

## AED2 - Aula 26

### Caminhos mínimos ponderados em DAGs e em grafos sem custos negativos

Hoje vamos falar do problema do caminho mínimo ponderado em grafos.

Neste problema recebemos como entrada:

- Um grafo  $G = (V, E)$ ,
- com custo  $c(e)$  em cada aresta e em  $E$

```
typedef struct noh Noh;  
struct noh {  
    int rotulo;  
    int custo;  
    Noh *prox;  
};
```

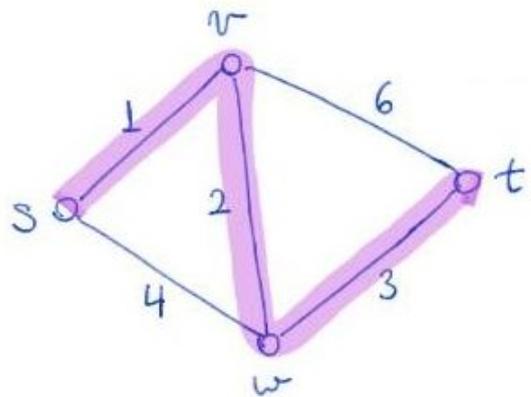
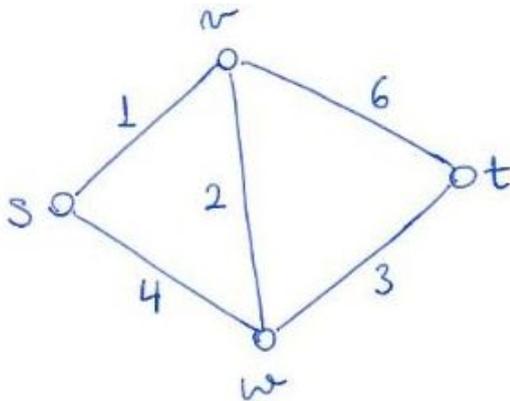
- e um vértice origem  $s$ .

E queremos encontrar:

- O valor do caminho mínimo de  $s$  até cada vértice  $v$  em  $V$ ,
  - i.e., a distância de  $s$  a  $v$ .
- Também gostaríamos que esses caminhos fossem devolvidos.

Exemplo de grafo com custos nas arestas:

- Caminho mínimo de  $s$  até  $t$ .



Mas, busca em largura não encontra caminhos mínimos

- ao explorar o grafo em camadas centradas em  $s$ ?
  - Por que voltamos a esse problema? O que mudou?

Resp.: Busca em largura só funciona em grafos não ponderados,

- ou seja, naqueles em que toda aresta tem custo uniforme.
- Note que, se aplicarmos busca em largura no exemplo anterior,

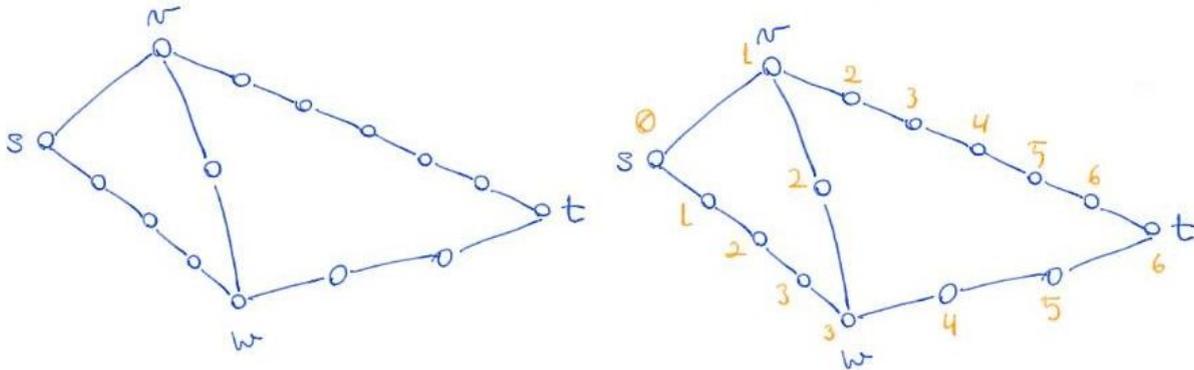
- o ela não nos devolve corretamente o caminho mínimo de s a t.

Para contornar este problema, observe que podemos converter

- uma entrada do problema de caminhos mínimos ponderados
  - o em uma entrada não ponderada. Como?

Resp.: Substituindo cada aresta e com custo  $c(e)$

- por um caminho com  $c(e)$  arestas e  $c(e) - 1$  novos vértices.
  - o O seguinte exemplo mostra esse procedimento

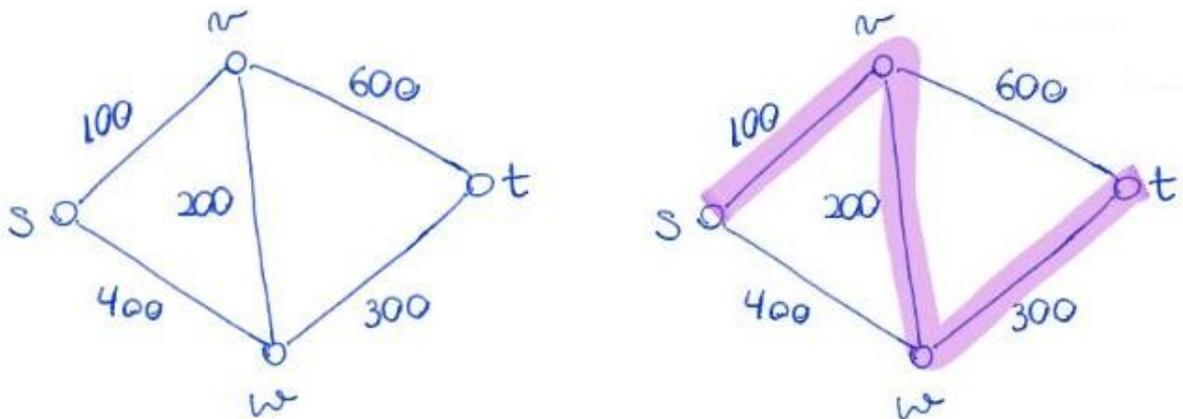


- Se utilizarmos o algoritmo de busca em largura neste grafo
  - o resolvemos o problema de caminhos mínimos ponderado.

No entanto, essa solução pode não ser eficiente. Por que?

Resp.: Porque o grafo modificado tem o número de vértices e arestas

- aumentado em proporção ao custo  $c( )$  das arestas.
  - o O seguinte exemplo evidencia isto



- Observe que, o grafo modificado pode crescer exponencialmente,
  - o pois um custo que é representado usando  $k$  de bits
    - pode implicar da ordem de  $2^k$  de novos vértices.

