

UNIVERSIDADE ESTADUAL DO MARANHÃO – UEMA
CENTRO CIÊNCIAS E TECNOLOGIA – CCT
Curso: Engenharia da Computação
Turma: 2020.2

Implementação do algoritmo de Fleury

Ciro Dourado de Oliveira
Ítalo Torres Lima
João Gabriel Pereira
Pedro Augusto Sousa Noleto

SÃO LUÍS – MARANHÃO
2021

1. INTRODUÇÃO

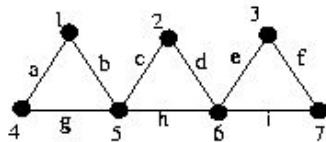
Grafo é uma estrutura matemática que é elaborada a partir de dois conjuntos: Um conjunto de vértices (pontos) ligados a partir de outro conjunto de arestas (linhas). Esses elementos, que juntos são chamados de grafo, são utilizados para a simplificação e/ou solução de problemas do mundo real, modelando-os de maneira em que um problema complexo possa se tornar mais fácil de ser visualizado.

Para que um grafo possa ser bem modelado e de fácil compreensão é necessário que esse possua uma quantidade de elementos finitos, ou seja, valores bem definidos e por isso esse tema é estudado na matemática e na computação sendo considerado como “discreto”. Porém a utilização de grafos não se resume a isso, como explicado antes, os grafos são muito utilizados para resolver problemas em geral, sendo esses, da vida real ou problemáticas computacionais.

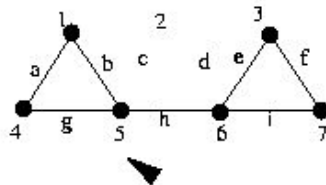
Para exemplificar, na área computacional são utilizados em estruturas de dados, grafos de árvores binárias que são muito utilizados na explicação para estruturação de informações armazenadas e acessadas pelo computador. Já no mundo real, os grafos são utilizados para qualquer problema que envolva algum tipo de relacionamento entre os valores que tenham sido descritos, como por exemplo um problema de tráfego em uma cidade ou avenida.

2. DESCRIÇÃO ACERCA DO ALGORITMO DE FLEURY

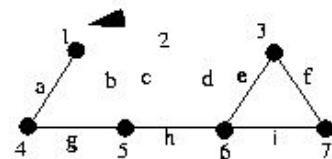
É um algoritmo de tempo polinomial que pode criar ou determinar o ciclo euleriano, caso esse exista e o grafo atenda as propriedades necessárias. O algoritmo de Fleury tem como base a ideia de não queimar as suas pontes. Para melhor exemplificar, temos as imagens abaixo, que demonstram a funcionalidade do algoritmo:



(a)



(b)



(c)

Nesse exemplo o algoritmo começará no vértice 6 onde ele tem um leque de quatro outros vértices que ele possa ir, então ele escolhe ir para o vértice 2, lá ele só tem a possibilidade de utilizar a ponte da aresta c que lhe lança para o vértice 5, lá ele tem como oportunidade 3 outros vértices para ir, mas o caminho h é uma ponte então ele o dispensa e segue pela aresta b e chega no vértice 1, a partir desse ponto ele irá andar mais três etapas e voltará a origem, completando assim o circuito euleriano.

3. IMPLEMENTAÇÃO DO ALGORITMO

A linguagem Rust foi escolhida para a implementação do Algoritmo de Fleury. Trata-se de uma linguagem multiparadigma, ou seja, com ela, é possível programar em diversos métodos de desenvolvimento. Um dos pontos fortes dessa linguagem trata-se da possibilidade de utilizar-se dos paradigmas funcional, orientado a objetos e imperativo. Por ser uma linguagem eficiente e possuir bibliotecas específicas para grafos, o Rust mostrou-se de extrema eficiência para a implementação do código para a análise gráfica.

