Experiência de programador

Joao Pedro Schmitt

Caminho/ciclo euleriano de um grafo simples

schmittjoaopedro . Java 9 09+00:00 Junho 09+00:00 2018 6 Minutes

Problema

Dado um grafo simples $G = (V, A)$ composto de V vértices e A arestas não disjunto, um caminho euleriano é um caminho que visita cada aresta uma única vez e um ciclo euleriano é um caminho que começa e termina no mesmo vértice. O grau de um vértice representa o número de arestas incidentes.

Para um grafo possuir caminho euleriano ele deve possuir exatamente dois vértices com grau ímpar. Em contraste, para um grafo possuir ciclo euleriano todas as arestas devem possuir grau par.

Como encontrar de forma eficiente, se existir, um caminho ou ciclo euleriano de um grafo?

Grafo

Antes de apresentar o algoritmo, as classes Java abaixo são usadas para modelar os grafos. Foi selecionado a implementação usando lista de adjacências e devido ao uso de HashMap todas operações de inserção, consulta e remoção de arestas podem ser feitas em tempo constante.

```
1  public class Graph {
2
3      private HashMap nodes = new HashMap();
4
5      public void addEdge(int i, int j) { } // O(1)
6
7      public Node getNode(int i) { } // O(1)
8
9      public List getAdjacent(int i) { } // O(1)
10
11     public void delEdge(Node i, Node j) { } // O(1)
12
13     public List getNodes() { } // O(1)
14
15     public int getSize() { } // O(1)
16 }
17
18 public class Node {
19
20     public int value;
21
22     public Map adjacency = new HashMap();
23
24     public int degree() { } // O(1)
25 }
```

Algoritmo de Fleury

O algoritmo de Fleury é um algoritmo de tempo polinomial capaz de determinar o caminho ou ciclo euleriano, se existir, para um grafo que atenda as propriedades apresentadas anteriormente na definição do problema. Ele fuciona com base no conceito “Don’t burn your bridges” (Não queime suas pontes). A seguir é apresentado o algoritmo juntamente da análise de complexidade de tempo e espaço. A apresentação do algoritmo é feita de baixo pra cima, começando da função mais especialista para a mais genérica.

Função: DFS

Para que o algoritmo de Fleury funcione e que o conceito de Não queime suas pontes seja efetivo, uma busca em profundidade (Depth-first Search – DFS) adaptada ao problema em questão é necessária. A busca DFS calcula a cobertura de um dado vértice. Basicamente, a cobertura indica quantos vértices podem ser alcançados partindo de uma determinada origem. O DFS adaptado é apresentado no algoritmo a seguir.

```java
public static int DFS(Set visited, Node from) {
    int count = 1; // O(1)
    visited.add(from); // O(1)
    Node[] nodes = from.adjacency.values().toArray(new Node[]{}); // O(1)
    for (Node to : nodes) { // O(degree(from))
         if (!visited.contains(to)) { // O(1)
             count = count + DFS(visited, to); // O(1)
         }
    }
    return count; // O(1)
}
```

O DFS geralmente lida com uma lista de vértices, porém no algoritmo de Fleury será necessário processar somente o vértice corrente para encontrar a quantidade de vértices cobertos, e por esse motivo o algoritmo é adaptado para esse cenário. Portanto, essa versão do DFS funciona da seguinte maneira: dado um vértice de origem from, um processo recursivo é iniciado para cada vértice adjacente de forma à contar a quantidade de vértices cobertos.

Para análisar essa algoritmo, denote como $|V|$ o número de vértices de G e $|A|$ o número de arestas de G, considerando um grafo simples em cada vértice nesse algoritmo será executada $|grau(V)|$ comparações. Assim, temos que:

$$\sum_{v \in V} |grau(v)| = 2|A| = O(|A|)$$

A somatória resulta em $2|A|$ porque dado um vértice i e um vértice j, existem duas arestas $(i,j)\, e\, (j,i)$, que vão ser encontradas uma vez no vértice i e uma vez no vértice j. Assim a complexidade de tempo é de $O(|A|)$ no pior caso. Detalhe, o uso do HashSet não influencia no custo, pois ele garante um tempo de consulta e inserção de nós visitado com complexidade constante. Adicionalmente, como o processo recursivo é aberto para cada vértice não visitado, temos que a complexidade de espaço é $O(|V|)$ no pior caso.

