Instituto Tecnológico y de Estudios Superiores de Monterrey

Análisis y diseño de algoritmos avanzados GPO(601)

Actividad 3. Algoritmo de Dinic

Estudiantes:

Hugo Muñoz Rodríguez / A01736149

Gerardo Deústua Hernández / A01736455

Periodo Agosto – Diciembre 2023

30 / Noviembre / 2023

Algoritmo de Dinic

Intuición detrás del algoritmo

Niveles y Capacidad Residual:
El algoritmo de Dinic emplea la noción de niveles en los nodos del grafo. Estos niveles se calculan mediante BFS desde el nodo fuente. Los niveles permiten restringir la búsqueda de caminos de aumento, evitando la repetición de caminos similares.

Además, utiliza la noción de capacidad residual en las aristas. Si una arista tiene una capacidad y ya se ha enviado un flujo a lo largo de esa arista, la capacidad residual es la cantidad de flujo que aún puede pasar por esa arista, permitiendo encontrar caminos de aumento eficientemente.

Búsqueda de Caminos de Aumento:
El algoritmo busca caminos desde el nodo fuente hasta el nodo objetivo que tengan capacidad residual positiva. Utiliza la información de niveles para encontrar estos caminos de manera más eficiente. Estos caminos se encuentran utilizando BFS que solo avanza a nodos de mayor nivel para garantizar la eficiencia en la búsqueda.

Actualización de Flujos:
Una vez que se encuentra un camino de aumento, el algoritmo actualiza el flujo a lo largo de ese camino, aumentando el flujo y disminuyendo la capacidad residual en las aristas que forman el camino. Esto se hace de manera consistente, asegurándose de mantener la validez del flujo.

Iteraciones y Mejora Incremental:
El algoritmo sigue buscando caminos de aumento y actualizando flujos hasta que no se pueda encontrar más camino de aumento desde el nodo fuente al nodo objetivo. A medida que avanza, cada iteración incrementa el flujo total en la red, aumentando progresivamente hacia el flujo máximo posible.

Elaboración actividad 3

Implementamos el algoritmo de Dinic para encontrar el flujo máximo en un grafo dirigido con capacidades en sus aristas. Nuestro enfoque se basa en una serie de funciones que nos permiten realizar pasos específicos del algoritmo.

Inicialmente, leemos el grafo desde un archivo “.txt” de entrada, definiendo nodos, bordes y sus capacidades. Posteriormente, visualizamos el grafo en diferentes etapas utilizando funciones específicas para mostrar nodos, bordes, capacidades y niveles calculados para cada nodo.

```python
def read_graph(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()
        source, target, num_edges = map(int, lines[0].split())
        edges = [tuple(map(int, line.split())) for line in lines[1:]]

    graph = nx.DiGraph()
    graph.add_nodes_from(range(source, target + 1))
    graph.add_edges_from([(u, v, {'capacity': c, 'flow': 0}) for u, v, c in edges])

    return graph, source, target

def visualize_graph(graph, levels=None, path=None, title="Graph"):
    pos = nx.spring_layout(graph)

    if levels:
        node_colors = [levels[node] for node in graph.nodes()]
        edge_labels = {(u, v): graph[u][v]['capacity'] for u, v in graph.edges()}
        nx.draw(graph, pos, with_labels=True, font_weight='bold', node_color=node_colors, cmap=plt.cm.Blues, width=2)
        nx.draw_networkx_edge_labels(graph, pos, edge_labels=edge_labels, font_color='red')
    elif path:
        edge_colors = ['red' if edge in path else 'black' for edge in graph.edges()]
        nx.draw(graph, pos, with_labels=True, font_weight='bold', edge_color=edge_colors, width=2)
    else:
        edge_labels = {(u, v): graph[u][v]['capacity'] for u, v in graph.edges()}
        nx.draw(graph, pos, with_labels=True, font_weight='bold', width=2)
        nx.draw_networkx_edge_labels(graph, pos, edge_labels=edge_labels, font_color='red')

    plt.title(title)
    plt.show()
```

