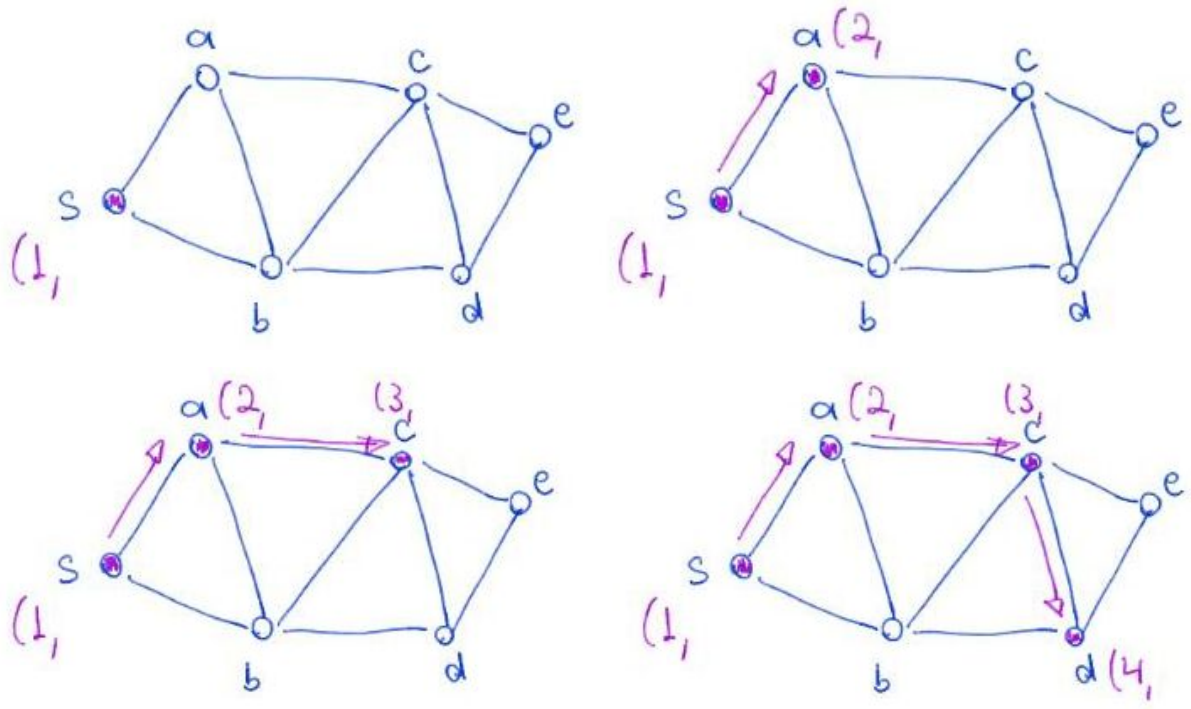
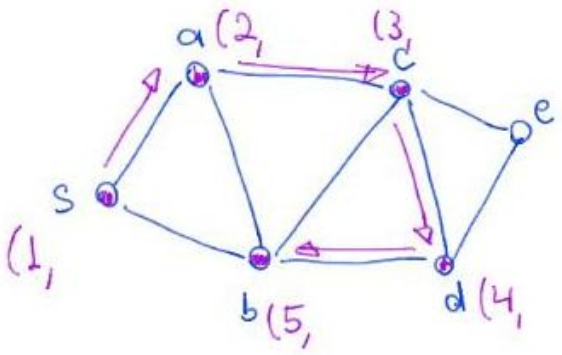
AED2 - Aula 23
Busca em profundidade, conectividade

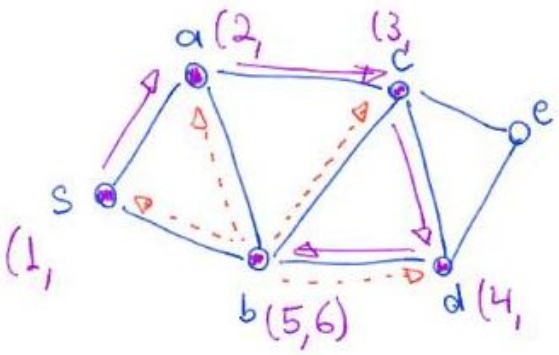
Hoje vamos estudar a busca em profundidade
- também conhecida por DFS (Depth-First Search).
- Esta busca explora um caminho do grafo
    - até que não haja mais para onde estendê-lo.
- Então volta pelo caminho percorrido,
    - procurando outras rotas ainda não visitadas.

