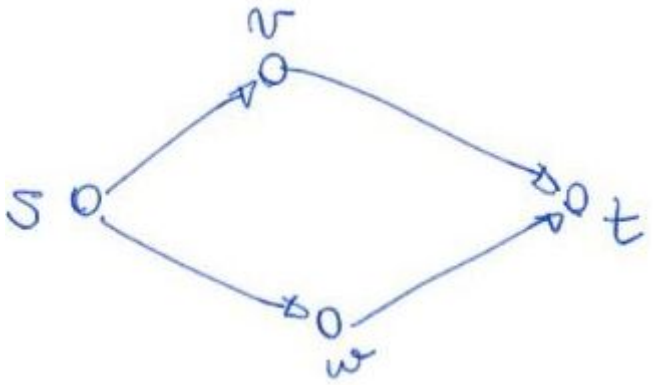
AED2 - Aula 24
Ordenação topológica, DFS, DAGs aleatórios

A primeira aplicação 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

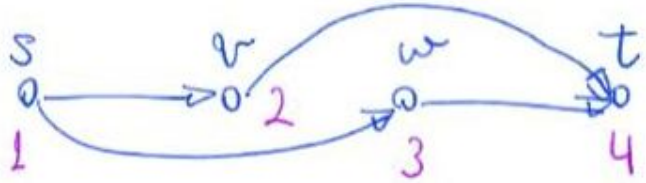
- o que é um DAG,
    - ou seja, um grafo orientado que não possui ciclos.

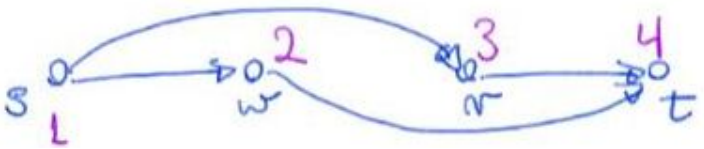


- e o que é uma ordenação topológica.

Ordenação topológica é uma ordem/rotulação f dos vértices de um grafo, tal que:

- U_{v \in V} {f(v)} = {1, ..., n},
    - i.e., cada vértice tem exatamente um rótulo inteiro em [1, n].
- Para qualquer arco **(u, v)** temos **f(u) < f(v)**.





Note que em ambas as ordenações s é o primeiro vértice e que nenhum arco entra em s

Lo chamamos esses vértices de fontes (ou sources)

De modo complementar, as ordenações terminam com o vértice t, do qual nenhum arco sai

Lo chamamos esses vértices de sorvedouros (ou sinks)

A motivação para este problema é

- encontrar uma ordem possível para realizar um sequência de tarefas
    - de modo a respeitar as restrições de precedência entre estas tarefas,
        - que são representadas pelos arcos.

Uma relação interessante (e importante para nossa aplicação)

- é que um grafo orientado
    - é acíclico se e somente se possui uma ordenação topológica.

(<--) primeiro vamos mostrar a volta através da contrapositiva

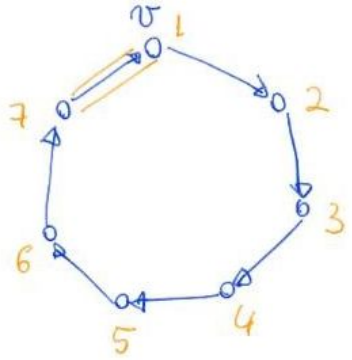
A contrapositiva de: a --> b é ~b --> ~a, portanto, a contra-positiva de:

- o grafo orientado possui uma ordenação topológica --> o grafo ser acíclico

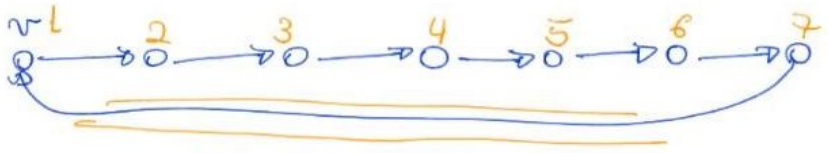
é

- o grafo orientado ter um ciclo --> o grafo não possui ordenação topológica.

