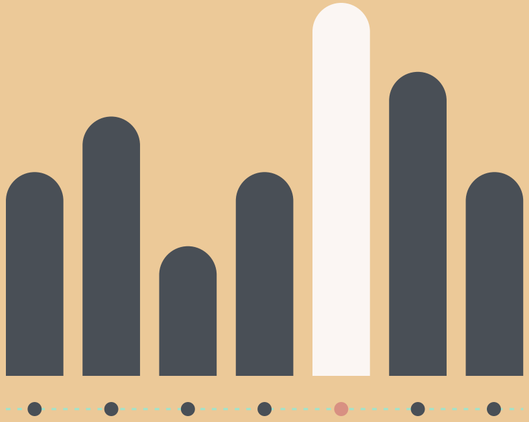
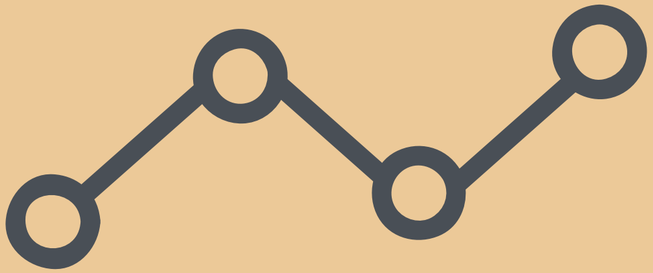
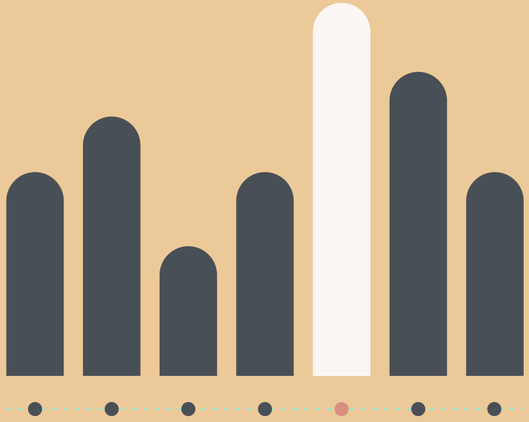
DCC059 - TEORIA DOS GRAFOS

Trabalho parte 2

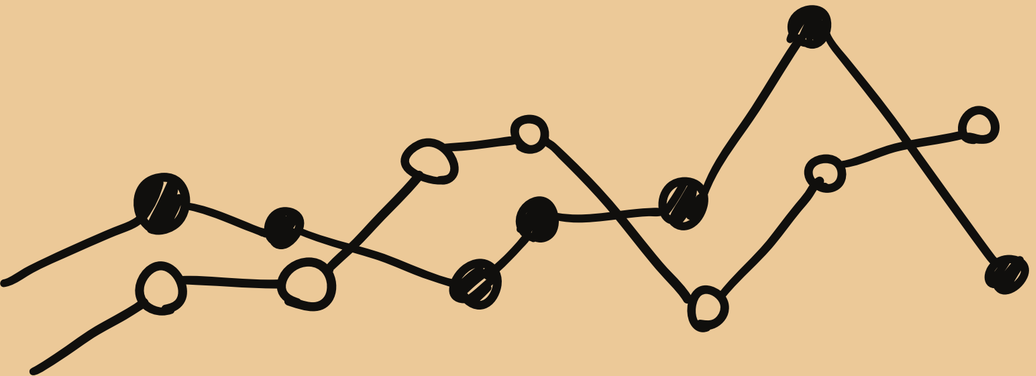


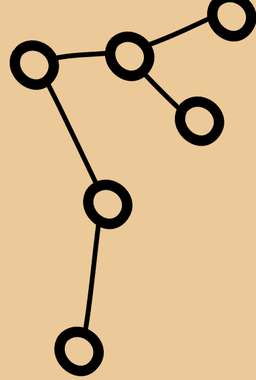




ALOCAÇÃO DINÂMICA
GRAFO MATRIZ

```
GrafoMatriz::GrafoMatriz(int numVertices, bool direcionado, bool po
{
    tamanhoMatriz = 10;

    if (numVertices <= 0)
    {
        numVertices = 1;
    }

    while (tamanhoMatriz < numVertices)
    {
        tamanhoMatriz *= 2;
    }

    matrizAdj = new int *[tamanhoMatriz];
    for (int i = 0; i < tamanhoMatriz; i++)
    {
        matrizAdj[i] = new int[tamanhoMatriz];
        for (int j = 0; j < tamanhoMatriz; j++)
        {
            matrizAdj[i][j] = 0; // Inicializa com 0 (sem aresta)
        }
    }
}
```





