

AED2 - Aula 22

Busca em profundidade, conectividade

Busca em grafos

Busca em grafos é, possivelmente, a operação

- mais básica
- importante
- e genérica

para se fazer em um grafo.

Isso porque ela é fundamental para

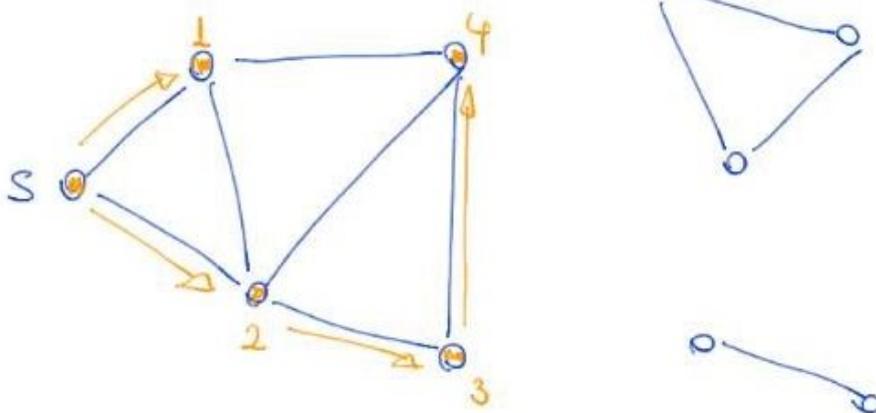
- obter informações sobre conectividade,
 - i.e., determinar como e com quem se conecta cada vértice.
- Além disso, ela pode ser especializada em vários tipos de busca.

A busca em grafos genérica encontra todos os vértices

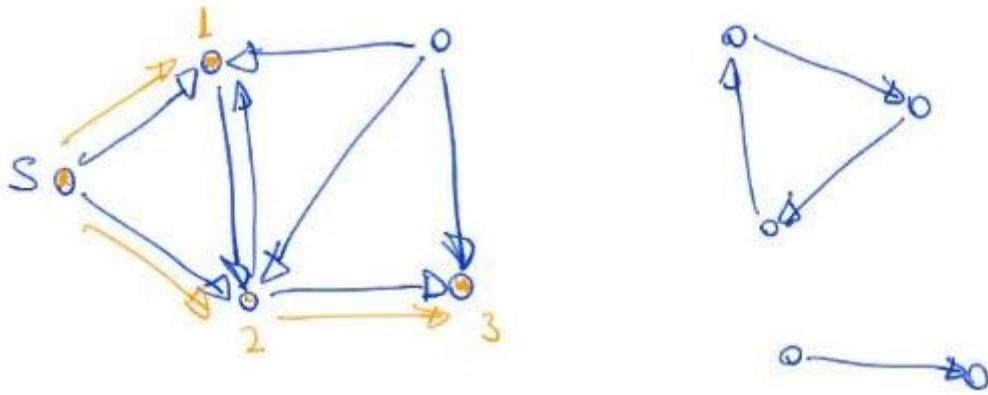
- que podem ser alcançados a partir de um vértice inicial.
- A ordem em que se visita os vértices é arbitrária,
 - e a única restrição é ir sempre
 - de um vértice encontrado para um não encontrado.

Exemplos de busca a partir de s

- em um grafo não orientado



- e em um grafo orientado



- Note que, em um grafo orientado,
 - a busca respeita a orientação dos arcos.

Pseudocódigo:

```

buscaGenerica(grafo G=(V,E), vértice s) {
  para v ∈ V
    marque v como não encontrado
  marque s como encontrado
  enquanto existir uma aresta (u, v) com u encontrado e v não encontrado
    marque v como encontrado
}

```

Corretude:

- Ao fim do algoritmo, um vértice v foi encontrado
 - se, e somente se, existe um caminho em G de s até v.

Demonstração:

(-->) Vamos provar a ida por indução.

O caso base vale porque no início o único vértice encontrado é o próprio s

- e é conhecido um caminho trivial de s para s.

