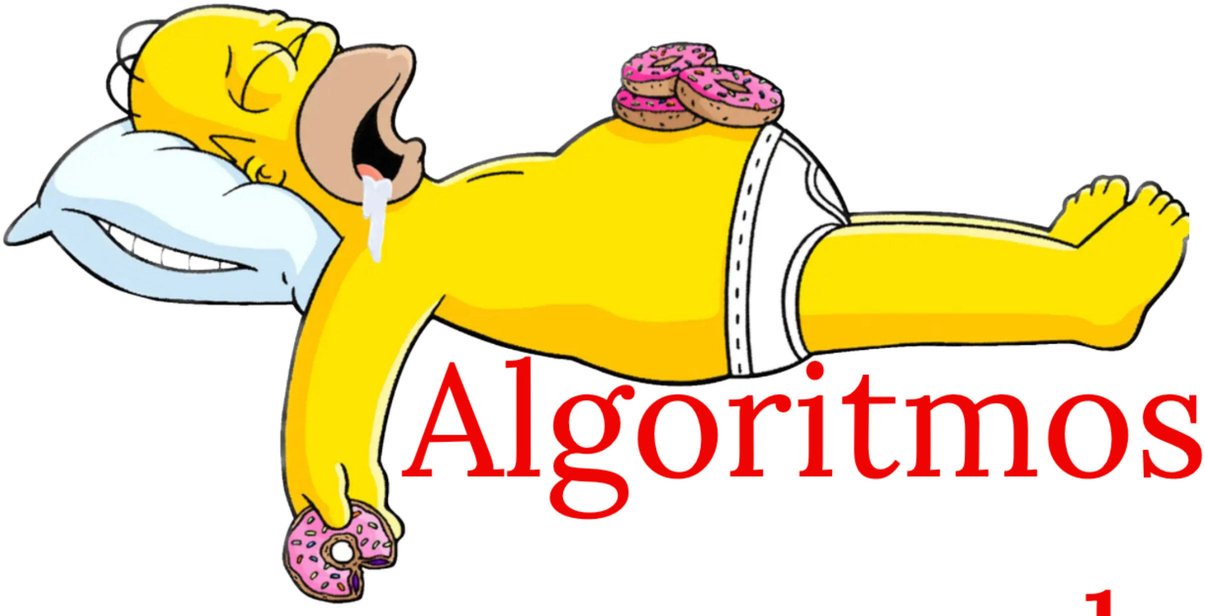
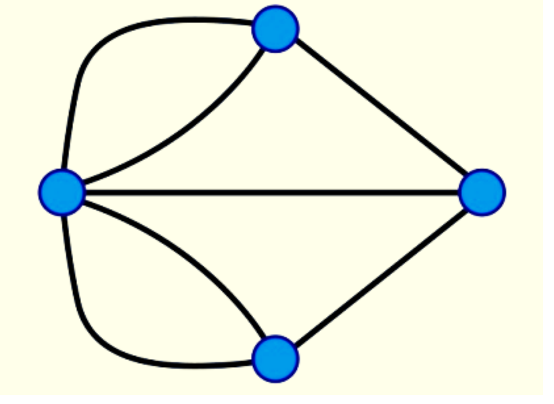


Algoritmos greedy

Mercedes León Chaves
Lydia vanDillewijn Soto
Laura Zafra Alarcos

Índice

Introducción
En esta práctica abordaremos un problema de Grafos, en el cual a partir de uno, pretenderemos obtener el camino por el cual podemos recorrer todos los nodos de estos sin repetir ninguna arista en este.



Metodología Greedy

- **¿Se puede resolver mediante Greedy?**

La idea básica de Greedy consiste en seleccionar en cada momento lo mejor entre un conjunto de candidatos, sin tener en cuenta lo ya hecho, hasta obtener una solución para el problema.

Para la resolución de un problema con el enfoque Greedy, ha de reunir las siguientes 6 características, que no son necesarias, pero si suficientes:

- 1.- <u>Un conjunto de candidatos:</u> los diferentes nodos del grafo.
- 2.- <u>Una lista de candidatos ya usados:</u> lista que se va definiendo según se vaya avanzando en la ejecución del algoritmo.
- 3.- <u>Un criterio (función) solución que dice cuando un conjunto de candidatos forma una solución (no necesariamente óptima):</u> cuando se llega a recorrer un camino que consigue pasar por todos los nodos del grafo.
- 4.- <u>Un criterio que dice cuándo un conjunto de candidatos (sin ser necesariamente una solución) es factible, es decir, podrá llegar a ser una solución (no necesariamente óptima):</u> cuando en el proceso de obtención del camino, no se llegue a repetir ninguna arista.
- 5.- <u>Una función de selección que indica en cualquier instante cuál es el candidato más prometedor de los no usados todavía:</u> el candidato más prometedor según vamos obteniendo el camino, será ir pasando por los nodos con un mayor grado, es decir, aquellos nodos en los que desembocan un mayor número de aristas.
- 6.- <u>Una función objetivo que a cada solución le asocia un valor, y que es la función que intentamos optimizar (a veces coincide con la de selección):</u> conseguir que se recorran todas las aristas.