- Se um DAG tem um ciclo, então ele não tem ordenação topológica



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



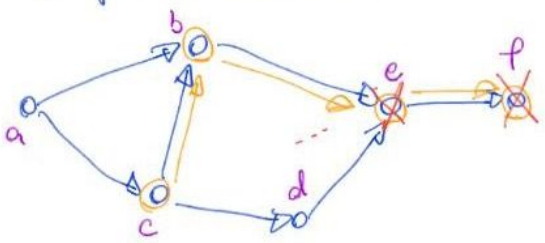
No caso em que o grafo orientado possui um ciclo,

- vamos supor por contradição que temos uma ordenação topológica f válida.
- Nesta ordenação algum vértice v do ciclo tem o menor rótulo, mas
    - como trata-se de um ciclo, existe um outro vértice w do ciclo
        - que tem um arco (w, v) incidindo em v.
- Pela escolha de v, sabemos que o rótulo de v é menor que o rótulo de w,
    - ou seja, f(v) < f(w).
- No entanto, para ser uma ordenação topológica válida,
    - como existe o arco (w, v), deveríamos ter f(w) < f(v).
- Chegamos a um absurdo.

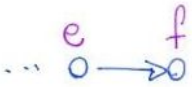
(-->) para provar a ida vamos fazer uma prova construtiva

- Se um DAG não tem ciclos, então ele tem ordenação topológica

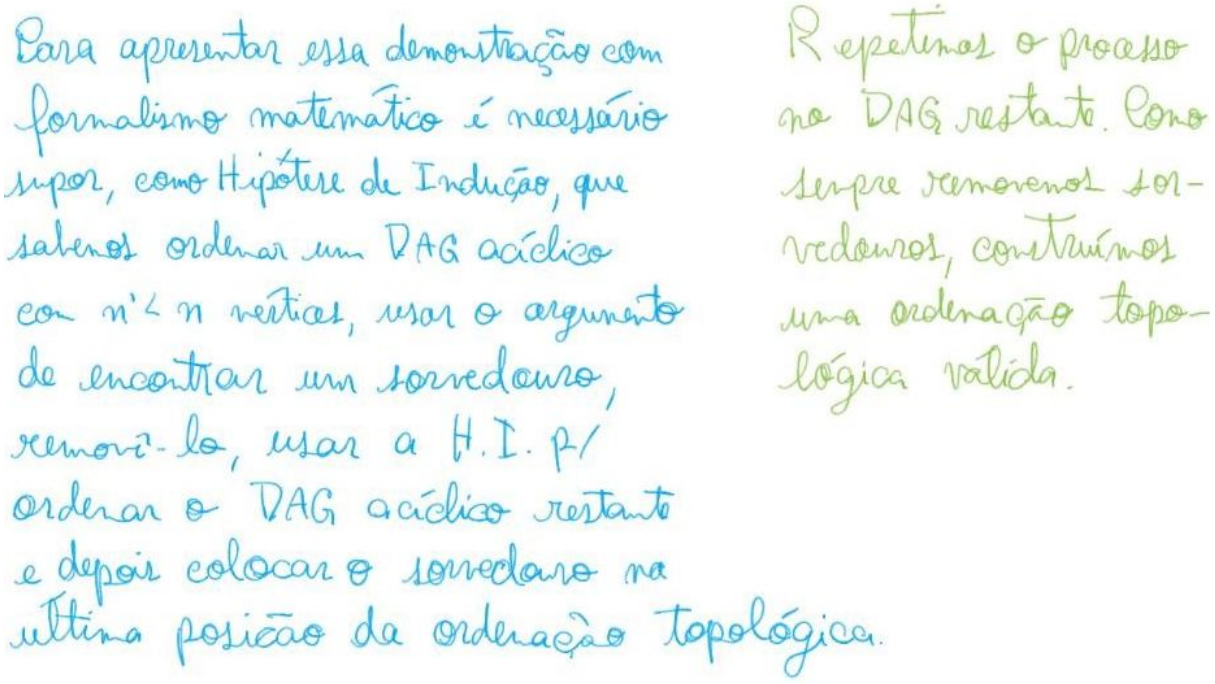
Exemplo p/ prova construtiva:



Eventualmente o caminho laranja termina em um vértice sorvedouro f



Removemos esse vértice f e o colocamos na posição final da ordenação topológica.



Já que o grafo é acíclico, vamos seguir um caminho neste grafo.