Antes de estudar um algoritmo para o problema

- de caminhos mínimos ponderados em grafos gerais,

- vamos estudar este problema em um tipo de grafo específico.
- Este caso particular do problema possui uma solução eficiente,
  - que se baseia na solução para um problema
    - que estudamos recentemente.

### Caminhos mínimos ponderados em DAGs

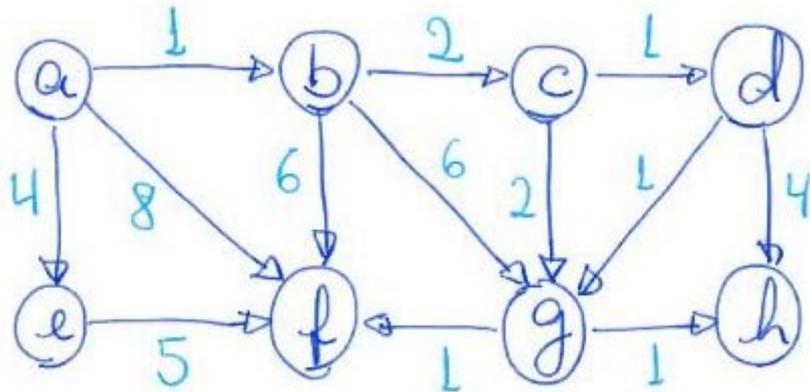
Se  $s$  for nosso vértice origem, note que,

- se considerarmos todos os caminhos a partir de  $s$ 
  - que chegam em um vértice  $v$ ,
- e escolhermos o menor destes caminhos,
  - teremos o caminho de custo mínimo de  $s$  a  $v$ ,
  - e o valor deste caminho é a distância desejada.

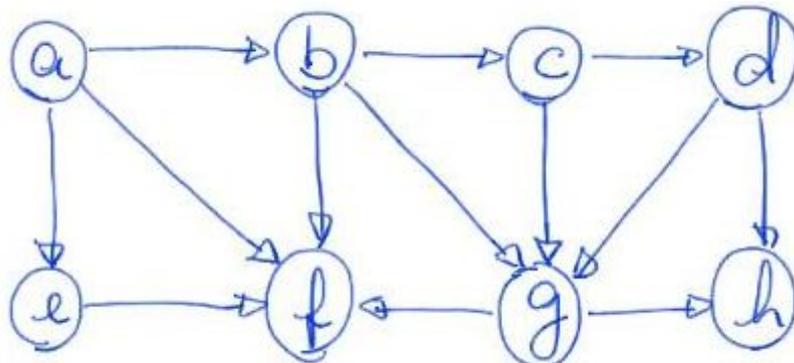
A ideia central do algoritmo é encontrar uma ordem

- para visitar os vértices do grafo dirigido acíclico (DAG),
  - que nos permita encontrar eficientemente
    - todos os caminhos que chegam em cada vértice.
- Como veremos a seguir, essa ordem é a ordenação topológica.

Considere o seguinte grafo dirigido acíclico com custos nos arcos.



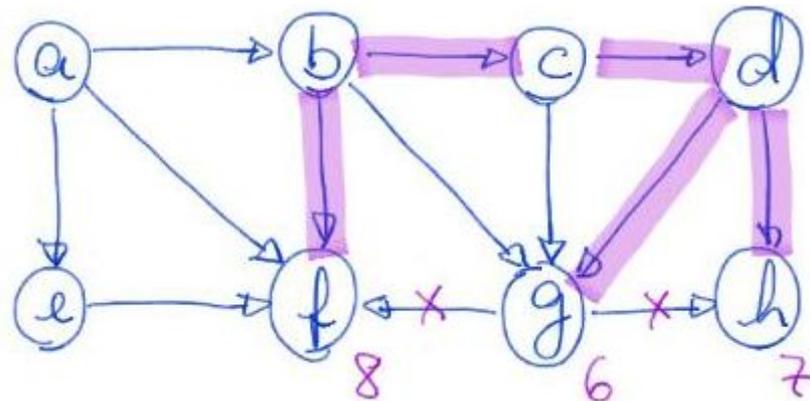
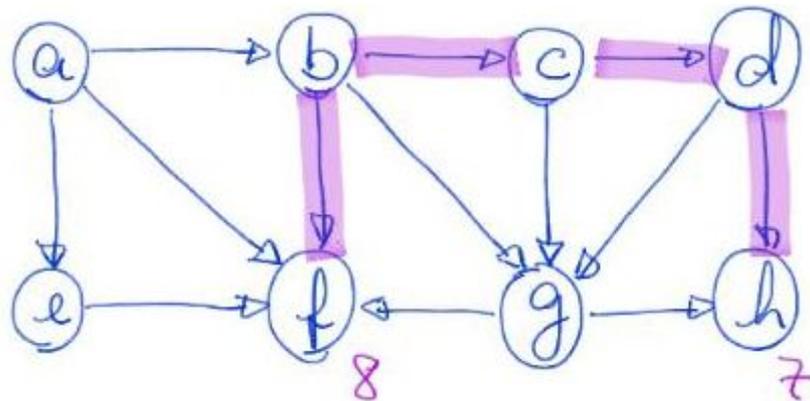
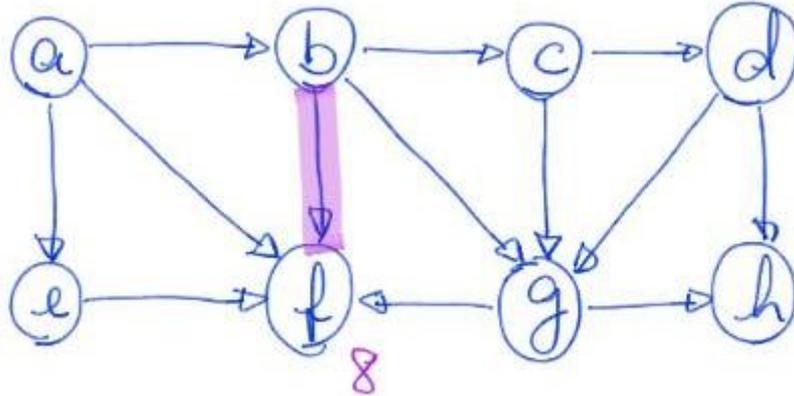
Vamos encontrar uma ordenação topológica para este DAG.