Como podemos ver en nuestro caso, todas estas características se cumplen, por lo que se llega a la conclusión de que es un algoritmo correcto para este problema.

Pero según el problema explicado en la introducción del documento, el Matemático Leonhard Euler, desarrollo el siguiente teorema.

Teorema de Euler

-Si todos los vértices de un grafo son de grado impar, entonces no existen circuitos eulerianos.

-Si un grafo es conexo y todos sus vértices son de grado par, existe al menos un circuito euleriano.

Un circuito euleriano es un circuito que pasa por cada arista del grafo una y solo una vez.

Por lo que se puede plantear un algoritmo para resolver este problema, pero en el caso en el que todos los nodos sean impares, ese grafo no nunca podrá llegar a un camino.

- **Diseño de las componentes Greedy del algoritmo**

Componente Greedy	Descripción
G	Grafo
S	Solución del Problema
v	Nodo Seleccionado
w	Nodo siguiente
X	Elemento de C que Maximiza SELEC (X)
C	Lista de Candidatos
g	Grado de cada Nodo

- **Nueva plantilla de diseño Greedy**

Este sería el pseudocódigo planteado en la presentación de la práctica:

1. Se parte de un nodo dado v del grafo G.
2. Si G contiene sólo un nodo v, el algoritmo termina.
3. Si hay una única arista a que incide en v, entonces llamamos w al otro vértice que conecta la arista a, y la quitamos del grafo. Vamos después al paso 5. En otro caso, seguimos en el paso 4.
4. Como hay más de un lado que incide en v, elegimos uno de estos (lo llamamos w) de modo que al quitarlo del grafo G, el grafo siga siendo conexo. Cogemos la arista que une v con w y la quitamos del grafo.
5. Cambiamos el nodo v por el nodo w y volvemos al paso 3 hasta que terminemos de hacer el circuito de Euler.

FUNCIÓN GREEDY

```
S = Conjunto Vacío
Mientras S no sea una solución y C no esté vacío Hacer:
    X = elementos de C que maximiza SELEC(X)
    C = C - {X}
    Si (S U {X} es factible Entonces S = S U {X}
Si S es una solución entonces devolver S
caso contrario "NO HAY SOLUCIÓN"
```

