***Imagen que contiene Texto

Descripción generada automáticamente******Un conjunto de letras negras en un fondo negro

Descripción generada automáticamente con confianza baja***

***INSTITUTO POLITÉCNICO NACIONAL***

***Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas***

***INGENIERIA EN INFORMATICA***

-

***ALGORITMOS COMPUTACIONALES***

***Algoritmo de Dijkstra***

***Secuencia:3NM30***

***Nombre de integrantes***

|  |  |
| --- | --- |
|  | ***Sandoval González Katya Daniela*** |
|  | ***De Los Santos Lemus Luis Edgar*** |
|  | ***Saldaña Enriquez Carlos Abraham*** |
|  | ***Saucedo Segura César Máximo*** |
|  | ***González Lopez Luis Alberto*** |

***Profesor: Dr. José Luis Goytia***

***Fecha de entrega: 17/06/2024***

El algoritmo de Dijkstra es un algoritmo clásico utilizado en teoría de grafos para encontrar las rutas más cortas desde un nodo origen a todos los otros nodos en un grafo con pesos no negativos en las aristas. Fue propuesto por el científico informático Edsger W. Dijkstra en 1956.

**Funcionamiento del Algoritmo de Dijkstra**

1. Inicialización:
   1. Se asigna una distancia de 0 al nodo origen y una distancia infinita a todos los demás nodos.
   2. Se crea un conjunto de nodos no visitados, inicialmente incluyendo todos los nodos del grafo.
2. Selección del Nodo Actual:
   1. Se selecciona el nodo no visitado con la distancia mínima (inicialmente el nodo origen).
3. Actualización de Distancias:
   1. Para el nodo actual, se revisan todos sus vecinos.
   2. Se calcula la distancia total desde el nodo origen a cada vecino a través del nodo actual.
   3. Si esta distancia es menor que la distancia actualmente registrada para el vecino, se actualiza la distancia.
4. Marcado del Nodo Actual como Visitado:
   1. Una vez que se hayan revisado todos los vecinos del nodo actual, se marca este nodo como visitado y se elimina del conjunto de nodos no visitados.
5. Repetición:
   1. Se repiten los pasos 2 a 4 hasta que todos los nodos hayan sido visitados.

**Ventajas y Desventajas**

**Ventajas**:

* Es eficiente para grafos con pesos no negativos.
* Encuentra la ruta más corta desde un nodo a todos los demás nodos.

**Desventajas**:

* No puede manejar grafos con pesos negativos (para eso se usa el algoritmo de Bellman-Ford).
* Su eficiencia puede verse afectada en grafos muy grandes o densos.

**Complejidad**

La complejidad temporal del algoritmo de Dijkstra es O(V²) para una implementación simple, donde V es el número de nodos. Con el uso de colas de prioridad (heaps), la complejidad puede reducirse a O((V + E)log, donde E es el número de aristas.

**Aplicaciones**

* Sistemas de navegación y GPS.
* Redes de telecomunicaciones.
* Planificación de rutas y logística.
* Análisis de redes sociales.

El algoritmo de Dijkstra es una herramienta fundamental en informática y matemáticas aplicadas, ofreciendo una manera eficiente de resolver problemas de caminos más cortos en grafos.

**Explicación del Código**

1. **Algoritmo de Dijkstra**:

* Se utiliza una cola de prioridad (heapq) para gestionar los nodos a visitar y sus distancias acumuladas.
* Se inicializan todas las distancias a infinito excepto la del nodo de inicio, que se establece en 0.
* Se actualizan las distancias de los nodos vecinos si se encuentra un camino más corto.

1. **Visualización con Matplotlib y NetworkX**:

* Se crea un grafo G usando networkx y se añaden los nodos y aristas del grafo.
* Se dibuja el grafo con matplotlib, incluyendo las etiquetas de las aristas para mostrar los pesos.
* Se resaltan los caminos más cortos desde el nodo inicial a todos los demás nodos en color rojo.

**Creación del Grafo:**

 labels = nx.get\_edge\_attributes(G, 'weight'): Obtiene los pesos de las aristas.

 G = nx.Graph(): Crea un objeto grafo vacío.

 for node, edges in graph.items()...: Añade los nodos y aristas al grafo G utilizando los datos del diccionario graph.

**Resaltado de los Caminos Más Cortos:**

 shortest\_paths = nx.single\_source\_dijkstra\_path(G, start\_node): Obtiene los caminos más cortos desde el nodo de inicio utilizando la implementación de Dijkstra de NetworkX.

 for target\_node, path in shortest\_paths.items():...: Recorre cada camino más corto y dibuja las aristas en rojo para resaltarlas.