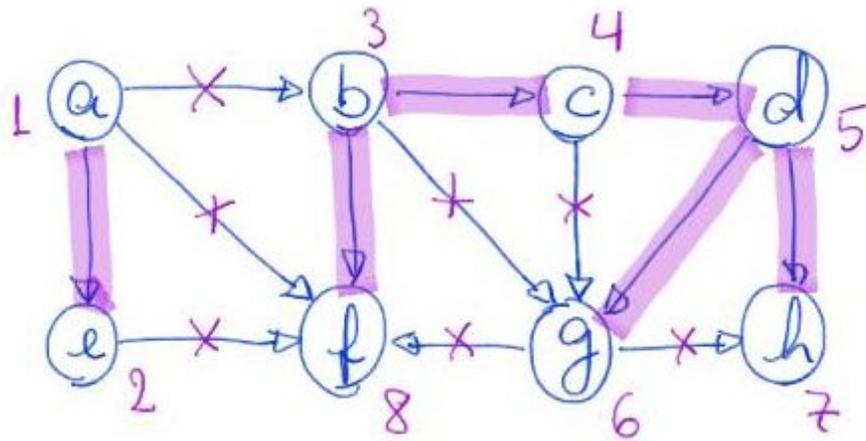
- Desconsideramos os custos nos arcos, pois eles não afetam a ordenação.

Para realizar a ordenação realizamos um loop da busca em profundidade (DFS),

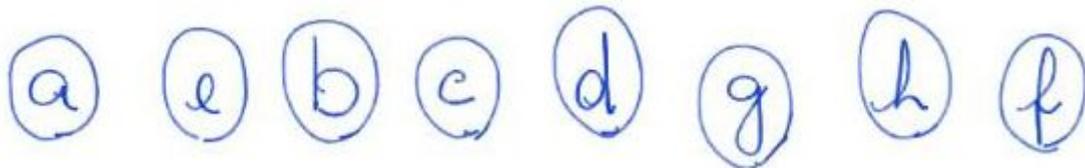
- i.e., invocamos a DFS a partir de cada vértice não visitado,
  - e esta rotula cada vértice que for “finalizado” em ordem decrescente,
- lembrando que um vértice é finalizado
  - quando todos os caminhos a partir dele já foram explorados.
- 1ª DFS começa em b



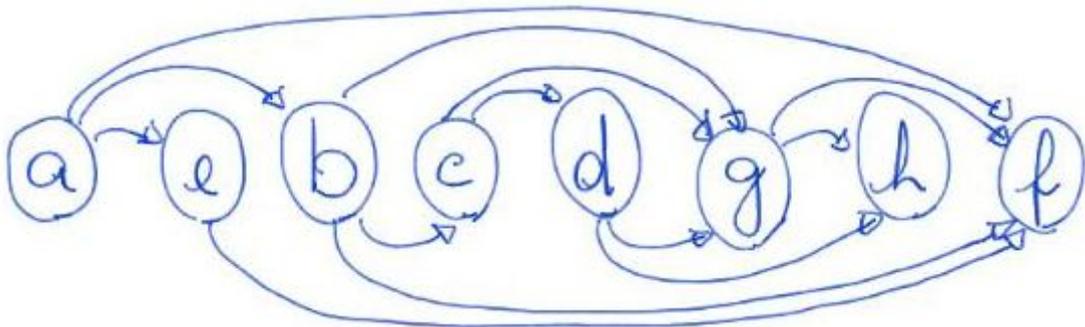




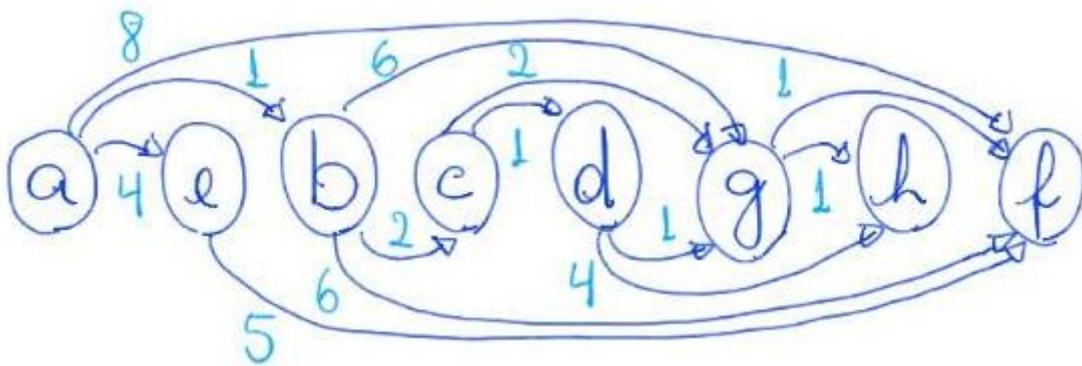
Linearizando o DAG segundo a ordem topológica encontrada



- Note que, como esperado, todos os arcos vão da esquerda para a direita.



- Agora voltamos a considerar os custos, já que vamos calcular as distâncias.

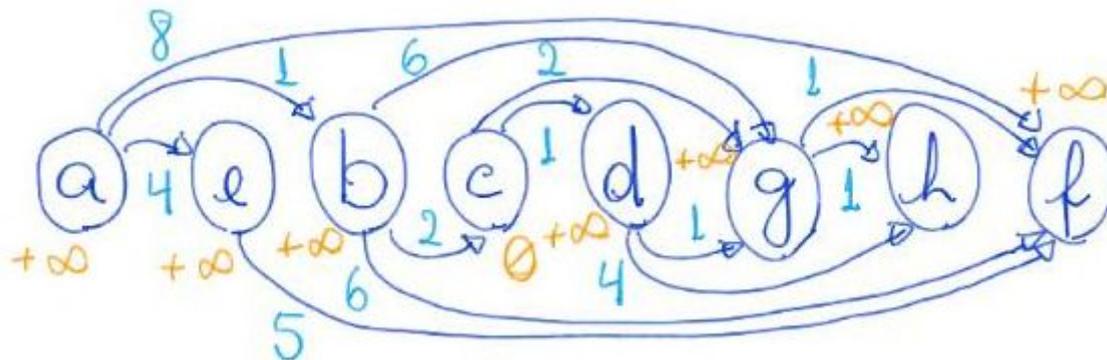


Marcamos cada vértice  $v$  em  $V$  com:

- $dist[v] = +\infty$
- $pred[v] = NULL$

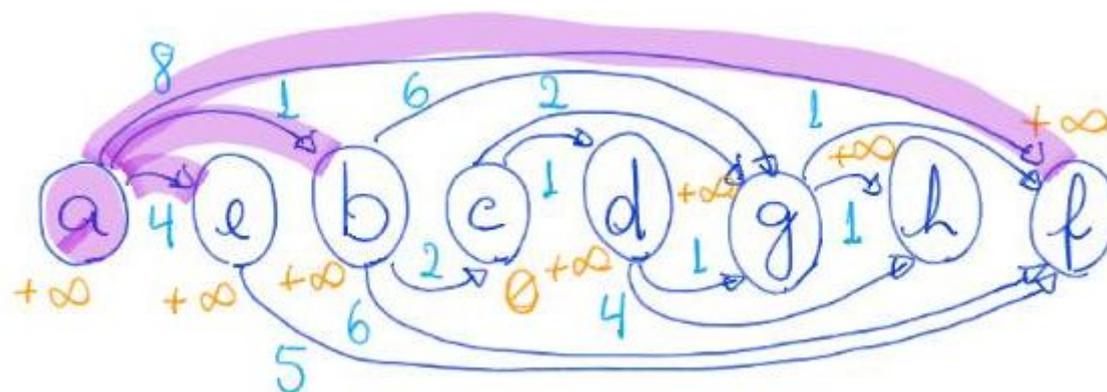
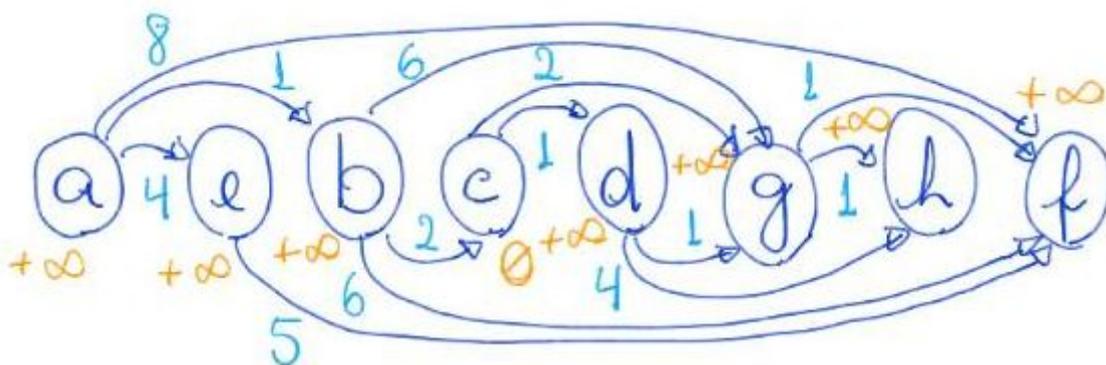
A exceção é o vértice origem  $s$ , que começa com distância 0, i.e.,  $dist[s] = 0$ .

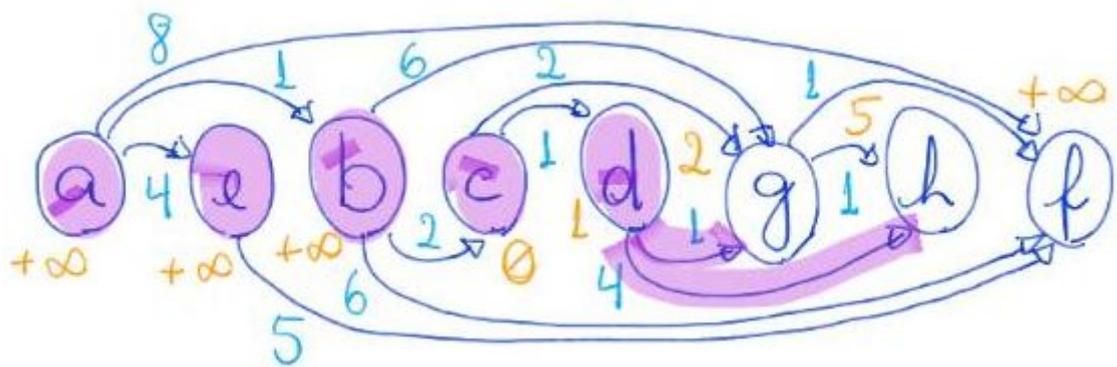
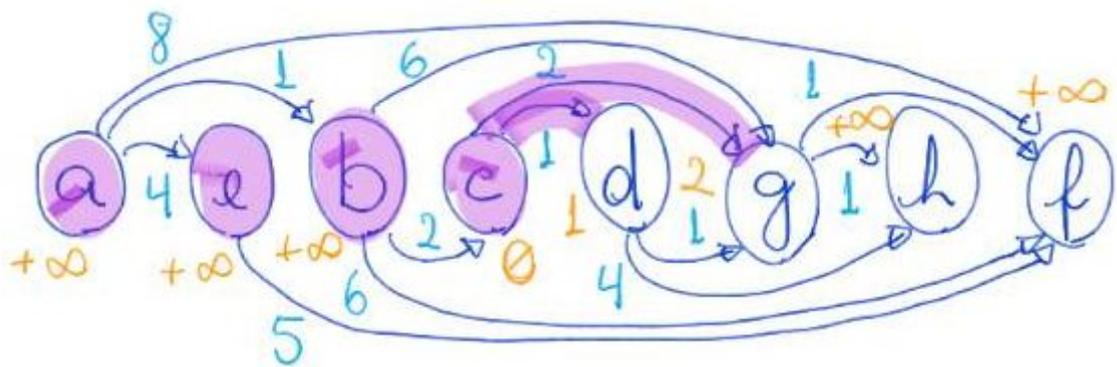
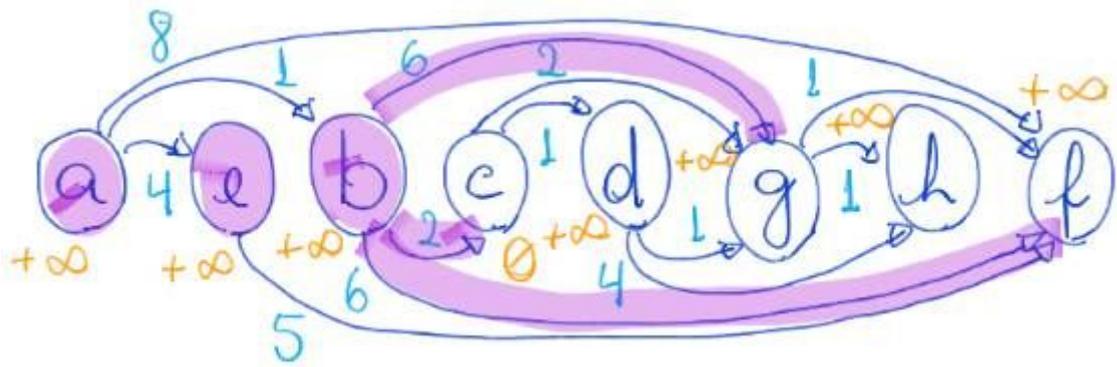
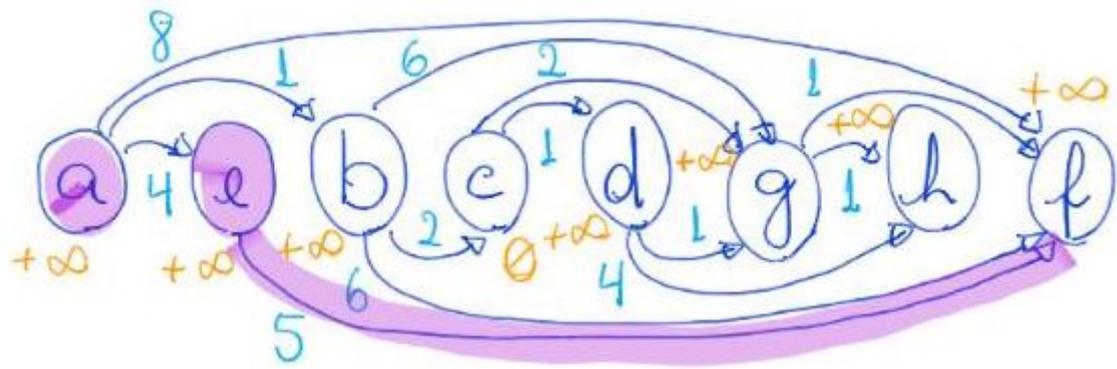
No nosso exemplo, seja  $c$  o vértice origem.

