PCO001
ALGORITMOS E ESTRUTURAS DE DADOS

Seminário - Grafos

Carlos Henrique Reis

Universidade Federal de Itajubá - UNIFEI

31 de maio de 2024

Sumário

Considerações Iniciais

Considerações Iniciais

- Em diversas aplicações computacionais, surge a necessidade de **modelar**, representar e analisar algum conjunto de conexões entre pares de objetos.

Considerações Iniciais

- Em diversas aplicações computacionais, surge a necessidade de **modelar**, representar e analisar algum conjunto de conexões entre pares de objetos.
- Para lidar com essas questões, a Ciência da Computação oferece uma estrutura fundamental: o grafo. Grafos são abstrações matemáticas que modelam conjuntos de objetos (vértices) e suas conexões (arestas), permitindo representar e analisar relacionamentos complexos de forma eficiente.

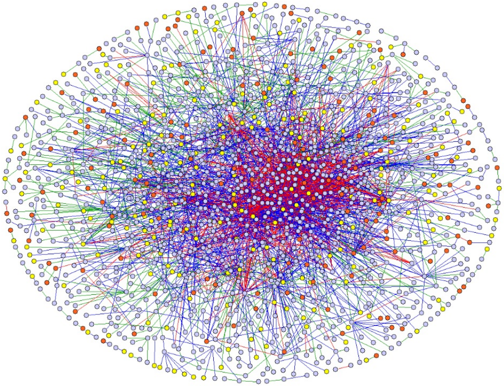


Figura: Exemplo modelagem em grafos

Considerações Iniciais

Como o Google Maps consegue calcular qual é a menor distância entre a minha casa e o supermercado?

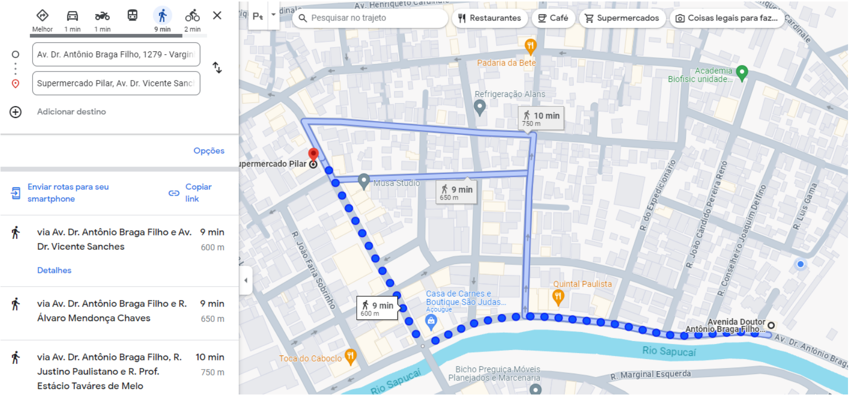


Figura: Exemplo caminho ideal entre dois pontos

Considerações Iniciais

- Uma das formas seria modelar a rede de ruas, levando em conta distância, direção ou o tempo estimado para percorrer aquele trecho, ou seja, podemos tratar as esquinas como nossos objetos e as ruas como conjunto de conexões.
- Assim o aplicativo pode utilizar algoritmos poderosos para encontrar o caminho com o menor peso total, ou seja, o trajeto mais eficiente entre a minha casa e o supermercado.

Figura: Exemplo da modelagem de um grafo

Definições

Definições

- Praticamente qualquer objeto pode ser representado como um grafo.
- Um grafo $G(V, A)$ é definido por dois conjuntos
  - Conjunto V de vértices (não vazio): Objetos simples que podem ter nome e outros atributos.
  - Conjunto A de arestas: Utilizadas para conectar pares de vértices.

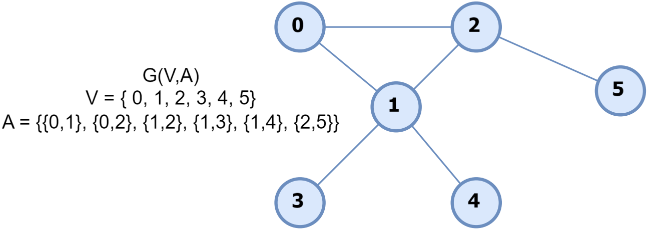


Figura: Representação matemática de um grafo

Definições

- **Vértice** é cada um dos itens representados no grafo.
  - O seu significado depende da natureza do problema modelado como: Pessoas, uma tarefa em um projeto, lugares em um mapa, etc.
- **Aresta (ou arco)** liga dois vértices e diz qual a relação entre eles
  - Dois vértices são **adjacentes** se existir uma aresta ligando eles: Pessoas (parentesco entre elas ou amizade), tarefas de um projeto (pré-requisito entre as tarefas), lugares de um mapa (estradas que existem ligando os lugares), etc

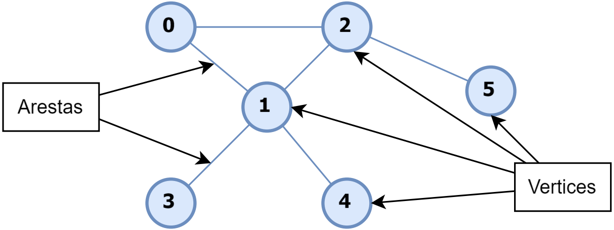


Figura: Vértices e arestas de um grafo

Tipos de arestas

- **Direcionadas**: A conexão tem uma direção específica, indicando um fluxo ou relação assimétrica.
- **Não direcionadas**: A conexão não tem direção, representando uma relação simétrica.
- **Ponderadas**: As arestas possuem um valor numérico (peso) associado, que pode representar custo, distância, capacidade, etc.

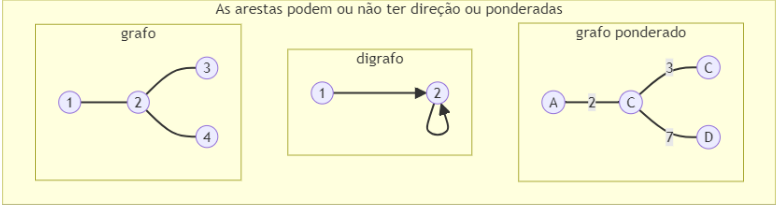
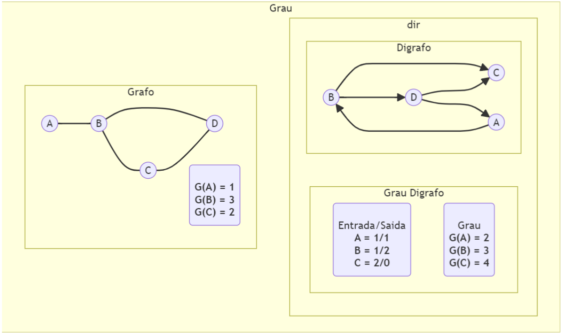


