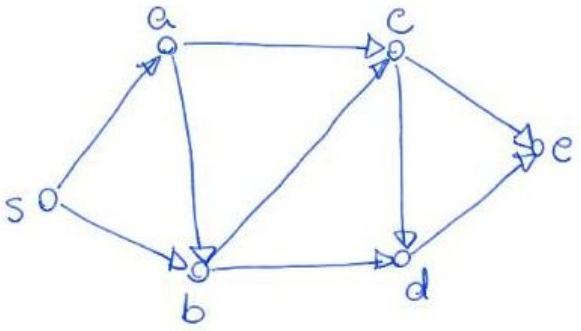
AED2 - Aula 23
Ordenação topológica, DFS, DAGs aleatórios

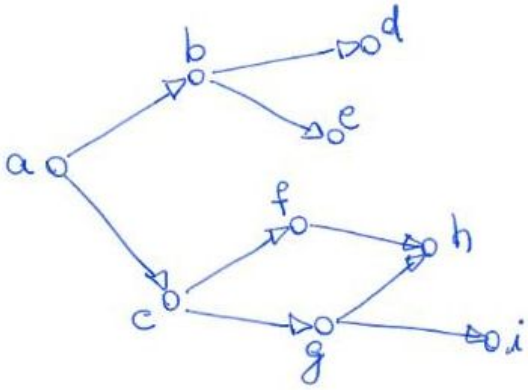
A primeira aplicação específica da busca em profundidade que veremos

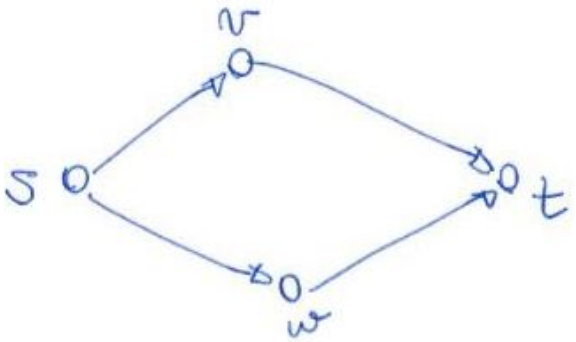
- é encontrar uma ordenação topológica num grafo dirigido acíclico,
    - também conhecido por DAG (Directed Acyclic Graph).