ADICIONAR ARESTA

```cpp
void GrafoMatriz::adicionar_aresta(int origem, int destino, float peso)
{
    if (origem < 0 || origem > numVertices || destino < 0 || destino > numVertices)
    {
        cout << "Erro: indices de origem ou destino invalidos.\n";
        return;
    }

    if (ponderadoArestas)
    {
        matrizAdj[origem][destino] = peso;
        if (!direcionado)
        {
            matrizAdj[destino][origem] = peso;
        }
    }
    else
    {
        matrizAdj[origem][destino] = 1;
        if (!direcionado)
        {
            matrizAdj[destino][origem] = 1;
        }
    }
}
```

REMOVER ARESTA

```cpp
void GrafoMatriz::remover_aresta(int origem, int destino)
{
    if (origem < 0 || origem >= numVertices || destino < 0 || destino >= numVertices)
    {
        cout << "Erro: indices de origem ou destino invalidos.\n";
        return;
    }

    matrizAdj[origem][destino] = 0;
    if (!direcionado)
    {
        matrizAdj[destino][origem] = 0;
    }
}
```

GRAFO MATRIZ

ADICIONAR VÉRTICE

```cpp
// Adiciona um vértice ao grafo
void GrafoMatriz::adicionar_vertice(int id, float peso)
{
    int novoNumVertices = numVertices + 1;
    numVertices = novoNumVertices;

    if (novoNumVertices >= tamanhoMatriz)
    {
        cout << "Aumentando matriz de adjacencia...\n";
        int antigoTamanhoMatriz = tamanhoMatriz;
        tamanhoMatriz = max(tamanhoMatriz * 2, novoNumVertices);
        int **novaMatriz = new int *[tamanhoMatriz];

        for (int i = 0; i < tamanhoMatriz; i++)
        {
            novaMatriz[i] = new int[tamanhoMatriz]();
        }

        for (int i = 0; i < antigoTamanhoMatriz; i++)
        {
            for (int j = 0; j < antigoTamanhoMatriz; j++)
            {
                novaMatriz[i][j] = matrizAdj[i][j];
            }
        }

        for (int i = 0; i < antigoTamanhoMatriz; i++)
        {
            delete[] matrizAdj[i];
        }
        delete[] matrizAdj;
        matrizAdj = novaMatriz;
    }
}
```

REMOVER VÉRTICE

```cpp
// Remove um vertice do grafo
void GrafoMatriz::remover_vertice(int id)
{
    if (id < 0 || id >= numVertices)
    {
        cout << "Erro: indice de vertice invalido.\n";
        return;
    }

    int novoNumVertices = numVertices - 1; int **novaMatriz = new int *[tamanhoMatriz]

    for (int i = 0; i < tamanhoMatriz; i++)
    {
        novaMatriz[i] = new int[tamanhoMatriz]();
    }
    for (int i = 0, ni = 0; i < tamanhoMatriz; i++)
    {
        if (i == id)
            continue;
        for (int j = 0, nj = 0; j < tamanhoMatriz; j++)
        {
            if (j == id)
                continue;
            novaMatriz[ni][nj] = matrizAdj[i][j];
            nj++;
        }
        ni++;
    }
    for (int i = 0; i < numVertices; i++)
    {
        delete[] matrizAdj[i];
    }
    delete[] matrizAdj;
    matrizAdj = novaMatriz;
    numVertices = novoNumVertices;
}
```