Em termos específicos acerca da linguagem, a biblioteca escolhida foi a *petgraph*, que fornece ferramentas úteis para a implementação e análise de grafos. Com ela, é possível atribuir valores para as arestas e para os vértices de forma arbitrária, além da possibilidade de trabalhar-se com arestas direcionadas ou não.

Primeiramente, importa-se a biblioteca utilizando-se dos comandos encontrados na documentação da biblioteca em Rust. Destaca-se o fato de que, ao importar a biblioteca, é necessário explicitar o tipo de grafo que será utilizado ao longo do código, no caso do algoritmo implementado, serão utilizados grafos não-direcionados, então, para isso, o comando *UnGraph* (encontrado em algumas linhas de código) realiza a representação de um grafo sem direção específica.

Ao realizar este procedimento, implementou-se a primeira função: obter os vértices de grau ímpar - vértices que possuem um número ímpar de arestas saindo ou entrando - essa função é essencial para o algoritmo de Fleury, pois os Grafos Eulerianos possuem 0 ou 2 vértices de grau ímpar.

Essa função obtém o índice de cada vértice em um vetor de vértices e analisa um por um. Se o resto da divisão do número do grau por 2 for diferente de 0, o vértice será considerado de grau ímpar. Posteriormente, ao diferenciar os vértices pares dos ímpares, implementou-se outra função para obter o valor exato do número de vértices de grau ímpar através da função nativa do Rust *len()*.

A terceira função implementada é do tipo booleano e consiste em verificar se o grafo passado pelo usuário ao programa possui um Caminho Euleriano ou não através de duas condições:

- O grafo precisa ser cíclico e não-direcionado;
- O número de graus ímpares precisa ser 0 ou 2 para retornar *true*, caso contrário retornará *false*.

Ambas as condições precisam ser satisfeitas simultaneamente pelo grafo. É de extrema importância o fato de que um grafo de 2 vértices ímpares, para ter um Caminho Euleriano, deve começar em qualquer vértice de grau ímpar e terminar o caminho em outro.

O próximo passo consiste em imprimir a trilha que será seguida no grafo e o algoritmo de Fleury propriamente dito. Para isso, foi implementada outra função que utiliza-se das funções anteriormente especificadas. Ela define o vértice inicial da trilha e o seu posterior, removendo os outros vértices para que não sejam exibidos.

O algoritmo imprime sempre dois vértices ligados por uma aresta até que o caminho seja, finalmente, percorrido por completo. Na última impressão será mostrado o último vértice e o seu antecessor.

Finalmente, na função *main* um grafo vazio é declarado e, posteriormente são criadas variáveis, cada uma representando um vértice. Cada vértice recebe um valor, no caso do algoritmo, serão valores inteiros de 1 a 4, essa atribuição é feita pela função *add_node()* nativa da biblioteca. Após isso, as arestas são adicionadas através da função *add_edge()*. Essa função recebe 3 parâmetros: os dois primeiros recebem os vértices conectados pela aresta, e o último recebe um valor qualquer para a aresta, nesse caso uma *string* do tipo “vértice1 - vértice2”. Para finalizar, aplica-se a função na qual o algoritmo de Fleury foi implementada, será imprimido toda a trilha feita no Grafo Euleriano.

4. VALIDAÇÃO DA IMPLEMENTAÇÃO

Consideremos um grafo com 6 vértices e 8 arestas, como está na figura 4.1: o qual possui exatamente 2 vértices de grau ímpar (ou seja, se liga a um número ímpar de outros nós), e o restante de grau par - os nós 2 e 1 representam os vértices de grau ímpar, onde o 2 se liga com os vértices 1, 3 e 5; e o 1 se liga com 2, 3, 4, 5 e 6.