Implementación

```
#include<iostream>

#include<queue>

#include <fstream>

using namespace std;
const int NODE = 8;
bool grafo[NODE][NODE];
bool temp_grafo[NODE][NODE];
```

```
/**
 * Lee desde un fichero de texto el grafo
 * @param fichero
 * @param grafo
 */
void readFromFile(const char fichero[], bool grafo[NODE][NODE]) {

  int nodo;
  ifstream fi;
  //Abro fichero
  fi.open(fichero);

  if (fi) { //si no hay errores
    cout << endl << "Leyendo el fichero " << fichero << endl;
    //Inicialización de la matriz
    for (int i = 0; i < NODE; i++) {
      for (int j = 0; j < NODE; j++) {
        fi >> nodo;
        if (nodo == 1) grafo[i][j] = true;
        else grafo[i][j] = false;
      }
    }
  } else {
    cout << "Error en la lectura del fichero" << endl;
  }

  //Cierro fichero
  fi.close();
}
```

```
/**
 * Función que muestra que nodo hay que coger primero
 * @return El nodo por el que se empieza el circuito
 */
int primer_vertice() {
  for (int i = 0; i < NODE; i++) {
    int g = 0;
    for (int j = 0; j < NODE; j++) {
      if (temp_grafo[i][j])
        g++; //encuentra el grado de cada nodo
    }
    if (g % 2 != 0) // cuando el grado de los vértices es impar
      return i; //i es un nodo de grado impar
  }
  return 0; //cuando todos los vertices tienen grado par, se empieza por el nodo 0
}
```

```
/**
 * Esta función comprueba si una arista entre el vértice u y el vértice v es un puente en el grafo.
 * Para hacer esto, se comprueba si el vértice v tiene más de una arista conectada.
 * @param u un nodo del grafo
 * @param v otro nodo del grafo
 * @return Si v tiene mñas de una arista conectada, la arista no es un puente y se devuelve false.
 * De lo contrario, la arista es un puente y se devuelve true.
 */
bool es_puente(int u, int v) {
  int g = 0; // grado del nodo
  for (int i = 0; i < NODE; i++)
    if (temp_grafo[v][i])
      g++;
  if (g > 1) {
    return false; //el nodo no forma un puente
  }
  return true; //el nodo forma un puente
}
```

```
/**
 * Esta función cuenta el número de aristas en el grafo.
 * Recorre todos los elementos de la matriz de adyacencia tempGraph
 * y cuenta el número de elementos no nulos.
 * @return Número de aristas en el grafo
 */
int numero_aristas() {
  int num = 0;
  for (int i = 0; i < NODE; i++)
    for (int j = i; j < NODE; j++)
      if (temp_grafo[i][j])
        num++;
  return num;
}
```

```
 * Esta función realiza el recorrido de Fleury en el grafo a partir del vértice
 * de partida inicio.
 * La función recorre todos los vértices del grafo y comprueba si hay una arista
 * que pueda ser eliminada sin formar un puente.
 * Si se encuentra una arista que cumple estas condiciones, se elimina y se avanza
 * al siguiente vértice.
 * Este proceso continúa hasta que no hay más aristas disponibles para eliminar.
 * Durante el proceso, la función imprime los vértices visitados para formar
 * el recorrido de Euler.
 * @param inicio el nodo en el que se empieza el recorrido
 **/
queue < int > fleury(int inicio) {
  queue < int > solucion;
  int arista = numero_aristas();
  int u = inicio;

  while (arista > 0) {
    bool hay_camino_euleriano = false;

    for (int v = 0; v < NODE; v++) {
      if (temp_grafo[u][v] && (!es_puente(u, v) || arista == 1)) {
        hay_camino_euleriano = true;
        cout << u << "--" << v << " ";
        solucion.push(u);
        solucion.push(v);
        temp_grafo[u][v] = temp_grafo[v][u] = 0;
        arista--;
        u = v;
        break;
      }
    }

    if (!hay_camino_euleriano) {
      for (int v = 0; v < NODE; v++) {
        if (temp_grafo[u][v]) {
          cout << u << "--" << v << " ";
          solucion.push(u);
          solucion.push(v);
          temp_grafo[u][v] = temp_grafo[v][u] = 0;
          arista--;
          u = v;
          break;
        }
      }
    }
  }
  return solucion;
}
```

```
int main() {
  queue < int > v; //CONJUNTO VACIO
  readFromFile("grafoA.txt", grafo);
  for (int i = 0; i < NODE; i++) //copiar el grafo principal al temp_grafo
    for (int j = 0; j < NODE; j++)
      temp_grafo[i][j] = grafo[i][j];
  cout << "El circuito de Euler sería: ";
  v = fleury(primer_vertice());

  cout << endl << "Cola: "; // Mostrar todos los elementos del CONJUNTO-solucion
  while (!v.empty()) {
    cout << v.front() << " -- ";
    v.pop();
  }
  cout << endl;

}
```

Grafos

Para implementar los grafos de modo que se puedan leer desde un archivo de texto, usaremos una notación en forma de matriz. En este enfoque, se usará un array bidimensional cuadrada, con tantas filas/columnas como nodos haya. El tipo de dato contenido serán bool, que indiquen si dos nodos son adyacentes. Por ejemplo, para el grafo no dirigido con tres nodos A, B y C, donde A está conectado a B y B está conectado a C, el archivo de texto sería de la forma:

```
0 1 0
1 0 1
0 1 0
```

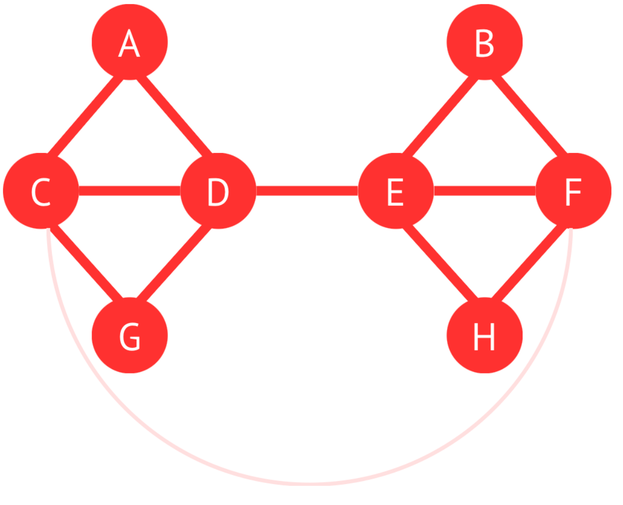
Donde la fila/columna 0 representa al nodo A, la 1 al B y la 2 al C. Como podemos ver, consideramos que un nodo no es adyacente a sí mismo.