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

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



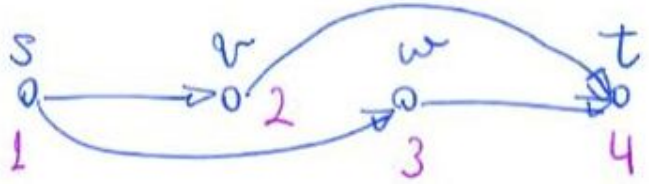


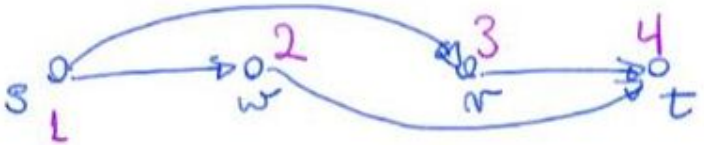


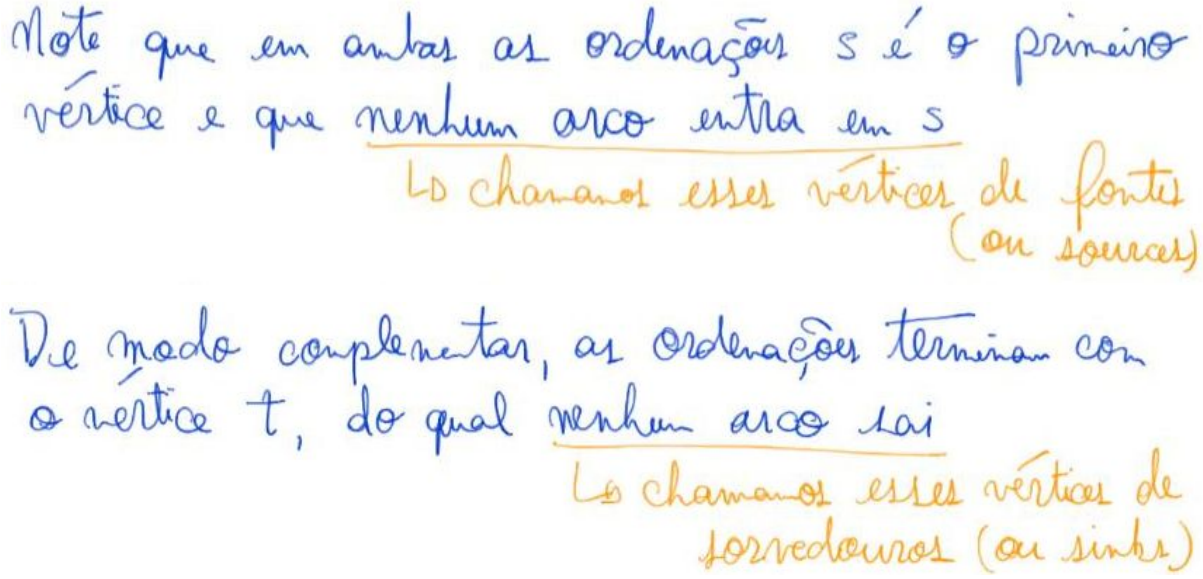
- e o que é uma ordenação topológica

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).







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

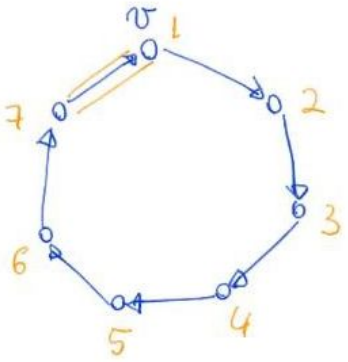
é

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

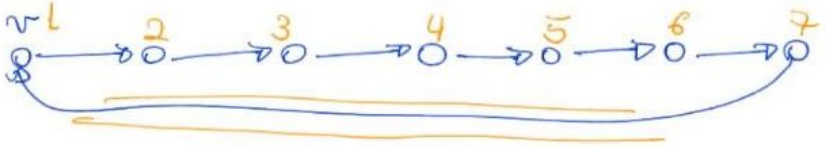
No caso em que o grafo orientado possui um ciclo,

- 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.

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



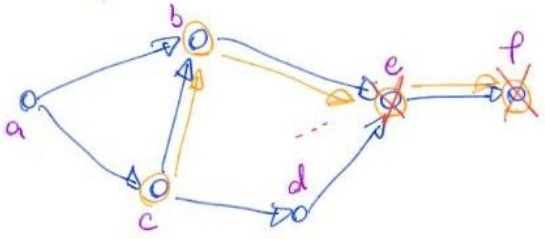
Observe que algum vértice do ciclo tem que ter o menor rótulo dentre os vértices do mesmo, e isso leva a uma aresta que viola a propriedade da ordenação topológica



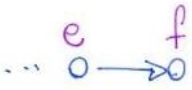
(-->) para provar a ida vamos fazer uma prova construtiva

