado importante por ofrecer un método eficiente para determinar el camino más corto desde un nodo origen hacia todos los otros nodos dentro de un grafo ponderado con pesos no negativos, su importancia radica en la combinación de simplicidad conceptual y eficiencia computacional mediante el uso de estructuras como colas. Por otra lado, los grafos bipartitos permiten representar de forma clara escenarios donde los elementos se dividen en dos conjuntos independientes, sin conexiones internas entre ellos, lo que facilita la detección de relaciones de dependencia, problemas de asignación, emparejamientos máximos y validación de estructuras libres de ciclos impares. El análisis de estos conceptos permite comprender la naturaleza de las conexiones, optimizar desplazamientos o flujos dentro de un sistema y aplicar estrategias algorítmicas de resolución basadas tanto en las propiedades matemáticas como en las representaciones computacionales. El siguiente informe presenta los fundamentos teóricos de Dijkstra y de los grafos bipartitos, explica sus características principales, su utilidad, su funcionamiento general y la forma en que ambos se complementan dentro del estudio de grafos complejos.

Definiciones fundamentales

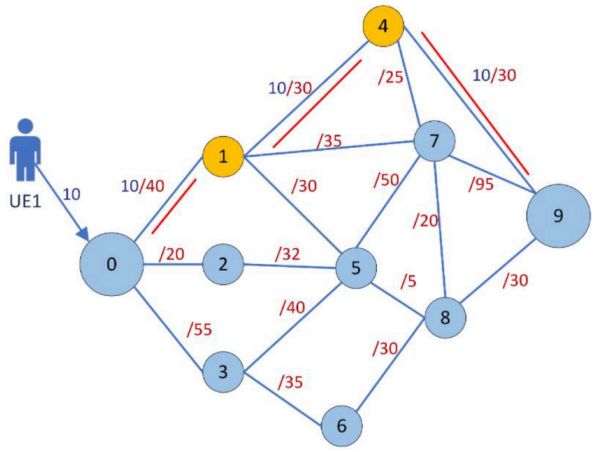
Algoritmo Dijkstra

Según (Sánchez Torrubia & Lozano Terrazas, 2002, #) “El algoritmo de Dijkstra es un algoritmo eficiente (de complejidad O(n^2) donde n es el número de vértices) que sirve para encontrar el camino de coste mínimo desde un nodo origen a todos los demás nodos del

grafo” hay que mencionar que esto se realiza mediante los pesos que tienen las aristas, tomar en cuenta que el algoritmo solo se puede utilizar en grafos ponderados siendo estos dirigidos o no dirigidos, pero que no posean pesos negativos ya que esto puede afectar en los resultados obtenidos del algoritmo.

Figura 1

Algoritmo Dijkstra.



Nota. Tomado de AnAdaptive Data Traffic Control Scheme with Load Balancing in a Wireless Network, por Z. Zhang y A. Duan, 2022, https://doi.org/10.3390/sym14102164.

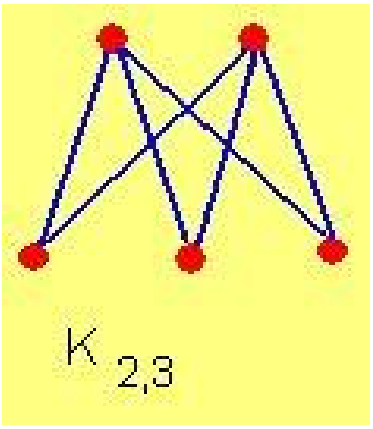
Los escenarios en donde se ven más el uso de este algoritmo son: Google Maps (distancia o tiempo mínimo), Redes de computadoras (elegir la ruta con menor latencia), logística y transporte (rutas más económicas entre centros de distribución), videojuegos (recorrido más corto para los personajes), IA y robótica (planificación de trayectorias), entre otros más.

Grafos bipartidos

Un grafo bipartido es un grafo G = (V, A) cuyos vértices pueden dividirse en dos subconjuntos, donde no tienen ningún elemento en común, es decir son disjuntos. (Labory Pesquer, 2018, #) menciona “Si además cada uno de los vértices de V_1 está unidos a todos los de V_2 se dice que es un grafo bipartido completo y si $|V_1|$ = r y $|V_1|$ = s lo denotamos $K_{r,s}$.”