- Eventualmente (depois de no máximo n-1 arcos)
    - esse caminho tem que acabar,
        - caso contrário teríamos que repetir algum vértice
            - o que resultaria em um ciclo.
- Note que, o último vértice deste caminho não pode ter arcos saindo dele,
    - caso contrário o caminho não teria acabado.
- Chamamos vértices sem arcos de saída de vértices sorvedouros
    - ou, do inglês, sinks.
- Notem ainda que é seguro colocar este vértice
    - como o último da nossa ordenação topológica,
        - ou seja, colocar o rótulo n nele.
- Feito isso, podemos removê-lo do grafo e recomeçar o processo.
- Já que ao remover um vértice sorvedouro (e os arcos que incidiam nele)
    - nós não criamos ciclos, o grafo resultante continua acíclico.
- Repetindo o mesmo raciocínio para encontrar sorvedouro no grafo restante,
    - ou mais formalmente usando indução matemática,
        - temos a demonstração do resultado.
- Além disso, temos uma ideia para um algoritmo para este problema.

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)
        defina f(v) = rotulo-atual
        rotulo-atual--
}
```

Código para identificar componentes

- com grafo implementado por lista de adjacência
- e usando busca em profundidade

```
void ordenacaoTopologica(Graph G, int *ordTopo)
{
   int i, k, *visitado;
   visitado = malloc(G->n * sizeof(int));
   /* inicializa todos como não visitados e sem ordem topologica */
   for (i = 0; i < G->n; i++)
   {
       visitado[i] = 0;
       ordTopo[i] = -1;
   }
   k = G->n;
   for (i = 0; i < G->n; i++)
       if (visitado[i] == 0)
       {
           buscaProfOrdTopoR(G, i, visitado, ordTopo, &k);
           // buscaProfOrdTopoI(G, i, visitado, ordTopo, &k);
       }
   free(visitado);
}
```

Código da busca em profundidade recursiva adaptado para ordenação topológica

```
void buscaProfOrdTopoR(Graph G, int v, int *visitado, int *ordTopo, int *pk)
{
   int w;
   link p;
```

```
    visitado[v] = 1;
    /* para cada vizinho de v que ainda não foi visitado */
    p = G->A[v];
    while (p != NULL)
    {
        w = p->index;
        if (visitado[w] == 0)
            buscaProfOrdTopoR(G, w, visitado, ordTopo, pk);
        p = p->next;
    }
    ordTopo[v] = (*pk)--;
}
```

Código da busca em profundidade iterativa adaptado para ordenação topológica

```
void buscaProfOrdTopoI(Graph G, int s, int *visitado, int *ordTopo, int *pk)
{
    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 (visitado[v] == 0) // se v nao foi visitado
        {
            visitado[v] = 1;
            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 (visitado[w] == 0)
                    p[t++] = w; // empilha o vizinho
                q = q->next;
            }
        }
        else if (ordTopo[v] == -1) // se v ja foi visitado e sua ordem topologica ainda nao
foi atribuida
            ordTopo[v] = (*pk)--;
```

