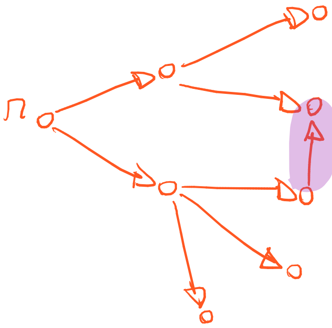
AED2 - Aula 23
Ordenação topológica, DFS, DAGs aleatórios

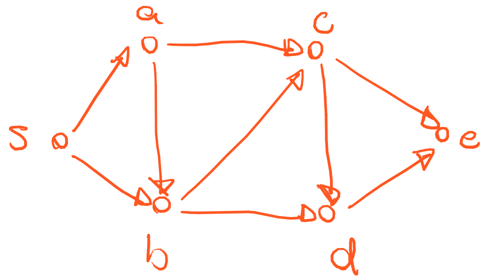
A primeira aplicação específica da busca em profundidade que veremos

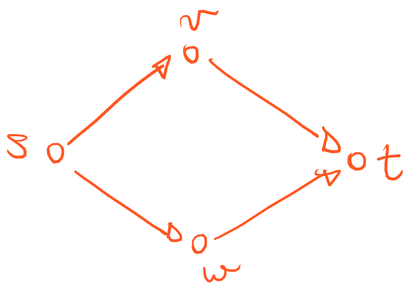
- é encontrar uma ordenação topológica num grafo dirigido acíclico,
  - também conhecido por DAG (Directed Acyclic Graph).

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

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

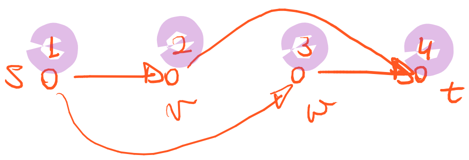


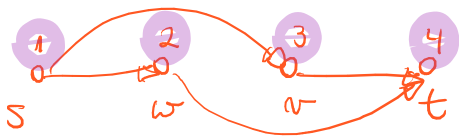




Também precisamos definir uma ordenação topológica, que corresponde a

- uma rotulação f dos vértices de um grafo, tal que $\bigcup_{v \in V} \{f(v)\} = \{1, \dots, n\}$
  - i.e., cada vértice tem exatamente um rótulo inteiro em [1, n];
- e na qual, para qualquer arco (u, v) temos $f(u) < f(v)$





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,
    - possui uma ordenação topológica.

$\forall (u, v) \quad f(u) < f(v)$

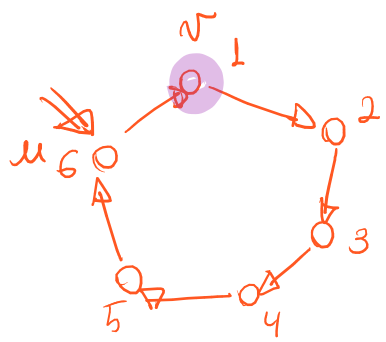
$\Uparrow$

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.

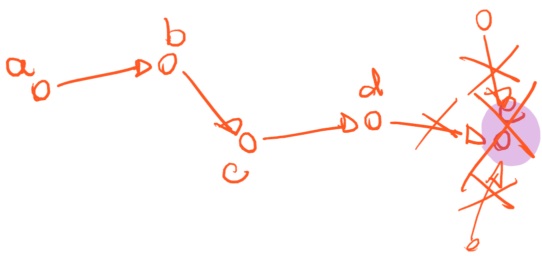


$\exists (u, v) : f(u) > f(v)$

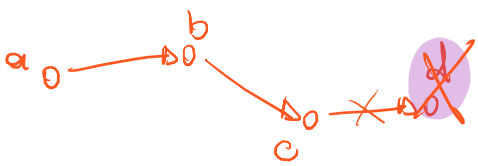
↳ contradiz o grafo ter uma ordenação topológica

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

$\forall (u, v) : f(u) < f(v)$



$f(e) = n$



$f(d) = n-1$

- Repetindo o raciocínio para encontrar um sorvedouro no DAG 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.

Loop Busca Prof ($G = (V, E)$)
marque todo $v \in V$ como ñ visitado
rotulo-atual = n
para cada $v \in V$
se v ñ foi visitado
busca Prof Rec (G, v)

busca Prof Rec (G, v)
marque v como visitado
para cada arco (v, w)
se w ñ foi visitado
busca Prof Rec (G, w)
f(v) := rotulo-atual
rotulo-atual --

Quiz3: qual a relação entre o rótulo da ordenação topológica

- e o tempo de término de um nó?