Figura 2

Grafo bipartido completo.



Nota. Tomado de Teoría de Grafos: Grafos Bipartidos, por Claudio Cifuentes M, 2007, https://teoriadegrafos.blogspot.com/2007/03/grafos-bipartidos.html.

El uso de grafos bipartidos son usados para relacionar, emparejar, asignar elementos sin mezclar grupos, por ejemplo: en asignación de recursos (estudiantes con proyectos), sistemas de recomendación (clientes con productos), redes y comunicación (antenas con dispositivos), entre varios más.

Representación gráfica

Para representar el algoritmo Dijkstra y los grafos bipartidos, se usará ejercicios de cada uno y lo representaremos mediante software.

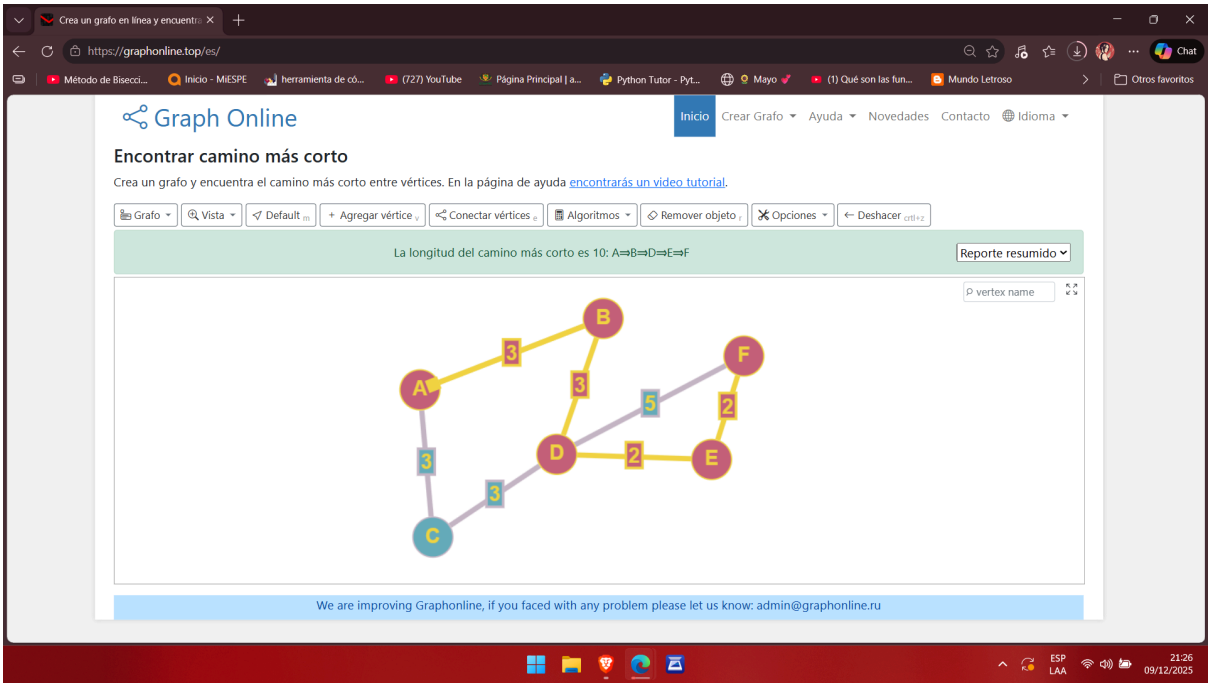
Ejercicio de Algoritmo Dijkstra

En una provincia hay 6 ciudades conectadas por redes de carreteras, se necesita encontrar la ruta más corta de la ciudad A hasta la ciudad F, tomando en cuenta las distancias que hay entre las ciudades.

Red de ciudades: A→B 3 km, A→C 1 km, B→D 4 km, C→D 4 km, D→E 2 km, D→F 5 km, E→F 2 km.

Figura 3

Representación de Algoritmo Dijkstra.



Nota. Elaboración propia utilizando Graph Online, https://graphonline.top/es/.

Solución: Como se puede ver en la imagen el camino más corto para resolver el ejercicio planteado es A→B→D→E→F que da un total de 10 km, siendo este el uno de los caminos más cortos.