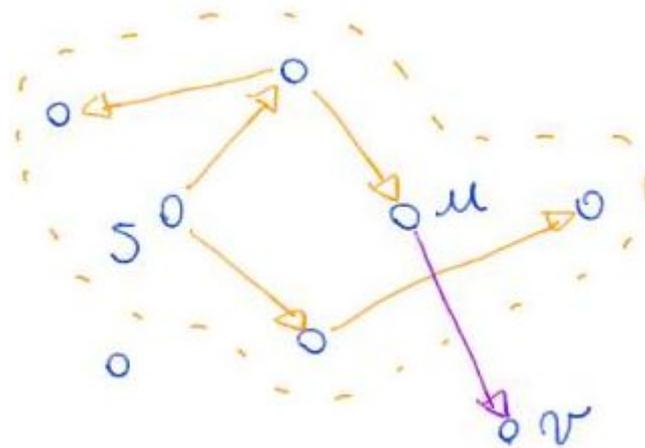
Nossa hipótese de indução é que a afirmação é verdadeira

- até a iteração anterior ao algoritmo encontrar o vértice v.
- Mais precisamente, supomos que é conhecido um caminho de s
 - até qualquer vértice encontrado antes do início da iteração
 - em que o algoritmo encontra v,
 - e que estes caminhos só usam vértices encontrados.

Desenvolvemos o passo assim:

- Na iteração em que o algoritmo encontra v,
 - ele o faz através de uma aresta (u, v),

- sendo que u já estava encontrado.



- Pela hipótese de indução, sabemos que existe um caminho de s até u .
- Concatenando o caminho de s até u com a aresta (u, v) ,
 - temos um caminho de s até v .
- Note que o caminho de s até u não passa por v
 - porque ele só usa vértices encontrados previamente.

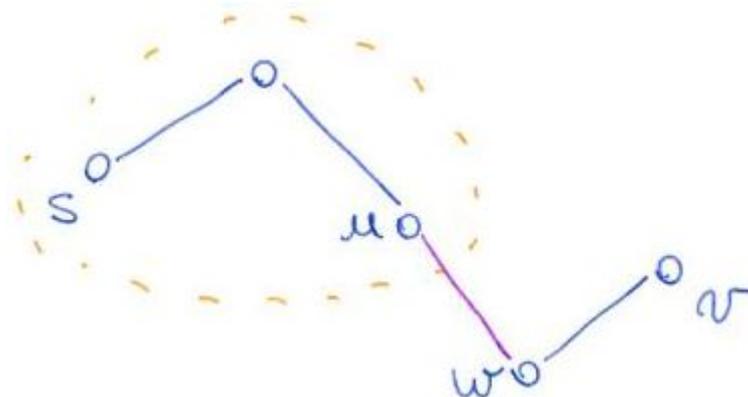
(<--> Para provar a volta supomos, por contradição,

- que existe um caminho de s até v , mas v não foi encontrado.

Lembramos que neste tipo de prova queremos chegar a um absurdo.

Começamos percorrendo o caminho de s até v ,

- mas paramos ao encontrar a primeira aresta
 - que leva de um vértice encontrado para um vértice não encontrado.



- Note que, tal aresta deve existir, pois
 - o caminho começa com um vértice encontrado (s)
 - e termina com um não encontrado (v).
- Digamos que esta aresta é (u, w) ,
 - sendo que u pode ser o próprio s e w pode ser o próprio v .

De posse da aresta (u, w) consideramos o comportamento do algoritmo

- e notamos que ele não pára enquanto existir uma aresta do tipo de (u, w) ,
 - ou seja, que tem origem encontrada e destino não encontrado.
- Portanto, temos um absurdo e concluímos a demonstração.

Q.E.D.

Eficiência:

- Em cada iteração do algoritmo, vamos sempre
 - de um vértice encontrado para um não encontrado,
- nunca visitando um vértice
 - ou percorrendo um arco mais que uma vez.
- Isso sugere que a eficiência do algoritmo é proporcional
 - ao número de vértices mais o número de arestas do grafo.
- Note que, se o grafo não for orientado
 - consideramos cada aresta até duas vezes.
- Vamos analisar mais detalhadamente a eficiência da busca
 - em suas subseqüentes implementações.