Código para identificar componentes

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

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

Código da busca em profundidade recursiva adaptado para ordenação topológica

```
void buscaProfOrdTopoR(Grafo G, int v, int *visitado, int *ordTopo,
            int *prot_atual) {
    int w;
    Noh *p;
    visitado[v] = 1;
    /* para cada vizinho de v que ainda não foi visitado */
    for (p = G->A[v]; p != NULL; p = p->prox) {
        w = p->rotulo;
        if (visitado[w] == 0)
            buscaProfOrdTopoR(G, w, visitado, ordTopo, prot_atual);
    }
    ordTopo[v] = (*prot_atual)--;
}
```

Código da busca em profundidade iterativa adaptado para ordenação topológica

```
void buscaProfOrdTopoI(Grafo G, int origem, int *visitado,
            int *ordTopo, int *prot_atual) {
    int v, w;
    Noh *p;
    int topo = 0; // pilha implementada em vetor
    int *pilha = malloc(G->m * sizeof(int));
    pilha[topo++] = origem; // colocando vértice origem na pilha
    /* enquanto a pilha dos ativos (encontrados
    mas não visitados) não estiver vazia */
    while (topo > 0) {
        v = pilha[--topo]; // remova o mais recente da pilha
        if (visitado[v] == 0) { // se v não foi visitado
            visitado[v] = 1;
            pilha[topo++] = v; // empilha o vértice. Por que?
            /* para cada vizinho que ainda não foi visitado */
            for (p = G->A[v]; p != NULL; p = p->prox;) {
                w = p->rotulo;
                if (visitado[w] == 0)
                    pilha[topo++] = w; // empilha o vizinho
            }
        }
        else if (ordTopo[v] == -1) // se v já foi visitado e sua
ordem topológica ainda não foi atribuída
            ordTopo[v] = (*prot_atual)--;
    }
}
```

Análise de eficiência:

- A eficiência de tempo deste algoritmo é $O(n + m)$,
    - derivada da eficiência da busca em profundidade.

Análise de corretude:

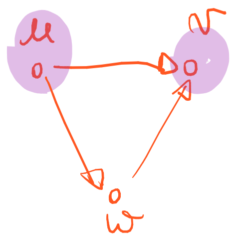
- Para verificar que nosso algoritmo obtém uma ordenação topológica correta,
    - vamos considerar um arco (u, v) qualquer
        - e queremos mostrar que $f(u) < f(v)$.
- Lembrem que a rotulação ocorre quando o nó é finalizado
    - e que os valores dos rótulos são decrescentes.

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 $f(u) < f(v)$.

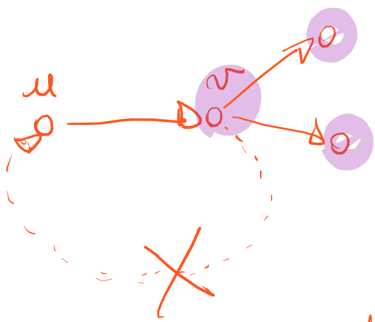


v será finalizado
antes de u

v recebe um rótulo
maior que o de u

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

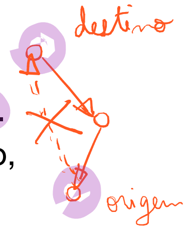
- Sabemos que não existe caminho de v para u,
    - caso contrário este caminho junto do arco (u, v)
        - formaria um ciclo.
- Portanto, v será rotulado antes de u ser visitado.
- Eventualmente, em outra chamada do laço externo
    - o vértice u será visitado e rotulado.
- Novamente, como os rótulos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



v será finalizado
antes de v ser visitado

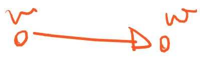
↳ ñ pode existir
porque o grafo é acíclico

Geração aleatória de DAGs



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

- vamos modificar nossos geradores de grafos aleatórios para produzir DAGs.
- A princípio, poderíamos testar, antes de inserir um arco, se ele gera um 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].
- Note que, nossa permutação mapeia vértices para posições na ordenação.



A princípio, para simplificar, considere a ordenação canônica,

- i.e., perm[v] = v + 1, para v em [0, n).

```c
perm = malloc(n * sizeof(int));
for (i = 0; i < n; i++)
    perm[i] = i + 1;
```

— por que?

- 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

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

— $\{0, \dots, RAND\_MAX\}$

— $\in [0, 1)$

$\in \{0, \dots, n-1\} \in [0, n)$

```c
Grafo DAGaleatorio1(int n, int m, int *perm) {
    Grafo G = inicializaGrafo(n);
    while (G->m < m) {
        int v = verticeAleatorio(G);
        int w = verticeAleatorio(G);
// verificando se o arco respeita a ordenação do DAG dada por perm[]
        if (perm[v] < perm[w])
            insereArcoGrafo(G, v, w);
    }
    return G;
}
```

— se (v, w) respeita a ordenação perm[ ]

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.

