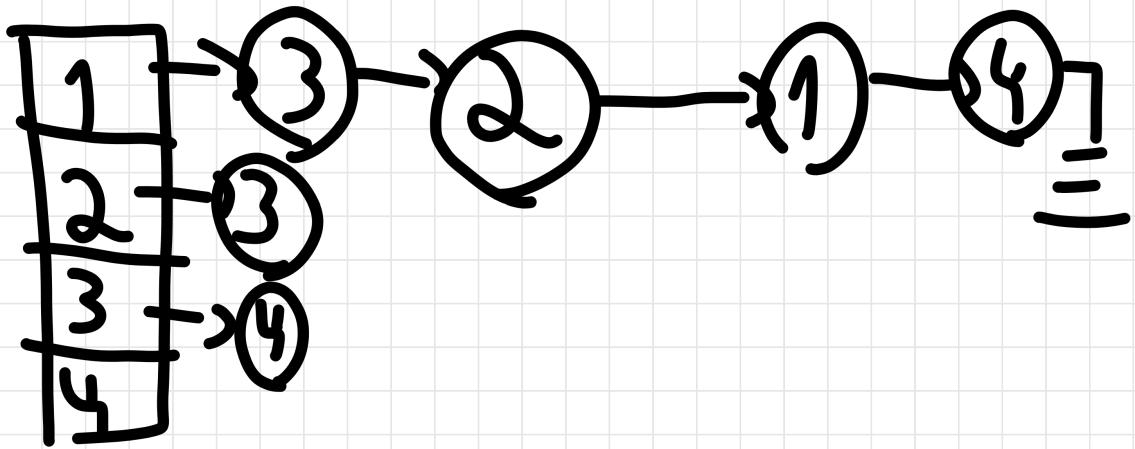
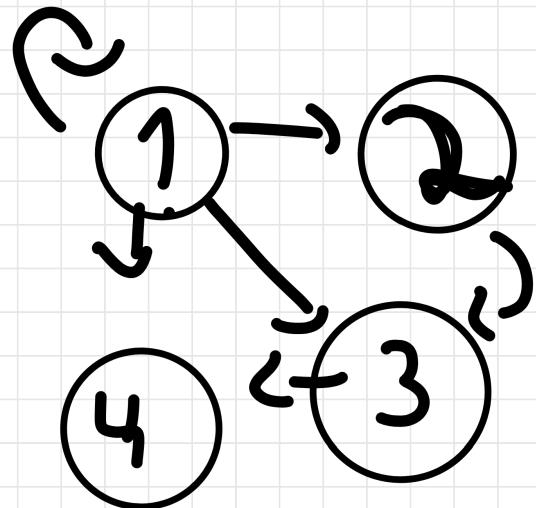




GRAFOS

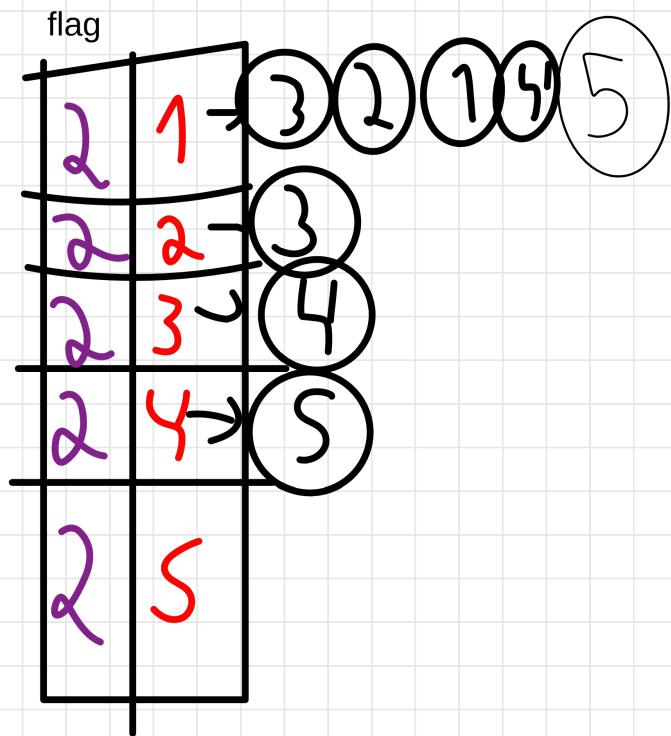
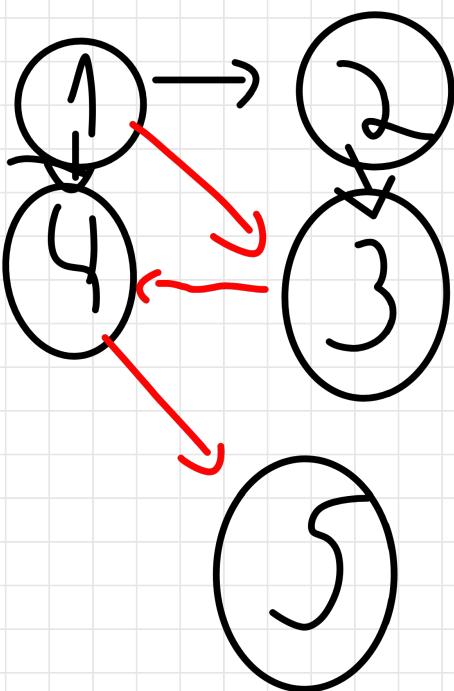