Ejercicio de Grafo Bipartido

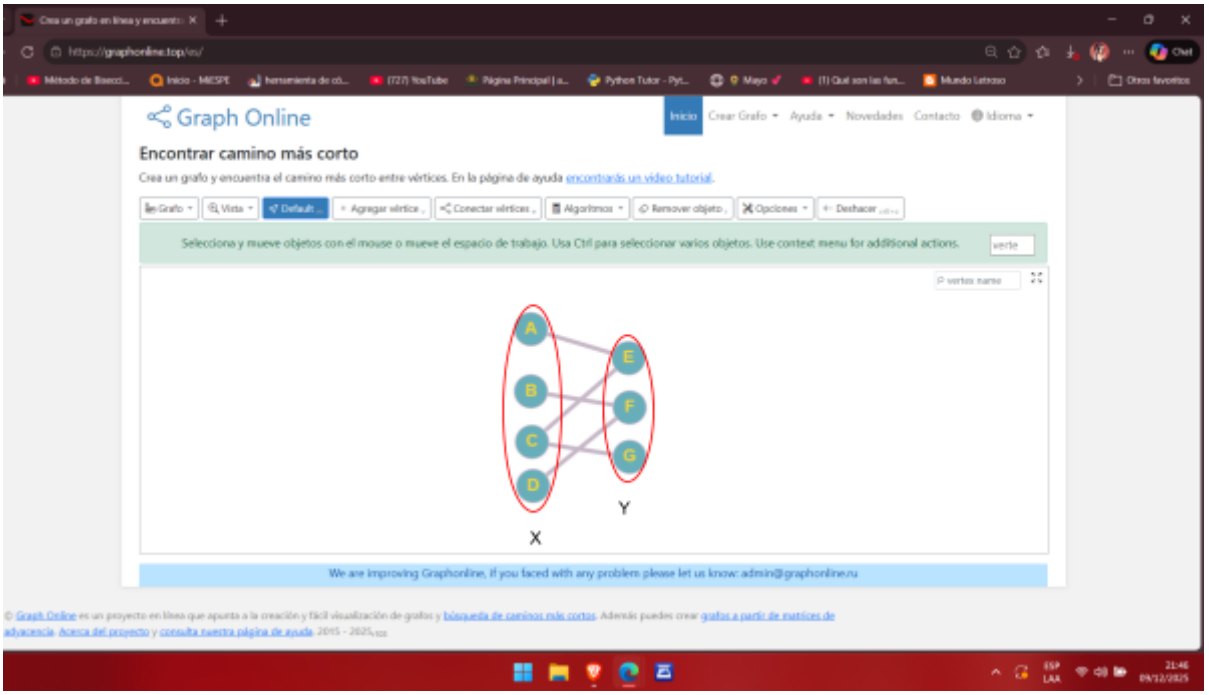
Teniendo el grafo G con vértices V = {A,B,C,D,E,F,G} y aristas {(A,E),(B,F),(C,E),(C,G),(D,F)}

a) Divida los vértices en dos subconjuntos X y Y.

b) Indique si es un grafo bipartido y justifique.

Figura 4

Representación de grafo bipartido.



Nota. Elaboración propia utilizando Graph Online, https://graphonline.top/es/.

Solución: Como se puede visualizar en la imagen se dibujó y separó los vértices en dos subconjuntos X y Y, se evidencia que es un grafo bipartido por que está dividido en dos subconjuntos y es disjunto.

Sintaxis general y estructura técnica en C++

Algoritmo Dijsktra en C++

En C++, los grafos ponderados utilizan el algoritmo Dijkstra para encontrar el camino más corto de un nodo fuente hasta otro nodo del grafo, según (CPP Scripts, 2024) “El algoritmo opera bajo el principio de explorar progresivamente el nodo más cercano que aún no ha sido visitado”.

El grafo se representa mediante una lista de adyacencia o una matriz de adyacencia. La lista es lo más común para representar un grafo, se implementa con:

```
vector<vector<pair<int,int>>>grafo;
```

Donde el vector principal es el nodo y cada pair<int,int> almacena el nodo destino y el peso de la arista. Así esto permite un acceso eficiente a los vecinos de cada nodo.

