Lista grafos

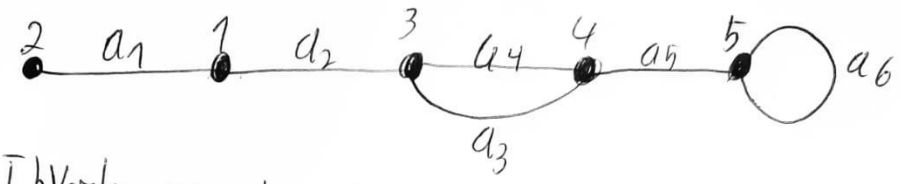
Professor:Diego Nunes Brandão

Aluno: Nicolas Vycas Nery

Matrícula: 2012383BCC

Questão 1:

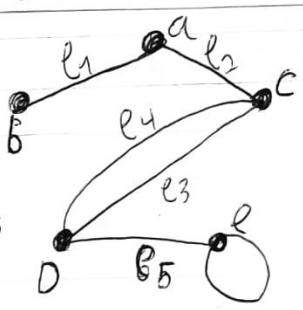
1)

a) 

b) I) Vértices não adjacentes: 3 e 5, ou 4 e 1, ou 3 e 2

II) Um laço: a6

III) grau vértice 3: 3

c) 

$V_i = \{a, b, c, D, e\}$

$E_i = \{e_1, e_2, e_3, e_4, e_5, e_6\}$

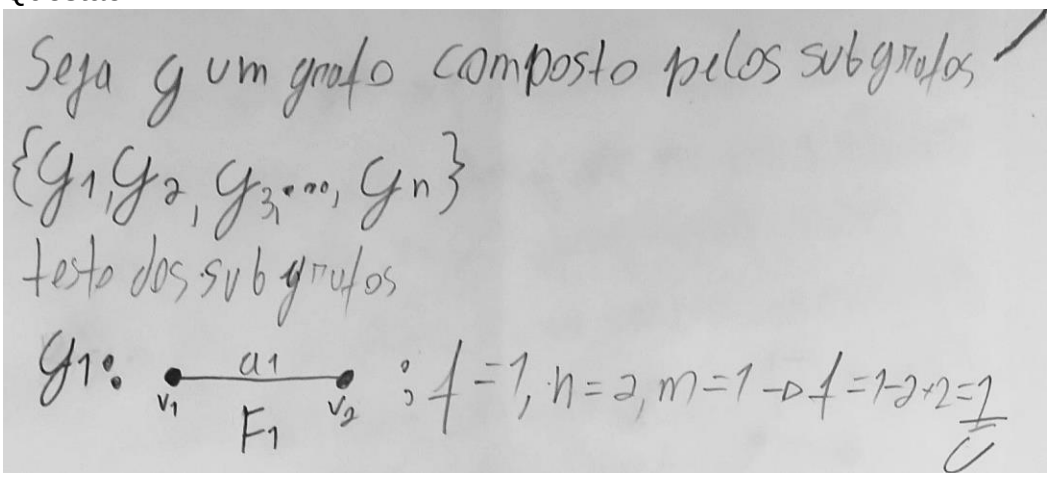
D) Matriz de adjacência

$$A = \begin{vmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{vmatrix}$$

Lista de adjacência

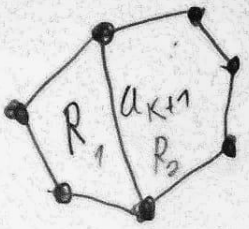
1 = {2, 3}
2 = {1}
3 = {1, 4}
4 = {3, 5}
5 = {5, 4}

Questão 2:


Seja g um grafo composto pelos subgrafos
{g1, g2, g3, ..., gn}
teste dos subgrafos
g1:
a1
v1
v2
F1
: f = 1, n = 2, m = 1 → f = 1 - 2 + 2 = 1

Hipótese: assumindo que $f_K = m_K + n_K + 2$, é válida para um subgrafo G_K qualquer, ela deve valer para o grafo G_{K+1}, que é obtido adicionando uma nova aresta ou vértice a g_K

g_{K+1} (I)



a_{K+1} dividiu o grafo e deu a nova Região. logo

$f_{K+1} = (f_K) + 1$

$m_{K+1} = (m_K) + 1$

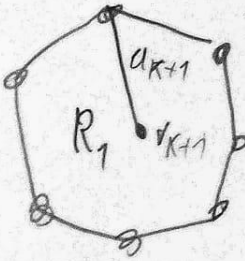
$n_{K+1} = n_K$

Logo:

$(f_{K+1}) = (m_K) + 1 + n_K + 2$

$f_K = m_K + n_K + 2$

g_{K+1} (II)



a_{K+1} não dividiu o grafo em duas novas faces, logo:

$f_{K+1} = f_K$

$m_{K+1} = (m_K) + 1$

$n_{K+1} = (n_K) + 1$

logo:

$f_{K+1} = m_K + 1 + (n_K + 1) + 2$

$f_{K+1} = (m_K + 1 - n_K - 1) + 2$

$f_{K+1} = m_K - n_K + 2$

$f_{K+1} = f_K$

Questão 3:

```
/*
 * Nicolas Vycas Nery
 * 23/05/2020
 */
#include <iostream>

template<typename Struct>
struct Item {
    Struct dado;
    Item *proximo;


    Item() {}


    Item(Struct dado) {
        this->dado = dado;
    }
};

template<typename Struct>
struct Lista {
    Item<Struct> *cabeca;


    void remover(Item<Struct> *item) {
        if (item == cabeca) {
            Item<Struct> *antigo = cabeca;
            cabeca = cabeca->proximo == nullptr ? cabeca = nullptr : cabeca->proximo;
            delete antigo;
        } else {
            Item<Struct> *anterior = cabeca;
            while (anterior->proximo != nullptr && anterior->proximo != item) {
                anterior = anterior->proximo;
            }
            if (anterior->proximo != nullptr) {
                anterior->proximo = anterior->proximo->proximo;
                delete item;
            }
        }
    }


    void remover(char item) {
        if (item == cabeca->dado) {
            Item<char> *antigo = cabeca;
            cabeca = cabeca->proximo == nullptr ? cabeca = nullptr : cabeca->proximo;
            delete antigo;
        } else {
            Item<char> *anterior = cabeca;
            while (anterior->proximo != nullptr && anterior->proximo->dado != item) {
                anterior = anterior->proximo;
            }
            if (anterior->proximo != nullptr) {
                Item<char> *antigo = anterior->proximo;
                anterior->proximo = anterior->proximo->proximo;


                delete antigo;
            }
        }
    }
```

```
    void inserir(Item<Struct> *item) {
        if (cabeca == nullptr) {
            cabeca = item;
        } else {
            Item<Struct> *ulitmo = cabeca;
            while (ulitmo->proximo != nullptr) {
                ulitmo = ulitmo->proximo;
            }
            ulitmo->proximo = item;
        }
    }

    Item<Struct> *ultimo() {
        Item<Struct> *ultimo = cabeca;
        while (ultimo->proximo != nullptr)
            ultimo = ultimo->proximo;
        return ultimo;
    }

    bool possui(Item<char> item) {
        Item<Struct> *ultimo = cabeca;
        while (ultimo->proximo != nullptr) {
            if (ultimo->dado == item.dado) return true;
            ultimo = ultimo->proximo;
        }
        return false;
    }
};

struct Grafo {
    int numeroVertices;
    Lista<Lista<char>> listaListasAdjacencia;

    Grafo() {
        this->numeroVertices = 0;
        this->listaListasAdjacencia = *new Lista<Lista<char>>;
    }

    Item<Lista<char>> *getLista(char vertice) {
        if (existe(vertice)) {
            Item<Lista<char>> *ultimo = listaListasAdjacencia.cabeca;
            while (ultimo != nullptr) {
                if (ultimo->dado.cabeca->dado == vertice)
                    return ultimo;
                ultimo = ultimo->proximo;
            }
        } else
            return nullptr;
    }

    Lista<char> *getVizinhos(char vertice) {
        if (existe(vertice)) {
            Item<Lista<char>> *ultimo = listaListasAdjacencia.cabeca;
            while (ultimo->proximo != nullptr) {
                if (ultimo->dado.cabeca->dado == vertice)
                    return &ultimo->dado;
                ultimo = ultimo->proximo;
            }
        } else
            return nullptr;
    }
```

```
void imprimirAdjacencia(char vertice) {
    if (existe(vertice)) {
        Item<Lista<char>> *listaVerice = getLista(vertice);
        if (listaVerice->proximo == nullptr) {
            std::cout << vertice << ": sem vizinhos";
        } else {
            Item<char> *vizinho = getVizinhos(vertice)->cabeca;
            std::cout << vizinho->dado << ": ";
            vizinho = vizinho->proximo;
            while (vizinho != nullptr) {
                std::cout << vizinho->dado;
                vizinho = vizinho->proximo;
                if (vizinho == nullptr) std::cout << "";
                else std::cout << ",";
            }
        }
    } else
        std::cout << "Vértice " << vertice << " não existe";
}

void imprimirGrafo() {
    std::cout << "Imprimindo grafo" << std::endl;
    if (listaListasAdjacencia.cabeca != nullptr) {
        Item<Lista<char>> *vertice = listaListasAdjacencia.cabeca;
        while (vertice != nullptr) {
            imprimirAdjacencia(vertice->dado.cabeca->dado);
            std::cout << std::endl;
            vertice = vertice->proximo;
        }
        std::cout << "Total vertices: " << numeroVertices << std::endl;
    } else {
        std::cout << "O grafo esta vazio" << std::endl;
    }
}

bool existe(char vertice) {
    Item<Lista<char>> *proximo = listaListasAdjacencia.cabeca;
    while (proximo != nullptr) {
        if (proximo->dado.cabeca->dado == vertice) return true;
        proximo = proximo->proximo;
    }
