

MC-202  
Algoritmos em Grafos

Lehilton Pedrosa

Universidade Estadual de Campinas

Segundo semestre de 2022

## Corte de material

- uma fábrica utiliza precisa cortar papelão retangular



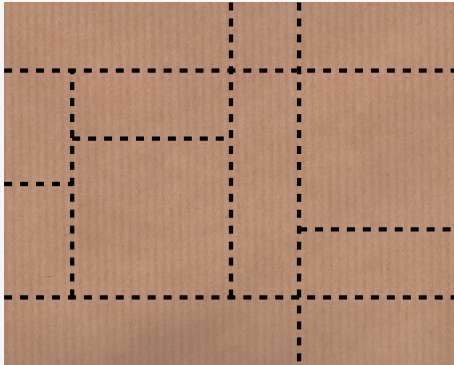
## Corte de material

- uma fábrica utiliza precisa cortar papelão retangular
- ela utilizada uma grande guilhotina (maior que o papelão)



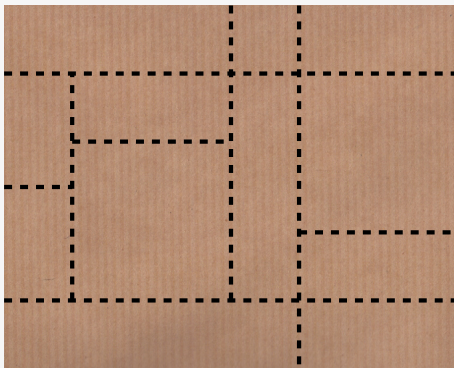
## Corte de material

- uma fábrica utiliza precisa cortar papelão retangular
- ela utilizada uma grande guilhotina (maior que o papelão)
- existe um padrão de corte para evitar desperdício



## Corte de material

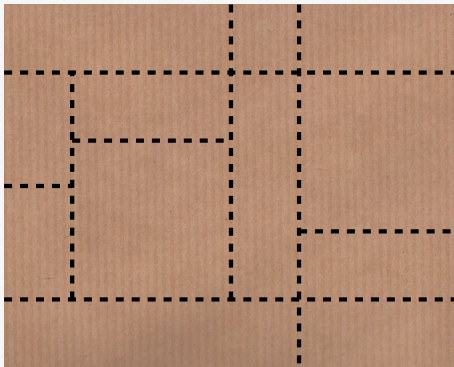
- uma fábrica utiliza precisa cortar papelão retangular
- ela utilizada uma grande guilhotina (maior que o papelão)
- existe um padrão de corte para evitar desperdício



1. o corte pode ser feito pela guilhotina?

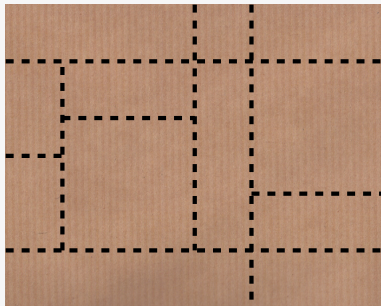
## Corte de material

- uma fábrica utiliza precisa cortar papelão retangular
- ela utilizada uma grande guilhotina (maior que o papelão)
- existe um padrão de corte para evitar desperdício

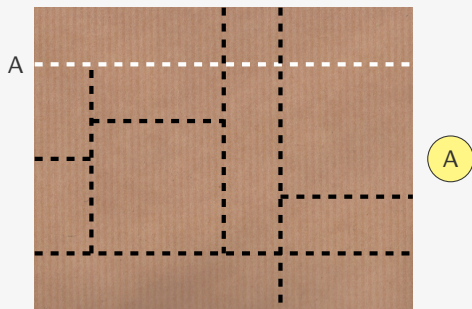


1. o corte pode ser feito pela guilhotina?
2. em que ordem eles devem ser feitos?

# Modelando como um grafo



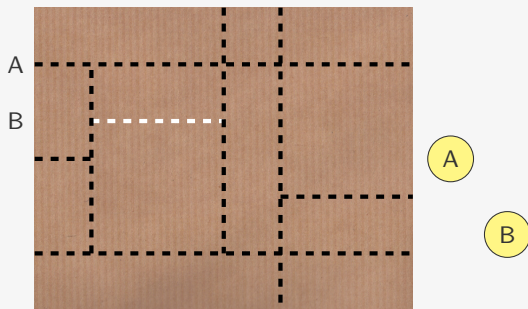
# Modelando como um grafo



- **vértices:** consideramos cada corte da guilhotina

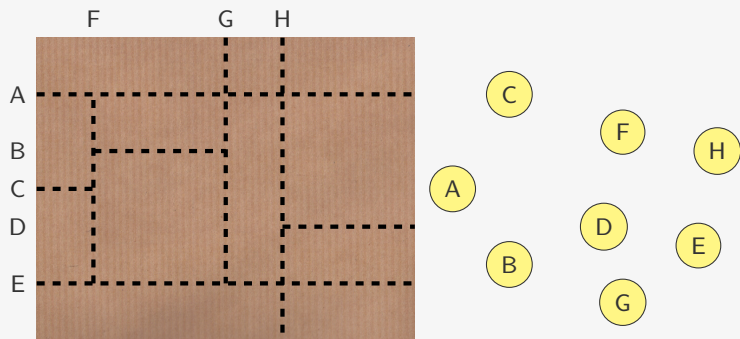


## Modelando como um grafo



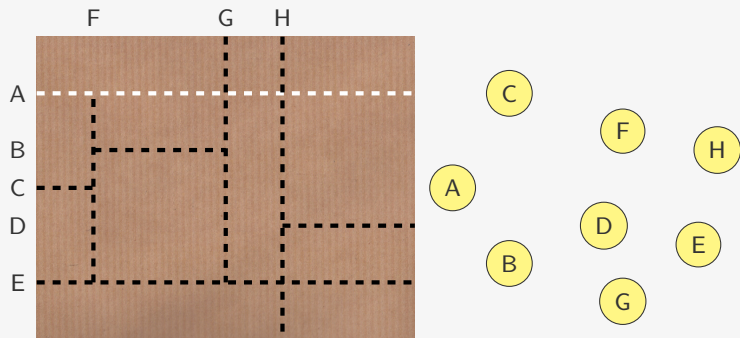
- **vértices:** consideramos cada corte da guilhotina

## Modelando como um grafo



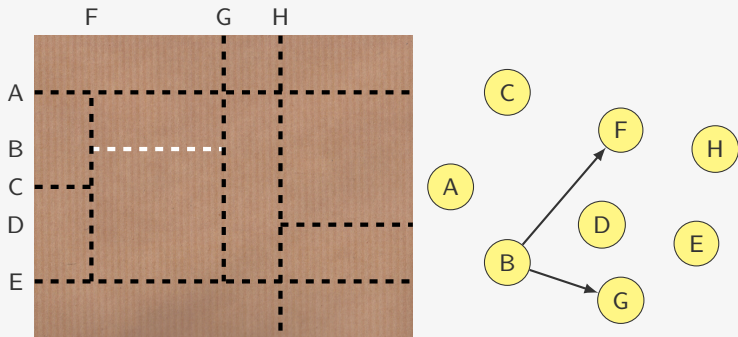
- **vértices:** consideramos cada corte da guilhotina

## Modelando como um grafo



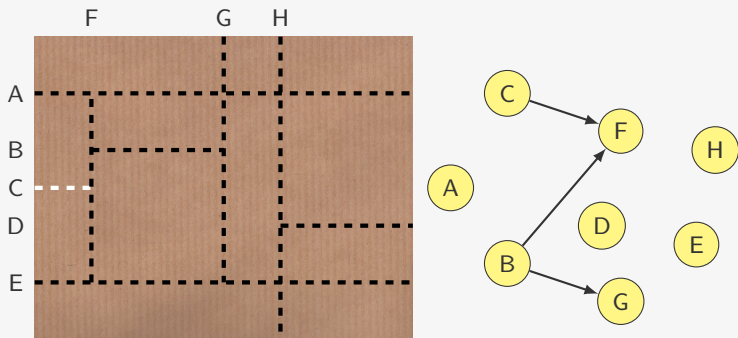
- **vértices:** consideramos cada corte da guilhotina
- **arestas:** se um corte depende de um corte anterior

# Modelando como um grafo



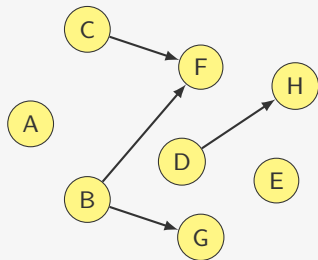
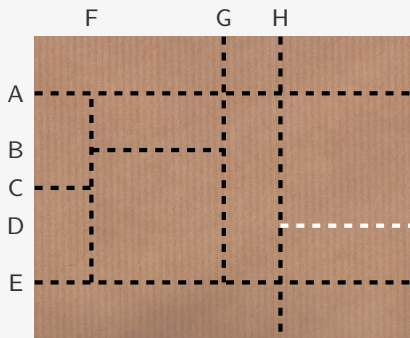
- **vértices:** consideramos cada corte da guilhotina
- **arestas:** se um corte depende de um corte anterior

