Ruta Óptima (Octubre de 2025)

Steve Pineda, Alexis Murcia,
Brayana Fernandez
Facultad de ciencias básicas e Ingeniería, Universidad de los llanos, Villavicencio - Meta, Colombia.

Resumen - En este laboratorio se estudian y aplican los algoritmos clásicos para el cálculo de rutas más cortas en modelos de redes, específicamente Dijkstra y Bellman-Ford, analizando su funcionamiento, restricciones y eficiencia en grafos dirigidos y ponderados. La práctica consiste en ingresar y validar la matriz de adyacencia de un grafo, seleccionar nodos de origen y destino, y ejecutar ambos algoritmos para obtener la ruta más corta y su costo asociado. Además, se implementa una aplicación en Python —como continuidad de la práctica anterior— apoyada en NetworkX y Matplotlib, que permite construir el grafo, visualizarlo y resaltar el recorrido óptimo calculado por cada algoritmo. El sistema genera la representación gráfica del grafo, muestra el camino mínimo encontrado, el peso total y los datos relevantes en una consola de resultados. Con ello se obtienen los insumos formales y visuales necesarios para documentar el comportamiento de Dijkstra y Bellman-Ford, comparando su desempeño y aplicabilidad en distintos tipos de grafos.

Índice de Términos - rutas más cortas; algoritmo de Dijkstra; algoritmo de Bellman-Ford; teoría de grafos; matriz de adyacencia; grafos dirigidos; grafos ponderados; costo del camino; visualización de grafos; Python; NetworkX; Matplotlib.

I. INTRODUCCIÓN

La teoría de grafos proporciona un marco matemático para modelar redes de transporte, comunicación o flujo, donde los nodos representan entidades y las aristas describen conexiones con costos o distancias asociadas. En este laboratorio se utiliza la matriz de adyacencia como representación formal del grafo para validar su estructura y analizar elementos esenciales como nodos, aristas y pesos, paso previo para aplicar los algoritmos de rutas más cortas. El trabajo se centra en la implementación y análisis de los algoritmos Dijkstra y Bellman-Ford, fundamentales para determinar caminos mínimos entre un nodo origen y un destino en grafos dirigidos y ponderados. Dijkstra permite obtener rutas óptimas cuando todos los pesos son positivos, mientras que Bellman-Ford extiende esta capacidad incluso ante la presencia de pesos negativos, además de detectar ciclos de costo negativo.

De manera complementaria, se emplea Python junto con las librerías NetworkX y Matplotlib para construir el grafo a partir de la matriz ingresada, visualizarlo y resaltar la ruta más corta calculada por cada algoritmo. La herramienta desarrollada permite seleccionar nodos, ejecutar ambos métodos, mostrar el costo total del recorrido y generar salidas gráficas y textuales en una consola integrada. Esto facilita la comprensión práctica del comportamiento de los algoritmos y su aplicabilidad en distintos modelos de redes.

II. MARCO TEÓRICO

La teoría de grafos proporciona el marco matemático para modelar sistemas donde existen conexiones entre entidades, representadas mediante un conjunto de nodos V y un conjunto de aristas A. En el contexto del cálculo de rutas más cortas, los grafos se consideran generalmente dirigidos y ponderados, es decir, cada arista posee un sentido y un peso que representa costo, distancia, tiempo u otra magnitud relevante del problema.

Un camino entre dos nodos es una secuencia ordenada de aristas que los conecta, y el costo total del camino corresponde a la suma de los pesos asociados a cada arista recorrida. El problema de la ruta más corta consiste en determinar, para un nodo origen dado, el camino de costo mínimo hacia un destino o hacia todos los demás nodos del grafo.

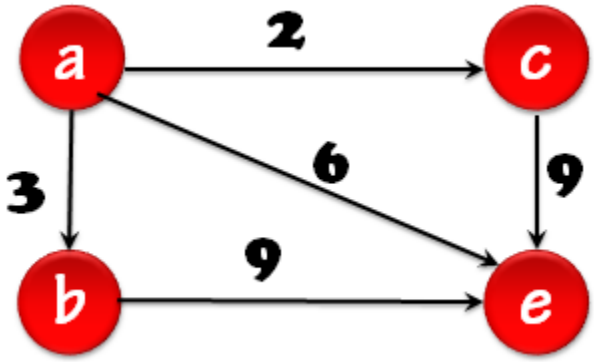

Fig. 1. Grafo no dirigido.

Para representar formalmente un grafo en un programa, se utiliza con frecuencia la **matriz de adyacencia** (Fig. 2). En esta matriz, cada entrada aij indica el peso de la arista que va del nodo i al nodo j. Si no existe conexión directa entre estos nodos, el valor suele ser 0 o un símbolo que represente ausencia de arista; para los algoritmos de rutas más cortas, esta ausencia se interpreta como un valor infinito. Esta estructura permite validar fácilmente la consistencia del grafo y aplicar sobre él los algoritmos correspondientes.