Funcionamiento técnico en C++

- **Nodos:** Representa los elementos del grafo, en C++ se pueden identificar como índices enteros almacenados en un vector.
- **Aristas:** Es la conexión entre nodos, estas tienen un peso asignado que nos da el costo, distancia o tiempo del recorrido.
- Pesos: Son valores numéricos, importantes para el cálculo de las distancias mínimas, en el código se implementan junto al nodo destino mediante estructuras como pair.

Estructuras auxiliares del algoritmo

- **Inicialización:** Mediante un

  vector<bool> distancia;

  se puede identificar la distancia del nodo origen como 0 hacia los demás nodos como ∞.

- **Cola de prioridad:** Se utiliza una cola de prioridad

  priority_queue<pair<int,int>,vector<pair<int,int>>, greater<pair<int,int>>> pq;

para seleccionar el nodo no visitado con la menor distancia encontrada.

- **Aristas relajante:** Aquí gracias al

```
if (distancia[u] + peso(u,v) < distancia[v]) {
distancia[v] = distancia[u] + peso(u,v);
 }
```

permite actualizar las distancias mínimas de los nodos vecinos cuando encuentra un camino de menor costo.

Recorrido

El algoritmo Dijkstra sigue un recorrido sistemático el cual se basa en una estrategia voraz llamada Greedy, los pasos de esta son los siguientes:

1. Se inicia todo los nodos con ∞ excluyendo al nodo origen.
2. Inserta el nodo con la menor distancia.
3. Extrae el nodo con la menor distancia.
4. Relajar las aristas adyacentes, actualizando las distancias si se logra obtener un camino más corto.
5. Marca el nodo como visitado.
6. El proceso se repite hasta que todos los nodos sean visitados.

Operaciones

- Inicialización de estructuras.
- Actualización de las distancias.
- Marcar nodos visitados.
- Comparación de caminos con distancia mínima.

Grafos bipartidos en C++

Los grafos bipartidos se representan con la misma estructura de datos que se vio anteriormente con el algoritmo dijkstra, donde la lista de adyacencia es la forma más común

para la eficiencia de memoria. Solo que en esta ocasión la bipartición depende de un algoritmo de verificación con una complejidad O(V + E) donde:

V: es el número de vértices

E: es el número de aristas

la cual va determinar si los vértices pueden dividirse en dos conjuntos siendo estos disjuntos. Por eso existen algoritmos como BFS (Breadth-first search) y DFS (Depth First Search) los cuales como menciona (OIA-Wiki, 2017) “estos algoritmos son ideales para detectar si un grafo es bipartito o no. Simplemente arrancamos desde cualquier nodo, vamos pintando de colores distintos al vecino del cual vengo, y si puedo pintar todos sin tener adyacentes del mismo color, es bipartito; si no, no.”

Funcionamiento técnico en C++

- **Nodos:** Representa elementos en el grafo, cada nodo pertenece a uno de los dos subconjuntos divididos y esto se identifica mediante un color o etiqueta durante la ejecución del algoritmo.
- **Aristas:** Es la relación entre los nodos, estas deben conectarse a nodos de conjuntos opuestos, y mediante el algoritmo de “bipartite check” se puede verificar si es bipartido o no.

Estructuras auxiliares del algoritmo

- **Vector de visitados:** Mediante

  vector<bool> visitado;

  se identifica si el nodo ya fue visitado durante el recorrido.
- **Conjuntos:** Gracias a

  vector<int> conjunto;

  logra almacenar el conjunto al que pertenece cada nodo, en este caso como es bipartido solo se permite dos subconjuntos.

- **Cola BFS:** Se utiliza

  queue<int> bfs;

  para implementar BFS, lo cual permite recorrer el grafo por niveles, lo cual asegura la asignación alternada de colores para los conjuntos de nodos vecinos.

- **Variable lógica:** Es usada

  bool bipartito;

  la cual ayuda a identificar si el grafo cumple o no con la condición de bipartición.

Recorrido

Los pasos para la verificación de si el grafo es bipartido o no es:

1. Se asignan dos colores a los nodos.
2. Se alterna el color cada vez que se atraviesa una arista.
3. Explorar progresivamente los nodos vecinos, verificando que cada nodo sea diferente su color al del nodo del cual se accede.

