UNIVERSIDAD DE GRANADA

Práctica 3
Algoritmos Voraces

Joaquín Sergio García Ibáñez
Juan Navarro Maldonado

Índice:

1.- Formalizar la descripción del funcionamiento del método

1.1.- ¿Se puede resolver mediante Greedy?

El método Greedy es una técnica que se emplea para la resolución de problemas de optimización, en la que se toman decisiones locales óptimas en cada paso con la esperanza de que el resultado global sea óptimo. Podemos verificar que nuestro problema se puede realizar mediante una resolución Greedy si cumple las siguientes condiciones:

1. **Subestructura óptima:** La solución óptima del problema se puede obtener a partir de la solución óptima de subproblemas más pequeños; es decir; en cada iteración, el algoritmo encontrará un camino óptimo entre dos nodos.
2. **Elección Greedy:** En cada paso del algoritmo, se toma la decisión que parece ser la mejor en cada momento, sin considerar las decisiones futuras.
3. **Propiedad de no arrepentimiento:** Después de tomar una decisión en cada paso, no se podría cambiar de opinión; es decir; nunca se volverá hacia atrás.

Por tanto podemos afirmar que es posible una solución mediante un método Greedy, ya que se puede encontrar una solución óptima.

1.2.- Diseño de las componentes greedy del algoritmo

La idea general del algoritmo es que debe de encontrar un camino euleriano en un grafo no dirigido.

1. **Lista de candidatos:** Los nodos con sus aristas del grafo inicial.
2. **Lista de candidatos utilizados:** Los nodos insertados en el camino solución.
3. **Criterio de selección:** Seleccionamos siempre la primera arista disponible, la cual va a proporcionarnos el camino solución.
4. **Criterio de factibilidad:** Una vez insertado el candidato, se eliminará dicha arista del grafo para evitar que se repita dicho camino.
5. **Función solución:** El algoritmo se detendrá siempre que regrese al nodo inicial.
6. **Función objetivo:** Encontrar un camino euleriano en un grafo no dirigido.

1.3.- Adaptación de la plantilla de diseño a las componentes propuestas

PROCEDIMIENTO euler(graph, circuit, v)
  MIENTRAS la lista de vecinos de v no esté vacía:
    SELECCIONAR el primer vecino w de v.
    ELIMINAR la arista (v,w) del grafo
    PARA i desde 0 hasta el tamaño de la lista de vecinos de w:
      SI el i-ésimo vecino de w es igual a v:
        ELIMINAR la arista (w,v) del grafo
        SALIR del ciclo para evitar eliminar múltiples aristas
      FIN SI
    FIN PARA
    Continúa recorrido desde w mediante llamada recursiva a euler(graph, circuit, w)
  FIN MIENTRAS
  AGREGAR v al circuito de Euler
FIN PROCEDIMIENTO

1.4.- Estudio de la optimalidad

Para demostrar que nuestra solución propuesta no es óptima debemos de encontrar un contraejemplo. De la forma en la que está planteado dicho problema, cualquier solución que implique cumplir todas las condiciones del camino de Euler (pasar por todos los nodos y aristas sin repetir estas últimas) siempre será una solución óptima sin importar el camino elegido.

2.- Implementación del algoritmo en C++

Para la implementación hemos decidido implementar el grafo como una matriz (vector<vector<int>> grafo) en la que definiremos cada fila como los posibles nodos y sus columnas las aristas que tiene cada nodo. Por ejemplo la inicializacion de un posible grafo seria:

```
vector<vector<int>> grafo = {{1,2}, {0,2,3,4}, {0,1,3,4}, {1,2,4,5}, {1,2,3,5}, {3,4}};
```

También para el diseño del algoritmo hemos decidido implementar una llamada recursiva para en la que en cada iteración, se seleccionará la arista más prometedora del grafo; es decir; por la que se va a construir el camino.

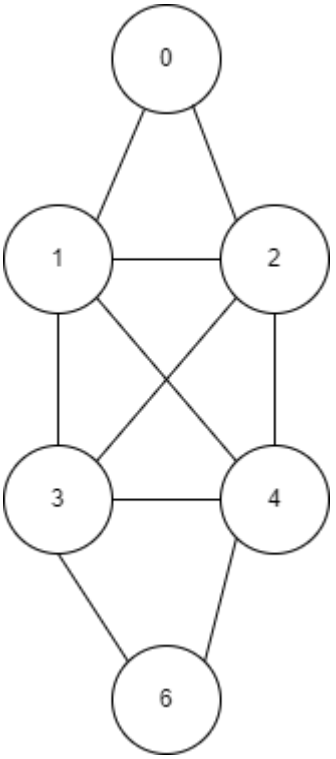
```
void fleury(vector<vector<int>>& grafo, vector<int>& camino,
int v) {
    while (!grafo[v].empty()) {
        int w = grafo[v][0];  // Selecciona el primer vecino
    de v
        grafo[v].erase(grafo[v].begin());  // Elimina la
    arista (v,w) del grafo
        for (int i = 0; i < grafo[w].size(); i++) {
            if (grafo[w][i] == v) {  // Elimina la arista
    (w,v) del grafo
                grafo[w].erase(grafo[w].begin() + i);
                break;
            }
        }
        fleury(grafo, camino, w);  // Continúa el
    recorrido desde w
    }
    camino.push_back(v);  // Agrega v al circuito de Euler
}
```