M	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	1
3	0	1	0	1	0
4	1	0	1	0	1
5	0	1	0	1	0

Fig.2. Matriz de adyacencia

Dentro de los métodos clásicos para resolver el problema de rutas más cortas se encuentran dos algoritmos fundamentales:

Algoritmo de Dijkstra

El algoritmo de **Dijkstra** es uno de los métodos clásicos para resolver el problema de rutas más cortas en grafos **dirigidos o no dirigidos**, siempre que **todos sus pesos sean no negativos**. Su funcionamiento se basa en dos ideas principales: la **relajación de aristas** y la **selección del nodo con menor distancia tentativa**.

El algoritmo parte asignando al nodo origen una distancia inicial de **0**, mientras que al resto de nodos se les asigna **infinito**. Luego, mantiene un conjunto de nodos "no visitados" y, en cada iteración, selecciona aquel con la distancia acumulada más baja. Este nodo se considera "definitivo", puesto que —gracias a la ausencia de pesos negativos— se garantiza que no existirá un camino alterno más corto hacia él.

Una vez seleccionado, Dijkstra **relaja** todas las aristas que salen de dicho nodo. La relajación consiste en verificar si avanzar hacia un vecino reduce el costo previamente almacenado; si el nuevo costo es menor, se actualiza la distancia y se registra el nodo predecesor para poder reconstruir el camino óptimo. Este proceso se repite hasta haber fijado las distancias de todos los nodos alcanzables o hasta llegar al nodo destino.

En esencia, Dijkstra combina eficiencia y exactitud mediante una estructura de cola de prioridad que optimiza la selección del "mejor" nodo en cada paso. Por ello, es ampliamente utilizado en aplicaciones reales como sistemas de navegación, redes de comunicación y optimización de rutas logísticas, siempre bajo la condición de pesos no negativos.

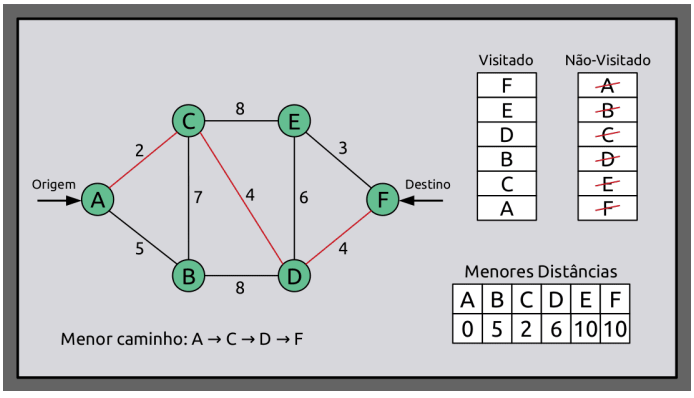

Fig. 3. Algoritmo Djisktra.

Algoritmo de Bellman-Ford

El algoritmo de **Bellman-Ford** resuelve el mismo problema que Dijkstra —encontrar el costo mínimo desde un nodo origen hacia todos los demás— pero con una capacidad adicional crucial: **permite trabajar con grafos que contienen pesos negativos**. Esta característica lo hace especialmente útil en modelos donde pueden existir costos, ganancias o penalizaciones que disminuyen el valor total de un recorrido.

Su procedimiento se basa en aplicar el proceso de **relajación de todas las aristas del grafo de forma exhaustiva**. A diferencia de Dijkstra, que selecciona nodos específicos en cada paso, Bellman-Ford realiza una relajación completa durante $|V|-1|V| - 1|V|-1$ iteraciones, donde $|V||V||V|$ es el número de nodos. El fundamento teórico detrás de esto es que el camino más largo posible en cantidad de aristas dentro de un grafo sin ciclos tiene a lo sumo $|V|-1|V| - 1|V|-1$ conexiones, de modo que cualquier ruta óptima debe quedar completamente determinada tras ese número de relajaciones.

Posterior a estas iteraciones, Bellman-Ford ejecuta una etapa adicional clave: **verificar la presencia de ciclos de costo negativo**. Si al realizar una nueva relajación alguna distancia aún puede disminuir, significa que el grafo contiene un ciclo negativo accesible desde el origen, lo que implica que no existe una solución válida para el problema, ya que el costo podría reducirse indefinidamente.

Aunque su complejidad es mayor que la de Dijkstra, Bellman-Ford destaca por su robustez y aplicabilidad en contextos como análisis financiero, redes con penalizaciones dinámicas y modelos donde pueden existir retroalimentaciones que representan reducciones de costo.

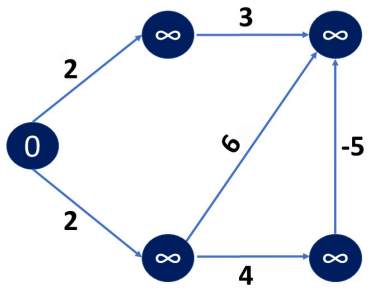

Fig. 4. Algoritmo Bellman-Ford.

III. METODOLOGÍA

El desarrollo del laboratorio se realizó mediante una aplicación interactiva implementada en **Python**, cuyo propósito es permitir la construcción y análisis de un grafo a partir de su **matriz de adyacencia**, y aplicar sobre él los algoritmos de **Dijkstra** y **Bellman-Ford** para el cálculo de rutas más cortas. La metodología integra la lectura y validación de la matriz, la construcción del grafo dirigido o no dirigido, la visualización interactiva y la ejecución paso a paso de los algoritmos, generando resultados gráficos y textuales.