Calculamos los niveles de cada nodo en el grafo desde el nodo fuente mediante una búsqueda en amplitud (BFS). Estos niveles son cruciales para controlar el flujo de las iteraciones en el algoritmo de Dinic.

```python
def calculate_levels(graph, source):
    levels = {source: 0}
    queue = collections.deque([source])

    while queue:
        current_node = queue.popleft()
        for successor in graph.successors(current_node):
            if successor not in levels:
                levels[successor] = levels[current_node] + 1
                queue.append(successor)

    return levels

def visualize_graph_with_levels(graph, levels, title="Graph with Levels"):
    pos = nx.spring_layout(graph)
    node_colors = [levels[node] for node in graph.nodes()]
    edge_labels = {(u, v): graph[u][v]['capacity'] for u, v in graph.edges()}
    nx.draw(graph, pos, with_labels=True, font_weight='bold', node_color=node_colors, cmap=plt.cm.Blues)
    nx.draw_networkx_edge_labels(graph, pos, edge_labels=edge_labels)

    for node, (x, y) in pos.items():
        plt.text(x, y, f"Level: {levels[node]}", fontsize=8, ha='center', va='center', bbox=dict(facecolor='white', alpha=0.5))

    plt.title(title)
    plt.show()
```

Utilizamos el BFS para encontrar caminos desde el nodo fuente hasta el nodo objetivo, considerando únicamente los bordes con capacidad disponible. Luego, actualizamos el flujo a lo largo de los caminos de aumento, disminuyendo las capacidades residuales en las aristas y aumentando los flujos.

```python
def bfs(graph, source, target):
    visited = set()
    queue = collections.deque([(source, [source])])

    while queue:
        current_node, path = queue.popleft()
        visited.add(current_node)

        for successor in graph.successors(current_node):
            if successor not in visited and graph[current_node][successor]['capacity'] > 0:
                if successor == target:
                    return path + [successor]
                queue.append((successor, path + [successor]))

    return []

def update_flow(graph, path):
    min_capacity = min(graph[u][v]['capacity'] for u, v in zip(path[:-1], path[1:]))

    for u, v in zip(path[:-1], path[1:]):
        graph[u][v]['capacity'] -= min_capacity
        graph[u][v]['flow'] += min_capacity

        if v in graph:
            if u in graph[v]:
                graph[v][u]['capacity'] += min_capacity
                graph[v][u]['flow'] -= min_capacity
            else:
                graph.add_edge(v, u, capacity=min_capacity, flow=-min_capacity)
```

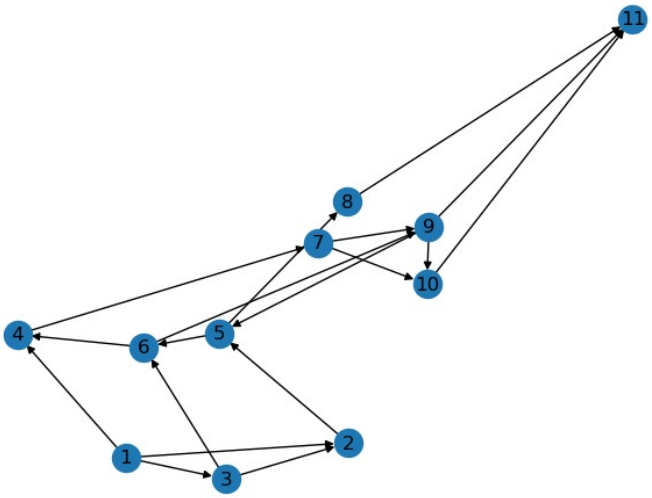
Nuestra implementación del algoritmo de Dinic utiliza estas funciones para realizar pasos clave del algoritmo, como la búsqueda de caminos de aumento y la actualización de flujos, repitiendo estos procesos hasta que no se pueda encontrar un camino de aumento adicional.