Usaremos un array estático de dos dimensiones para almacenar el grafo, considerando que ya conocemos el número de nodos en el grafo:

```
const int NODE = 3;
```

```
bool grafo[NODE][NODE];
```

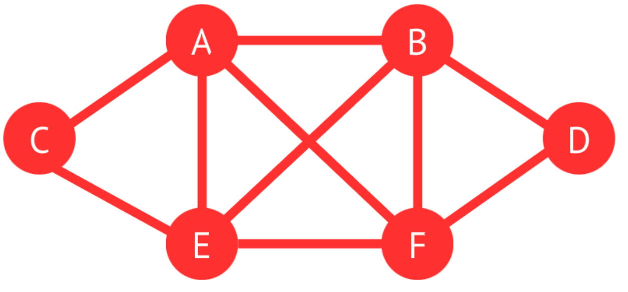
A continuación podemos ver los grafos que hemos tomado como ejemplo y sus respectivas representaciones en archivos de texto:



Representación gráfica del primer grafo de ejemplo

```
0 0 1 1 0 0 0 0
0 0 0 0 1 1 0 0
1 0 0 1 0 0 1 0
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
0 1 0 0 1 0 1 0
0 0 1 1 0 0 0 0
0 0 0 0 1 1 0 0
```

grafoA.txt - Contenido del archivo de texto referente al primer grafo de ejemplo



Representación gráfica del segundo grafo de ejemplo

```
0 1 1 0 1 1
1 0 0 1 1 1
1 0 0 0 1 0
0 1 0 0 0 1
1 1 1 0 0 1
1 1 0 1 1 0
```

grafoB.txt - Contenido del archivo de texto referente al segundo grafo de ejemplo

Ejecución

```
El circuito de Euler sería: 2--0 0--3 3--2 2--6 6--3 3--4 4--1 1--5 5--4 4--7 7--5
Cola: 2 -- 0 -- 0 -- 3 -- 3 -- 2 -- 2 -- 6 -- 6 -- 3 -- 3 -- 4 -- 4 -- 1 -- 1 -- 5 -- 5 -- 4 -- 4 -- 7 -- 7 -- 5 --

Process finished with exit code 0
```

Ejecución con grafoA

```
El circuito de Euler sería: 0--1 1--3 3--5 5--0 0--2 2--4 4--1 1--5 5--4 4--0
Cola: 0 -- 1 -- 1 -- 3 -- 3 -- 5 -- 5 -- 0 -- 0 -- 2 -- 2 -- 4 -- 4 -- 1 -- 1 -- 5 -- 5 -- 4 -- 4 -- 0

Process finished with exit code 0
```

Ejecución con grafoB

Donde 0 representa al nodo A, 1 representa al nodo B, 2 representa al nodo C...

Observación:

Si todos los vértices tienen un grado par, el recorrido puede comenzar en cualquier vértice, aunque en nuestro código lo comenzaremos con el primer nodo. De lo contrario, se debe comenzar en un vértice con grado impar para asegurar que el recorrido termine en otro vértice con grado impar, y al recorrer otros nodos no obstaculicemos el recorrido.

En los ejemplos:

- En el Grafo A, este tiene 8 nodos, los cuales todos ellos tienen grado par excepto dos. Para que el algoritmo pueda funcionar correctamente, si no todos los nodos de un grafo son pares, el nodo inicial debe ser un impar, para poder recorrer todo el grafo sin borrar alguna arista importante antes de quitar otra arista.
- En el Grafo B, este tienen seis nodos, los cuales todos ellos son pares, por lo que se puede empezar el recorrido por el nodo 0.

