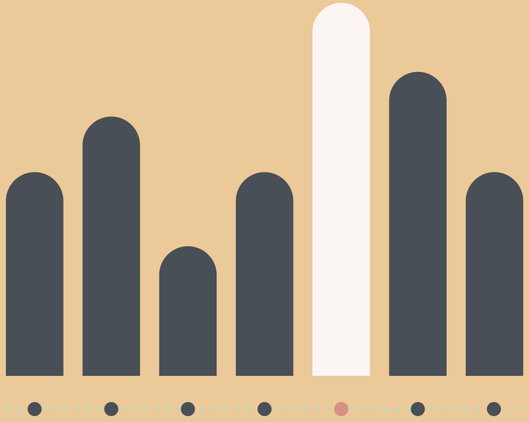
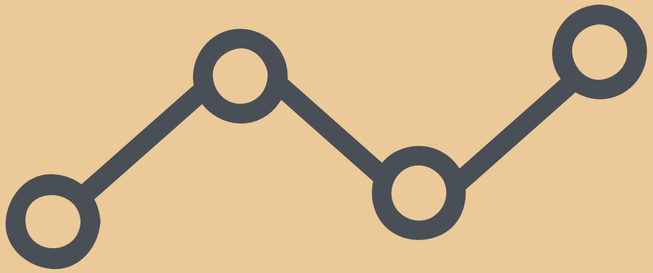
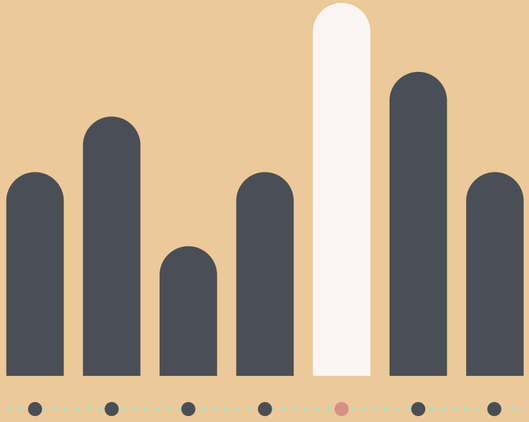
DCC059 - TEORIA DOS GRAFOS

Trabalho parte 1







CONSTRUTOR E FUNÇOES GET

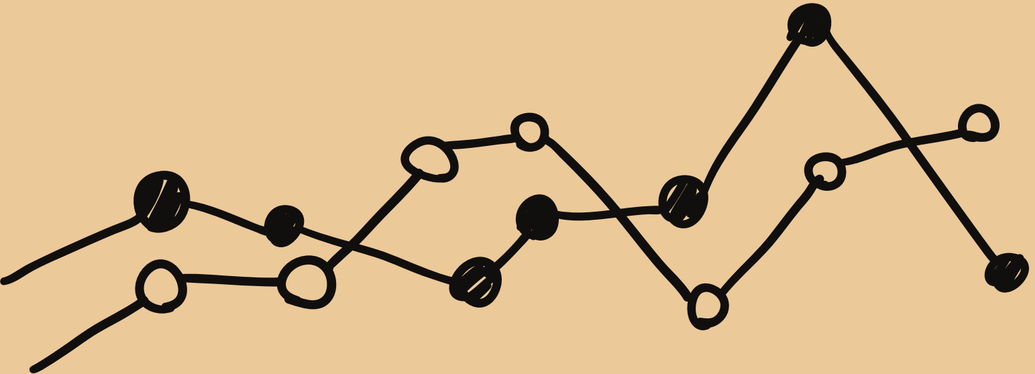
```
GrafoMatriz::GrafoMatriz(int numVertices, bool
{
    matrizAdj = new int *[nVertices];
    for (int i = 0; i < nVertices; i++)
    {
        matrizAdj[i] = new int[nVertices];
        for (int j = 0; j < nVertices; j++)
        {
            matrizAdj[i][j] = 0; // Inicializa
        }
    }
}
```