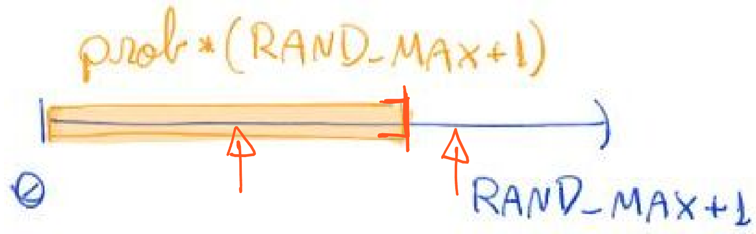
```
Grafo DAGaleatorio2_1(int n, int m, int *perm) {
    // ajuste no cálculo da probabilidade, pois número máximo de
arcos num DAG é menor
    double prob = (double)m / n / (n - 1) * 2;
    Grafo G = inicializaGrafo(n);
    for (int v = 0; v < n; v++)
        for (int w = 0; w < n; w++) {
// verificando se o arco respeita a ordenação do DAG dada por perm[]
            if (perm[v] < perm[w])
                if (rand() < prob * (RAND_MAX + 1.0))
                    insereArcoNaoSeguraGrafo(G, v, w);
        }
    return G;
}
```

$prob = \frac{m}{\#\text{ máx. de arcos}} = \frac{m}{\frac{n(n-1)}{2}}$

$\{0, \ldots, RAND\_MAX\}$ $\in [0,1)$

$\frac{rand()}{(RAND\_MAX+1)} < prob$

$\in [0, 1)$

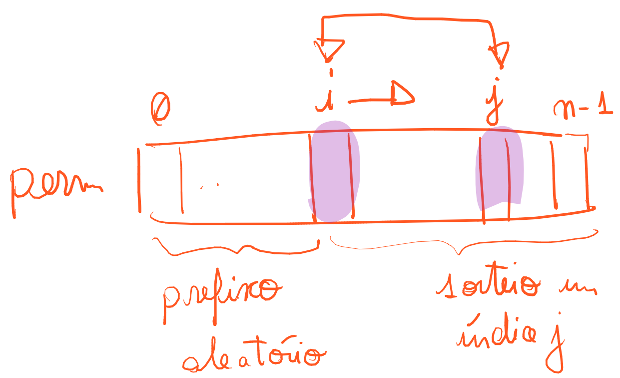


Para o caso geral, uma maneira eficiente de transformar uma permutação qualquer

- em uma permutação aleatória e uniforme dos vértices
    - é o algoritmo Embaralhamento de Knuth,
- criado pelo famoso [Donald Knuth](#).

Ideia:

- Dada uma permutação em um vetor,
    - percorrer o vetor da esquerda para a direita
- e em cada iteração escolher uniforme e aleatoriamente
    - um elemento do sufixo do vetor
        - para trocar com o elemento da posição corrente.



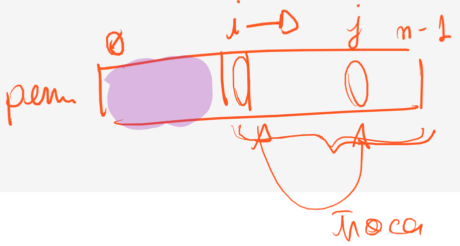
Código:

```c
/* Sorteia um inteiro em [0, n) */
int uniformeAleat(int n) {
    return (int)((double)rand() / (double)(RAND_MAX + 1) * n);
}

// Knuth shuffles
int *permAleat(int *perm, int n) {
    int i, j;
    for (i = 0; i < n - 1; i++) { // Quiz4: Por que n - 1?
        j = i + uniformeAleat(n - i);
        troca(&perm[i], &perm[j]);
    }
    return perm;
}
```

$\in [0,1)$ $\in \{0,1,\dots,n-1\}$ $\in \{0,\dots,RAND\_MAX\}$ $\in [0,n)$

$\in \{i,\dots,n-1\}$ $\in \{0,\dots,n-i-1\}$



Invariante e corretude:

- 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)!). $= \frac{1}{n.(n-1)\dots(n-i+1)}$
- Ao final das iterações v[0 .. n - 1] é uma permutação
  - escolhida com probabilidade 1 / n!, $= 1/(n!/(n-i)!) = \frac{1}{\frac{n!}{(n-\cancel{(n-1)})!}} = \frac{1}{n!}$
- sendo que n! é o número total de permutações com n elementos.

Eficiência de tempo: O(n).

Eficiência de espaço: O(1).

Nos nossos algoritmos usamos permutações para mapear vértices $perm[v] = K$

- na posição dos mesmo na ordenação topológica.
- Usando permutações no sentido inverso, i.e.,
  - mapeando uma posição da ordenação
    - no rótulo do vértice que a ocupará,
  - podemos projetar algoritmos mais eficientes.
- Quiz5: O que muda esses algoritmos? E qual a mudança na eficiência deles?

Destaco que, permutações aleatórias são úteis para:

- Testar empiricamente o comportamento de caso médio dos algoritmos,
  - especialmente porque este costuma ser mais difícil de analisar
    - do que pior e melhor caso.
- Projetar algoritmos probabilísticos para problemas difíceis,
  - ao permitir que façamos escolhas aleatórias
    - em passos arbitrários de algoritmos determinísticos,
  - a fim de explorar melhor o espaço de soluções.