## Modelando como um grafo



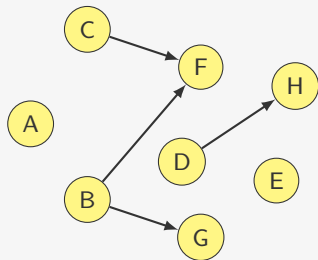
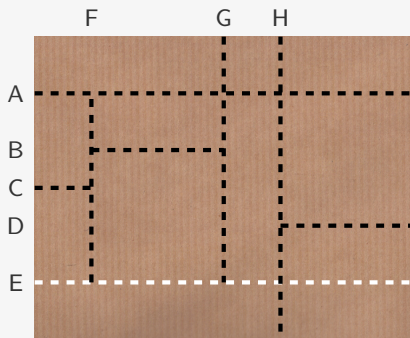
- **vértices:** consideramos cada corte da guilhotina
- **arestas:** se um corte depende de um corte anterior

# Modelando como um grafo



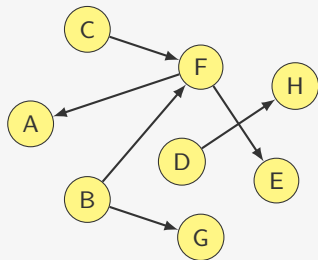
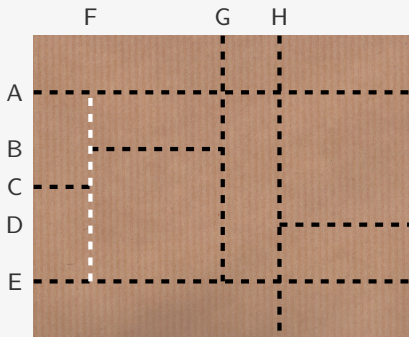
- **vértices:** consideramos cada corte da guilhotina
- **arestas:** se um corte depende de um corte anterior

## Modelando como um grafo



- **vértices:** consideramos cada corte da guilhotina
- **arestas:** se um corte depende de um corte anterior

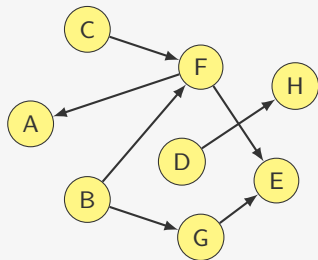
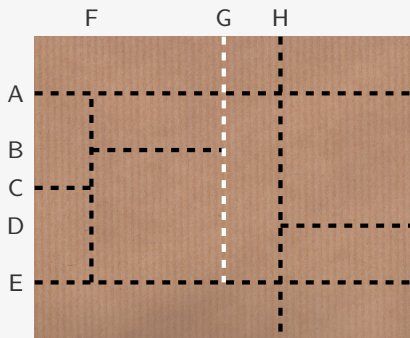
# Modelando como um grafo



- **vértices:** consideramos cada corte da guilhotina
- **arestas:** se um corte depende de um corte anterior

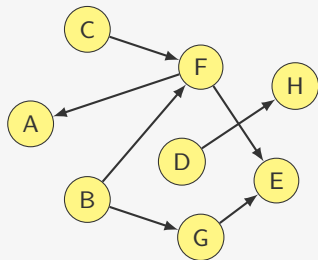
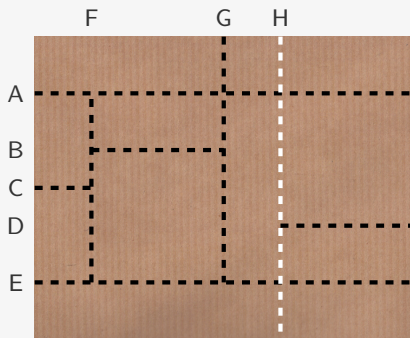


# Modelando como um grafo



- **vértices:** consideramos cada corte da guilhotina
- **arestas:** se um corte depende de um corte anterior

# Modelando como um grafo



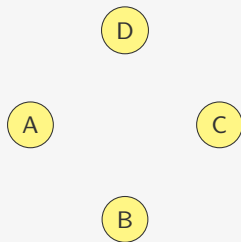
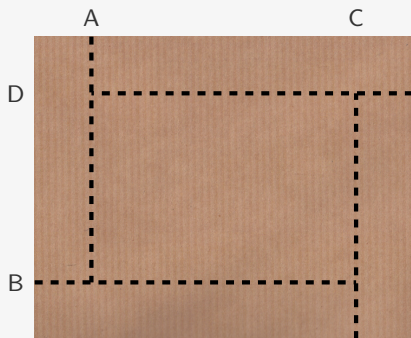
- **vértices:** consideramos cada corte da guilhotina
- **arestas:** se um corte depende de um corte anterior

## Viabilidade do padrão

Sempre é possível cortar um padrão com a guilhotina?

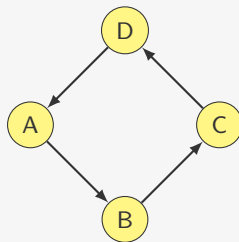
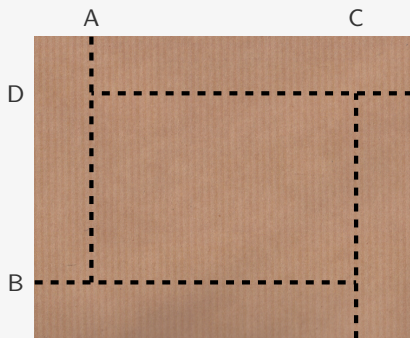
# Viabilidade do padrão

Sempre é possível cortar um padrão com a guilhotina?



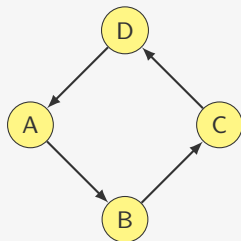
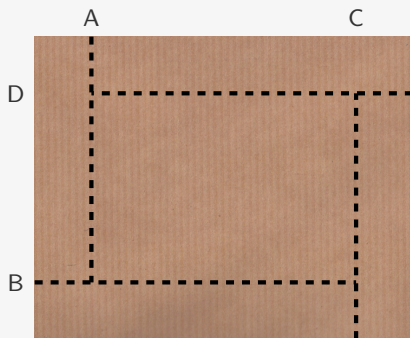
# Viabilidade do padrão

Sempre é possível cortar um padrão com a guilhotina?



# Viabilidade do padrão

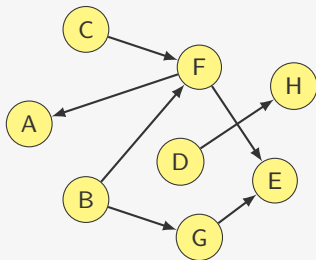
Sempre é possível cortar um padrão com a guilhotina?



**Conclusão:** Um padrão de corte é viável se, e somente se, o grafo dos cortes for **acíclico**.

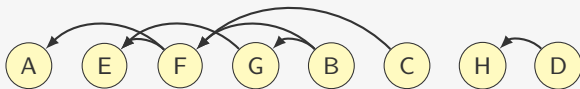
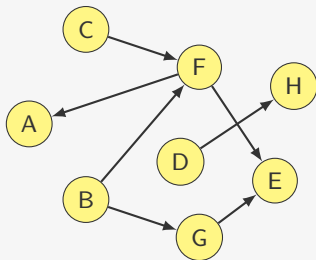
# Ordenação topológica

Uma **ordenação topológica** de um grafo **acíclico** é uma ordenação dos vértices cujas arestas estão na **mesma direção**.



## Ordenação topológica

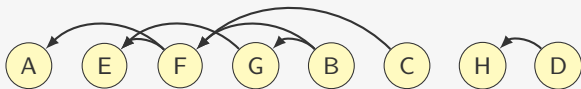
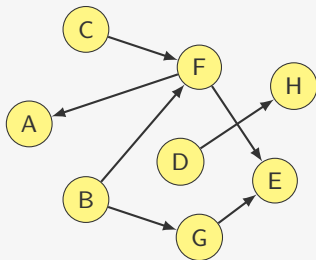
Uma **ordenação topológica** de um grafo **acíclico** é uma ordenação dos vértices cujas arestas estão na **mesma direção**.





## Ordenação topológica

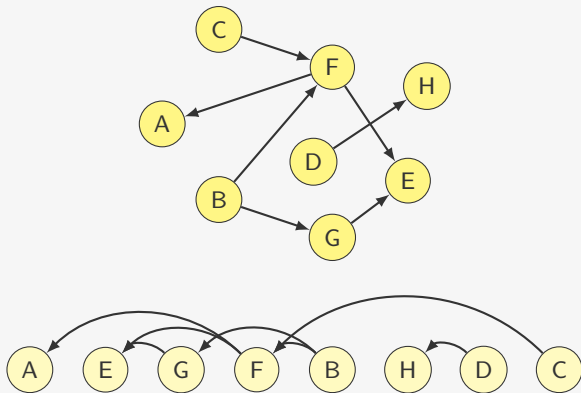
Uma **ordenação topológica** de um grafo **acíclico** é uma ordenação dos vértices cujas arestas estão na **mesma direção**.



Observe que pode haver **várias** ordenações topológicas.

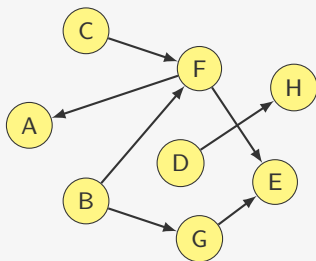
## Ordenação topológica

Uma **ordenação topológica** de um grafo **acíclico** é uma ordenação dos vértices cujas arestas estão na **mesma direção**.

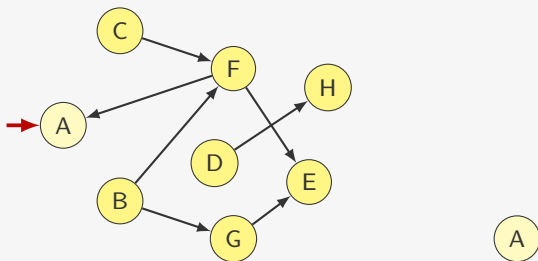


Observe que pode haver **várias** ordenações topológicas.