Vetor de Listas Ligadas



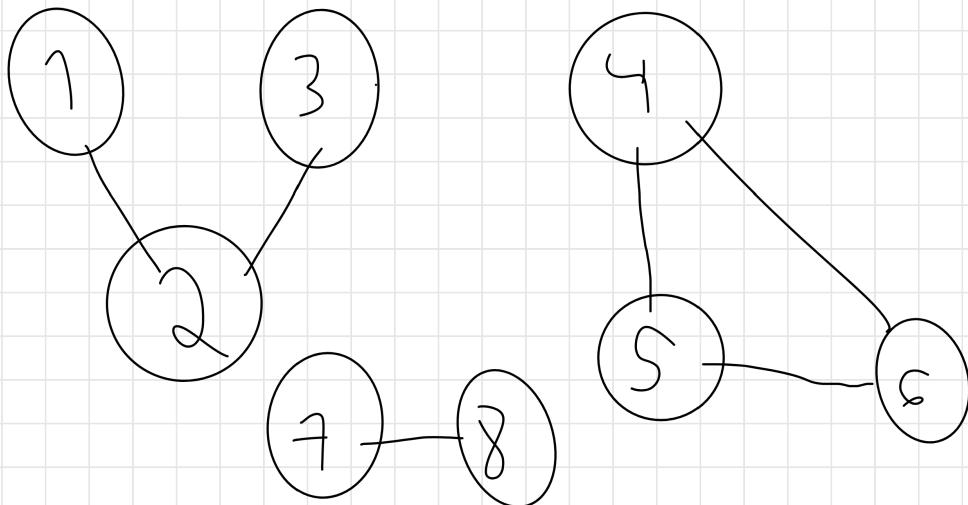
Busca em Grafo

OBJ: a partir de um vertice inicial, percorre todos os vertices alcançaveis. Int flag é um auxiliar para controle da busca, para evitar repetição



Passos de resolver problemas com grafos:

1. Modelar o grafo:



2. É de busca?

1

para ser um problema de busca, eu tenho que estar preocupado se há conexões a partir de certos elementos

3. Se for de busca, escolher entre profundidade x largura.

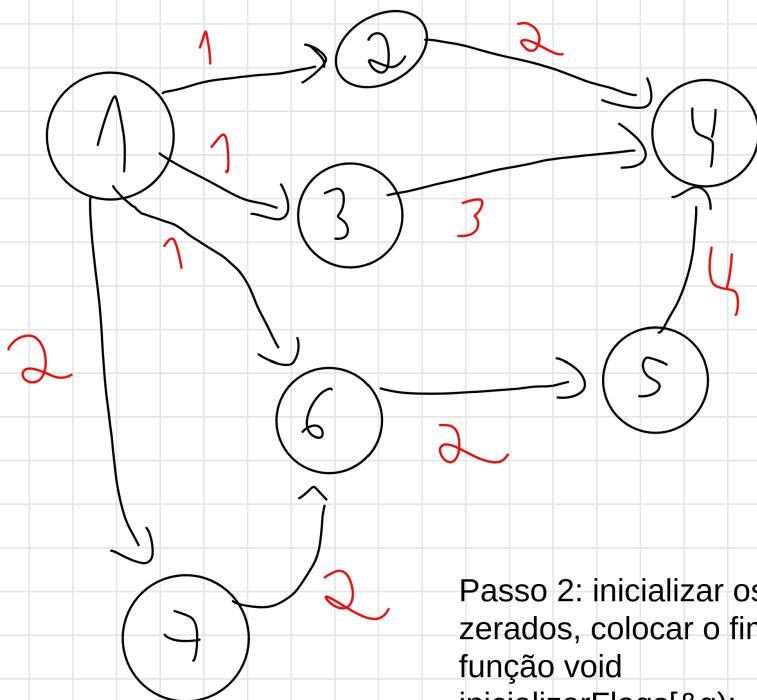
no caso de profundidade, o que eu faço com os outros desconexos? simples, inicie a busca pelo próximo NO que possui flag = 0 EX 2

NO	flags
1	2
2	2
3	2
4	0
5	0
6	0
7	0
8	0

```
ZeroFlags(Vertex* g){  
    int i = 0;  
    int j;  
    for(j = 0; j <= V; j++){  
        if(g[j].flag == 0){  
            i = j;  
            break;  
        }  
    }  
    if(i == 0) return NULL;  
    prof(g, i);  
    printf("\n");  
}
```

EX 3

Minha área:



Passo 1: mude o struct de NO:

```
typedef struct s{
    struct s* prox;
    int adj;
    int cia;
}NO;
```

Passo 2: inicializar os flags para todos serem zerados, colocar o fim como falso, e fazer uma função void

```
inicializarFlags[&g];
bool fim = false;
```

```
void prof(Vertice* g, int i, int j, int c, bool* fim){
    g[i].flag = 1;
    if(*fim) return;
    NO* p = g[i].inicio;
    while(p){
        if(p->cia == c && g[p->adj].flag == 0){
            prof(g, p->adj, j, c, fim);
            if(*fim) return;
        }
        p = p->prox;
    }
    g[i].flag = 2;
    if(p->adj == j) *fim = true;
}
```

