Algoritmo de Ford-Fulkerson

Brayan Rodolfo Barajas Ochoa

Kevin Javier Lozano Galvis

Juan Camilo Londoño Jaimes

María Angélica Serrano Mora

Héctor Niño Quiñonez

Estructuras de Datos y Análisis de Algoritmos

Ingeniería de Sistemas

Facultad de Ingenierías Fisicomecánicas

Universidad Industrial de Santander

Bucaramanga, Colombia

2019

**Algoritmo de Ford-Fulkerson**

**Introducción**

**Lester Randolph Ford Jr.** al continuar los pasos de su padre Ford Sr. también hizo una enorme contribución al campo de las matemáticas. Su trabajo con **Delbert Ray Fulkerson** ha puesto la base de casi toda la investigación en flujos de grafos. El artículo de Ford y de Fulkerson (1956) con el problema de flujo máximo estableció el famoso teorema del flujo máximo - mínimo corte.

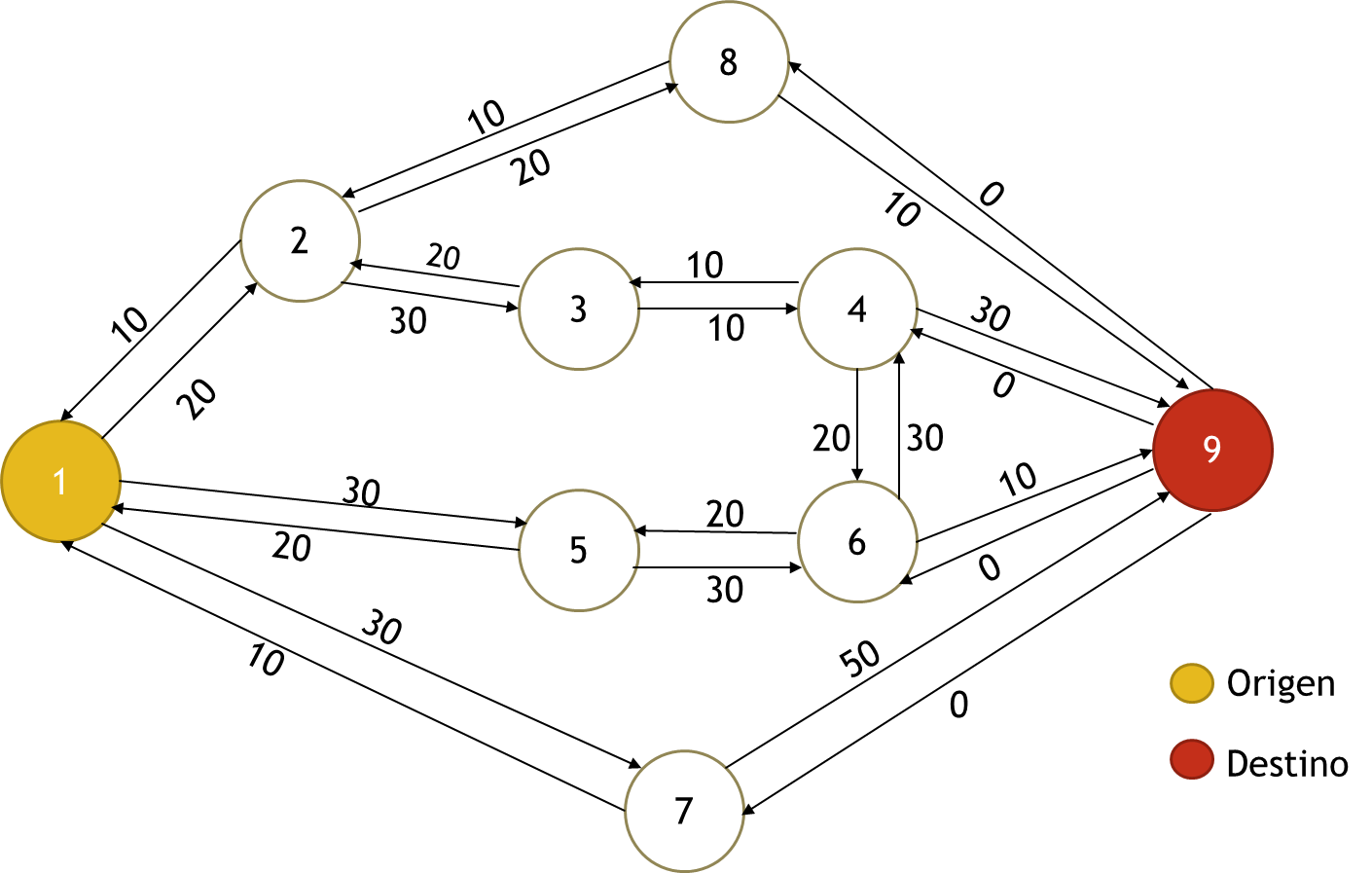
Se puede considerar un grafo como una red de flujo. Donde un nodo fuente produce o introduce en la red cierta cantidad de algún tipo de material, y un nodo sumidero lo consume. Cada arco, por tanto, puede considerarse como un conducto que tiene cierta capacidad de flujo.

