UNIVERSIDAD DE GRANADA

Algorítmica

2º Grado en Ingeniería Informática Y ADE

PRÁCTICA 3: Algoritmo Greedy

ÍNDICE

INTRODUCCIÓN

En esta práctica se nos plantea el clásico problema de los puentes de Königsberg, que consiste en encontrar un camino que cruce cada uno de los siete puentes de la ciudad una sola vez y regrese al punto de partida. Sin embargo, este camino es imposible de encontrar.

Para resolver problemas similares, se utiliza la teoría de grafos y se busca un camino de Euler, es decir, un camino que recorre cada arista del grafo exactamente una vez.

Tienen que cumplirse dos condiciones: que sea un **grafo conexo**, y que en cada uno de sus nodos tenga un **número par de aristas**

Para encontrar el camino de Euler se utiliza el algoritmo de Greedy que es un algoritmo de búsqueda que selecciona la mejor opción en cada paso con la esperanza de encontrar una solución óptima global.

Posteriormente, vamos a implementar el algoritmo en un programa en C++, y ejecutaremos dos ejemplos de grafos de Euler para demostrar su funcionamiento.

Finalmente, calcularemos su eficiencia teórica y daremos una conclusión final.

EJERCICIO 1

(Formalizarlo, si es posible, como un algoritmo Greedy)

¿SE PUEDE RESOLVER CON GREEDY?

El problema de encontrar un camino de Euler en un grafo conexo se refiere a encontrar un camino que pase por cada arista del grafo exactamente una vez, además de que comience y termine en el mismo nodo.

Sí, es posible formalizarlo con un algoritmo Greedy, ya que resuelve un problema de optimización mediante una serie de decisiones locales óptimas en cada etapa, para encontrar, o no, una solución global óptima.

Como en el problema planteado, nuestro objetivo es encontrar un camino que pase por todas las aristas, se van tomando decisiones locales óptimas al elegir la siguiente arista por la que pasar y eliminarla del grafo. Esta elección conlleva a una solución global óptima, ya que garantiza que se visiten todas las aristas del grafo exactamente una vez.

Para resolver el problema de minimización del paso por aristas para recorrer todas las aristas, se puede seguir la siguiente metodología de un algoritmo voraz:

- Diseñar una lista de candidatos.
- Identificar una lista de candidatos ya utilizados.
- Diseñar una función solución.
- Diseñar un criterio de factibilidad, que dice cuándo un conjunto de candidatos puede llegar a ser solución.
- Diseñar una función de selección del candidato más prometedor para formar parte de la solución.
- Encontrar una función objetivo de minimización/maximización.
- Adaptar la estructura del procedimiento Greedy al problema y resolverlo.

COMPONENTES GREEDY

Las ***componentes Greedy*** del algoritmo son las siguientes:

1. ***Conjunto de candidatos:*** estaría compuesto por todas las aristas del grafo.

2. ***Función solución:*** determinaría si se han visitado todas las aristas del grafo.

3. ***Función factibilidad:*** determinaría si hay una arista entre el vértice actual y el vértice seleccionado por la función de selección.

4. ***Función selección:*** elegiría el siguiente vértice a visitar. En el caso del algoritmo implementado en nuestro programa, la función de selección elige el vértice adyacente con el mayor grado.

5. ***Función objetivo:*** encontraría un circuito de Euler en el grafo.

ADAPTACIÓN A LAS COMPONENTES

1. Inicializar el conjunto de candidatos con todas las aristas del grafo.

2. Mientras no se haya encontrado una solución:
    a. Seleccionar una arista adyacente al vértice actual utilizando la función de selección.
    b. Comprobar si la arista seleccionada es factible utilizando la función de factibilidad.
    c. Si la arista es factible, agregarla a la solución y eliminarla del conjunto de candidatos.
    d. Actualizar el vértice actual al otro extremo de la arista seleccionada.

