SIN343 - Desafios de Programação

Algoritmos em Grafos

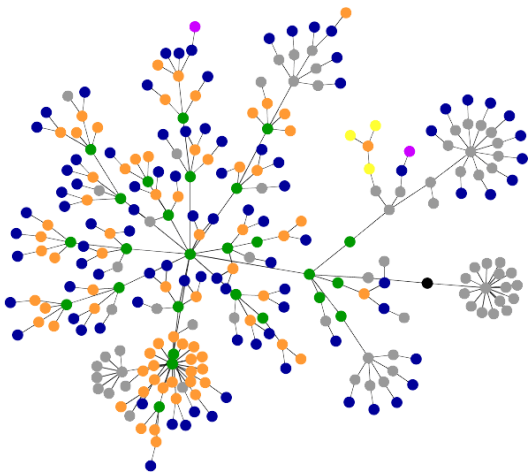
Prof. Pablo Munhoz

pablo.munhoz@ufv.br

Universidade Federal de Viçosa
Campus Rio Paranaíba

Grafos

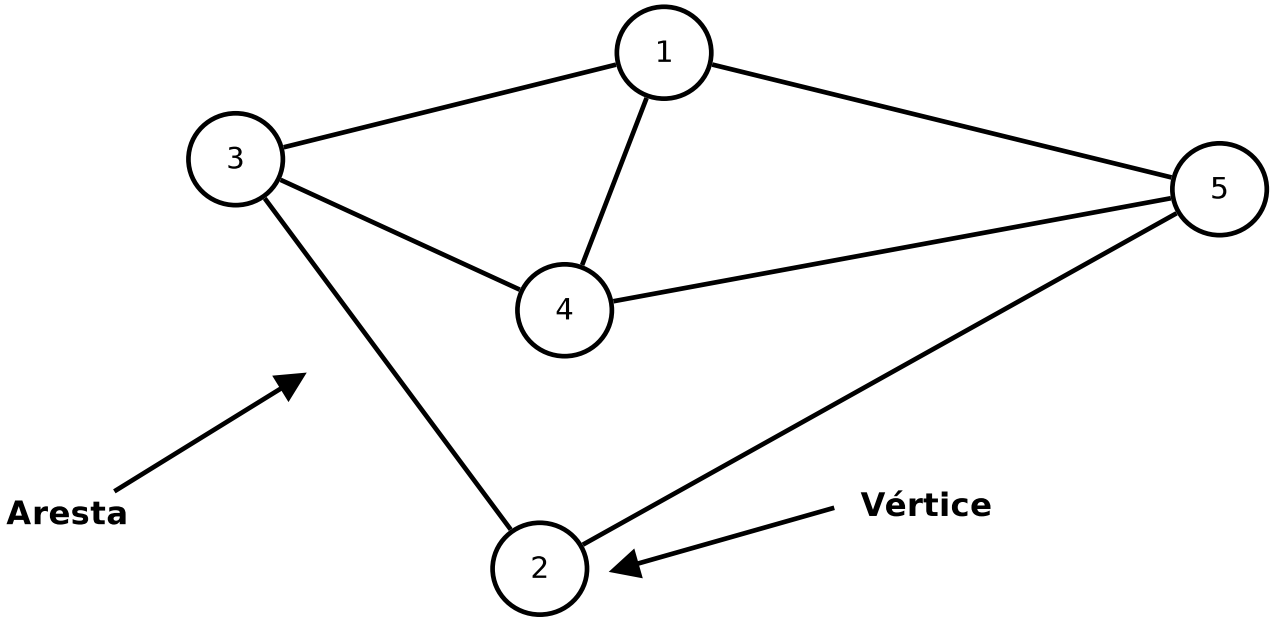
- Forma de representação muito utilizada para conjuntos de pontos e ligações
- Grande representatividade: Engenharia, Computação, Matemática, Economia, etc.
- Exemplos de informações representadas por grafos:
  - Mapas de rodovias
  - Redes de computadores
  - Redes de transporte e de comunicação
  - Circuitos impressos
  - Redes lógicas
- Assim, definição de um tipo abstrato: `GRAFO`



Grafos

Conceitos básicos

- Grafo: conjunto de vértices e arestas (arcos)
- Vértice: objeto simples que pode conter um nome e atributos
- Arestas: representam a conexão entre dois vértices