El algoritmo de Ford-Fulkerson propone buscar [caminos](https://es.wikipedia.org/wiki/Camino_(teor%C3%ADa_de_grafos)) en los que se pueda aumentar el flujo, hasta que se alcance el flujo máximo. La idea es encontrar una ruta de penetración con un flujo positivo neto que una los nodos origen y destino.

**Tipo de grafo que utiliza**

El algoritmo de Ford-Fulkerson maneja grafos de tipo dirigidos y etiquetados.

En este caso, se trabajará con el siguiente grafo, en el cual se maneja como grafo doblemente dirigido para facilitar su explicación:



**Determinar El flujo máximo**

Teóricamente, para hallar el flujo máximo se debe partir de la siguiente ecuación; aplicada a cada recorrido:

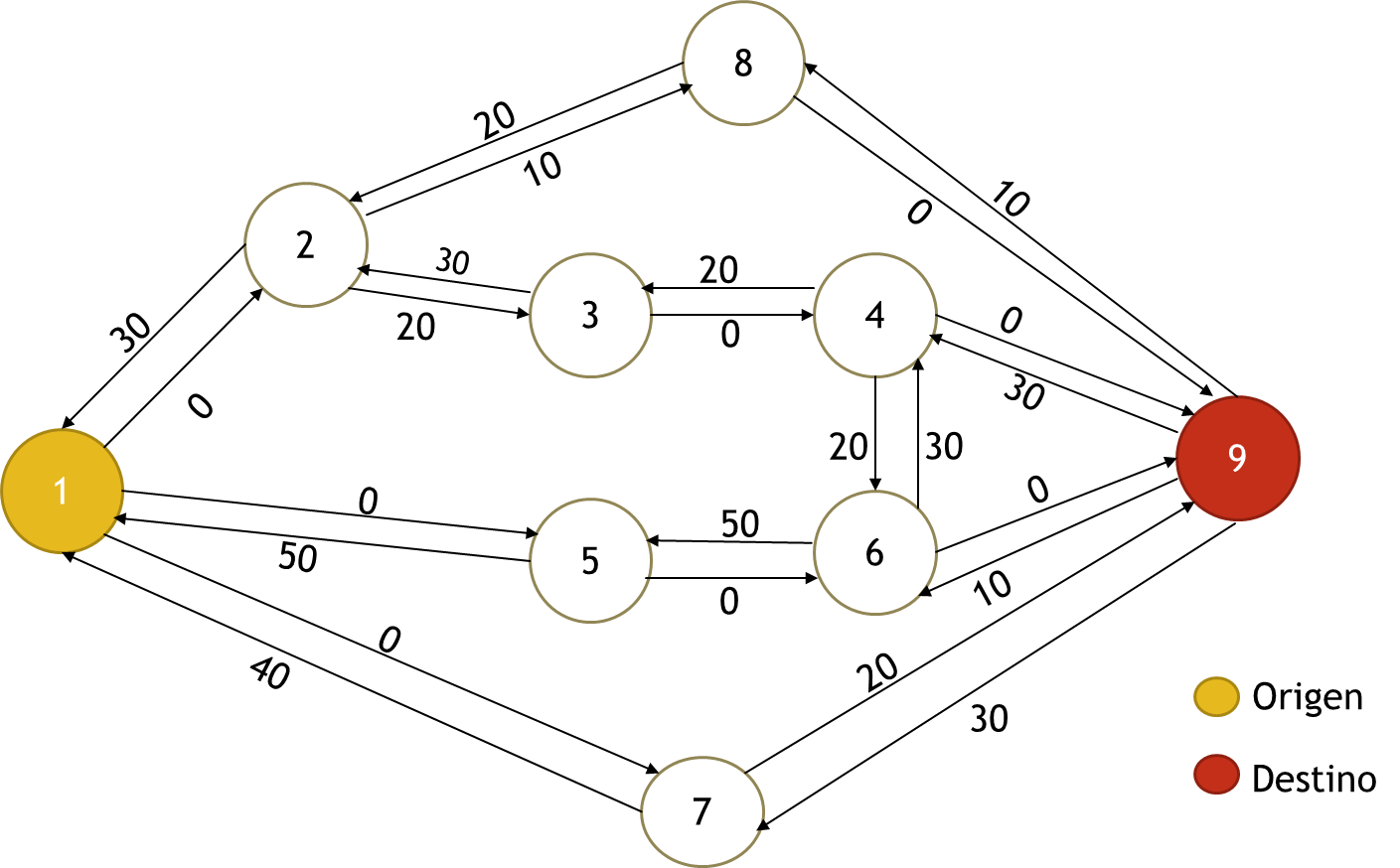
Donde:

C: capacidad

i, j: son los índices de los nodos

K: es el menor flujo que se traspasa del nodo origen al nodo destino

Al realizar el proceso para todo el grafo, el resultado es el siguiente:



**Estructura de datos utilizada en el algoritmo**

Este algoritmo se trabaja preferiblemente con matriz de adyacencia, para con esta realizar búsqueda por anchura, la cual utiliza la estructura de datos Fila, para obtener todos los posibles caminos desde el vértice de origen hasta el vértice de destino.

Grafo representado por matriz de adyacencia:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| **1** | 0 | 20 | 0 | 0 | 30 | 0 | 30 | 0 | 0 |
| **2** | 10 | 0 | 30 | 0 | 0 | 0 | 0 | 20 | 0 |
| **3** | 0 | 20 | 0 | 10 | 0 | 0 | 0 | 0 | 0 |
| **4** | 0 | 0 | 10 | 0 | 0 | 20 | 0 | 0 | 30 |
| **5** | 20 | 0 | 0 | 0 | 0 | 30 | 0 | 0 | 0 |
| **6** | 0 | 0 | 0 | 30 | 20 | 0 | 0 | 0 | 10 |
| **7** | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 50 |
| **8** | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 10 |
| **9** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Código**

#include "stdafx.h"

#include <iostream>

using namespace System;

using namespace std;

// Numero de vertices del grafo

#define V 9

struct nodo

{

int dato;

nodo \*sig;

};

class Fila {

private:

nodo \*frente;

nodo \*final;

public:

Fila();

char meter(int valor);

char sacar(int &valor);

~Fila();

void limpiarFila();

nodo\* frenteFila();

};

Fila::Fila() {

frente = NULL;

final = NULL;

}

Fila::~Fila() {

nodo \*apunt;

apunt = frente;

while (frente != NULL) {

frente = apunt->sig;

delete apunt;

apunt = frente;

}

}

char Fila::meter(int valor) {

nodo \*nuevo;

nuevo = new nodo;

if (nuevo == NULL) return 0;

else {

nuevo->dato = valor;

if (final == NULL) frente = nuevo;

else final->sig = nuevo;

nuevo->sig = NULL;

final = nuevo;

return 1;

}

}

char Fila::sacar(int &valor) {

nodo \*apunt;

apunt = frente;

if (apunt == NULL) return 0;

else {

valor = apunt->dato;

if (frente->sig == NULL) {

frente = NULL;

final = NULL;

}

else frente = apunt->sig;

delete apunt;

return 1;

}

}

void Fila::limpiarFila() {

nodo \*apunt;

apunt = frente;

while (frente != NULL) {

frente = apunt->sig;

delete apunt;

apunt = frente;

}

}

nodo\* Fila::frenteFila() {

return frente;