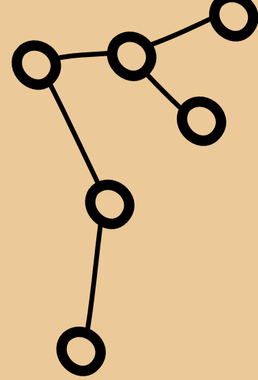
```
int Grafo::get_ordem()
{
    return nVertices;
}

bool Grafo::eh_direcionado()
{
    return direcionado;
}

bool Grafo::vertice_ponderado()
{
    return ponderadoVertices;
}

bool Grafo::aresta_ponderada()
{
    return ponderadoArestas;
}
```





GET_GRAU()

GRAFO MATRIZ

```
int GrafoMatriz::get_grau()
{
    int ordem = nVertices;
    int grauMaior = 0;
    for (int i = 0; i < ordem; i++)
    {
        int grauAtual = 0;
        for (int j = 0; j < ordem; j++)
        {
            if (matrizAdj[i][j] != 0)
            {
                grauAtual++;
            }
        }
        if (grauAtual > grauMaior)
        {
            grauMaior = grauAtual;
        }
    }
    return grauMaior;
}
```

GRAFO LISTA

```
int GrafoLista::get_grau()
{
    int grauMaior = 0;
    for (int i = 0; i < nVertices; ++i) {
        if(grauMaior < vertices[i].getGrauVertice())
            grauMaior = vertices[i].getGrauVertice();
    }
    return grauMaior;
}
```

EH_COMPLETO()

GRAFO MATRIZ

```cpp
bool GrafoMatriz::eh_completo() {
    for (int i = 0; i < nVertices; i++) {
        for (int j = 0; j < nVertices; j++) {
            if (i != j && matrizAdj[i][j] == 0) {
                return false;
            }
        }
    }
    return true;
}
```

GRAFO LISTA

```cpp
bool GrafoLista::eh_completo(){
    for (int i = 0; i < nVertices; i++){
        for (int j = 0; j< nVertices; j++){
            if (i != j)
            {
                if(direcionado && !vertices[i].existeAresta(j) &&
                !vertices[j].existeAresta(i)) //verificar
                implementação da matriz
                {
                    return false; //se não exite aresta o grafo
                    não é completo
                }
                if (!direcionado && !vertices[j].existeAresta(i))
                {
                    return false;
                }
            }
        }
    }
    return true;
}
```

EH_ARVORE()

GRAFO MATRIZ

```cpp
bool GrafoMatriz::eh_arvore() {
    // Verifica se o grafo é conexo
    if (!ehConexo()) {
        return false;
    }

    // Verifica se o grafo possui ciclos
    bool* visitado = new bool[nVertices];
    for (int i = 0; i < nVertices; i++) {
        visitado[i] = false;
    }

    bool temCiclo = temCicloDFS(0, visitado, -1);

    delete[] visitado;

    // Um grafo é uma árvore se for conexo e não tiver ciclos
    return !temCiclo;
}
```

GRAFO LISTA

```cpp
bool GrafoLista::eh_arvore() {
    if (nVertices == 0) return false;

    bool *visitado = new bool[nVertices]; // Cria um vetor de
    visitados

    for (int i = 0; i < nVertices; i++) {
        visitado[i] = false; // Inicializa o vetor de visitados
        com false
    }

    // Verifica se o grafo possui ciclo
    if (temCicloDFS(0, visitado, -1)) {
        delete[] visitado;
        return false;
    }

    // Verifica se o grafo é conexo
    for (int i = 0; i < nVertices; i++) {
        visitado[i] = false; // Reinicializa o vetor de visitados
        com false
    }
    DFS(0, visitado);

    for (int i = 0; i < nVertices; i++) {
        if (!visitado[i]) { // Se algum vértice não foi visitado,
        o grafo não é conexo
            delete[] visitado;
            return false;
        }
    }

    delete[] visitado;
    return true;
}
```