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

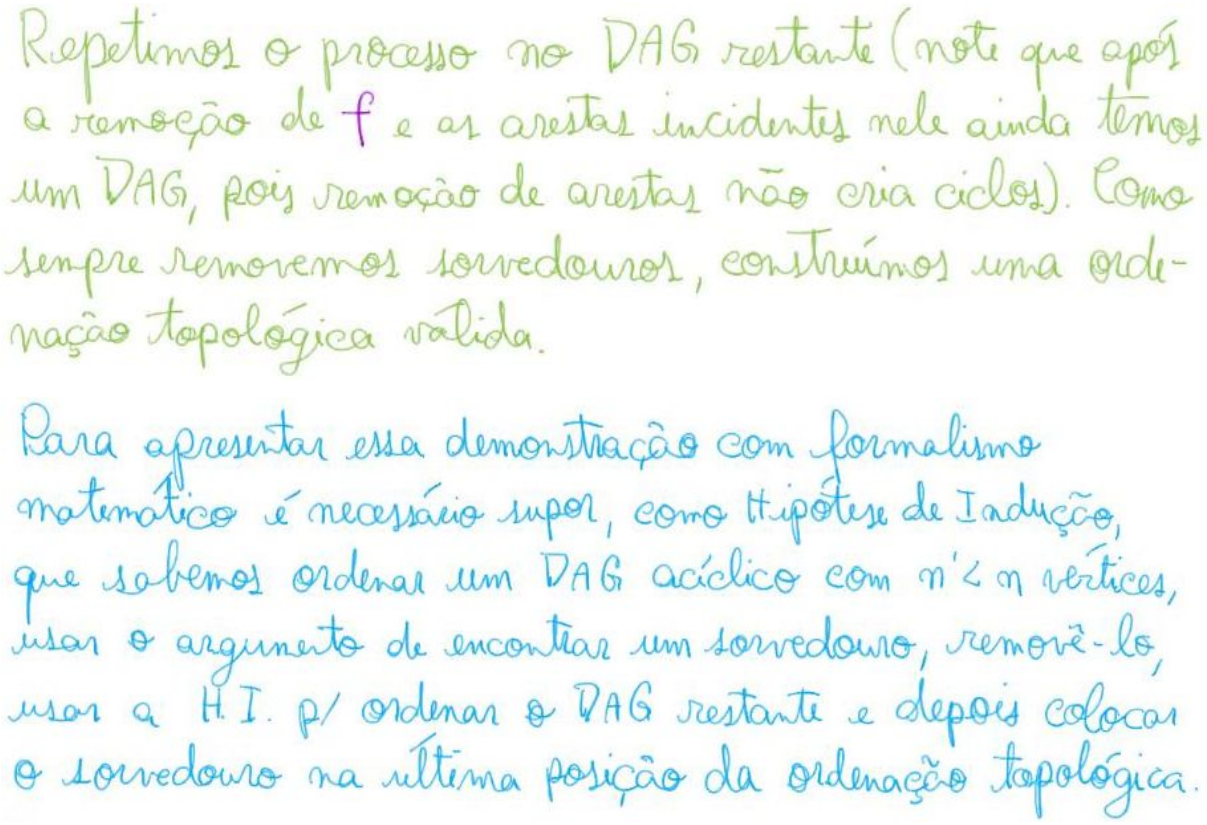
Exemplo p/ prova construtiva:



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 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.

```
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--
}
```

Código para identificar componentes

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

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

Código da busca em profundidade recursiva adaptado para ordenação topológica

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

Código da busca em profundidade iterativa adaptado para ordenação topológica

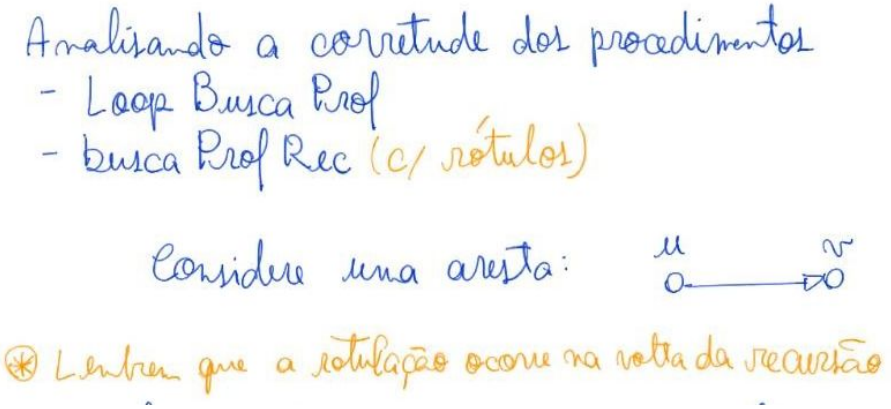
```
void buscaProfOrdTopoI(Grafo G, int origem, int *visitado,
        int *ordTopo, int *protulo_atual) {
    int v, w;
    Noh *p;
    // pilha implementada em vetor
    int *pilha;
    int topo = 0;
    pilha = malloc(G->m * sizeof(int));
    /* colocando vértice origem na pilha */
    pilha[topo++] = 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 (visitado[v] == 0) // se v não foi visitado
```

```
        {
            visitado[v] = 1;
            pilha[topo++] = v; // empilha o vértice pra saber quando
marcar o tempo de término
            /* para cada vizinho deste que ainda não foi visitado */
            p = G->A[v];
            while (p != NULL) {
                w = p->rotulo;
                if (visitado[w] == 0)
                    pilha[topo++] = w; // empilha o vizinho
                p = p->prox;
            }
        }
        else if (ordTopo[v] == -1) // se v ja foi visitado e sua
ordem topologica ainda nao foi atribuida
            ordTopo[v] = (*protulo_atual)--;
    }
}
```