Função: isBridge

Dados dois vértices v_{from} e v_{to} a função isBridge verifica se a visita da aresta (v_{from}, v_{to}) não irá deixar o grafo disjunto. Essa verificação é necessária para garantir as propriedades do caminho e ciclo euleriano, assim a remoção de uma aresta não irá impossibilitar a visita dos próximos vértices não visitados.

```java
public static boolean isBridge(Graph graph, Node from, Node to) {
    if (from.adjacency.size() == 1) { // O(1)
         return false; // O(1)
    }
    int bridgeCount = DFS(new HashSet(), to); // O(|A|)
    graph.delEdge(from, to); // O(1)
    int nonBridgeCount = DFS(new HashSet(), to); // O(|A|)
    graph.addEdge(from.value, to.value); // O(1)
    return nonBridgeCount < bridgeCount; // O(1)
}
```

Nesse algoritmo, inicialmente é executado a DFS para determinar a quantidade de nós que podem ser alcançados a partir do nó origem v_{from} não considerando a remoção da aresta (v_{from}, v_{to}). Então é removido a aresta (v_{from}, v_{to}) e é executado o DFS para recalcular a quantidade de vértices cobertos sem essa aresta. Se a remoção da aresta reduz a cobertura dos vértices então essa aresta é ponte, o que significa que ela irá separar o grafo e tornar uma das partes inacessível.

A analise dessa função, como ela depende de duas DFS, tem custo no pior caso de complexidade de tempo de $O(|A|) + O(|A|) = O(|A|)$. Em relação a complexidade de espaço, como essa função possui consumo constante $O(1)$ a complexidade é determinada pela DFS, que é de $O(|V|)$.

Função: getEulerPath recursiva

A função getEulerPath é um procedimento recursivo para encontrar um caminho/ciclo a partir de um nó de origem. O algoritmo é apresentado a seguir:

```
1  public static void getEulerPath(Graph graph, List path, Node from) {
2      Node[] nodes = from.adjacency.values().toArray(new Node[]{}); // O(1)
3      for (Node to : nodes) { // O(grau(from))
4           if (!isBridge(graph, from, to)) { // O(|A|)
5               path.add(to); // O(1)
6               graph.delEdge(from, to); // O(1)
7               getEulerPath(graph, path, to); // O(1)
8               break; // O(1)
9           }
10     }
11 }
```

Esse algoritmo funciona da seguinte forma: dado um nó de origem v_{from}, se uma dada aresta (v_{from}, v_{to}) não é definida como ponte ou é a única aresta para sair de v_{from} então a mesma é removida do grafo e o registro de avanço para v_{to} é adicionado à lista path. Assim, garante-se que caminho reversos não serão feitos.

A análise do algoritmo pode ser feito da seguinte forma, em cada vértice existem $|grau(v_{from})|$ arestas, e a contagem de cada aresta é feita duas vezes, (v_{from}, v_{to}) e (v_{to}, v_{from}). No pior caso irá se percorrer todas as arestas duas vezes $2|A| = O(|A|)$. Invocando a função isBridge de custo $O(|A|)$ em cada iteração, resulta-se em uma complexidade de tempo de O(|A|) x O(|A|) =O(|A|^2). Em contraste, no melhor caso em que para cada vértice a primeira aresta da lista de adjacências sempre satisfaz a condição de não ser uma ponte e o circuito é fechado ao atingir o último vértice, teremos $|V| * O(|A|)$. Porém, como prevalece o pior caso, temos complexidade de tempo de O(|A|^2).

A análise da complexidade de espaço dessa função, por ser um processo recursivo e invocar uma DFS, no pior caso é calculado como $O(|A|) + O(|V|) = O(|A| + |V|)$. O pior caso ocorre quando todos as arestas são percorridas para fechar o caminho ou ciclo.

