Ciclo Euleriano

Relatório do Trabalho Final

Análise e Síntese de Algoritmos

Docente: João Patrício

Aluno: João Victor do Rozário Recla

22 de janeiro de 2024

Conteúdo

1 Introdução

1.1 Trabalho Final

Dentre as opções de tema para a realização do trabalho final da disciplina de **Análise e Síntese de Algoritmos**, está o tema **Ciclo Euleriano**. Para essa atividade tem-se que identificar um ciclo euleriano num grafo, com base no algoritmo visto em aula, e realizar uma análise e discussão do desempenho do algoritmo desenvolvido nos grafos de entrada, que devem ser gerados com um dado número de vértices.

1.2 Desenvolvimento

A linguagem de programação escolhida para a implementação dos algoritmos, e desenvolvimento do trabalho, foi o **C++**. A estrutura de grafos foi representada, computacionalmente, por meio de listas de adjacência, e construída numa **classe** com os métodos necessários para sua utilização. Para identificar o **Ciclo Euleriano** num grafo, outra **classe** foi desenvolvida contendo algoritmos de percurso e verificações em grafos, utilizados para identificar um **Grafo Euleriano**.

2 Algoritmos

2.1 Implementação do Menu

A partir da **main** é chamado um menu para auxiliar na criação de um grafo com **N** vértices. É possível escolher, por meio de outro menu, entre gerar um grafo totalmente aleatório ou informar as arestas, dado um valor **M**.

Após a criação do grafo é verificado se ele é **conexo** e se todos os seus vértices possuem **grau par**, condições necessárias para um grafo ser euleriano e admitir um **Tour de Euler**. Caso o grafo seja euleriano ele é percorrido em ciclos, usando o algoritmo de **FLEURY**, até que um **Tour de Euler** seja montado (O tour se inicia a partir do vértice inicial).

2.1.1 main()

```
/*  Main.    */
//============
int main(){

    Criar_Grafo();
    Ciclo_Euleriano *Ciclo = new Ciclo_Euleriano(G);

    // Verifica se o grafo eh um grafo de euler.
    //  * (Se eh Conexo e se todos os vertices tem grau par).
    if(Ciclo->Verificar_Grafo_Conexo_() && Ciclo->Verificar_Graus_Pares_()){

        Ciclo->Tour_Euler___FLEURY_();  // Encontra o tour de euler no grafo.
        Ciclo->Imprimir_Tour_Euler_();  // Imprime o tour.
    }
    else cout << "\n|-| O GRAFO NAO EH EULERIANO." << endl;

    delete(Ciclo);
    return 0;
}
```

Codigos/Menu.cpp

2.1.2 MENU: Criar_Grafo()

```
void Criar_Grafo(){

    ll V, W;     // Vertices.
    ll N, M;     // Numero de vertices e arestas.

    // Tamanho do grafo.
    cout << "\n|--| Informe a quantidade de vertices do grafo:  ";
    cin >> N;
    G = new Grafos(N);

    // Escolha do grafo.
    if(Escolher_Grafo_Aleatorio() == true)
        G->Gerar_Grafo_Aleatorio();
    else{
        // Numero de arestas.
        cout << "\n\n|--| Informe a quantidade de arestas do grafo: ";
        cin >> M;

        // Loop: Insercao dos vertices.
        while(M-- > 0){
            cout << "\n|--| ================================ (0" << M+1 << ")" <<
                endl;
            cout << "|--| Informe o vertice de partida:    ";
            cin >> V;
            cout << "|--| Informe o vertice de chegada:    ";
            cin >> W;

            // Verifica se os vertices estao dentro dos limites.
            if((V <= G->QV) && (W <= G->QV) && (V > 0) && (W > 0)){
                V--; W--;   // Indexacao.
                G->Inserir_Aresta_(V, W);
                G->Inserir_Aresta_(W, V);
            }
            else{
                M++;
                cout << "|--| [INTERVALO DE VERTICES INVALIDO !]" << endl;
            }
        }
    }

    // Imprime o grafo criado.
    G->Imprimir_Grafo_();
}
```

