FACULTAD DE INGENIERIA

Introducción a la Investigación de Operaciones

Evaluación a distancia 2020

Integrantes	Grupo 66
Francisco Casarotti	xxxxxxxx
Mathias Casarotti	xxxxxxxx
Nikolas Remlinger	xxxxxxxx
Micaella Toledo	xxxxxxxx

Índice

Introducción

En este informe se plantea un análisis detallado al artículo publicado en enero del 1992 por los autores Kwang Shin y Steve Corder del departamento de Sistemas de Información Informática de la Universidad estatal de Arkansas. El mismo plantea un método que utiliza el algoritmo de etiquetado Ford-Fulkerson para encontrar la solución al flujo máximo de un grafo con una menor cantidad de iteraciones y recorridos sin la necesidad de una rutina que consuma mucho tiempo para buscar rutas de aumento de flujo más cortas. Implementando dicho sistema en el algoritmo Edmond y Karp (creado por el matemático Jack Edmonds y el científico computacional Richard Karp, del cual hablaremos más adelante) se obtiene un algoritmo computacionalmente más eficiente que el de Ford Fulkerson.

Desarrollo

¿Por qué elegimos este artículo?

Considerando la competitividad y la globalización en el mercado contemporáneo, apuntar hacia la eficiencia es un factor fundamental a tener en cuenta. El artículo que seleccionamos se basa en un ajuste al algoritmo de Ford-Fulkerson que busca hacerlo más eficiente.

Nuestro grupo está constituido por dos estudiantes de Ingenieria en Computacion, uno de Ingenieria Civil y la cuarta integrante de Ingeniería Eléctrica. Teoría de grafos es un tema que está ampliamente relacionada con las 3 orientaciones, por lo que nos pareció pertinente abordarlo desde distintos ángulos.

Ford-Fulkerson tiene varias aplicaciones en dichas Ingenierías, algunos ejemplos de ellos pueden ser: en Ing. en Computación se puede utilizar al modelar redes de computadoras, en Ing. Civil para optimizar la construcción de carreteras y en Ing. Eléctrica al modelar redes eléctricas.

Relación con el Curso: Ford-Fulkerson

Para comenzar explicaremos brevemente el algoritmo propuesto por Ford Fulkerson, tema que se desarrolla en la materia Introducción a la Investigación de Operaciones.

El mismo propone buscar iterativamente caminos entre la fuente y el sumidero en los que se pueda aumentar el flujo, finalizando cuando no existan más de estos. Su nombre viene dado por sus creadores, L. R. Ford, Jr. y D. R. Fulkerson.

Este algoritmo no especifica cómo se recorre el grafo para hallar dicho camino, por lo que se podría utilizar DFS (Deep First Search) llevándonos a un algoritmo ineficiente en términos computacionales, ya que se podría encontrar un camino de aumento de flujo que no sea el más corto. Un camino más largo implica una mayor probabilidad de que el “cuello de botella” (arista del camino que permite el menor aumento de flujo) sea menor, de esta forma se tendría que en cada iteración el aumento de flujo posiblemente no sea el máximo disponible, lo que lleva a un algoritmo menos eficiente como se señaló previamente.

En el curso los caminos de aumento de flujo se construyen utilizando un algoritmo similar al de Dijkstra, basado en BFS (Breadth First Search), que encuentra el camino

mínimo entre dos nodos (en este caso fuente y sumidero) tomando en cuenta la variante de las capacidades y el flujos.

El artículo, al igual que en el curso, plantea hallar un camino mínimo utilizando un etiquetado de nodos, de izquierda a derecha y de arriba hacia abajo. Además, el recorrido para hallar dicho camino será óptimo cuando se realiza de izquierda a derecha y usando BFS al igual que en el algoritmo de Edmond-Karp.

Etiquetado y Recorrido : *"Fixed-Order Scanning"*

Es fundamental el cómo se recorren y etiquetan los nodos en el algoritmo. Para explicar mejor este procedimiento el artículo se apoya en la figura 1 como objeto de análisis. En esta red, hay un total de 32 caminos posibles desde el nodo fuente (nodo 1) hasta el nodo sumidero (nodo 8), incluidas las rutas que potencialmente implican flujos de arco inverso.