    return false;
}

void adicionarVertice(char vertice) {
    if (!existe(vertice)) {
        Item<Lista<char>> *novo = new Item<Lista<char>>;
        novo->dado = *new Lista<char>;
        novo->dado.inserir(new Item<char>(vertice));
        if (listaListasAdjacencia.cabeca == nullptr) listaListasAdjacencia.cabeca = novo;
        else {
            Item<Lista<char>> *ultimo = listaListasAdjacencia.ultimo();
            ultimo->proximo = novo;
        }
        numeroVertices++;
        std::cout << "Adicionado vértice:" << vertice << std::endl;
    } else {
        std::cout << "Vértice:" << vertice << " já existe, não pode ser adicionado" << std::endl;
    }

}
```

```
    void removerVertice(char vertice) {
        if (existe(vertice)) {
            Item<char> *vizinho = this->getVizinhos(vertice)->cabeca->proximo;
            while (vizinho != nullptr) {
                removerVizinhaca(vizinho->dado, vertice);
                vizinho = vizinho->proximo;
            }
            Item<Lista<char>> *listaVertice = getLista(vertice);
            listaListasAdjacencia.remover(listaVertice);
            numeroVertices--;
            std::cout << "Removido vértice:" << vertice << std::endl;
        } else {
            std::cout << "Vértice:" << vertice << " não existe, não pode ser removido" << std::endl;
        }
    }

    void AdicionaVizinhaca(char verticeA, char verticeB) {
        if (existe(verticeA) && existe(verticeB)) {
            if (verticeA == verticeB) {
                getLista(verticeA)->dado.inserir(new Item<char>(verticeB));
                std::cout << verticeA << " é vizinho de " << verticeB << std::endl;
            } else {
                getLista(verticeB)->dado.inserir(new Item<char>(verticeA));
                getLista(verticeA)->dado.inserir(new Item<char>(verticeB));
                std::cout << verticeA << " e " << verticeB << " agora são vizinhos" << std::endl;
            }
        } else {
            std::cout << "Não é possível relacionar " << verticeA << " e " << verticeB
                      << " , um dos vertices ou os dois não existem " << std::endl;
        }
    }

    void removerVizinhaca(char verticeA, char verticeB) {
        if (existe(verticeA) && existe(verticeB)) {
            if (testeVizinhaca(verticeA, verticeB)) {
                getLista(verticeA)->dado.remover(verticeB);
                getLista(verticeB)->dado.remover(verticeA);
            } else
                std::cout << "Não é possível remover relação entre " << verticeA << " e " << verticeB
                          << ", os vertices não compartilhão um vizinhança" << std::endl;
        } else
            std::cout << "Não é possível remover relação entre " << verticeA << " e " << verticeB
                      << " , um dos vertices ou os dois não existem " << std::endl;
    }

    bool testeVizinhaca(char verticeA, char verticeB) {
        if (existe(verticeA) && existe(verticeB)) {
            Lista<char> *vizinhosA = getVizinhos(verticeA);
            Lista<char> *vizinhosB = getVizinhos(verticeB);
            std::cout << verticeA << " e " << verticeB << " são vizinhos" << std::endl;
            return vizinhosA->possui(verticeB) && vizinhosB->possui(verticeA);
        } else {
            std::cout << verticeA << " e " << verticeB << " não são vizinhos" << std::endl;
            return false;
        }
    }
};









int main() {
```

```
    Grafo *grafo = new Grafo;

    grafo->adicionarVertice('A');
    grafo->adicionarVertice('B');
    grafo->adicionarVertice('C');
    grafo->adicionarVertice('D');
    grafo->adicionarVertice('E');
    grafo->imprimirGrafo();

    std::cout << std::endl;
    grafo->AdicionaVizinhaca('B', 'A');
    grafo->AdicionaVizinhaca('C', 'D');
    grafo->AdicionaVizinhaca('A', 'A');
    grafo->AdicionaVizinhaca('C', 'C');

    std::cout << std::endl;
    grafo->imprimirGrafo();

    std::cout << std::endl;
    grafo->adicionarVertice('G');
    grafo->adicionarVertice('H');
    grafo->adicionarVertice('K');

    std::cout << std::endl;
    grafo->AdicionaVizinhaca('G', 'A');
    grafo->AdicionaVizinhaca('G', 'B');
    grafo->AdicionaVizinhaca('G', 'D');
    grafo->AdicionaVizinhaca('G', 'E');
    grafo->AdicionaVizinhaca('H', 'H');
    grafo->removerVizinhaca('A', 'B');

    std::cout << std::endl;
    grafo->imprimirGrafo();

    std::cout << std::endl;
    grafo->AdicionaVizinhaca('C', 'G');
    grafo->AdicionaVizinhaca('C', 'H');
    grafo->AdicionaVizinhaca('C', 'K');
    grafo->AdicionaVizinhaca('C', 'B');
    grafo->AdicionaVizinhaca('C', 'C');

    std::cout << std::endl;
    grafo->imprimirGrafo();

    std::cout << std::endl;
    std::cout << std::endl;
    grafo->removerVertice('C');

    std::cout << std::endl;
    std::cout << std::endl;
    grafo->removerVertice('G');

    std::cout << std::endl;
    std::cout << std::endl;
    grafo->imprimirGrafo();
}
```