Busca em profundidade

Agora vamos estudar uma especialização da busca genérica,

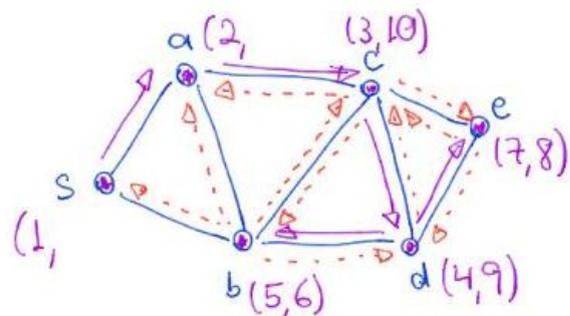
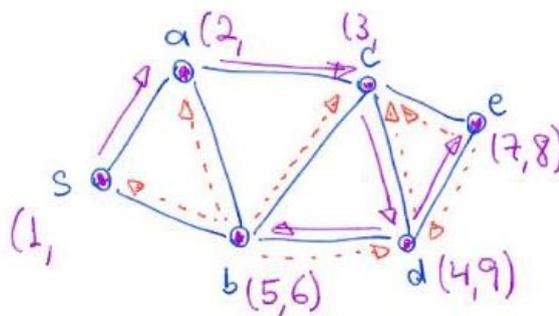
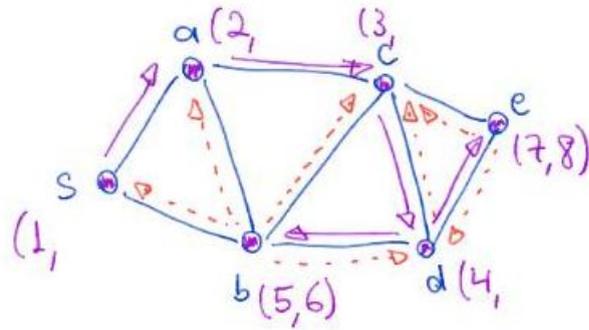
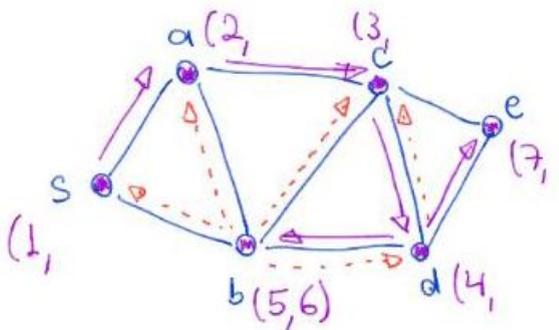
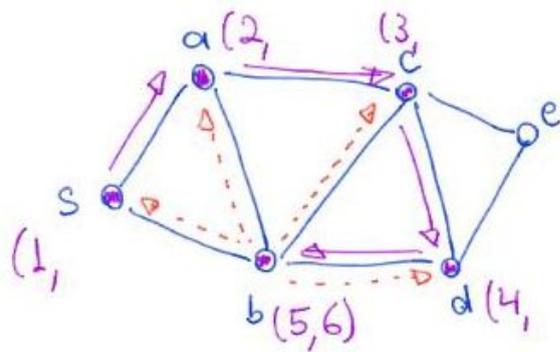
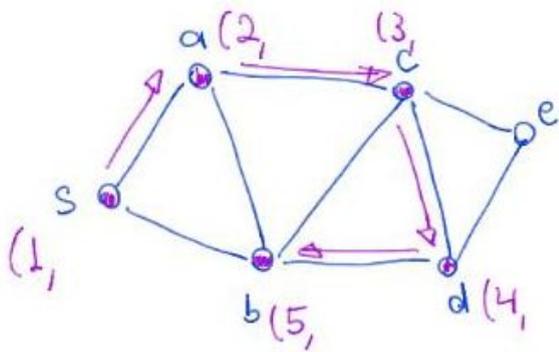
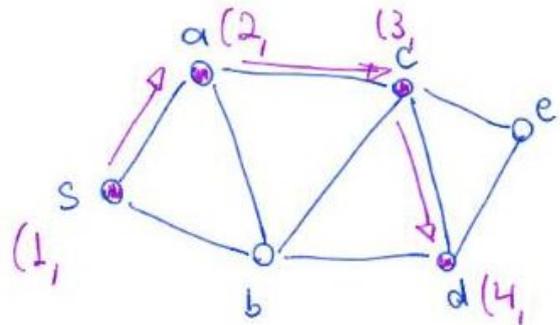
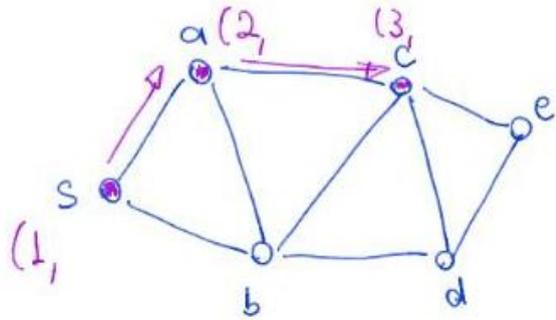
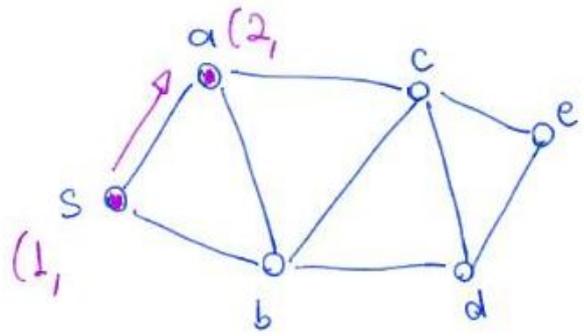
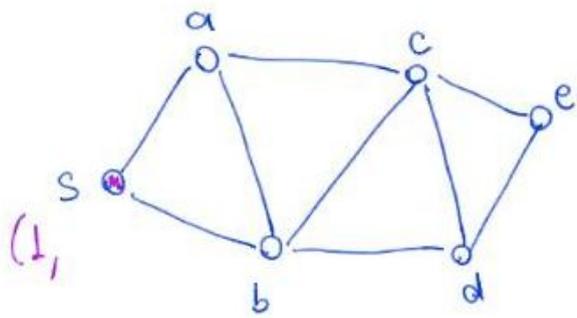
- chamada de busca em profundidade,
 - também conhecida por DFS (Depth-First Search).
- Esta busca explora um caminho do grafo
 - até que não haja mais para onde estendê-lo.
- Então volta pelo caminho percorrido,
 - procurando outras rotas ainda não visitadas.