El proceso metodológico aplicado fue el siguiente:

1. `leer_matriz()` – Lectura y validación de datos

Este módulo se encarga de obtener los valores ingresados por el usuario en la matriz de adyacencia.

El sistema recorre cada celda de entrada, verifica que contenga un valor numérico válido (incluyendo pesos negativos), y construye una estructura bidimensional correspondiente al grafo

definido por el usuario.

Además, implementa un mecanismo de validación que garantiza que:

- No existen celdas vacías o caracteres no numéricos.
- En grafos no dirigidos, la diagonal principal sea igual a cero.
- La matriz sea coherente con el tamaño seleccionado por el usuario.

Este proceso asegura la integridad del grafo antes de su construcción.

```python
def leer_matriz(self):
    try:
        n = len(self.campos)
        matriz = []
        for i in range(n):
            fila = []
            for j in range(n):
                val = self.campos[i][j].text().strip()
                if val == "":
                    val = "0"
                if not (val.lstrip("-").isdigit()):···
                val = int(val)
                fila.append(val)
            matriz.append(fila)
        return matriz
    except ValueError as e:
        QMessageBox.critical(self, "Error", f"Error de entrada: {e}")
        return None
```

Fig. 4. Función Leer Matriz.

2. `crear_grafo()` – *Construcción del grafo dirigido/no dirigido*

Posteriormente, el método `crear_grafo()` recibe la matriz validada y construye la estructura interna del grafo utilizando las herramientas de la librería NetworkX. Durante este proceso, se determina automáticamente si la matriz corresponde a un grafo dirigido o no dirigido mediante la verificación de su simetría. Una vez identificado el tipo de grafo, se crean los nodos y se agregan las aristas con sus respectivos pesos, estructurando así una representación formal del grafo que permitirá aplicar los algoritmos de rutas más cortas bajo condiciones reales de conectividad y ponderación.

```python
def crear_grafo(self, matriz):
    dirigido = self.es_dirigido(matriz)
    G = nx.DiGraph() if dirigido else nx.Graph()
    n = len(matriz)
    G.add_nodes_from(range(1, n + 1))
    for i in range(n):
        for j in range(n):
            peso = matriz[i][j]
            if peso != 0:
                G.add_edge(i + 1, j + 1, weight=peso)
    return G, dirigido
```

Fig. 5. Función Crear Grafo.

3. `generar_grafo()` – *Control general del proceso*

El método `generar_grafo()` actúa como coordinador del proceso completo. A partir de la matriz ingresada, invoca los métodos de lectura, validación y creación del grafo, establece la disposición inicial de los nodos en el plano mediante un algoritmo de distribución de fuerzas y actualiza la visualización general del modelo. Además de preparar la representación gráfica, este método genera información textual complementaria como la matriz formateada, la lista de adyacencia y el conjunto de vértices y aristas. De esta forma, se consolidan en una sola rutina tanto la construcción como la presentación inicial del grafo.

```python
def generar_grafo(self):
    matriz = self.leer_matriz()
    if matriz is None:
        return
    valido, mensaje = self.validar_matriz(matriz)
    if not valido:
        QMessageBox.critical(self, "Error", mensaje)
        return
    self.highlight_edges = set()
    self.highlight_nodes = set()
    self.G, self.dirigido = self.crear_grafo(matriz)
    self.pos = nx.spring_layout(self.G, seed=42, k=0.6)
    self.mostrar_grafo()
    self.text_console.clear()
    self.text_console.append("Matriz de adyacencia:")
    self.text_console.append(self.formatear_matriz(matriz))
    self.text_console.append("")
    vertices = list(self.G.nodes())
    aristas = list(self.G.edges())
    self.text_console.append(f"V = {vertices}")
    self.text_console.append(f"E = {aristas}")
    self.text_console.append("")
    self.mostrar_lista_adyacencia()
    self.detectar_ciclos(silent=True)
```

Fig. 6. Función Generar Grafo.

4. `mostrar_grafo()` – *Visualización gráfica*

La representación gráfica del grafo se realiza a través del método `mostrar_grafo()`, que se encarga de dibujar los nodos, aristas y pesos en un entorno interactivo basado en Matplotlib. Esta función incorpora elementos visuales que permiten distinguir aristas dirigidas, pesos, ciclos, rutas especiales y nodos resaltados. Asimismo, habilita una interacción directa con el usuario mediante el movimiento manual de nodos, facilitando la comprensión de la estructura del grafo. El objetivo de este módulo es proporcionar una visualización clara, precisa y dinámica que facilite el análisis del modelo.

```
nx.draw_networkx_nodes(self.G, self.pos, node_color=color_nodos,
                       node_size=tam_nodo, edgecolors="black",
                       linewidths=1.5, ax=self.ax)
if self.highlight_nodes:
    nx.draw_networkx_nodes(···

nx.draw_networkx_labels(self.G, self.pos, font_color="white",···

tipo = "DIRIGIDO" if self.dirigido else "NO DIRIGIDO"
self.ax.set_title(f"GRAFO {tipo}", fontsize=18, fontweight='bold',
                  color='#1d3557', pad=15)
self.ax.axis("off")
self.fig.set_facecolor(fondo)
self.canvas.draw()
```

Fig. 7. Función Mostrar Grafo.

