Instituto Artek

Alumno Joab Ramos Vivas

Matricula ACM30587

Proyecto: Max flow min cut

Materia Análisis de algoritmos

Profesor: Miguel Ángel Márquez Hidalgo

Índice

Antecedentes

Max Flow (Flujo Máximo) y Min Cut (Corte Mínimo):

Max Flow (Flujo Máximo): En teoría de redes, el problema del flujo máximo trata de encontrar la cantidad máxima de flujo que puede pasar a través de una red de nodos y arcos desde una fuente (source) hasta un sumidero (sink) mientras se respetan las capacidades de los arcos. El objetivo es maximizar la cantidad de flujo que se puede enviar de la fuente al sumidero sin exceder las capacidades de los arcos. Este problema tiene aplicaciones en una variedad de campos, incluyendo la planificación de redes de transporte, la distribución de recursos y la optimización de flujos de información.

Min Cut (Corte Mínimo): El corte mínimo en una red es un conjunto de arcos que, si se eliminan de la red, impiden que la fuente y el sumidero estén conectados. La capacidad de un corte mínimo se define como la suma de las capacidades de los arcos en ese corte. En otras palabras, representa la capacidad mínima necesaria para desconectar la fuente del sumidero en la red. En el contexto del flujo máximo, el teorema del corte mínimo establece que el valor del flujo máximo en una red es igual a la capacidad mínima de un corte mínimo.

Importancia de los Algoritmos:

Los algoritmos que resuelven el problema del flujo máximo, como Ford-Fulkerson, Edmonds-Karp y Dinic, son fundamentales en la optimización y la teoría de redes debido a su amplia gama de aplicaciones. Aquí se menciona la importancia de estos algoritmos:

Ford-Fulkerson: Este algoritmo introdujo por primera vez el concepto de flujo máximo y corte mínimo en las redes. Aunque es muy efectivo en muchos casos, no garantiza la terminación en todas las situaciones. Sin embargo, su idea central de encontrar caminos aumentantes sigue siendo la base de muchos otros algoritmos.

Edmonds-Karp: Este algoritmo es una variante específica del Ford-Fulkerson que utiliza búsqueda en anchura (BFS) para encontrar caminos aumentantes. La importancia de Edmonds-Karp radica en su capacidad para garantizar la terminación en tiempo polinómico, lo que lo hace adecuado para aplicaciones en las que la eficiencia es fundamental.

Dinic: El algoritmo de Dinic es altamente eficiente y se basa en la búsqueda en anchura múltiple. A diferencia de otros algoritmos, como Ford-Fulkerson, Dinic trabaja con niveles de nodos y utiliza estructuras de datos avanzadas para acelerar el proceso de búsqueda. Es especialmente útil en redes con capacidades grandes y se considera uno de los algoritmos más rápidos para resolver el problema del flujo máximo.

Los algoritmos de flujo máximo y corte mínimo son esenciales en la optimización de redes y la planificación de flujos en una variedad de aplicaciones del mundo real. Permiten encontrar soluciones eficientes para problemas complejos de asignación de recursos y transporte, y sus variantes ofrecen diferentes enfoques para resolver estos problemas en función de las necesidades de eficiencia y terminación garantizada.

Teorema

El **teorema de flujo máximo y corte mínimo** establece que el flujo máximo desde una fuente determinada a un sumidero determinado en una red es igual a la suma mínima de un corte que separa la fuente del sumidero.

Definición

La red que se analiza aquí es un gráfico dirigido G = (V, E) de vértices conectados por aristas con pesos. Los datos fluyen desde un nodo de origen (grado de entrada 0) a un nodo receptor (grado de salida 0). Cada peso denota la capacidad del borde que representa el flujo máximo a través de ese borde. El teorema conecta dos cantidades, el flujo máximo y el corte de capacidad mínima. Para profundizar más, necesitamos definir el concepto de flujo y corte.

Flow

Para simplificar, **el flujo** se puede imaginar como un flujo físico de un fluido a través de la red que se mueve desde la fuente hasta el sumidero a través de los bordes dirigidos. Cada borde tendrá un flujo que no puede ser mayor que la capacidad del borde. Piense en ello como agua que fluye a través de una tubería, donde el flujo no puede ser mayor que la capacidad de la tubería.

Cut

Un **corte st** divide el gráfico en dos subconjuntos separados , con la fuente en una parte y el sumidero en otra. Un corte solo será válido si separa completamente la fuente del sumidero, es decir, que no exista un camino que los conecte. Alternativamente, un corte es un conjunto de aristas cuya eliminación divide la red en dos mitades X e Y, donde $s \in X$ y $t \in Y$. Además, un corte tiene una capacidad que es igual a la suma **de** las capacidades de las aristas en el corte.