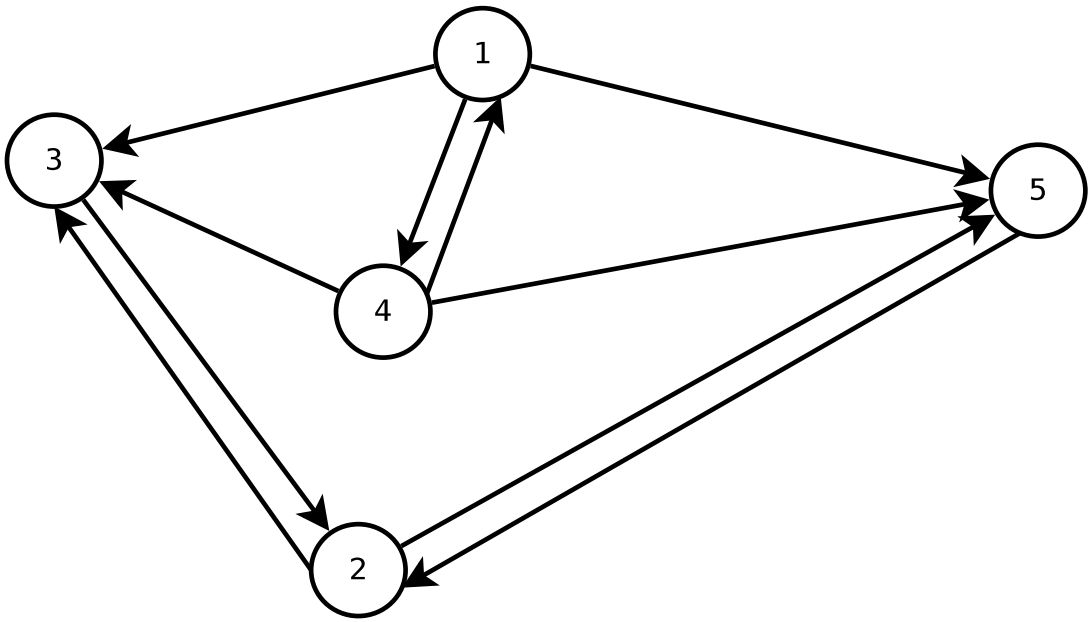


Notação: G(V,A)

- G: grafo
- V: vértices
- A: arestas

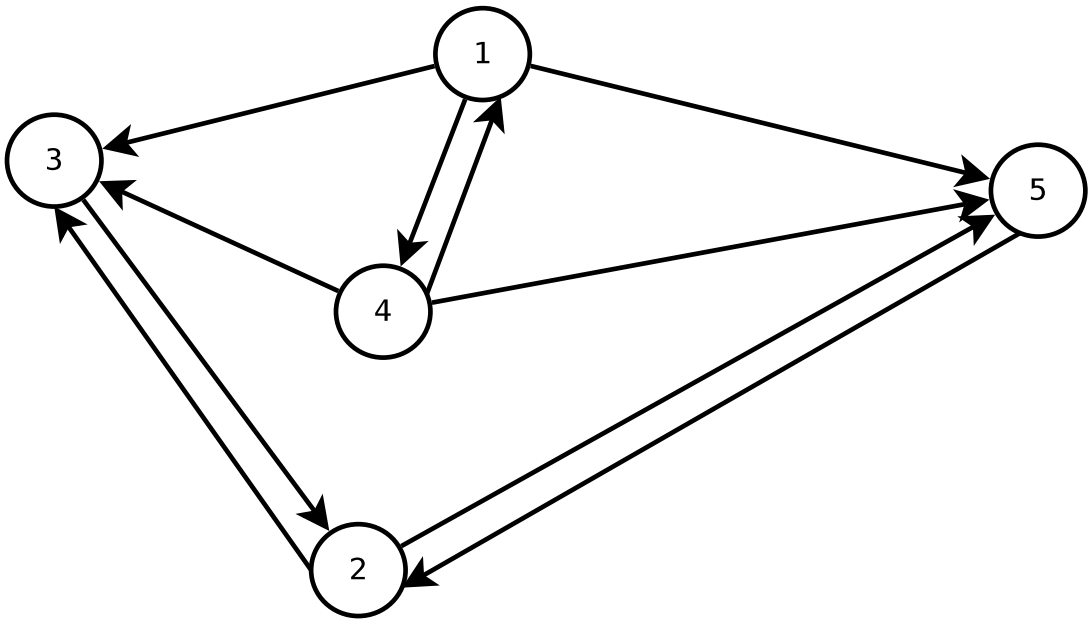
Grafos direcionados

- Grafo que possui uma aresta especial: também chamada de arco
- Grafo onde uma aresta (u, v) sai o vértice u e entra no vértice v.
- O vértice u é **adjacente** ao vértice v
- Podem haver aresta de um vértice para ele mesmo
- $(u, v) \neq (v, u)$