Gráfico

Descripción generada automáticamente

Podemos observar que con cada cambio de grafo, realiza una nuevo grafo con los nodos en distintas partes, sin embargo siempre llevando al camino mas corto posible

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

**Código 1:**

import heapq

import matplotlib.pyplot as plt

import networkx as nx

def dijkstra(graph, start):

    # Initialize distances with infinity and set the distance to the start node to zero

    distances = {node: float('infinity') for node in graph}

    distances[start] = 0

    # Priority queue to store nodes and their distances from the start node

    priority\_queue = [(0, start)]

    while priority\_queue:

        current\_distance, current\_node = heapq.heappop(priority\_queue)

        if current\_distance > distances[current\_node]:

            continue

        for neighbor, weight in graph[current\_node].items():

            distance = current\_distance + weight

            if distance < distances[neighbor]:

                distances[neighbor] = distance

                heapq.heappush(priority\_queue, (distance, neighbor))

    return distances

# Define the graph as an adjacency list

graph = {

    'A': {'B': 1, 'C': 4},

    'B': {'A': 1, 'C': 2, 'D': 5},

    'C': {'A': 4, 'B': 2, 'D': 1},

    'D': {'B': 5, 'C': 1}

}

# Run Dijkstra's algorithm

start\_node = 'A'

distances = dijkstra(graph, start\_node)

print("Distances from node", start\_node, ":", distances)

# Visualization

G = nx.Graph()

# Add edges to the graph

for node, edges in graph.items():

    for neighbor, weight in edges.items():

        G.add\_edge(node, neighbor, weight=weight)

# Define positions of the nodes for visualization

pos = nx.spring\_layout(G)

# Draw the graph

plt.figure(figsize=(8, 6))

nx.draw(G, pos, with\_labels=True, node\_color='lightblue', node\_size=500, font\_size=10, font\_weight='bold')

labels = nx.get\_edge\_attributes(G, 'weight')

nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=labels)

# Highlight the shortest paths

shortest\_paths = nx.single\_source\_dijkstra\_path(G, start\_node)

for target\_node, path in shortest\_paths.items():

    path\_edges = list(zip(path, path[1:]))

    nx.draw\_networkx\_edges(G, pos, edgelist=path\_edges, edge\_color='red', width=2)

plt.title("Shortest Paths from Node " + start\_node)

plt.show()

**Codigo 2:**

import heapq

import matplotlib.pyplot as plt

import networkx as nx

import matplotlib.animation as animation

from matplotlib.widgets import Button

# Define the graph as an adjacency list

graph = {

    'A': {'B': 1, 'C': 4},

    'B': {'A': 1, 'C': 2, 'D': 5},

    'C': {'A': 4, 'B': 2, 'D': 1},

    'D': {'B': 5, 'C': 1}

}

# Visualization

G = nx.Graph()

for node, edges in graph.items():

    for neighbor, weight in edges.items():

        G.add\_edge(node, neighbor, weight=weight)

pos = nx.spring\_layout(G)

# Initialize figure and axis

fig, ax = plt.subplots(figsize=(8, 6))

plt.subplots\_adjust(bottom=0.2)  # Adjust the bottom to fit the button

nx.draw(G, pos, with\_labels=True, node\_color='lightblue', node\_size=500, font\_size=10, font\_weight='bold', ax=ax)

labels = nx.get\_edge\_attributes(G, 'weight')

nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=labels, ax=ax)

plt.title("Graph Visualization")

# Add a button

ax\_button = plt.axes([0.45, 0.05, 0.1, 0.075])

button = Button(ax\_button, 'Continue')

# Initialize the priority queue and distances for Dijkstra

distances = {node: float('infinity') for node in graph}

previous\_nodes = {node: None for node in graph}

start\_node = 'A'

distances[start\_node] = 0

priority\_queue = [(0, start\_node)]

visited = set()

# Global variable to hold the animation

ani = None

def dijkstra\_animated(event):

    global ani  # Use the global variable to store the animation

    def update(num):

        if priority\_queue:

            current\_distance, current\_node = heapq.heappop(priority\_queue)

            if current\_distance > distances[current\_node]:

                return

            visited.add(current\_node)

            for neighbor, weight in graph[current\_node].items():

                distance = current\_distance + weight

                if distance < distances[neighbor]:

                    distances[neighbor] = distance

                    previous\_nodes[neighbor] = current\_node

                    heapq.heappush(priority\_queue, (distance, neighbor))

            ax.clear()

            nx.draw(G, pos, with\_labels=True, node\_color='lightblue', node\_size=500, font\_size=10, font\_weight='bold', ax=ax)

            labels = nx.get\_edge\_attributes(G, 'weight')

            nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=labels, ax=ax)

            # Highlight the shortest path edges

            path\_edges = []

            for node in visited:

                if previous\_nodes[node] is not None:

                    path\_edges.append((previous\_nodes[node], node))

            nx.draw\_networkx\_edges(G, pos, edgelist=path\_edges, edge\_color='red', width=2, ax=ax)

            nx.draw\_networkx\_nodes(G, pos, nodelist=visited, node\_color='green', node\_size=500, ax=ax)

    ani = animation.FuncAnimation(fig, update, frames=len(graph) \* 2, repeat=False)

    plt.title("Dijkstra's Algorithm Animation")

    plt.draw()

# Connect the button to the function

button.on\_clicked(dijkstra\_animated)

plt.show()