5. `ejecutar_dijkstra()` – *Implementación del algoritmo de Dijkstra*

El algoritmo de Dijkstra se implementa mediante el método `ejecutar_dijkstra()`, el cual desarrolla de forma detallada todas las etapas del procedimiento clásico para grafos con pesos no negativos. Este método inicializa las distancias, selecciona el nodo no visitado con menor costo acumulado y actualiza las distancias de los nodos vecinos mediante relajación de aristas. Durante el proceso, se registran los predecesores necesarios para reconstruir el camino óptimo. Además, el sistema produce una salida textual paso a paso, lo que permite observar la evolución del algoritmo, y finalmente presenta el camino más corto resaltado dentro del grafo.

```
#def ejecutar_dijkstra(self):
#✓ 1. Inicialización de distancias y estructuras
dist = {n: float("inf") for n in self.G.nodes()}
dist[inicio] = 0
previo = {}
visitados = set()
#✓ 2. Bucle principal de selección del nodo con menor distancia
while len(visitados) < len(self.G.nodes()):
    u = min((n for n in self.G.nodes() if n not in visitados),
            key=lambda x: dist[x])
    visitados.add(u)
#✓ 3. Relajación de aristas
for v in self.G.neighbors(u):
    peso = self.G[u][v].get("weight", 1)
    if dist[u] + peso < dist[v]:
        dist[v] = dist[u] + peso
        previo[v] = u
#✓ 4. Reconstrucción del camino (backtracking)
camino = [fin]
while camino[-1] != inicio:
    camino.append(previo[camino[-1]])
camino.reverse()
```

Fig. 8. Partes más importantes de la función Dijkstra.

6. `ejecutar_bellman_ford()` – *Implementación del algoritmo de Bellman-Ford*

En cuanto al algoritmo de Bellman-Ford, este se implementa a través del método `ejecutar_bellman_ford()`. A diferencia de Dijkstra, este procedimiento permite trabajar con grafos que contienen pesos negativos. El método recorre todas las aristas un número de veces igual a la cantidad de nodos menos uno, relajando las distancias siempre que se encuentre un camino más corto. Tras finalizar las iteraciones, se ejecuta una verificación adicional para detectar la existencia de ciclos negativos. Si el grafo es válido, el método reconstruye el camino óptimo entre origen y destino y lo presenta tanto textual como gráficamente. Esta implementación permite comparar el comportamiento de ambos algoritmos bajo distintas condiciones del grafo.

```
# def ejecutar_bellman_ford(self):
# ✓ 1. Inicialización de distancias
dist = {n: float("inf") for n in self.G.nodes()}
dist[inicio] = 0
previo = {}
# ✓ 2. Relajación de aristas durante |V| - 1 iteraciones
for _ in range(len(self.G.nodes()) - 1):
    for (u, v, datos) in self.G.edges(data=True):
        peso = datos.get("weight", 1)
        if dist[u] != float("inf") and dist[u] + peso < dist[v]:
            dist[v] = dist[u] + peso
            previo[v] = u
# ✓ 3. Detección de ciclos negativos
for (u, v, datos) in self.G.edges(data=True):
    peso = datos.get("weight", 1)
    if dist[u] != float("inf") and dist[u] + peso < dist[v]:
        raise Exception("El grafo contiene un ciclo negativo")
# ✓ 4. Reconstrucción del camino (backtracking)
camino = [fin]
while camino[-1] != inicio:
    camino.append(previo[camino[-1]])
camino.reverse()
```

Fig. 9. Partes más importantes de la función ejecutar Bellman-Ford.

7. `resaltar_camino_ui()` – *Resaltado de la ruta más corta*

El proceso de resaltado de la ruta más corta se lleva a cabo mediante el método `resaltar_camino_ui()`, que toma los resultados obtenidos por los algoritmos de Dijkstra o Bellman-Ford y destaca visualmente las aristas y nodos que componen el trayecto óptimo. Este método verifica además que el camino solicitado exista y que el grafo sea compatible con el algoritmo elegido, especialmente en presencia de pesos negativos. Su función es transformar el resultado numérico en una representación visual intuitiva dentro del grafo.

```
def resaltar_camino_ui(self):
    if self.G is None:
        QMessageBox.warning(self, "Sin grafo", "Genera primero el grafo.")
        return
    inicio = int(self.spin_inicio.value())
    fin = int(self.spin_fin.value())
    if inicio not in self.G.nodes() or fin not in self.G.nodes():···
    if self.tiene_pesos_negativos():···
    try:···
    except nx.NetworkXNoPath:···
    except Exception as e:
        self.text_console.append(f"Error buscando camino: {e}")
        QMessageBox.critical(self, "Error", f"Error buscando camino: {e}")
```

Fig. 10. Función resaltar camino más corto.