Análise de eficiência:

- A eficiência 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,

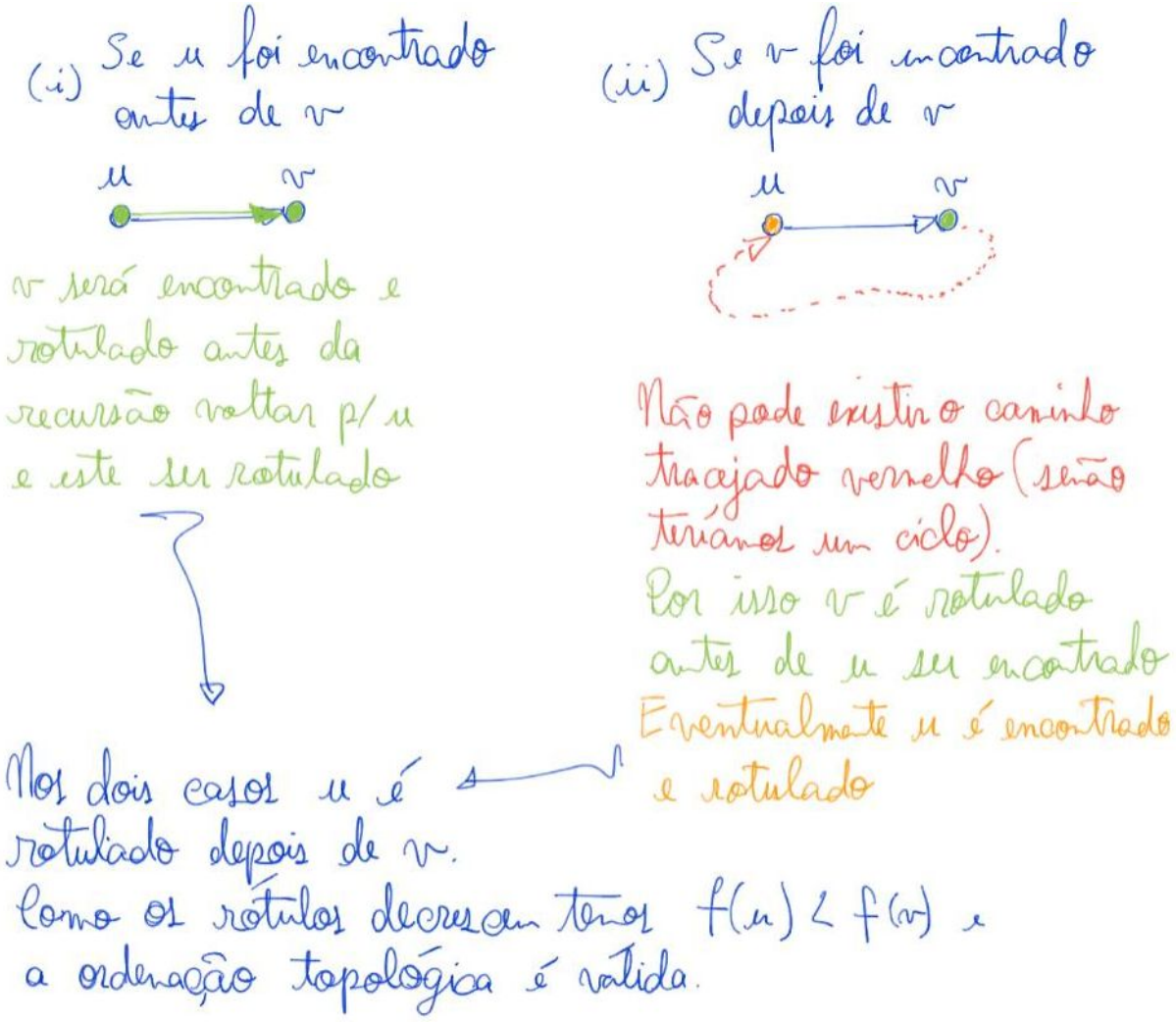
- vamos considerar um arco (u, v) qualquer
    - e queremos mostrar que f(u) < f(v).