Codigos/Menu.cpp

2.1.3 MENU: Escolher_Grafo()

```
bool Escolher_Grafo_Aleatorio(){

    string Escolha;
    bool Grafo_Aleatorio = false;

    while(1){
        // Escolha.
        cout << "\n|--| Deseja gerar um grafo aleatorio ? [SIM / NAO]" << endl;
        cout << "|--| ";
        cin >> Escolha;

        // Decisoes.
        if((Escolha == "SIM") || (Escolha == "sim")){
            Grafo_Aleatorio = true;
            break;
        }
        if((Escolha == "NAO") || (Escolha == "nao"))
            break;
    }
    return Grafo_Aleatorio; // Resultado.
}
```

Codigos/Menu.cpp

2.2 Classe de Grafos

A estrutura escolhida para representar os grafos, computacionalmente, foi a lista de adjacência, implementada utilizando-se uma **struct** como lista encadeada para os vértices adjacentes. Os grafos gerados são criados e gerenciados a partir de uma **classe** construida com os métodos necessários.

2.2.1 Estrutura dos Vértices

```
/*  Estrutura de um Vertice.    */
typedef struct Vertice{

    ll No;          // Identificador do vertice.
    ll Grau;        // Grau do vertice.
    set<ll> Adj;    // Lista de adjacencia do vertice.

    // Construtor.
    Vertice(ll Indicador = 0){

        this->No = Indicador;
        this->Grau = 0;
    };
}Vertice;
```

Codigos/Grafos.cpp

2.2.2 Estrutura dos Grafos

```
/*  Classe de grafos.   */
class Grafos {

    public: ll QV;  // Quantidade de vertices.
    public: ll QE;  // Quantidade de arestas.
    Vertice *Vert;  // Vetor de vertices.


    /*  Metodo responsavel por criar e
        inicializar a estrutura do grafo.   */
    //=======================================
    public: Grafos(ll V){

        // Inicializacao das variaveis.
        this->QV   = V;
        this->QE   = 0;
        this->Vert = new Vertice[V]();

        if(this->Vert)
            for(ll i = 0; i < this->QV; i++)
                this->Vert[i].No = i;   // Identificacao dos vertices (Para imprimir).
        else SEM_MEMORIA
    }
```

Codigos/Grafos.cpp

2.2.3 MÉTODO: Gerar_Grafo_Aleatorio()

```
    /*  Metodo para gerar um grafo
        completamente aleatorio.    */
    //================================
    public: void Gerar_Grafo_Aleatorio(){

        srand(time(NULL));                      // Semente para geracao de numeros
            aleatorios.

        ll V, W;
        ll E = (this->QV * (this->QV-1)) / 2;   // Numero maximo de arestas.
        E = rand() % E +1;                      // Numero definitivo de arestas.

        // Geracao aleatoria dos vertices do grafo.
        while(E != -1){

            V = rand() % this->QV;  // Vertice de partida.
            W = rand() % this->QV;  // Vertice de chegada.

            Inserir_Aresta_(V, W);  // Insercao da aresta (V, W).
            Inserir_Aresta_(W, V);  // Insercao da aresta (W, V).

            if(V != W)  E--;
        }
    }
```

Codigos/Grafos.cpp

2.2.4 MÉTODO: Inserir_Aresta()

```
    /*  Metodo responsavel por inserir
        uma aresta (V, W) no grafo. */
    //================================
    public: void Inserir_Aresta_(int V, int W){

        if(V != W){

            set<ll> *Aux = &(this->Vert[V].Adj);

            // Se o vertice W nao existe na lista.
            if((*Aux).find(W) == (*Aux).end()){

                this->QE++;             // Atualiza a quantidade de arestas no grafo.
                (*Aux).insert(W);       // Insere W na lista de adjacencia de V.
                this->Vert[V].Grau++;   // Atualiza o grau do vertice V.
            }
        }
    }
```

Codigos/Grafos.cpp

2.2.5 MÉTODO: Remover_Aresta()

```
    /*  Metodo responsavel por remover
        uma aresta (V, W) do grafo. */
    //================================
    public: void Remover_Aresta_(int V, int W){

        set<ll> *Aux = &(this->Vert[V].Adj);

        // Se o vertice W existe na lista.
        if((*Aux).find(W) != (*Aux).end()){

            this->QE--;             // Atualiza a quantidade de arestas no grafo.
            (*Aux).erase(W);        // Remove W da lista de adjacencia de V.
            this->Vert[V].Grau--;   // Atualiza o grau do vertice V.
        }
    }
```