GRAFO MATRIZ

CALCULAR_MENOR_DIST

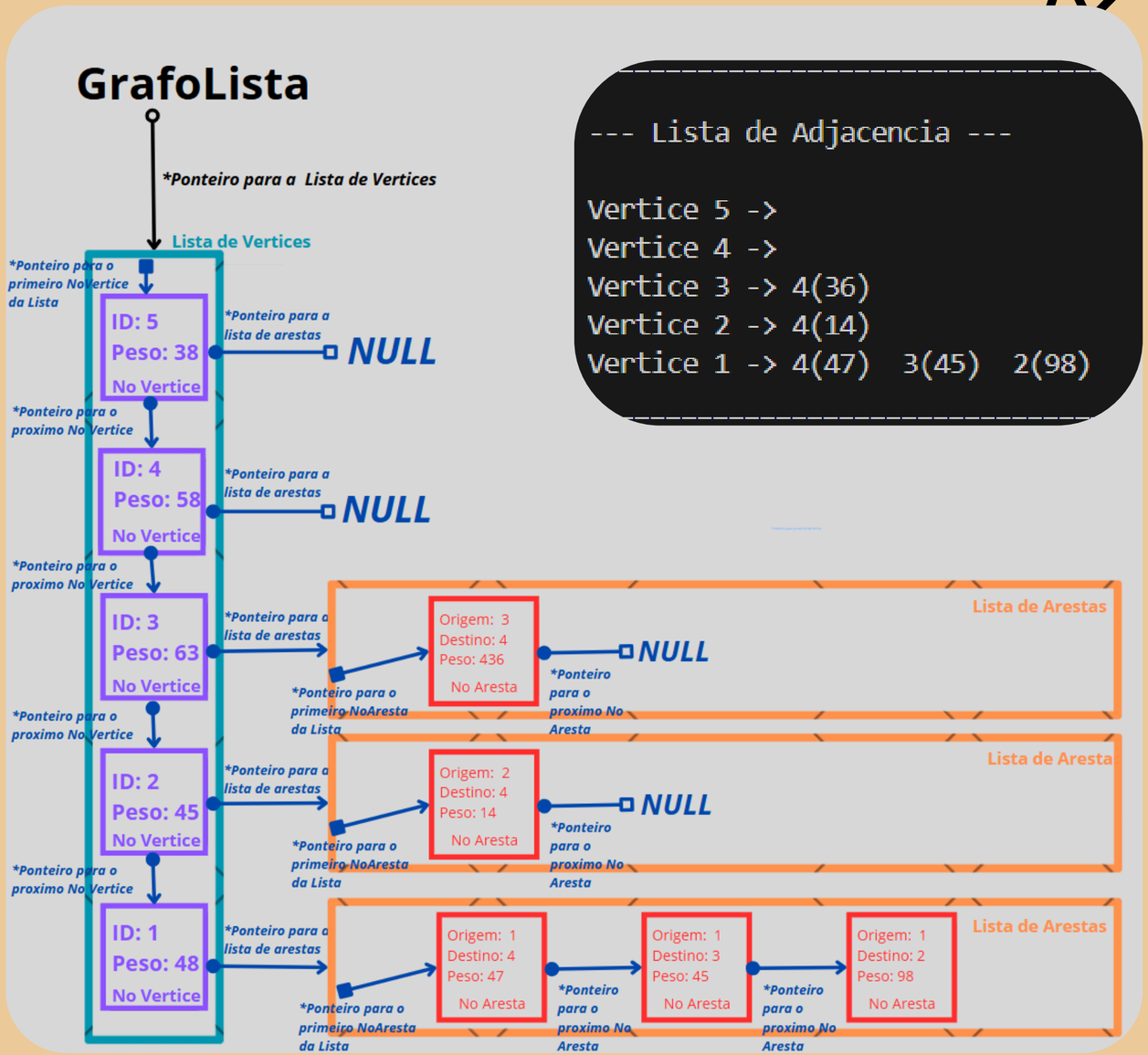
```cpp
// Calcula menor distancia entre dois vertices
int GrafoMatriz::calcula_menor_dist(int origem, int destino)
{
    const int INF = 1000000; // Valor grande para representar infinito
    int *dist = new int[numVertices + 1];
    int *prev = new int[numVertices + 1];
    bool *visitado = new bool[numVertices + 1];

    for (int i = 1; i <= numVertices; i++)
    {
        dist[i] = INF;
        prev[i] = -1;
        visitado[i] = false;
    }

    dist[origem] = 0;

    // Loop principal do algoritmo de Dijkstra
    for (int i = 1; i <= numVertices; i++)
    {
        int u = -1;

        // Encontra o vertice nao visitado com a menor distancia
        for (int j = 1; j <= numVertices; j++)
        {
            if (!visitado[j] && (u == -1 || dist[j] < dist[u]))
            {
                u = j;
            }
        }

        // Se a menor distancia e infinita, todos os vertices restantes sao inacessiveis
        if (dist[u] == INF)
        {
            break;
        }

        visitado[u] = true;

        // Atualiza as distancias dos vizinhos do vertice atual
        for (int v = 1; v <= numVertices; v++)
        {
            if (matrizAdj[u][v] != 0 && dist[u] + matrizAdj[u][v] < dist[v])
            {
                dist[v] = dist[u] + matrizAdj[u][v];
                prev[v] = u;
            }
        }
    }

    int menorDist = dist[destino];

    delete[] dist;
    delete[] prev;
    delete[] visitado;

    return menorDist;
}
```