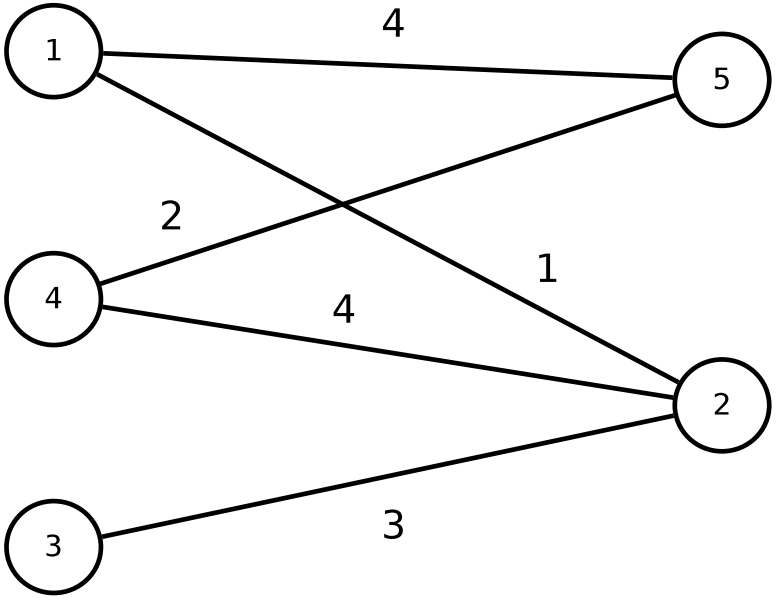
Grafos direcionados

- Grafo que possui uma aresta especial: também chamada de arco
- Grafo onde uma aresta (u, v) sai o vértice u e entra no vértice v.
- O vértice u é **adjacente** ao vértice v
- Podem haver aresta de um vértice para ele mesmo
- $(u, v) \neq (v, u)$

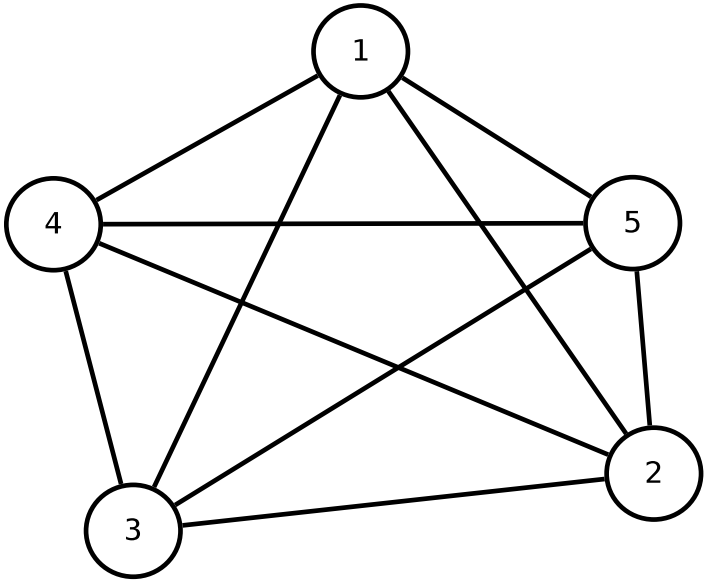


Classificações de Grafos

- **Grafo ponderado:** possui pesos associados às arestas
- **Grafo bipartido:** grafo não direcionado, no qual V pode ser particionado em dois conjunto V_1 e V_2
  - $(u, v) \in A$: sse $u \in V_1$ e $v \in V_2$, ou vice-versa
- **Grafo completo:** grafo onde todos os pares de vértices são adjacentes



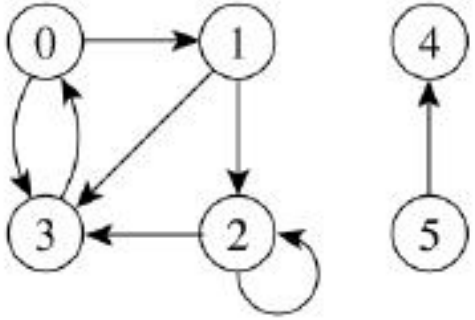
Grafo Bipartido ponderado



Grafo Completo

Representações

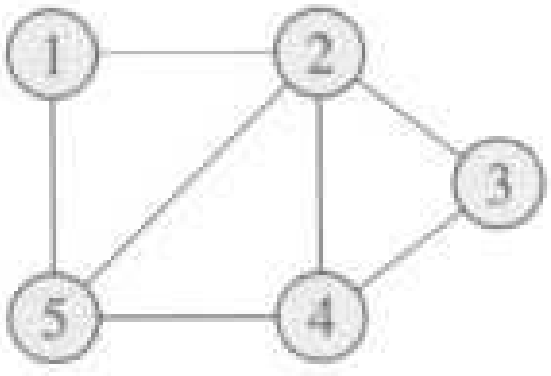
- Matriz de adjacência



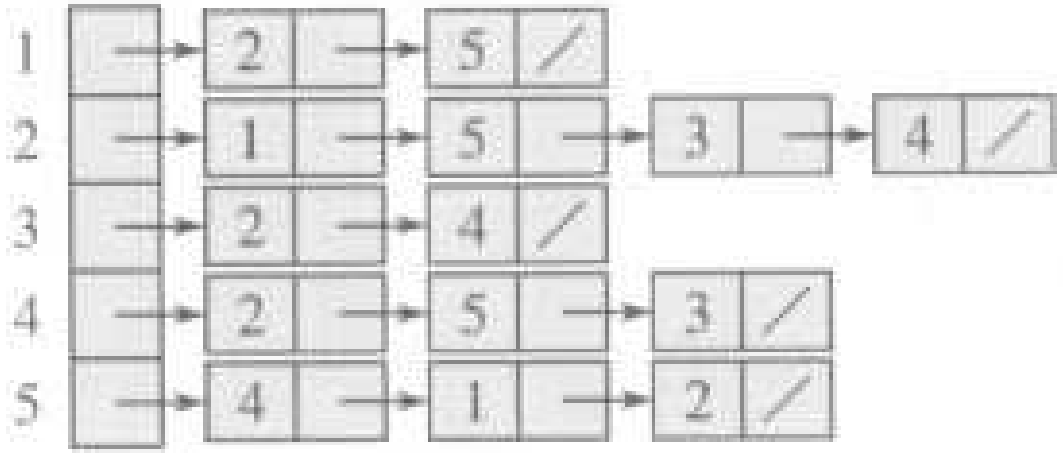
	0	1	2	3	4	5
0		1		1		
1			1	1		
2			1	1		
3	1					
4						
5						