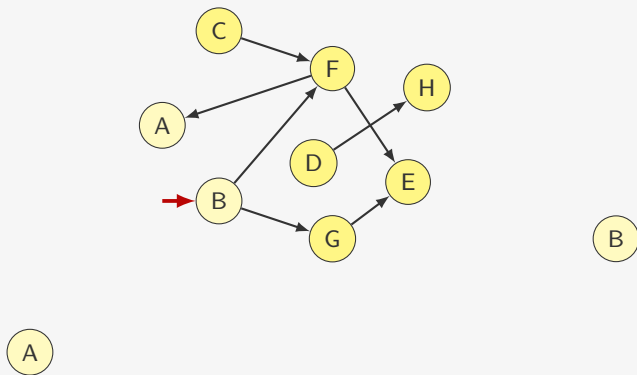
## Encontrando uma ordem de corte



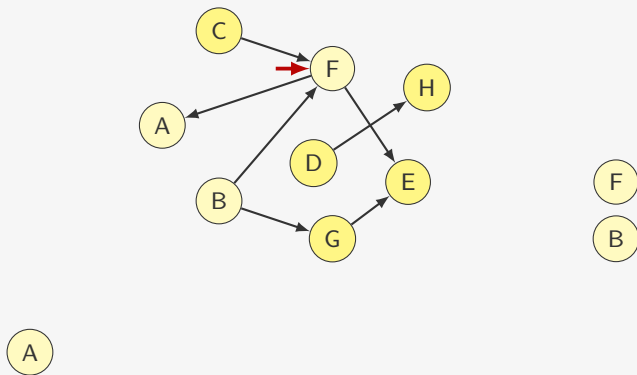
## Encontrando uma ordem de corte



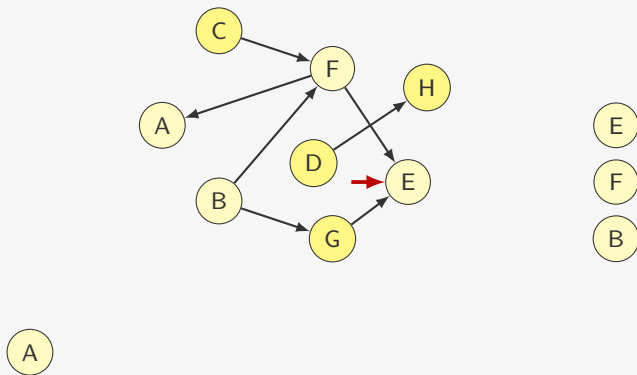
## Encontrando uma ordem de corte



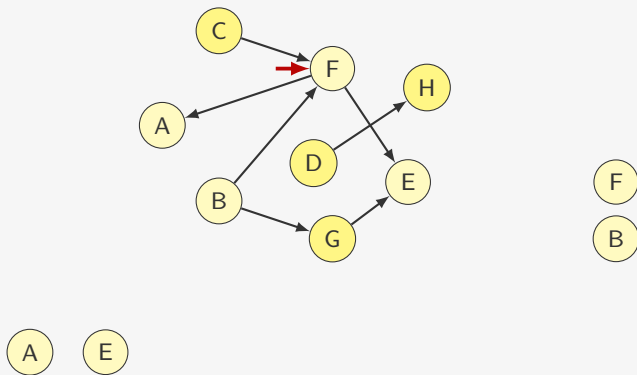
## Encontrando uma ordem de corte



## Encontrando uma ordem de corte

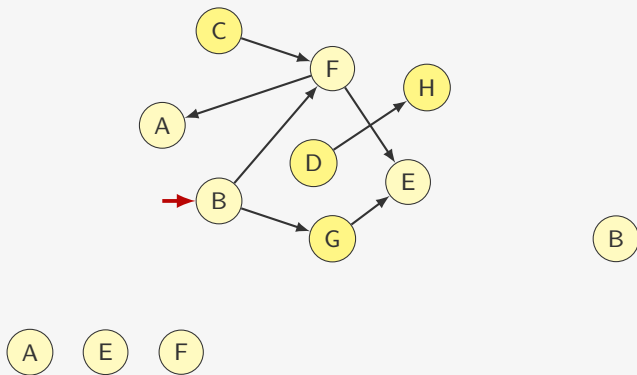


## Encontrando uma ordem de corte

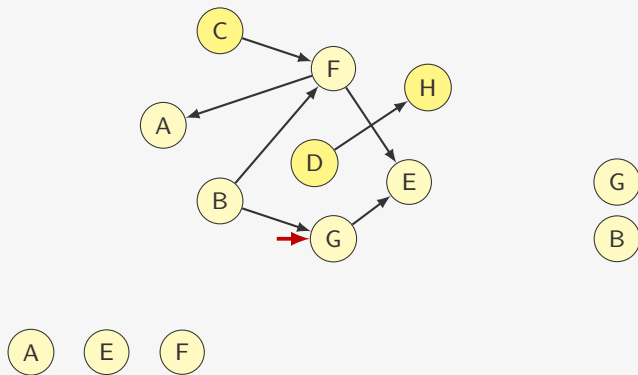




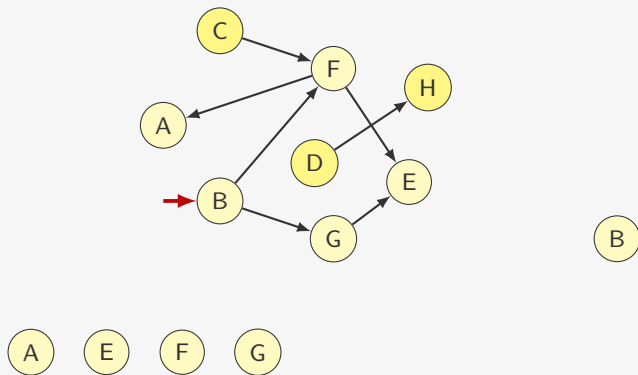
## Encontrando uma ordem de corte



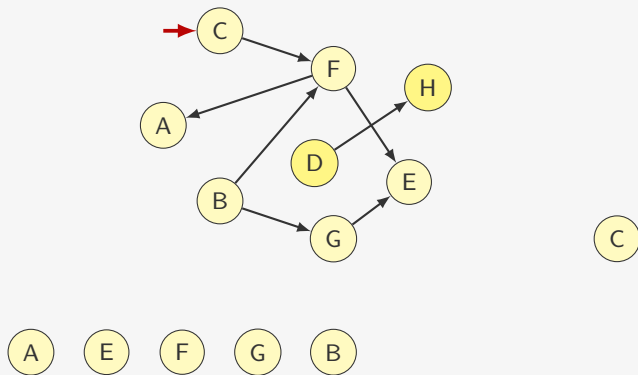
## Encontrando uma ordem de corte



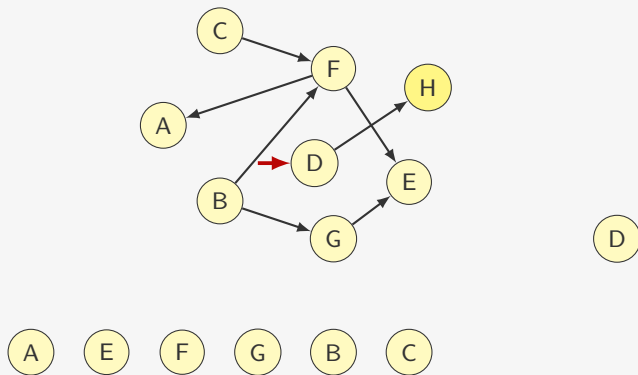
## Encontrando uma ordem de corte



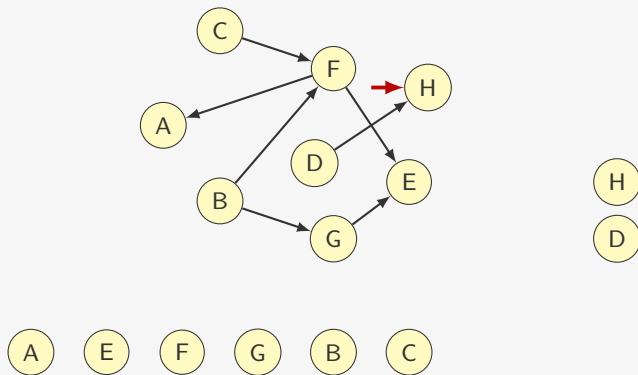
## Encontrando uma ordem de corte



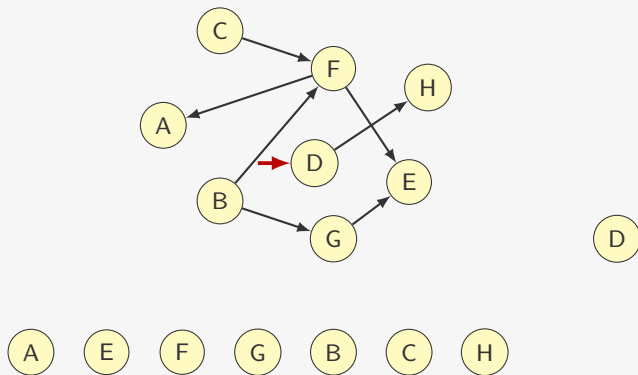
## Encontrando uma ordem de corte



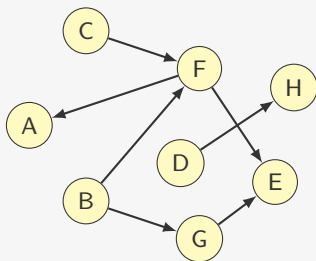
## Encontrando uma ordem de corte



## Encontrando uma ordem de corte

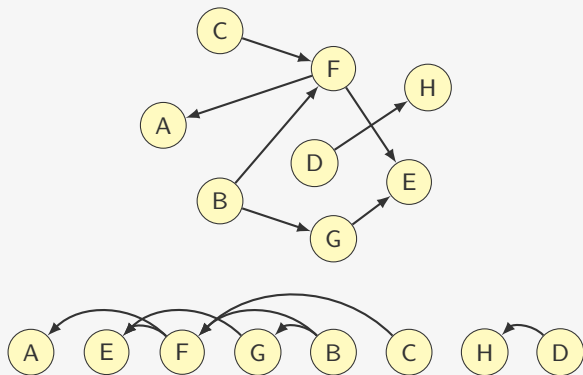


## Encontrando uma ordem de corte





## Encontrando uma ordem de corte



# Como encontrar uma ordenação topológica?

Considere um vértice  $u$  do DAG:

# Como encontrar um ordenação topológica?

Considere um vértice  $u$  do DAG:

- Todo  $v$  tal que  $(u, v)$  é arco deve aparecer antes de  $u$

# Como encontrar um ordenação topológica?

Considere um vértice  $u$  do DAG:

- Todo  $v$  tal que  $(u, v)$  é arco deve aparecer antes de  $u$
- Todo  $w$  tal que  $(v, w)$  é arco deve aparecer antes de  $v$

# Como encontrar um ordenação topológica?

Considere um vértice  $u$  do DAG:

- Todo  $v$  tal que  $(u, v)$  é arco deve aparecer antes de  $u$
- Todo  $w$  tal que  $(v, w)$  é arco deve aparecer antes de  $v$
- E assim por diante

# Como encontrar um ordenação topológica?

Considere um vértice  $u$  do DAG:

- Todo  $v$  tal que  $(u, v)$  é arco deve aparecer antes de  $u$
- Todo  $w$  tal que  $(v, w)$  é arco deve aparecer antes de  $v$
- E assim por diante

Devemos considerar todos  $w$  para os quais existe caminho de  $u$  para  $w$  antes de considerar  $u$

# Como encontrar um ordenação topológica?

Considere um vértice  $u$  do DAG:

- Todo  $v$  tal que  $(u, v)$  é arco deve aparecer antes de  $u$
- Todo  $w$  tal que  $(v, w)$  é arco deve aparecer antes de  $v$
- E assim por diante

Devemos considerar todos  $w$  para os quais existe caminho de  $u$  para  $w$  antes de considerar  $u$

- Lembra uma pós-ordem em árvores binárias...

# Como encontrar um ordenação topológica?

Considere um vértice  $u$  do DAG:

- Todo  $v$  tal que  $(u, v)$  é arco deve aparecer antes de  $u$
- Todo  $w$  tal que  $(v, w)$  é arco deve aparecer antes de  $v$
- E assim por diante

Devemos considerar todos  $w$  para os quais existe caminho de  $u$  para  $w$  antes de considerar  $u$

- Lembra uma pós-ordem em árvores binárias...

Como encontrar todo  $w$  tal que existe caminho de  $u$  para  $w$ ?



# Como encontrar um ordenação topológica?

Considere um vértice  $u$  do DAG:

- Todo  $v$  tal que  $(u, v)$  é arco deve aparecer antes de  $u$
- Todo  $w$  tal que  $(v, w)$  é arco deve aparecer antes de  $v$
- E assim por diante

Devemos considerar todos  $w$  para os quais existe caminho de  $u$  para  $w$  antes de considerar  $u$

- Lembra uma pós-ordem em árvores binárias...

Como encontrar todo  $w$  tal que existe caminho de  $u$  para  $w$ ?

- Busca em profundidade

# Implementação

```
1 void ordenacao_topologica(p_grafo g) {
```

# Implementação

```
1 void ordenacao_topologica(p_grafo g) {  
2     int s, *visitado = malloc(g->n * sizeof(int));
```

# Implementação

```
1 void ordenacao_topologica(p_grafo g) {  
2     int s, *visitado = malloc(g->n * sizeof(int));  
3     for (s = 0; s < g->n; s++)  
4         visitado[s] = 0;
```

# Implementação

```
1 void ordenacao_topologica(p_grafo g) {  
2     int s, *visitado = malloc(g->n * sizeof(int));  
3     for (s = 0; s < g->n; s++)  
4         visitado[s] = 0;  
5     for (s = 0; s < g->n; s++)  
6         if (!visitado[s])
```

# Implementação

```
1 void ordenacao_topologica(p_grafo g) {
2     int s, *visitado = malloc(g->n * sizeof(int));
3     for (s = 0; s < g->n; s++)
4         visitado[s] = 0;
5     for (s = 0; s < g->n; s++)
6         if (!visitado[s])
7             visita_rec(g, visitado, s);
```

# Implementação

```
1 void ordenacao_topologica(p_grafo g) {
2     int s, *visitado = malloc(g->n * sizeof(int));
3     for (s = 0; s < g->n; s++)
4         visitado[s] = 0;
5     for (s = 0; s < g->n; s++)
6         if (!visitado[s])
7             visita_rec(g, visitado, s);
8     free(visitado);
}
```

# Implementação

```
1 void ordenacao_topologica(p_grafo g) {
2     int s, *visitado = malloc(g->n * sizeof(int));
3     for (s = 0; s < g->n; s++)
4         visitado[s] = 0;
5     for (s = 0; s < g->n; s++)
6         if (!visitado[s])
7             visita_rec(g, visitado, s);
8     free(visitado);
9     printf("\n");
10 }
```



# Implementação

```
1 void ordenacao_topologica(p_grafo g) {
2     int s, *visitado = malloc(g->n * sizeof(int));
3     for (s = 0; s < g->n; s++)
4         visitado[s] = 0;
5     for (s = 0; s < g->n; s++)
6         if (!visitado[s])
7             visita_rec(g, visitado, s);
8     free(visitado);
9     printf("\n");
10 }

1 void visita_rec(p_grafo g, int *visitado, int v) {
```

# Implementação

```
1 void ordenacao_topologica(p_grafo g) {
2     int s, *visitado = malloc(g->n * sizeof(int));
3     for (s = 0; s < g->n; s++)
4         visitado[s] = 0;
5     for (s = 0; s < g->n; s++)
6         if (!visitado[s])
7             visita_rec(g, visitado, s);
8     free(visitado);
9     printf("\n");
10 }
```

```
1 void visita_rec(p_grafo g, int *visitado, int v) {
2     p_no t;
3     visitado[v] = 1;
```

# Implementação

```
1 void ordenacao_topologica(p_grafo g) {
2     int s, *visitado = malloc(g->n * sizeof(int));
3     for (s = 0; s < g->n; s++)
4         visitado[s] = 0;
5     for (s = 0; s < g->n; s++)
6         if (!visitado[s])
7             visita_rec(g, visitado, s);
8     free(visitado);
9     printf("\n");
10 }
```

```
1 void visita_rec(p_grafo g, int *visitado, int v) {
2     p_no t;
3     visitado[v] = 1;
4     for (t = g->adj[v]; t != NULL; t = t->prox)
```

# Implementação

```
1 void ordenacao_topologica(p_grafo g) {
2     int s, *visitado = malloc(g->n * sizeof(int));
3     for (s = 0; s < g->n; s++)
4         visitado[s] = 0;
5     for (s = 0; s < g->n; s++)
6         if (!visitado[s])
7             visita_rec(g, visitado, s);
8     free(visitado);
9     printf("\n");
10 }
```

```
1 void visita_rec(p_grafo g, int *visitado, int v) {
2     p_no t;
3     visitado[v] = 1;
4     for (t = g->adj[v]; t != NULL; t = t->prox)
5         if (!visitado[t->v])
```

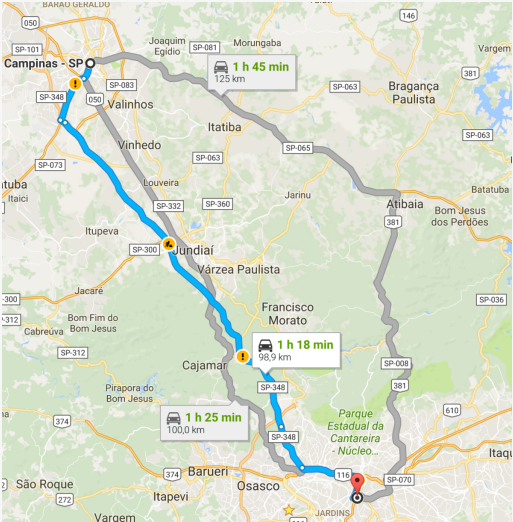
# Implementação

```
1 void ordenacao_topologica(p_grafo g) {
2     int s, *visitado = malloc(g->n * sizeof(int));
3     for (s = 0; s < g->n; s++)
4         visitado[s] = 0;
5     for (s = 0; s < g->n; s++)
6         if (!visitado[s])
7             visita_rec(g, visitado, s);
8     free(visitado);
9     printf("\n");
10 }
```

```
1 void visita_rec(p_grafo g, int *visitado, int v) {
2     p_no t;
3     visitado[v] = 1;
4     for (t = g->adj[v]; t != NULL; t = t->prox)
5         if (!visitado[t->v])
6             visita_rec(g, visitado, t->v);
7     printf("%d ", v);
8 }
```

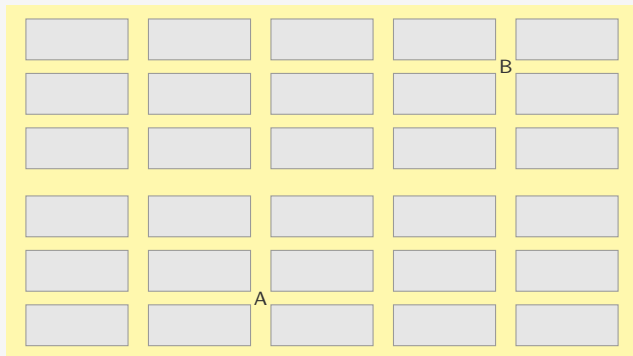
# Encontrando o menor caminho

Como encontrar o menor tempo para ir de A para B?