Si se encuentra algun conflicto de color entonces se dice que el grafo no es bipartido.

Operaciones

- Inicialización de la estructura de colores
- Selección de un nodo no visitado
- Asignación de un color inicial
- Recorrido del grafo utilizando BFS o DFS.
- Verificación de conflictos con nodos vecinos.
- Conclusión sobre si es o no bipartido.

Ventajas y desventajas Algoritmo de Dijkstra y Grafos Bipartitos

Algoritmo de Dijkstra

Ventajas

- Encuentra el camino mas optimo y corto lo que garantiza el mínimo costo desde un nodo origen hacia todos los demás.
- Funciona bien para grafos que sean muy grandes, especialmente si utilizamos colas de prioridad.
- Es eficiente cuando las aristas no son negativas, como para redes reales(rutas, mapas, etc).
- Su lógica es intuitiva y está basada en la expansión del nodo con distancia mínima.

Desventajas

- Al existir aristas negativas, dijkstra no funcionará correctamente.
- Puede ser muy lento sin una estructura adecuada, generando un costo computacional demasiado elevado en O(n^2).
- Procesa caminos de un solo origen si lo que se quiere es calcular el camino mínimo entre muchos pares de nodos.
- Requiere que el grafo esté completamente cargado en memoria.

Grafos Bipartidos

Ventajas

- Permiten modelar relaciones entre dos conjuntos disjuntos.
- Se puede colorear en solo 2, verificación rápida de bipartición con BFS o DFS.
- Soporta algoritmos muy eficientes.
- Evitan ciclos impares.

Desventajas

- Son limitados ya que no permiten conexiones dentro del mismo conjunto.

- No representan bien redes con ciclos impares.
- Algunos algoritmos requieren grafos no bipartidos.
- Para modelar ciertos problemas, obligan a dividir la información de manera forzada.

Dijkstra vs Bellman-Ford

Tabla 1

Diferencias principales entre Dijkstra y Bellman-Ford

Característica	Dijkstra	Bellman-Ford
Pesos negativos	No permite pesos negativos.	Permite pesos negativos.
Detección de ciclos negativos	No detecta ciclos negativos.	Detecta ciclos negativos.
Complejidad	Su complejidad es rápida O((V+E)log V).	Es mucho más lento en comparación con Dijkstra siendo O(V.E).
Optimalidad	Produce el camino más corto si no hay pesos negativos volviéndolo óptimo en estos casos.	Produce caminos más cortos incluso con pesos negativos.
Aplicacion tipica	Aplicado a Gps, redes, pathfinding y mapas.	Aplicado en finanzas, redes con pérdidas/ganancias, teorías de grafos.
Tipo de grafo	El tipo de grafo se maneja con G(V,E) con pesos >= 0.	El tipo de grafo puede ser cualquiera excepto con ciclos negativos para el nodo de inicio.

Nota. Características principales sobre ambos algoritmos en un cuadro comparativo.

Grafos bipartidos vs no bipartidos

Tabla 2

Tabla comparativa entre Grafos Bipartidos y no Bipartidos

Grafos Bipartidos	Grafos no Bipartidos
Se puede dividir en dos conjuntos que sean disjuntos, sin necesidad de compartir vértices.	Todos los elementos pueden relacionarse entre sí, no necesita una división obligatoria.
Solo se permite conexiones entre vértices de diferentes conjuntos.	Permite conexiones entre cualquier par de vértices.
Para poder permitirse la bipartición, estos no contienen ciclos impares.	Puede contener ciclos de cualquier tipo, incluidos ciclos impares.
Posee una estructura más regular, facilitando verificar propiedad mediante recorridos.	Posee una estructura más general e irregular, lo que requiere un análisis más amplio para verificar propiedades.
Se colorea utilizando solamente 2 colores.	Dependiendo de su complejidad puede necesitar más de dos colores.
El no poseer ciclos impares ayuda en una clasificación clara entre los dos subconjuntos.	El poseer ciclos impares dificulta separar los vértices en dos grupos.
La bipartición establece límites sobre el tipo de aristas que pueden existir.	Ofrece una mayor libertad al no poseer una limitación respecto a las aristas.
La estructura posee una organización definida por dos partes que deben mantenerse separadas.	La estructura posee una naturaleza libre del grafo sin divisiones ni subconjuntos definidos.