Figura: Exemplos de arestas

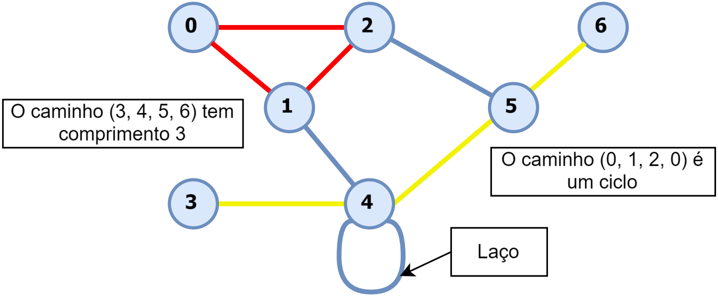
Grau de um Vértice

- Em grafos não direcionados
  - O grau de um vértice indica quantas arestas estão conectadas a ele. Em outras palavras, é o número de vizinhos que o vértice possui.
- Grau em Dígrafos (Grafos Direcionados):
  - O grau de um vértice é o número de arestas que saem dele (out-degree) mais o número de arestas que chegam nele (in-degree).



Ciclos, caminhos e laços

- Um laço é uma aresta que conecta um vértice a ele mesmo. Laços podem representar auto referências ou loops em um sistema
- Um caminho é uma sequência de vértices conectados por arestas. Em um caminho válido, cada vértice é adjacente ao próximo vértice na sequência.
- O comprimento de um caminho é definido pelo número de arestas que o compõem (ou, equivalentemente, o número de vértices menos 1).
- Um ciclo é um caminho que começa e termina no mesmo vértice. Ou seja, é um caminho fechado.
- OBS: A principal diferença entre um caminho e um ciclo é que um caminho tem pontos inicial e final distintos, enquanto um ciclo começa e termina no mesmo vértice.



Tipos de Grafos

Tipos de Grafos

Grafo trivial

- Possui um único vértice e nenhuma aresta

Grafo simples

- Grafo não direcionado, sem laços e sem arestas paralelas (multigrafo)

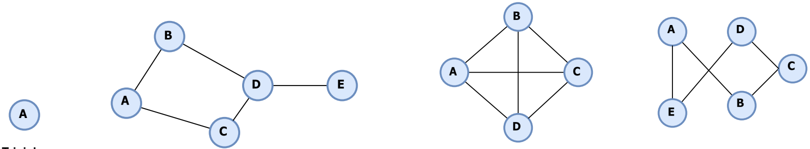
Grafo completo

- Grafo simples onde cada vértice se conecta a todos os outros vértices do grafo.

Grafo regular

- Grafo onde todos os seus vértices possuem o mesmo grau (número de arestas ligadas a ele)

Todo grafo completo é também regular



Trivial

Simples

Completo

Regular (Grau = 2)

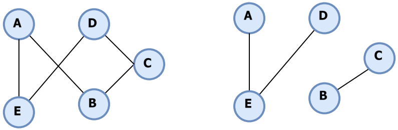
Tipos de Grafos

Subgrafo

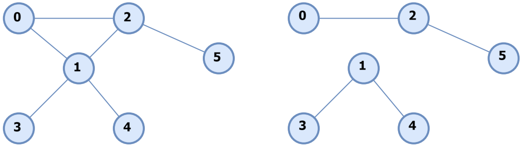
- Um grafo menor contido dentro de um grafo maior. Usado para analisar partes específicas de um grafo.

Grafo conexo e desconexo

- Grafo conexo: existe um caminho ligando quaisquer dois vértices.
- Quando isso não acontece, temos um grafo desconexo



Subgrafo induzido pelo conjunto de vértices {A, B, C, D, E}



Conexo

Desconexo

Tipos de Grafos

Grafo bipartido

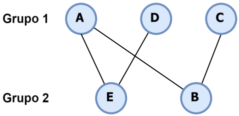
- Vértices divididos em dois conjuntos, com arestas conectando apenas vértices de conjuntos diferentes. Modela relações entre dois tipos de entidades.

Grafos isomorfos

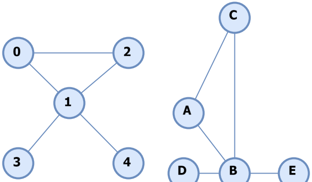
- Mesma estrutura, mas com diferentes rótulos nos vértices e arestas. Essencialmente, o mesmo grafo rearranjado e "renomeado".

Grafos Ponderados

- Arestas possuem pesos numéricos, representando custos, distâncias ou outras métricas.



Bipartido



Grafos Isomorfos

Grau	
G(0) = 2	G(A) = G(0)
G(1) = 4	G(B) = G(1)
G(2) = 2	G(C) = G(2)
G(3) = 1	G(D) = G(3)
G(4) = 1	G(E) = G(4)

Tipos de Grafos

Grafo Euleriano

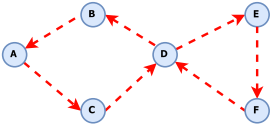
- Possui um ciclo que visita todas as arestas exatamente uma vez. Útil para problemas de roteamento.

Grafo Semi-Euleriano

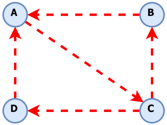
- Possui um caminho (não necessariamente um ciclo) que visita todas as arestas exatamente uma vez.

Grafo Hamiltoniano

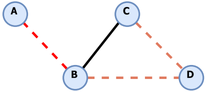
- Possui um caminho que visita todos os vértices exatamente uma vez.



Grafo Euleriano - Caminho D-E-F-D-B-A-C-D



Grafo Semi Euleriano - Caminho C - D - A - C - B - A



Grafo Hamiltoniano

Representação

Representação

Existem diferentes maneiras de representar um grafo, as mais comuns são a **Matriz de adjacência** e **Lista de adjacência**.

Representação

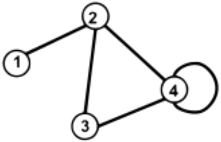
Existem diferentes maneiras de representar um grafo, as mais comuns são a **Matriz de adjacência** e **Lista de adjacência**.

Matriz de adjacência

- Estrutura: Matriz *NxN* (N = número de vértices).
- Arestas: Representadas por marcas na posição (i, j) da matriz, indicando conexão entre vértices i e j.
- Custo Computacional: $O(N^2)$ - alto custo, especialmente para grafos com muitos vértices.