8. `detectar_ciclos()` – *Análisis estructural del grafo*

Finalmente, el método `detectar_ciclos()` realiza un análisis estructural del grafo identificando la existencia de ciclos. Para grafos dirigidos, emplea procedimientos especializados para encontrar ciclos simples, mientras que en grafos no dirigidos utiliza una base de ciclos. El resultado se muestra en la consola y puede ser resaltado en la representación gráfica. Este análisis

complementa la comprensión estructural del modelo de red y permite anticipar condiciones que podrían afectar los algoritmos de rutas más cortas, como los ciclos negativos.

```python
# def ejecutar_bellman_ford(self):
# ✓ 1. Inicialización de distancias
dist = {n: float("inf") for n in self.G.nodes()}
dist[inicio] = 0
previo = {}
# ✓ 2. Relajación de aristas durante |V| - 1 iteraciones
for _ in range(len(self.G.nodes()) - 1):
    for (u, v, datos) in self.G.edges(data=True):
        peso = datos.get("weight", 1)
        if dist[u] != float("inf") and dist[u] + peso < dist[v]:
            dist[v] = dist[u] + peso
            previo[v] = u
# ✓ 3. Detección de ciclos negativos
for (u, v, datos) in self.G.edges(data=True):
    peso = datos.get("weight", 1)
    if dist[u] != float("inf") and dist[u] + peso < dist[v]:
        raise Exception("El grafo contiene un ciclo negativo")
# ✓ 4. Reconstrucción del camino (backtracking)
camino = [fin]
while camino[-1] != inicio:
    camino.append(previo[camino[-1]])
camino.reverse()
```

Fig. 11. Función de detección de ciclos.

IV. RESULTADOS

El programa desarrollado permitió construir y analizar de manera exitosa diversos grafos definidos por el usuario a través de matrices de adyacencia editables desde la interfaz gráfica. Durante la ejecución de la práctica, fue posible generar grafos dirigidos y no dirigidos, visualizar sus estructuras, identificar aristas y pesos, y reconocer características relevantes como la conectividad, la existencia de ciclos y la organización del grafo mediante su lista de adyacencia. La aplicación mostró correctamente las representaciones visuales solicitadas en cada caso, permitiendo verificar el comportamiento del modelo de red ingresado.

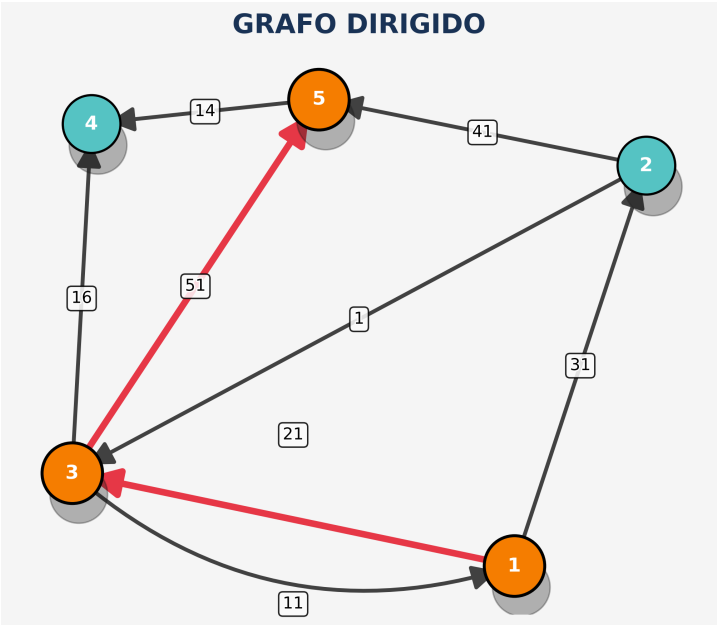

Fig. 12. Ruta más corta usando dijkstra.

La figura 12 presenta el grafo correspondiente a la figura 13, sobre el cual se aplicó el algoritmo de Dijkstra para determinar el camino más corto desde el nodo 1 hasta el nodo 4. El resultado se muestra resaltado en color naranja, lo que permite identificar de manera visual e inmediata la trayectoria óptima calculada por el algoritmo.

