Algorítmica

Práctica 3

Algoritmos Greedy

Trabajo realizado por:	León Elliott Fuller Toni Martí Albons Alejandro Madueño Buciegas

Índice:

Formalización Greedy

Comprobación

Si estudiamos el problema dado, podemos deducir que el algoritmo propuesto cumple la propiedad de elección voraz, ya que mediante la selección de soluciones locales óptimas, podemos llegar a una solución óptima global.
Además al aplicar el algoritmo en una subestructura del grafo (también grafo de euler), se encuentra una solución óptima, por lo que cabe de esperar que al añadir más componentes (vértices y aristas), siempre y cuando el grafo se mantenga como un grafo de euler, se encontrará solución óptima.

Diseño de componentes

Inicialización: El nodo inicial es seleccionado de forma arbitraria, y tanto la lista de vértices o nodos visitados como los no visitados están vacías inicialmente. La lista de nodos visitados servirá de conjunto de soluciones parciales.

Función de selección: El nodo a visitar, será seleccionado en función del número de aristas incidentes no visitadas, siendo el nodo a visitar el que mayor número de aristas incidentes no visitadas tenga.

Criterio de factibilidad: Se verifica que el nodo seleccionado sea adyacente al nodo actual y que la arista que los conecta no ha sido visitada antes. Si el nodo seleccionado no cumple estos criterios, se selecciona un nodo diferente.

Función de solución: Si se han visitado todas las aristas del grafo, se devuelve la lista de nodos visitados que conforman el circuito de Euler.

Función de objetivo: En este caso, al no haber costes y no pedirse una solución óptima a nivel de coste, o menor número de nodos visitados, el algoritmo tratará de mejorar la posibilidad de encontrar o no solución. Debido a esto no es necesario establecer una función objetivo.

Plantilla Greedy

En base a las explicaciones expuestas en el apartado anterior, la plantilla Greedy que describe el algoritmo dado será la siguiente:

```
S = ∅
A = ∅
G = Vertices del grafo
v ∈ G

Si|G| ≤ 1
   Error no hay vertices en el grafo o solo hay uno
Si |G| > 1
   S = S ∪ v      //insertamos v en el conjunto solucion

   Mientras que |v| > 0 :       //mientras que haya aristas incidentes en v
      Si |v| = 1     // si hay una única arista incidente
         G = G − G ∩ v
         v = candidato
         S = S ∪ v

   Para toda arista a que incida en v:
      cuando una mantiene el grafo conexo trás su eliminación:
         G = G − G ∩ v
         S = S ∪ v
         v = candidato

Devolvemos S
```

El vértice ***candidato*** será el elegido por la función de selección.

Implementación del algoritmo

Una vez visto el diseño de la plantilla Greedy, la implementación del algoritmo general es el siguiente:

Función que comprueba si el grafo es conexo trás la eliminación de una arista:

```
template <typename T>
bool mismaComponenteConexaAlDesconectar(const Graph<T>& graph, Node<T>& nodo1, Node<T>& nodo2) { // O(n²logn)
    /*
    iniciamos la lista con el nodo

    mientras familiares != comprobados: SE ALTERA ENTRE LAS LISTAS DE LOS DOS NODOS INICIALES
        cojemos uno en familiares que no este en comprobados
        para cada vecino de este nodo:
            comprobamos que no este en familiares del otro nodo inicial -> SI LO ESTA ES CONEXO
            lo metemos en familiares

    SI FAMILIARES == COMPROBADOS Y NO TIENE TANTOS NODOS COMO EL GRAFO, NO ES CONEXO
    */
    if (graph.node_n() <= 1 or &nodo1 == &nodo2) return true;

    nodo1.removeEdgeTo(nodo2);

    std::set<Node<T>*> familiares1, familiares2;
    std::queue<Node<T>*> siguiente_a_comprobar1, siguiente_a_comprobar2;
    familiares1.insert(&nodo1);
    siguiente_a_comprobar1.push(&nodo1);
    familiares2.insert(&nodo2);
    siguiente_a_comprobar2.push(&nodo2);

    while( (not siguiente_a_comprobar1.empty()) and
           (not siguiente_a_comprobar2.empty()) ) { // O(n²logn)

        Node<T> &comprobando1 = *(siguiente_a_comprobar1.front());
        siguiente_a_comprobar1.pop();
        std::list<Node<T>*> vecinos1 = comprobando1.getConnections();
        Node<T> * vecino_anterior1_ptr = nullptr;
        for(Node<T>* vecino1_ptr : vecinos1) {  // O(nlogn)
            if (vecino1_ptr == vecino_anterior1_ptr) continue;

            if(familiares1.insert(vecino1_ptr).second) {
                if(familiares2.find(vecino1_ptr) != familiares2.end()) { // O(logn)
                    nodo1.createEdgeTo(nodo2);
                    return true;
                }

                siguiente_a_comprobar1.push(vecino1_ptr);
            }

            vecino_anterior1_ptr = vecino1_ptr;
        }

        Node<T> &comprobando2 = *(siguiente_a_comprobar2.front());
        siguiente_a_comprobar2.pop();
        std::list<Node<T>*> vecinos2 = comprobando2.getConnections();
        Node<T> * vecino_anterior2_ptr = nullptr;
        for(Node<T>* vecino2_ptr : vecinos2) { // O(nlogn)
            if (vecino2_ptr == vecino_anterior2_ptr) continue;

            if(familiares2.insert(vecino2_ptr).second) {
                if(familiares1.find(vecino2_ptr) != familiares1.end()) { // O(logn)
                    nodo2.createEdgeTo(nodo1);
                    return true;
                }

                siguiente_a_comprobar2.push(vecino2_ptr);
            }

            vecino_anterior2_ptr = vecino2_ptr;
        }
    }

    nodo1.createEdgeTo(nodo2);
    return false;
}
```

Esta función, se encarga, en líneas generales de comprobar los nodos vecinos al nodo al que se va a eliminar la arista para comprobar si es conexo.

Función que encuentra el circuito de euler:

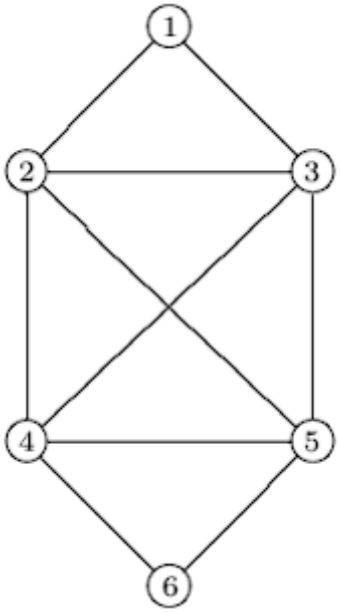
```
/**
 * @pre graph es un grafo de Euler
 * @post elimina todos los ejes del grafo
 */
template <typename T>
std::list<Node<T>*> findEulerCircuit(Graph<T>& graph) {
    std::list<Node<T> *> circuit;
    if (graph.node_n() == 0)
        return circuit;

    Node<T>* actual = &graph.getNode();
    circuit.push_back(actual);

    while(actual->degree() > 0) { // O(n^4*logn^2)
        std::list<Node<T> *> connections = actual->getConnections();
        auto connection_candidate = connections.begin();

        if (actual->degree() == 1) {
            actual->removeEdgeTo(**connection_candidate);
            circuit.push_back(*connection_candidate);
            actual = *connection_candidate;
            continue;
        }

        while(not mismaComponenteConexaAlDesconectar(graph, *actual, **connection_candidate)) { // O(n^3logn
)
            connection_candidate++;
        }
        actual->removeEdgeTo(**connection_candidate);
        circuit.push_back(*connection_candidate);
        actual = *connection_candidate;
    }

    return circuit;
}
```

Esta función implementa la plantilla greedy con la ayuda de la función auxiliar previamente expuesta.