3. Devolver la solución encontrada.

EJERCICIO 2

(Implementación del algoritmo)

PSEUDOCÓDIGO

```
FUNCION imprimirCircuito(circuito: vector de enteros)
PARA i DESDE tamaño de circuito - 1 HASTA 0 CON PASO -1
        ESCRIBIR circuito[i]
        SI i ES VERDADERO ENTONCES
        ESCRIBIR " -> "
        FIN SI
FIN PARA
FIN FUNCION

FUNCION encontrarSiguienteVertice(u: entero) :
entero : maxGrado <- -1
entero: siguienteVertice <- -1
PARA v DESDE 0 HASTA N CON PASO +1
        SI grafo[u][v] ES VERDADERO ENTONCES
        grado <- 0
        PARA w DESDE 0 HASTA N CON PASO +1
        SI grafo[v][w] ES VERDADERO ENTONCES
        grado <- grado + 1
        FIN SI
        FIN PARA
        SI grado > maxGrado ENTONCES
        maxGrado <- grado
        siguienteVertice <- v
        FIN SI
        FIN SI
FIN PARA
DEVOLVER siguienteVertice
FIN FUNCION

FUNCION esFactible(u: entero, v: entero) : booleano
DEVOLVER grafo[u][v]
FIN FUNCION

FUNCION esSolucion(circuito: vector de enteros) : booleano
PARA u  DESDE 0 HASTA N CON PASO +1
        PARA v DESDE u + 1 HASTA N CON PASO +1
        SI grafo[u][v] ES VERDADERO ENTONCES
        DEVOLVER falso
        FIN SI
        FIN PARA
FIN PARA
DEVOLVER verdadero
FIN FUNCION
```

```
MAIN Y VARIABLES GLOBALES

VARIABLES GLOBALES:
grafo: matriz de enteros de tamaño MAX x MAX
N: entero

MAIN:
fichero: archivo de entrada

SI argc < 2 ENTONCES
   ESCRIBIR "NO SE HA ENVIADO EL FICHERO QUE CONTIENE LA MATRIZ DEL GRAFO"
FIN SI

ABRIR fichero con nombre argv[1]

SI fichero está abierto ENTONCES
   LEER N desde fichero

   PARA i DESDE 0 HASTA N CON PASO 1
      PARA j DESDE 0 HASTA N CON PASO 1
         LEER grafo[i][j] desde fichero
      FIN PARA
   FIN PARA

   CERRAR fichero

   circuito: vector de enteros
   pila: pila de enteros

   AGREGAR 0 a pila      //SE ELIGE EN QUE NODO VA A COMENZAR EL ALGORITMO

   MIENTRAS pila no esté vacía
      u <- elemento en la cima de pila
      v <- encontrarSiguienteVertice(u)
      SI v != -1 ENTONCES
         grafo[u][v] <- 0
         grafo[v][u] <- 0
         AGREGAR v a pila
      SINO
         AGREGAr u a circuito
         ELIMINAR elemento en la cima de pila
      FIN SI
   FIN MIENTRAS

   SI esSolucion(circuito) ES VERDADERO ENTONCES
      SI primer elemento de circuito == último elemento de circuito ENTONCES
         ESCRIBIR N
         LLAMAR imprimirCircuito(circuito)
      SINO
         ESCRIBIR "NO ES UN CIRCUITO DE EULER"
      FIN SI
   FIN SI
SINO
   LANZAR excepción "No se ha podido abrir el fichero"
FIN SI

DEVOLVER 0
```

