

# Algoritmo de Fleury Algoritmos de Busca Algoritmo de Dijkstra

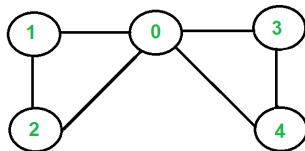
Vinicius Gasparini  
Filipe Cattoni

17/05/2019

# Ciclo Euleriano

Um grafo não direcionado possui um ciclo Euleriano se as duas condições forem verdadeiras.

- Todos os vértices com graus diferentes de zero estão conectados.
- Todos os vértices têm grau par.

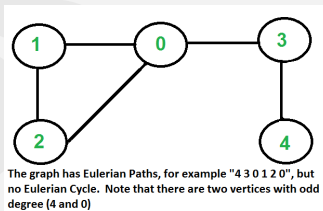


The graph has Eulerian Cycles, for example "2 1 0 3 4 0 2"  
 Note that all vertices have even degree

# Caminho Euleriano

Um grafo não direcionado possui Caminho Euleriano se as duas condições forem verdadeiras.

- Todos os vértices com graus diferentes de zero estão conectados.
- Se dois vértices tiverem um grau ímpar e todos os outros vértices tiverem um grau par.



# Implementação

```
int is_eulerian() {
    /* The function returns one of the following answers
       0 --> If graph is not Eulerian
       1 --> If graph is Semi-Eulerian (has a Euler path)
       2 --> If graph is Eulerian (has a Euler cycle) */

    if (is_connected() == false)
        return 0;

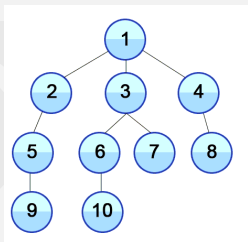
    int odd = 0;
    for (auto vertex:edges){
        if (vertex_degree(vertex) & 1)
            odd++;
    }
    if (odd == 1 or odd > 2)
        return 0;
    else if (odd == 2)
        return 1;
    return 2;
}
```

# Implementação

```
bool is_connected() {  
    bool visited[n_nodes] = {false};  
    int i;  
    for (i = 0; i < n_nodes; i++)  
        if (edges[i].size() != 0)  
            break;  
    if (i == n_nodes)  
        return true;  
  
    DFS(i, visited);  
  
    for (i = 0; i < n_nodes; i++){  
        if (visited[i] == false and edges[i].size() > 0) return false;  
    }  
    return true;  
}
```

# BFS

Na Busca em Largura (em inglês *Breadth-First Search*) o algoritmo começa pelo vértice raiz e explora todos os vértices vizinhos deste. Para cada um desses vértices visitados, exploramos os seus vértices vizinhos ainda não visitados e assim por diante, até que ele encontre o alvo da busca ou percorra o grafo todo.

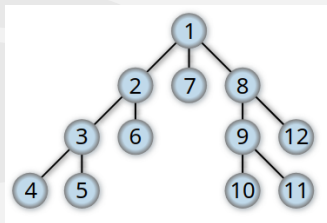


# Implementação

```
void BFS(int root) {  
    bool visited[this->size()] = {false};  
    queue<int> queue;  
    visited[root] = true;  
    queue.push(root);  
    while(!queue.empty()) {  
        root = queue.front();  
        queue.pop();  
        for (auto adj:edges[root]) {  
            if (!visited[adj.d]) {  
                visited[adj.d] = true;  
                queue.push(adj.d);  
            }  
        }  
    }  
}
```


# DFS

Na busca em profundidade (em inglês *Depth-First Search*) o algoritmo começa num nó raiz e explora tanto quanto possível cada um dos seus ramos, antes de retroceder. Isto é, a partir do nó raiz, visita-se um de seus vizinhos, a partir deste vizinho visita outro vizinho. Caso não possua nenhum vértice adjacente ou que não tenha sido visitado, a busca retorna ao nó anterior.





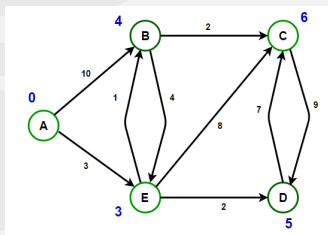
# Implementação



```
void DFS(int root, bool visited[]) {  
    visited[root] = true;  
    for (auto edge:edges[root]){  
        if (!visited[edge.d]) DFS(edge.d, visited);  
    }  
}
```

# Caminho Mínimo

O algoritmo de Dijkstra é um algoritmo usado para encontrar a menor distância entre vértices no grafo. Ele funciona de forma muito parecida aos algoritmos de busca, especificamente o BFS, porém mantém um vetor de distâncias do vértice raiz que é atualizado durante o percurso com o menor peso acumulado.



# Implementação

```
vector<int> dijk(int root) {  
  
    queue<int> queue;  
    vector<int> dist(n_nodes, INT_MAX);  
    vector<int> visited(n_nodes, 0);  
  
    dist[root] = 0;  
    visited[root] = 1;  
    queue.push(root);  
  
    while (!queue.empty()) {  
  
        int cur = queue.front();  
        queue.pop();  
  
        for (auto adj:edges[cur]) {  
            dist[adj.d] = min(dist[adj.d], dist[cur]+adj.w);  
            if (visited[adj.d] == 0) {  
                visited[adj.d] = 1;  
                queue.push(adj.d);  
            }  
        }  
    }  
  
    return dist;  
}
```