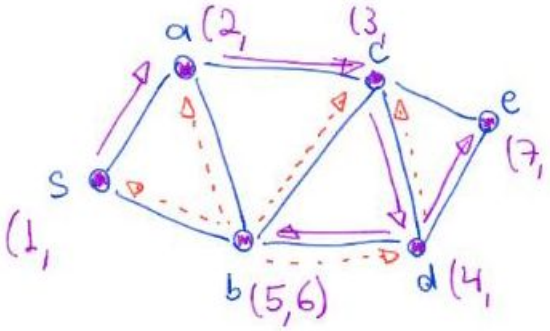
O comportamento da DFS está intimamente relacionado
- com a estrutura de dados pilha (stack ou LIFO),
- e ela também pode ser implementada utilizando recursão.

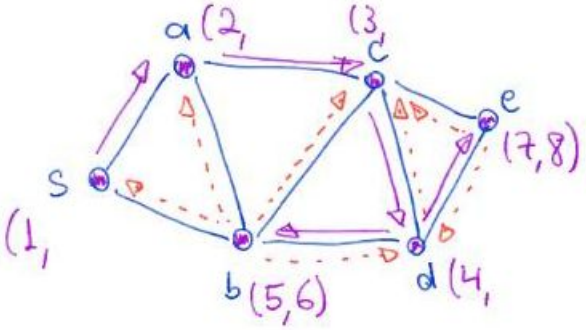
Exemplo de busca em profundidade:
- Em roxo o caminho e ordem de chegada e saída dos vértices.
- Em vermelho arestas que, quando verificadas, já tinham destino encontrado.

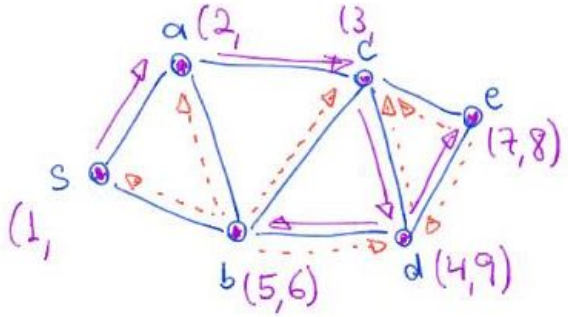


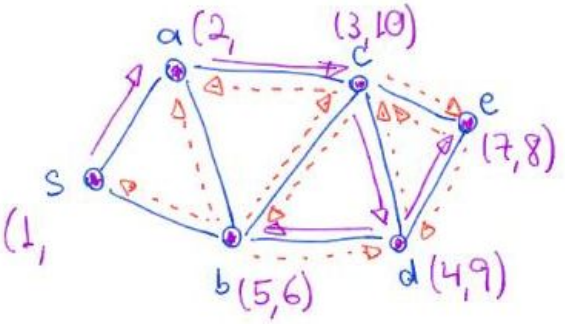


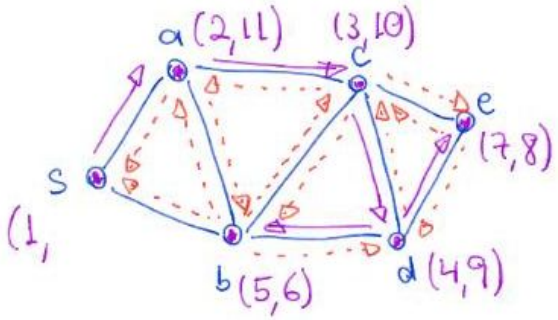


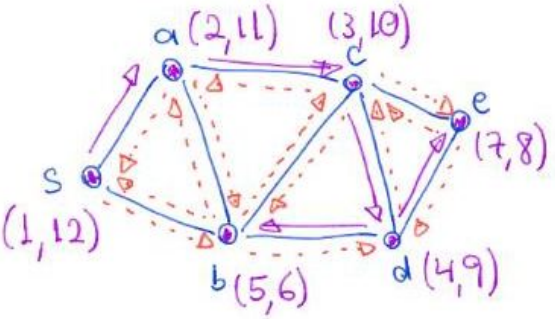












Antes de começar a busca em profundidade, é necessário uma inicialização

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

```
void buscaProf(Graph G, int s, int *ordem_chegada, int *ordem_saida)
{
   int i, k;

   /* inicializa todos como não encontrados */
```

```
    for (i = 0; i < G->n; i++)
    {
        ordem_chegada[i] = -1;
        ordem_saida[i] = -1;
    }
    k = 1;
    buscaProfR(G, s, ordem_chegada, ordem_saida, &k);
    // buscaProfI(G, s, ordem_chegada, ordem_saida, k);
}
```

- 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 (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 k
    - a partir de um apontador pk. Por que?