Nota. Cuadro comparativo entre los dos tipos de grafos.

Ejemplo de código Algoritmo Dijkstra

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

class Grafo {
private:
    int n; // número de nodos
    vector<vector<pair<int,int>>> adj; // lista de adyacencia

public:
    // Constructor: inicializa el grafo con n nodos
    Grafo(int nodos) {
        n = nodos;
        adj.resize(n);
    }

    // Método para agregar arista (u -> v con peso w)
    void agregarArista(int u, int v, int peso) {
        // Insertamos el vecino y el peso en la lista del nodo u
        adj[u].push_back({v, peso});
    }

    /*
        Método Dijkstra:
        ----------------
        Calcula la distancia mínima desde un nodo origen
        a todos los demás utilizando una cola de prioridad.
    */
    vector<int> dijkstra(int origen) {
        const int INF = 1e9;
        vector<int> dist(n, INF);
        dist[origen] = 0;

        // Cola de prioridad min-heap (guarda (distancia, nodo))
        priority_queue<pair<int,int>,
                       vector<pair<int,int>>,
                       greater<pair<int,int>>> pq;

        pq.push({0, origen});

        while (!pq.empty()) {
            int dActual = pq.top().first;
            int u = pq.top().second;
            pq.pop();

            // Si ya existe una distancia mejor, ignoramos esta
            if (dActual > dist[u]) continue;

            // Recorremos los vecinos del nodo u
            for (auto &par : adj[u]) {
                int v = par.first;
                int peso = par.second;

                // Relajación (actualizar distancia si encontramos algo mejor)
                if (dist[u] + peso < dist[v]) {
```

```
                    dist[v] = dist[u] + peso;
                    pq.push({dist[v], v});
                }
            }
        }

        return dist;
    }
};

int main() {
    /*
        Creamos un grafo de 5 nodos y simulamos algo parecido
        a un mapa sencillo donde cada peso representa la distancia
        entre lugares.
    */

    Grafo mapa(5);

    // Agregamos conexiones entre lugares
    mapa.agregarArista(0, 1, 4);
    mapa.agregarArista(0, 2, 2);
    mapa.agregarArista(2, 1, 1);
    mapa.agregarArista(1, 3, 3);
    mapa.agregarArista(2, 3, 6);
    mapa.agregarArista(3, 4, 2);

    // Calculamos rutas desde el punto 0
    vector<int> distancias = mapa.dijkstra(0);

    cout << "Distancias minimas desde el nodo 0:\n";
    for (int i = 0; i < distancias.size(); i++) {
        cout << "0 -> " << i << " = " << distancias[i] << endl;
    }