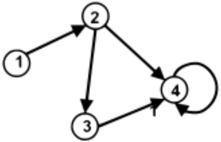
Matriz de adjacência

GRAFO



	1	2	3	4
1	0	1	0	0
2	1	0	1	1
3	0	1	0	1
4	0	1	1	1

DIGRAFO



	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	0	0	0	1

Representação

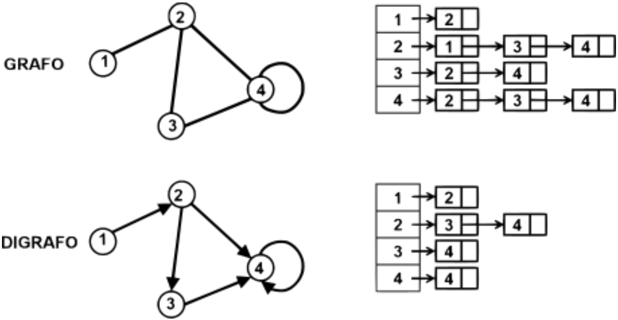
Existem diferentes maneiras de representar um grafo, as mais comuns são a **Matriz de adjacência** e **Lista de adjacência**.

Representação

Existem diferentes maneiras de representar um grafo, as mais comuns são a **Matriz de adjacência** e **Lista de adjacência**.

Lista de adjacência

- Estrutura: Cada vértice possui uma lista contendo seus vértices adjacentes (vizinhos).
- Arestas: Representadas pela presença de um vértice na lista de outro.
- Custo Computacional: $O(V + E)$ - mais eficiente para grafos esparsos (V = vértices, E = arestas).



TAD Grafo

TAD Grafo

- Importante considerar os algoritmos em grafos como tipos abstratos de dados.
- Conjunto de operações associado a uma estrutura de dados.
- Independência de implementação para as operações.
- Devemos levar em conta o tipo de representação que iremos trabalhar
- Para nossa implementação iremos usar a representação por Lista de adjacência

Implementação

```
// Arquivo: Grafo.h
typedef struct Grafo *GrafoPtr;

// Arquivo: Grafo.cpp
#include "Grafo.h" // Inclui a definicao da TAD Grafo

// Definicao da estrutura Grafo
struct Grafo {
    int NumVertices;
    int NumArestas;
    bool orientado;
    int *Grau;
    NoPtr *ListaAdj;
};

// Programa principal
GrafoPtr G; // Declaracao de um ponteiro para a estrutura Grafo
```

Algumas operações sobre a TAD Grafo

1. *IniciaGrafo*(*Grafo*, *N*, *Ponderado*): Cria um grafo vazio.
2. *InsereAresta*(*Grafo*, *V*1, *V*2, *Peso*): Insere uma aresta no grafo.
3. *ExisteAresta*(*Grafo*, *V*1, *V*2): Verifica se existe uma determinada aresta.
4. *ListaAdjacencia*(*Grafo*, *V*): Obtém a lista de vértices adjacentes a um determinado vértice
5. *RetiraAresta*(*Grafo*, *V*1, *V*2): Retira uma aresta do grafo.
6. *ImprimeGrafo*(*Grafo*): Imprime um grafo.
7. *LiberaGrafo*(*Grafo*): Liberar o espaço ocupado por um grafo.
8. *GrauVertice*(*Grafo*, *V*): Retorna o número de arestas incidentes (o número de vizinhos)
9. *BuscaEmLargura*(*Grafo*, *VInical*): Realiza uma Busca em Largura (BFS) a partir do vértice V
10. *BuscaEmProfundidade*(*Grafo*, *VInicial*): Realiza uma Busca em Profundidade (DFS)
11. *MenorCaminho*(*Grafo*, *VInicial*): Encontra o caminho de menor custo (Dijkstra)

Implementação - Criando Grafo

```
// Arquivo Grafo.h
#include "ListaDinEncadeada.h"// Tad Auxiliar

void InicializaGrafo(GrafoPtr &, int, bool);

// Grafo.cpp
void InicializaGrafo(GrafoPtr &G, int N, bool orientado) {
    G = new Grafo;
    G->NumVertices = N;
    G->NumArestas = 0;
    G->orientado = orientado;
    G->Grau = new int[N];
    G->ListaAdj = new NoPtr[N];
    for (int i = 0; i < N; i++) {
        G->Grau[i] = 0;
        IniciaLista(G->ListaAdj[i]);
    }
}

// Programa principal
GrafoPtr G; // Declaracaoo de uma variavel do tipo Grafo
InicializaGrafo(G, 4, false); // Inicializa o grafo com 4 vertices e nao orientado
```

Implementação - Liberando o grafo

```
// Arquivo Grafo.h
void LiberaGrafo(GrafoPtr &);

// Programa principal
LiberaGrafo(G); // Libera a memoria alocada para o grafo

// Arquivo Grafo.cpp
void LiberaGrafo(GrafoPtr &G) {
    if (G == nullptr) {
        return;
    }
    for (int i = 0; i < G->NumVertices; i++) {
        LiberaLista(G->ListaAdj[i]);
    }
    delete[] G->Grau;
    delete[] G->ListaAdj;
    delete G;
}
```

Implementação - Inserindo aresta

```
// Arquivo Grafo.h
void InsereAresta(GrafoPtr &, int, int, float);

// Programa principal
GrafoPtr G;
int N = 4;

InicializaGrafo(G, N, false);
InsereAresta(G,0,1,0);
InsereAresta(G,1,3,0);
InsereAresta(G,2,4,0);
InsereAresta(G,1,4,0);

LiberaGrafo(G);

//CONTINUA...
```

Implementação - Inserindo aresta

```cpp
// Arquivo Grafo.cpp
void InsereAresta(GrafoPtr &G, int Origem, int Destino, float Peso) {
    if (G == nullptr) {
        return;
    } else if (Origem < 0 || Origem >= G->NumVertices) {
        return;
    } else if (Destino < 0 || Destino >= G->NumVertices) {
        return;
    }

    InsereOrdenadoLista(G->ListaAdj[Origem], Destino, Peso);
    G->Grau[Origem]++;
    G->Grau[Destino]++;
    if (!G->orientado && Origem != Destino) {
        InsereOrdenadoLista(G->ListaAdj[Destino], Origem, Peso);
    }
    G->NumArestas++;
}
```