```
    }
}
```

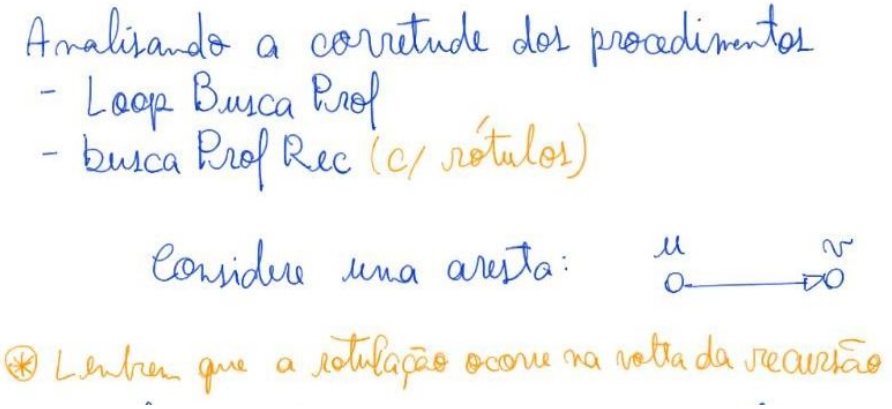
Análise de eficiência:

- A eficiência deste algoritmo é O(n + m),
    - derivado da eficiência da busca em profundidade.

Análise de corretude:

Para verificar que ele obtém uma ordenação topológica correta,

- vamos considerar um arco (u, v) qualquer
- e queremos mostrar que f(u) < f(v).



Temos dois casos:

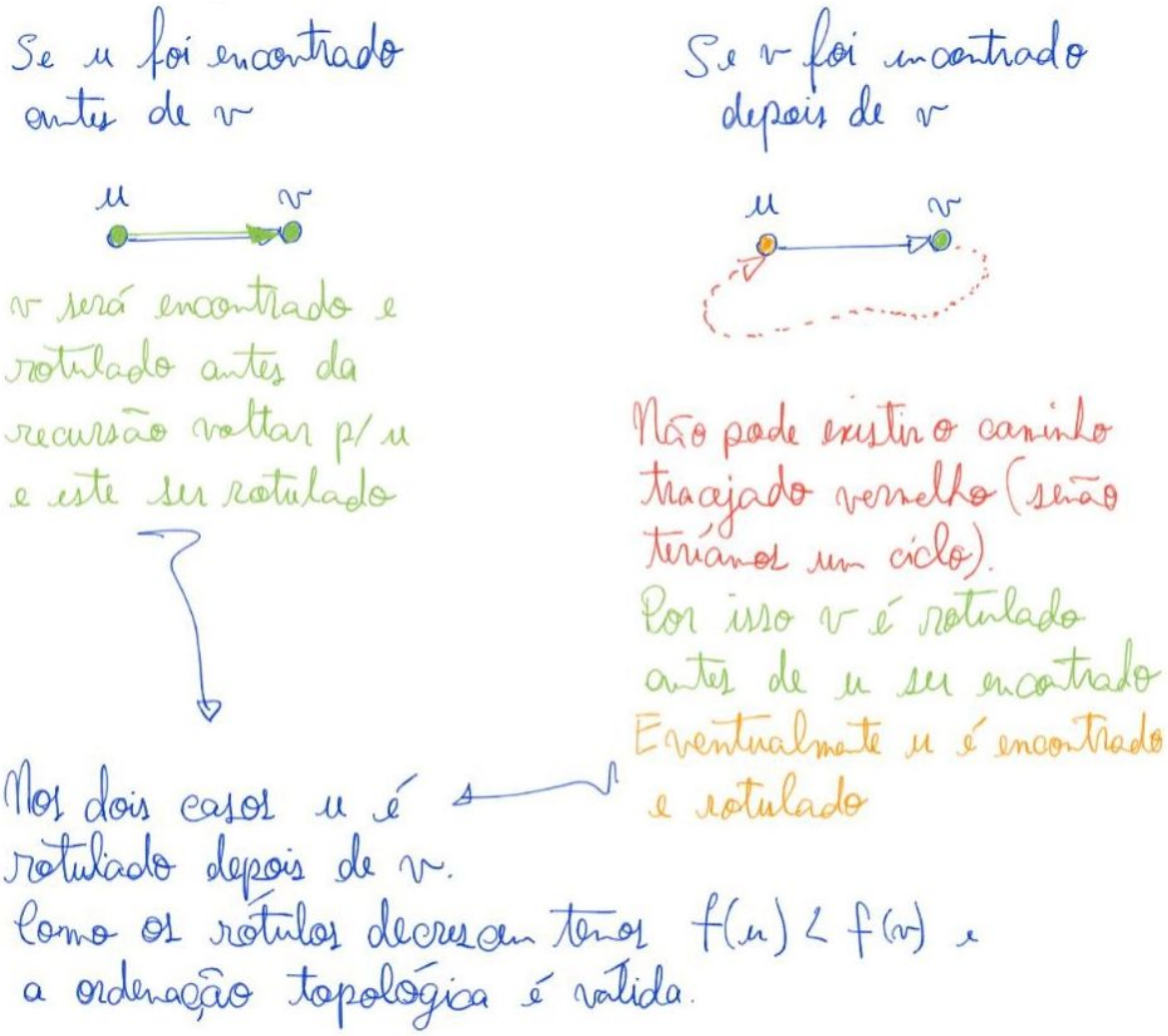
- (i) se u for visitado antes de v.
- (ii) se v for visitado antes de u.

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 arco (u, v).
- Como os rótulos só decrescem, temos f(u) < f(v).

Analisando 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).



Geração aleatória de DAGs

Uma vez que aprendemos mais quando brincamos com nosso objeto de estudo

- como podemos modificar nossos geradores de grafos aleatórios
  - para produzir DAGs?
- A princípio poderíamos testar, antes de inserir um arco, se ele gera ciclo,
  - usando alguma busca em grafo.
- No entanto, no início da aula mostramos que
  - um grafo dirigido é acíclico ⇔ ele tem ordenação topológica.
- Então, podemos escolher aleatória e uniformemente uma ordenação,
  - que é simplesmente uma permutação **perm[ ]** dos vértices do grafo,
- e testar, antes de inserir cada arco **(v, w)**,
  - se ele respeita essa ordenação,
    - i.e., se **perm[v] < perm[w]**.
- A princípio, para simplificar, considere a ordenação canônica,
  - i.e., perm[v] = v + 1, para v em [0, n).