    return 0;
}
```

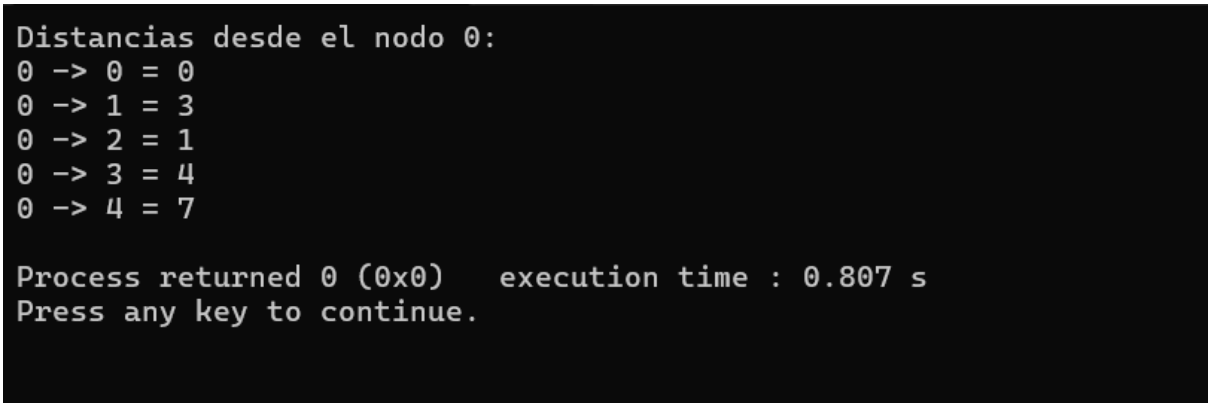
Explicación breve del código

Este programa encapsula la lógica de manejo de nodos, aristas y el algoritmo de Dijkstra, la idea es no trabajar directamente en el main, por lo que primero en la clase se crea una lista de adyacencia donde cada nodo tiene sus vecinos y pesos. Segundo se implementa un método agregarArista() para agregar conexiones al grafo, después dentro de la misma clase se crea el método dijkstra(), que se encarga de calcular las rutas más cortas usando una cola de prioridad.

Al final el main() solo construye el grafo, agrega unas rutas simuladas y se llama al método del TDA, lo que imprime las distancias desde el nodo 0 a los demás.

Figura 5

Compilación del código del algoritmo Dijkstra.



Nota. Captura de pantalla de la ejecución en el compilador de Dev-C + + de la verificación de cada nodo empezando desde 0.

Ejemplo de código Grafo bipartidos

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

class GrafoBipartido {
private:
    int procesos;
    int recursos;
    vector<vector<int>> grafo;        // Procesos -> Recursos
    vector<int> asignacionRecurso;     // Recurso -> Proceso
    vector<bool> visitado;

public:
    // Constructor
    GrafoBipartido(int p, int r) {
        procesos = p;
        recursos = r;
        grafo.resize(procesos);
        asignacionRecurso.assign(recursos, -1);
        visitado.resize(procesos);
    }

    // Agregar arista Proceso -> Recurso
    void agregarArista(int proceso, int recurso) {
        grafo[proceso].push_back(recurso);
    }

    // BFS para asignar un recurso a un proceso
    bool bfs(int procesoInicial) {
        queue<int> q;
```

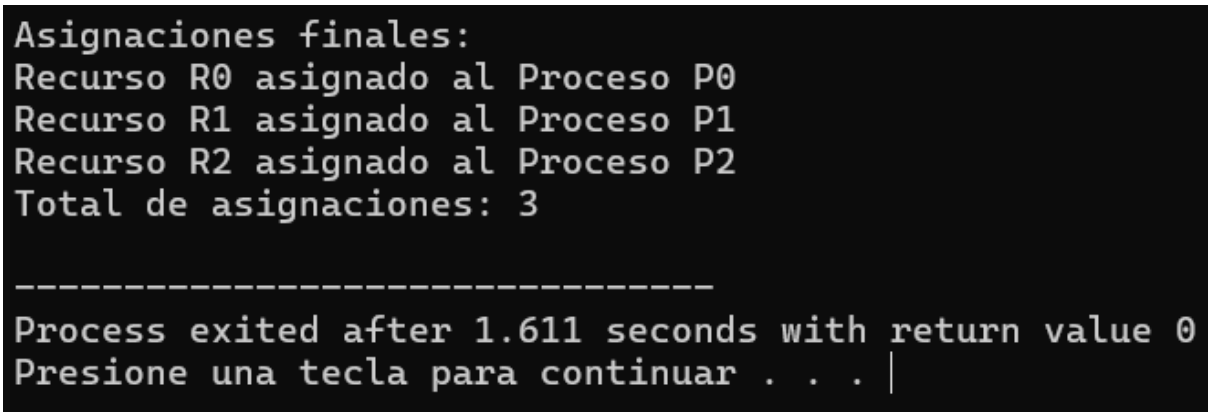
```
        q.push(procesoInicial);
        fill(visitado.begin(), visitado.end(), false);
        while (!q.empty()) {
            int p = q.front();
            q.pop();

            for (int r : grafo[p]) {
                int procesoAsignado = asignacionRecurso[r];

                // Recurso libre
                if (procesoAsignado == -1) {
                    asignacionRecurso[r] = p;
                    return true;
                }
                // Intentar reasignar
                else if (!visitado[procesoAsignado]) {
                    visitado[procesoAsignado] = true;
                    q.push(procesoAsignado);
                }
            }
        }
        return false;
    }

    // Ejecutar la asignacion de recursos
    void asignarRecursos() {
        int asignaciones = 0;
        for (int p = 0; p < procesos; p++) {
            if (bfs(p)) {
                asignaciones++;
            }
        }
        cout << "Asignaciones finales:\n";
        for (int r = 0; r < recursos; r++) {
            if (asignacionRecurso[r] != -1) {
                cout << "Recurso R" << r
                     << " asignado al Proceso P"
                     << asignacionRecurso[r] << endl;
            }
        }
        cout << "Total de asignaciones: " << asignaciones << endl;
    }
};
int main() {
    GrafoBipartido g(3, 3);

    // Conexiones Proceso -> Recursos
    g.agregarArista(0, 0);
    g.agregarArista(0, 1);
    g.agregarArista(1, 1);
    g.agregarArista(2, 0);
    g.agregarArista(2, 2);

    g.asignarRecursos();

    return 0;
}
```