}

/\* Devuelve verdadero si hay un camino desde el origen ‘s’ hasta el destino ‘t’ en el grafo residual. Además, llena parent[] para guarder el camino \*/

bool bfs(int rGraph[V][V], int s, int t, int parent[])

{

// Se crea el vector de visitados, se marcan todos como no visitados

bool visited[V];

memset(visited, 0, sizeof(visited));

// Se crea una fila, se inserta el vértice de origen en la fila y se marca como   
// visitado

Fila f;

int valor, u;

f.meter(s);

visited[s] = true;

parent[s] = -1;

// Ciclo BFS

while (f.frenteFila() != NULL)

{

f.sacar(valor);

u = valor;

for (int v = 0; v < V; v++)

{

if (visited[v] != true && rGraph[u][v] > 0)

{

f.meter(v);

parent[v] = u;

visited[v] = true;

}

}

}

// Si se llega al destino, se devuelve true, de lo contrario flase

return (visited[t] == true);

}

// Retorna flujo máximo desde s hasta t del grafo dado

int fordFulkerson(int graph[V][V], int s, int t)

{

int u, v;

// Se crea un grafo residual que se llena con las capacidades iniciales del grafo

// original

int rGraph[V][V]; // Grafo residual donde rGraph[i][j] indica la capacidad residual

// de la arista de i a j si hay una arista entre ellos

for (u = 0; u < V; u++)

for (v = 0; v < V; v++)

rGraph[u][v] = graph[u][v];

int parent[V]; // Este vector se llena en el BFS para guardar el camino

int max\_flow = 0; // El flujo máximo se inicializa en cero

// Se entra al ciclo que aumenta el flujo desde que exista un camino de ‘s’ a ‘t’

while (bfs(rGraph, s, t, parent))

{

// Encontrar el flujo máximo a través del camino dado

int path\_flow = INT\_MAX;

for (v = t; v != s; v = parent[v])

{

u = parent[v];

if (path\_flow > rGraph[u][v]) {

path\_flow = rGraph[u][v];

}

}

cout << "Flujo del patron = " << path\_flow << endl << endl;

// Se actualizan las capacidades residuals de las aristas y las inversas

// en el patrón

for (v = t; v != s; v = parent[v])

{

u = parent[v];

rGraph[u][v] -= path\_flow;

rGraph[v][u] += path\_flow;

}

cout << "Grafo Residual modificado:" << endl;

for (u = 0; u < V; u++) {

for (v = 0; v < V; v++) {

cout << rGraph[u][v] << " ";

}

cout << endl;

}

cout << endl;

// Añadir el flujo del camino al flujo máximo total

max\_flow += path\_flow;

cout << "Flujo maximo = " << max\_flow << endl << endl;

}

cout << "El flujo maximo posible es: ";

// Retorna el flujo máximo

return max\_flow;

}

int main()

{

// Matriz de adyacencia del grafo de prueba

int graph[V][V] = {

{ 0, 20, 0, 0, 30, 0, 30, 0, 0 },

{ 10, 0, 30, 0, 0, 0, 0, 20, 0 },

{ 0, 20, 0, 10, 0, 0, 0, 0, 0 },

{ 0, 0, 10, 0, 0, 20, 0, 0, 30 },

{ 20, 0, 0, 0, 0, 30, 0, 0, 0 },

{ 0, 0, 0, 30, 20, 0, 0, 0, 10 },

{ 10, 0, 0, 0, 0, 0, 0, 0, 50 },

{ 0, 10, 0, 0, 0, 0, 0, 0, 10 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0 }

};

cout << fordFulkerson(graph, 0, 8) << endl;

system("pause");

return 0;

}

**¿Cómo funciona el algoritmo?**

El algoritmo empieza definiendo el grafo, junto con un grafo “auxiliar” que parte siendo el mismo que el original, a partir del cual se le empieza a realizar búsqueda por amplitud sucesivamente para así ir encontrando los caminos que permiten llegar desde el vértice de inicio al de llegada; Para cada uno de estos caminos se calcula la cantidad de flujo máximo que puede pasar a través de este el cual se suma a una variable que guarda el flujo máximo total. Posteriormente se aplica la fórmula enunciada anteriormente a cada peso del camino en un grafo “auxiliar”, de tal manera que permita identificar a dicho camino como ya recorrido.