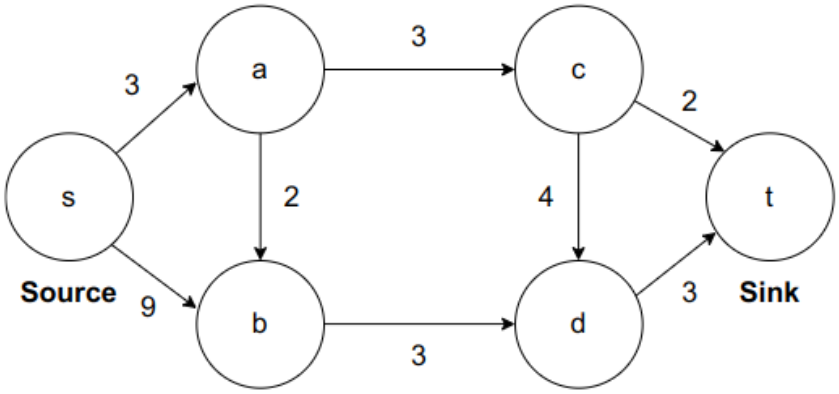
Procedimiento

El objetivo es encontrar el corte mínimo de capacidad que dictará el flujo máximo alcanzable en una red de flujo.

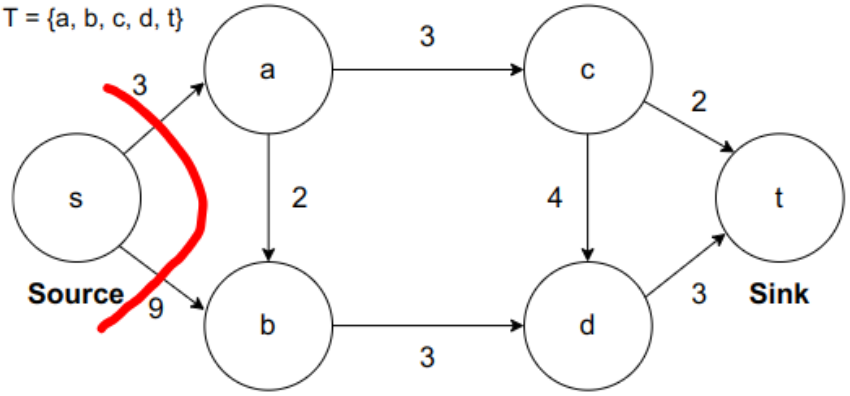
A continuación, se puede apreciar los pasos que se deben seguir en Max Flow min cut.

El procedimiento para encontrar el flujo máximo en un grafo dirigido usando el algoritmo Max Flow - Min Cut implica la búsqueda de caminos desde un nodo de origen hacia un nodo de destino, con el objetivo de enviar la máxima cantidad de flujo posible a lo largo de estos caminos sin exceder las capacidades de las aristas.

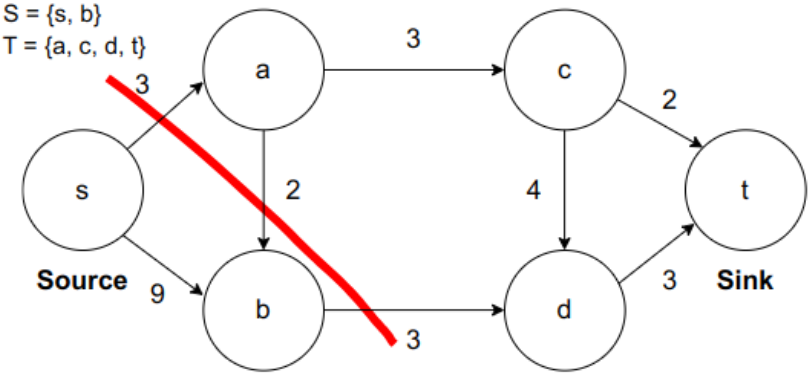
Este proceso se repite hasta que ya no se pueden encontrar más caminos aumentantes, y el valor total del flujo enviado representa el flujo máximo. Opcionalmente, el concepto de corte mínimo se utiliza para identificar las aristas que dividen el grafo en dos conjuntos disjuntos, lo que también equivale al valor del flujo máximo.