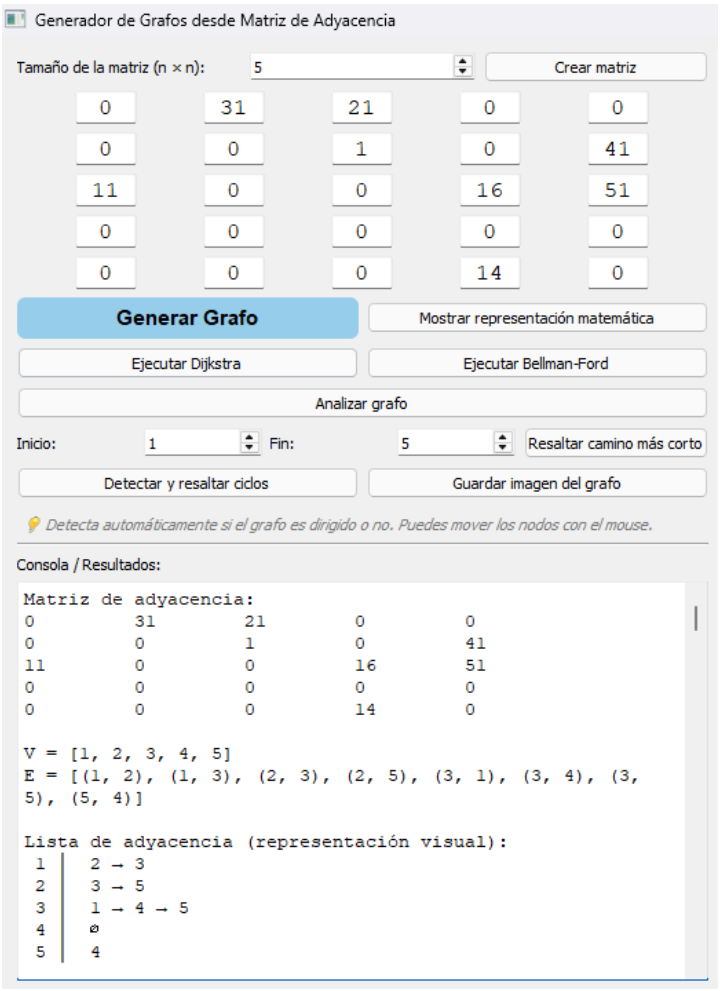

Fig. 13. Matriz de adyacencia con validaciones.

En la figura 13 se observa la interfaz en la que el usuario ingresa la matriz de adyacencia y define los pesos de las aristas para generar el grafo correspondiente. Una vez creada la matriz, la herramienta permite construir el grafo y analizarlo mediante los botones disponibles para ejecutar los algoritmos de Dijkstra o Bellman-Ford. En la parte inferior, la consola muestra de manera estructurada los resultados obtenidos, incluyendo la matriz ingresada, el conjunto de vértices y aristas, la lista de adyacencia y el registro detallado del procedimiento realizado. Esta visualización facilita comprender cómo se interpreta y procesa la información proporcionada por el usuario dentro del modelo de red.

Posteriormente en la consola inferior del programa se muestran los pasos a seguir para hallar el camino más corto con dijkstra.

=== Algoritmo de Dijkstra ===

Nodo de inicio: 1, Nodo final: 5

Inicialización:

{1: 0, 2: inf, 3: inf, 4: inf, 5: inf}

→ Se elige el nodo 1 con menor distancia no visitado (0).

→ Se actualizan los vecinos: [2, 3]

Nodo visitado: 1

Nodo | Distancia

---

1 | 0

2 | 31

3 | 21

4 | ∞

5 | ∞

→ Se elige el nodo 3 con menor distancia no visitado (21).

→ Se actualizan los vecinos: [1, 4, 5]

Nodo visitado: 3

Nodo | Distancia

---

1 | 0

2 | 31

3 | 21

4 | 37

5 | 72

→ Se elige el nodo 2 con menor distancia no visitado (31).

→ Se actualizan los vecinos: [3, 5]

Nodo visitado: 2

Nodo | Distancia

---

1 | 0

2 | 31

3 | 21

4 | 37

5 | 72

→ Se elige el nodo 4 con menor distancia no visitado (37).

→ Se actualizan los vecinos: []

Nodo visitado: 4

Nodo | Distancia

---

1 | 0

2 | 31

3 | 21

4 | 37

5 | 72

→ Se elige el nodo 5 con menor distancia no visitado (72).

→ Se actualizan los vecinos: [4]

Nodo visitado: 5

Nodo | Distancia

---

1 | 0

2 | 31

3 | 21

4 | 37

5 | 72

Camino óptimo encontrado: [1, 3, 5]

Distancia mínima: 72

✅ El camino más corto desde 1 hasta 5 es [1, 3, 5], con un costo total de 72. Esto se obtuvo seleccionando sucesivamente el nodo con menor distancia acumulada.

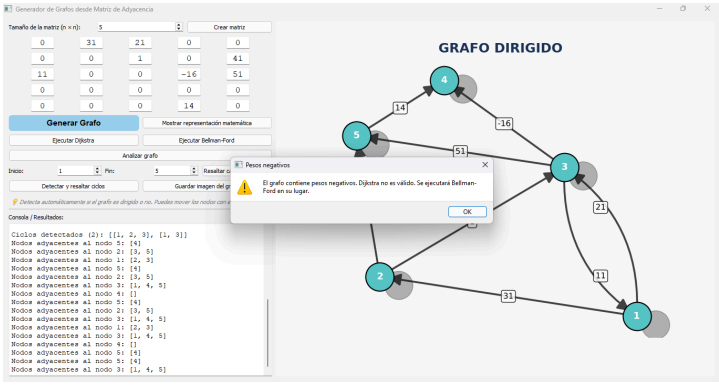

Fig. 14. Validación de pesos negativos.