CÓDIGO

```
#include <iostream>
#include <fstream>
#include <stack>
#include <vector>
using namespace std;

const int MAX = 1000; // NO vamos a crear una matriz dinámica para este
algoritmo, vamos a hacer una matriz estática
int grafo[MAX][MAX];
int N;

void imprimirCircuito(vector<int> circuito) {
   for (int i = circuito.size() - 1; i >= 0; i--) {
       cout << circuito[i];
       if (i) cout << " -> "; // Para que solo muestres -> en el caso de que
i sea verdadero, para que no haya un -> de más
   }
}

int encontrarSiguienteVertice(int u) {
   int maxGrado = -1;
   int siguienteVertice = -1;
   for (int v = 0; v < N; v++) {
       if (grafo[u][v]) {
           int grado = 0;
           for (int w = 0; w < N; w++) {
               if (grafo[v][w]) grado++;
           }
           if (grado > maxGrado) {
               maxGrado = grado;
               siguienteVertice = v;
           }
       }
   }
   return siguienteVertice;
}

bool esFactible(int u, int v) {
   return grafo[u][v];
}

bool esSolucion(vector<int> circuito) {
   for (int u = 0; u < N; u++) {
       for (int v = u + 1; v < N; v++) {
           if (grafo[u][v]) return false;
       }
   }
   return true;
}
```

```
int main(int argc, char* argv[]) {
   ifstream fichero;
   if (argc <2){
       cerr<<"NO SE HA ENVIADO EL FICHERO QUE CONTIENE LA MATRIZ DEL GRAFO"<<endl;
   }
   fichero.open(argv[1]);
   if (fichero) {
       fichero >> N;

       for (int i = 0; i < N; i++) {
           for (int j = 0; j < N; j++) {
               fichero >> grafo[i][j];
           }
       }
       fichero.close();

       vector<int> circuito;
       stack<int> pila;

       pila.push(0); //SE ELIGE QUE NODO EMPIEZA

       while (!pila.empty()) {
           int u = pila.top();
           int v = encontrarSiguienteVertice(u);
           if (v != -1) {
               grafo[u][v] = grafo[v][u] = 0;
               pila.push(v);
           }
           else {
               circuito.push_back(u);
               pila.pop();
           }
       }
       if (esSolucion(circuito)) {
           if ( circuito.front() == circuito.back()){
               cout << N << endl; //Mostrar cuantos nodos tiene el grafo
               imprimirCircuito(circuito);
           }
           else{
               cerr<<"NO ES UN CIRCUITO DE EULER";
           }
       }
   }
   else {
       throw ios::failure("No se ha podido abrir el fichero");
   }
   return 0;
}
```

Explicación del código:

1. **Lectura del grafo**: El código comienza leyendo el tamaño del grafo y la matriz de adyacencia desde un archivo. El grafo se representa mediante una matriz estática grafo[MAX][MAX], donde grafo[i][j] representa la existencia de una arista entre los nodos i y j.

2. **Inicialización**: Se crean una pila llamada **pila** y un vector llamado **circuito.** La pila se utiliza para **almacenar los nodos visitados** durante la construcción del circuito, mientras que el vector circuito almacenará los nodos en el orden en que se agregan al circuito.

3. **Búsqueda del circuito**: El algoritmo utiliza un enfoque de **búsqueda en profundidad** para encontrar el circuito hamiltoniano. Comienza apilando el nodo de partida (0) en la pila, en este caso es 0 por defecto.

4. **Construcción del circuito:** Mientras la pila no esté vacía, el algoritmo toma el último nodo de la pila y lo asigna a la variable u. Luego, utiliza la función **encontrarSiguienteVertice(u)** para encontrar el siguiente vértice adyacente no visitado con el **mayor número de aristas conectadas**, y lo asigna a la variable v.

5. **Agregar nodos al circuito:** Si se encuentra un siguiente vértice v, se marca la arista entre **u y v como visitada y se apila v**. Si no se encuentra ningún vértice adyacente no visitado, se agrega el nodo u al vector circuito y se desapila un elemento de la pila.

6. **Verificación de la solución**: Una vez que la construcción del circuito ha terminado, se verifica si el circuito obtenido es una solución válida utilizando la función **esSolucion(circuito)**. En este caso, la función **verifica si no hay aristas no visitadas en el grafo** (en el caso en el que se encuentre un “1” en la matriz, devolverá falso). Si el circuito es una solución válida, se llama a la función imprimirCircuito(circuito) para mostrar el circuito en orden inverso.

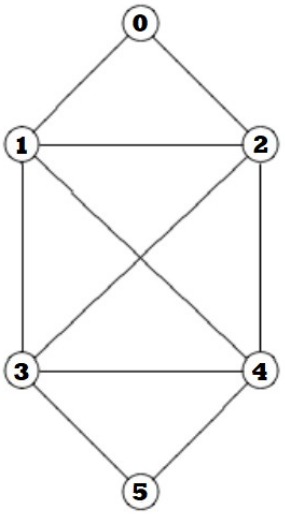
En resumen, el algoritmo greedy utilizado en este código sigue una estrategia de búsqueda en profundidad, donde en cada paso se selecciona el siguiente vértice adyacente no visitado con el mayor número de aristas conectadas. Esto ayuda a maximizar la densidad de aristas en el circuito y buscar una solución aproximada al problema del circuito hamiltoniano.