Representações

- Lista de adjacência



(a)

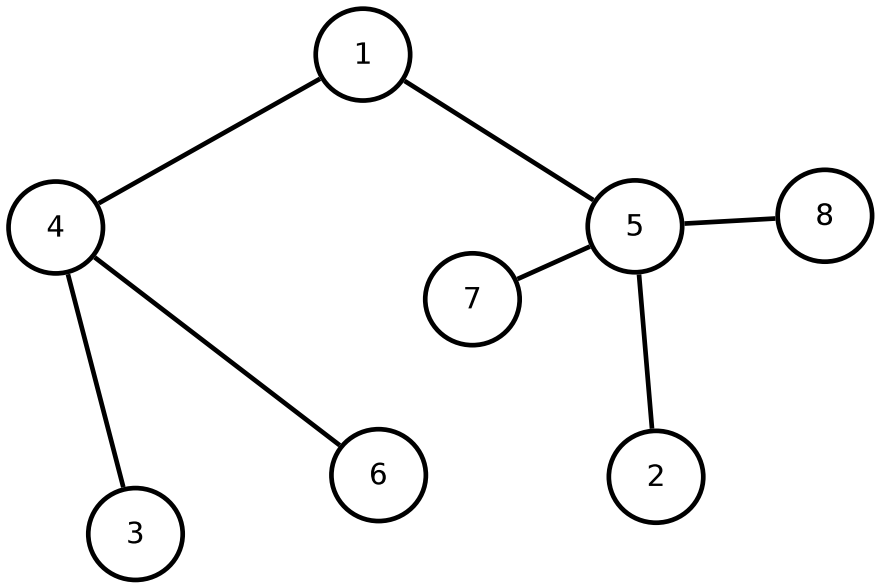


(b)

Grafos – Árvore

Árvores

- Tipo especial de Grafo onde não existem ciclos
- É um grafo conexo
- Toda árvore é um grafo, mas nem todo grafo é uma árvore



Problema da árvore geradora mínima

Considere um grafo não-direcionado, conectado e ponderado. O objetivo é encontrar o caminho mais curto de tal maneira que os arcos forneçam um caminho entre todos os pares de nós. Conhecido como *Minimum Spanning Tree* (MST)

Aplicações:

- Projeto de redes de comunicação
- Projeto de rodovias, ferrovias, etc
- Projeto de redes de transmissão de energia

Algoritmos:

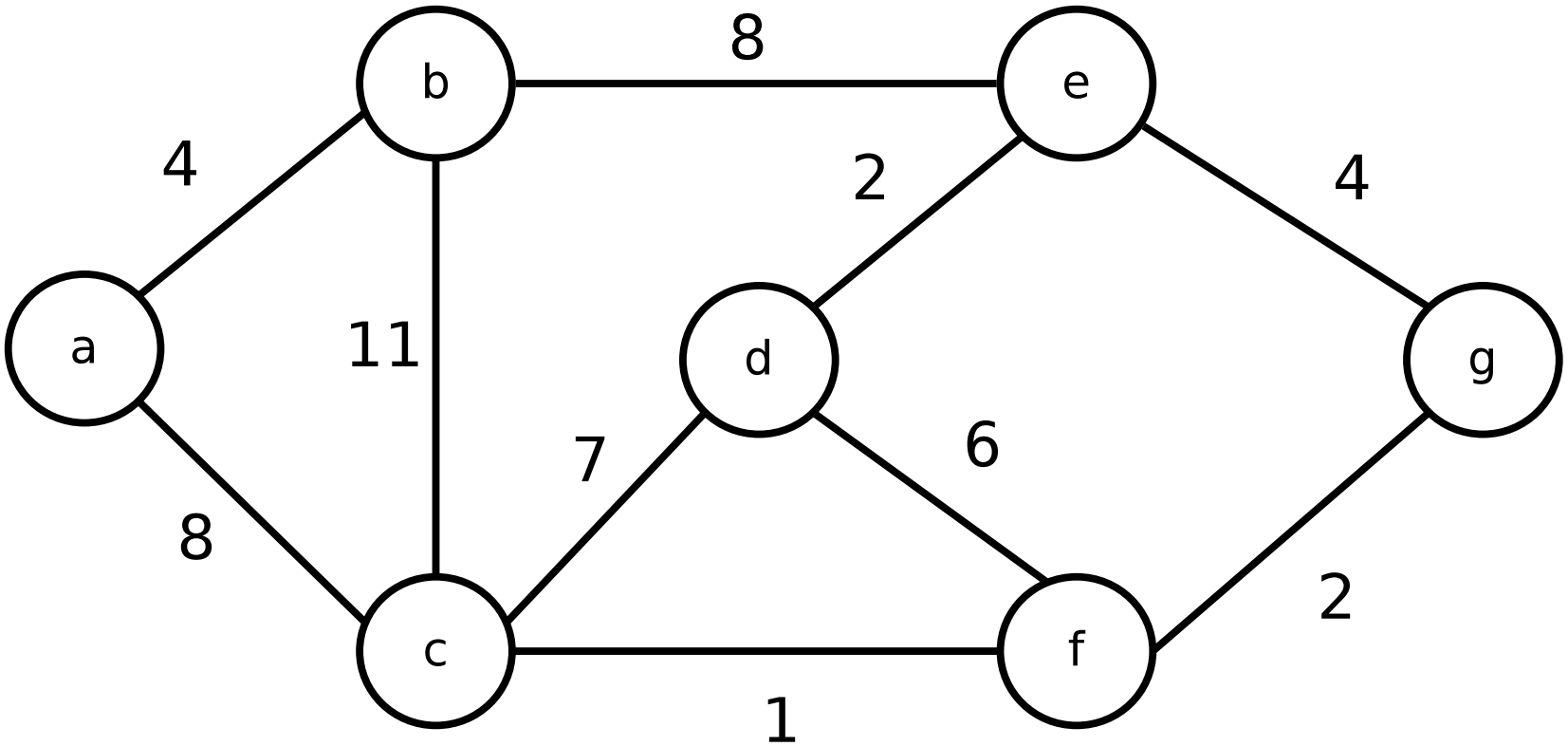
- Algoritmo de Prim
- Algoritmo de Kruskal