Mas as vezes, no caminho não queremos transitar por um lugar específico, para evitar isso basta colocar um campo "percorrível" no struct vertice

EX 4: encontrar caminho i => j sem passar por zonas invalidas

Passo 1: mudar o Struct vertice

```
typedef struct Vertice{
    NO* inicio;
    int flag;
    bool valido;
};
```

Passo 3, função:

```
void prof(Vertice* g, int i, int j, bool* fim){
    g[i].flag = 1;
    if(*fim) return;
    NO* p = g[i].inicio;
    while(p){
        if(g[p->adj].valido == true && g[p->adj].flag == 0{
            prof(g, p->prox, j, fim);
            if(p->adj == j) *fim = true;
            if(*fim) return;
            prof(g, p->prox, j, fim);
            if(*fim) return;
        }
        p = p->prox
    }
    if(p->adj == j) *fim = true;
}
```

Passo 2: re fazer a função prof sabendo que:

- I) não há mais parametro c (pois não é limitado a apenas uma companhia)
- II) testar se g[p->adj].valido é true (ou seja, só mais uma checagem)

Ex 5 : Quantas horas de Voo entre i e j?

PS: Não é o custo mínimo, é só o tempo de qualquer caminho que for encontrado

Passo 1: altere o struct do NO(pois a propriedade de custo é do vôo, e não do local) e coloque o campo duração no struct do Vertice (para guardar a informação de quanto tempo demora pra chegar do ponto de origem até aquele local)

```
typedef struct s {
    struct s* prox;
    int adj;
    int horas;
}

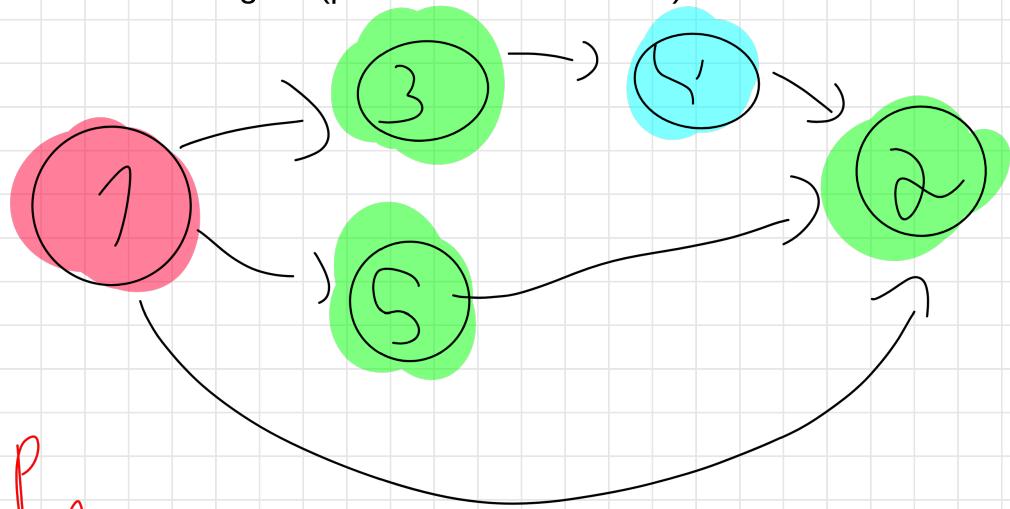
typedef struct Vertice{
    NO* inicio;
    int flag;
    int duracao
}
```

Passo 2: re-faça a função de busca por profundidade:

```
int x;
for(x = 0; x <= V; x++) g[x].duracao = 0;
inicializarFlags(&g);
bool fim = false;
```

```
void prof(Vertice* g, int i, int j, bool* fim){
    g[i].flag = 1;
    if(*fim) return;
    NO* p = g[i].inicio;
    while(p){
        if(g[p->adj].flag == 0){
            g[p->adj].duracao = g[i].duracao + p->horas;
            if(p->adj == j) *fim = true;
            if(*fim) return;
            prof(g, p->prox, j, fim);
            if(*fim) return;
        }
        p = p->prox
    }
}
```

Busca em Largura (p/ caminho mais curto)

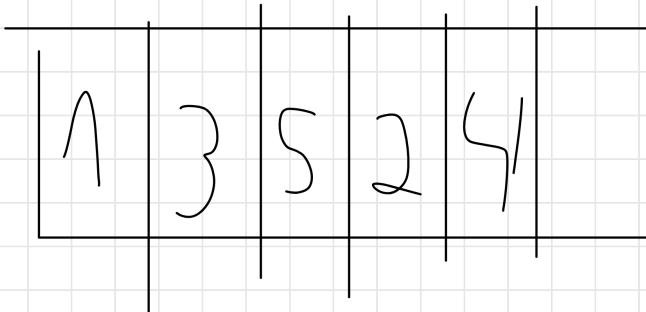


Busca em nível

P₁

P₂

Fila f



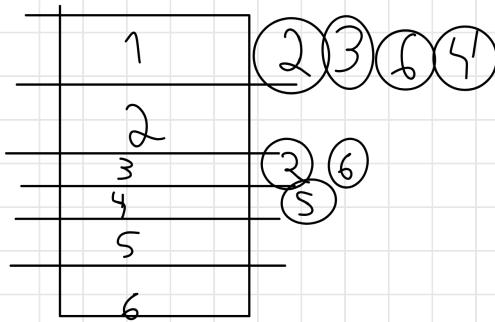
Passos:

1. Entrar em i e marcar flag = 1
2. enquanto houver fila
3. i = sair;
4. marcar flag como 2
5. enfileira o entrar na fila todos os adjascentes não descobertos
6. repete

```

void largura(Vertice* g, int i){
    zerarFlags(g);
    Fila* f;
    inicializarFila(&f);
    EntrarFila(&f, i);
    g[i].flag = 1; //discovered
    while(f){
        i = SairFila(&f);
        g[i].flag = 2;
        NO* p = g[i].inicio;
        while(p){
            if(g[p->adj].flag == 0){
                g[p->adj].flag = 1;
                EntrarFila(&f, p->adj);
            }
            p = p->prox;
        }
    }
}

```



Ordem de conclusao (flag == 2)

Profundidade: 2, 6, 3, 5, 4, 1

Largura: 1, 2, 3, 6, 4, 5

Ordem de descoberta:

Profundidade: 1, 2, 3, 6, 4, 5

Largura: 1, 2, 3, 6, 4, 5

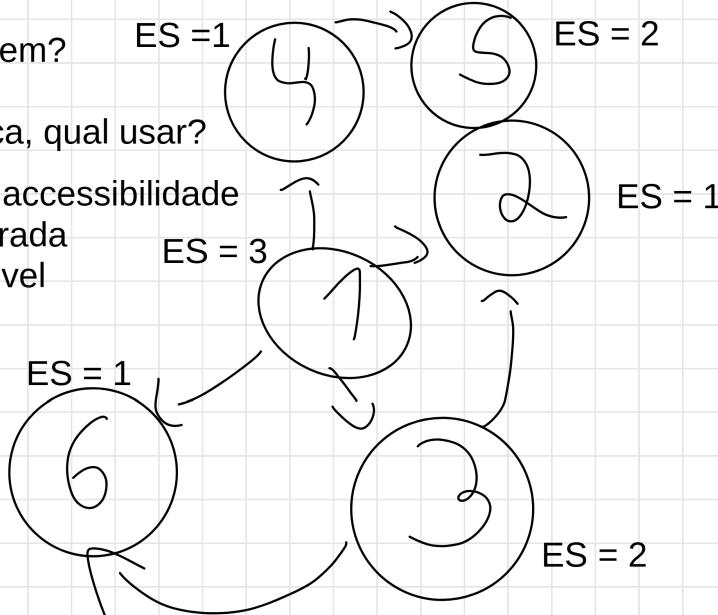
(I) modelagem?

(II) busca?

(III) se busca, qual usar?

ES = 1 ES = 2

ES = 1
ES = 2 : Moderada
ES 3: Acessivel



```
int largura(Vertice* g, int i, int x){  
    zerarFlags(g);
```

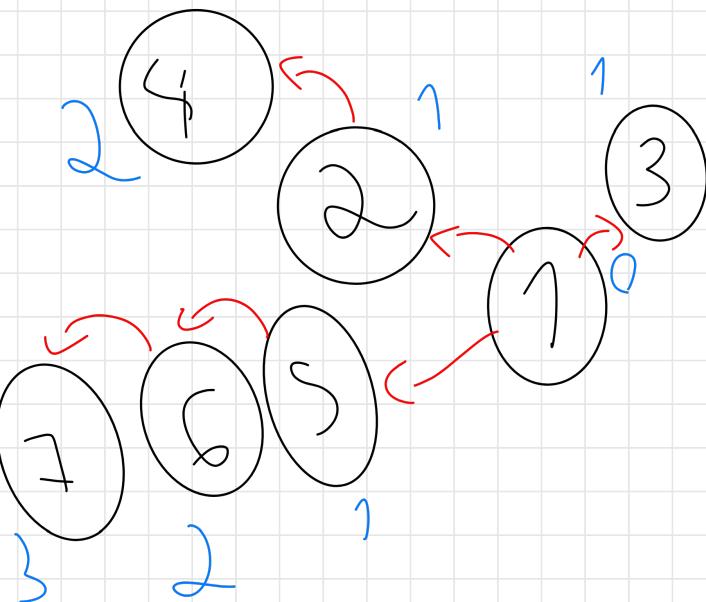
```
    Fila* f;  
    inicializarFila(&f);  
    EntrarFila(&f, i);  
    g[i].flag = 1; //discovered  
    while(f){  
        i = SairFila(&f);
```

Encontrar o primeiro elemento com o ES solicitado

```
        g[i].flag = 2;  
        if(g[i].es == x){  
            while(f) SairFila(&f);  
            return i;  
        }  
        NO* p = g[i].inicio;  
        while(p){  
            if(g[p->adj].flag == 0){  
                g[p->adj].flag = 1;  
                EntrarFila(&f, p->adj);  
            }  
            p = p->prox;
```

Busca em Largura final

- (i) modular (tipo no vertice)
- (ii) busca? sim
- (iii) largura x prof?

