GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: CAMINO Y CICLO DE EULER

1. Introducción

En la teoría de grafos, un camino euleriano es un camino que pasa por cada arista una y solo una vez. Un ciclo o circuito euleriano es un camino cerrado que recorre cada arista exactamente una vez. El problema de encontrar dichos caminos fue discutido por primera vez por Leonhard Euler, en el famoso problema de los puentes de Königsberg.

Sobre como determinar si un grafo presenta o no un camino de euler y como encontrarlo lo veremos en esta guía.

2. Conocimientos previos

2.1. Leonhard Euler

Leonhard Euler fue un matemático y físico suizo. Se trata del principal matemático del siglo XVIII y uno de los más grandes y prolíficos de todos los tiempos, muy conocido por el número de Euler (e), número que aparece en muchas fórmulas de cálculo y física.

2.2. Camino en un grafo

En teoría de grafos, un camino (en inglés, *walk*,y en ocasiones traducido también como recorrido) es una sucesión de vértices y aristas dentro de un grafo, que empieza y termina en vértices, tal que cada vértice es incidente con las aristas que le siguen y le preceden en la secuencia. Dos vértices están conectados o son accesibles si existe un camino que forma una trayectoria para llegar de uno al otro; en caso contrario, los vértices están desconectados o bien son inaccesibles.

2.3. Ciclo en un grafo

La palabra ciclo se emplea en teoría de grafos para indicar un camino cerrado en un grafo, es decir, en que el nodo de inicio y el nodo final son el mismo.

3. Desarrollo

Antes de buscar el camino o ciclo euleriano debemos comprobar que existe el mismo para esto nos vamos apoyar el los siguientes teoremas:

- **Teorema 1:** Sea G un grafo o multigrafo no dirigido. Entonces G tiene un ciclo de Euler si, y solo si, es conexo y todo vértice tiene grado par. Diremos que es un grafo Euleriano.
- **Teorema 2:** Sea G un grafo o multigrafo no dirigido. Entonces G tiene un camino de Euler si, y solo si, es conexo y tiene solo dos vértices de grado impar. Diremos que el grafo es semi-Euleriano.

Los resultados obtenidos para grafos no dirigidos pueden extenderse de inmediato para grafos dirigidos. En un grafo dirigido el grado o valencia de entrada de un vértice es el numero de lados incidentes hacia este y el grado de salida es el numero de lados que son incidentes desde este.

- **Teorema 4:** Un grafo dirigido tiene un ciclo de Euler si y solo si es conexo y el grado de entrada de cualquier vértice es igual a su salida.
- **Teorema 5:** Un grafo dirigido tiene un camino de Euler si y solo si es conexo y el grado de entrada de cualquier vértice es igual a su grado de salida con la posible excepción de solo dos vértices. Para estos dos vértices el grado de entrada de uno de ellos es mayor que su grado de salida y el grado de entrada del otro es menor que su grado de salida.

3.1. Algoritmo de Fleury

El algoritmo de Fleury es un algoritmo elegante pero ineficiente que data de 1883. Considere un grafo que se sabe que tiene todas las aristas en el mismo componente y como máximo dos vértices de grado impar. El algoritmo comienza en un vértice de grado impar o, si el grafo no tiene ninguno, comienza con un vértice elegido arbitrariamente. En cada paso, elige la siguiente arista en la ruta para que sea uno cuya eliminación no desconecte el grafo, a menos que no exista tal arista, en cuyo caso elige la arista restante que queda en el vértice actual. Luego se mueve al otro extremo de esa arista y elimina la arista. Al final del algoritmo no quedan aristas, y la secuencia a partir de la cual se eligieron las aristas forma un ciclo euleriano si el gráfico no tiene vértices de grado impar, o un camino euleriano si hay exactamente dos vértices de grado impar.

Mientras que el recorrido del gráfico en el algoritmo de Fleury es lineal en el número de aristas, es decir O(E), también debemos tener en cuenta la complejidad de detectar puentes . Si vamos a volver a ejecutar el algoritmo de búsqueda de puentes de tiempo lineal de Tarjan después de eliminar cada arista, el algoritmo de Fleury tendrá una complejidad de tiempo de O(E^2). Un algoritmo dinámico de búsqueda de puentes de Thorup permite mejorar esto O($E \cdot \log^3 E \cdot \log \log E$), pero sigue siendo significativamente más lento que los algoritmos alternativos.