Problema da árvore geradora mínima

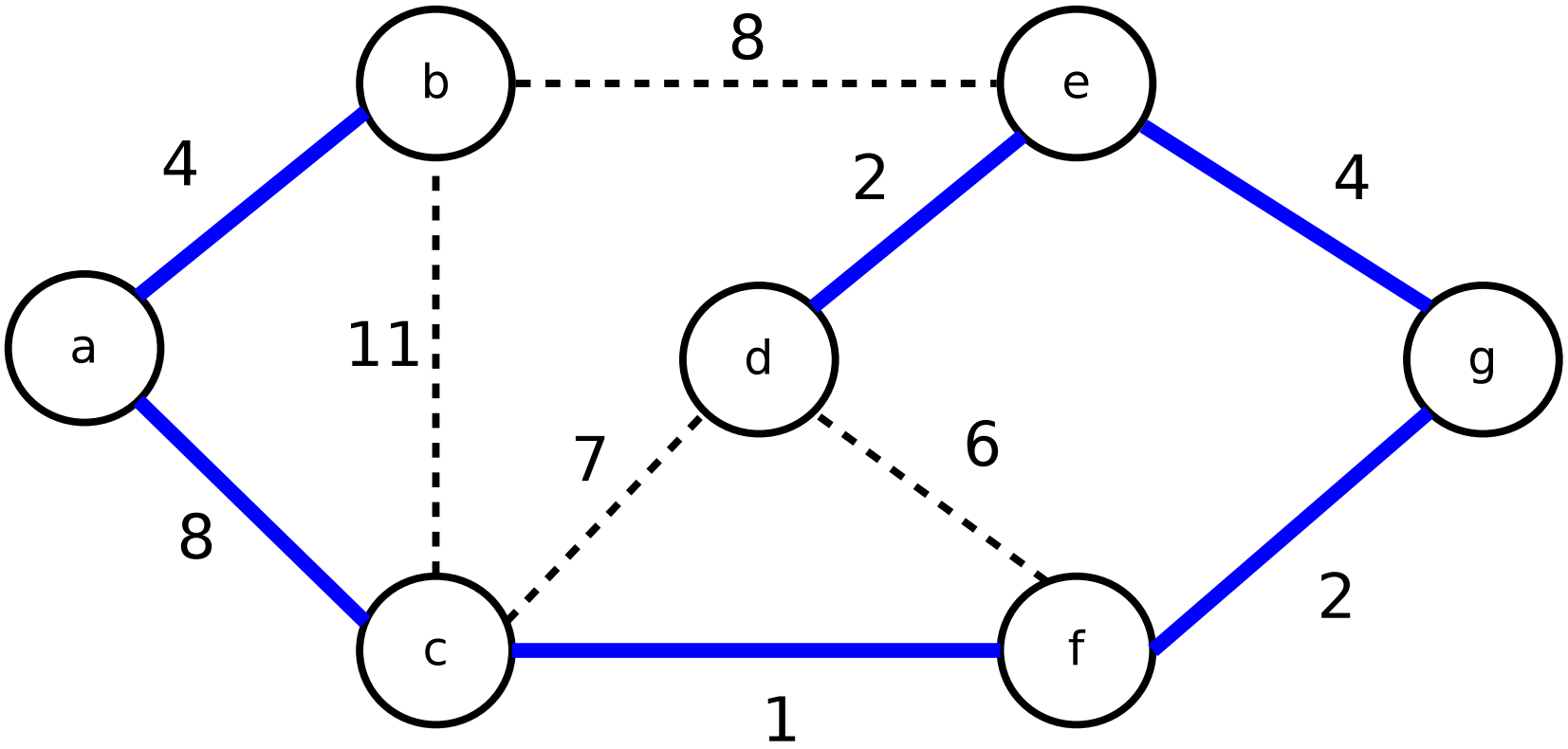
Algoritmo de Prim

- Criado em 1957
- Seja X o conjunto de vértices gerados pela MST, inicialmente vazio
- Iniciar de um vértice arbritário e crescer até que todos os vértices estejam em X
- Escolha a aresta de menor custo que liga vértices de X a um vértice que não está em $V \setminus X$
- Caminha vértice à vértice