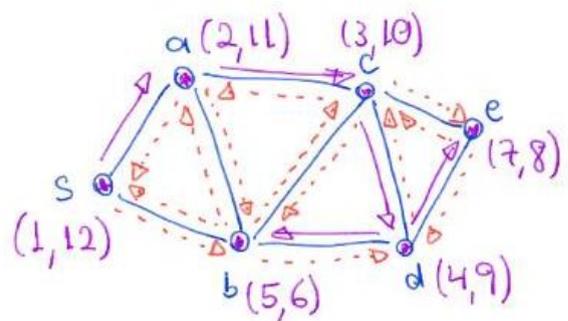
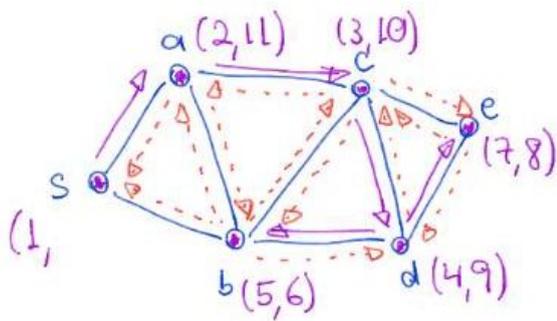
O comportamento da DFS está intimamente relacionado

- com a estrutura de dados pilha (stack ou LIFO),
- e ela também pode ser implementada utilizando recursão.