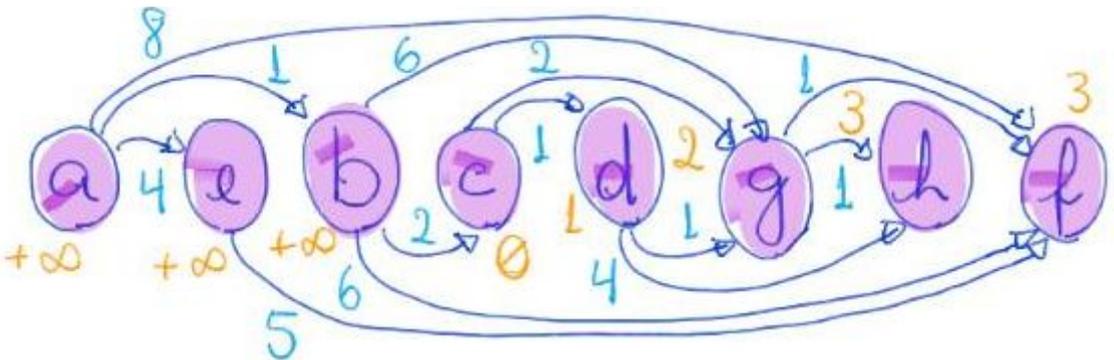
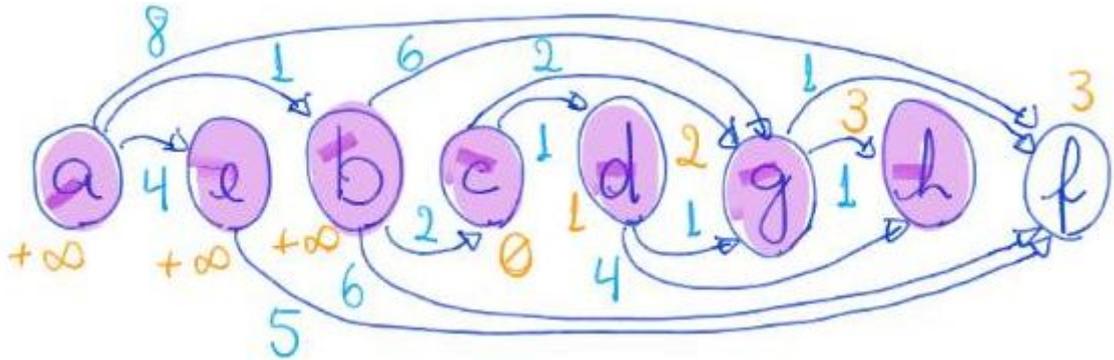
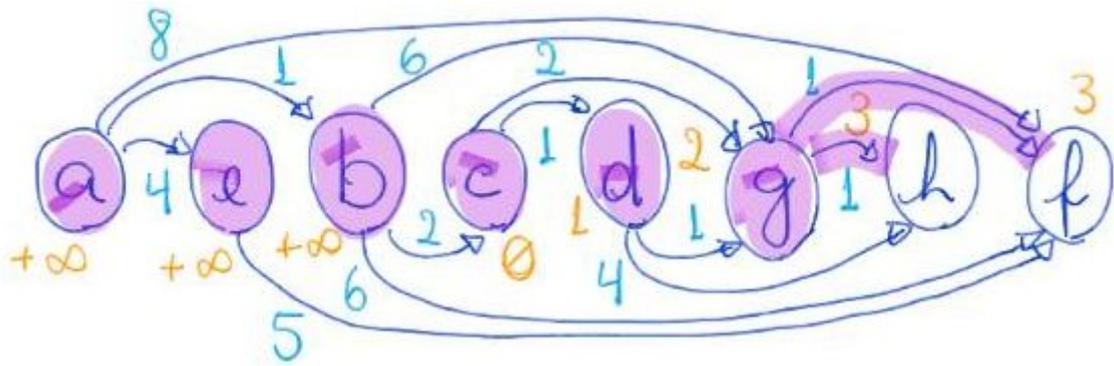


Agora, visitamos os vértices da esquerda para a direita,

- i.e., seguindo a ordenação topológica,
  - e em cada vértice visitado consideramos os arcos que saem dele.
- Para cada arco  $(u, v)$  considerado
  - se  $\text{dist}[u] + c(u, v) < \text{dist}[v]$  atualizamos
    - $\text{dist}[v] = \text{dist}[u] + c(u, v)$
    - $\text{pred}[v] = u$