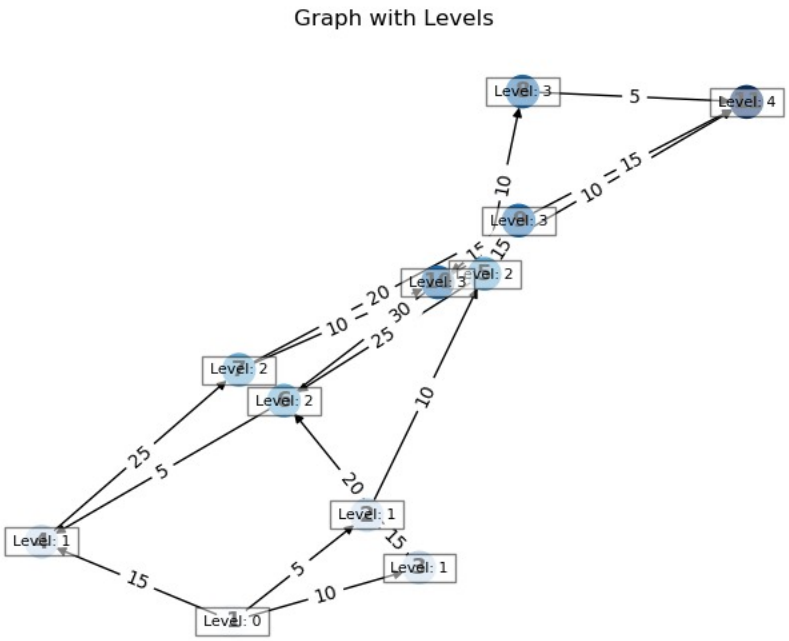
```python
def dinic(graph, source, target):
    levels = calculate_levels(graph, source)
    visualize_graph_with_levels(graph, levels, title="Graph with Levels (Before Dinic)")

    while levels[target] is not None:
        path = bfs(graph, source, target)

        if not path:
            break

        visualize_graph(graph, path=path, title="Graph with Augmenting Path")
        update_flow(graph, path)

        levels = calculate_levels(graph, source)
        visualize_graph_with_levels(graph, levels, title="Graph with Updated Levels")
```

Finalmente, dividimos el código en secciones que van desde la lectura del grafo hasta la visualización del grafo final después de aplicar el algoritmo de Dinic. Esta estructura clara nos permitió seguir una ejecución paso a paso del algoritmo y la visualización de los resultados en cada iteración.

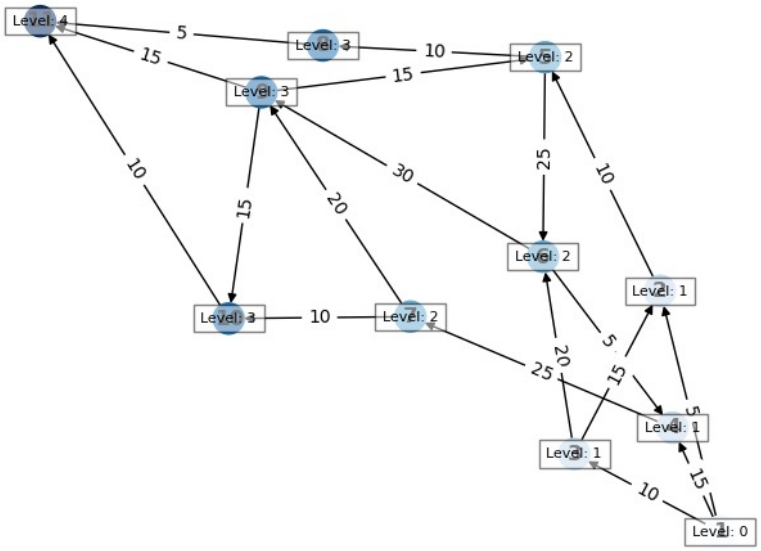
```python
#%%
# Lectura de archivo de texto
file_path = "input.txt"  #<---------------------------- Nombre de archivo
g, s, t = read_graph(file_path)

# Dibujado del grafo leido en el txt
labels = dict([(n, n) for n in g.nodes()])
nx.draw(g, labels=labels)
plt.show()

# Inicio de algoritmo de Dinics, la primera función es el BFS para el calculo de niveles y despues realiza el dibujado del grafo de nivel
levels = calculate_levels(g, s)
visualize_graph_with_levels(g, levels, title="Graph with Levels")

#%%
# Realiza la actualizacion de flujo dependiendo los pesos y dibuja los resultados obtenidos
dinic(g, s, t)

#%%
# Dibujado final del grafo resultante
visualize_graph(g, title="Final Graph")
```