La figura 14 evidencia el mecanismo de validación implementado por la aplicación para garantizar la correcta selección del algoritmo de rutas más cortas. Al intentar ejecutar Dijkstra sobre un grafo que contiene aristas con pesos negativos, el sistema detecta automáticamente esta condición y muestra un mensaje de advertencia indicando que Dijkstra no es aplicable bajo estas circunstancias. En consecuencia, la herramienta procede a ejecutar de manera automática el algoritmo de Bellman-Ford, el cual sí es capaz de manejar pesos negativos sin comprometer la validez del resultado. Este proceso asegura que el cálculo del camino mínimo se realice utilizando el método adecuado según las características del grafo, evitando errores y preservando la coherencia en el análisis del modelo de red.

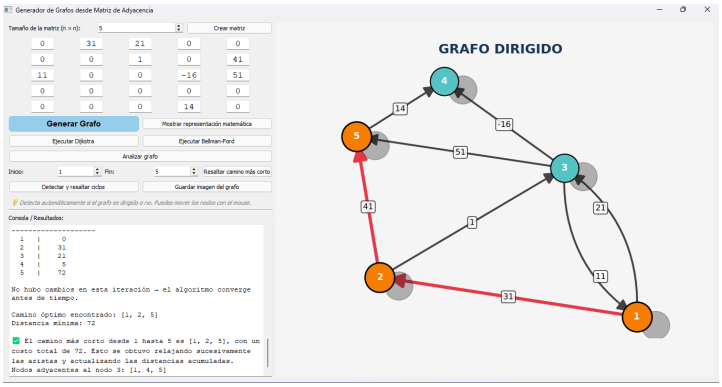

Fig 15. .ruta más corta con el algoritmo bellman-ford.

La figura muestra el resultado de la ejecución del algoritmo de Bellman-Ford sobre un grafo dirigido con aristas que incluyen pesos negativos. Tras detectar esta condición en la matriz de adyacencia, el sistema seleccionó automáticamente Bellman-Ford como método de cálculo para garantizar la validez del resultado. En la parte derecha se visualiza el grafo con su estructura completa, mientras que las aristas que conforman el camino mínimo entre el nodo 1 y el nodo 5 aparecen resaltadas en color rojo, permitiendo identificar de manera inmediata la trayectoria óptima. En la consola de resultados se muestra el proceso iterativo del algoritmo, evidenciando la relajación de aristas, la convergencia anticipada y la reconstrucción final del camino más corto con un costo total de 72. Esta representación confirma tanto la correcta aplicación del algoritmo como la utilidad de la herramienta para analizar grafos con pesos negativos.

Paso a paso de Bellman-Ford:

=== Algoritmo de Bellman-Ford ===

Nodo de inicio: 1, Nodo final: 5
Inicialización:
Nodo | Distancia
---
1 | 0
2 | ∞
3 | ∞
4 | ∞
5 | ∞
Iteración 1:
→ Se relaja la arista (1, 2) con peso 31 → dist[2] = 31
→ Se relaja la arista (1, 3) con peso 21 → dist[3] = 21
→ Se relaja la arista (2, 5) con peso 41 → dist[5] = 72
→ Se relaja la arista (3, 4) con peso -16 → dist[4] = 5
Nodo | Distancia
---
1 | 0
2 | 31
3 | 21
4 | 5
5 | 72
Iteración 2:
Nodo | Distancia
---
1 | 0
2 | 31
3 | 21
4 | 5
5 | 72
No hubo cambios en esta iteración → el algoritmo converge antes de tiempo.
Camino óptimo encontrado: [1, 2, 5]
Distancia mínima: 72
✅ El camino más corto desde 1 hasta 5 es [1, 2, 5], con un costo total de 72. Esto se obtuvo relajando sucesivamente las aristas y actualizando las distancias acumuladas.

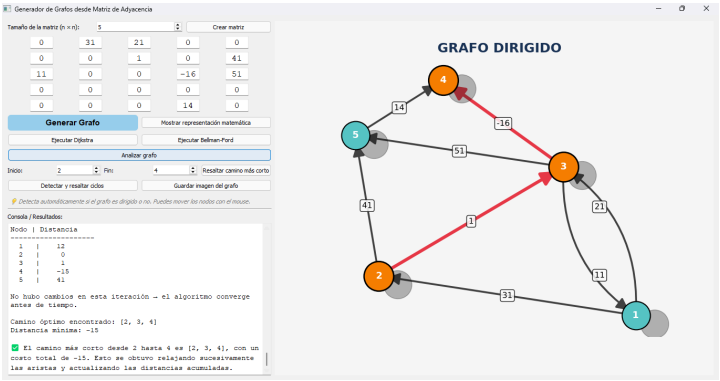

Fig. 16. Función de detección de ciclos.

La figura muestra el cálculo del camino más corto después de modificar los nodos de inicio y fin dentro de la interfaz. En este caso, el usuario seleccionó como nodo de origen el nodo 2 y como destino el nodo 4. Debido a la presencia de un peso negativo en una de las aristas del grafo, el sistema aplicó nuevamente el algoritmo de Bellman-Ford, el cual permite manejar correctamente este tipo de estructuras. En la visualización del grafo pueden observarse resaltadas en color rojo las aristas que conforman la

trayectoria óptima [2,3,4][2, 3, 4][2,3,4], mientras que en la consola se registran las distancias actualizadas en cada iteración, la convergencia del algoritmo y el costo final del recorrido, que en este caso corresponde a −15-15−15. Esta ejecución demuestra la versatilidad de la herramienta al permitir analizar distintos pares de nodos y recalcular dinámicamente la ruta más corta según los parámetros establecidos por el usuario.