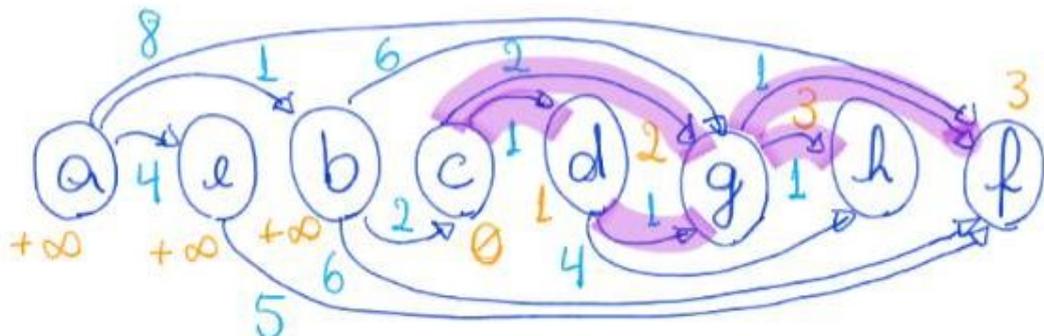


Note que, na iteração em que visitamos um vértice,

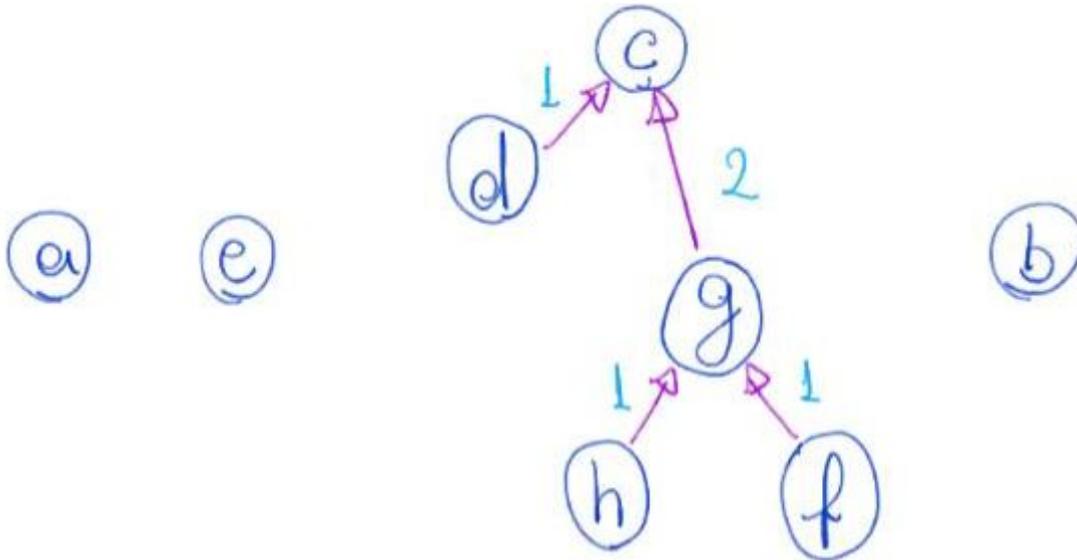
- todos os caminhos que chegam nele já foram considerados.
- Por isso, a distância da origem até o vértice é calculada corretamente.

Observe que, cada “apontador” no vetor pred corresponde

- ao antecessor de um vértice em seu caminho mínimo.
- Por isso, pred permite reconstruir os caminhos mínimos
  - da origem até cada vértice alcançável.



- Note que, esta coleção de caminhos
  - forma uma árvore de caminhos mínimos enraizada na origem.



Pseudocódigo do algoritmo:

```

distanciasDAG(DAG G=(V,E), custos c, vértice s) {
  para v ∈ V
    dist[v] = +∞
    pred[v] = NULL
  dist[s] = 0

  ordem = ordenaçãoTopológica(G);
  para cada v ∈ V, seguindo ordem
    para cada arco (v, w)
      se dist[w] > dist[v] + c(v, w)
        dist[w] = dist[v] + c(v, w)
        pred[w] = v
}

```

Duas curiosidades são que:

- Este algoritmo funciona mesmo que os arcos tenham custos negativos,
  - pois em um DAG não é possível formar circuitos de custo negativo.
- Ele também pode ser adaptado para
  - o problema de encontrar caminhos de custo máximo,
    - que em geral é bem mais difícil,
      - justamente por ter de lidar com circuitos.

Eficiência:

- $O(n + m)$ , sendo  $n$  o número de vértices e  $m$  o número de arcos.
  - Resultado deriva da eficiência da busca em profundidade (DFS)

- e da passada linear pelos vértices,
  - em que cada arco é considerado uma única vez.
- Note que, assim como no algoritmo de Kosaraju
  - para encontrar componentes fortemente conexos
- é importante que a ordenação topológica devolva a ordem
  - em um vetor indexado pela posição de cada vértice
    - e cujos conteúdos são os rótulos dos vértices.

Código do algoritmo para cálculo de distâncias em um DAG:

```
void distanciasDAG(Grafo G, int origem, int *dist, int *pred) {
    int i, *ordTopo;
    int v, w, custo;
    Noh *p;
    for (i = 0; i < G->n; i++) {
        dist[i] = __INT_MAX__;
        pred[i] = -1;
    }
    dist[origem] = 0;
    ordTopo = malloc((G->n + 1) * sizeof(int));
    ordenacaoTopologica(G, ordTopo);
    for (i = 1; i <= G->n; i++) {
        v = ordTopo[i];
        p = G->A[v];
        while (p != NULL) {
            w = p->rotulo;
            // aumento da estrutura do grafo para armazenar custos dos arcos
            custo = p->custo;
            if (dist[w] > dist[v] + custo) {
                dist[w] = dist[v] + custo;
                pred[w] = v;
            }
            p = p->prox;
        }
    }
    free(ordTopo);
}
```

Código da ordenação topológica com pequenas modificações para esta aplicação:

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

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;
    }
    // ordenação armazenada em vetor indexado por posição
    ordTopo[*protulo_atual] = v;
    (*protulo_atual)--;
}
```

**Algoritmo de Dijkstra**

Agora retornamos ao problema

- de encontrar caminhos mínimos ponderados em grafos gerais,
  - e para tratar este problema,
- vamos estudar um dos maiores clássicos da computação,
  - o algoritmo para caminhos mínimos de Dijkstra.

Primeiro, vamos relembrar

- o algoritmo de busca em largura para cálculo de distâncias,
  - que é generalizado pelo algoritmo de Dijkstra.

```
distancias(grafo G=(V,E), vértice s) {
  para v \in V
    dist[v] = +\inf
    pred[v] = NULL
  dist[s] = 0
  seja Q uma fila inicializada com o vértice s
  enquanto Q != \empty
    remova um vértice v do início de Q
    para cada aresta (v, w)
      se w não foi encontrado, i.e, dist[w] == +\inf
        dist[w] = dist[v] + 1
        pred[w] = v
        insira w no final de Q
}
```

A seguir, para simplificar, vamos supor que todos os vértices do grafo

- são alcançáveis a partir do vértice s.
- Se esse não for o caso, podemos focar nos vértices alcançáveis
  - realizando uma busca a partir de s antes,
    - ou modificando levemente o algoritmo de Dijkstra.