Exemplo de busca em profundidade:

- Em roxo está destacado o percurso realizado pela busca em profundidade
 - bem como a ordem de chegada e saída dos vértices.
- Usamos o termo tempo de início (ou de abertura) de um vértice,
 - como sinônimo de ordem de chegada,
- e tempo de término (ou de fechamento) de um vértice,
 - como sinônimo de ordem de saída.
- Em vermelho são arestas que, quando verificadas,
 - já tinham seu vértice destino encontrado.





Observem que, os tempos de início e término dos vértices nos contam

- quais vértices foram alcançados a partir de um determinado vértice.

Mais especificamente, se um vértice u tem tempo de início i e de término t ,

- todo vértice com tempo de início maior que i e menor que t
 - foi alcançado a partir de u .

Antes de começar a busca em profundidade, é necessário uma inicialização

- em que todo vértice em V é marcado como não visitado.

```
void buscaProf(Grafo G, int origem, int *ordem_chegada,
               int *ordem_saida) {
    int i, tempo;
    /* inicializa todos como não encontrados */
    for (i = 0; i < G->n; i++)
    {
        ordem_chegada[i] = -1;
        ordem_saida[i] = -1;
    }
    tempo = 1;
    buscaProfR(G, origem, ordem_chegada, ordem_saida, &tempo);
    // buscaProfI(G, origem, ordem_chegada, ordem_saida, tempo);
}
```

- Note que, esta inicialização leva tempo linear no número de vértices,
 - i.e., $O(n)$.

Busca em profundidade recursiva

```
buscaProfRec(grafo G=(V,E), vértice v) {
    marque v como visitado
```

```

    para cada aresta (v, w)
        se w não foi visitado
            buscaProfRec(grafo G=(V,E), vértice w)
}

```

Corretude:

- Encontra todos os vértices alcançáveis, ou seja,
 - para os quais existe caminho a partir de v.
- Segue da corretude da busca genérica, já que é um caso particular daquela.

Eficiência:

- Leva tempo $O(n_v + m_v)$, onde n_v e m_v são, respectivamente,
 - o número de vértices e arestas da componente do vértice v
 - da primeira chamada da recursão.
- Resultado segue porque
 - cada vértice da componente será visitado uma vez,
 - antes de ser marcado como visitado,
 - e cada aresta será considerada no máximo
 - duas vezes (no caso de grafos não orientados)
 - ou apenas uma vez (no caso dos orientados),
 - já que uma aresta só é considerada
 - quando seu vértice está sendo visitado.

A seguir temos implementações recursivas de busca em profundidade

- que registram a ordem de chegada e saída de cada nó.
- Observe que as funções recursivas recebem o valor tempo
 - a partir de um apontador ptempo. Por que?

Código busca em profundidade com grafo implementado por matriz de adjacência.

```

void buscaProfR(Grafo G, int v, int *ordem_chegada,
                int *ordem_saida, int *ptempo) {
    int w;
    ordem_chegada[v] = (*ptempo)++;
    /* para cada vizinho de v que ainda não foi visitado */
    for (w = 0; w < G->n; w++)
        if (G->A[v][w] == 1 && ordem_chegada[w] == -1)
            buscaProfR(G, w, ordem_chegada, ordem_saida, ptempo);
    ordem_saida[v] = (*ptempo)++;
}

```

- Qual a eficiência deste algoritmo?

- $O(n_s * n)$, sendo s o vértice origem. Por que?

Código busca em profundidade com grafo implementado por listas de adjacência.

```
void buscaProfR(Grafo G, int v, int *ordem_chegada,
               int *ordem_saida, int *ptempo) {
    int w;
    Noh *p;
    ordem_chegada[v] = (*ptempo)++;
    /* para cada vizinho de v que ainda não foi visitado */
    p = G->A[v];
    while (p != NULL)
    {
        w = p->rotulo;
        if (ordem_chegada[w] == -1)
            buscaProfR(G, w, ordem_chegada, ordem_saida, ptempo);
        p = p->prox;
    }
    ordem_saida[v] = (*ptempo)++;
}
```

- Qual a eficiência deste algoritmo?
 - $O(n_s + m_s)$, sendo s o vértice origem. Por que?