Codigos/Grafos.cpp

2.2.6 MÉTODO: Imprimir_Grafo()

```
    /*  Metodo responsavel por imprimir a estrutura
        do grafo exibindo as listas de adjacencias. */
    //=================================================
    public: void Imprimir_Grafo_(){

        cout << endl;
        cout << "|-| ============= |-|" << endl;
        cout << "|-| Grafo G:      |-|" << endl;
        cout << "|-| ============= |-|" << endl;
        for(ll V = 0; V < this->QV; V++){

            // Imprime o vertice de partida, V.
            cout << "    " << V+1 << ":";

            // Loop: Imprime a lista de adjacencia de V.
            for(auto& VertAdj: this->Vert[V].Adj)
                cout << " " << VertAdj +1;
            cout << endl;
        }
        cout << "|-| ============= |-|" << endl;
    }
```

Codigos/Grafos.cpp

2.2.7 MÉTODO: Deletar_Grafo()

```
    /*  Metodo responsavel por deletar a estrutura
        do grafo removendo as listas de adjacencias.    */
    //=================================================
    public: ~Grafos(){

        // Deleta as listas de adjacencias de cada vertice do Grafo.
        for(ll V = 0; V < this->QV; V++)
            this->Vert[V].Adj.clear();
    }
};
```

Codigos/Grafos.cpp

2.3 Ciclo Euleriano

Para encontrar um **Tour de Euler** basta percorrer um **Grafo Euleriano**, a partir de um vértice inicial, passar por todas as arestas sem repeti-las e retornar ao vértice de partida.

Isso pode ser realizado através do percurso em profundidade que acaba por terminar com um ciclo, num grafo **conexo**. Utilizando apenas esse método é possível que o grafo não seja totalmente explorado, pois o ciclo encontrado, sem repetição de arestas, pode pertencer a um subgrafo.

Uma solução seria remover, do grafo original, o subgrafo que gerou o ciclo e refazer o percurso até que outro ciclo seja encontrado, repetindo o processo até que todas as arestas do grafo sejam exploradas.

O algoritmo de **FLEURY** é um método que constrói essa solução, se utilizando da definição de **pontes** em grafos para garantir que todas as arestas do grafo sejam exploradas, evitando becos.

2.3.1 Estrutura da Classe

```cpp
/*  Classe de algoritmos para resolver os
    problemas que envolvem ciclos eulerianos.   */
class Ciclo_Euleriano {

    private: ll K;              // Quantidade de vertices no 'subgrafo de euler' do
        grafo 'G'.
    private: ll Raiz;           // Indicador do vertice de partida do tour.
    public:  Grafos *G;         // G: Representacao computacional de um grafo.
    private: ll Ponte_v;        // Indicador de que um vertice 'V' eh extremidade de
        uma ponte.
    private: ll Ponte_w;        // Indicador de que um vertice 'W' eh extremidade de
        uma ponte.
    private: vector<ll> Tour;   // Vetor de vertices que representam o 'Tour de Euler'
        no grafo.


    /*  Construtor da classe.   */
    //============================
    public: Ciclo_Euleriano(Grafos *G, ll Inicio = 0){
        this->G = G;
        this->K = G->QV;
        this->Ponte_v = -1;
        this->Ponte_w = -1;
        this->Raiz = Inicio;
        this->Tour.push_back(Inicio);   // Vertice raiz do tour.
    }
```

Codigos/Ciclo_Euleriano.cpp

2.3.2 MÉTODO: Verificar_Graus_Pares_()

```cpp
    /*  Metodo para verificar se todos
        os vertices do grafo sao pares.
        (CONDICAO PARA UM GRAFO SER EULERIANO)  */
    //=============================================
    public: bool Verificar_Graus_Pares_(){

        ll i = this->G->QV;
        bool Vertices_Pares = true; // Indica que o grafo eh euleriano (inicialmente).

        while(--i > -1){

            // Verifica se algum vertice tem grau impar.
            if((this->G->Vert[i].Grau % 2) != 0){
                Vertices_Pares = false; // Indica que o grafo nao eh euleriano
                    (conclusao).
                break;
            }
        }

        return Vertices_Pares;
    }
```