En los experimentos realizados, la herramienta generó adecuadamente la representación gráfica de cada grafo, integrando los pesos, direcciones y relaciones entre nodos de forma visualmente clara. En la Fig. 15, el sistema mostró un grafo dirigido creado a partir de una matriz ingresada manualmente, junto con su matriz formateada, su lista de adyacencia y los ciclos presentes. Esta visualización permitió comprobar que la estructura creada por el algoritmo coincidía con la información suministrada por el usuario, lo que demuestra la correcta transformación de la matriz de adyacencia en un modelo gráfico funcional.

Posteriormente, se generaron grafos con aristas de pesos diferentes en ambas direcciones, como se observa en la Fig. 15. En este caso, la herramienta reflejó de forma precisa la asimetría en los pesos, evidenciando que el programa detecta y construye correctamente grafos dirigidos no simétricos. Este comportamiento permitió validar que la aplicación distingue entre aristas bidireccionales con pesos iguales y aristas con pesos diferenciados, lo cual influye directamente en los algoritmos de rutas más cortas.

El sistema también permitió calcular e identificar caminos mínimos entre nodos específicos. La Fig. 16 muestra un ejemplo en el que el usuario selecciona un nodo de inicio y un nodo destino, y la aplicación generó el camino más corto utilizando el algoritmo correspondiente. La visualización resaltó las aristas involucradas en la ruta óptima, permitiendo analizar de manera intuitiva el recorrido seleccionado por el algoritmo. Además del resultado gráfico, el programa generó una explicación textual detallada del proceso, mostrando las distancias actualizadas y los nodos previamente visitados.

Estos resultados en conjunto demuestran que la herramienta cumple con los requerimientos del laboratorio, permitiendo analizar, visualizar y resolver problemas de rutas más cortas en modelos de redes definidos mediante matrices de adyacencia.

V. CONCLUSIONES

1. El desarrollo del laboratorio permitió cumplir satisfactoriamente con el objetivo de aplicar los algoritmos clásicos de rutas más cortas —Dijkstra y Bellman-Ford— sobre grafos definidos mediante matrices de adyacencia. La herramienta construida integró de manera efectiva la teoría de grafos con una implementación computacional interactiva, facilitando el análisis estructural y algorítmico de diferentes modelos de redes.

2. La interfaz desarrollada demostró ser un recurso útil para visualizar y comprender la estructura de un grafo, permitiendo observar de forma inmediata la representación de nodos, aristas, pesos y direccionalidad. La posibilidad de editar la matriz, detectar ciclos, mover nodos y resaltar caminos ayudó a consolidar la comprensión de conceptos como conectividad, bidireccionalidad, ponderación y estructuras cíclicas.

3. La implementación manual de los algoritmos de Dijkstra y Bellman-Ford permitió evidenciar las diferencias fundamentales entre ambos métodos, especialmente en su comportamiento frente a pesos negativos y en la forma en que actualizan las distancias durante sus iteraciones. La visualización del proceso y del resultado reforzó la comprensión del funcionamiento interno de estos algoritmos y su utilidad para resolver problemas de optimización sobre redes.

4. Asimismo, los experimentos realizados mostraron que el uso de herramientas computacionales mejora significativamente la interpretación de los modelos de grafos, al presentar de forma clara tanto la estructura como los cálculos derivados. Esto evidencia que la combinación de teoría matemática y representación visual contribuye a una comprensión más profunda de los problemas de rutas más cortas.

5. En conjunto, el laboratorio permitió no solo aplicar los conceptos teóricos, sino también desarrollar competencias prácticas relacionadas con el análisis, construcción y resolución de grafos en contextos reales de la ingeniería. Los resultados obtenidos confirman la importancia de estos modelos en áreas como transporte, comunicaciones, redes informáticas y optimización de procesos.

VI. BIBLIOGRAFÍA

- [1] R. Diestel, Graph Theory, 5th ed. Springer, 2017.
- [2] J. L. Gersting, Mathematical Structures for Computer Science, 7th ed. W.H. Freeman, 2014.
- [3] A. A. Hagberg, D. A. Schult, and P. J. Swart, “Exploring network structure, dynamics, and function using NetworkX,” Proceedings of the 7th Python in Science Conference, pp. 11–15, 2008.
- [4] R. Sedgewick and K. Wayne, Algorithms, 4th ed. Addison-Wesley Professional, 2011.
- [5] G. Chartrand and L. Lesniak, Graphs & Digraphs, 5th ed. Chapman and Hall/CRC, 2010.
- [6] M. Newman, Networks: An Introduction, Oxford University Press, 2010.
- [7] Python Software Foundation, “NetworkX Documentation,” 2024. [Online]. Available: https://networkx.org
- [8] J. D. Hunter, “Matplotlib: A 2D graphics environment,” Computing in Science & Engineering, vol. 9, no. 3, pp. 90–95, 2007.