EJERCICIO 3

(Dos ejemplos de grafos de Euler)

A continuación, vamos a ver dos ejemplos que hemos planteado para resolver con nuestro algoritmo. Tenemos que pasarle a nuestro programa el **número de nodos (N)** para crear la **matriz N x N**, cuya **construcción consta de 0 y 1**, marcando con un 1 aquellos nodos donde se unen dos de ellos (formando una arista).
Como podemos observar, nos muestra para cada nodo su camino de Euler. En el caso en el que se repiten aristas, no aparezcan todas ellas, o no empiece y termine por el mismo nodo, muestra por pantalla que no hay circuito de Euler.

EJEMPLO I



	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	1	1	1	0
2	1	1	0	1	1	0
3	0	1	1	0	1	1
4	0	1	1	1	0	1
5	0	0	0	1	1	0

```
/home/flo/Desktop/ALG/ALG_PRACTICA_3/algoritmo_greedy/cmake-build-debug/algoritmo_greedy ../tests/grafo1.txt
El grafo tiene 6 nodos
Camino comenzando por el nodo [0]:  0 -> 1 -> 2 -> 3 -> 4 -> 1 -> 3 -> 5 -> 4 -> 2 -> 0
```

```
/home/flo/Desktop/ALG/ALG_PRACTICA_3/algoritmo_greedy/cmake-build-debug/algoritmo_greedy ../tests/grafo1.txt
El grafo tiene 6 nodos
Camino comenzando por el nodo [1]:  1 -> 2 -> 3 -> 4 -> 1 -> 0 -> 2 -> 4 -> 5 -> 3 -> 1
```

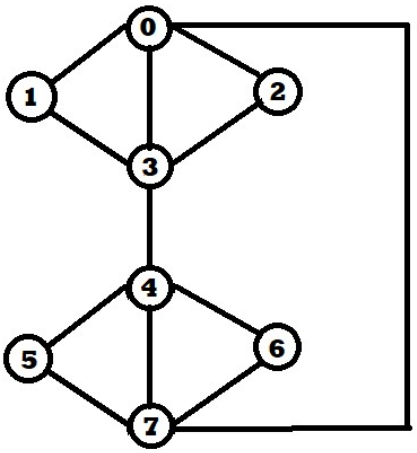
```
/home/flo/Desktop/ALG/ALG_PRACTICA_3/algoritmo_greedy/cmake-build-debug/algoritmo_greedy ../tests/grafo1.txt
El grafo tiene 6 nodos
Camino comenzando por el nodo [2]:  2 -> 1 -> 3 -> 4 -> 2 -> 0 -> 1 -> 4 -> 5 -> 3 -> 2
```

```
/home/flo/Desktop/ALG/ALG_PRACTICA_3/algoritmo_greedy/cmake-build-debug/algoritmo_greedy ../tests/grafo1.txt
El grafo tiene 6 nodos
Camino comenzando por el nodo [3]:  3 -> 1 -> 2 -> 4 -> 3 -> 2 -> 0 -> 1 -> 4 -> 5 -> 3
```

```
/home/flo/Desktop/ALG/ALG_PRACTICA_3/algoritmo_greedy/cmake-build-debug/algoritmo_greedy ../tests/grafo1.txt
El grafo tiene 6 nodos
Camino comenzando por el nodo [4]:  4 -> 1 -> 2 -> 3 -> 4 -> 2 -> 0 -> 1 -> 3 -> 5 -> 4
```

```
/home/flo/Desktop/ALG/ALG_PRACTICA_3/algoritmo_greedy/cmake-build-debug/algoritmo_greedy ../tests/grafo1.txt
El grafo tiene 6 nodos
Camino comenzando por el nodo [5]:  5 -> 3 -> 1 -> 2 -> 4 -> 1 -> 0 -> 2 -> 3 -> 4 -> 5
```