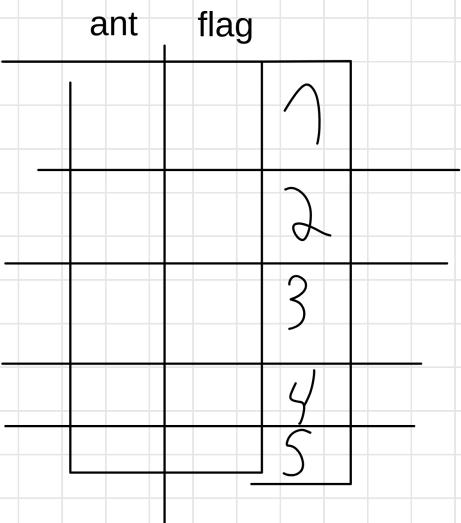
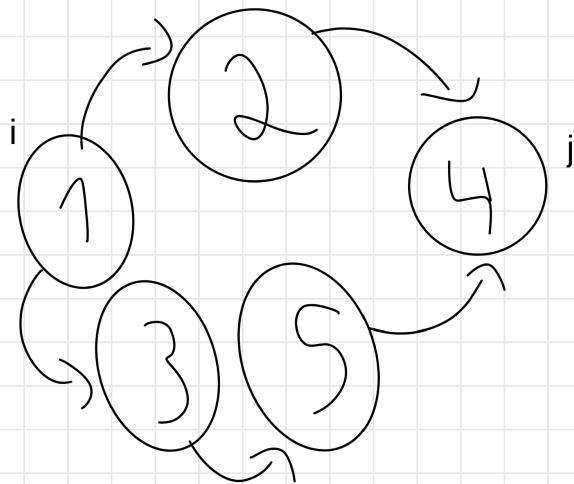


Problema, achar todos os vertices que estao a uma distancia x do vertice 1

nivel flag tipo

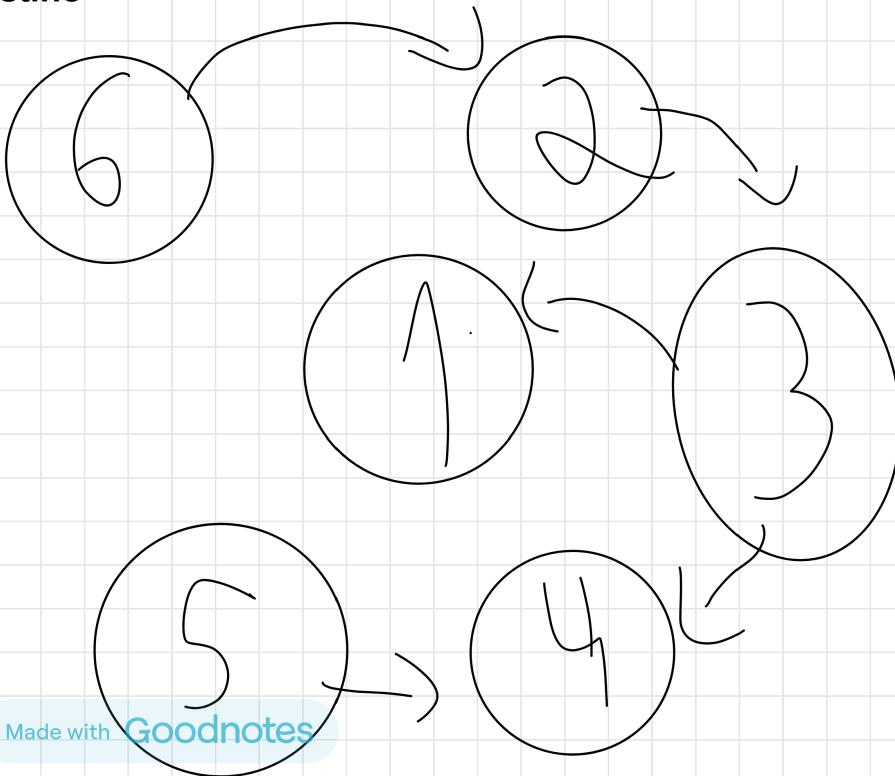
	0	0	1
	0	0	2
	0	1	3
	0	1	4
	0	0	5
	0	0	6
	0	0	7

Comprimento: 2



Comprimento: 3

Ex 1: Exibir Vértices de onde é possível alcançar o destino



Modo: Fazer uma busca em cada um dos vértices até o destino, seja a busca de profundidade ou largura (os dois servem). PS: Algoritmo pouco eficiente

```
void exibirOrigens(int m[V][V], int dest, int flag[V]){
    int i;
    for(i = 0; i <= V; i++){
        if(i == dest || flag[i]>0) continue;
        BuscaLargura(m, i, dest, flag);
    }
    for(i = 0; i <= V; i++){
        if(flag[i] > 0) printf("%d ", i);
    }
}
```

**EX 2: Dado um possível caminho int C = {6, 2, 3, 1, 1, 1, 2}
Verifica se é um caminho válido, retornando true/false**

```
bool caminhoExiste(Vertice* g, int tamanho ,int c[tamanho]){
    if(tamanho < 2) return false;
    NO** ant = NULL;
    for(int j; j < tamanho-1; j++){
        ant = &g[c[j]].inicio;
        if(!arestaExiste(g, c[j], c[j+1], ant))return false;
    }
    return true;
}
```