Eficiencia

- Para la función 'primer_vertice':

```
/**
 * Función que muestra que nodo hay que coger primero
 * @return El nodo por el que se empieza el circuito
 */
int primer_vertice(){
    for(int i = 0; i<NODE; i++){
        int g = 0;
        for(int j = 0; j<NODE; j++){
            if(temp_grafo[i][j])
                g++; //encuentra el grado de cada nodo   o(1)   o(1)   o(n)   o(n^2)   o(n^2)
        }
        if(g % 2 != 0) // cuando el grado de los vértices es impar   o(1)
            return i; //i es un nodo de grado impar   o(1)
    }
    return 0; //cuando todos los vertices tienen grado par, se empieza por el nodo 0   o(1)
}
```

- Para la función 'es_puente':

```
/**
 * Esta función comprueba si una arista entre el vértice u y el vértice v es un puente en el grafo.
 * Para hacer esto, se comprueba si el vértice v tiene más de una arista conectada.
 * @param u un nodo del grafo
 * @param v otro nodo del grafo
 * @return Si v tiene mas de una arista conectada, la arista no es un puente y se devuelve false. De lo contrario, la arista es un puente y se devuelve true.
 */
bool es_puente(int u, int v){
    int g = 0; // grado del nodo   o(1)
    for(int i = 0; i<NODE; i++)
        if(temp_grafo[v][i])   o(1)   o(n)
            g++;   o(1)
    if(g>1){
        return false; //el nodo no forma un puente   o(1)   o(1)   o(n)
    }
    return true; //el nodo forma un puente   o(1)
}
```

- Para la función 'numero_aristas':

```
/**
 * Esta función cuenta el número de aristas en el grafo.
 * Recorre todos los elementos de la matriz de adyacencia tempGraph y cuenta el número de elementos no nulos.
 * @return Número de aristas en el grafo
 */
int numero_aristas(){
    int num = 0;   o(1)
    for(int i = 0; i<NODE; i++)
        for(int j = i; j<NODE; j++)
            if(temp_grafo[i][j])   o(1)   o(n)   o(n^2)   o(n^2)
                num++;   o(1)
    return num;   o(1)
}
```

- Para la función 'fleury':

```
 * Esta función realiza el recorrido de Fleury en el grafo a partir del vértice de partida inicio.
 * La función recorre todos los vértices del grafo y comprueba si hay una arista que pueda ser eliminada sin formar un puente.
 * Si se encuentra una arista que cumple estas condiciones, se elimina y se avanza al siguiente vértice.
 * Este proceso continúa hasta que no hay más aristas disponibles para eliminar.
 * Durante el proceso, la función imprime los vértices visitados para formar el recorrido de Euler.
 * @param inicio el nodo en el que se empieza el recorrido
**/
queue<int> fleury(int inicio) {
    queue<int> solucion;              o(1)
    int arista = numero_aristas();    o(n^2)
    int u = inicio;                   o(1)

    while (arista > 0) {                                                        o(n^2)
        bool hay_camino_euleriano = false;   o(1)

        for (int v = 0; v < NODE; v++) {
            if (temp_grafo[u][v] && (!es_puente(u, v) || arista == 1)) {
                hay_camino_euleriano = true;   o(1)
                cout << u << "--" << v << " ";
                solucion.push(u);
                solucion.push(v);
                temp_grafo[u][v] = temp_grafo[v][u] = 0;      o(1)   o(n)   o(n^2)
                arista--;
                u = v;
                break;
            }
        }

        if (!hay_camino_euleriano) {                                            o(n^2)
            for (int v = 0; v < NODE; v++) {
                if (temp_grafo[u][v]) {
                    cout << u << "--" << v << " ";  o(1)
                    solucion.push(u);                                      o(n^2)
                    solucion.push(v);                                 o(n)
                    temp_grafo[u][v] = temp_grafo[v][u] = 0;   o(1)   o(n)
                    arista--;
                    u = v;
                    break;
                }
            }
        }
    }
    return solucion;   o(1)
}
```

Instrucciones de compilación y ejecución

Para realizar la ejecución del código, tenemos el código en un .cpp y para la representación del grafo se han creado dos .txt de entrada de fichero para la representación de estos y su lectura mediante la función readFromFile. El código se ha compilado y ejecutado directamente desde un IDE (en nuestro caso, CLion).

Es necesario que el fichero de texto indicando las conexiones entre los nodos y el ejecutable se encuentren en el mismo directorio en el momento de ejecución, o bien dejarse indicada la ruta donde se encuentra el fichero de texto como parámetro en la llamada a la función de lectura.

Cada vez que se ejecuta el programa con un grafo diferente, es necesario cambiar dos aspectos en el código y recopilarlo:

1. nombre o ruta del fichero de texto en la llamada a readFromFile
2. número de nodos que conforman el grafo en la declaración de la constante

```
readFromFile("nombrefichero.txt", grafo);
```

```
const int NODE = 8;
```