EJEMPLO II



	0	1	2	3	4	5	6	7
0	0	1	1	1	0	0	0	1
1	1	0	0	1	0	0	0	0
2	1	0	0	1	0	0	0	0
3	1	1	1	0	1	0	0	0
4	0	0	0	1	0	1	1	1
5	0	0	0	0	1	0	0	1
6	0	0	0	0	1	0	0	1
7	1	0	0	0	1	1	1	0

```
/home/flo/Desktop/ALG/ALG_PRACTICA_3/algoritmo_greedy/cmake-build-debug/algoritmo_greedy ../tests/grafo2.txt
El grafo tiene 8 nodos
Camino comenzando por el nodo [0]:  0 -> 3 -> 4 -> 7 -> 5 -> 4 -> 6 -> 7 -> 0 -> 1 -> 3 -> 2 -> 0
```

```
/home/flo/Desktop/ALG/ALG_PRACTICA_3/algoritmo_greedy/cmake-build-debug/algoritmo_greedy ../tests/grafo2.txt
El grafo tiene 8 nodos
Camino comenzando por el nodo [1]:  1 -> 0 -> 3 -> 4 -> 7 -> 5 -> 4 -> 6 -> 7 -> 0 -> 2 -> 3 -> 1
```

```
/home/flo/Desktop/ALG/ALG_PRACTICA_3/algoritmo_greedy/cmake-build-debug/algoritmo_greedy ../tests/grafo2.txt
El grafo tiene 8 nodos
Camino comenzando por el nodo [2]:  2 -> 0 -> 3 -> 4 -> 7 -> 5 -> 4 -> 6 -> 7 -> 0 -> 1 -> 3 -> 2
```

```
/home/flo/Desktop/ALG/ALG_PRACTICA_3/algoritmo_greedy/cmake-build-debug/algoritmo_greedy ../tests/grafo2.txt
El grafo tiene 8 nodos
Camino comenzando por el nodo [3]:  3 -> 0 -> 7 -> 4 -> 5 -> 7 -> 6 -> 4 -> 3 -> 1 -> 0 -> 2 -> 3
```

```
/home/flo/Desktop/ALG/ALG_PRACTICA_3/algoritmo_greedy/cmake-build-debug/algoritmo_greedy ../tests/grafo2.txt
El grafo tiene 8 nodos
Camino comenzando por el nodo [4]:  4 -> 3 -> 0 -> 1 -> 3 -> 2 -> 0 -> 7 -> 4 -> 5 -> 7 -> 6 -> 4
```

```
/home/flo/Desktop/ALG/ALG_PRACTICA_3/algoritmo_greedy/cmake-build-debug/algoritmo_greedy ../tests/grafo2.txt
El grafo tiene 8 nodos
Camino comenzando por el nodo [5]:  5 -> 4 -> 3 -> 0 -> 1 -> 3 -> 2 -> 0 -> 7 -> 4 -> 6 -> 7 -> 5
```

```
/home/flo/Desktop/ALG/ALG_PRACTICA_3/algoritmo_greedy/cmake-build-debug/algoritmo_greedy ../tests/grafo2.txt
El grafo tiene 8 nodos
Camino comenzando por el nodo [6]:  6 -> 4 -> 3 -> 0 -> 1 -> 3 -> 2 -> 0 -> 7 -> 4 -> 5 -> 7 -> 6
```

```
/home/flo/Desktop/ALG/ALG_PRACTICA_3/algoritmo_greedy/cmake-build-debug/algoritmo_greedy ../tests/grafo2.txt
El grafo tiene 8 nodos
Camino comenzando por el nodo [7]:  7 -> 0 -> 3 -> 1 -> 0 -> 2 -> 3 -> 4 -> 7 -> 5 -> 4 -> 6 -> 7
```

EFICIENCIA TEÓRICA

Analicemos la eficiencia teórica de cada función por separado:

Función `imprimirCircuito(vector<int> circuito)`:

```
Para cada elemento en circuito:  // O(N)
    Imprimir el elemento  // O(1)
```

Complejidad de tiempo: O(N), donde N es el número de elementos en el vector `circuito`. La función recorre el vector e imprime cada elemento una vez. Como imprimir un elemento es una operación constante, la complejidad total es O(N).