Ex 3: Qual o vértice mais distante de i segundo o caminho mais curto possível

Passo 1: Busca em largura registrando distâncias

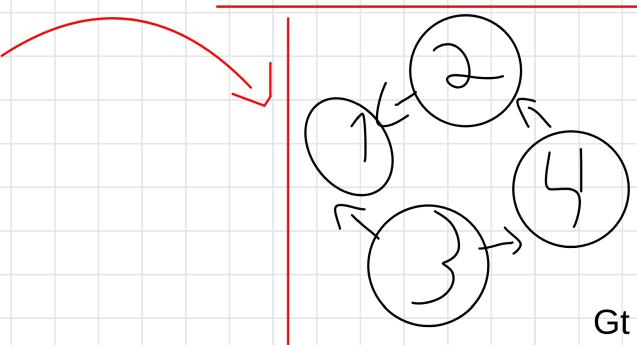
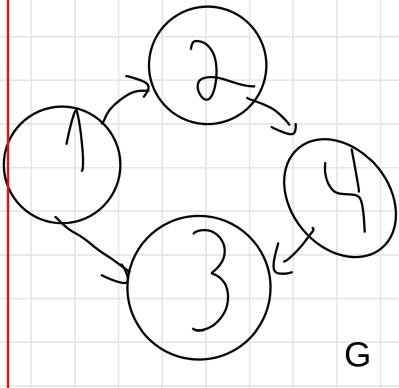
Passo 2: Depois da busca, verificar qual a maior distância

Elemento i tem dist = 0 e os demais é -1 à medida que avança, a distância[j] = distância[i] + 1

```
int maiorCaminho(Vertice*g, int i){  
    int resposta[V];  
    int j;  
    for(j = 0; j <= V; j++){  
        if(j == i) continue;  
        resposta[j] = menorCaminhoLargura(g, j, i);  
    }  
    int response = 0;  
    for(j = 0; j < V; j++){  
        if(response > resposta[j]) response = resposta[j];  
    }  
    return response;  
}
```

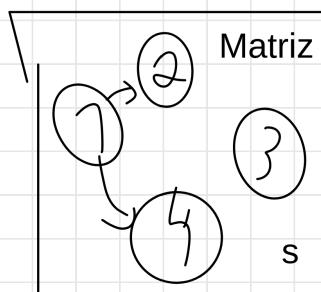
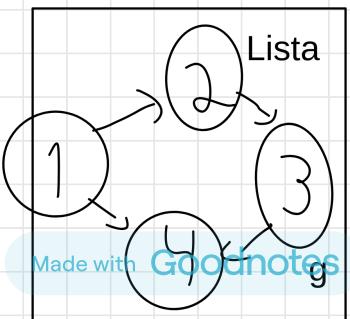
```
int menorCaminhoLargura(Vertice*g, int j, int i){  
    zerarFlags(g);  
    Fila* f = inicializarFila();  
    EntrarFila(f, i);  
    g[i].flag = 1; //discovered  
    g[i].custo = 0;  
    while(f){  
        i = SairFila(f);  
        g[i].flag = 2;  
        NO* p = g[i].inicio;  
        while(p){  
            if(g[p->adj].flag == 0){  
                g[p->adj].flag = 1;  
                g[p->adj].custo = p->custo + 1;  
                if(p->adj == i) return g[p->adj].custo;  
                EntrarFila(f, p->adj);  
            }  
            p = p->prox;  
        }  
    }  
}
```

Ex 4: Encontrar grafo transposto em listas de adjacência



```
Vertice* copiaTransposta(Vertice* g, int V){  
    Vertice* copia = inicializarVertice(V);  
    NO* atual = NULL;  
    for(int i = 0; i <= V; i++){  
        atual = g[i].inicio;  
        while(atual){  
            NO* novo = (NO*)malloc(sizeof(NO));  
            novo->adj = i;  
            novo->prox = copia[atual->adj].inicio;  
            copia[atual->adj].inicio = novo;  
            atual = atual->prox;  
        }  
    }  
    return copia;  
}
```

Ex 5: Verificar se todas as arestas de s ocorrem em g



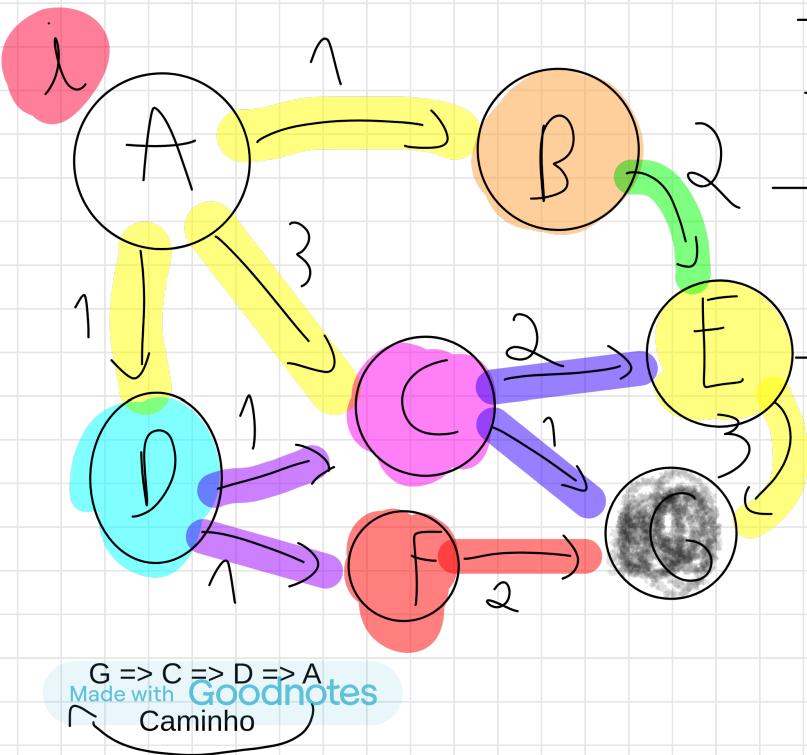
```

bool subgrafo(int m[V][V], Vertice* g){
    for(int i = 0; i <= V; i++){
        for(int j = 0; j <= V; j++){
            if(m[i][j] == 1){
                if(!arestaExistente(g, i, j)) return false;
            }
        }
    }
    return true;
}