Implementação - Retira Aresta

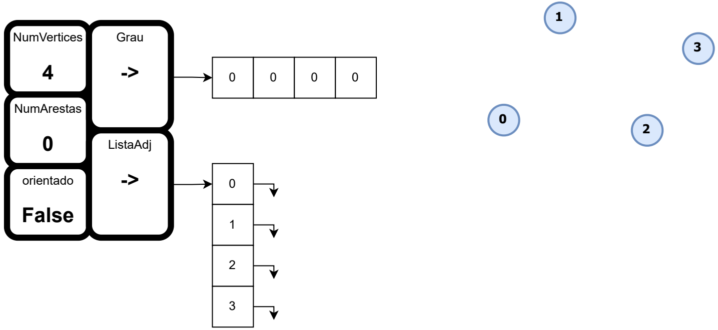
```
// Arquivo Grafo.h
void RetiraAresta(GrafoPtr &, int, int);

// Arquivo Grafo.cpp
void RetiraAresta(GrafoPtr &G, int Origem, int Destino) {
    if (G == nullptr) {
        return;
    } else if (Origem < 0 || Origem >= G->NumVertices) {
        return;
    } else if (Destino < 0 || Destino >= G->NumVertices) {
        return;
    }

    if (RetiraLista(G->ListaAdj[Origem], Destino)) {
        G->Grau[Origem]--;
        G->Grau[Destino]--;
        if (!G->orientado) {
            RetiraLista(G->ListaAdj[Destino], Origem);
        }
        G->NumArestas--;
    }
}

// Programa principal
GrafoPtr G;
InicializaGrafo(G, 2, true); // Inicializa o grafo com 2 vertices e orientado

InsereAresta(G,0,1,0);
RetiraAresta(G,0,1,0);
```

Implementação - Imprime Aresta e Grau Vértice

```
void ImprimeGrafo(GrafoPtr G) {
    if (G == nullptr) {
        return;
    }

    for (int i = 0; i < G->NumVertices; i++) {
        cout << i << " -> ";
        ImprimeLista(ListaAdjacencia(G, i));
        cout << endl;
    }
}


int GrauVertice(GrafoPtr G, int Vertice) {
    if (G == nullptr) {
        return -1;
    } else if (Vertice < 0 || Vertice >= G->NumVertices) {
        return -1;
    }

    return G->Grau[Vertice];
}
```

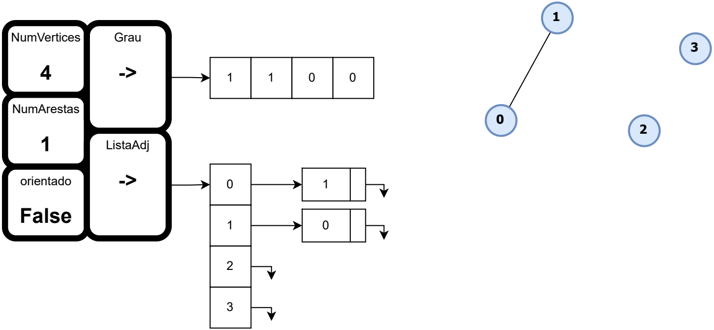
Teste Mesa

```
InicializaGrafo(G, 4, false);
```



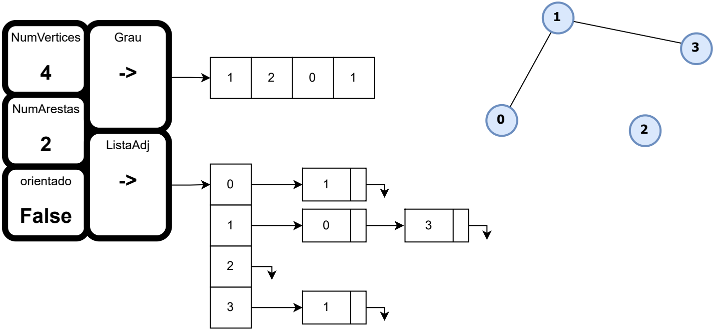
Teste Mesa

```
InsereAresta(G, 0, 1, 0);
```



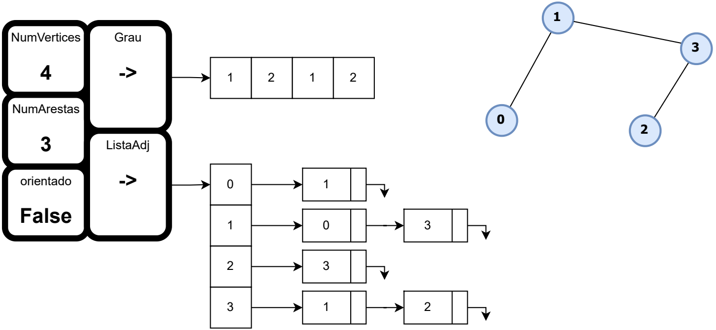
Teste Mesa

```
InsereAresta(G, 0, 1, 0);
InsereAresta(G, 1, 3, 0);
```



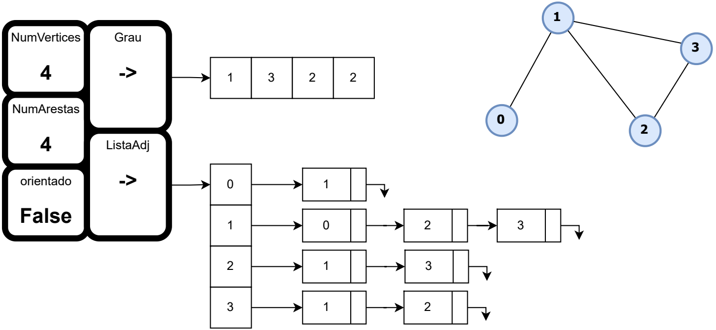
Teste Mesa

```
InsereAresta(G, 0, 1, 0);
InsereAresta(G, 1, 3, 0);
InsereAresta(G, 2, 3, 0);
```



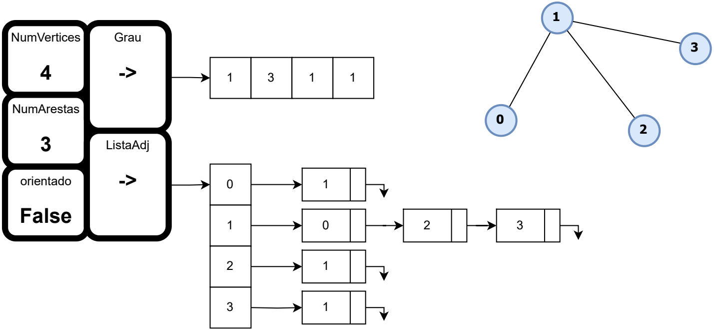
Teste Mesa

```
InsereAresta(G, 0, 1, 0);
InsereAresta(G, 1, 3, 0);
InsereAresta(G, 2, 3, 0);
InsereAresta(G, 2, 1, 0);
```