3.2. Algoritmo de Hierholzer

El artículo de 1873 de Hierholzer proporciona un método diferente para encontrar ciclos de Euler que sea más eficiente que el algoritmo de Fleury:

- Elija cualquier vértice de inicio v , y siga un rastro de aristas desde ese vértice hasta volver a v . No es posible atascarse en ningún vértice que no sea v , porque el grado par de todos los vértices asegura que, cuando el camino entre en otro vértice w , debe haber una arista sin usar que salga de w . El recorrido así formado es un recorrido cerrado, pero puede que no cubra todos los vértices y aristas del grafo inicial.
- Siempre que exista un vértice u que pertenezca al recorrido actual pero que tenga aristas adyacentes que no formen parte del recorrido, se inicia otro recorrido desde u , siguiendo aristas no utilizadas hasta volver a u , y se une el recorrido así formado al anterior recorrido.
- Dado que asumimos que el grafo original está conectado , repetir el paso anterior agotará todos las aristas del grafo.

Mediante el uso de una estructura de datos como una lista doblemente enlazada para mantener el conjunto de aristas no utilizadas que inciden en cada vértice, para mantener la lista de

vértices en el recorrido actual que tienen aristas no utilizadas y para mantener el recorrido en sí, las operaciones individuales del algoritmo (encontrar bordes no utilizados que salen de cada vértice, encontrar un nuevo vértice de inicio para un recorrido y conectar dos recorridos que comparten un vértice) se puede realizar en tiempo constante cada uno, por lo que el algoritmo general toma un tiempo lineal, O(E).

Este algoritmo también se puede implementar con un deque. Debido a que solo es posible atascarse cuando el deque representa un recorrido cerrado, se debe rotar el deque eliminando los bordes de la cola y agregándolos a la cabeza hasta que se despegue, y luego continuar hasta que se tengan en cuenta todos los bordes. Esto también lleva un tiempo lineal, ya que el número de rotaciones realizadas nunca es mayor que (Intuitivamente, cualquier mala arista se mueve a la cabeza, mientras que las aristas nuevas se agregan a la cola)

Para encontrar el camino euleriano/ciclo euleriano, podemos usar la siguiente estrategia: encontramos todos los ciclos simples y los combinamos en uno: este será el ciclo euleriano. Si el grafo es tal que la ruta euleriana no es un ciclo, agregue la arista que falta, encuentre el ciclo euleriano y luego elimine la arista adicional.

Buscar todos los ciclos y combinarlos se puede hacer con un procedimiento recursivo simple:

```
procedimiento FindEulerPath(V)
  1. iterar a traves de todas las aristas que salen del vertice V;
        eliminar esta arista del grafo,
        y llame a FindEulerPath desde el segundo extremo de esta arista;
  2. agregue el vertice V a la respuesta.
```

Pero podemos escribir el mismo algoritmo en la versión no recursiva:

```
pila St;
ponga el vertice inicial en St;
hasta que St este vacio
  sea V el valor en la parte superior de St;
  si grado(V) = 0, entonces
    suma V a la respuesta;
    quitar V de la parte superior de St;
  sino
    encuentre cualquier arista que salga de V;
    eliminarla del grafo;
    ponga el segundo extremo de esta arista en St;
```

Es fácil comprobar la equivalencia de estas dos formas del algoritmo. Sin embargo, la segunda forma es obviamente más rápida y el código será mucho más eficiente.

Primero, el función verifica el grado de los vértices: si no hay vértices con un grado impar, entonces el grafo tiene un ciclo de Euler, si hay 2 vértices con un grado impar, entonces en el grafo solo hay un camino de Euler (pero no un ciclo de Euler), si hay más de 2 tales vértices, entonces en el grafo no hay ciclo de Euler o camino de Euler. Para encontrar el camino de Euler (no un ciclo),

hagamos esto: si $V1$ y $V2$ son dos vértices de grado impar, luego solo agregue una arista $(V1, V2)$, en el grafo resultante encontramos el ciclo de Euler (obviamente existirá), y luego eliminamos la arista *ficticia* $(V1, V2)$ de la respuesta Buscaremos el ciclo de Euler exactamente como se describe arriba (versión no recursiva), y al mismo tiempo al final de este algoritmo verificaremos si el grafo estaba conectado o no (si el grafo no estaba conectado, entonces al final del algoritmo algunas aristas permanecerán en el grafo, y en este caso necesitamos imprimir -1). Finalmente, la función tiene en cuenta que pueden existir vértices aislados en el grafo.