Algoritmo de Prim – Exemplo



Algoritmo de Prim – Exemplo

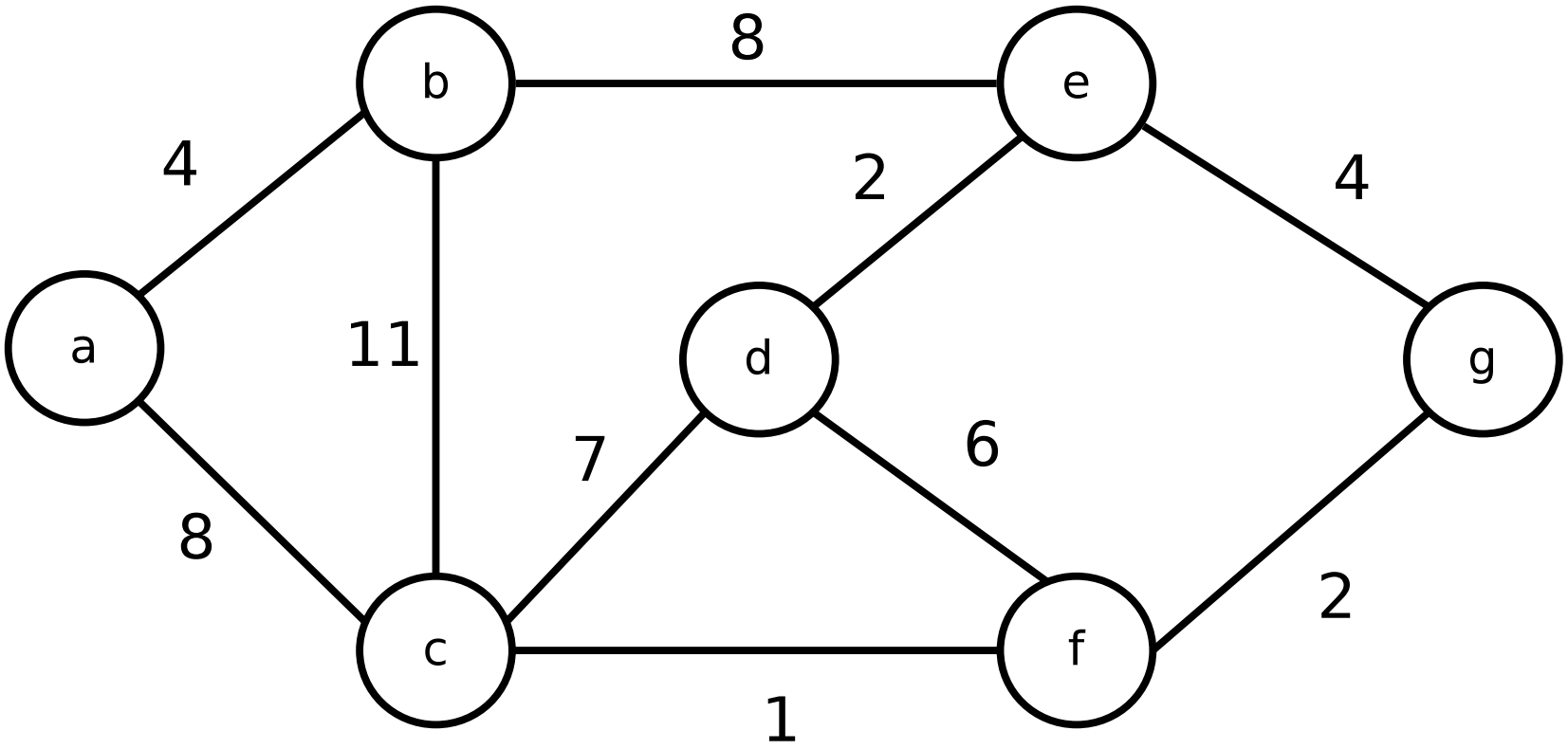


Problema da árvore geradora mínima

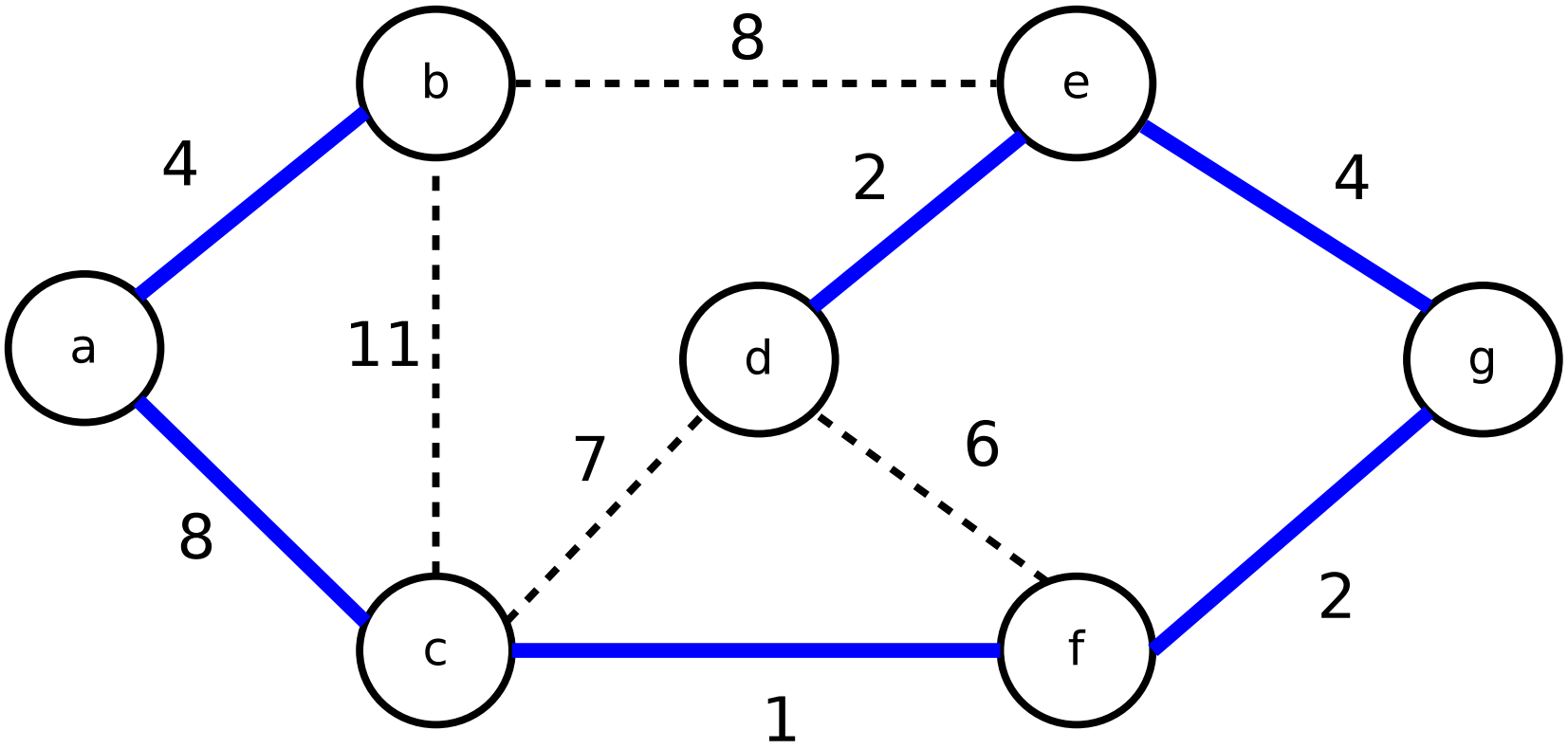
Algoritmo de Kruskal

- Criado em 1957
- Seja X o conjunto de arestas pertencentes à MST, inicialmente vazio
- Escolha a aresta de menor peso entre todas as arestas que não conectam quaisquer dois vértices em A
- Proceda aresta por aresta, de forma que ciclos não sejam formados