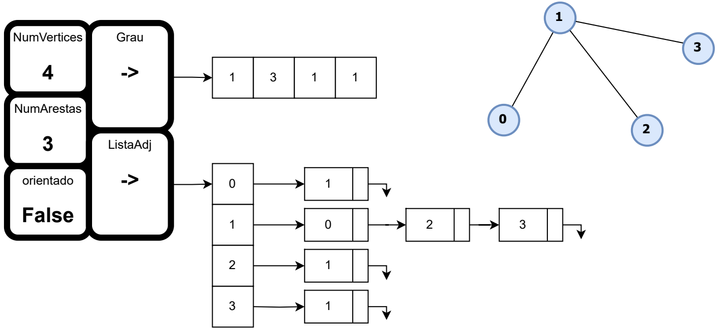
Teste Mesa

```
RetiraAresta(G, 2, 3, 0);
```



Teste Mesa

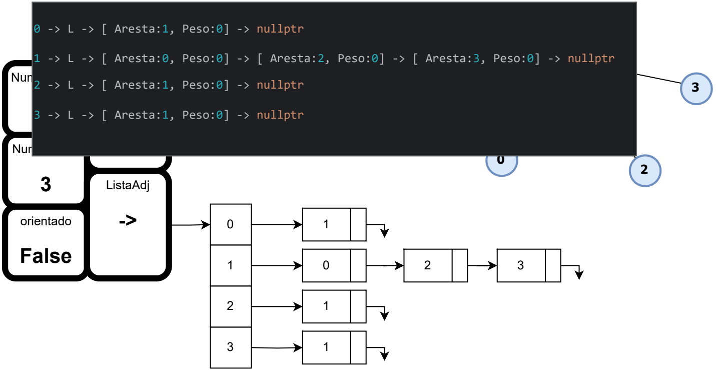
```
ImprimeGrafo(G);
```



Teste Mesa

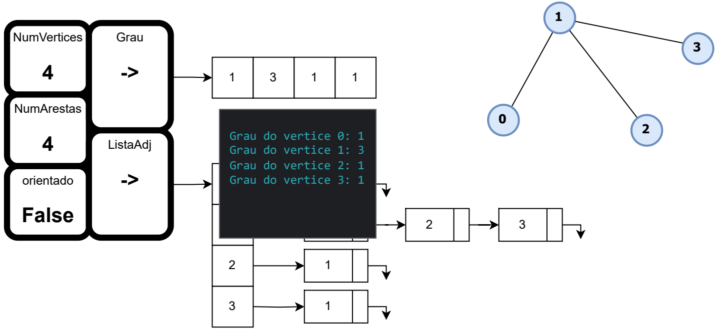
```
ImprimeGrafo(G);
```

```
0 -> L -> [ Aresta:1, Peso:0] -> nullptr
1 -> L -> [ Aresta:0, Peso:0] -> [ Aresta:2, Peso:0] -> [ Aresta:3, Peso:0] -> nullptr
2 -> L -> [ Aresta:1, Peso:0] -> nullptr
3 -> L -> [ Aresta:1, Peso:0] -> nullptr
```



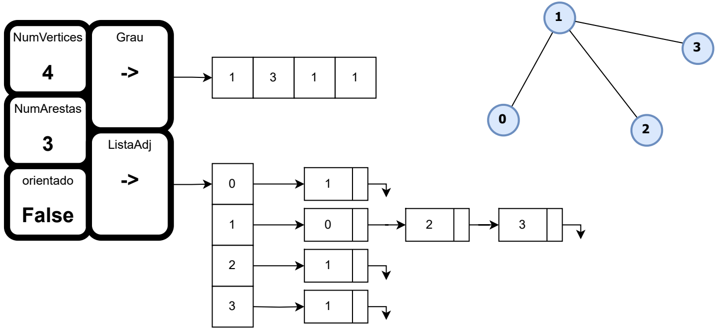
Teste Mesa

```
for (int i = 0; i < 4; ++i) {
  cout << "Grau do vertice " << i << ": " << GrauVertice(G, i) << endl;
}
```



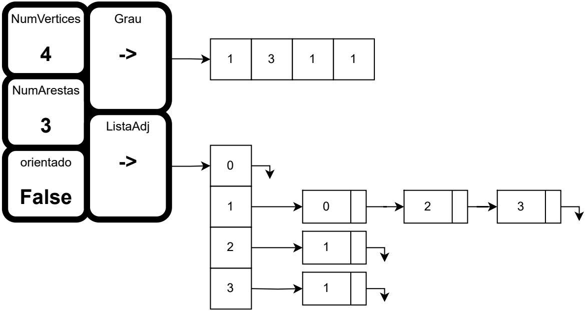
Teste Mesa

```
LiberaGrafo(G);
```



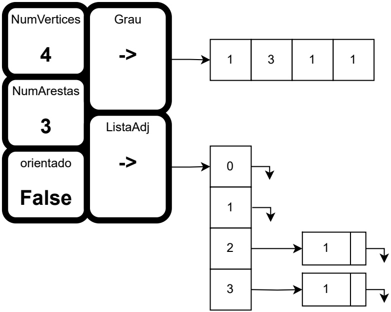
Teste Mesa

```
LiberaGrafo(G);
(libera as listas de adjacências)
```



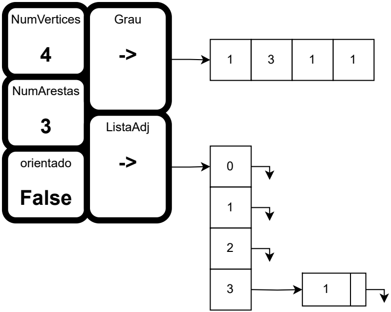
Teste Mesa

```
LiberaGrafo(G);
(libera as listas de adjacências)
```



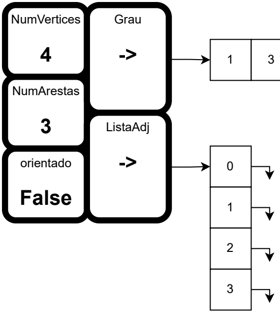
Teste Mesa

`LiberaGrafo(G);`
`(libera as listas de adjacências)`



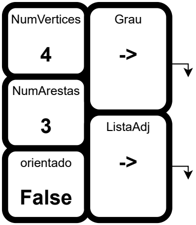
Teste Mesa

```
LiberaGrafo(G);
(libera as listas de adjacências)
```



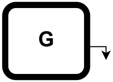
Teste Mesa

```
LiberaGrafo(G);
```



Teste Mesa

```
LiberaGrafo(G);
```



Busca em Grafos

Busca em Grafos

- Consiste em explorar um grafo
- Buscas em grafos são processos algorítmicos que exploram sistematicamente os nós e arestas de um grafo.
- O objetivo é encontrar um caminho entre dois nós específicos, um nó específico ou todos os nós do grafo.
- Vários problemas em grafos podem ser resolvidos efetuando uma busca
- A busca pode visitar todos ou apenas um subconjunto dos vértices.