De esta forma, se vuelve a realizar la búsqueda por amplitud, pero ahora sin tomar en cuenta el camino anteriormente recorrido y así sucesivamente hasta que ya no quede ninguno.

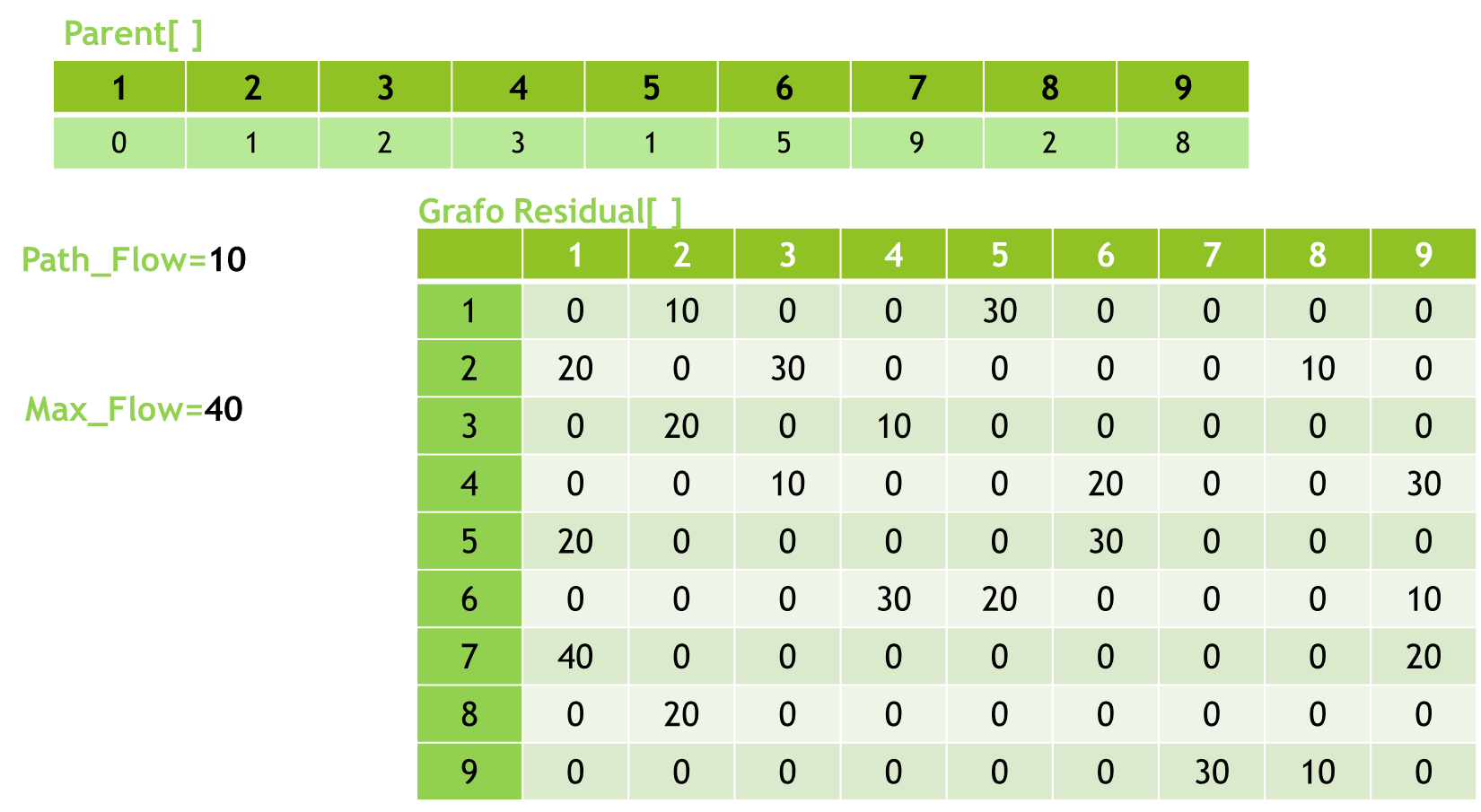
**Prueba de escritorio**

Haremos la prueba de escritorio, para el grafo mostrado anteriormente:

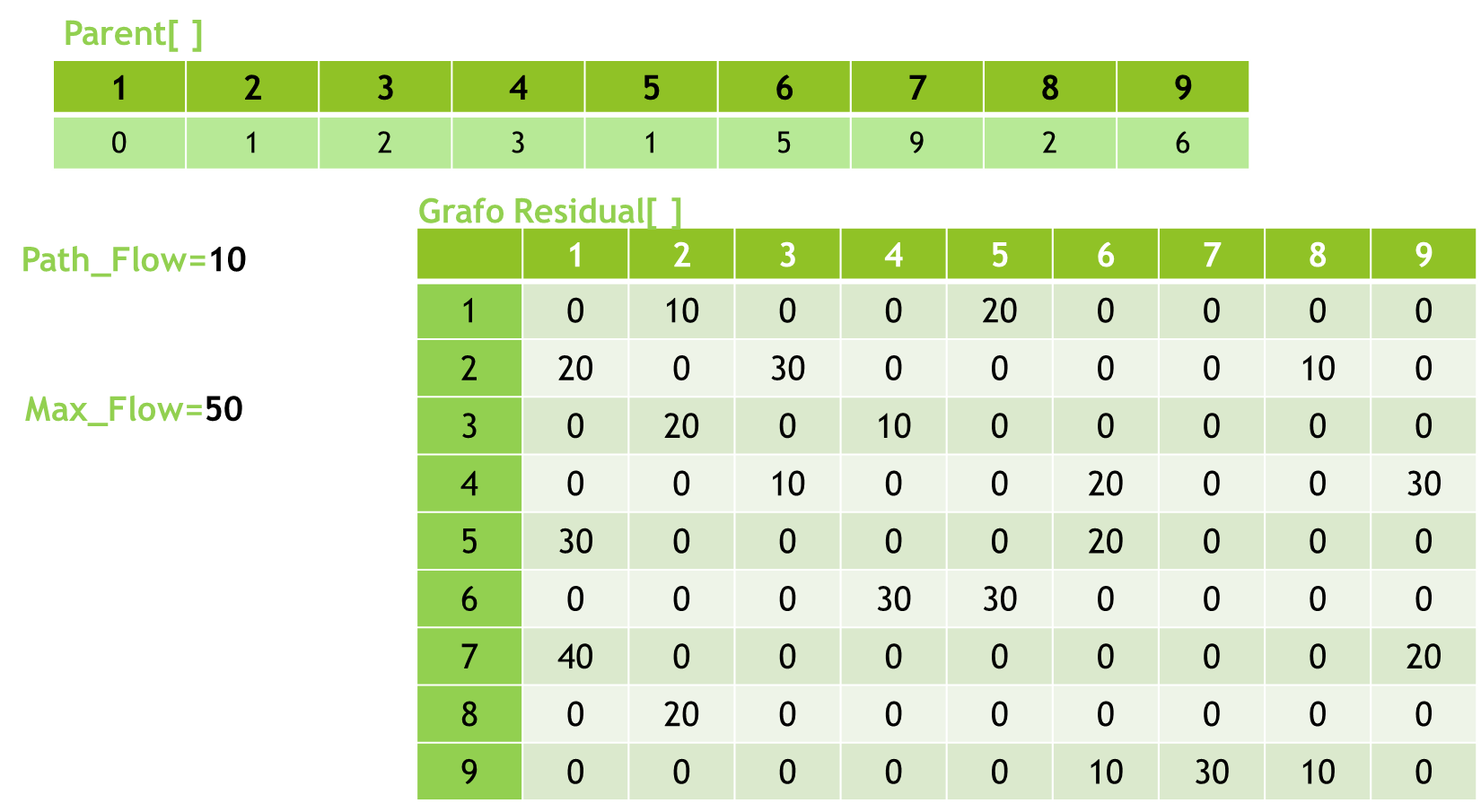
* ***Iteración 1:***



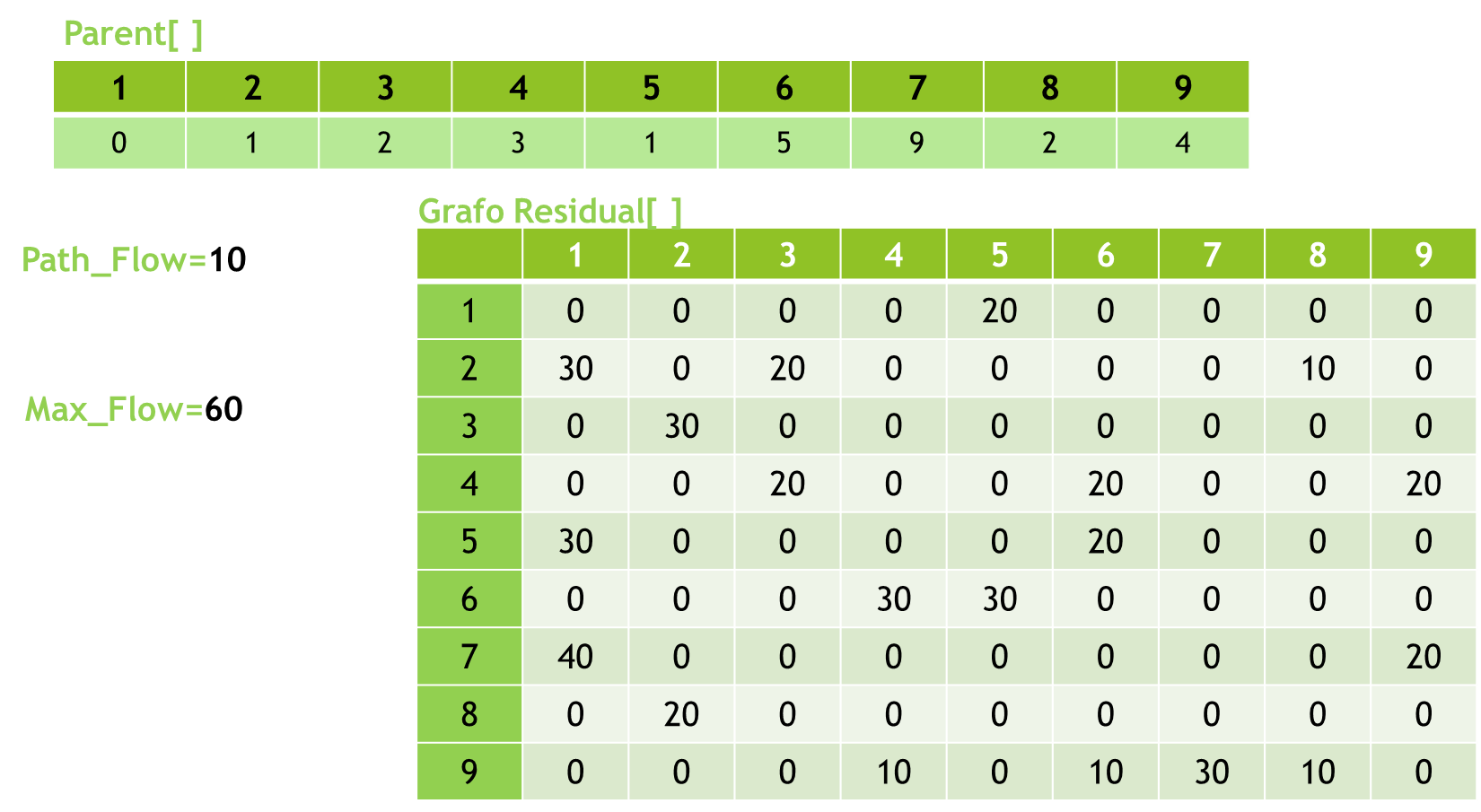
* ***Iteración 2:***

******

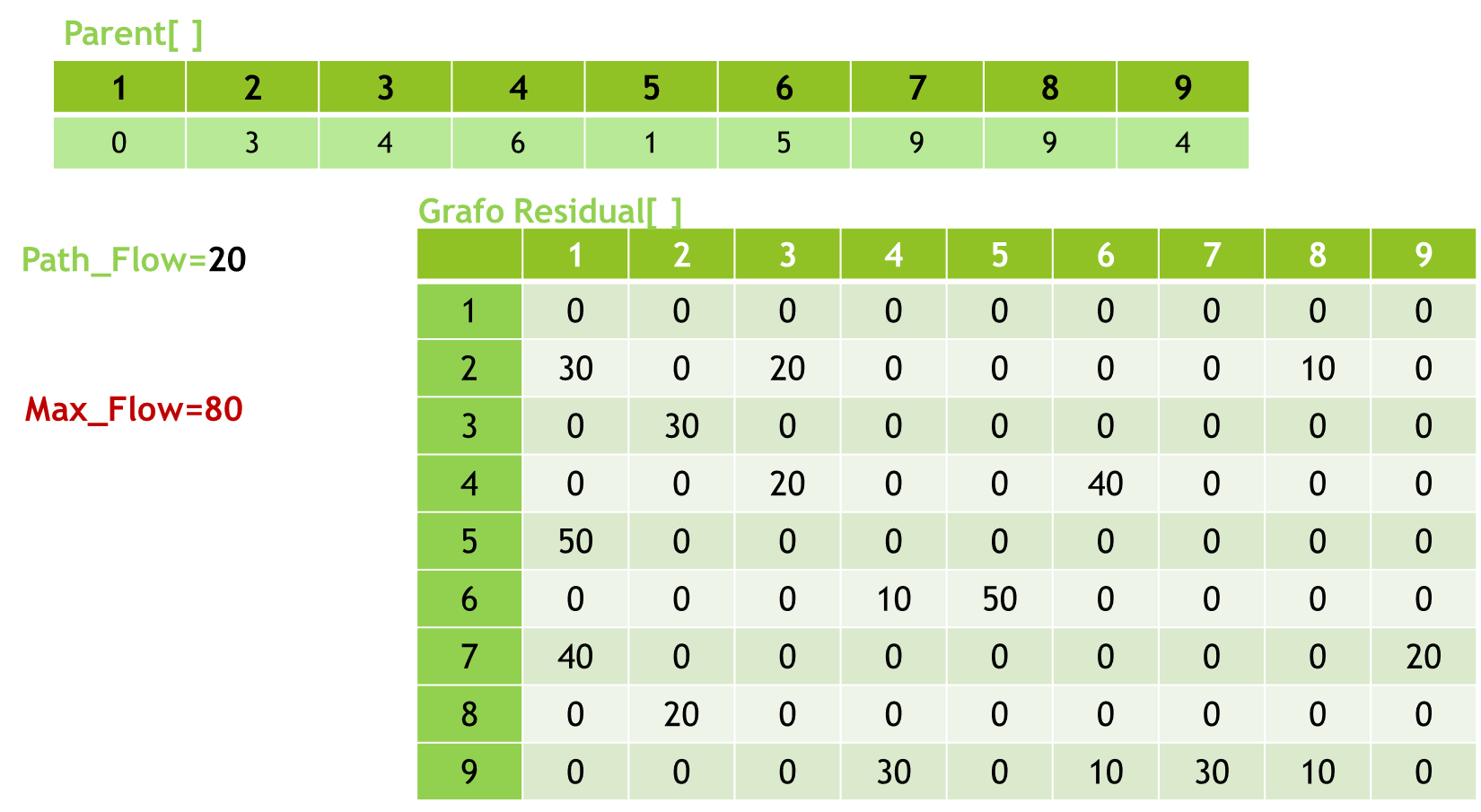
* ***Iteración 3:***

******

* ***Iteración 4:***



* ***Iteración 5:***

******

**Orden de complejidad**

El orden de complejidad es:

Donde:

V: número de vértices

f: flujo máximo

**Aplicaciones**

Este algoritmo puede ser utilizado para resolver modelos de:

* Transporte de mercancías (logística de aprovisionamiento y distribución)
* Flujo de gases y líquidos por tuberías
* Componentes o piezas en líneas de montaje
* Corriente en redes eléctricas
* Paquetes de información en redes de comunicaciones
* Tráfico ferroviario
* Sistema de regadíos
* Sistema de vías públicas

**Bibliografía**

* Algoritmo Ford-Fulkerson – Grafos - software para la construcción, edición y análisis de grafos. (2019). Recuperado de <http://arodrigu.webs.upv.es/grafos/doku.php?id=algoritmo_ford_fulkerson>
* Algoritmo de Ford-Fulkerson. (2019). Recuperado de <https://es.wikipedia.org/wiki/Algoritmo_de_Ford-Fulkerson>