OUTPUTS

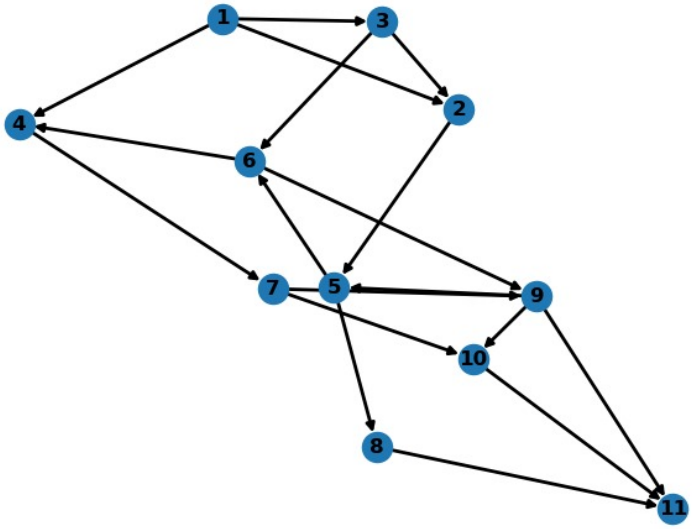

11
8
9
7
10
4
6
5
2
1
3


Graph with Levels
Level: 3
5
Level: 4
10
10
15
Level: 3
15
15
Level: 3
Level: 2
10
20
30
25
Level: 2
Level: 2
25
5
20
10
Level: 1
Level: 1
15
15
5
10
Level: 0

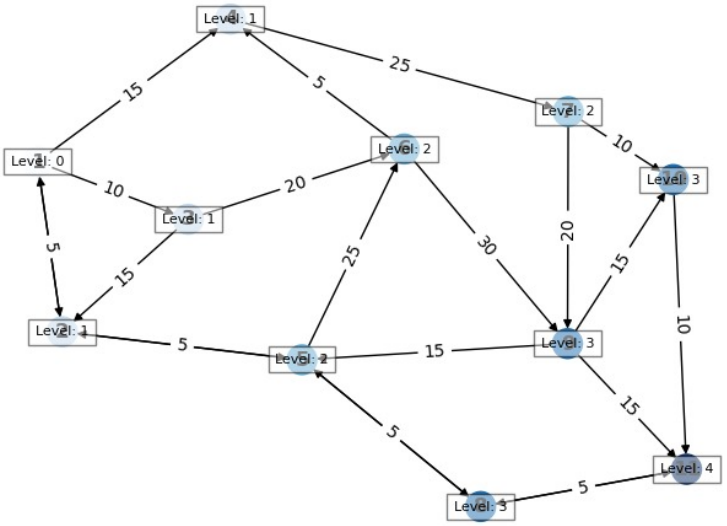
Graph with Levels (Before Dinic)