Busca em Grafos

Existem vários tipos de algoritmos de busca que podemos realizar em um grafo. Os três principais:

- Busca em largura
- Busca em profundidade
- Busca pelo menor caminho

Busca em largura (BFS)

- Objetivo explorar um grafo de forma sistemática, visitando todos os nós em ordem crescente de distância do nó inicial.
- Partindo de um vértice inicial, a busca explora todos os vizinhos de um vértice. Em seguida, para cada vértice vizinho, ela repete esse processo, visitando os vértices ainda inexplorados.
- Pode ser usado para:
  1. Encontrar componentes conectados
  2. Encontrar todos os vértices conectados a apenas um componente
  3. Encontrar menor caminho entre dois vértices
  4. Testar bipartição em grafos

Busca em largura (BFS) - Algoritmo

- Esse algoritmo faz uso do conceito de fila
- O grafo é percorrido de maneira sistemática, primeiro marcando como "visitados"todos os vizinhos de um vértice e em seguida começa a visitar os vizinhos de cada vértice na ordem em que eles foram marcados.
- Para realizar essa tarefa, uma fila é utilizada para administrar a visitação dos vértices.

```
Algoritmo BuscaEmLargura(G, s)
    Entrada: "Grafo G, vertice inicial s"
    Saida: "Vertices de G numerados em ordem de distancia de s"

        Inicializa todos os vertices de G como nao visitados

        Fila <- {s}
        Numere s

        Enquanto Fila nao estiver vazia Faca
            Retire um vertice v da Fila
            Para cada vizinho w de v Faca
                Se w nao estiver numerado Entao
                    Numere w
                    Ponha w na Fila
```

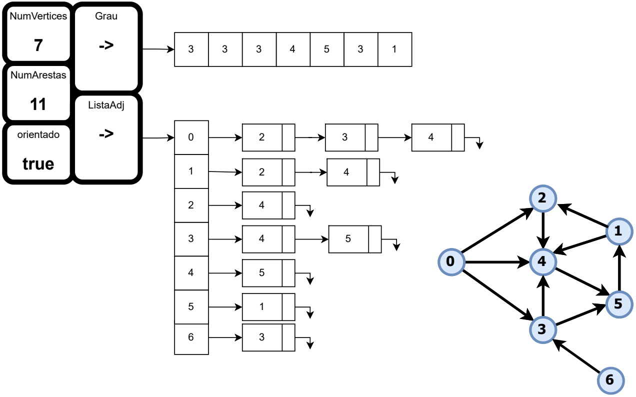
Busca em largura (BFS) - Algoritmo

```
void BuscaEmLargura(GrafoPtr G, int Vertice, int *Visitado) {
    FilaPtr F;
    IniciaFila(F, G->NumVertices);
    for (int i = 0; i < G->NumVertices; i++) {
        Visitado[i] = -1;// Inicializa todos os vertices como nao visitados
    }

    Visitado[Vertice] = 0;
    Enfileira(F, Vertice);

    while (!FilaVazia(F)) {
        int V;
        RetiraFila(F, V);
        NoPtr Aux = ListaAdjacencia(G, V);
        while (Aux != nullptr) {
            int Aresta;
            float Peso;
            NoPtr Lig;
            retornaElemento(Aux, Aresta, Peso, Lig);
            if (Visitado[Aresta] == -1) {
                Visitado[Aresta] = Visitado[V] + 1;
                Enfileira(F, Aresta);
            }
            Aux = Lig;
        }
    }
    LiberaFila(F);
}
```

Teste Mesa

Considere o grafo G definido pelos arcos 0 – 2, 0 – 3, 0 – 4, 1 – 2, 1 – 4, 2 – 4, 3 – 4, 3 – 5, 4 – 5, 5 – 1

```
InicializaGrafo(G, 7, true);
```



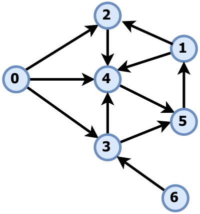
Teste Mesa

Considere o grafo G definido pelos arcos 0 – 2, 0 – 3, 0 – 4, 1 – 2, 1 – 4, 2 – 4, 3 – 4, 3 – 5, 4 – 5, 5 – 1

```
BuscaEmLargura(G, 0, Distancias);
```

Fila				
0				
2	3	4		
	3	4		
		4	5	
			5	
				1
				6
Fila vazia!				

Distancias						
0	1	2	3	4	5	6
0	-1	-1	-1	-1	-1	-1
0	-1	1	1	1	-1	-1
0	-1	1	1	1	-1	-1
0	-1	1	1	1	2	-1
0	-1	1	1	1	2	-1
0	3	1	1	1	2	-1
0	3	1	1	1	2	-1
0	3	1	1	1	2	-1



Busca em profundidade DFS

- Objetivo é visitar todos os vértices e numerá-los na ordem em que são descobertos.
- A busca em profundidade não resolve um problema específico. Ela ajuda a compreender o grafo com que estamos lidando, revelando sua forma e reunindo informações (representadas pela numeração dos vértices) que ajudam a responder perguntas sobre o grafo. .
- Pode ser usado para:
  1. Encontrar componentes conectados e fortemente conectados;
  2. Ordenação topológica de um grafo;
  3. Procurar a saída de um labirinto;
  4. Verificar se um grafo é completamente conexo

Busca em profundidade - Algoritmo

- Na busca em profundidade, as arestas são exploradas a partir do vértice inicial *v* mais recentemente descoberto que ainda tem arestas não exploradas saindo dele.
- Quando todas as arestas de v são exploradas, a busca volta ao vértice anterior a v (backtracking) para seguir arestas ainda não exploradas.
- Ideia é identificar os caminhos a partir do vértice inicial.

Busca em profundidade - Algoritmo

Função BuscaEmProfundidade

- Esta função é a interface para a busca em profundidade.
- Recebe o grafo e o vértice inicial.
- Inicializa o vetor de distâncias com -1.
- o Chama a função **BuscaEmProfundidadeRecursiva** para iniciar a busca a partir do vértice inicial.

```
void BuscaEmProfundidade(GrafoPtr G, int VerticeInicial, int *Profundidade) {
    if (G == nullptr) {
        return;
    } else if (VerticeInicial < 0 || VerticeInicial >= G->NumVertices) {
        return;
    }

    // Inicializa distancias
    for (int i = 0; i < G->NumVertices; i++) {
        Profundidade[i] = -1; //define todos os vertices como nao visitados
    }

    int cont = 0;
    BuscaEmProfundidadeRecursiva(G, VerticeInicial, Profundidade, cont);
}
```