Busca em profundidade implementada com pilha

```
buscaProfPilha(grafo G=(V,E), vértice s) {
    para v \in V
        marque v como não visitado
    seja P uma pilha inicializada com o vértice s
    enquanto P != \empty
        remova um vértice v do topo de P
        se v não foi visitado
            marque v como visitado
            para cada aresta (v, w)
                se w não foi visitado
                    insira w no topo de P
}
```

Corretude:

- o algoritmo encontra todos os vértices alcançáveis a partir de s .
- Esse resultado segue da corretude do algoritmo de busca genérica,
 - já que a busca em profundidade é um caso particular daquela.

Eficiência:

- O algoritmo leva tempo $O(n)$ para
 - marcar todos os vértices do grafo como não visitados.
- O restante do algoritmo leva tempo $O(n_s + m_s)$,
 - sendo n_s e m_s o número de vértices e arestas
 - da componente que contém o vértice s .
- Isso porque cada aresta do componente do vértice s
 - é visitada no máximo duas vezes (no caso do grafo ser não orientado).
- Para perceber isso, observe que o algoritmo
 - só visita as arestas de um vértice após remover este vértice da pilha
 - e ele já não ter sido visitado.
- No entanto, neste caso o vértice é marcado como visitado
 - antes do algoritmo visitar suas arestas.
- Portanto, uma aresta qualquer será visitada
 - no máximo uma vez a partir de cada vértice extremo
 - (no caso de um grafo não orientado)
 - ou apenas uma vez a partir de sua cauda
 - (no caso de um grafo dirigido).
- Notamos que, embora vértices visitados não sejam adicionados à pilha,
 - existe a possibilidade de um vértice ser colocado
 - mais de uma vez na pilha, antes de ter sido visitado.
- Mas, nesse caso ele será marcado como visitado
 - na primeira vez que for removido da pilha,
 - e nas vezes subsequentes será descartado.
- Destacamos que o número total de inserções (e remoções)
 - que podem ocorrer na pilha ao longo de toda a execução do algoritmo
 - é limitada pela soma dos graus de entrada dos vértices.
- Isso porque, para um vértice ser colocado mais de uma vez,
 - ele tem que ser destino de diversas arestas.
- Com isso concluímos que o algoritmo executa
 - um número de passos (e operações da pilha) limitado superiormente
 - pelo número de vértices mais arestas da componente de s ,
 - ou seja, $O(n_s + m_s)$.

A seguir temos implementações iterativas com pilha de busca em profundidade

- que registram a ordem de chegada e saída de cada nó.
 - O que elas fazem para registrar corretamente a ordem de saída?
- Observe que as funções recursivas recebem o valor tempo diretamente.

- Por que não precisam operar com um apontador ptempo?

Código busca em profundidade com grafo implementado por matriz de adjacência.

```
void buscaProfI(Grafo G, int origem, int *ordem_chegada,
               int *ordem_saida, int tempo) {
    int v, w;
    // pilha implementada em vetor
    int *pilha;
    int topo = 0;
    pilha = malloc(G->m * sizeof(int));
    /* colocando a origem na pilha */
    pilha[topo++] = origem;
    /* enquanto a pilha dos ativos (encontrados
    mas não visitados) não estiver vazia */
    while (topo > 0)
    {
        /* remova o mais recente da pilha */
        v = pilha[--topo];
        if (ordem_chegada[v] == -1) // se v nao foi visitado
        {
            ordem_chegada[v] = tempo++;
            pilha[topo++] = v; // empilha o vértice v pra saber
            quando marcar o tempo de término
            /* para cada vizinho de v que ainda não foi visitado */
            for (w = 0; w < G->n; w++)
                if (G->A[v][w] == 1 && ordem_chegada[w] == -1)
                    pilha[topo++] = w; // empilha o vizinho
            }
            else if (ordem_saida[v] == -1)
                ordem_saida[v] = tempo++;
        }
        free(pilha);
    }
}
```

- Qual a eficiência deste algoritmo?
 - $O(n_s * n)$, sendo s o vértice origem. Por que?