Função: getEulerPath

A última função getEulerPath, dado um grafo, retorna o caminho ou ciclo Euleriano se o grafo respeitar as propriedades de caminho ou ciclo Euleriano, caso contrário é retornado um erro. A seguir é apresentado o algoritmo:

```
1  public static List getEulerPath(Graph graph) throws Exception {
2      List path = new ArrayList(); // O(1)
3      int oddCount = getOddVerticesCount(graph); // O(|V|)
4      if (oddCount == 0) { // O(1)
5           path.add(graph.getNodes().get(0)); // O(1)
6           getEulerPath(graph, path, path.get(0)); // O(|A|^2)
7      } else if (oddCount == 2) {
8           path.add(getEulerPathOrigin(graph)); // O(|V|)
9           getEulerPath(graph, path, path.get(0)); // O(|A|^2)
10     } else {
11          throw new Exception("Euler properties infringed.");
12     }
13     return path; // O(1)
14 }
```

A função getOddVerticesCount retorna o número de vértices ímpar, essa função tem complexidade $O(|V|)$ porque é necessário validar o grau de todos os vértices. Em relação a busca do caminho/ciclo, caso o grafo possua um ciclo (todos os vértices tem grau par) é adicionado como origem o primeiro vértice (poderia ser qualquer vértice) para invocar a função getEulerPath. Na outra situação em que o grafo possui um caminho euleriano (possui dois vértices de grau ímpar) é adicionado o primeiro vértice de grau ímpar como origem no caminho e é invocado a função getEulerPath. No caso que o algoritmo não lança um erro, podemos calcular a complexidade de tempo da seguinte forma O(|V|) + O(|A|^2) = O(|A|^2) no pior caso. A complexidade de espaço é definida pela função getEulerPath recursiva, $O(|A| + |V|)$.

A implementação do algoritmo em Java está disponível no seguinte link: https://github.com/schmittjoaopedro/paa/blob/master/src/main/java/com/github/schmittjoaopedro/graph/FleuryAdjListAlgorithm.java (https://www.google.com/url?q=https://github.com/schmittjoaopedro/paa/blob/master/src/main/java/com/github/schmittjoaopedro/graph/FleuryAdjListAlgorithm.java&sa=D&ust=1525576866880000)

Referências

[1]"Fleury's Algorithm for printing Eulerian Path or Circuit – GeeksforGeeks", GeeksforGeeks, 2018. [Online]. Available: https://www.geeksforgeeks.org/fleurys-algorithm-for-printing-eulerian-path/ (https://www.geeksforgeeks.org/fleurys-algorithm-for-printing-eulerian-path/). [Accessed: 06- May- 2018].

[2]People.ku.edu, 2018. [Online]. Available: http://people.ku.edu/~jlmartin/courses/math105-F11/Lectures/chapter5-part2.pdf (http://people.ku.edu/~jlmartin/courses/math105-F11/Lectures/chapter5-part2.pdf). [Accessed: 06- May- 2018].

Com as etiquetas :
analise de algoritmos,
analise de complexidade,
caminho euleriano,
ciclo euleriano,
java

Publicado por schmittjoaopedro

Graduado como bacharel em Sistemas de Informação pelo Centro Universitário Católica de Santa Catarina campus Jaraguá do Sul. Formado no Ensino Médio pelo Senai com Técnico em Redes de Computadores Articulado. Atualmente desenvolvedor JEE/Web em Sistemas de Engenharia na WEG. Pesquisador no período de faculdade em Informática pela Católica de Santa Catarina. Contato 47 - 99615 2305 E-mail: schmittjoaopedro@gmail.com Web page: https://joaoschmitt.wordpress.com/ Linkedin: https://www.linkedin.com/in/joao-pedro-schmitt-60847470/ Curriculum lattes: http://lattes.cnpq.br/9304236291664423 Twitter: @JooPedroSchmitt Ver todos os artigos de schmittjoaopedro