Cut 1 Capacity: 3 + 9 = 12
Result of cut:
S = {s}
T = {a, b, c, d, t}

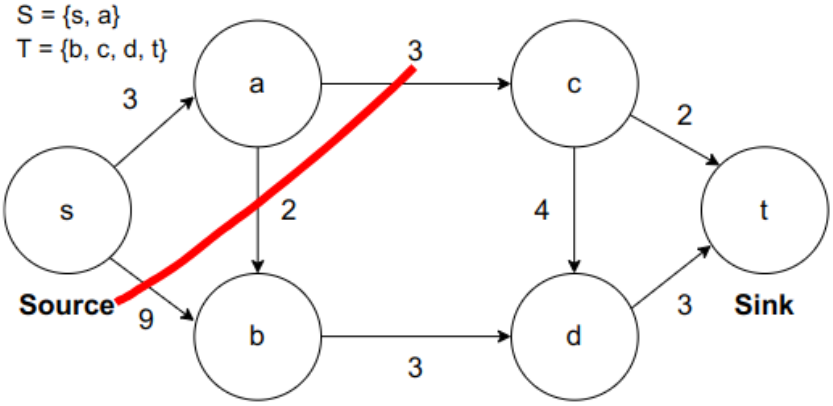


Cut 2 Capacity: 3 + 0 + 3 = 6
Result of cut:
S = {s, b}
T = {a, c, d, t}

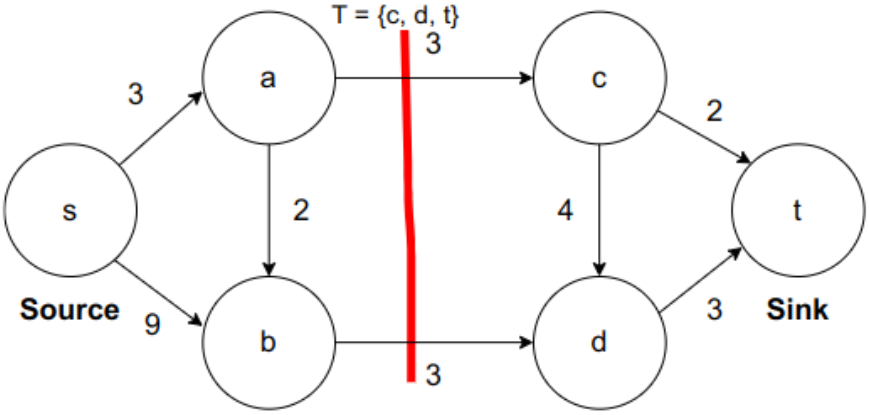


edge (a,b) is not counted as its direction is from sink subset towards source subset
Edges are only counted if they flow from source towards sink (left to right)

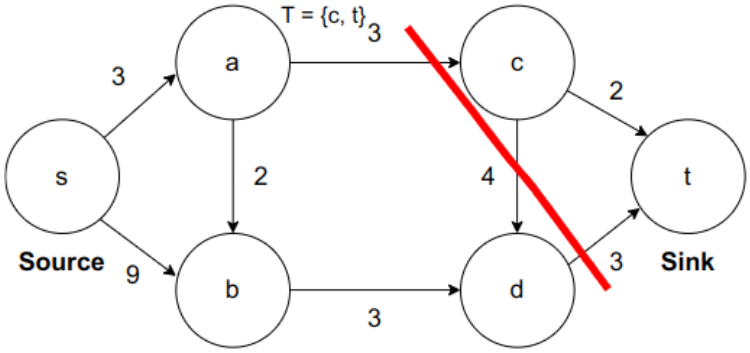
Cut 3 Capacity: 3 + 2 + 9 = 14
Result of cut:
S = {s, a}
T = {b, c, d, t}



Cut 4 Capacity: 3 + 3 = 6
Result of cut:
S = {s, a, b}
T = {c, d, t}

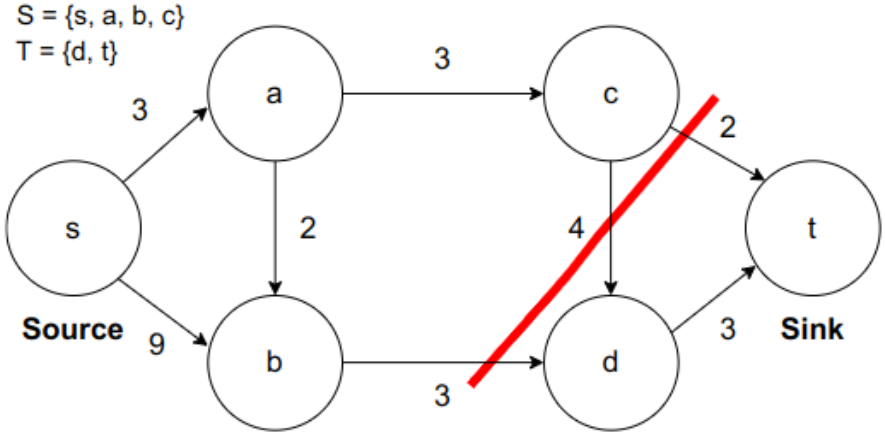


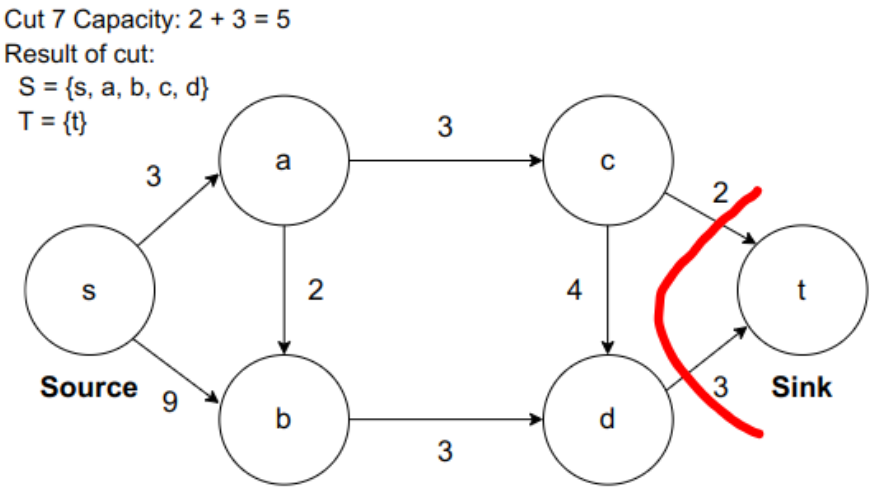
Cut 5 Capacity: 3 + 0 + 3 = 6
Result of cut:
S = {s, a, b, d}
T = {c, t}



edge (c,d) is not counted as its direction is from sink subset towards source subset
Edges are only counted if they flow from source towards sink (left to right)