CALCULAR MAIOR_MENOR_DIST

```cpp
// Percorre todos os pares de vértices e busca a maior das menores distâncias
int Grafo::calcula_maior_menor_dist() {
    int maiorMenorDist = 0;
    int verticeOrigem = -1;
    int verticeDestino = -1;

    for (int i = 1; i <= numVertices; i++) {
        for (int j = 1; j <= numVertices; j++) {
            if (i != j) {
                int menorDist = calcula_menor_dist(i, j);
                if (menorDist != 1000000 && menorDist > maiorMenorDist) {
                    maiorMenorDist = menorDist;
                    verticeOrigem = i;
                    verticeDestino = j;
                }
            }
        }
    }

    if (verticeOrigem != -1 && verticeDestino != -1) {
        cout << "Maior menor distancia: (" << verticeOrigem << "-" << verticeDestino << ") = " << maiorMenorDist << endl;
    } else {
        cout << "Nao ha caminhos validos no grafo." << endl;
    }

    return maiorMenorDist;
}
```

GRAFO LISTA

```
Lucas Rocha, 14 hours ago | 4 authors (Lucas Rocha and others)
class GrafoLista : public Grafo {
protected:
    ListaAdjVertice* listaAdjVertices;
public:
    // Construtor e Destrutor
    GrafoLista(int numVertices, bool direcionado, bool ponderadoVertices, bool po
    ~GrafoLista();

    // Funcoes auxiliares
    int get_num_vizinhos(int id) override;
    void dfs(int id, bool* visitado) override;
    bool existe_vertice(int id) override;

    // Funcoes de maniplacao de vertices e arestas
    void adicionar_vertice(int id, float peso = 0.0) override;
    void adicionar_aresta(int origem, int destino, float peso = 1.0) override;
    void remover_vertice(int id) override;
    void remover_aresta(int origem, int destino) override;
    void remover_primeira_aresta(int id) override;
```



ADICIONAR VERTICE

```
// Adiciona um vertice a lista
void ListaAdjVertice::adicionar_vertice(int id, float peso) {
    NoVertice* novoNo = new NoVertice(id, peso);
    novoNo->setProximo(this->cabeca);
    this->cabeca = novoNo;

    //cout << "Adicionado Vertice " << novoNo->getIdVertice() << endl;
}
```

GRAFO LISTA REMOVER VERTICE

```cpp
// Remove um vertice da lista
void ListaAdjVertice::remover_vertice(int id) {
    NoVertice* atual = this->cabeca;
    NoVertice* anterior = nullptr;
    NoVertice* remover =  nullptr;
    while (atual != nullptr) {
        if (atual->getIdVertice() == id) {
            if (anterior == nullptr) {
                this->cabeca = atual->getProximo();
            } else {
                anterior->setProximo(atual->getProximo());
            }
            remover = atual;
        }
        // Remove as arestas que apontam para o vertice a ser removido
        atual->remover_aresta(id);
        // Atualiza os ponteiros
        anterior = atual;
        atual = atual->getProximo();
    }
    // Remove o vertice
    //cout << "Removendo vertice " << remover->getIdVertice() << endl;
    delete remover;

    // Recalculando ID dos vertices
    atual = this->cabeca;
    while (atual != nullptr) {
        if (atual->getIdVertice() > id) {
            atual->setIdVertice(atual->getIdVertice() - 1);
        }
        atual = atual->getProximo();
    }

    // Reorganizando as arestas
    atual = this->cabeca;
    while (atual != nullptr) {
        NoAresta* arestaAtual = atual->getArestas()->getCabeca();
        while (arestaAtual != nullptr) {
            if (arestaAtual->getDestino() > id) {···
            arestaAtual = arestaAtual->getProximo();
        }
        atual = atual->getProximo();
    }
```