```
perm = malloc(n * sizeof(int));
for (i = 0; i < n; i++)
```

```
        perm[i] = i + 1;
```

- Vamos verificar como os dois métodos de geração de grafos aleatórios
    - que vimos anteriormente, são modificados para gerar DAGs.
- Código do método 1, que sorteia os extremos de cada arco

```
/* A função randV() devolve um vértice aleatório do grafo G. Vamos supor que G->V <=
RAND_MAX. */
int randV(Graph G)
{
   double r;
   r = rand() / (RAND_MAX + 1.0);
   return r * G->n;
}

Graph graphRand1(int n, int m, int *perm)
{
   Graph G = graphInit(n);
   while (G->m < m)
   {
       int v = randV(G);
       int w = randV(G);
       if (perm[v] < perm[w]) // verificando se o arco respeita a orientação do DAG da por
perm
           graphInsertArc(G, v, w);
   }
   return G;
}
```

- Código do método 2, que considera cada arco, e “joga uma moeda”,
    - com probabilidade m / número máximo de arcos,
        - para decidir se ele será inserido.

```
Graph graphRand2(int n, int m, int *perm)
{
   double prob = (double)m / n / (n - 1) * 2; // ajuste no cálculo da probabilidade, pois
número máximo de arcos num DAG é menor
   Graph G = graphInit(n);
   for (int v = 0; v < n; v++)
       for (int w = 0; w < n; w++)
           if (perm[v] < perm[w]) // verificando se o arco respeita a orientação do DAG da
por perm
               if (rand() < prob * (RAND_MAX + 1.0))
                   graphInsertArc(G, v, w);
   return G;
}
```

- Para o caso geral, uma maneira eficiente (tempo linear) de transformar
    - uma permutação qualquer em uma permutação aleatória dos vértices,
        - escolhida com probabilidade uniforme,
            - é o algoritmo Embaralhamento de Knuth.

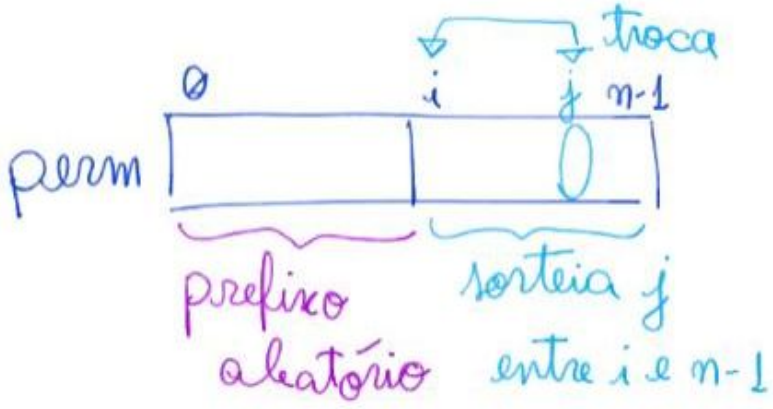
```
/* Sorteia um inteiro em [0, n) */
```

```c
int uniformeAleat(int n)
{
   return (double)rand() / (RAND_MAX + 1) * n;
}

void troca(int *a, int *b)
{
   int aux;
   aux = *a;
   *a = *b;
   *b = aux;
}

// Knuth shuffles
int *permAleat(int *perm, int n)
{
   int i, j;
   for (i = 0; i < n - 1; i++)
   {
       j = i + uniformeAleat(n - i);
       troca(&perm[i], &perm[j]);
   }
   return perm;
}
```

- Eficiência: O(n).
- Invariante:
    - no início de cada iteração do laço
        - v[0 .. n - 1] é uma permutação do vetor original,
        - v[0 .. i - 1] é um prefixo escolhido com prob. 1 / (n! / (n - i)!).



    - Ao final das iterações v[0 .. n - 1] é uma permutação
        - escolhida com probabilidade 1 / n!
- Ressalto que, permutações aleatórias são úteis em outros contextos,
    - como no projeto de algoritmos probabilísticos para problemas difíceis,
        - ao permitir que façamos escolhas aleatórias
            - em passos arbitrários de algoritmos determinísticos.