```

Exercicio proposto: Grafos iguais: compara um grafo em forma matricial e outro em forma de lista e retorna falso caso sejam diferentes e verdadeiro caso sejam iguais

Caminho de custo minimo (Algoritmo de Dijkstra)



Vértice	P	Dist	Via
A	1	0	0
B	1	1	A
C	1	1+1	D
D	1	1	A
E	1	1+2	B
F	1	1+1	D
G	1	2+1	C

Passos:

1. Dado z, Concluído ($P = 1$)
2. Atualiza adjacentes não concluidos de z
3. Escolhe um novo z não concluído de menor dist de todo o grafo.
Repete 1-z até acabar z's

OBS: Nao pode possuir arestas com pesos negativos

```
void dijkstra(Vertice* g, int i){  
    int j;  
    for(j = 0; j <= V; j++){  
        g[j].flag = 0;  
        g[j].custo = INT_MAX;  
        g[j].via = -1;  
    }  
    g[i].custo = 0;  
    g[i].via = 0;  
    int z = i;  
    while(z > -1){  
        g[z].flag = 1;//Passo 1  
        //inicio passo 2  
        NO* p = g[z].inicio;  
        while(p){  
            if(g[p->adj].flag == 0){  
                int aux = g[p->adj].custo;  
                if(aux < g[p->adj].custo){  
                    g[p->adj].custo = aux;  
                    g[p->adj].via = z;  
                }  
            }  
            p = p->prox;  
        }  
        //fim passo 2, inicio passo 3  
        z = -1;  
        int menorCusto = INT_MAX;  
        for(j = 1; j <= V; j++){  
            if(g[j].flag == 0){  
                if(g[j].custo < menorCusto){  
                    menorCusto = g[j].custo;  
                    z = j;  
                }  
            }  
        }  
    }    //Fim do passo 3 }//Custo muito alto para esse algoritmo, o  
utilizado na realidade é feito de outra maneira utilizando uma fila de prioridade  
nessa busca
```

Ex 1: Contar vértices alcançaveis a partir de i

```
int largura(int m[V][V], int i, int flag[V]){
    int count = 0;
    Fila* F;
    inicializarFila(F);
    EntrarFila(F, i);
    flag[i] = 1;
    while(F){
        i = SairFila(F);
        flag[i] = 2;
        count++;
        int j;
        for(j = 0; j <= V; j++){
            if(m[i][j] == 1 && flag[j] == 0){
                flag[j] = 1;
                EntrarFila(F, j);
            }
        }
    }
    return count;
}
```

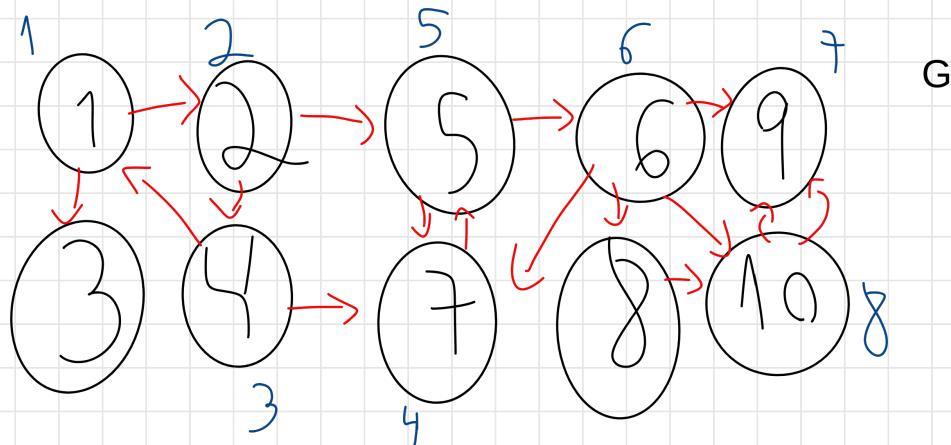
Exemplo de uma questão que valeria 2 pontos*

Componentes fortemente conectados (ou conexos)