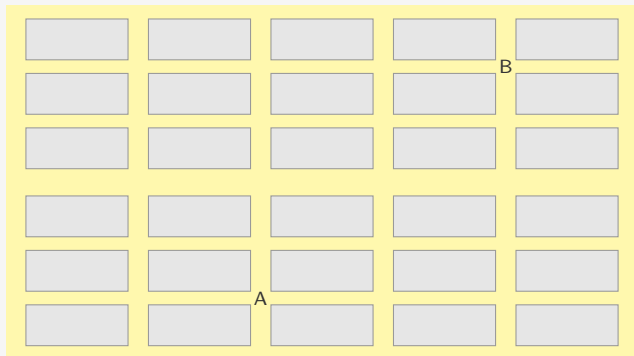
## Encontrando o menor caminho

Como encontrar o menor tempo para ir de A para B?



## Encontrando o menor caminho

Como encontrar o menor tempo para ir de A para B?

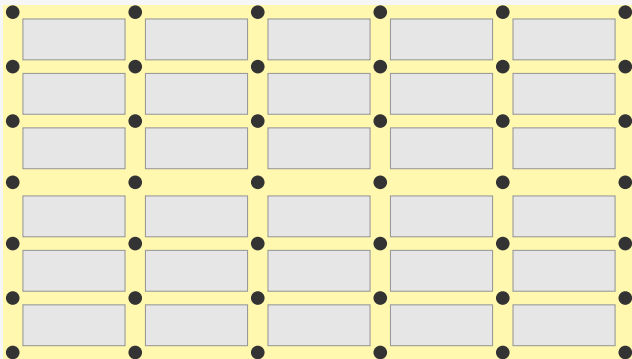


Modelamos como um digrafo **com pesos nos arcos**:



## Encontrando o menor caminho

Como encontrar o menor tempo para ir de A para B?

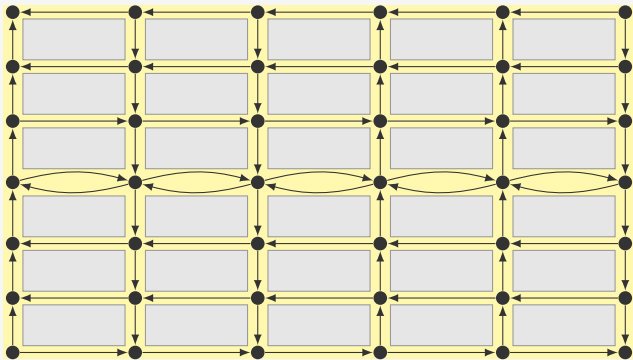


Modelamos como um digrafo **com pesos nos arcos**:

- Um vértice em cada cruzamento

## Encontrando o menor caminho

Como encontrar o menor tempo para ir de A para B?

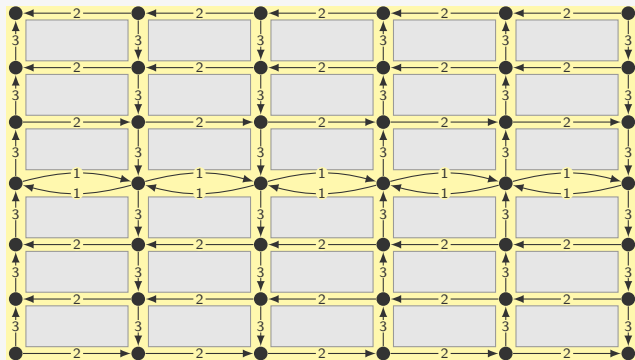


Modelamos como um digrafo **com pesos nos arcos**:

- Um vértice em cada cruzamento
- Um arco entre vértices consecutivos

# Encontrando o menor caminho

Como encontrar o menor tempo para ir de A para B?

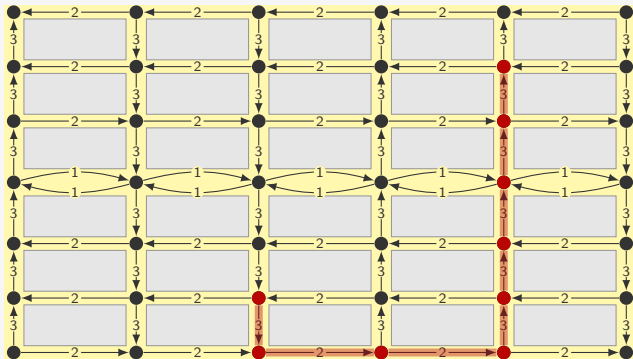


Modelamos como um digrafo **com pesos nos arcos**:

- Um vértice em cada cruzamento
- Um arco entre vértices consecutivos
- O peso do arco  $(u, v)$  é o tempo de viagem de  $u$  para  $v$

# Encontrando o menor caminho

Como encontrar o menor tempo para ir de A para B?



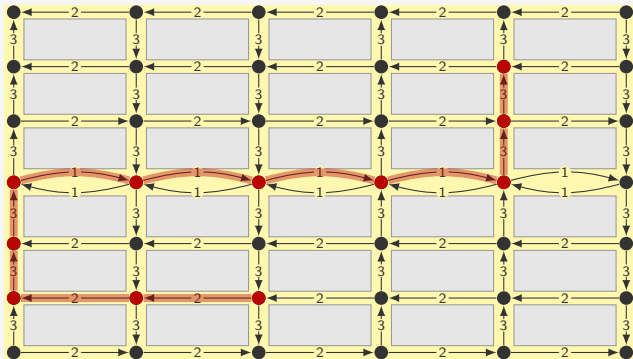
Modelamos como um digrafo **com pesos nos arcos**:

- Um vértice em cada cruzamento
- Um arco entre vértices consecutivos
- O peso do arco  $(u, v)$  é o tempo de viagem de  $u$  para  $v$

Tempo de percurso do caminho: **22**

# Encontrando o menor caminho

Como encontrar o menor tempo para ir de A para B?



Modelamos como um digrafo **com pesos nos arcos**:

- Um vértice em cada cruzamento
- Um arco entre vértices consecutivos
- O peso do arco  $(u, v)$  é o tempo de viagem de  $u$  para  $v$

Tempo de percurso do caminho: **20**

## Digrafos com pesos nas arestas - Representação

Como representar grafos com pesos nas arestas?

## Digrafos com pesos nas arestas - Representação

Como representar grafos com pesos nas arestas?

Listas de Adjacência:

## Digrafos com pesos nas arestas - Representação

Como representar grafos com pesos nas arestas?

Listas de Adjacência:

- Basta adicionar um campo **peso** no Nó da lista ligada



## Digrafos com pesos nas arestas - Representação

Como representar grafos com pesos nas arestas?

Listas de Adjacência:

- Basta adicionar um campo **peso** no Nó da lista ligada

Matriz de Adjacências:

# Digrafos com pesos nas arestas - Representação

Como representar grafos com pesos nas arestas?

Listas de Adjacência:

- Basta adicionar um campo **peso** no Nó da lista ligada

Matriz de Adjacências:

- Podemos indicar que não há arco usando peso **0**

# Digrafos com pesos nas arestas - Representação

Como representar grafos com pesos nas arestas?

Listas de Adjacência:

- Basta adicionar um campo **peso** no Nó da lista ligada

Matriz de Adjacências:

- Podemos indicar que não há arco usando peso **0**
  - Isso nem sempre é uma boa opção

# Digrafos com pesos nas arestas - Representação

Como representar grafos com pesos nas arestas?

Listas de Adjacência:

- Basta adicionar um campo **peso** no Nó da lista ligada

Matriz de Adjacências:

- Podemos indicar que não há arco usando peso **0**
  - Isso nem sempre é uma boa opção
  - Podemos trocar por **-1** ou então **INT\_MAX**

# Digrafos com pesos nas arestas - Representação

Como representar grafos com pesos nas arestas?

Listas de Adjacência:

- Basta adicionar um campo **peso** no Nó da lista ligada

Matriz de Adjacências:

- Podemos indicar que não há arco usando peso **0**
  - Isso nem sempre é uma boa opção
  - Podemos trocar por **-1** ou então **INT\_MAX**
- Ou fazemos uma struct com dois campos

# Digrafos com pesos nas arestas - Representação

Como representar grafos com pesos nas arestas?

Listas de Adjacência:

- Basta adicionar um campo **peso** no Nó da lista ligada

Matriz de Adjacências:

- Podemos indicar que não há arco usando peso **0**
  - Isso nem sempre é uma boa opção
  - Podemos trocar por **-1** ou então **INT\_MAX**
- Ou fazemos uma struct com dois campos
  - um indica se há arco ou não

# Digrafos com pesos nas arestas - Representação

Como representar grafos com pesos nas arestas?