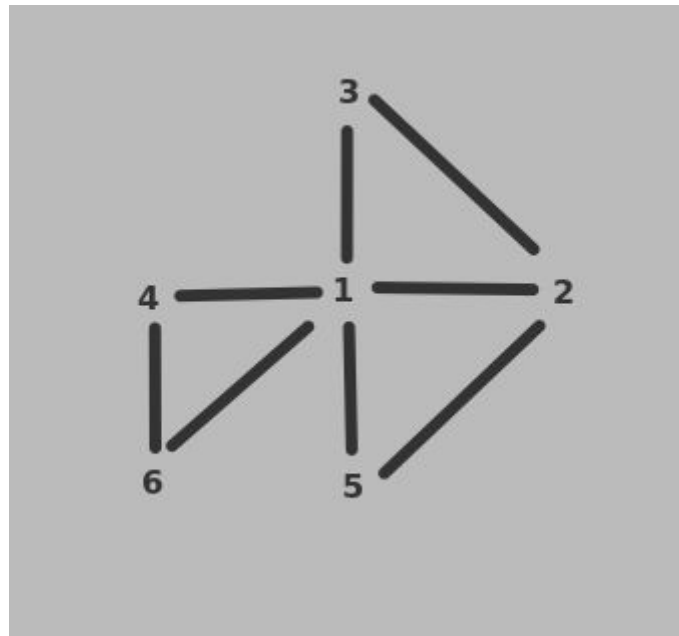


Figura 4.1 - Grafo usado de exemplo

Essa observação é importante pois mais para frente será demonstrado que é uma das etapas para que o programa decida por onde começará a aplicar o algoritmo, já que, caso se comece por um ponto qualquer, pode acontecer de nem todas as arestas serem percorridas, o que nos faria concluir que o algoritmo é falho em sua teoria.

Depois de ser sido instanciada uma estrutura (observe a main, figura 4.2) idêntica ao grafo exemplificado, ela deve ser passada para a função que aplicará ou não o algoritmo; e por não aplicar, deve-se entender que os grafos passíveis de serem testados tem certos pré-requisitos: ele deve possuir um caminho Euleriano, o que significa que o grafo em questão deve apresentar ao mesmo tempo um ciclo entre os nós, e um número limitado de nós de grau ímpar (no máximo 2, ou nenhum).

```

fn main() {
    let mut _graph = UnGraph::new_undirected();

    let node1 = _graph.add_node(1);
    let node2 = _graph.add_node(2);
    let node3 = _graph.add_node(3);
    let node4 = _graph.add_node(4);
    let node5 = _graph.add_node(5);
    let node6 = _graph.add_node(6);

    _graph.add_edge(node1, node2, "1-2");
    _graph.add_edge(node1, node3, "1-3");
    _graph.add_edge(node1, node4, "1-4");
    _graph.add_edge(node1, node5, "1-5");
    _graph.add_edge(node1, node6, "1-6");
    _graph.add_edge(node2, node3, "2-3");
    _graph.add_edge(node2, node5, "2-5");
    _graph.add_edge(node4, node6, "4-6");

    apply_fleury(&_graph);
} // end main

```

Figura 4.2 - Instanciação do grafo

```

fn apply_fleury(graph: &UnGraph) {
    //clear_terminal();

    match has_an_eulerian_path(graph) {
        true => fleury_trail(&mut graph.clone()),
        false => print!("Unable to apply!\n")
    }
} // end apply_fleury

```

Figura 4.3 - Condição para a aplicação do algoritmo

```
fn has_an_eulerian_path (graph: &UnGraph) -> bool {
  let condition_one = is_cyclic_undirected(graph);
  let condition_two = match count_odd_degree_nodes(graph) {
    0 | 2 => true,
    _ => false
  };
  condition_one && condition_two
} // end has_an_eulerian_path
```