Por último también hemos decidido implementar una función que nos permite saber si un grafo es Euleriano o no antes de ejecutar el algoritmo.

```
bool esEuleriano(vector<vector<int>>& grafo) {
    for (int i = 0; i < grafo.size(); i++) {
        if (grafo[i].size() % 2 != 0) {
            return false;
        }
    }
    return true;
}
```

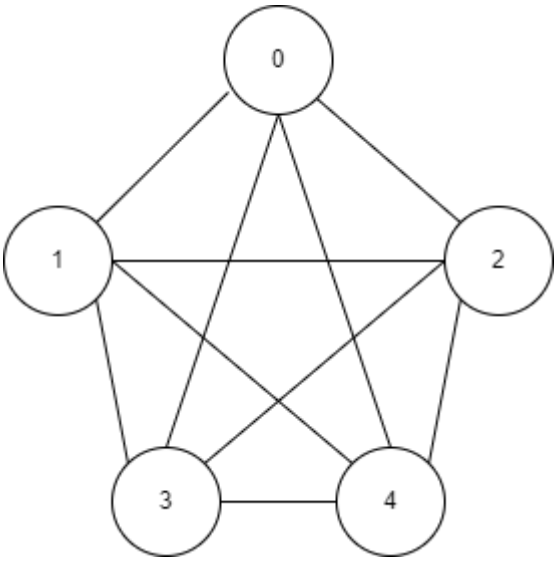
3.- Propuesta de los ejemplos de grafos

Grafo guión:



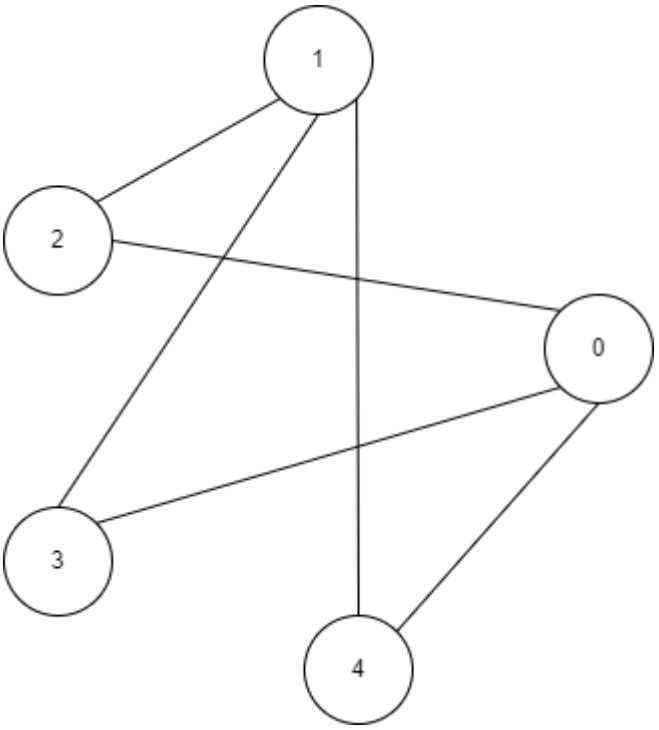

```
Implementacion Practica3
Seleccion del grafo:
        1.- Grafo prueba
        2.- Grafo 1
        3.- Grafo 2
        4.- Grafo no euleriano
1
El camino es:
0 2 4 5 3 4 1 3 2 1 0
```

Primer grafo propuesto:



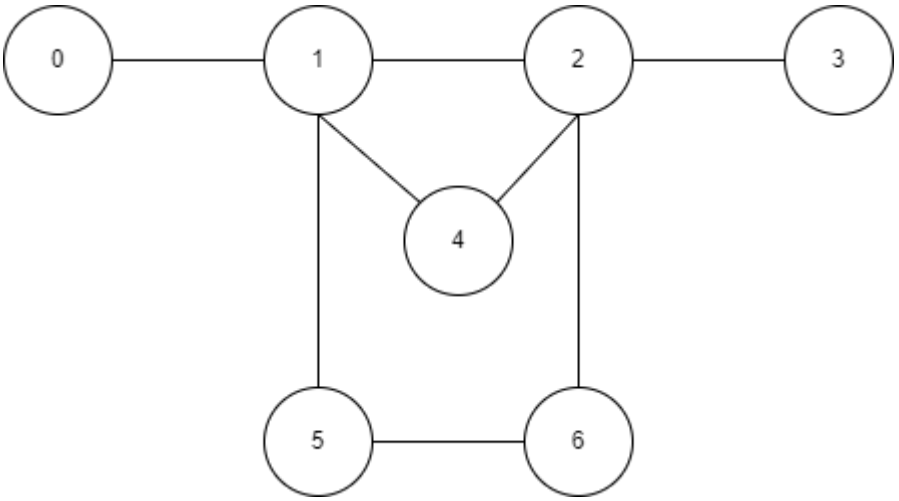

```
Implementacion Practica3
Seleccion del grafo:
        1.- Grafo prueba
        2.- Grafo 1
        3.- Grafo 2
        4.- Grafo no euleriano
2
El camino es:
0 4 3 2 4 1 3 0 2 1 0
```