Ejemplos de solución

Grafo de 6 nodos

El grafo a solucionar es el expuesto en el guión, concretamente:



Para ello es necesario que en el fichero de código fuente ***main.cpp***, esté la siguiente línea de código:

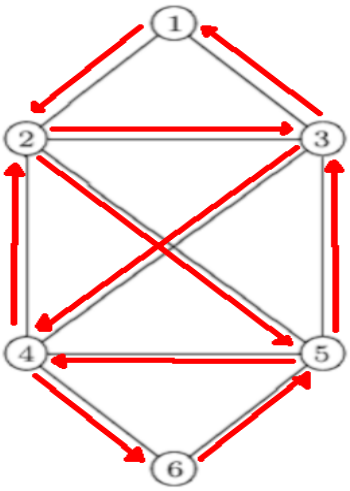
```
graph.load("./data/example_ejemplo_guion.txt");
```

Compilamos con ***g++ -std=c++17 ./include/* ./src/main.cpp -o ./bin/main.bin*** y ejecutamos.

En este caso el resultado dado es:

```
Circuito de Euler encontrado: 1 -> 2 -> 3 -> 4 -> 2 -> 5 -> 4 -> 6 -> 5 -> 3 -> 1 -> Inicio
```

De forma gráfica sería tal que:



Grafo de Euler de 100 nodos

En este caso se ha creado un ejemplo con 100 nodos exactamente. Para poder ejecutar su resolución es necesario, que en el fichero de código fuente *main.cpp*, esté la siguiente línea de código:

```
graph.load("./data/example_euler_graph.txt");
```

Compilamos con ***g++ -std=c++17 ./include/* ./src/main.cpp -o ./bin/main.bin*** y ejecutamos.

En este caso el resultado dado es:

```
Circuito de Euler encontrado: 1 -> 1 -> 2 -> 3 -> 4 -> 4 -> 5 -> 5 -> 6 -> 6 -> 6 -> 6 -> 7
 -> 7 -> 7 -> 7 -> 8 -> 9 -> 9 -> 9 -> 10 -> 10 -> 10 -> 11 -> 12 -> 12 -> 13 -> 14 -> 14 -
> 14 -> 14 -> 15 -> 16 -> 17 -> 18 -> 3 -> 23 -> 22 -> 20 -> 19 -> 19 -> 19 -> 18 -> 18 ->
82 -> 6 -> 26 -> 25 -> 16 -> 71 -> 14 -> 27 -> 13 -> 40 -> 39 -> 38 -> 37 -> 12 -> 43 -> 30
 -> 3 -> 98 -> 2 -> 57 -> 12 -> 69 -> 40 -> 40 -> 41 -> 27 -> 26 -> 26 -> 85 -> 4 -> 52 ->
51 -> 44 -> 43 -> 43 -> 43 -> 42 -> 41 -> 41 -> 41 -> 41 -> 41 -> 27 -> 27 -> 28 -> 28 -> 2