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 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

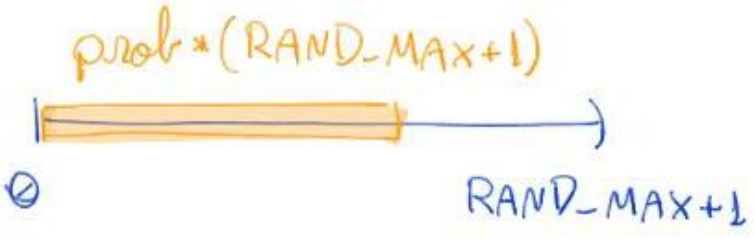
```
/* A função verticeAleatorio() devolve um vértice aleatório
do grafo G. Vamos supor que G->n <= RAND_MAX. */
int verticeAleatorio(Grafo G) {
    double r = (double)rand() / ((double)RAND_MAX + 1.0);
    return (int)(r * G->n);
}

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 orientação do DAG dada por perm
        if (perm[v] < perm[w])
            insereArcoGrafo(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.

```
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 orientação do DAG dada por perm
            if (perm[v] < perm[w])
                if (rand() < prob * (RAND_MAX + 1.0))
                    insereArcoNaoSeguraGrafo(G, v, w);
        }
    return G;
}
```

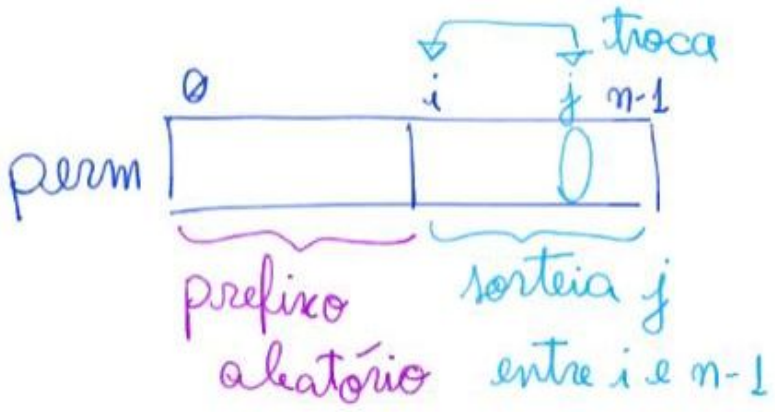


Para o caso geral, uma maneira eficiente de transformar uma permutação qualquer

- 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++) {
        j = i + uniformeAleat(n - i);
        troca(&perm[i], &perm[j]);
    }
    return perm;
}
```

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)!).
- Ao final das iterações v[0 .. n - 1] é uma permutação
  - escolhida com probabilidade 1 / n!,
- sendo que n! é o número total de permutações com n elementos.

Eficiência de tempo: O(n).

Eficiência de espaço: O(1).

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.