Segundo grafo propuesto:



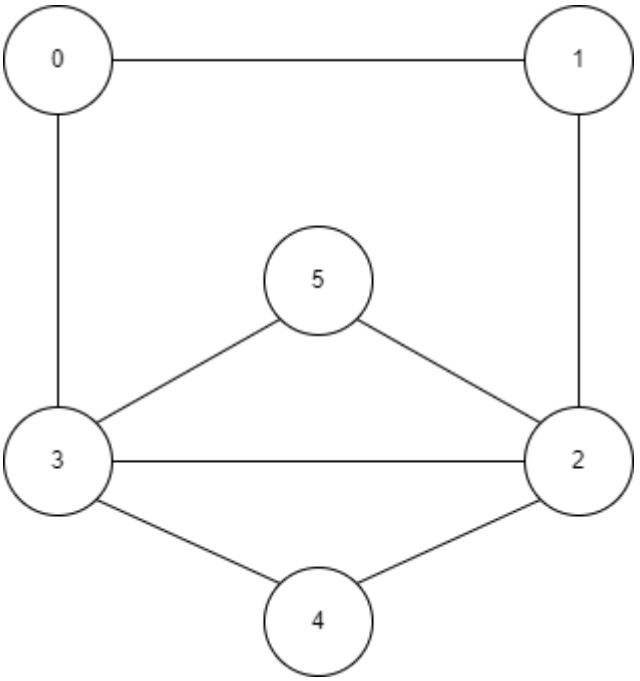

```
Implementacion Practica3
Seleccion del grafo:
        1.- Grafo prueba
        2.- Grafo 1
        3.- Grafo 2
        4.- Grafo no euleriano
3
El grafo dado no es euleriano
```

Tercer grafo propuesto:




```
Implementacion Practica3
Seleccion del grafo:
        1.- Grafo prueba
        2.- Grafo 1
        3.- Grafo 2
        4.- Grafo no euleriano
4
El grafo dado no es euleriano
```

Cuarto grafo propuesto:




```
Implementacion Practica3
Seleccion del grafo:
        1.- Grafo prueba
        2.- Grafo 1
        3.- Grafo 2
        4.- Grafo no euleriano
        5.- Grafo 3

5
El camino es:
0 3 5 2 4 3 2 1 0
```

4.- Cálculo de la eficiencia en el peor de los casos

Para calcular la eficiencia de este algoritmo podemos ver que está formado por un bucle while, y dentro de este hay unas sentencias de eficiencia $O(1)$, junto a un bucle for el cual se va a ejecutar n veces siendo n el número de aristas que tiene el nodo del grafo, por lo tanto usando la fórmula del bucle for tendrá eficiencia $O(n)$, más adelante fuera del bucle for pero dentro del bucle while hace una llamada recursiva de tamaño n-1 ya que anteriormente va a quitar una de las aristas del nodo, tendrá una ecuación de recurrencias de esta forma:

$T(n) = n + T(n-1)$ ECUACIÓN LINEAL NO HOMOGÉNEA

Parte Homogénea: $T(n) - T(n-1) = 0$; $pH(x) = x - 1$

Parte No Homogénea: $n = b1^{n} \cdot q1(n)$ entonces

$b1 = 1,\ q1(n) = n\ con\ grado\ d1 = 1$ Por lo tanto:

$p(x) = (x-1)\cdot(x-b1)^{d1} + 1 = (x-1)\cdot(x-1)^{2} = (x-1)^{3}$; $R1 = 1\ M1 = 3$

$t\,n = c10\ 1n^{n}n^{0} + c11\ 1^{n}n^{1} + c12\ 1^{n}n^{2}$

Por lo tanto el algoritmo tiene eficiencia: $O(n^{2})$

5.- Ejecución del código

Para compilar el código simplemente hace falta ejecutar:

g++ -o <nombre_ejecutable> practica3.cpp

Una vez se se compile y se ejecute el código, se abrirá un menú en el que podemos seleccionar el grafo sobre el que queremos usar el algoritmo.

Para probar cualquier otro grafo hay que tener en cuenta que a la hora de introducirlo como .txt tenemos que añadirlo de la siguiente manera:

- Cada nodo hay que ponerlo en una línea
- Al final de cada nodo escribir '-1'
- Las aristas se añaden escribiendo el nodo destino

Ejemplo para el grafo del guión:

```
1 2 -1
0 2 3 4 -1
0 1 3 4 -1
1 2 4 5 -1
1 2 3 5 -1
3 4 -1
```