8 -> 29 -> 29 -> 29 -> 29 -> 30 -> 14 -> 58 -> 25 -> 24 -> 23 -> 23 -> 23 -> 23 -> 23 -> 59
 -> 58 -> 57 -> 53 -> 52 -> 55 -> 54 -> 53 -> 53 -> 93 -> 18 -> 98 -> 72 -> 71 -> 35 -> 34
-> 34 -> 33 -> 33 -> 33 -> 33 -> 32 -> 31 -> 30 -> 30 -> 30 -> 36 -> 35 -> 35 -> 65 -> 44 -
> 44 -> 44 -> 44 -> 45 -> 20 -> 20 -> 20 -> 20 -> 21 -> 21 -> 21 -> 22 -> 22 -> 22 -> 22 ->
 22 -> 50 -> 49 -> 49 -> 49 -> 48 -> 47 -> 6 -> 100 -> 55 -> 55 -> 56 -> 56 -> 57 -> 74 ->
22 -> 63 -> 24 -> 68 -> 2 -> 81 -> 57 -> 94 -> 16 -> 78 -> 28 -> 63 -> 31 -> 31 -> 83 -> 30
 -> 47 -> 47 -> 47 -> 46 -> 15 -> 68 -> 30 -> 74 -> 63 -> 62 -> 62 -> 61 -> 20 -> 85 -> 28
-> 91 -> 60 -> 59 -> 59 -> 70 -> 68 -> 35 -> 79 -> 68 -> 36 -> 36 -> 37 -> 37 -> 37 -> 73 -
> 27 -> 64 -> 60 -> 60 -> 60 -> 60 -> 61 -> 61 -> 61 -> 77 -> 9 -> 88 -> 51 -> 51 -> 50 ->
93 -> 87 -> 27 -> 92 -> 81 -> 80 -> 80 -> 80 -> 80 -> 80 -> 79 -> 77 -> 69 -> 68 -> 6
7 -> 66 -> 42 -> 42 -> 87 -> 36 -> 94 -> 68 -> 68 -> 69 -> 69 -> 70 -> 70 -> 70 -> 71 -> 63
 -> 63 -> 64 -> 65 -> 65 -> 66 -> 90 -> 45 -> 45 -> 45 -> 46 -> 46 -> 83 -> 82 -> 72 -> 73
-> 73 -> 73 -> 74 -> 74 -> 74 -> 75 -> 75 -> 76 -> 76 -> 77 -> 77 -> 77 -> 78 -> 79 -> 79 -
> 89 -> 85 -> 38 -> 38 -> 99 -> 25 -> 25 -> 25 -> 25 -> 25 -> 96 -> 40 -> 95 -> 87 -> 81 ->
 82 -> 82 -> 97 -> 71 -> 71 -> 71 -> 96 -> 83 -> 83 -> 84 -> 85 -> 85 -> 85 -> 86 -> 87 ->
87 -> 88 -> 88 -> 88 -> 89 -> 90 -> 91 -> 91 -> 91 -> 92 -> 93 -> 93 -> 94 -> 94 -> 9
4 -> 95 -> 95 -> 96 -> 97 -> 97 -> 97 -> 98 -> 98 -> 98 -> 99 -> 99 -> 99 -> 100 -> 100 ->
100 -> 100 -> 1 -> Inicio
```