Figura 4.4 - Condições associadas ao caminho Euleriano

Quando a condição principal é satisfeita, então o algoritmo pode ser aplicado. Um nó deste grafo deve ser selecionado com base em um padrão: caso haja graus ímpares, o vértice de menor grau é o ponto de partida para se percorrermos as arestas; caso não haja graus ímpares, qualquer vértice é selecionado (no código, o primeiro vértice adicionado foi arbitrariamente escolhido).

```
fn apply_fleury(graph: &UnGraph) {
  //clear_terminal();

  match has_an_eulerian_path(graph) {
    true => fleury_trail(&mut graph.clone()),
    false => print!("Unable to apply!\n")
  }
} // end apply_fleury

fn fleury_trail(graph: &mut UnGraph) {
  let mut node_a = fleury_start_point(graph);

  loop {
    let node_b = match edges_from(node_a, graph).get(0) {
      Some(node) => *node,
      None => break
    };

    let a_b = graph.find_edge(node_a, node_b);
    graph.remove_edge(a_b.unwrap());

    print!("{}", node_a, node_b, "\n",
      graph.node_weight(node_a).unwrap(),
      graph.node_weight(node_b).unwrap());

    node_a = node_b;
  }
} // end fleury_trail
```

Figura 4.5 - passada na condição principal, a primeira coisa a ser feita é encontrar o nó inicial (fleury_start_point)


```
fn fleury_start_point(graph: &UnGraph) -> NodeIndex {
    match count_odd_degree_nodes(graph) {
        2 => lowest_odd_degree(graph),
        _ => first_node(graph)
    }
} // end fleury_start_point
```

Figura 4.6 - escolha baseada em quantos nós de grau ímpar existem.
 Caso haja, deve ser o nó de menor grau ímpar.
 Caso não, o primeiro nó inserido é o escolhido

Então, o programa começa a percorrer cada aresta, removendo sempre que se passa por ela. Como a escolha das arestas é indiferente, em termos de código, nada foi feito para que se seleccionasse algum caminho específico, o primeiro encontrado é o percorrido.

```
let mut node_a = fleury_start_point(graph);

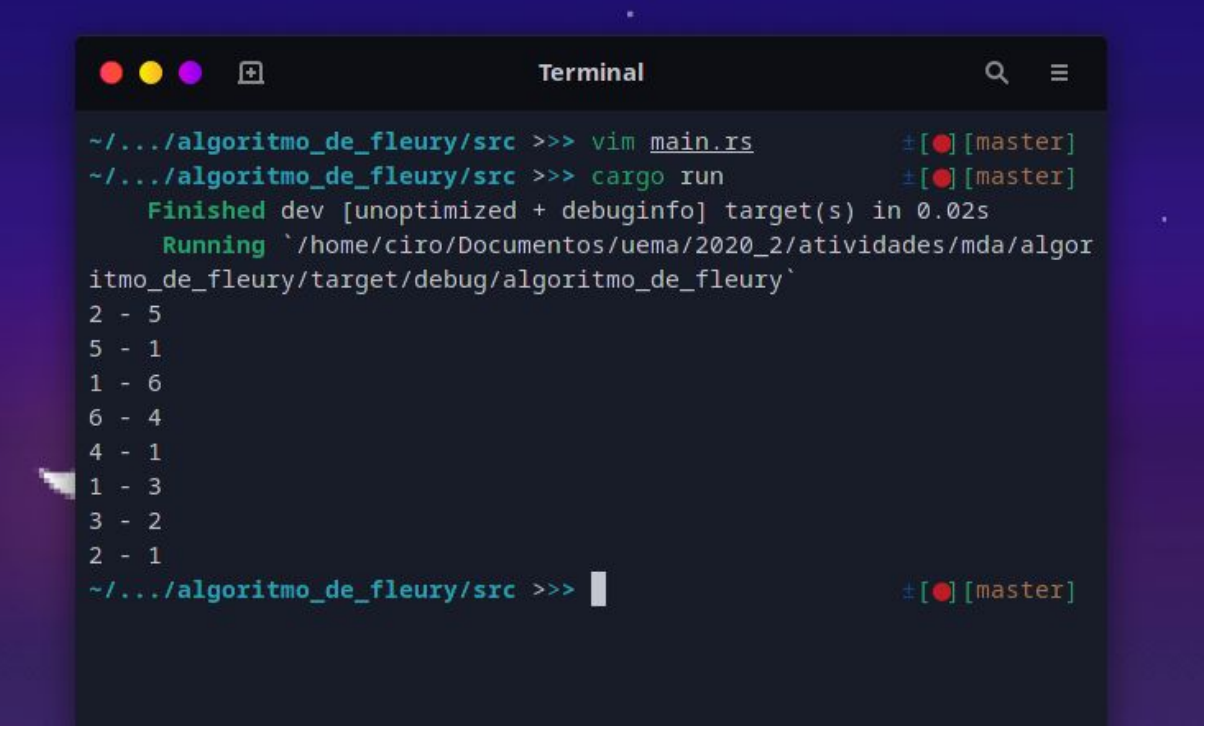
loop {
    let node_b = match edges_from(node_a, graph).get(0) {
        Some(node) => *node,
        None => break
    };

    let a_b = graph.find_edge(node_a, node_b);
    graph.remove_edge(a_b.unwrap());

    print!("{}", node_a, node_b, "\n",
        graph.node_weight(node_a).unwrap(),
        graph.node_weight(node_b).unwrap());

    node_a = node_b;
}
```