Cut 6 Capacity: 3 + 4 + 2 = 9
Result of cut:
S = {s, a, b, c}
T = {d, t}





Cut	Capacity
Cut 1	12
Cut 2	6
Cut 3	14
Cut 4	6
Cut 5	6
Cut 6	9
Cut 7	5

Cut 7 is the minimal Cut
The value of the Maximal flow is equal to the capacity of the Minimal cut

Conclusion: Max Flow = Min Cut = 5

Teorema de flujo máximo y corte mínimo

El **teorema de flujo máximo y corte mínimo** establece que el flujo máximo desde una fuente determinada a un sumidero determinado en una red es igual a la suma mínima de un corte que separa la fuente del sumidero.

Para este proyecto agregue 3 posibles algoritmos que sirven para resolver Max Flow min cut los cuales son Ford Fulkerson, Edmonds-Karp y Dinic:

Link para poder comprar las 3 alternativas.

https://colab.research.google.com/drive/1T3KYeZyqA4-2EG8bLSvsFn2wWMcm-NbT?usp=sharing

Algoritmo

Seudocódigo Ford Fulkerson.

Paso 1: Inicialización

- Comenzamos con un flujo máximo igual a cero.
- Creamos un grafo residual que es una copia del grafo original. Inicialmente, todas las capacidades residuales en el grafo residual son iguales a las capacidades originales.

Paso 2: Búsqueda de Caminos Aumentantes

- Utilizamos una búsqueda en amplitud (BFS) para encontrar un camino desde el nodo origen hasta el nodo sumidero en el grafo residual.
- La búsqueda en amplitud nos ayuda a encontrar un camino que todavía tiene capacidad residual disponible en sus aristas.
- Si encontramos un camino desde el origen al sumidero, continuamos al Paso 3. De lo contrario, hemos encontrado el flujo máximo y terminamos.

Paso 3: Cálculo del Flujo Máximo en el Camino Aumentante

- A lo largo del camino aumentante que encontramos en el Paso 2, buscamos la capacidad residual mínima. Esto se convierte en la cantidad máxima de flujo que podemos enviar a lo largo de ese camino.
- Actualizamos el flujo máximo acumulando esta cantidad en el flujo máximo total.

Paso 4: Actualización de las Capacidades Residuales

- Actualizamos las capacidades residuales en el grafo residual. Restamos el flujo calculado en el Paso 3 de las capacidades residuales a lo largo del camino aumentante.
- También, añadimos el flujo en sentido opuesto en el grafo residual para permitir que el flujo regrese si es necesario en iteraciones futuras.

Paso 5: Repetición

- Regresamos al Paso 2 y continuamos buscando más caminos aumentantes en el grafo residual.
- Repetimos los Pasos 2, 3 y 4 hasta que no se pueda encontrar ningún camino aumentante desde el origen al sumidero en el grafo residual.

Paso 6: Fin del Algoritmo

- Cuando no podemos encontrar más caminos aumentantes, hemos alcanzado el flujo máximo en la red.
- En este punto, el algoritmo se detiene, y el valor del flujo máximo calculado es la respuesta.

Paso 7: Opcional - Cálculo del Corte Mínimo

- Si es necesario, podemos realizar un recorrido adicional en el grafo residual para encontrar el corte mínimo.
- El corte mínimo representa las aristas que dividen el grafo en dos partes, de manera que cualquier camino desde el origen al sumidero atraviese el corte. Estas aristas tienen capacidad cero en el grafo residual.

```
# Inicializar la clase con el grafo dado.
def __init__(self, graph):
    self.graph = graph  # Grafo residual (capacidades residuales)
    self.ROW = len(graph)  # Número de filas en el grafo
    self.COL = len(graph[0])  # Número de columnas en el grafo

# Función que realiza una búsqueda en amplitud (BFS) para encontrar un camino aumentante desde el origen al sumidero.
def BFS(self, s, t, parent):
    Crear una lista 'visited' para rastrear los nodos visitados y establecer todos los elementos en False
    Crear una cola 'queue' para BFS
    Agregar el nodo origen 's' a la cola
    Marcar el nodo origen 's' como visitado en 'visited'

    Mientras la cola no esté vacía:
        Tomar el primer nodo 'u' de la cola
        Para cada vecino 'ind' y capacidad residual 'val' en la fila 'u' del grafo:
            Si el vecino 'ind' no ha sido visitado y la capacidad residual 'val' es mayor que 0:
                Agregar el vecino 'ind' a la cola
                Marcar el vecino 'ind' como visitado en 'visited'
                Almacenar 'u' como el padre del vecino 'ind' en 'parent'

    Devolver True si el nodo sumidero 't' ha sido visitado, de lo contrario, devolver False

# Función principal que implementa el algoritmo Ford-Fulkerson para encontrar el flujo máximo.
def FordFulkerson(self, source, sink):
    Crear una lista 'parent' para rastrear los padres de los nodos en el camino aumentante y establecer todos los elementos en -1
    Inicializar el flujo máximo 'max_flow' en 0
```

```
Mientras haya un camino aumentante desde el origen al sumidero utilizando BFS:
    Inicializar el flujo en el camino aumentante 'path_flow' como infinito
    Establecer 's' como el sumidero

    Mientras 's' no sea igual al origen:
        Buscar la capacidad residual mínima en el camino aumentante y almacenarla en 'path_flow'
        Actualizar 's' como el padre de 's' en 'parent'

    Establecer 'v' como el sumidero

    Mientras 'v' no sea igual al origen:
        Obtener el nodo padre 'u' de 'v' desde 'parent'
        Restar el flujo del camino aumentante 'path_flow' de la capacidad residual de la arista entre 'u' y 'v'
        Agregar el flujo en sentido opuesto 'path_flow' en el grafo residual entre 'v' y 'u'
        Actualizar 'v' como el padre de 'v' en 'parent'

    Acumular el flujo del camino aumentante al flujo máximo total 'max_flow'

Devolver 'max_flow'
```

