PAA - Aula 10
Busca em Profundidade e Ordenação Topológica

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.

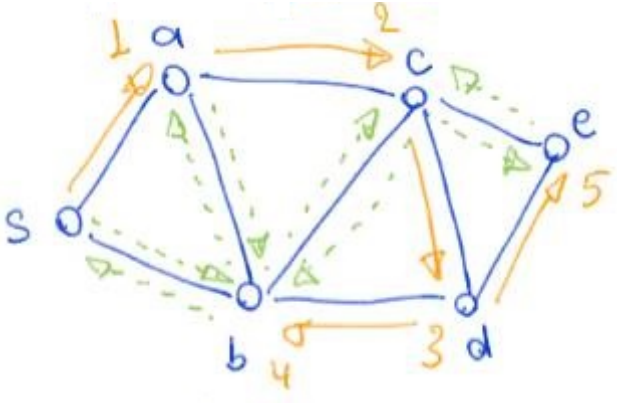
O comportamento da DFS está intimamente relacionado

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

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.

Exemplo de busca em profundidade:



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.

Pseudocódigo da 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.

- Resultado segue da corretude do algoritmo de busca genérica.

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.

- Para ver 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.
- Nesse 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,
    - um vértice pode ser empilhado várias vezes, antes de ser visitado.
- 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.
- Assim, concluímos que o algoritmo executa um número de passos
    - limitado superiormente pelo número de vértices
        - mais arestas da componente de s, ou seja, $O(n_s + m_s)$.

Pseudocódigo da 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.

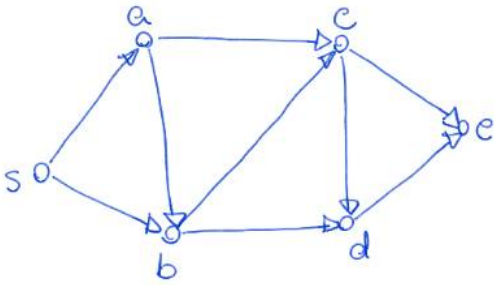
Ordenação topológica

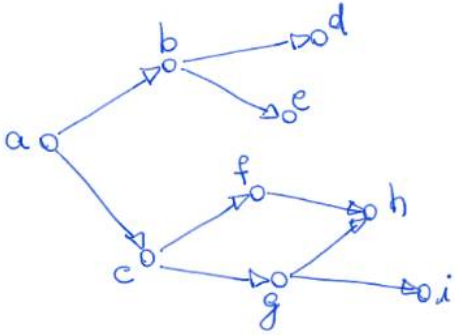
A primeira aplicação específica da busca em profundidade que veremos

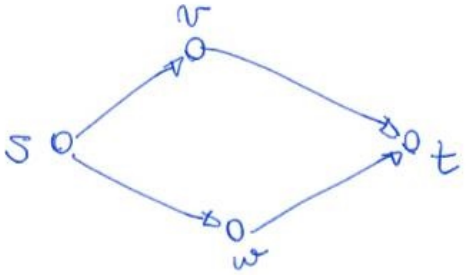
- é encontrar uma ordenação topológica num grafo dirigido acíclico,
    - também conhecido por DAG (Directed Acyclic Graph).

Para tanto, primeiro precisamos entender o que é um DAG,

- ou seja, um grafo orientado que não possui ciclos.