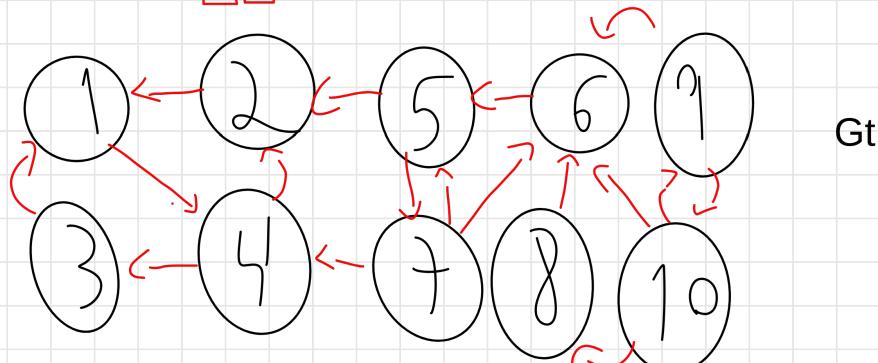
Passo 1: Criar grafo transposto G^t

Passo 2: Computar ordem de conclusão em profundidade em g

Passo 3: Começando pelo último vértice descoberto, fazer busca em G^t . Tudo que for alcançável é parte do mesmo grupo, desde que ainda não incluído



Prof: 10, 9, 8, 6, 5, 7, 4, 2, 3, 1



Grupo 1: {1, 2, 3, 4} ínicio em 1

Grupo 2: {5, 6, 7} ínicio em 7

Grupo 3: {8} ínicio em 8

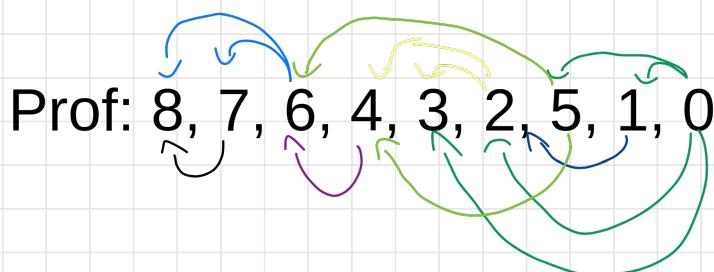
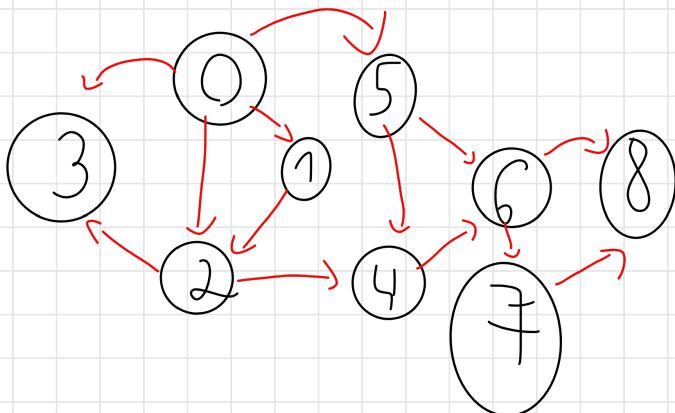
Grupo 4: {9, 10} ínicio em 9

Ordenação Topológica

*Grafos dirigidos

*Acíclicos

*Aresta representa um pré-requisito



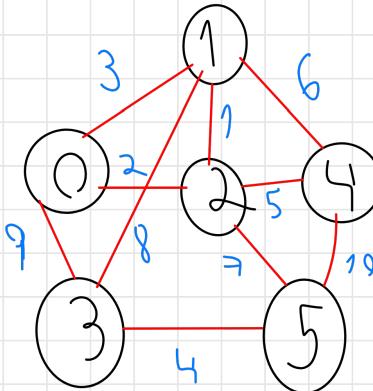
Da direita para a esquerda dá a ordem que cada um deve ser feito

Árvore geradora mínima

*Algoritmo de Kruskal

*Algoritmo de Prim

(Para Grafo ponderado e não dirigido)



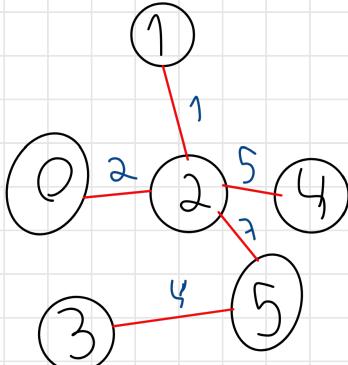
Qual escolher?

Dependendo do grafo, cada algoritmo vai ser melhor do que o outro.

Como caso tenham muitas arestas, seria melhor utilizar o Prim

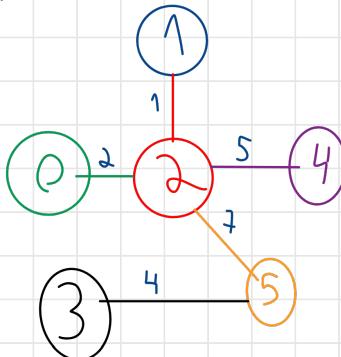
Kruskal:

1. Criar floresta de árvores triviais (Pegue sómente os vértice)
2. Selecionar aresta de menor custo que não provoque ciclo
3. Repetir até conectar todos os vértices



Prim:

1. Escolhe vértice inicial
2. Expande área segura inserindo aresta de menor peso
3. Repete



Ordem da zona Segura:
1, 2, 0, 4, 5, 3

Custo = 19