Estudio de la eficiencia

Para estudiar la eficiencia del algoritmo completo hemos de estudiar dos funciones por separado:

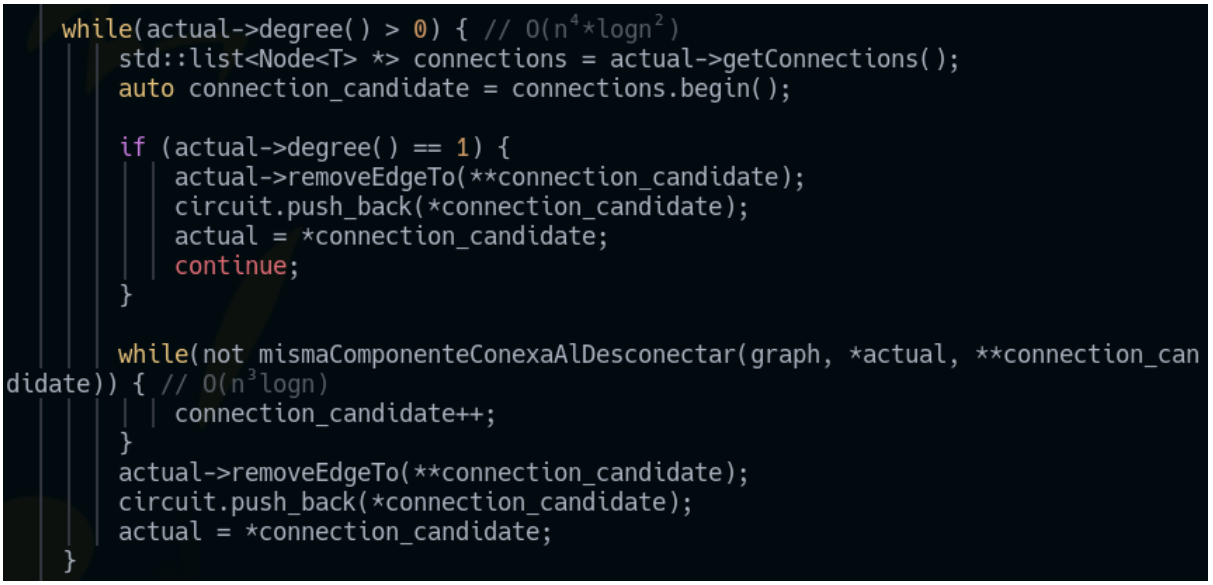
Función que comprueba si el grafo sigue siendo conexo al eliminar una arista:

```
while( (not siguiente_a_comprobar1.empty()) and
       (not siguiente_a_comprobar2.empty()) ) { // O(n²logn)

    Node<T> &comprobando1 = *(siguiente_a_comprobar1.front());
    siguiente_a_comprobar1.pop();
    std::list<Node<T>*> vecinos1 = comprobando1.getConnections();
    Node<T> * vecino_anterior1_ptr = nullptr;
    for(Node<T>* vecino1_ptr : vecinos1) {  // O(nlogn)
        if (vecino1_ptr == vecino_anterior1_ptr) continue;

        if(familiares1.insert(vecino1_ptr).second) {
            if(familiares2.find(vecino1_ptr) != familiares2.end()) { // O(logn)
                nodo1.createEdgeTo(nodo2);
                return true;
            }

            siguiente_a_comprobar1.push(vecino1_ptr);
        }

        vecino_anterior1_ptr = vecino1_ptr;
    }

    Node<T> &comprobando2 = *(siguiente_a_comprobar2.front());
    siguiente_a_comprobar2.pop();
    std::list<Node<T>*> vecinos2 = comprobando2.getConnections();
    Node<T> * vecino_anterior2_ptr = nullptr;
    for(Node<T>* vecino2_ptr : vecinos2) { // O(nlogn)
        if (vecino2_ptr == vecino_anterior2_ptr) continue;

        if(familiares2.insert(vecino2_ptr).second) {
            if(familiares1.find(vecino2_ptr) != familiares1.end()) { // O(logn)
                nodo2.createEdgeTo(nodo1);
                return true;
            }

            siguiente_a_comprobar2.push(vecino2_ptr);
        }

        vecino_anterior2_ptr = vecino2_ptr;
    }
}
```

Como podemos observar el núcleo de este método se encuentra principalmente en unos bucles anidados, si estudiamos su eficiencia, comprobamos que el método ***find()*** de la estructura usada para almacenar las listas de vértices (***set***) tiene una eficiencia de **O(log(n))**. Este método se ejecuta **n** veces dentro del bucle *for*, y a su

vez este bucle, se ejecuta **n** veces dentro del bucle *while*. En conclusión, el método tendrá una eficiencia de **O(n^2*log(n))**.

Función que encuentra el circuito de euler:



```
while(actual->degree() > 0) { // O(n^4*logn^2)
    std::list<Node<T> *> connections = actual->getConnections();
    auto connection_candidate = connections.begin();

    if (actual->degree() == 1) {
        actual->removeEdgeTo(**connection_candidate);
        circuit.push_back(*connection_candidate);
        actual = *connection_candidate;
        continue;
    }

    while(not mismaComponenteConexaAlDesconectar(graph, *actual, **connection_candidate)) { // O(n^3logn)
        connection_candidate++;
    }
    actual->removeEdgeTo(**connection_candidate);
    circuit.push_back(*connection_candidate);
    actual = *connection_candidate;
}
```

La parte clave del algoritmo está formada por dos bucles while anidados. En el bucle interno se va a ejecutar **n** veces la función que hemos estudiado con anterioridad, lo que da una eficiencia de **O(n^3*log(n))**. Este bucle interno a su vez se va a ejecutar, en el peor de los casos **n-1**, suponiendo que el vértice actual tenga conexiones con todos los otros vértices.
El algoritmo al completo da una complejidad de **O(n^4*log(n))**.