GRAFO LISTA ADICIONAR / REMOVER ARESTA

```cpp
// Adiciona uma aresta a lista
void ListaAdjAresta::adicionar_aresta(int origem, int destino, float peso) {
    // Verifica se a aresta ja existe
    NoAresta* atual = this->cabeca;
    while (atual != nullptr) {
        if (atual->getOrigem() == origem && atual->getDestino() == destino) {
            //cout << "Erro: Aresta " << origem << " -> " << destino << " ja ex
            return;
        }
        atual = atual->getProximo();
    }

    // Adiciona uma nova aresta
    NoAresta* novaAresta = new NoAresta(origem, destino, peso);
    novaAresta->setProximo(this->cabeca);
    this->cabeca = novaAresta;
    //cout << "Adicionada Aresta " << novaAresta->getOrigem() << " -> " << nova
}

// Remove uma aresta da lista
void ListaAdjAresta::remover_aresta(int origem, int destino) {
    NoAresta* atual = this->cabeca;
    NoAresta* anterior = nullptr;
    while (atual != nullptr) {
        if (atual->getOrigem() == origem && atual->getDestino() == destino) {
            if (anterior == nullptr) {
                this->cabeca = atual->getProximo();
            } else {
                anterior->setProximo(atual->getProximo());
            }
            delete atual;
            //cout << "Removida Aresta " << origem << " -> " << destino << endl
            return;
        }
        anterior = atual;
        atual = atual->getProximo();
    }
    //cout << "Erro: Aresta " << origem << " -> " << destino << " nao existe."
}
```

GRAFO LISTA

CALCULAR MENOR_DIST

```cpp
int GrafoLista::calcula_menor_dist(int origem, int destino) {
    const int INF = 1000000;
    int dist[numVertices + 1];
    bool visitado[numVertices + 1];

    for (int i = 1; i <= numVertices; i++) {
        dist[i] = INF;
        visitado[i] = false;
    }

    dist[origem] = 0;

    for (int i = 1; i <= numVertices; i++) {
        int u = -1;
        for (int j = 1; j <= numVertices; j++) {
            if (!visitado[j] && (u == -1 || dist[j] < dist[u])) {
                u = j;
            }
        }

        if (dist[u] == INF) {
            break;
        }

        visitado[u] = true;
        NoAresta* atual = listaAdjVertices->getVertice(u)->getArestas()->getCabeca();
        while (atual != nullptr) {
            int v = atual->getDestino();
            float peso = atual->getPeso();
            if (dist[u] + peso < dist[v]) {
                dist[v] = dist[u] + peso;
            }
            atual = atual->getProximo();
        }
    }
    return (dist[destino] == INF) ? -1 : dist[destino];
}
```

CALCULAR MAIOR MENOR_DIST

```cpp
int Grafo::calcula_maior_menor_dist() {
    int maiorMenorDist = 0;
    int verticeOrigem = -1;
    int verticeDestino = -1;

    for (int i = 1; i <= numVertices; i++) {
        for (int j = 1; j <= numVertices; j++) {
            if (i != j) {
                int menorDist = calcula_menor_dist(i, j);
                if (menorDist != 1000000 && menorDist > maiorMenorDist) {
                    maiorMenorDist = menorDist;
                    verticeOrigem = i;
                    verticeDestino = j;
                }
            }
        }
    }

    if (verticeOrigem != -1 && verticeDestino != -1) {
        cout << "Maior menor distancia: (" << verticeOrigem << "-" << verticeDestino << ") = " << maiorMenorDist << endl;
    } else {
        cout << "Nao ha caminhos validos no grafo." << endl;
    }

    return maiorMenorDist;
}
```

ALUNOS

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