Figura 4.7 - loop onde os nós são seleccionados para serem percorridos.
 As arestas, como mostrado, sempre são as que estão entre os nós "A" e "B"

A terminal window titled "Terminal" with standard macOS window controls (red, yellow, green buttons and a zoom icon). The terminal shows the following commands and output:

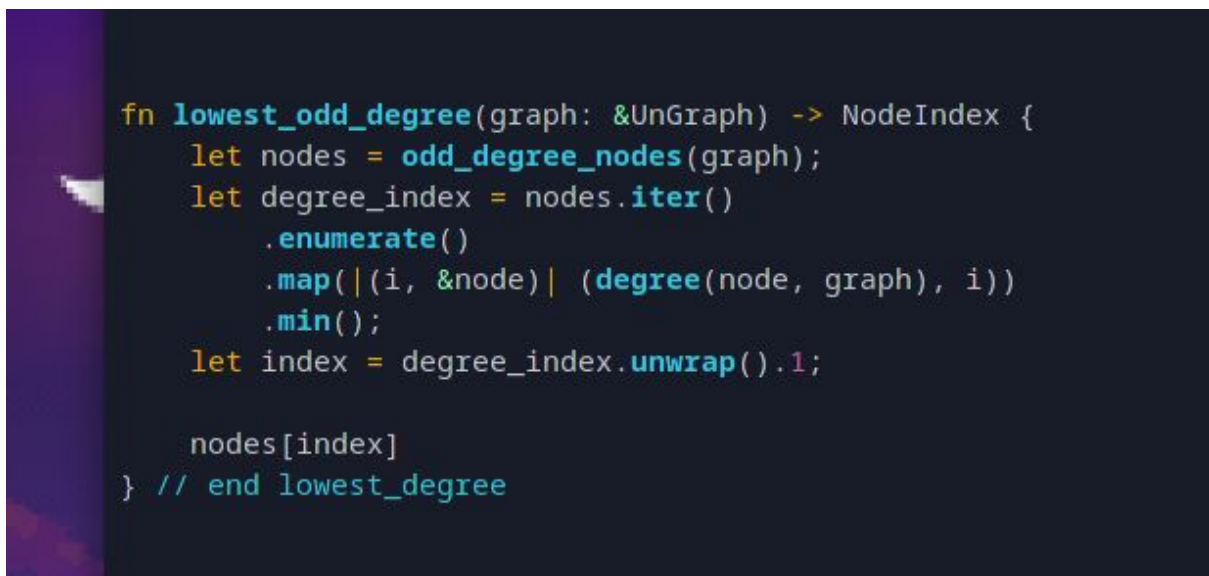
```
~/.../algoritmo_de_fleury/src >>> vim main.rs ±[●][master]
~/.../algoritmo_de_fleury/src >>> cargo run ±[●][master]
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
    Running `/home/ciro/Documentos/uema/2020_2/atividades/mda/algoritmo_de_fleury/target/debug/algoritmo_de_fleury`
2 - 5
5 - 1
1 - 6
6 - 4
4 - 1
1 - 3
3 - 2
2 - 1
~/.../algoritmo_de_fleury/src >>> ±[●][master]
```