4. Implementación

4.1. C++

```
vector<int> findPathEuler(vector<vector<int> > matrix_ady, int n) {
   vector<int> deg(n); vector<int> path; path.assign(1,-1);
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j)
         deg[i] += matrix_ady[i][j];
   }
   int first = 0;
   while (first < n && !deg[first]) ++first;

   if(first == n){return path;}

   int v1 = -1, v2 = -1;
   bool bad = false;
   for (int i = 0; i < n; ++i) {
      if (deg[i] & 1) {
         if (v1 == -1) v1 = i;
         else if (v2 == -1) v2 = i;
         else bad = true;
      }
   }

   if (v1 != -1){
      ++matrix_ady[v1][v2]; ++matrix_ady[v2][v1];
   }
   stack<int> st; st.push(first);
   vector<int> res;
   while (!st.empty()) {
      int v = st.top(); int i;
      for (i = 0; i < n; ++i)
         if (matrix_ady[v][i]) break;
      if (i == n) { res.push_back(v); st.pop();}
      else { --matrix_ady[v][i];
             --matrix_ady[i][v]; st.push(i); }
   }
   if (v1 != -1) {
      for (size_t i = 0; i + 1 < res.size(); ++i) {
```

```
            if ((res[i] == v1 && res[i + 1] == v2)
                || (res[i] == v2 && res[i + 1] == v1)) {
               vector<int> res2;
               for (size_t j = i + 1; j < res.size(); ++j)
                  res2.push_back(res[j]);
               for (size_t j = 1; j <= i; ++j)
                  res2.push_back(res[j]);
               res = res2;
               break;
            }
         }
      }

      for (int i = 0; i < n; ++i) {
         for (int j = 0; j < n; ++j) {
            if (matrix_ady[i][j]) bad = true;
         }
      }

      if (bad){ return path; }
      else {
         path.clear();
         for (int x : res)
            path.push_back(x);
      }
   }
```

Esta implementación esta pensada para multigrafo no dirigido con bucles

La siguiente implementación halla el ciclo de Euler en un grafo dirigido asumiendo que todos los nodos sus grados de entrada y salida son iguales y el grafo es conexo. Al método se le pasa el grafo y el nodo por el cual quiere comenzar el ciclo.

```
vector<int> eulerCycle(vector<vector<int> > graph, int v) {
   vector<int> curEdge (graph.size(),0);
   vector<int> res; stack<int> st ;
   st.push(v);
   while (!st.empty()) {
      v = st.pop();
      while (curEdge[v] < graph[v].size()) {
         st.push(v); v = graph[v][curEdge[v]++];
      }
      res.push_back(v);
   }
   reverse(res.begin(),res.end());
   return res;
}
```

Implementación del algoritmo de Hierholzer que realiza DFS desde el vértice inicial y permite

determinar si hay camino (*eulerian_path*) o ciclo (*eulerian_tour*) de euler para grafos no dirigidos.

```
#define fst first
#define snd second
#define all(c) ((c).begin()), ((c).end())

struct graph {
   int n;
   struct edge { int src, dst, rev; };
   vector<vector<edge>> adj;
   graph() : n(0) { }
   //graph(int n) : n(n), adj(n) { }

   void add_edge(int src, int dst) {
      n = max(n, max(src, dst)+1);
      if (adj.size() < n) adj.resize(n);
      adj[src].push_back({src, dst, (int)adj[dst].size()});
      adj[dst].push_back({dst, src, (int)adj[src].size()-1});
   }

   vector<int> path;
   void visit(int u) {
      while (!adj[u].empty()) {
         auto e = adj[u].back(); adj[u].pop_back();
         if (e.src >= 0){
            adj[e.dst][e.rev].src = -1;
            visit(e.dst);
         }
      }
      path.push_back(u);
   }
   vector<int> eulerian_path() {
      int m = 0, s = -1;
      for (int u = 0; u < n; ++u) {
         m += adj[u].size();
         if (adj[u].size() % 2 == 1) s = u;
      }
      path.clear(); if (s >= 0) visit(s);
      if (path.size() != m/2 + 1) return {};
      return path;
   }
   vector<int> eulerian_tour() {
      int m = 0, s = 0;
      for (int u = 0; u < n; ++u) {
         m += adj[u].size();
         if (adj[u].size() > 0) s = u;
      }
      path.clear(); visit(s);
      if (path.size() != m/2 + 1 || path[0] != path.back()) return {};
      return path;
```

```
    }
};
```