Codigos/Ciclo_Euleriano.cpp

2.3.3 MÉTODO: Percurso_Profundidade_()

```cpp
    /*  Metodo para realizar o percurso
        em profundidade no grafo.    */
    //================================
    void Percurso_Profundidade_(ll V, ll &Alc, vector<ll> &Alcancados){

        Alcancados[V] = ++Alc;  // Indica que o vertice foi alcancados.

        // Loop: Percorre a lista de adjacencia de V.
        for(auto& W: this->G->Vert[V].Adj){

            // Verifica se o vertice W nao foi alcancado.
            if(Alcancados[W] == 0){

                /*  PONTES:
                    Uma aresta (V, W), ou (W, V), eh ignorada no percurso
                    em profundidade caso esteja indicada, como possivel
                    ponte, nas variaveis 'Ponte_v' e 'Ponte_w'.         */
                //=========================================================
                if(!(((Ponte_v == V) && (Ponte_w == W)) || ((Ponte_v == W) && (Ponte_w
                    == V))))
                    Percurso_Profundidade_(this->G->Vert[W].No, Alc, Alcancados);   //
                        Explora o grafo em profundidade.
            }
        }
    }
```

Codigos/Ciclo_Euleriano.cpp

2.3.4 MÉTODO: Verificar_Grafo_Conexo_()

```
    /*  Metodo para verificar se o grafo eh
        conexo usando percurso em profundidade.
        (CONDICAO PARA UM GRAFO SER EULERIANO)  */
    //============================================
    public: bool Verificar_Grafo_Conexo_(){

        ll Alc = 0;                             // Numero de vertices alcancados.
        vector<ll> Alcancados(this->G->QV, 0);  // Vetor para indicar se um vertice ja
            foi alcancado.

        this->Percurso_Profundidade_(this->G->Vert[Raiz].No, Alc, Alcancados);

        /*  Verifica se todos os vertices, no
            subgrafo de euler, foram alcancados.    */
        //============================================
        if(Alc != this->K) return false;    // Desconexo.
        return true;                        // Conexo.
    }
```

Codigos/Ciclo_Euleriano.cpp

2.3.5 MÉTODO: Verificar_Ponte_()

```
    /*  Metodo para verificar se uma aresta
        (V, W), do grafo, eh uma ponte. */
    //====================================
    bool Verificar_Ponte_(int V, int W){

        /*  PONTES:
            Indicacao das extremidades de uma possivel ponte.
            A aresta (V, W) passa a ser ignorada no percurso
            em profundidade para verificar se o subgrafo de
            de euler continua conexo sem a aresta.  */
        //================================================
        this->Ponte_v = V;
        this->Ponte_w = W;

        // Verifica se o grafo se torna desconexo.
        if(Verificar_Grafo_Conexo_() == false) return true; // A aresta eh realmente
            uma ponte.
        return false;                                       // A aresta nao eh uma
            ponte.
    }
```

Codigos/Ciclo_Euleriano.cpp

2.3.6 MÉTODO: Tour_Euler___FLEURY_()

```
    //====================================
    public: void Tour_Euler___FLEURY_(){

        ll V, W;              // Extremidades de uma aresta.
        set<ll> *Arestas;     // Lista de adjacencia auxiliar.

        // Loop: Enquanto houver arestas no grafo.
        while(this->G->QE != 0){

            V       = Tour.back();              // ESCOLHA DE UM VERTICE:       Ultimo
                vertice alcancado no tour.
            Arestas = &(this->G->Vert[V].Adj);  // ESCOLHA DAS ARESTAS (V, W):  Lista
                de arestas para iniciar o tour.

            // Loop: Percorre a lista de adjacencia de V.
            for(auto& auxW: (*Arestas)){

                /*  A aresta (V, W) eh escolhida se existe
                    apenas ela, ou se (V, W) nao eh ponte.  */
                if(((*Arestas).upper_bound(auxW) == (*Arestas).end())
                || (Verificar_Ponte_(V, auxW) == false)){
                    W = auxW;
                    break;
                }
            }

            // Remocao da aresta (V, W) do subgrafo.
            //=========================================
            this->G->Remover_Aresta_(V, W); // Remove a aresta (V, W) de G.
            this->G->Remover_Aresta_(W, V); // Remove a aresta (W, V) de G.
            this->Tour.push_back(W);        // Insere a aresta (V, W) no Tour.

            /*  Verifica se V nao possui mais vertices
                adjacentes apos a remocao da aresta (V, W). */
            if((this->G->Vert[V].Grau == 0))
                this->K--;  // Nesse caso atualiza-se o numero de vertices no subgrafo
                    de euler do grafo G.
        }
    }
```