Tiempo de ejecucion

Podemos identificar el tiempo de ejecución en función del tamaño de entrada del grafo, que está representado por la variable `self.ROW` (número de nodos).

Inicialización del grafo: La inicialización del grafo se realiza una vez al crear una instancia de la clase `Graph`. No tiene un impacto significativo en el tiempo de ejecución, ya que es una operación de asignación de listas y su complejidad es O (ROW * COL), donde ROW es el número de filas y COL es el número de columnas del grafo.

Búsqueda en Amplitud (BFS): La función `BFS` se utiliza para encontrar un camino aumentante desde el origen al sumidero, donde V es el número de nodos (self.ROW) y E es el número de aristas en el grafo residual. En el peor caso, BFS visita todos los nodos y aristas del grafo, por lo que su complejidad es lineal en función del tamaño del grafo.

Algoritmo Ford-Fulkerson: El algoritmo de Ford-Fulkerson utiliza BFS para encontrar caminos aumentantes y realiza operaciones en el grafo residual. En el peor caso, el algoritmo puede ejecutar BFS hasta que no se encuentren más caminos aumentantes. En cada ejecución de BFS, se actualizan las capacidades residuales en el grafo, lo que implica recorrer las aristas del camino aumentante. Ya que el número máximo de iteraciones es limitado por el flujo máximo, se puede decir que el tiempo de ejecución del algoritmo Ford-Fulkerson es O (f * (V + E)), donde 'f' es el flujo máximo y 'V' y 'E' son el número de nodos y aristas del grafo residual, respectivamente.

El tiempo de ejecución del algoritmo Ford-Fulkerson depende en gran medida del número de iteraciones necesarias para encontrar el flujo máximo. El peor caso ocurre cuando el algoritmo realiza muchas iteraciones de BFS, lo que

puede llevar a un tiempo de ejecución más largo en grafos con muchas aristas. El algoritmo Ford-Fulkerson se expresa como O (f * (V + E)), donde 'f' es el flujo máximo, 'V' es el número de nodos y 'E' es el número de aristas en el grafo residual.

Codigo (Resultados)

```python
from collections import defaultdict

# Definimos una clase llamada 'Graph' que representa un grafo con capacidades en las aristas.
class Graph:
    def __init__(self, graph):
        self.graph = graph  # Grafo residual (capacidades residuales)
        self.ROW = len(graph)  # Número de filas en el grafo
        self.COL = len(graph[0])  # Número de columnas en el grafo

    # Esta función realiza una búsqueda en amplitud (BFS) para encontrar un camino aumentante desde el origen al sumidero.
    def BFS(self, s, t, parent):
        visited = [False] * self.ROW  # Inicializamos una lista para rastrear los nodos visitados
        queue = []  # Creamos una cola para BFS
        queue.append(s)  # Comenzamos desde el nodo origen 's'
        visited[s] = True  # Marcamos el nodo origen como visitado
        while queue:
            u = queue.pop(0)  # Tomamos el primer nodo de la cola
            for ind, val in enumerate(self.graph[u]):
                # Verificamos si el nodo vecino no ha sido visitado y si la capacidad residual es mayor que cero
                if visited[ind] == False and val > 0:
                    queue.append(ind)  # Agregamos el nodo vecino a la cola
                    visited[ind] = True  # Marcamos el nodo vecino como visitado
                    parent[ind] = u  # Almacenamos el padre del nodo vecino en 'parent'
        # Devolvemos True si llegamos al nodo sumidero, de lo contrario, False
        return True if visited[t] else False

    # Función principal que implementa el algoritmo Ford-Fulkerson para encontrar el flujo máximo.
    def FordFulkerson(self, source, sink):
        parent = [-1] * self.ROW  # Inicializamos una lista para rastrear los padres de los nodos en el camino aumentante
        max_flow = 0  # Inicializamos el flujo máximo en cero
```

```python
        # Paso 2: Mientras haya un camino aumentante desde el origen al sumidero
        while self.BFS(source, sink, parent):
            path_flow = float("inf")  # Inicializamos el flujo en el camino aumentante como infinito
            s = sink

            # Paso 3: Encuentra el flujo máximo en el camino aumentante
            while s != source:
                # Buscamos la capacidad residual mínima en el camino aumentante
                path_flow = min(path_flow, self.graph[parent[s]][s])
                s = parent[s]

            # Paso 4: Actualiza las capacidades residuales en el grafo
            v = sink
            while v != source:
                u = parent[v]
                # Restamos el flujo del camino aumentante de la capacidad residual de la arista
                self.graph[u][v] -= path_flow
                # Añadimos el flujo en sentido opuesto en el grafo residual
                self.graph[v][u] += path_flow
                v = parent[v]

            # Acumulamos el flujo del camino aumentante al flujo máximo total
            max_flow += path_flow

        # Devolvemos el flujo máximo calculado
        return max_flow
```

```python
# Ejemplo de uso
graph = [[0, 16, 13, 0, 0, 0],
        [0, 0, 10, 12, 0, 0],
        [0, 4, 0, 0, 14, 0],
        [0, 0, 9, 0, 0, 20],
        [0, 0, 0, 7, 0, 4],
        [0, 0, 0, 0, 0, 0]]

g = Graph(graph)

source = 0
sink = 5

# Llamamos a FordFulkerson para calcular el flujo máximo
max_flow = g.FordFulkerson(source, sink)

print("Flujo Máximo:", max_flow)
```

```
Flujo Máximo: 23
```