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

```
void buscaProfR(Graph G, int v, int *ordem_chegada, int *ordem_saida, int *pk)
{
   int w;
   ordem_chegada[v] = (*pk)++;
   /* 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, pk);
   ordem_saida[v] = (*pk)++;
}
```

- Qual a eficiência deste algoritmo?

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

```
void buscaProfR(Graph G, int v, int *ordem_chegada, int *ordem_saida, int *pk)
{
   int w;
   link p;
   ordem_chegada[v] = (*pk)++;
   /* para cada vizinho de v que ainda não foi visitado */
   p = G->A[v];
   while (p != NULL)
   {
       w = p->index;
       if (ordem_chegada[w] == -1)
           buscaProfR(G, w, ordem_chegada, ordem_saida, pk);
       p = p->next;
   }
   ordem_saida[v] = (*pk)++;
}
```

- Qual a eficiência deste algoritmo?

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 não estar explorado.
    - No entanto, neste caso o vértice é marcado como explorado
        - 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 explorados não sejam adicionados à pilha,
    - existe a possibilidade de um vértice ser colocado
        - mais de uma vez na pilha, antes de ter sido explorado.
    - Mas, nesse caso ele será marcado como explorado
        - 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 k diretamente.
    - Por que não precisam operar com um apontador pk?

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

```
void buscaProfI(Graph G, int s, int *ordem_chegada, int *ordem_saida, int k)
{
   int v, w;
   // pilha implementada em vetor
   int *p;
   int t = 0;
   p = malloc(G->m * sizeof(int));

   /* colocando s na pilha */
   p[t++] = s;
   /* enquanto a pilha dos ativos (encontrados
   mas não visitados) não estiver vazia */
   while (t > 0)
   {
       /* remova o mais recente da pilha */
       v = p[--t];
       if (ordem_chegada[v] == -1) // se v nao foi visitado
       {
           ordem_chegada[v] = k++;
           p[t++] = v; // empilha o vértice pra saber quando marcar o tempo de
término
           /* para cada vizinho deste que ainda não foi visitado */
           for (w = 0; w < G->n; w++)
               if (G->A[v][w] == 1 && ordem_chegada[w] == -1)
                   p[t++] = w; // empilha o vizinho
       }
       else if (ordem_saida[v] == -1)
           ordem_saida[v] = k++;
```

```
    }
    free(p);
}
```

- Qual a eficiência deste algoritmo?

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

```
void buscaProfI(Graph G, int s, int *ordem_chegada, int *ordem_saida, int k)
{
    int v, w;
    link q;
    // pilha implementada em vetor
    int *p;
    int t = 0;
    p = malloc(G->m * sizeof(int));

    /* colocando s na pilha */
    p[t++] = s;
    /* enquanto a pilha dos ativos (encontrados
    mas não visitados) não estiver vazia */
    while (t > 0)
    {
        /* remova o mais recente da pilha */
        v = p[--t];
        if (ordem_chegada[v] == -1) // se v nao foi visitado
        {
            ordem_chegada[v] = k++;
            p[t++] = v; // empilha o vértice pra saber quando marcar o tempo de
término
            /* para cada vizinho deste que ainda não foi visitado */
            q = G->A[v];
            while (q != NULL)
            {
                w = q->index;
                if (ordem_chegada[w] == -1)
                    p[t++] = w; // empilha o vizinho
                q = q->next;
            }
        }
        else if (ordem_saida[v] == -1)
            ordem_saida[v] = k++;
```

```
    }
}
```

- Qual a eficiência deste algoritmo?

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 ou BFS,
    - 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 busca(G, v, num_comps) uma variante de uma DFS ou BFS
    - que atribui rótulo num_comps para todo vértice w encontrado,
        - i.e., que faz comp[w] = num_comps
- Observe que num_comps só é incrementado quando uma busca termina
    - e a execução volta para o laço principal de 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 lista de adjacência
- e usando busca em profundidade.

```
void buscaCompR(Graph G, int v, int *comp, int k)
{
   int w;
```

```
    link p;
    comp[v] = k;
    /* para cada vizinho de v que ainda não foi visitado */
    p = G->A[v];
    while (p != NULL)
    {
        w = p->index;
        if (comp[w] == -1)
            buscaCompR(G, w, comp, k);
        p = p->next;
    }
}

void identComponetes(Graph G, int *comp)
{
    int i, k;
    /* inicializa todos como não pertencentes */
    for (i = 0; i < G->n; i++)
        comp[i] = -1;
    k = 0;
    for (i = 0; i < G->n; i++)
        if (comp[i] == -1)
        {
            k++;
            buscaCompR(G, i, comp, k);
        }
}
```