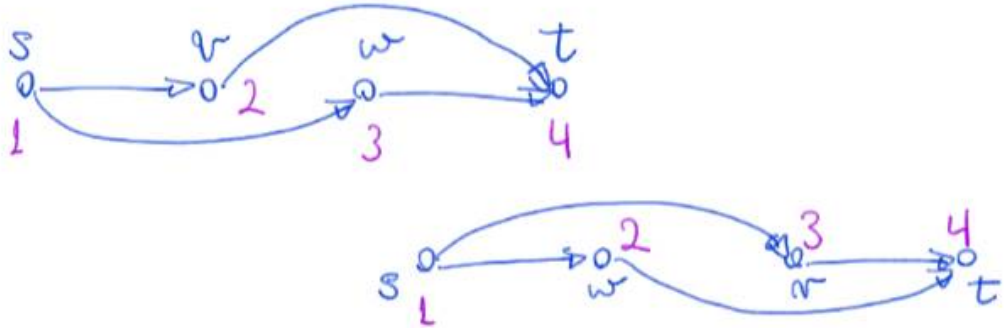






Também precisamos definir uma ordenação topológica, que corresponde a

- uma rotulação f dos vértices de um grafo, tal que
    - i.e., cada vértice tem exatamente um rótulo inteiro em [1, n];
- e na qual, para qualquer arco (u, v) temos



Notem que, em ambas as ordenações anteriores s é o primeiro vértice,

- o que ocorre porque nenhum arco entra em s.
    - Chamamos esses vértices de fontes (ou sources).
- De modo complementar, as ordenações terminam com o vértice t,
    - do qual nenhum arco sai.
        - Chamamos esses vértices de sorvedouros (ou sinks).
- Quiz1: Uma ordenação topológica sempre começa numa fonte e termina em um sorvedouro?
- Quiz2: Um grafo pode ter várias fontes e/ou sorvedouros?

A motivação para o problema de obter uma ordenação topológica

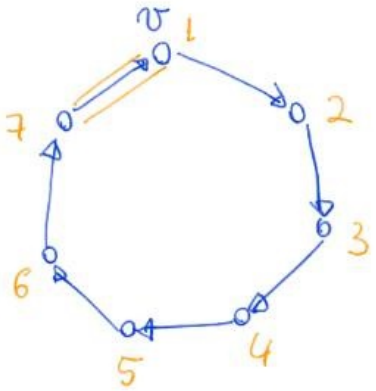
- é encontrar uma ordem para realizar uma sequência de tarefas,
    - que respeite as restrições de precedência entre as tarefas,
        - as quais são representadas pelos arcos.

Antes de resolver o problema, vamos estudar uma relação interessante

- (e importante para nossa aplicação): Um grafo orientado
    - é acíclico se, e somente se, ele possui uma ordenação topológica.

Demonstração: (←) Primeiro, vamos mostrar a volta.

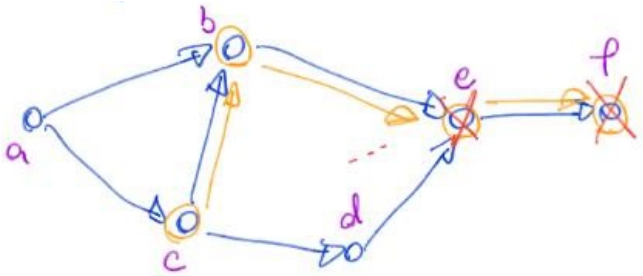
- No caso em que o grafo orientado possui uma ordenação topológica,
    - vamos supor, por contradição, que ele possui um ciclo.



Observe que algum vértice do ciclo tem que ter o menor rótulo dentre os vértices do mesmo, e isso leva a uma aresta que viola a propriedade da ordenação topológica

(→) para provar a ida vamos fazer uma prova construtiva.

- Já que o grafo é acíclico, vamos seguir um caminho neste grafo.
- Eventualmente (depois de no máximo n-1 arcos) esse caminho acaba
    - em um vértice que não tem arcos saindo dele.
- Caso contrário o caminho não teria acabado (se os arcos fossem para novos vértices) ou teríamos um ciclo (se os arcos fossem para vértices já visitados).



Chamamos vértices sem arcos de saída de sorvedouros (ou do inglês, sinks).