4.2. Java

```
public List<Integer> findPathEuler(int[][] matrix_ady, int n) {
    int[] deg = new int[n]; Arrays.fill(deg, 0);
    List<Integer> path = new ArrayList<Integer>();
    path.add(-1);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j)
            deg[i] += matrix_ady[i][j];
    }
    int first = 0;
    while (first < n && deg[first]==0) ++first;
    if (first == n) return path;

    int v1 = -1, v2 = -1; boolean bad = false;
    for (int i = 0; i < n; ++i) {
        if ((deg[i] & 1) == 1) {
            if (v1 == -1) v1 = i;
            else if (v2 == -1) v2 = i;
            else bad = true;
        }
    }

    if (v1 != -1) {
        ++matrix_ady[v1][v2];
        ++matrix_ady[v2][v1];
    }

    Stack<Integer> st = new Stack<Integer>();
    st.push(first);
    List<Integer> res = new ArrayList<Integer>();
    while (!st.empty()) {
        int v = st.peek(); int i;
        for (i = 0; i < n; ++i)
            if(matrix_ady[v][i] != 0) break;
        if(i == n){res.add(v);st.pop();}
        else {
            --matrix_ady[v][i]; --matrix_ady[i][v];
            st.push(i);
        }
    }
    if(v1 != -1) {
        for(int i = 0; i + 1 < res.size(); ++i) {
            if((res.get(i) == v1 && res.get(i + 1) == v2) || (res.get(i) == v2 &&
                res.get(i + 1) == v1)) {
```

```
            List<Integer> res2 = new ArrayList<Integer>();
            for(int j = i + 1; j < res.size(); ++j)
               res2.add(res.get(j));
            for (int j = 1; j <= i; ++j)
               res2.add(res.get(j));
            res.clear();
            res.addAll(res2);
            break;
         }
      }
   }

   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         if (matrix_ady[i][j] != 0) bad = true;
      }
   }

   if (bad) { return path;}
   else {
      path.clear();
      for (Integer x : res) path.add(x);
   }
   return path;
}
```

Esta implementación esta pensada para multigrafo no dirigido con bucles

La siguiente implementación halla el ciclo de Euler en un grafo dirigido asumiendo que todos los nodos sus grados de entrada y salida son iguales y el grafo es conexo. Al método se le pasa el grafo y el nodo por el cual quiere comenzar el ciclo.

```
public static List<Integer> eulerCycle(List<Integer>[] graph, int v) {
   int[] curEdge = new int[graph.length];
   List<Integer> res = new ArrayList<>();
   Stack<Integer> stack = new Stack<>();
   stack.add(v);
   while (!stack.isEmpty()) {
      v = stack.pop();
      while (curEdge[v] < graph[v].size()) {
         stack.push(v);
         v = graph[v].get(curEdge[v]++);
      }
      res.add(v);
    }
    Collections.reverse(res);
    return res;
}
```

5. Complejidad

La complejidad de todos estos algoritmos es obviamente lineal con respecto al número de aristas. Solamente el algoritmo de Hierholzer que realiza DFS su complejidad es O($N + N$) siendo N los vertices y M las aristas.

6. Aplicaciones

Damos aquí un problema de ciclo euleriano clásico: el problema de Domino.

Hay N fichas de dominó, como se le conoce, en ambos extremos de la ficha de dominó está escrito un número (generalmente del 1 al 6, pero en nuestro caso no es importante). Desea colocar todas las fichas de dominó en una fila de modo que coincidan los números de dos fichas de dominó adyacentes cualesquiera, escritos en su lado común. Las fichas de dominó pueden girar.

Reformular el problema. Sean los números escritos en la parte inferior los vértices del grafo, y las fichas de dominó las aristas grafo (cada ficha de dominó con números (a, b) son las aristas (a, b) y (b, a)). Entonces nuestro problema se reduce al problema de encontrar el camino Euleriano en este grafo.

Los caminos eulerianos se utilizan en bioinformática para reconstruir la secuencia de ADN a partir de sus fragmentos. También se utilizan en el diseño de circuitos semiconductor de óxido de metal complementario (*CMOS*) para encontrar un orden óptimo de puertas lógicas. Hay algunos algoritmos para procesar árboles que se basan en un recorrido de Euler por el árbol (donde cada arista se trata como un par de arcos).

7. Ejercicios propuestos

A continuación una lista de ejercicios que se pueden resolver utilizando los elementos abordados en esta guía:

- DMOJ - Animando las Vacas
- CSES - Mail Delivery
- CSES - Teleporters Path
- UVA - 10054 The Necklace