Código busca em profundidade com grafo implementado por listas de adjacência.

```
void buscaProfI(Grafo G, int origem, int *ordem_chegada,
                int *ordem_saida, int tempo) {
    int v, w;
    Noh *p;
    // pilha implementada em vetor
    int *pilha;
    int topo = 0;
    pilha = malloc(G->m * sizeof(int));
    /* colocando a origem na pilha */
    pilha[topo++] = origem;
    /* enquanto a pilha dos ativos (encontrados
    mas não visitados) não estiver vazia */
    while (topo > 0)
    {
        /* remova o mais recente da pilha */
        v = pilha[--topo];
        if (ordem_chegada[v] == -1) // se v nao foi visitado
        {
            ordem_chegada[v] = tempo++;
            pilha[topo++] = v; // empilha o vértice v pra saber
            quando marcar o tempo de término
            /* para cada vizinho de v que ainda não foi visitado */
            p = G->A[v];
            while (p != NULL)
            {
                w = p->rotulo;
                if (ordem_chegada[w] == -1)
                    pilha[topo++] = w; // empilha o vizinho
                p = p->prox;
            }
        }
        else if (ordem_saida[v] == -1)
            ordem_saida[v] = tempo++;
    }
}
```

```
}
```

- Qual a eficiência deste algoritmo?
 - $O(n_s + m_s)$, sendo s o vértice origem. Por que?

Componentes conexos de um grafo não-orientado

Um componente conexo de um grafo não-orientado

- é um conjunto de vértices tal que
 - existe caminho entre qualquer par de vértices do conjunto.

Para encontrar os componentes conexos de um grafo não-orientado,

- usamos uma busca em grafos, como a DFS,
 - que é invocada a partir de cada vértice do grafo.
- Cada invocação está associada com um rótulo/valor distinto,
 - que será atribuído a todos os vértices encontrados naquela busca.
- No final temos uma partição dos vértices do grafo.

```
componentes(grafo G=(V,E)) {  
    num_comps = 0  
    para v \in V  
        marque v como não encontrado  
    para v = 1 até n  
        se v não encontrado  
            num_comps++  
            busca(G, v, num_comps)  
}
```

- sendo $\text{busca}(G, v, \text{num_comps})$ uma variante da busca genérica
 - que atribui rótulo num_comps para todo vértice w encontrado,
 - i.e., que faz $\text{comp}[w] = \text{num_comps}$.
- Observe que num_comps só é incrementado quando uma busca termina
 - e a execução volta para o laço principal de $\text{componentes}(G)$.

Eficiência:

- $O(n + m)$, pois cada chamada da busca tem custo proporcional
 - ao número de vértices e arestas do componente conexo visitado.

Código para identificar componentes

- com grafo implementado por listas de adjacência
 - e usando busca em profundidade recursiva.

```
void identComponetes(Grafo G, int *comp) {
```

```

int v, num_comps;
/* inicializa todos os vértices como não pertencentes a qualquer
componente */
for (v = 0; v < G->n; v++)
    comp[v] = -1;
num_comps = 0;
for (v = 0; v < G->n; v++)
    if (comp[v] == -1)
    {
        num_comps++;
        buscaCompR(G, v, comp, num_comps);
    }
}

void buscaCompR(Grafo G, int v, int *comp, int comp_atual) {
    int w;
    Noh *p;
    comp[v] = comp_atual;
    /* para cada vizinho de v que ainda não foi visitado */
    p = G->A[v];
    while (p != NULL)
    {
        w = p->rotulo;
        if (comp[w] == -1)
            buscaCompR(G, w, comp, comp_atual);
        p = p->prox;
    }
}

```