Codigos/Ciclo_Euleriano.cpp

2.3.7 MÉTODO: Imprmir_Tour_Euler_()

```
    /*  Metodo para imprimir o
        tour de euler encontrado.   */
    //================================
    public: void Imprimir_Tour_Euler_(){

        cout << endl;
        cout << "|-| ============== |-|" << endl;
        cout << "|-| TOUR DE EULER:     ";
        for(auto& Vert: this->Tour)
            cout << " " << Vert +1;
        cout << "\n|-| ============== |-|" << endl;
    }
```

Codigos/Ciclo_Euleriano.cpp

2.3.8 MÉTODO: Deletar_Ciclo_Euleriano_()

```
    /*  Destructor da classe.   */
    //============================
    public: ~Ciclo_Euleriano(){
        this->Tour.clear();
        delete(this->G);
    }
};
```

Codigos/Ciclo_Euleriano.cpp

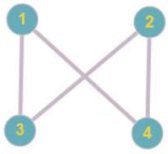
3 Resultados e Discussões

Saídas do algoritmo para alguns casos de teste.

3.1 Grafos Eulerianos

3.1.1 Grafo 01

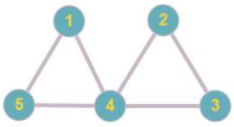
```
PS C:\Users\joao_victor\Desktop\Ciclo-Euleriano\output> & .\'Menu.exe'

  |--| Informe a quantidade de vértices do grafo:  4

  |--| Deseja gerar um grafo aleatório ? [SIM / NAO]
  |--| SIM

  |-| ============ |-|
  |-| Grafo G:     |-|
  |-| ============ |-|
      1: 3 4
      2: 3 4
      3: 1 2
      4: 1 2
  |-| ============ |-|

  |-| ============= |-|
  |-| TOUR DE EULER:     1 3 2 4 1
  |-| ============= |-|
PS C:\Users\joao_victor\Desktop\Ciclo-Euleriano\output>
```



(1) (2)

3.1.2 Grafo 02

```
PS C:\Users\joao_victor\Desktop\Ciclo-Euleriano\output> & .\'Menu.exe'

  |--| Informe a quantidade de vértices do grafo:  5

  |--| Deseja gerar um grafo aleatório ? [SIM / NAO]
  |--| SIM

  |-| ============ |-|
  |-| Grafo G:     |-|
  |-| ============ |-|
      1: 4 5
      2: 3 4
      3: 2 4
      4: 1 2 3 5
      5: 1 4
  |-| ============ |-|

  |-| ============= |-|
  |-| TOUR DE EULER:     1 4 2 3 4 5 1
  |-| ============= |-|
PS C:\Users\joao_victor\Desktop\Ciclo-Euleriano\output>
```

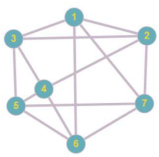


(3) (4)