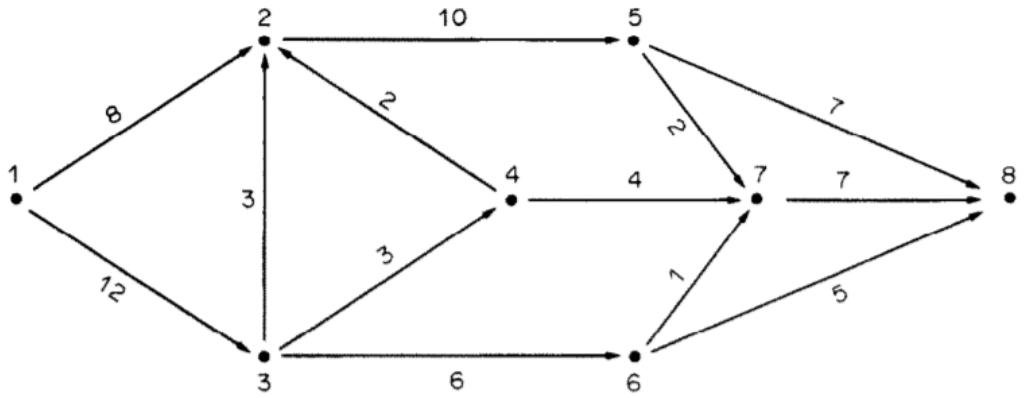


Figura 1 : 8 nodos, 13 arcos

Para la explicación del método se plantean tres casos donde se recorre la red de distinta forma y se estudia cual es el que tiene menor cantidad de iteraciones.

En el caso uno se plantea un etiquetado de nodos de izquierda a derecha y de arriba hacia abajo, y un recorrido de izquierda a derecha. En el caso dos se realiza el mismo etiquetado, pero se recorre el grafo de derecha a izquierda. Pasando al caso tres, se realiza un etiquetado de nodos de derecha a izquierda y de arriba hacia abajo (sin contar fuente y sumidero), y se recorre el grafo de izquierda a derecha al igual que en el caso uno.

Path		Flow Incr.
1.	(1,2), (2,5), (5,8)	7
2.	(1,3), (3,6), (6,8)	5
3.	(1,3), (3,4), (4,7), (7,8)	3
4.	(1,2), (2,5), (5,7), (7,8)	1
5.	(1,3), (3,6), (6,7), (7,8)	1
6.	(1,3), (3,2), (2,5), (5,7), (7,8)	1

Figura 2: caminos de aumento y flujo obtenido

En los tres casos se encuentran los mismos seis caminos de aumento de flujo como se muestra en la Figura dos (obteniendo un flujo máximo de dieciocho), la diferencia en los tres casos radica en la cantidad de iteraciones realizadas por el algoritmo para encontrar los caminos. En la figura tres se visualiza la cantidad de iteraciones realizadas para hallar cada uno de los caminos en cada caso, así como la cantidad total de iteraciones realizadas.

	Iteration						Total number of traversals
	1	2	3	4	5	6	
Case 1	1	1	1	1	1	2	7
Case 2	3	3	4	4	4	4	22
Case 3	2	2	3	3	3	3	16

Figura 3: iteraciones por camino

Claramente, el primer caso (escaneo en orden fijo) sería el de mayor eficiencia, eligiendo el camino más corto posible en la menor cantidad de iteraciones.

Algoritmo: Implementación

El artículo elegido presenta el algoritmo implementado en el lenguaje BASIC. A efectos de practicidad y mejorar la legibilidad del código, se optó por realizar una traducción del mismo al lenguaje C++, el cual se puede encontrar en el anexo. Cabe aclarar que el código original permite definir el grafo en tiempo real (incluyendo capacidad y flujos iniciales) lo cual no se realizó en el código de C++, sino que se cargó un grafo manualmente. A continuación se hace un breve análisis del mismo, citando breves fragmentos claves del código.