Listas de Adjacência:

- Basta adicionar um campo **peso** no Nó da lista ligada

Matriz de Adjacências:

- Podemos indicar que não há arco usando peso **0**
  - Isso nem sempre é uma boa opção
  - Podemos trocar por **-1** ou então **INT\_MAX**
- Ou fazemos uma struct com dois campos
  - um indica se há arco ou não
  - outro denota o peso do arco

## Caminhos mínimos

Queremos encontrar um caminho de peso mínimo de  $u$  para  $v$  no digrafo



## Caminhos mínimos

Queremos encontrar um caminho de peso mínimo de  $u$  para  $v$  no digrafo

- Consideramos que os pesos são **não-negativos**

## Caminhos mínimos

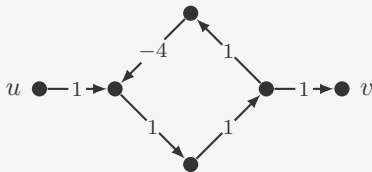
Queremos encontrar um caminho de peso mínimo de  $u$  para  $v$  no digrafo

- Consideramos que os pesos são **não-negativos**
- Se não, podemos querer percorrer um ciclo negativo infinitas vezes...

## Caminhos mínimos

Queremos encontrar um caminho de peso mínimo de  $u$  para  $v$  no digrafo

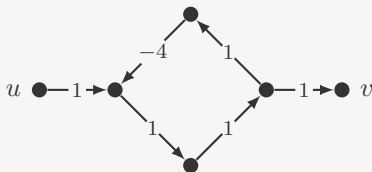
- Consideramos que os pesos são **não-negativos**
- Se não, podemos querer percorrer um ciclo negativo infinitas vezes...



## Caminhos mínimos

Queremos encontrar um caminho de peso mínimo de  $u$  para  $v$  no digrafo

- Consideramos que os pesos são **não-negativos**
- Se não, podemos querer percorrer um ciclo negativo infinitas vezes...

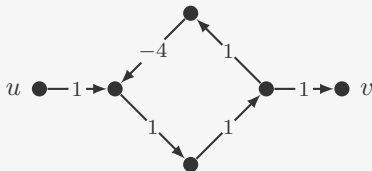


Como é o caminho mínimo de  $u$  para  $v$ ?

## Caminhos mínimos

Queremos encontrar um caminho de peso mínimo de  $u$  para  $v$  no digrafo

- Consideramos que os pesos são **não-negativos**
- Se não, podemos querer percorrer um ciclo negativo infinitas vezes...



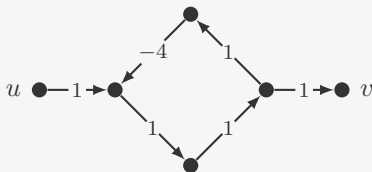
Como é o caminho mínimo de  $u$  para  $v$ ?

- Ou  $u$  é vizinho de  $v$

## Caminhos mínimos

Queremos encontrar um caminho de peso mínimo de  $u$  para  $v$  no digrafo

- Consideramos que os pesos são **não-negativos**
- Se não, podemos querer percorrer um ciclo negativo infinitas vezes...



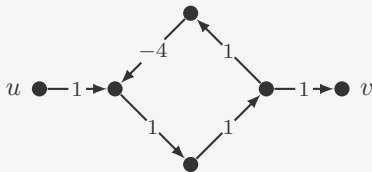
Como é o caminho mínimo de  $u$  para  $v$ ?

- Ou  $u$  é vizinho de  $v$
- Ou o caminho passa por um vizinho  $w$  de  $v$

## Caminhos mínimos

Queremos encontrar um caminho de peso mínimo de  $u$  para  $v$  no digrafo

- Consideramos que os pesos são **não-negativos**
- Se não, podemos querer percorrer um ciclo negativo infinitas vezes...



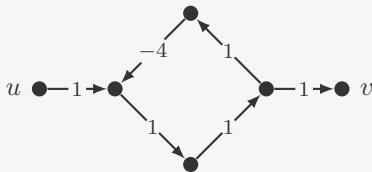
Como é o caminho mínimo de  $u$  para  $v$ ?

- Ou  $u$  é vizinho de  $v$
- Ou o caminho passa por um vizinho  $w$  de  $v$ 
  - Soma do peso do caminho de  $u$  para  $w$  e de  $(w, v)$  é mínima

## Caminhos mínimos

Queremos encontrar um caminho de peso mínimo de  $u$  para  $v$  no digrafo

- Consideramos que os pesos são **não-negativos**
- Se não, podemos querer percorrer um ciclo negativo infinitas vezes...



Como é o caminho mínimo de  $u$  para  $v$ ?

- Ou  $u$  é vizinho de  $v$
- Ou o caminho passa por um vizinho  $w$  de  $v$ 
  - Soma do peso do caminho de  $u$  para  $w$  e de  $(w, v)$  é mínima
  - Este caminho de  $u$  a  $w$  tem que ter peso mínimo



# Árvore de Caminhos mínimos

Árvore de caminhos mínimos (a partir de  $u$ ):

## Árvore de Caminhos mínimos

Árvore de caminhos mínimos (a partir de  $u$ ):

- Dado  $u$ , o algoritmo encontra uma árvore enraizada em  $u$

# Árvore de Caminhos mínimos

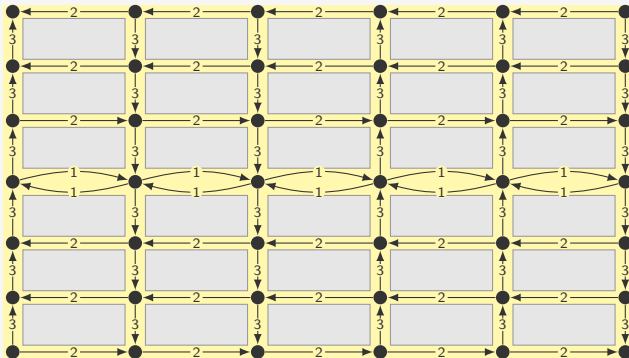
Árvore de caminhos mínimos (a partir de  $u$ ):

- Dado  $u$ , o algoritmo encontra uma árvore enraizada em  $u$
- De forma que o caminho de  $v$  para  $u$  na árvore seja um caminho mínimo de  $u$  para  $v$  no digrafo

# Árvore de Caminhos mínimos

Árvore de caminhos mínimos (a partir de  $u$ ):

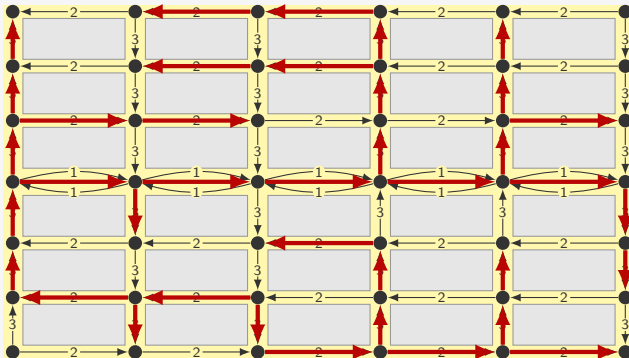
- Dado  $u$ , o algoritmo encontra uma árvore enraizada em  $u$
- De forma que o caminho de  $v$  para  $u$  na árvore seja um caminho mínimo de  $u$  para  $v$  no digrafo



# Árvore de Caminhos mínimos

Árvore de caminhos mínimos (a partir de  $u$ ):

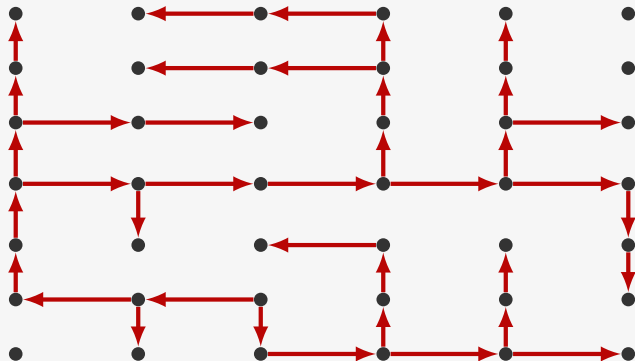
- Dado  $u$ , o algoritmo encontra uma árvore enraizada em  $u$
- De forma que o caminho de  $v$  para  $u$  na árvore seja um caminho mínimo de  $u$  para  $v$  no digrafo



# Árvore de Caminhos mínimos

Árvore de caminhos mínimos (a partir de  $u$ ):

- Dado  $u$ , o algoritmo encontra uma árvore enraizada em  $u$
- De forma que o caminho de  $v$  para  $u$  na árvore seja um caminho mínimo de  $u$  para  $v$  no digrafo



# Algoritmo de Dijkstra

Em um certo momento já construímos parte da árvore

## Algoritmo de Dijkstra

Em um certo momento já construímos parte da árvore

- Temos um conjunto de vértices que ainda não entraram



## Algoritmo de Dijkstra

Em um certo momento já construímos parte da árvore

- Temos um conjunto de vértices que ainda não entraram
- Alguns destes são vizinhos de vértices já na árvore

# Algoritmo de Dijkstra

Em um certo momento já construímos parte da árvore

- Temos um conjunto de vértices que ainda não entraram
- Alguns destes são vizinhos de vértices já na árvore
- Eles estão na **franja**