Aplicaciones

El algoritmo de flujo máximo y corte mínimo tiene una amplia gama de aplicaciones en diversos campos.

Redes de Transporte y Logística:

Transporte de Bienes: El algoritmo se utiliza para optimizar la distribución de bienes desde fábricas a almacenes o tiendas minoristas, maximizando la cantidad de productos transportados mientras se respetan las restricciones de capacidad en carreteras, ferrocarriles o rutas marítimas.

Rutas de Transporte Público: Ayuda a optimizar los horarios y rutas de autobuses, trenes y otros medios de transporte público, asegurando que los pasajeros sean transportados de manera eficiente.

Redes de Comunicación:

Enrutamiento de Datos: En las redes de comunicación, el algoritmo puede utilizarse para determinar las rutas óptimas para el envío de datos, maximizando la capacidad de transmisión a través de los enlaces de red.

Planificación de Redes de Telecomunicaciones: Ayuda a diseñar redes de telecomunicaciones eficientes, incluyendo la asignación de capacidad de transmisión en cables de fibra óptica y la gestión de tráfico en redes de telefonía móvil.

Distribución de Recursos:

Distribución de Energía Eléctrica: Se aplica para optimizar la distribución de energía eléctrica a través de la red eléctrica, evitando sobrecargas y asegurando que la electricidad llegue a los consumidores de manera eficiente.

Distribución de Agua: Ayuda a planificar el suministro de agua a través de una red de tuberías, asegurando que todas las áreas reciban suficiente agua y evitando pérdidas.

Flujo de Tráfico y Transporte:

Gestión de Tráfico Urbano: Permite optimizar la sincronización de semáforos y controlar las luces de tráfico para reducir la congestión vehicular en áreas urbanas.

Diseño de Redes de Carreteras: Ayuda a planificar la expansión de redes de carreteras y autopistas para maximizar la capacidad de tráfico y minimizar los atascos.

Flujo de Información y Redes Sociales:

Flujo de Información en Redes Sociales: Se utiliza para modelar la difusión de información, noticias o tendencias en redes sociales, identificando nodos clave para la propagación eficiente.

Enrutamiento de Mensajes en Redes de Comunicación: Facilita la transmisión eficiente de mensajes en redes de comunicación, como redes de sensores inalámbricos.

Industria de la Manufactura:

Planificación de la Producción: Ayuda a optimizar la programación de la producción y la asignación de recursos en plantas de fabricación, maximizando la eficiencia de la cadena de suministro.

Problemas de Emparejamiento y Asignación: El algoritmo también se utiliza en problemas de emparejamiento, como el problema de asignación de trabajadores a tareas o el problema de asignación de estudiantes a escuelas, optimizando la asignación de recursos.

Ingeniería de Software: En el diseño de compiladores y optimización de código, se utiliza para optimizar el flujo de datos y la asignación de registros en un programa.

Biología y Genética: En el análisis de redes metabólicas y flujos de metabolitos, así como en la asignación de secuencias genéticas.

Conclusiones

El algoritmo Max Flow Min Cut es una herramienta extremadamente versátil que se utiliza en una amplia variedad de aplicaciones en campos que van desde la logística y el transporte hasta las redes de comunicación y la biología. Su capacidad para resolver problemas de flujo de manera eficiente lo convierte en una herramienta esencial en la optimización de redes y flujos.

Ford-Fulkerson: Es el algoritmo base que introduce el concepto de flujo residual y caminos aumentantes. Aunque es efectivo, puede no ser eficiente en casos donde las capacidades de los bordes no son enteros.