FUNÇÃO EH_BIPARTIDO

GRAFO MATRIZ

```cpp
bool GrafoMatriz::verificarParticaoBipartida(int v, int subconjunto[])
{
    if (v == nVertices)
    {
        // Verifica se a partição atual é válida
        for (int i = 0; i < nVertices; i++)
        {
            for (int j = 0; j < nVertices; j++)
            {
                if (matrizAdj[i][j] != 0 && subconjunto[i] == subconjunto[j])
                {
                    return false; // Encontrou uma aresta entre vértices do me
                }
            }
        }
        return true; // Partição válida
    }

    // Tenta atribuir o vértice v ao conjunto 0
    subconjunto[v] = 0;
    if (verificarParticaoBipartida(v + 1, subconjunto))
    {
        return true;
    }

    // Tenta atribuir o vértice v ao conjunto 1
    subconjunto[v] = 1;
    if (verificarParticaoBipartida(v + 1, subconjunto))
    {
        return true;
    }

    // Nenhuma partição válida encontrada
    subconjunto[v] = -1;
    return false;
}
```

```cpp
bool GrafoLista::eh_bipartido(){
    int* subconjunto = new int[nVertices];
    for (int i = 0; i < nVertices; i++) {
        subconjunto[i] = -1; // Inicializa todos os vértices com
    }

    bool resultado = verificarParticaoBipartida(0, subconjunto);

    delete[] subconjunto;
    return resultado;
}
```

GRAFO LISTA

```cpp
bool GrafoLista::verificarParticaoBipartida(int v, int subconjunto[]) {
    if (v == nVertices) {
        // Verifica se a partição atual é válida
        for (int i = 0; i < nVertices; i++) {
            Aresta* aresta = vertices[i].getArestas();
            while (aresta != nullptr) {
                int j = aresta->getDestino();
                if (subconjunto[i] == subconjunto[j]) {
                    return false; // Encontrou uma aresta entre vértices
                }
                aresta = aresta->getProx();
            }
        }
        return true; // Partição válida
    }

    // Tenta atribuir o vértice v ao conjunto 0
    subconjunto[v] = 0;
    if (verificarParticaoBipartida(v + 1, subconjunto)) {
        return true;
    }

    // Tenta atribuir o vértice v ao conjunto 1
    subconjunto[v] = 1;
    if (verificarParticaoBipartida(v + 1, subconjunto)) {
        return true;
    }

    // Nenhuma partição válida encontrada
    subconjunto[v] = -1;
    return false;
}
```

FUNÇÃO N_CONEXO

GRAFO MATRIZ

```
void GrafoMatriz::DFS(int v, bool visitado[])
{
    visitado[v] = true;
    for (int i = 0; i < nVertices; i++)
    {
        if (matrizAdj[v][i] != 0 && !visitado[i])
        {
            DFS(i, visitado);
        }
    }
}
```

```
int GrafoMatriz::n_conexo()        Paulo Aqui
{
    bool *visitado = new bool[nVertices];
    for (int i = 0; i < nVertices; i++)
    {
        visitado[i] = false; // Inicializa
    }

    int count = 0;
    for (int v = 0; v < nVertices; v++)
    {
        if (!visitado[v])
        {
            DFS(v, visitado); // Realiza uma
            count++;          // Incrementa
        }
    }

    delete[] visitado; // Libera a memória
    return count;
}
```

GRAFO LISTA

```
void GrafoLista::DFS(int v, bool visited[]) {
    visited[v] = true; // Marca o vértice como visitad
    Aresta* aresta = vertices[v].getArestas();
    while (aresta != nullptr) {
        int destino = aresta->getDestino();
        if (!visited[destino]) {
            DFS(destino, visited); // Chama recursivam
        }
        aresta = aresta->getProx();
    }
}
```

POSSUI_ARTICULAÇÃO()

```
bool GrafoMatriz::possui_articulacao() {
    bool* visitado = new bool[nVertices];
    int* discovery = new int[nVertices];
    int* low = new int[nVertices];
    int* parent = new int[nVertices];
    bool articulacaoEncontrada = false;

    for (int i = 0; i < nVertices; i++) {
        visitado[i] = false;
        parent[i] = -1;
    }

    for (int i = 0; i < nVertices; i++) {
        if (!visitado[i]) {
            DFSArticulacao(i, visitado, discovery, low, parent,
            articulacaoEncontrada);
        }
    }

    delete[] visitado;
    delete[] discovery;
    delete[] low;
    delete[] parent;

    return articulacaoEncontrada;
}
```

POSSUI_ARTICULAÇÃO()