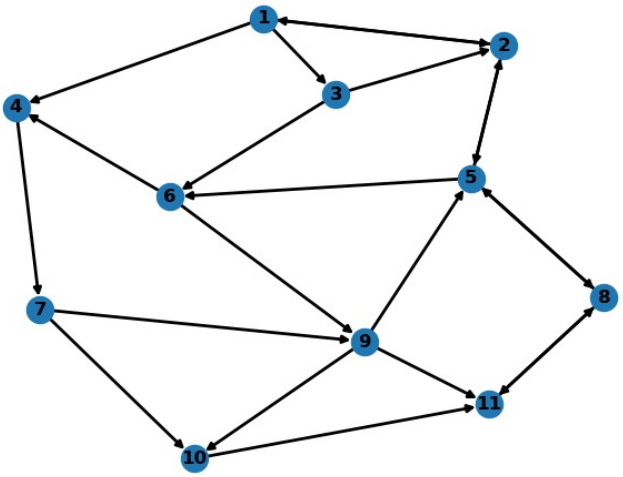
Graph with Augmenting Path



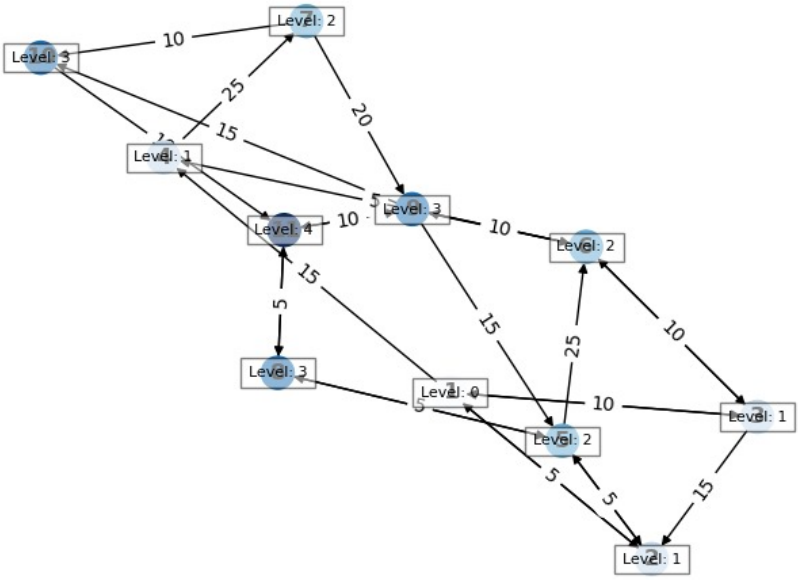
Graph with Updated Levels



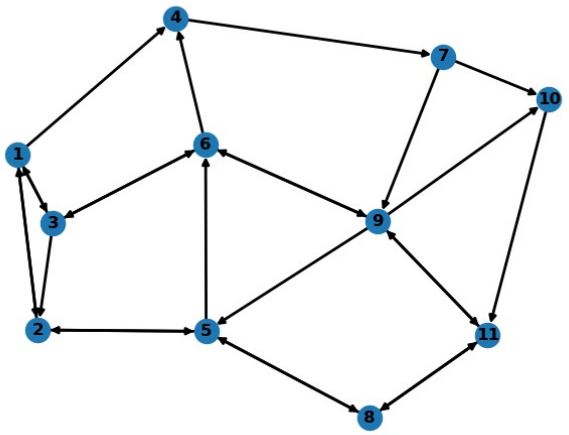
Graph with Augmenting Path



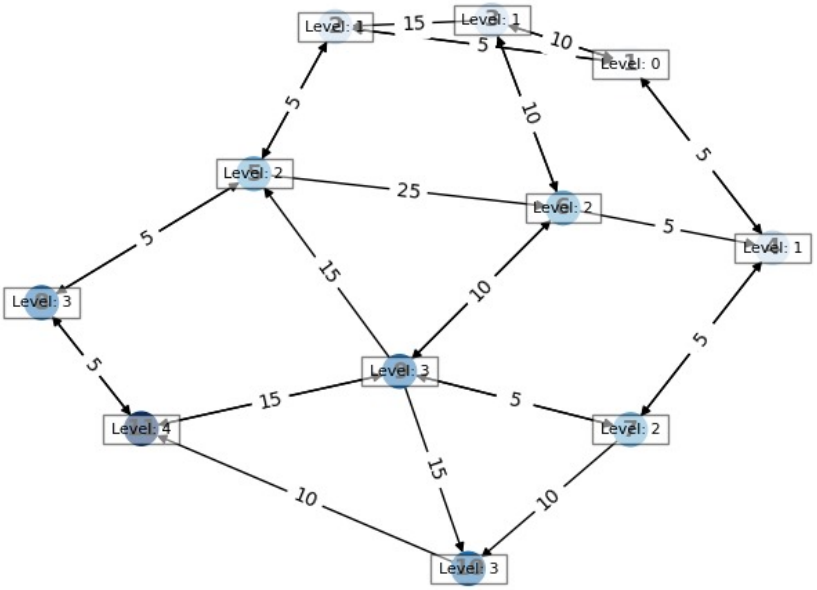
Graph with Updated Levels



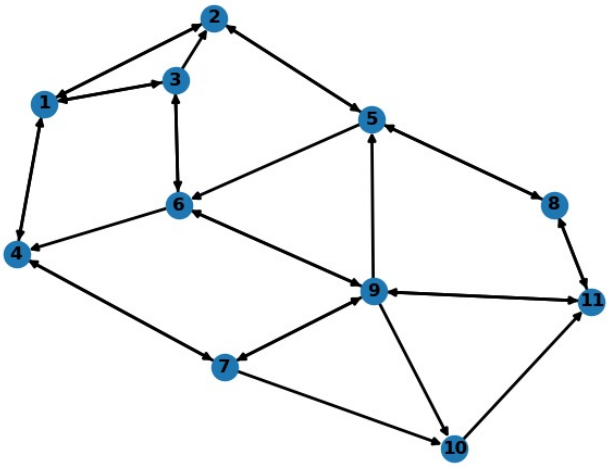
Graph with Augmenting Path



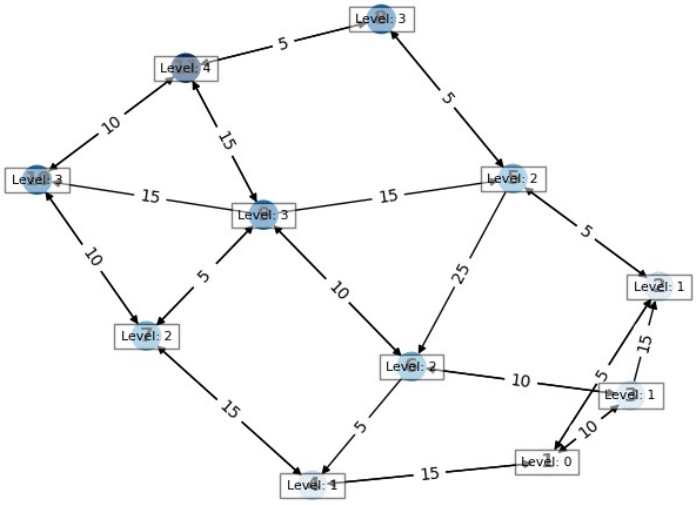
Graph with Updated Levels



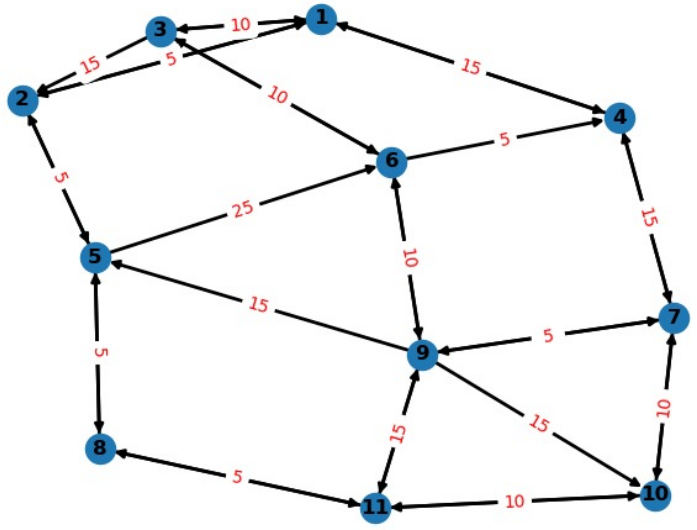
Graph with Augmenting Path



Graph with Updated Levels



Final Graph



Complejidad:

DINIC

El algoritmo de Dinic es conocido por su eficiencia en encontrar el flujo máximo en redes con capacidades en los bordes. En términos de complejidad, en el peor de los casos, la complejidad temporal del algoritmo de Dinic es de O(V^2 * E), donde “V” representa el número de nodos (vértices) en la red y “E” es el número de bordes (aristas).

Sin embargo, si se considera que las capacidades de los bordes están acotadas, la complejidad se reduce a O(E * V^2) . Esta mejora se debe a que, en este escenario, el número de iteraciones necesarias para encontrar el flujo máximo se reduce significativamente.

El algoritmo de Dinic es considerablemente más eficiente en comparación con otros algoritmos clásicos para encontrar flujos máximos.

BFS

BFS se utiliza para recorrer o buscar en grafos o árboles, nivel por nivel, comenzando desde un nodo inicial dado. La complejidad temporal de BFS es O(V + E), donde “V” representa el número de nodos (vértices) en el grafo y “E” es el número de bordes (aristas).