Edmonds-Karp: Es una mejora del algoritmo Ford-Fulkerson que utiliza BFS para encontrar caminos más cortos. Aunque garantiza la convergencia y funciona bien con capacidades enteras, puede no ser el más eficiente en redes densas debido a su complejidad temporal.

Dinic: Es otro algoritmo de flujo máximo que utiliza la idea de niveles en la red para acelerar la búsqueda de caminos. Es altamente eficiente y suele superar a Ford-Fulkerson y Edmonds-Karp en términos de velocidad, especialmente en redes dispersas.

La complejidad computacional es un factor importante a considerar al elegir un algoritmo de flujo máximo. Ford-Fulkerson tiene una complejidad exponencial en el peor caso, mientras que tanto Edmonds-Karp como Dinic tienen complejidades polinomiales. Por lo tanto, en la mayoría de los casos, se prefieren Edmonds-Karp o Dinic debido a su eficiencia.

Edmonds-Karp y Dinic garantizan la terminación debido a sus características específicas de búsqueda de caminos. En contraste, Ford-Fulkerson podría no converger si se utilizan capacidades no enteras.

Ford-Fulkerson permite capacidades fraccionarias en los bordes, lo que lo hace adecuado para problemas de flujo con valores no enteros. Sin embargo, esto puede complicar su convergencia.

La elección del algoritmo de flujo máximo depende de las características de la red y las restricciones del problema. En general, si se requiere un algoritmo confiable y eficiente, Edmonds-Karp o Dinic suelen ser opciones preferidas. Sin embargo, Ford-Fulkerson sigue siendo relevante en casos específicos.

El algoritmo Max Flow Min Cut es una herramienta esencial para la optimización de redes en una amplia gama de aplicaciones. La elección del algoritmo específico depende de la complejidad de la red y las necesidades del problema. Edmonds-Karp y Dinic son opciones confiables y eficientes en la mayoría de los casos, mientras que Ford-Fulkerson se utiliza en situaciones especiales donde se permiten capacidades fraccionarias o cuando se necesita una implementación personalizada.

Referencias

Datta, S. (2023, May 12). Minimum cut on a graph using a maximum flow algorithm. Baeldung. Retrieved from https://www.baeldung.com/cs/minimum-cut-graphs

Wayne, K. (2004). Max flow. Recuperado de https://www.cs.princeton.edu/courses/archive/spring06/cos226/lectures/maxflow.pdf

Chumbley, A., Vinegar, Z., Ross, E., y [otro autor]. (2023). Max-flow Min-cut Algorithm. Brilliant.org. Recuperado de https://brilliant.org/wiki/max-flow-min-cut-algorithm/

Punia, J. (2023). Maximum Flow and Minimum Cut. DSA Problem Solving for Interviews using Java. Scaler.com. Recuperado de https://www.scaler.com/topics/data-structures/maximum-flow-and-minimum-cut/

GeeksforGeeks. (2023). Find minimum s-t cut in a flow network. Recuperado de https://www.geeksforgeeks.org/minimum-cut-in-a-directed-graph/

Javed, M. S. (2023). What is the max-flow, min-cut theorem? Educative.io. Recuperado de https://www.educative.io/answers/what-is-the-max-flow-min-cut-theorem

VisualGo.net. (2023). Network Flow (Max Flow, Min Cut). Recuperado de https://visualgo.net/en/maxflow

Devadas, S. (2015). Lecture 13: Incremental Improvement: Max Flow, Min Cut. MIT OpenCourseWare. Recuperado de https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2015/resources/lecture-13-incremental-improvement-max-flow-min-cut/

Cheung, Y. M. (2023). The Maximum flow and the Minimum cut. Emory University. Recuperado de http://www.cs.emory.edu/~cheung/Courses/253/Syllabus/NetFlow/max-flow-min-cut.html

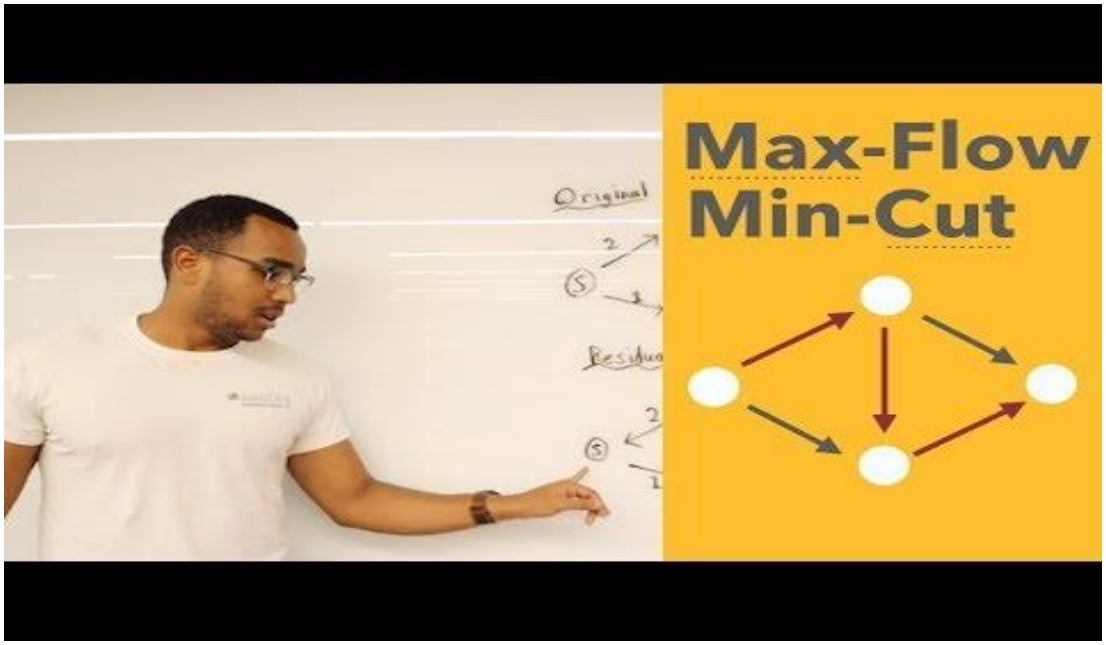
Moses, S. (2013). Slides 20 - Max-Flow Min-Cut. Recuperado de https://people.cs.umass.edu/~sheldon/teaching/mhc/cs312/2013sp/Slides/Slides20%20-%20Max-Flow%20Min-Cut.pdf