Se tomó de ejemplo el grafo de la Figura 1 para la implementación del mismo, como se puede apreciar en la Figura 4.

```
struct first_string{
    char signo;
    int nodo;
};
```

```
int i_node[13] = {0,0,1,2,2,2,3,3,4,4,5,5,6};
int j_node[13] = {1,2,4,1,3,5,1,6,6,7,6,7,7};
int capacidad[13] = {8,12,10,3,3,6,2,4,2,7,1,5,7};
int flow[13] = {0,0,0,0,0,0};
first_string first[8];
int second[8] = {100,0,0,0,0,0,0,0};
bool label[8] = {true,false,false,false,false,false,false,false};
```

Figura 4: implementacion del grafo de la figura 1

Para definir los arcos se definieron dos array, *i_node* y *j_node*, donde *i_node[k]* y *j_node[k]* representan el arco *i->j*, siendo *k* un natural menor al número de arcos, *i_node[k]* seria *i* y *j_node[k]* seria *j*. Como lo sugiere el nombre *capacidad[]* y *flow[]* representan las capacidades y flujos de cada arista. *first[], second[]* y *label[]* son utilizados para etiquetar los nodos guardando información sobre el origen del flujo, la cantidad del mismo y si han sido visitados previamente o no, respectivamente.

```
while(!scan_done){
    for(int i=0;i<num_arc;i++){ //recorre todos los arcos
        if(label[i_node[i]]&&first[j_node[i]].signo==' '&&(flow[i]<capacidad[i])){//si esta en el nodo i, y no paso por el nodo j, y tiene capacidad en el arco, voy
            first[j_node[i]].signo = '+';
            first[j_node[i]].nodo = i_node[i];
            //se manda todo lo que se puede, considerando el second del nodo i y el flujo aceptado por el arco
            if (capacidad[i] - flow[i] < second[i_node[i]])
                second[j_node[i]] = capacidad[i] - flow[i];
            else
                second[j_node[i]] = second[i_node[i]];
            label[j_node[i]] = true;//pase por j
            current_label++;//sumo un  nodo mas al recorrido

        } else if(!label[i_node[i]] && label[j_node[i]] && flow[i]>0) { //utilizo el grafo residual, en lugar de ir de i -> j, voy de j -> i y resto flow previo
            first[i_node[i]].signo = '-';
            first[i_node[i]].nodo = j_node[i];

            if(flow[i] < second[j_node[i]])//se manda todo lo que se pouede en el sentido contrario a la arista(j -> i)
                second[i_node[i]] = flow[i];
            else
                second[i_node[i]] = second[j_node[i]];
            label[i_node[i]] = true;
            current_label++;
        }
    }
    if(label[num_nodos-1] || current_label == prev_label)//si llegue al ultimo encontre un camino, por otro lado si no llegue al ultimo
        scan_done = true;                               //me fijo si desde el ulitmo escaneo recorrido algun nodo mas
    else
        prev_label = current_label;//actualizo para volver a recorrer el grafo
} // end while(!scan_done)
```