En un grafo denso (donde el número de bordes es cercano a V^2), la complejidad puede aproximarse a O(V^2). En un grafo disperso (donde el número de bordes es mucho menor que V^2), la complejidad se acerca a O(V).

BFS explora todos los nodos y bordes exactamente una vez, por lo que su complejidad se relaciona linealmente con la cantidad de nodos y bordes en el grafo.

Reflexiones

Gerardo Deustúa Hernández:

La implementación del algoritmo de Dinic demuestra una sólida comprensión de los conceptos fundamentales de flujo en redes. La elección de utilizar NetworkX y Matplotlib para visualizar el grafo y sus transformaciones mediante la ejecución con la libreta interactiva de jupyter fue algo interesante. El código presenta una estructura organizada, haciendo tratando de dividir las funciones para mayor claridad y legibilidad.

Hugo Muñoz Rodríguez:

Ayudar a optimizar el flujo en un grafo ha sido un desafío intrigante. Este código muestra cómo los algoritmos pueden resolver problemas del mundo real, como la optimización de redes, y cómo la visualización puede ayudar a comprender el flujo de datos en un grafo.

Además, esta experiencia me ha recordado la importancia de la colaboración. Los algoritmos como Dinic se basan en principios sólidos y colaboraciones entre nodos en una red, y de manera similar, mi aprendizaje se ha basado en la colaboración con mi compañero para obtener el resultado esperado.

LINK DE REPOSITORIO

https://github.com/GerryDH2807/Act-3-Algoritmo_de_Dinic