- Notem que é seguro colocar este vértice sorvedouro como o último vértice
    - da nossa ordenação topológica, ou seja, colocar o rótulo n nele.
- Feito isso, podemos removê-lo do grafo e repetir o processo,
    - já que o grafo resultante continua acíclico.
- Isso vale porque, ao remover um vértice sorvedouro
    - (e os arcos que incidiam nele) não criamos ciclos.
- Repetindo o raciocínio para encontrar um sorvedouro no DAG restante
    - ou, mais formalmente, usando indução matemática,
        - temos a demonstração do resultado.

Uma maneira bastante eficiente de implementar essa ideia de

- "remover" um sorvedouro por vez é usando busca em profundidade
    - com um laço externo sobre os vértices e um contador decrescente,
- como mostra o seguinte algoritmo.

```
LoopBuscaProf(grafo G=(V,E)):
        marque todos os vértices em V como não visitados
        rotulo-atual = n
        para cada v \in V
                se v não foi visitado
                        buscaProfRec(G, v)


buscaProfRec(grafo G=(V,E), vértice v):
        marque v como visitado
        para cada arco (v, w)
                se w não foi visitado
                        buscaProfRec(grafo G=(V,E), vértice w)
        f(v) = rotulo-atual
        rotulo-atual--
```

Análise de eficiência: A eficiência de tempo deste algoritmo é O(n + m),

- derivada da eficiência da busca em profundidade.

Análise de corretude: Para verificar que nosso algoritmo obtém

- uma ordenação topológica correta, vamos considerar um arco (u, v) qualquer
    - e queremos mostrar que f(u) < f(v).
- Lembrem que a rotulação ocorre quando o nó é finalizado
    - e que os valores dos rótulos são decrescentes.

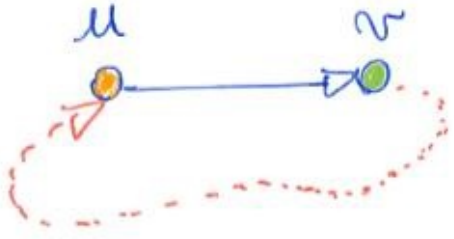
Temos dois casos:

- (i) se u for visitado antes de v.
- (ii) se v for visitado antes de u.

Analisando o caso (i), em que u foi visitado antes de v.

- Como a busca em profundidade não volta
    - até encontrar tudo que for possível,
- ela vai encontrar e rotular v antes de voltar e rotular u,
    - já que existe o arco (u, v).
- Como os rótulos só decrescem, temos f(u) < f(v).

Analisando o caso (ii), em que v foi visitado antes de u.

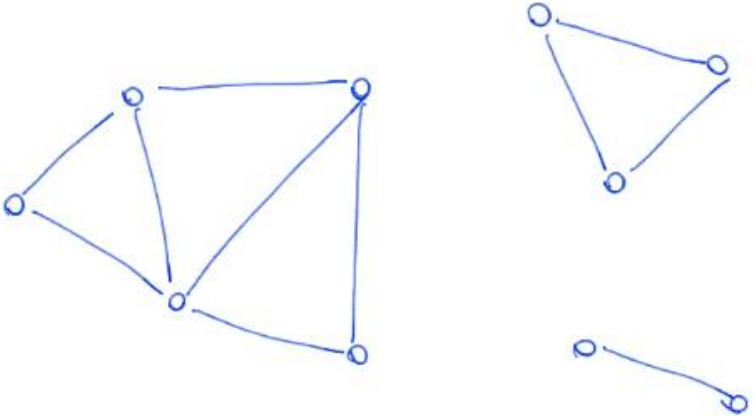


- Sabemos que não existe caminho de v para u,
    - caso contrário este caminho junto do arco (u, v)
        - formaria um ciclo.
- Portanto, v será rotulado antes de u ser visitado.
- Eventualmente, em outra chamada do laço externo
    - o vértice u será visitado e rotulado.
- Novamente, como os rótulos só decrescem, temos $f(u) < f(v)$.

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.

Pseudocódigo:

```
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)
```

Note que, busca(G, v, num_comps) é uma variante da busca genérica

- 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.