Moses, S. (2013). Slides 20 - Max-Flow Min-Cut. Recuperado de https://people.cs.umass.edu/~sheldon/teaching/mhc/cs312/2013sp/Slides/Slides20%20-%20Max-Flow%20Min-Cut.pdf

Ray, L. (2016, 16 de septiembre). Notes on Max Flow - Min Cut. Recuperado de http://www.cs.toronto.edu/~lalla/373s16/notes/MFMC.pdf

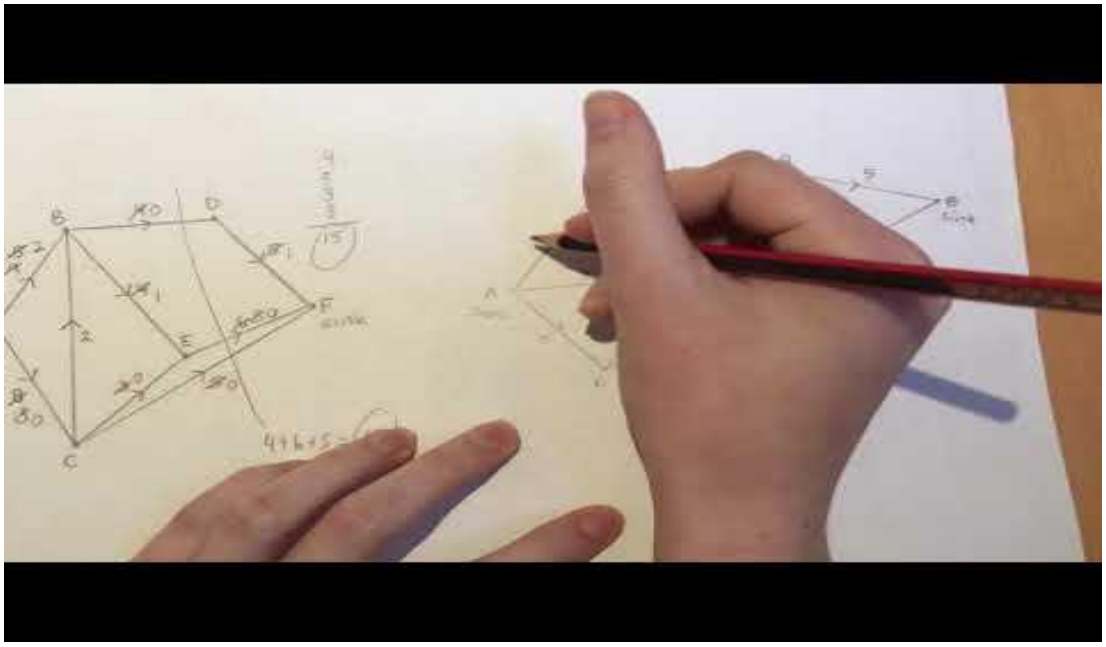
Serna, M., y Ortuño, J. (2016, 16 de septiembre). MaxFlow-fib: A Fast Algorithm for Computing the Maximum Flow in Directed Graphs. Recuperado de https://www.cs.upc.edu/~mjserna/docencia/grauA/P16/MaxFlow-fib.pdf

Luismi (2023, 16 de septiembre). Flujos de red: teorema de corte mínimo de flujo máximo (y algoritmo de Ford-Fulkerson). YouTube. Recuperado de Network Flows: Max-Flow Min-Cut Theorem (& Ford-Fulkerson Algorithm)



GT-Computabilidad, Complejidad, Teoría: Algoritmos (2023, 16 de septiembre). El teorema Max-Flow Min-Cut. YouTube. Recuperado de https://www.youtube.com/watch?v=I76yiYlnKxQ

Emily S (2023, 16 de septiembre). Caudal Máximo y Corte Mínimo de una Red. YouTube. Recuperado de Maximum Flow and Minimum Cut of a Network


B
D
F
E
C
115