Función `encontrarSiguienteVertice(int u)`:

```
maxGrado <- -1  // O(1)
siguienteVertice <- -1  // O(1)

Para cada vértice v en el rango de 0 a N-1:  // O(N)
    Si grafo[u][v] es verdadero entonces:  // O(1)
        grado <- 0  // O(1)

        Para cada vértice w en el rango de 0 a N-1:  // O(N)
            Si grafo[v][w] es verdadero entonces:  // O(1)
                grado <- grado + 1  // O(1)
            Fin Si

        Si grado > maxGrado entonces:  // O(1)
            maxGrado <- grado  // O(1)
            siguienteVertice <- v  // O(1)
        Fin Si
```

Complejidad de tiempo: O(N), donde N es el número de nodos en el grafo. El bucle externo itera a través de los vértices del grafo, lo cual requiere O(N) operaciones. El bucle interno también itera O(N) veces. Dado que todas las operaciones dentro de los bucles son constantes, la complejidad total es O(N).

Función `esFactible(int u, int v)`:

```
Retornar grafo[u][v]  // O(1)
```

Complejidad de tiempo: O(1). La función realiza una operación de acceso a la matriz `grafo` para verificar la factibilidad de una arista entre los nodos `u` y `v`. Esta operación es constante y no depende del tamaño del grafo.

Función `esSolucion(vector<int> circuito)`:

```
Para cada par de elementos en circuito:  // O(N^2)
    Si hay una arista entre los elementos entonces:  // O(1)
        Retornar falso  // O(1)
Retornar verdadero  // O(1)
```

Complejidad de tiempo: O(N^2), donde N es el número de elementos en el vector `circuito`. La función compara todos los pares de elementos en el vector para verificar si hay alguna arista entre ellos. Dado que hay O(N^2) pares de elementos en total, la complejidad es O(N^2).

- **Eficiencia Teórica del main:**

```
int main(int argc, char* argv[]) {
   ifstream fichero;
   if (argc < 2) {
        cerr << "NO SE HA ENVIADO EL FICHERO QUE CONTIENE LA MATRIZ DEL
GRAFO" << endl;
   }
   fichero.open(argv[1]);
   if (fichero) {
       fichero >> N;  // O(1)

       for (int i = 0; i < N; i++) {
           for (int j = 0; j < N; j++) {
               fichero >> grafo[i][j];  // O(1)
           }
       }
       fichero.close();  // O(1)

       vector<int> circuito;  // O(1)
       stack<int> pila;  // O(1)

       pila.push(0);  // O(1)

       while (!pila.empty()) {  // Hasta O(N)
           int u = pila.top();  // O(1)
           int v = encontrarSiguienteVertice(u);  // O(N)
           if (v != -1) {
               grafo[u][v] = grafo[v][u] = 0;  // O(1)
               pila.push(v);  // O(1)
           }
           else {
               circuito.push_back(u);  // O(1)
               pila.pop();  // O(1)
           }
       }
       if (esSolucion(circuito)) {  // O(N^2)
           if (circuito.front() == circuito.back()) {  // O(1)
               cout << N << endl;  // O(1)
               imprimirCircuito(circuito);  // O(N)
           }
           else {
               cerr << "NO ES UN CIRCUITO DE EULER";  // O(1)
           }
       }
   }
   else {
         throw ios::failure("No se ha podido abrir el fichero");  //
O(1)
   }
   return 0;  // O(1)
}
```

Las operaciones de apertura de archivo, lectura del valor `N`, cierre del archivo y verificación de la cantidad de argumentos tienen una complejidad de tiempo constante, **O(1)**.

- El bucle anidado que lee los elementos de la matriz de adyacencia tiene una complejidad de tiempo de **O(N^2)**, ya que hay dos bucles que iteran N veces cada uno.
- La inicialización de la pila, el bucle principal y la llamada a las funciones **`encontrarSiguienteVertice`, `esSolucion` e `imprimirCircuito`** tienen una complejidad de tiempo que depende de la cantidad de nodos en el grafo, **O(N)**.
- La verificación de si el circuito es un ciclo de Euler (`circuito.front() == circuito.back()`) se realiza en tiempo constante, **O(1).**

- **En general, la complejidad de tiempo del `main` es O(N^2), ya que la parte con mayor complejidad es el bucle anidado que lee los elementos de la matriz de adyacencia.**