# Algoritmo de Dijkstra

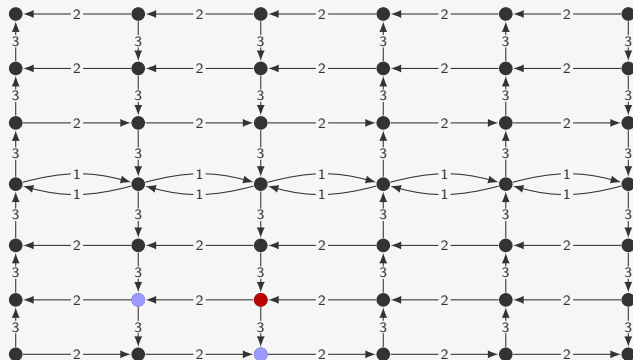
Em um certo momento já construímos parte da árvore

- Temos um conjunto de vértices que ainda não entraram
- Alguns destes são vizinhos de vértices já na árvore
- Eles estão na **franja**
- Pegamos o vértice na franja mais próximo de *u*

# Algoritmo de Dijkstra

Em um certo momento já construímos parte da árvore

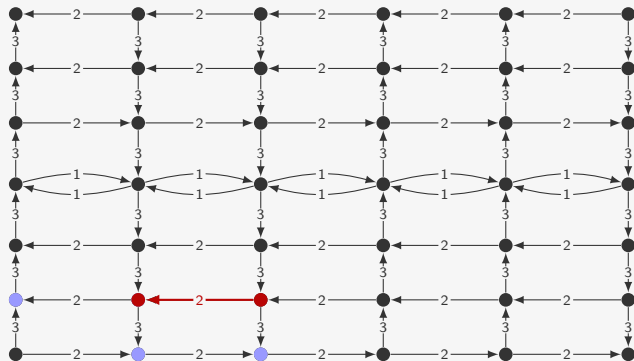
- Temos um conjunto de vértices que ainda não entraram
- Alguns destes são vizinhos de vértices já na árvore
- Eles estão na **franja**
- Pegamos o vértice na franja mais próximo de  $u$



# Algoritmo de Dijkstra

Em um certo momento já construímos parte da árvore

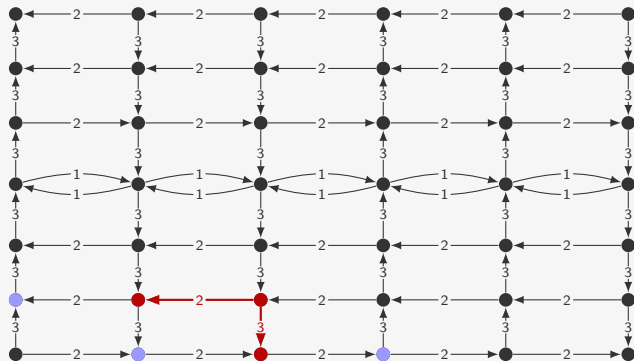
- Temos um conjunto de vértices que ainda não entraram
- Alguns destes são vizinhos de vértices já na árvore
- Eles estão na **franja**
- Pegamos o vértice na franja mais próximo de  $u$



# Algoritmo de Dijkstra

Em um certo momento já construímos parte da árvore

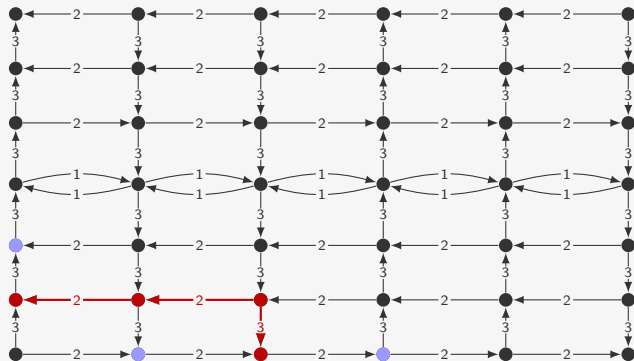
- Temos um conjunto de vértices que ainda não entraram
- Alguns destes são vizinhos de vértices já na árvore
- Eles estão na **franja**
- Pegamos o vértice na franja mais próximo de  $u$



# Algoritmo de Dijkstra

Em um certo momento já construímos parte da árvore

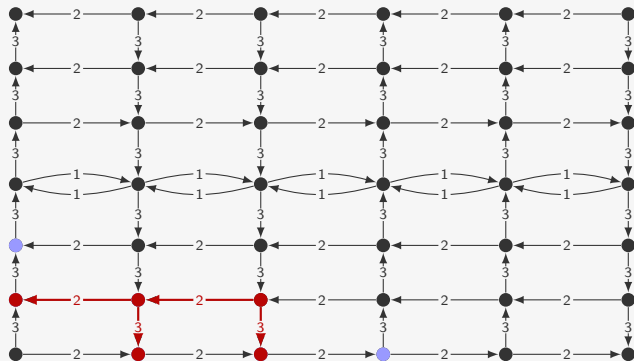
- Temos um conjunto de vértices que ainda não entraram
- Alguns destes são vizinhos de vértices já na árvore
- Eles estão na **franja**
- Pegamos o vértice na franja mais próximo de  $u$



# Algoritmo de Dijkstra

Em um certo momento já construímos parte da árvore

- Temos um conjunto de vértices que ainda não entraram
- Alguns destes são vizinhos de vértices já na árvore
- Eles estão na **franja**
- Pegamos o vértice na franja mais próximo de  $u$

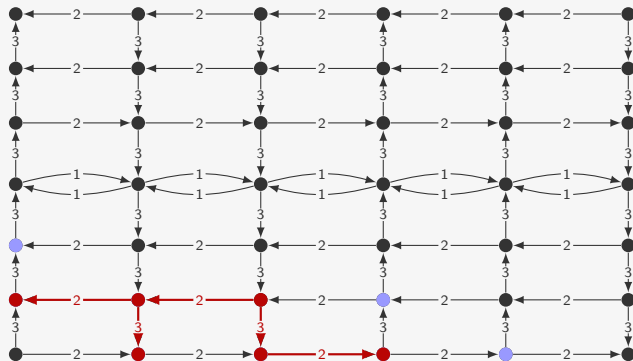




# Algoritmo de Dijkstra

Em um certo momento já construímos parte da árvore

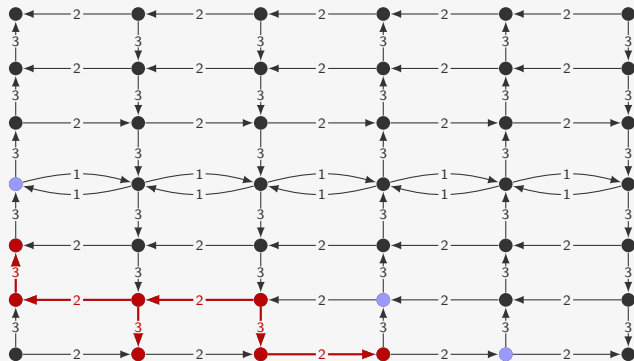
- Temos um conjunto de vértices que ainda não entraram
- Alguns destes são vizinhos de vértices já na árvore
- Eles estão na **franja**
- Pegamos o vértice na franja mais próximo de  $u$



# Algoritmo de Dijkstra

Em um certo momento já construímos parte da árvore

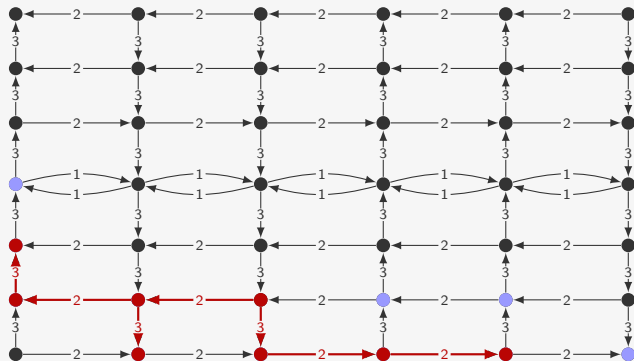
- Temos um conjunto de vértices que ainda não entraram
- Alguns destes são vizinhos de vértices já na árvore
- Eles estão na **franja**
- Pegamos o vértice na franja mais próximo de  $u$



# Algoritmo de Dijkstra

Em um certo momento já construímos parte da árvore

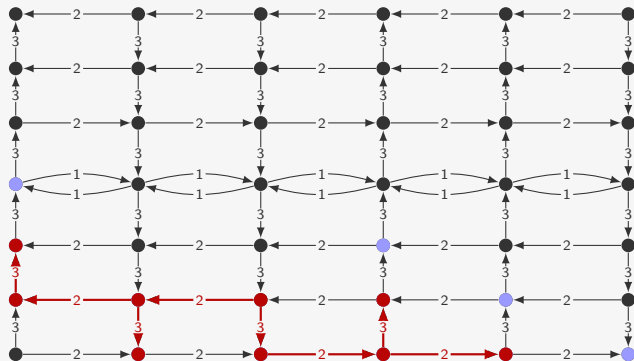
- Temos um conjunto de vértices que ainda não entraram
- Alguns destes são vizinhos de vértices já na árvore
- Eles estão na **franja**
- Pegamos o vértice na franja mais próximo de  $u$