```
Dijkstra(grafo G=(V,E), custos c, vértice s) {
  para todo v \in V
    dist[v] = +\inf
    pred[v] = NULL
  dist[s] = 0
  R = {} // conjunto de vértices já visitados
  enquanto R != V
    // escolha gulosa do algoritmo de Dijkstra
    pegar o vértice v em V \ R com menor valor de dist[]
    adicione v a R
```

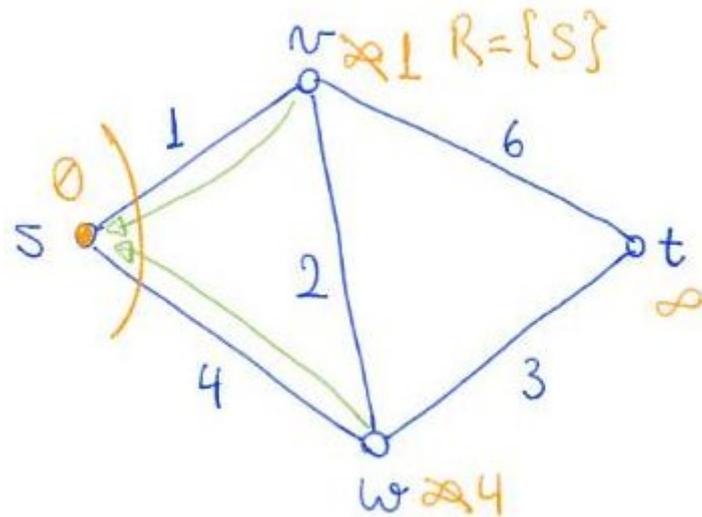
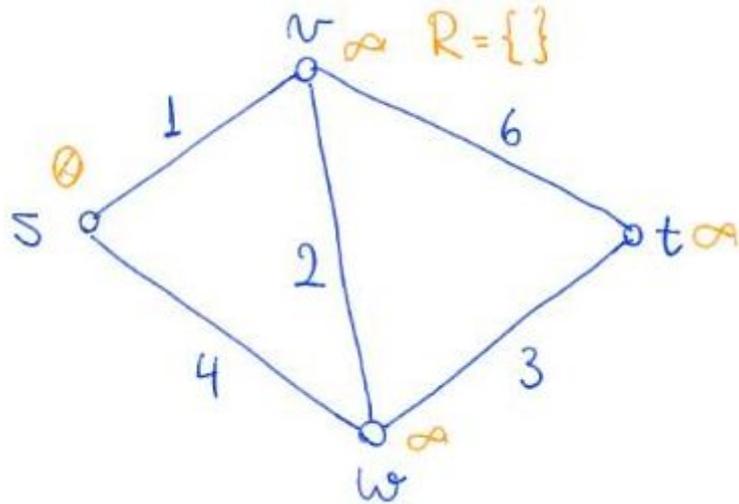
para toda aresta  $(v, w)$  com  $w$  em  $V \setminus R$   
 se  $\text{dist}[w] > \text{dist}[v] + c(v, w)$   
 $\text{dist}[w] = \text{dist}[v] + c(v, w)$   
 $\text{pred}[w] = v$

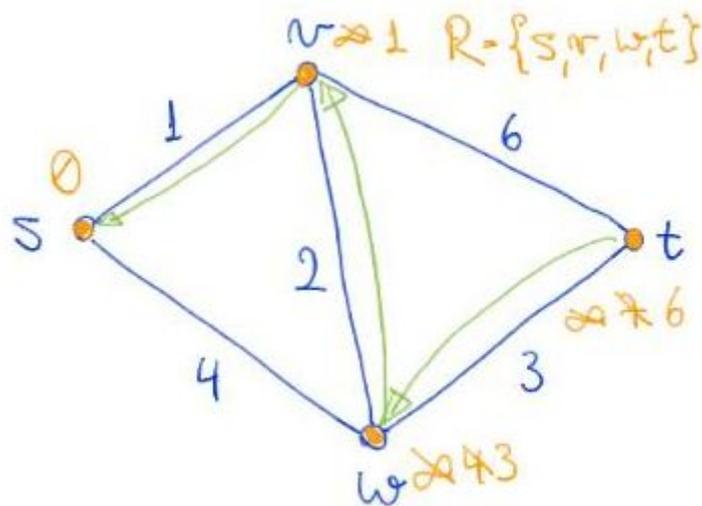
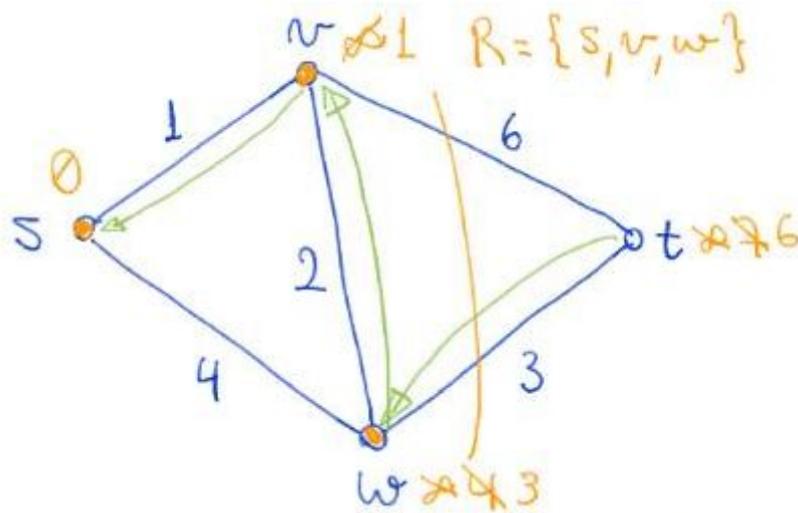
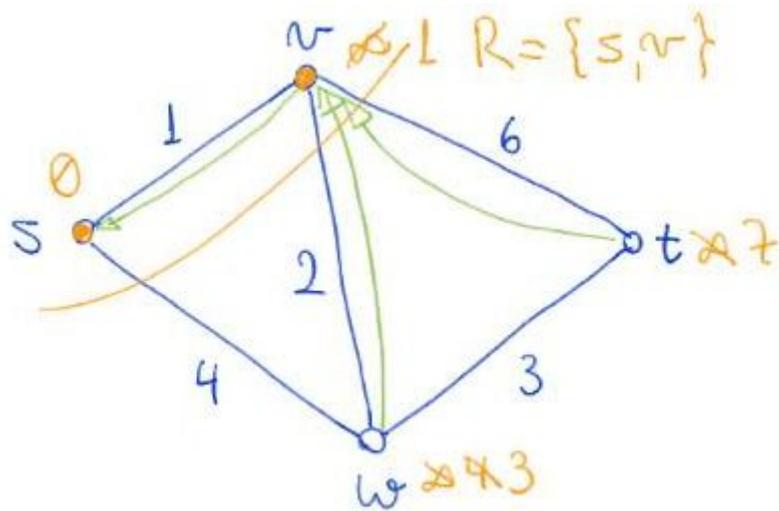
}

Notaram a semelhança com o algoritmo

- para cálculo de distâncias não ponderadas,
  - baseado na busca em largura?
- E com o algoritmo para cálculo de distâncias em DAGs,
  - baseado na busca em profundidade?

Exemplo de funcionamento do algoritmo:



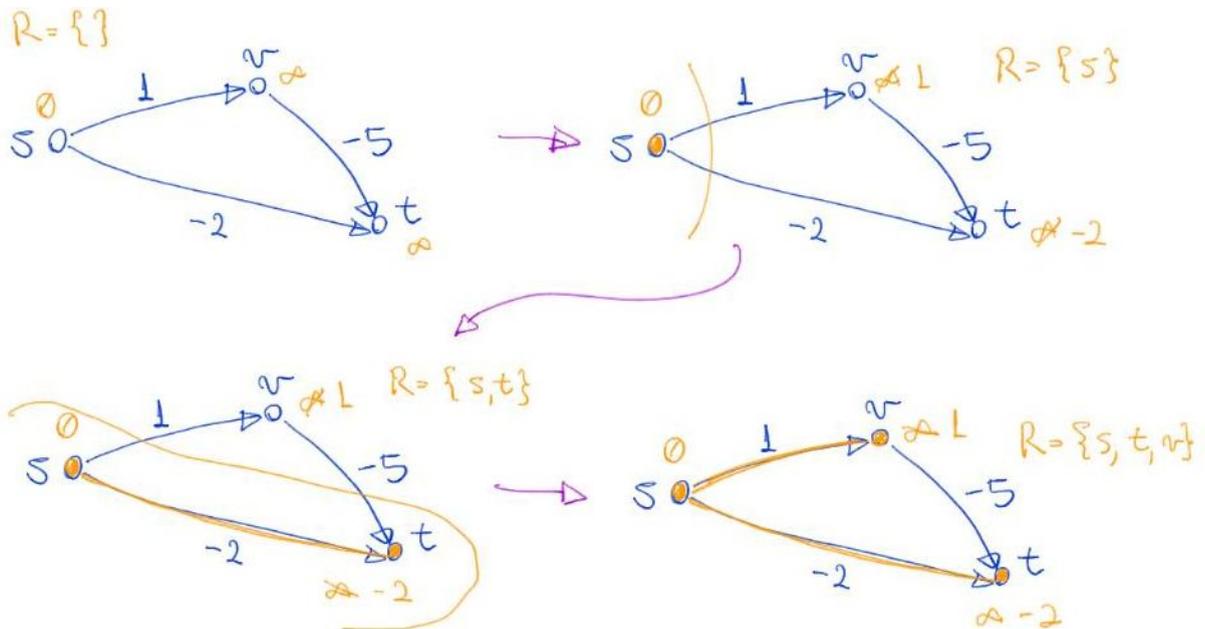


Antes de provarmos a corretude deste algoritmo,

- de tratarmos dos detalhes de implementação e da eficiência do mesmo,
  - vamos analisar suas limitações,
    - a fim de compreendê-lo melhor.

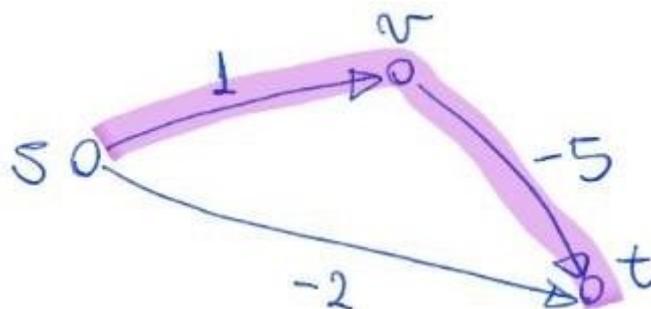
Em particular, este algoritmo pode não devolver a solução correta

- quando as arestas apresentam custo negativo.



Isso ocorre porque a escolha gulosa pode definir

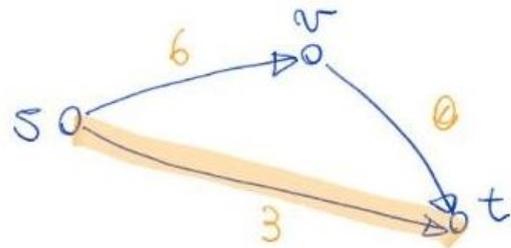
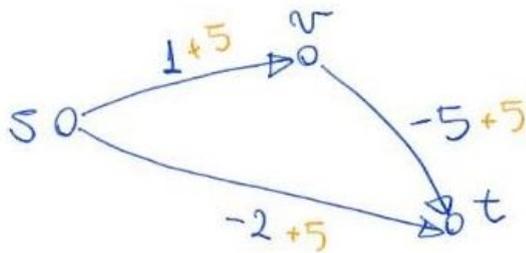
- uma certa distância para um vértice,
  - sem considerar um caminho menor,
- caso tal caminho “tenha” um custo maior na sua parte inicial,