Figura 5: iteracion para hallar el siguiente camino

En la Figura 5 se aprecia el sector del codigo que se encarga de hallar el siguiente camino de aumento de flujo. Cada vez que se entra nuevamente al “while”, se vuelven a recorrer todos los arcos en el “for”, para salir del “while” hay dos casos: que se llegue al último nodo, o que no se llegue al último nodo y no se hayan recorrido nuevos nodos con respecto a la iteración anterior, esto significa que no se encontró un camino de aumento de flujo entre la fuente y el sumidero.

```
if(!label[num_nodos - 1])//sino llegue al ultimo, es que no encontre un camino para aumentar flujo
    path = false;//actualiza para salir del while principal
else {//actualiza los flows en funcion del path que encontro
    trace = num_nodos - 1;
    trace_node = first[trace].nodo;
    source = false;
    while(!source){//quiero llegar a la fuente
        for(int i = num_arc-1;i>=0;i--){// se actualizan los flujos, respetando las capacidades de las aristas que tiene el trace
            if(j_node[i] == trace && i_node[i] == trace_node && first[trace].signo == '+'){
                                //si el arista llega a t, y si ese arista sale nodo que le envio flujo a
                                //t(es decir, que encontre el arista que buscaba), y etsa en +, es porque le esta llegando flujo
                flow[i]+=second[num_nodos - 1];
                trace = i_node[i];
                trace_node = first[trace].nodo;
            } else if(i_node[i] == trace && j_node[i] == trace_node && first[trace].signo == '-'){
                flow[i]-=second[num_nodos - 1];
                trace = j_node[i];
                trace_node = first[trace].nodo;
            }
        }
        if (trace == 0)
            source = true;
    }
}
```

Figura 6: rastreo de camino

En la Figura 6 se puede apreciar un “if - else”. Si el programa entra al “if” significa que no se llego al ultimo nodo, por lo que no hay un camino de aumento de flujo, se setea *path = false,* se sale del while principal y se procede a calcular el flujo máximo como se ve en la Figura 7.

El loop *while(!source)* dentro del “else” de la Figura 7 se encarga de “rastrear” el camino entre el sumidero y la fuente, además de actualizar los flujos de cada arista perteneciente al camino.

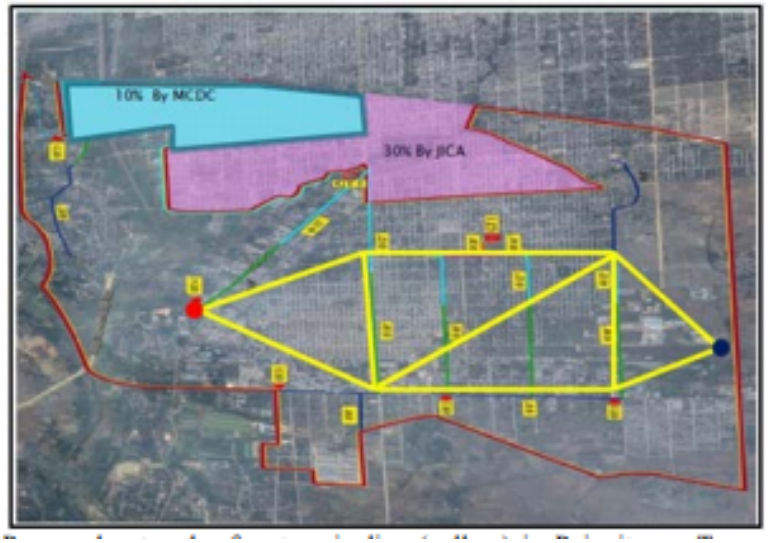
```
121     for(int i = 0;i<num_arc;i++){
122         if(label[i_node[i]] && !label[j_node[i]])
123             max_flow+=flow[i];
124     }
```

Figura 7: flujo maximo

Situación de la vida real

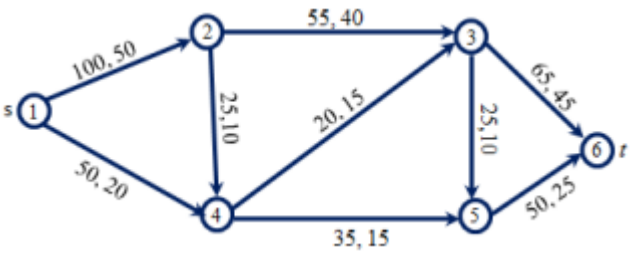
Investigando sobre implementaciones del método nos encontramos con múltiples aplicaciones sorpresivas, desde control de tráfico y sistemas de producción, hasta la eliminación de equipos en fase de juegos de cricket (adjunto en Artículo I referencias) . Sin embargo el ejemplo tal vez más frecuente de exponer es el de plan de distribución de aguas corrientes (flujo en tuberías).