Explicación breve del código

El programa se basa sobre un problema de asignación de recursos mediante un grafo bipartido representado por una lista de adyacencia `vector<vector<int>> grafo;`, donde los procesos y los recursos son parte de los subconjuntos que son disjuntos. Se hace el uso del algoritmo BFS para encontrar caminos alternativos que permitan asignar recursos libres o reasignar procesos previamente emparejados, por medio del `vector<bool> visitado;` se evita ciclos repetitivos. De esta forma como resultado se obtiene una asignación válida almacenada en un vector que maximiza el número de emparejamientos proceso-recurso.

Figura 6

Compilación del código de grafo bipartido.



Nota. Captura de pantalla de la ejecución en el compilador de Dev-C + +, sobre las asignaciones finales de proceso-recurso.

Aplicación práctica o caso de uso

- ***Aplicación del Algoritmo de Dijkstra: GOOGLE MAPS***

  Una de las aplicaciones del algoritmo de Dijkstra es la planificación de rutas en sistemas de navegación. Aplicaciones como Google Maps o Waze utilizan variaciones de Dijkstra y algoritmos similares para determinar el camino más corto entre dos puntos, considerando factores como distancia, tiempo estimado y

condiciones del tráfico. El grafo se modela con ciudades o intersecciones como vértices, y carreteras como aristas ponderadas según el tiempo o la distancia.

- ***Aplicación de Grafos Bipartidos: SISTEMA DE RECOMENDACIONES***

  Los grafos bipartidos se aplican ampliamente en sistemas de recomendación como en servicios como Netflix, Amazon o Spotify. En estos sistemas, se modelan dos tipos conjuntos, por un lado los usuarios y por otro el contenido como películas, música o artículos de venta. Los grafos permiten aplicar algoritmos eficientes para encontrar emparejamientos, similitudes o patrones que facilitan recomendar productos a usuarios con intereses similares.

Conclusiones

Tanto el algoritmo de Dijkstra como los grafos bipartidos representan son herramientas importantes por su capacidad para resolver problemas reales de optimización y modelado de relaciones. Dijkstra destaca por su eficiencia al calcular caminos mínimos en grafos con pesos no negativos, siendo una pieza clave en aplicaciones como navegación, logística, redes y sistemas de comunicación. Mientras que los grafos bipartidos permiten estructurar sistemas de emparejamiento y asignación de manera clara y eficiente. La comparación que se realizó con otros algoritmos, como Bellman-Ford y el proporcionar ejemplo de la vida real de la aplicación de estos casos, nos ayudan a comprender mejor su funcionamiento en un entorno real, demostrando que son herramientas esenciales para resolver problemas computacionales complejos.

Bibliografía

CPP Scripts. (2024, septiembre 21). *Dijkstra's Algorithm in C++: A Quick Guide*. https://cppscripts.com/dijkstra-cpp

Labory Pesquer, G. (2018, junio). *Alianzas en Grafos*. Universidad Politécnica de Madrid. https://oa.upm.es/51662/1/TFG_GLORIA_LABORY_PESQUER.pdf

OIA-Wiki. (2017, diciembre 26). *Grafo Bipartito*. https://wiki.oia.unsam.edu.ar/algoritmos-oia/grafos/grafos-bipartitos

Sánchez Torrubia, G., & Lozano Terrazas, V. M. (2002, junio 16). *Algoritmo de Dijkstra. Un Tutorial Interactivo*. Universidad Politécnica de Madrid. https://cursos.clavijero.edu.mx/cursos/006_md/modulo5/contenidos/documentos/Anexo21-Algoritmo.pdf