Busca em profundidade - Algoritmo

Função BuscaEmProfundidadeRecursiva

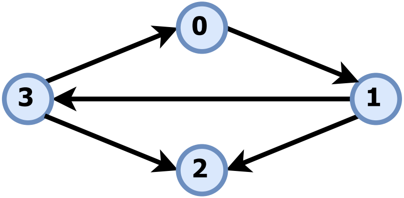
- Esta função recursiva realiza a busca em profundidade.
- Ela recebe o grafo, o vértice atual, o vetor de distâncias e um contador para a distância.
- O vértice atual é marcado com a distância atual.
- A função itera sobre os vértices adjacentes ao vértice atual.
- Se um vértice adjacente ainda não foi visitado (distância -1), a função é chamada recursivamente para esse vértice.

```
void BuscaEmProfundidadeRecursiva(GrafoPtr G, int Vertice, int *Profundidade, int &
      cont) {
    Profundidade[Vertice] = cont++;

    NoPtr Aux = ListaAdjacencia(G, Vertice);
    while (Aux != nullptr) {
        int Aresta;
        float Peso;
        NoPtr Lig;
        retornaElemento(Aux, Aresta, Peso, Lig);
        if (Profundidade[Aresta] == -1) {
            BuscaEmProfundidadeRecursiva(G, Aresta, Profundidade, cont);
        }
        Aux = Lig;
    }
}
```

Teste Mesa

Supondo essa implementação

```
GrafoPtr G;
int N = 4;

InicializaGrafo(G, N, true);

InsereAresta(G,0,1,0);
InsereAresta(G,1,2,0);
InsereAresta(G,1,3,0);
InsereAresta(G,3,2,0);
InsereAresta(G,3,0,0);

int *Profundidades = new int[N];
BuscaEmProfundidade(G, 0, Profundidades);
```



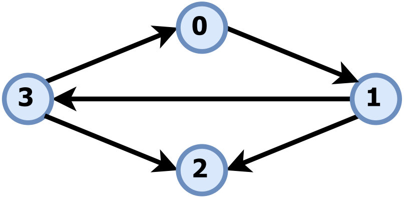
Teste Mesa

Simulação da DFS:

1. **Inicialização**: $Profundidades = [-1, -1, -1, -1]$, $cont = 0$, $VerticeInicial = 0$.
2. **DFS(0)**:
   - $Profundidades[0] = 0$ (cont incrementado para 1)
   - Explora vértice 1 (adjacente a 0)
3. **DFS(1)**:
   - $Profundidades[1] = 1$ (cont incrementado para 2)
   - Explora vértice 2 (adjacente a 1)
4. **DFS(2)**:
   - $Profundidades[2] = 2$ (cont incrementado para 3)
   - Sem vértices adjacentes não visitados. **Retorna para DFS(1)**
5. **DFS(1) continua**:
   - Explora vértice 3 (adjacente a 1)
6. **DFS(3)**:
   - $Profundidades[3] = 3$ (cont incrementado para 4)
   - Vértice 2 já foi visitado
   - Vértice 0 já foi visitado. Retorna para DFS(1
7. **DFS(1) continua**: Todos os vértices adjacentes a 1 foram explorados. Retorna para DFS(0).
8. **DFS(0) continua**: Todos os vértices adjacentes a 0 foram explorados. Finaliza a DFS.

Teste Mesa

Resultado e analise do vetor de profundidade: $Profundidades = [0, 1, 3, 2]$



- Ordem de Exploração
  1. O vértice 0 foi visitado primeiro ($Profundidade[0] = 0$)
  2. O vértice 1 foi visitado em segundo ($Profundidade[1] = 1$)
  3. O vértice 3 foi visitado em terceiro ($Profundidade[3] = 2$)
  4. O vértice 2 foi visitado por último ($Profundidade[2] = 3$)
- para uma análise mais completa é necessário, que durante a execução da DFS, registrar outros dados auxiliares como o pai de cada vértice na árvore.

Busca pelo menor caminho

- Partindo de um vértice inicial, calcula a menor distância desse vértice a todos os demais (Desde que exista um caminho entre eles)
- O comprimento pode ser o número de arestas que conectam os dois vértices ou a soma dos pesos das arestas que compõem esse caminho (grafo ponderado)
- Uma das maneiras de achar o menor caminho é utilizando o algoritmo de Dijkstra (um dos algoritmos mais conhecido)

Busca pelo menor caminho

Apresentando Dijkstra

- Nome: Edsger Wybe Dijkstra
- Criação do algoritmo: 1956
- Importância: Um dos algoritmos mais importantes da ciência da computação, usado em diversas aplicações.
- Trabalha com grafos e digrafos, ponderados ou não. No caso de um grafo ponderado, as arestas não podem ter pesos negativos

Passos do Algoritmo

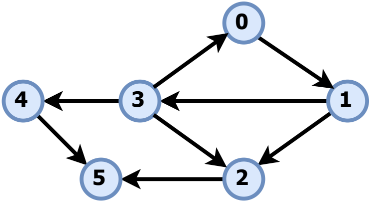
1. **Inicialização:**
   - Atribua a distância do vértice de origem para ele mesmo como 0.
   - Atribua a distância de todos os outros vértices como infinito, pois ainda não sabemos a distância real.
   - Defina o vértice predecessor de todos os vértices como indefinido, pois ainda não sabemos por qual vértice chegamos a cada um.
2. Comece no vértice de origem.
3. Explore os vértices adjacentes (vizinhos) e calcule a distância até eles.
4. Marque o vértice com a menor distância como "visitado".
5. Repita os passos 2 e 3 a partir do vértice visitado, explorando novos vizinhos.
6. Continue até alcançar o vértice de destino.

Passos do Algoritmo

```
// Arquivo: Grafo.h
#define INFINITO 2147483647
int MenorDistancia(int *, bool *, int);
void MenorCaminho(GrafoPtr, int, int *, int *);
void ImprimeCaminho(int *, int);

// Programa principal
GrafoPtr G;
int N = 6;
InicializaGrafo(G, N, true);// Inicializa um grafo com 6 vertices e orientado
InsereAresta(G,0,1,0);
InsereAresta(G,1,2,0);
InsereAresta(G,1,3,0);
InsereAresta(G,3,2,0);
InsereAresta(G,3,0,0);
InsereAresta(G,3,4,0);
InsereAresta(G,4,5,0);
InsereAresta(G,2,5,0);

int *Distancia = new int[N];
int *Precedente = new int[N];

MenorCaminho(G, 0, Distancia, Precedente);
cout << "Distancias e caminhos a partir do vertice 0:" << endl;
for (int i = 0; i < N; ++i) {
    cout << "Vertice " << i << ": Distancia = " << Distancia[i] << ", Caminho = ";
    ImprimeCaminho(Precedente, i);
    cout << i << endl;
}
```

Passos do Algoritmo

```
Distancias e caminhos a partir do vertice 0:
Vertice 0: Distancia = 0, Caminho = 0
Vertice 1: Distancia = 1, Caminho = 0 -> 1
Vertice 2: Distancia = 2, Caminho = 0 -> 1 -> 2
Vertice 3: Distancia = 2, Caminho = 0 -> 1 -> 3
Vertice 4: Distancia = 3, Caminho = 0 -> 1 -> 3 -> 4
Vertice 5: Distancia = 3, Caminho = 0 -> 1 -> 2 -> 5
```



Exemplo Prático com Neo4j

Exemplo Prático com Neo4j

O Neo4j é um banco de dados NoSQL (orientado a grafo) de código aberto que se destaca por sua capacidade de armazenar e consultar dados como um grafo, em vez de tabelas como nos bancos de dados relacionais.