# Algoritmo de Dijkstra

Em um certo momento já construímos parte da árvore

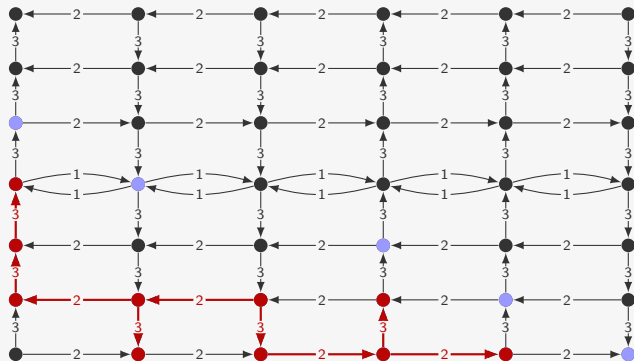
- Temos um conjunto de vértices que ainda não entraram
- Alguns destes são vizinhos de vértices já na árvore
- Eles estão na **franja**
- Pegamos o vértice na franja mais próximo de  $u$



# Algoritmo de Dijkstra

Em um certo momento já construímos parte da árvore

- Temos um conjunto de vértices que ainda não entraram
- Alguns destes são vizinhos de vértices já na árvore
- Eles estão na **franja**
- Pegamos o vértice na franja mais próximo de  $u$



# Implementação

# Implementação

## Grafo

```
1 typedef struct no *p_no;
2
3 struct no {
4     int v;
5     int peso;
6     p_no prox;
7 };
8
9 typedef struct grafo *p_grafo;
10
11 struct grafo {
12     int n;
13     p_no *adj;
14 };
```

# Implementação

## Grafo

```
1 typedef struct no *p_no;
2
3 struct no {
4     int v;
5     int peso;
6     p_no prox;
7 };
8
9 typedef struct grafo *p_grafo;
10
11 struct grafo {
12     int n;
13     p_no *adj;
14 };
```

## Heap binário

```
1 typedef struct {
2     int prioridade;
3     int vertice;
4 } Item;
5
6 typedef struct {
7     Item *v;
8     int *indice;
9     int n, tamanho;
10 } FP;
11
12 typedef FP * p_fp;
```



# Implementação

```
1 int * dijkstra(p_grafo g, int s) {
```

# Implementação

```
1 int * dijkstra(p_grafo g, int s) {  
2     int v, *pai = malloc(g->n * sizeof(int));
```

# Implementação

```
1 int * dijkstra(p_grafo g, int s) {
2     int v, *pai = malloc(g->n * sizeof(int));
3     p_no t;
4     p_fp h = criar_fprio(g->n);
```

# Implementação

```
1 int * dijkstra(p_grafo g, int s) {
2     int v, *pai = malloc(g->n * sizeof(int));
3     p_no t;
4     p_fp h = criar_fprio(g->n);
5     for (v = 0; v < g->n; v++) {
6         pai[v] = -1;
7         insere(h, v, INT_MAX);
8     }
```

# Implementação

```
1 int * dijkstra(p_grafo g, int s) {
2     int v, *pai = malloc(g->n * sizeof(int));
3     p_no t;
4     p_fp h = criar_fprio(g->n);
5     for (v = 0; v < g->n; v++) {
6         pai[v] = -1;
7         insere(h, v, INT_MAX);
8     }
9     pai[s] = s;
```

# Implementação

```
1 int * dijkstra(p_grafo g, int s) {
2     int v, *pai = malloc(g->n * sizeof(int));
3     p_no t;
4     p_fp h = criar_fprio(g->n);
5     for (v = 0; v < g->n; v++) {
6         pai[v] = -1;
7         insere(h, v, INT_MAX);
8     }
9     pai[s] = s;
10    diminuiprioridade(h, s, 0);
```

# Implementação

```
1 int * dijkstra(p_grafo g, int s) {
2     int v, *pai = malloc(g->n * sizeof(int));
3     p_no t;
4     p_fp h = criar_fprio(g->n);
5     for (v = 0; v < g->n; v++) {
6         pai[v] = -1;
7         insere(h, v, INT_MAX);
8     }
9     pai[s] = s;
10    diminuiprioridade(h, s, 0);
11    while (!vazia(h)) {
```

# Implementação

```
1 int * dijkstra(p_grafo g, int s) {
2     int v, *pai = malloc(g->n * sizeof(int));
3     p_no t;
4     p_fp h = criar_fprio(g->n);
5     for (v = 0; v < g->n; v++) {
6         pai[v] = -1;
7         insere(h, v, INT_MAX);
8     }
9     pai[s] = s;
10    diminuiprioridade(h, s, 0);
11    while (!vazia(h)) {
12        v = extrai_minimo(h);
```



# Implementação

```
1 int * dijkstra(p_grafo g, int s) {
2     int v, *pai = malloc(g->n * sizeof(int));
3     p_no t;
4     p_fp h = criar_fprio(g->n);
5     for (v = 0; v < g->n; v++) {
6         pai[v] = -1;
7         insere(h, v, INT_MAX);
8     }
9     pai[s] = s;
10    diminuiprioridade(h, s, 0);
11    while (!vazia(h)) {
12        v = extrai_minimo(h);
13        if (prioridade(h, v) != INT_MAX)
```

# Implementação

```
1 int * dijkstra(p_grafo g, int s) {
2     int v, *pai = malloc(g->n * sizeof(int));
3     p_no t;
4     p_fp h = criar_fprio(g->n);
5     for (v = 0; v < g->n; v++) {
6         pai[v] = -1;
7         insere(h, v, INT_MAX);
8     }
9     pai[s] = s;
10    diminuiprioridade(h, s, 0);
11    while (!vazia(h)) {
12        v = extrai_minimo(h);
13        if (prioridade(h, v) != INT_MAX)
14            for (t = g->adj[v]; t != NULL; t = t->prox)
```

# Implementação

```
1 int * dijkstra(p_grafo g, int s) {
2     int v, *pai = malloc(g->n * sizeof(int));
3     p_no t;
4     p_fp h = criar_fprio(g->n);
5     for (v = 0; v < g->n; v++) {
6         pai[v] = -1;
7         insere(h, v, INT_MAX);
8     }
9     pai[s] = s;
10    diminuiprioridade(h, s, 0);
11    while (!vazia(h)) {
12        v = extrai_minimo(h);
13        if (prioridade(h, v) != INT_MAX)
14            for (t = g->adj[v]; t != NULL; t = t->prox)
15                if (prioridade(h, v)+t->peso < prioridade(h, t->v)) {
```

# Implementação

```
1 int * dijkstra(p_grafo g, int s) {
2     int v, *pai = malloc(g->n * sizeof(int));
3     p_no t;
4     p_fp h = criar_fprio(g->n);
5     for (v = 0; v < g->n; v++) {
6         pai[v] = -1;
7         insere(h, v, INT_MAX);
8     }
9     pai[s] = s;
10    diminuiprioridade(h, s, 0);
11    while (!vazia(h)) {
12        v = extrai_minimo(h);
13        if (prioridade(h, v) != INT_MAX)
14            for (t = g->adj[v]; t != NULL; t = t->prox)
15                if (prioridade(h, v)+t->peso < prioridade(h, t->v)) {
16                    diminuiprioridade(h,t->v,prioridade(h,v)+t->peso);
```

# Implementação

```
1 int * dijkstra(p_grafo g, int s) {
2     int v, *pai = malloc(g->n * sizeof(int));
3     p_no t;
4     p_fp h = criar_fprio(g->n);
5     for (v = 0; v < g->n; v++) {
6         pai[v] = -1;
7         insere(h, v, INT_MAX);
8     }
9     pai[s] = s;
10    diminuiprioridade(h, s, 0);
11    while (!vazia(h)) {
12        v = extrai_minimo(h);
13        if (prioridade(h, v) != INT_MAX)
14            for (t = g->adj[v]; t != NULL; t = t->prox)
15                if (prioridade(h, v)+t->peso < prioridade(h, t->v)) {
16                    diminuiprioridade(h,t->v,prioridade(h,v)+t->peso);
17                    pai[t->v] = v;
18                }
19    }
```

# Implementação

```
1 int * dijkstra(p_grafo g, int s) {
2     int v, *pai = malloc(g->n * sizeof(int));
3     p_no t;
4     p_fp h = criar_fprio(g->n);
5     for (v = 0; v < g->n; v++) {
6         pai[v] = -1;
7         insere(h, v, INT_MAX);
8     }
9     pai[s] = s;
10    diminuiprioridade(h, s, 0);
11    while (!vazia(h)) {
12        v = extrai_minimo(h);
13        if (prioridade(h, v) != INT_MAX)
14            for (t = g->adj[v]; t != NULL; t = t->prox)
15                if (prioridade(h, v)+t->peso < prioridade(h, t->v)) {
16                    diminuiprioridade(h,t->v,prioridade(h,v)+t->peso);
17                    pai[t->v] = v;
18                }
19    }
20    return pai;
21 }
```

# Implementação

```
1 int * dijkstra(p_grafo g, int s) {
2     int v, *pai = malloc(g->n * sizeof(int));
3     p_no t;
4     p_fp h = criar_fprio(g->n);
5     for (v = 0; v < g->n; v++) {
6         pai[v] = -1;
7         insere(h, v, INT_MAX);
8     }
9     pai[s] = s;
10    diminuiprioridade(h, s, 0);
11    while (!vazia(h)) {
12        v = extrai_minimo(h);
13        if (prioridade(h, v) != INT_MAX)
14            for (t = g->adj[v]; t != NULL; t = t->prox)
15                if (prioridade(h, v)+t->peso < prioridade(h, t->v)) {
16                    diminuiprioridade(h,t->v,prioridade(h,v)+t->peso);
17                    pai[t->v] = v;
18                }
19    }
20    return pai;
21 }
```

Tempo:  $O(|E| \lg |V|)$