3.1.3 Grafo 03

```
PS C:\Users\joao_victor\Desktop\Ciclo-Euleriano\output> & .\'Menu.exe'

  |--| Informe a quantidade de vértices do grafo:  7

  |--| Deseja gerar um grafo aleatório ? [SIM / NAO]
  |--| NAO


  |--| Informe a quantidade de arestas do grafo: 14

  |--| ================================ (014)
  |--| Informe o vertice de partida:    1
  |--| Informe o vertice de chegada:    2

  |--| ================================ (013)
  |--| Informe o vertice de partida:    2
  |--| Informe o vertice de chegada:    3

  |--| ================================ (012)
  |--| Informe o vertice de partida:    3
  |--| Informe o vertice de chegada:    4

  |--| ================================ (011)
```

```
  |--| ================================ (01)
  |--| Informe o vertice de partida:    6
  |--| Informe o vertice de chegada:    1

  |-| ============ |-|
  |-| Grafo G:     |-|
  |-| ============ |-|
      1: 2 3 6 7
      2: 1 3 4 7
      3: 1 2 4 5
      4: 2 3 5 6
      5: 3 4 6 7
      6: 1 4 5 7
      7: 1 2 5 6
  |-| ============ |-|

  |-| ============= |-|
  |-| TOUR DE EULER:     1 2 3 1 6 4 2 7 5 3 4 5 6 7 1
  |-| ============= |-|
PS C:\Users\joao_victor\Desktop\Ciclo-Euleriano\output>
```



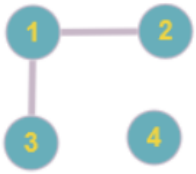
(5) (6) (7)

3.2 Grafos não Eulerianos

3.2.1 Grafo 01

```
PS C:\Users\joao_victor\Desktop\Ciclo-Euleriano> cd 'c:\Users\joao_victor\Desktop\Ciclo-Euleriano\output'
PS C:\Users\joao_victor\Desktop\Ciclo-Euleriano\output> & .\'Menu.exe'

  |--| Informe a quantidade de vértices do grafo:  4

  |--| Deseja gerar um grafo aleatório ? [SIM / NAO]
  |--| SIM

  |-| ============ |-|
  |-| Grafo G:     |-|
  |-| ============ |-|
     1: 2 3
     2: 1
     3: 1
     4:
  |-| ============ |-|

  |-| O GRAFO NÃO É EULERIANO.
PS C:\Users\joao_victor\Desktop\Ciclo-Euleriano\output>
```

(8)

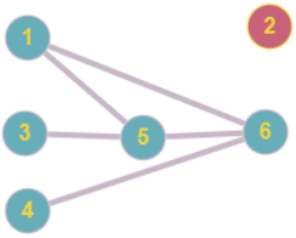


(9)

3.2.2 Grafo 02

```
PS C:\Users\joao_victor\Desktop\Ciclo-Euleriano\output> & .\'Menu.exe'

  |--| Informe a quantidade de vértices do grafo:  6

  |--| Deseja gerar um grafo aleatório ? [SIM / NAO]
  |--| SIM

  |-| ============ |-|
  |-| Grafo G:     |-|
  |-| ============ |-|
     1: 5 6
     2:
     3: 5
     4: 6
     5: 1 3 6
     6: 1 4 5
  |-| ============ |-|

  |-| O GRAFO NÃO É EULERIANO.
PS C:\Users\joao_victor\Desktop\Ciclo-Euleriano\output>
```

(10)

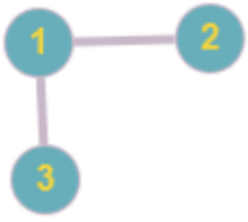


(11)

3.2.3 Grafo 03

```
PS C:\Users\joao_victor\Desktop\Ciclo-Euleriano\output> & .\'Menu.exe'

  |--| Informe a quantidade de vértices do grafo:  3

  |--| Deseja gerar um grafo aleatório ? [SIM / NAO]
  |--| NAO


  |--| Informe a quantidade de arestas do grafo: 2

  |--| ================================ (02)
  |--| Informe o vertice de partida:    1
  |--| Informe o vertice de chegada:    2

  |--| ================================ (01)
  |--| Informe o vertice de partida:    1
  |--| Informe o vertice de chegada:    3

  |-| ============ |-|
  |-| Grafo G:     |-|
  |-| ============ |-|
     1: 2 3
     2: 1
     3: 1
  |-| ============ |-|

  |-| O GRAFO NÃO É EULERIANO.
PS C:\Users\joao_victor\Desktop\Ciclo-Euleriano\output>
```

(12)



(13)