Modelo de Dados em Grafos

- **Nós (Nodes)**: Representação de entidades (vértices).
- **Relacionamentos (Relationships)**: Conexões entre nós (arestas).
- **Propriedades (Properties)**: Atributos que podem ser associados a nós e relacionamentos.

Cypher

Cypher é a linguagem de consulta declarativa de Neo4j e utiliza uma sintaxe inspirada em ASCII-art.

Elementos da Linguagem

1. **Nós**: Representados por parênteses `()`, os nós podem ter propriedades no formato chave:valor. Exemplo:

```
(u:Usuario {nome: "Ana", idade: 30})
```

2. **Relacionamentos**: Representados por setas `->` ou `<-` para indicar a direção, os relacionamentos conectam dois nós e também podem ter propriedades. Exemplo:

```
(a:Filme {titulo: "Matrix"})<-[:ATUOU]-(b:Ator {personagem: "Neo"})
```

3. **Cláusulas**: As cláusulas adicionam lógica às nossas consultas:
   - **MATCH**: Encontra padrões no grafo.
   - **WHERE**: Filtra resultados com base em condições.
   - **RETURN**: Define os dados que queremos retornar.
   - **CREATE**: Cria novos nós e relacionamentos.
   - **SET**: Atualiza propriedades de nós e relacionamentos.
   - **DELETE**: Remove nós e relacionamentos.

Cypher

Exemplos de Consultas em Cypher

1. Criação de Nós e Relacionamentos:

```
CREATE (m:Filme {titulo: "Matrix", ano: 1999})
CREATE (a:Ator {nome: "Keanu Reeves"})
CREATE (m)-[:ATUOU]->(a)
```

2. Consulta de Nós e Relacionamentos:

```
MATCH (a:Ator {nome: "Keanu Reeves"})-[:ATUOU]->(f:Filme)
RETURN f.titulo
```

3. Atualização de Dados:

```
MATCH (f:Filme {titulo: "Matrix"})
SET f.ano = 1998
```

4. Exclusão de Nós e Relacionamentos:

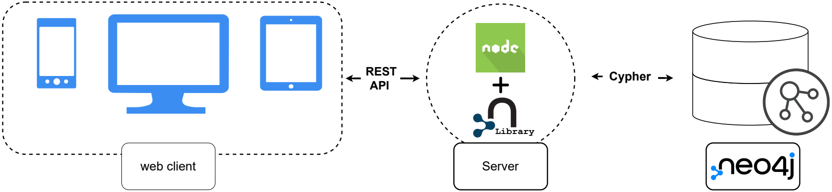
```
MATCH (p:Pessoa {nome: "Alice"})
DETACH DELETE p
```

5. Consulta com cláusula WHERE:

```
MATCH (p:Pessoa)
WHERE p.idade > 25
RETURN p.nome, p.idade
```

Hands on

Para ficar mais visível a aplicabilidade do neo4j, vamos discutir uma solução de uma rede social simples com onde o modelo de dados é baseado em grafos



No caso foi uma solução de rede social simples utilizando Angular para o frontend, nodeJs para backend e Neo4j como banco de dados

Funcionalidades

1. **Login Simples:** Usuários fazem login utilizando apenas o número de matrícula.
2. **Feed de Postagens:** Todos os usuários podem fazer publicações e visualizar as postagens de todos.
3. **Curtidas em Postagens:** Usuários podem curtir postagens.

Hands on

Modelagem no Neo4j

Visão detalhada da modelagem dos dados no Neo4j para nossa aplicação:

1. Nodos (Nodes):
   - **Usuário (User):** Representa cada usuário da rede social.
     - Propriedades: nome, matrícula.
   - **Postagem (Post):** Representa cada postagem no feed.
     - Propriedades: content (conteúdo da postagem), timestamp (data e hora da postagem), id (identificador único).
2. Relações (Relationships):
   - **POSTADO:** Conecta um usuário a uma postagem que ele criou.
     - Indica que um determinado usuário fez uma determinada postagem.
   - **CURTE (LIKES):** Conecta um usuário a uma postagem que ele curtiu.
     - Permite calcular o número de curtidas de cada postagem.

Hands on

Comandos Cypher utilizados no nosso sistema

1. Criação dos Usuários:

```
CREATE (u:User {matricula: "2024100417", nome: "Carlos Henrique Reis"})
```

2. Login(consulta usuário):

```
MATCH (u:User {matricula: "2024100417"}) RETURN u
```

3. Criar uma Postagem:

```
MATCH (u:User {matricula: $userId})
CREATE (u)-[:POSTADO]->(p:Post {id: $postId, content: $content, timestamp:
    timestamp()}) RETURN p
```

4. Obter Todas as Postagens:

```
MATCH (u:User)-[:POSTADO]->(p:Post)
OPTIONAL MATCH (p)<-[l:LIKES]-()
RETURN p, u.nome AS nome, COUNT(l) AS likes
ORDER BY p.timestamp DESC
```

5. Curtir uma Postagem:

```
MATCH (u:User {matricula: $userId}), (p:Post {id: $postId})
CREATE (u)-[:LIKES]->(p) RETURN p
```

Hora de experimentar!

No grafo já foi cadastrado os nodes users:

- Todos os alunos da nossa matéria
- E o professor - 0001

A demonstração está disponível aqui: `https://demo-grafos.carlos-henreis.com.br/`
Acesso é com número de matricula (o do professor é especial 0001)

Referências

Referências

1. Backes, A. (2017). *Estrutura de Dados Descomplicada-em Linguagem C*. Elsevier Brasil.
2. Feofiloff, P. *Algoritmos para Grafos*, `http://www.ime.usp.br/ pf/algoritmos_para_grafos/` (última visita 17/05/2024)
3. LAL, Mahesh. *Neo4j graph data modeling*. Packt Publishing Ltd, 2015.
4. ZIVIANI, Nivio et al. *Projeto de algoritmos: com implementações em Pascal e C*. Cengage, 2011.