Apoyándonos en el ejemplo expuesto en el Artículo II (ver referencias) veamos la siguiente figura:



Red de tuberías de agua propuesta (amarillo).

Mediante el siguiente grafo mostraremos la conexión de tuberías desde *s= fuente* de suministro de agua corriente, hacia *t= grifo* de la vivienda. Cabe destacar que las tuberías no necesariamente han de ser iguales en tamaño, y por tanto cada una será representada con su respectivo flujo/capacidad.

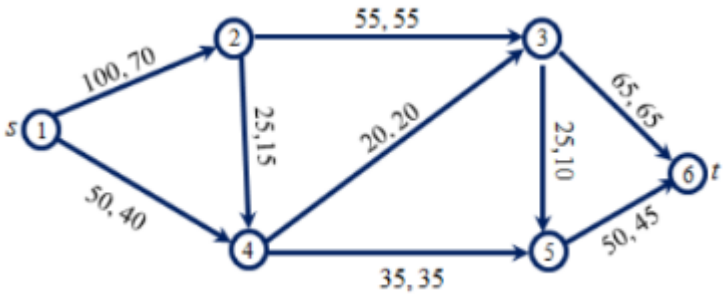


Representación en grafo de las tuberías propuestas. Nótese el flujo en lts/seg.

El algoritmo de Ford - Fulkerson se implementará para calcular el flujo máximo obtenible con la distribución planteada.

Para ello aplicando los pasos vistos en el estudio del algoritmo se comienza con el flujo inicial (dado) y se etiqueta al primer nodo con dicho flujo (s=1, 70), luego se continúa etiquetando los nodos y calculando sus flujos como se estudió en el análisis del algoritmo.

Se realizan varias iteraciones de re-etiquetado y cálculo como se prevé en el algoritmo hasta que finalmente no se encuentran rutas de aumento posibles, significando que obtuvimos el máximo.



Flujos máximos en la red de tuberías desde el nodo fuente hasta el grifo.

Obteniéndose que el flujo máximo será el flujo total fuera de la fuente (nodo 1) que es igual a flujo total en el grifo (nodo 6). En este caso,flujo total en s es 70 + 40 = 110 ,y en t es 65 + 45 = 110 por lo tanto, el máximo flujo posible para la red planteada resulta en 110 litros/seg.

Conclusiones

Se puede concluir que el algoritmo de Ford-Fulkerson es un método adecuado para averiguar cuánto es el flujo máximo de una red. Sin embargo en ciertas ocasiones éste presenta una desventaja práctica en cuanto a la cantidad de iteraciones necesarias para alcanzar este flujo máximo. Los autores del artículo plantean un ajuste a la hora de etiquetar los nodos y recorrer el grafo, reduciendo la cantidad de dichas iteraciones. El caso uno, en contraposición del dos y tres, es un claro ejemplo del mismo.

Otra de las ventajas que ofrece este ajuste es que el problema no se ve limitado por la cantidad de nodos adyacentes al nodo fuente y al nodo sumidero (hasta 9999), la capacidad de arco (hasta 999,999,999) tanto de valores enteros o reales y el flujo inicial que puede ser cero o distinto de cero. En casos de problemas de mayor número de nodos o capacidades basta para resolverlo algunas modificaciones menores.

Anexos

Código implementado en c++:

https://github.com/fcocasa/EdmondKarp/blob/master/EdmondKarp.cpp

Referencias

Shin, k., Corder, S., (1992). Implementing the Ford-Fulkerson labeling algorithm with fixed-order scanning. (Pergamon Press Ltd). Gran Bretaña.

Notas del curso. (2020). Introducción a la Investigación de Operaciones, Instituto de Computación, Universidad de la República, Uruguay.

Artículo I: Implementación en Cricket .
https://medium.com/swlh/real-world-network-flow-cricket-elimination-problem-55a3036a5d60

Artículo II: Implementación en red de tuberías.
researchgate.net/publication/330100794_Application_of_Ford-Fulkerson_Algorithm_to_Maximum_Flow_in_Water_Distribution_Pipeline_Network