GRAFO MATRIZ

```
void GrafoMatriz::DFSArticulacao(int v, bool visitado[], int discovery[], int low[],
int parent[], bool& articulacaoEncontrada) {
    static int tempo = 0;
    int filhos = 0;
    visitado[v] = true;
    discovery[v] = low[v] = ++tempo;

    for (int i = 0; i < nVertices; i++) {
        if (matrizAdj[v][i] != 0) {
            int u = i;
            if (!visitado[u]) {
                filhos++;
                parent[u] = v;
                DFSArticulacao(u, visitado, discovery, low, parent,
                articulacaoEncontrada);

                low[v] = min(low[v], low[u]);

                if (parent[v] == -1 && filhos > 1) {
                    articulacaoEncontrada = true;
                }

                if (parent[v] != -1 && low[u] >= discovery[v]) {
                    articulacaoEncontrada = true;
                }
            } else if (u != parent[v]) {
                low[v] = min(low[v], discovery[u]);
            }
        }
    }
}
```

GRAFO LISTA

```
void GrafoLista::DFSArticulacao(int v, bool visited[], int low[], int parent[],
bool &articulacao) {
    int children = 0;
    visited[v] = true;
    low[v] = v; // Inicializa low com o índice do vértice
    Aresta* aresta = vertices[v].getArestas();
    while (aresta != nullptr) {
        int destino = aresta->getDestino();
        if (!visited[destino]) {
            children++;
            parent[destino]
            DFSArticulacao(destino, visited, low, parent, articulacao);
            low[v] = (low[v] < low[destino]) ? low[v] : low[destino];
            if (parent[v] == -1 && children > 1)
                articulacao = true;
            if (parent[v] != -1 && low[destino] >= v)
                articulacao = true;
        } else if (destino != parent[v]) {
            low[v] = (low[v] < destino) ? low[v] : destino;
        }
        aresta = aresta->getProx();
    }
}
```

int destino

POSSUI_PONTE()

```
bool GrafoLista::possui_ponte() {
    bool *visited = new bool[nVertices];
    int *disc = new int[nVertices];
    int *low = new int[nVertices];
    int *parent = new int[nVertices];
    bool possuiPonte = false;
    for (int i = 0; i < nVertices; i++) {
        visited[i] = false;
        disc[i] = -1;
        low[i] = -1;
        parent[i] = -1;
    }
    for (int i = 0; i < nVertices; i++) {
        if (!visited[i]) {
            DFS_Ponte(i, visited, disc, low, parent, possuiPonte);
        }
    }
    delete[] visited;
    delete[] disc;
    delete[] low;
    delete[] parent;
    return possuiPonte;
}
```

POSSUI_PONTE()

GRAFO MATRIZ

```
void GrafoMatriz::DFSPonte(int v, bool visitado[], int discovery[], int low[], int
parent[], bool &ponteEncontrada)
{
    static int tempo = 0;
    visitado[v] = true;
    discovery[v] = low[v] = ++tempo;

    for (int i = 0; i < nVertices; i++)
    {
        if (matrizAdj[v][i] != 0)
        {
            int u = i;
            if (!visitado[u])
            {
                parent[u] = v;
                DFSPonte(u, visitado, discovery, low, parent, ponteEncontrada);

                low[v] = min(low[v], low[u]);

                if (low[u] > discovery[v])
                {
                    ponteEncontrada = true;
                }
            }
            else if (u != parent[v])
            {
                low[v] = min(low[v], discovery[u]);
            }
        }
    }
}
```

GRAFO LISTA

```
void GrafoLista::DFS_Ponte(int u, bool visited[], int disc[], int low[], int
parent[], bool &possuiPonte)
{
    static int tempo = 0;
    visited[u] = true;
    disc[u] = low[u] = ++tempo;
    Aresta *aresta = vertices[u].getArestas();

    while (aresta != nullptr) {
        int v = aresta->getDestino();
        if (!visited[v]) {
            parent[v] = u;
            DFS_Ponte(v, visited, disc, low, parent, possuiPonte);
            low[u] = std::min(low[u], low[v]);
            if (low[v] > disc[u]) {
                possuiPonte = true;
            }
        } else if (v != parent[u]) {
            low[u] = std::min(low[u], disc[v]);
        }
        aresta = aresta->getProx();
    }
}
```

ALUNOS

Fabio do Vale Affonso	202076021
Leandro Alvares	202065211A
Leticia Melgar Floro	201976008
Lucas Gonçalves Rocha	202265187AC
Paulo Vitor F. R de Aquino	202176014