Figura 4.8 - resultado da aplicação do algoritmo

5. ANÁLISE E DISCUSSÕES

O algoritmo funciona como deveria, observando todo o código construído em cima de sua base teórica. Há um porém que deve ser apontado: durante as revisões feitas antes da fase de desenvolvimento, nada foi citado quanto à escolha de qual dos nós de grau ímpar deveria ser o inicial, visto que em versões anteriores à mostrada aqui neste trabalho, o programa não funcionava como deveria em certos casos.

O padrão observado foi que, em certos grafos testados, quando o nó de menor grau ímpar era ignorado, nem todas as arestas eram percorridas, pois quando se chegava nele não haviam mais caminhos válidos, fazendo o programa parar. Por conta disso foi criada uma função específica para filtrar qual dos nós é o correto.

A screenshot of a code editor with a dark background and light-colored text. The code is written in Rust and defines a function named 'lowest_odd_degree'. The function takes a 'graph' of type '&UnGraph' as an argument and returns a 'NodeIndex'. The function's body includes: 1. A 'let' statement to get 'nodes' from 'odd_degree_nodes(graph)'. 2. A 'let' statement to create 'degree_index' from 'nodes.iter().enumerate().map(|(i, &node)| (degree(node, graph), i)).min()'. 3. A 'let' statement to get 'index' from 'degree_index.unwrap().1'. 4. A final line returning 'nodes[index]'. The function is enclosed in a block starting with 'fn' and ending with '}' and a comment '// end lowest_degree'.

```
fn lowest_odd_degree(graph: &UnGraph) -> NodeIndex {  
    let nodes = odd_degree_nodes(graph);  
    let degree_index = nodes.iter()  
        .enumerate()  
        .map(|(i, &node)| (degree(node, graph), i))  
        .min();  
    let index = degree_index.unwrap().1;  
  
    nodes[index]  
} // end lowest_degree
```

Figura 5.1 - Função responsável por sempre retornar o nó de menor grau ímpar

Seu funcionamento nada mais é do que: receber uma lista de todos os nós de grau ímpar, criar uma variável que associa cada grau dos nós com um índice específico (para que se possa acessar diretamente da lista ao fim), dessa variável extrair apenas o nó que possui o menor grau, e por fim, a partir da mesma, apenas acessar o índice relacionado. O retorno, como foi dito, é apenas o acesso do nó específico pelo índice encontrado.

6. CONCLUSÕES E TRABALHOS FUTUROS

Dada a observação da seção anterior deste trabalho, não há nada mais a ser acrescentado sobre problemas ou resultados, visto que funciona como deveria com os remendos feitos. A única observação a ser deixada aqui é sobre a eficiência; várias fontes online apontam sobre este algoritmo ser lento demais para que o caminho seja traçado, pois certas implementações requerem a constante marcação e verificação de quais caminhos são válidos de serem percorridos.

Em relação ao processo de escrita do código, foi uma ótima experiência, pois pudemos constatar que Rust, apesar de ser uma linguagem um tanto difícil de domar, é bastante flexível, e conta com boas bibliotecas prontas para a manipulação de estruturas como grafos. Neste trabalho, apenas funções básicas foram usadas, todo o aparato em volta do algoritmo de Fleury foi feito do zero com muito esforço e dedicação.

Palavras do desenvolvedor: eu usaria Rust novamente em trabalhos futuros, por ser uma linguagem extremamente versátil e moderna, com uma vasta documentação e comunidade ativa para ajudar a tirar dúvidas e dar um norte no desenvolvimento das coisas. Apesar de ser de baixo nível, diferentemente do C, muitos problemas sobre tratamento de erros ou falhas de memória são evitados, sem contar na facilidade de usar outros módulos (ou crates, na gramática Rust) ou encontrar ferramentas prontas para resolver problemas.