  - ainda que depois este custo seja reduzido
    - por conta de arestas de custo negativo.
- No nosso exemplo, o caminho  $s \rightarrow v \rightarrow t$ , que tem custo -4,



- nunca é considerado pelo algoritmo.

Poderíamos pensar em reduzir o problema com arestas negativas

- para o problema sem essas arestas.
- Uma tentativa seria somar o valor absoluto da aresta mais negativa
  - no comprimento de todas as arestas.
- Isso certamente faria com que o grafo não tivesse mais arestas negativas.



No entanto, o caminho mínimo entre os vértices seria alterado, pois

- caminhos com números diferentes de arestas
  - seriam afetados de modo distinto.

Como regra geral, quando tentamos eliminar custos negativos,

- se as soluções do seu problema tem número variado de objetos
  - (no caso de caminhos mínimos os objetos são as arestas)
- então somar um mesmo valor no custo de cada objeto
  - afetará mais o custo de soluções com maior cardinalidade,
    - e menos o custo daquelas com menor cardinalidade.
- Assim, não há garantia de que
  - a ordem relativa dos custos das soluções será preservado.
- Por isso, esse procedimento não é recomendado.

Por outro lado, se todas as soluções do seu problema

- tem a mesma cardinalidade,
  - i.e., o mesmo número de objetos,
- então somar um mesmo valor no custo de cada objeto
  - afetará homoganeamente o custo de todas as soluções.
- Neste caso, como a ordem relativa dos custos das soluções é preservada,
  - o procedimento é seguro.
- Este é o caso, por exemplo, do problema da árvore geradora mínima,
  - que é um problema central em otimização combinatória e grafos.