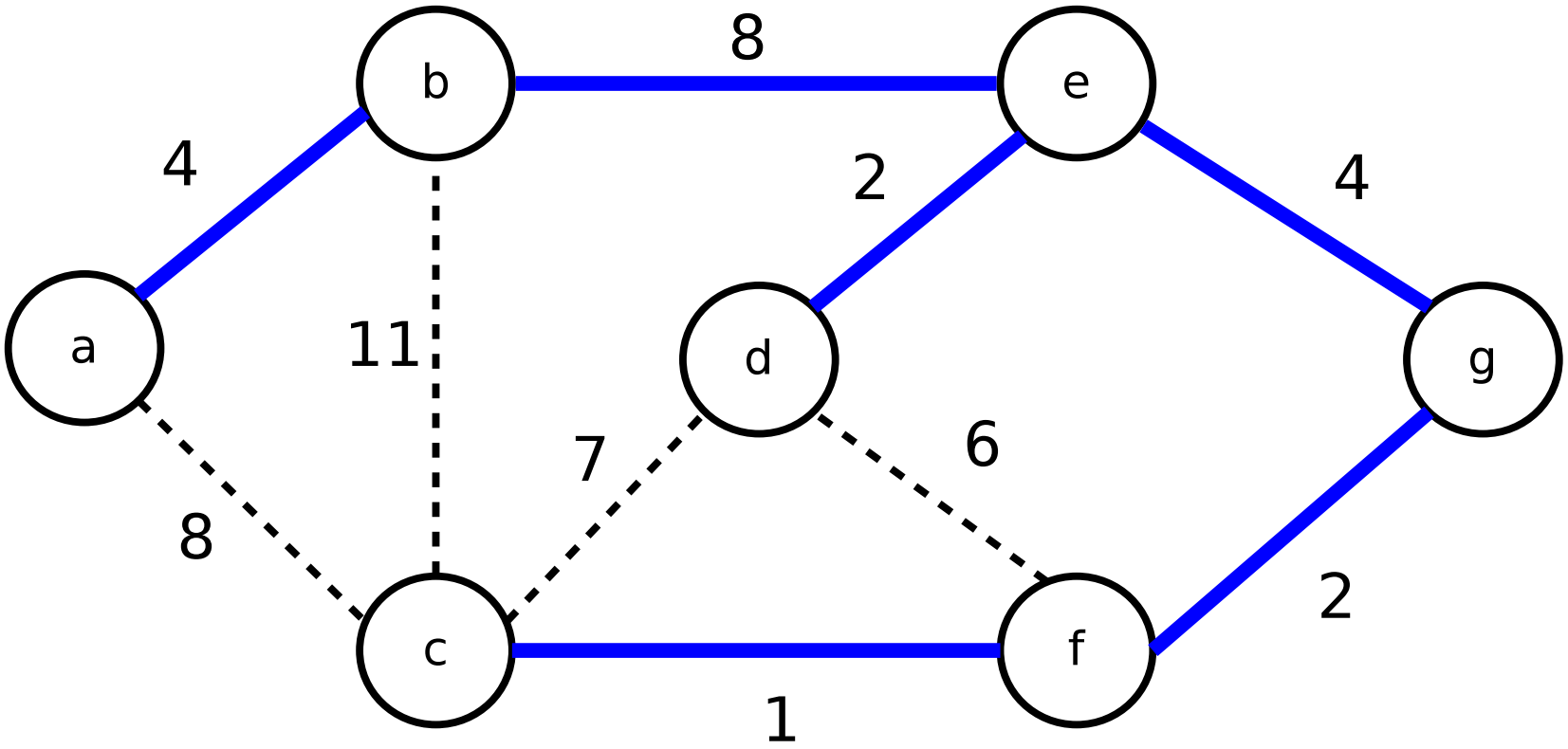
Algoritmo de Kruskal – Exemplo



Algoritmo de Kruskal – Exemplo



Algoritmo de Kruskal – Exemplo



Busca em Largura

- A busca em largura é um dos algoritmos mais simples para exploração de um grafo.
  - Dados um grafo G = (V, E) e um vértice s, chamado de **fonte**, a busca em largura sistematicamente explora as arestas de G de maneira a visitar todos os vértices alcançáveis a partir de s.

- Expande a fronteira entre vértices descobertos e não descobertos uniformemente através da largura da **fronteira**.
  - O algoritmo descobre todos os vértices a uma distância k do vértice origem antes de descobrir qualquer vértice a uma distância k + 1.

- O grafo pode ser direcionado ou não direcionado.

Busca em Largura

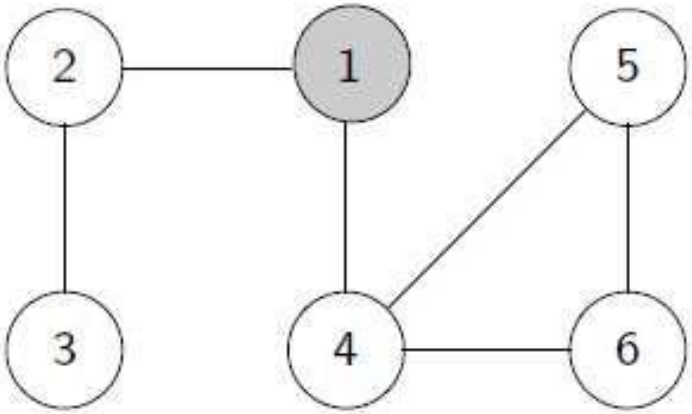
- **Algoritmo:**
  - Para controlar a busca, o algoritmo da Busca em Largura pinta cada vértice na cor branca, cinza ou preto.

  - Todos os vértices iniciam com a cor branca e podem, mais tarde, se tornar cinza e depois preto.
    - **Branca:** não visitado;
    - **Cinza:** visitado;
    - **Preta:** visitado e seus nós adjacentes visitados.

Busca em Largura

```
BuscaEmLargura(G, s)
  for each u ∈ V[G]
    c[u] ← white
    d[u] ← ∞
    π[u] ← NULL
  c[s] ← gray
  d[s] ← 0
  Q ← {s} //Queue
  while Q ≠ ∅
    u ← head[Q]
    for each v ∈ Adj[u]
      if c[v] = white
        c[v] ← gray
        d[v] ← d[u] + 1
        π[v] ← u
        enqueue(Q,v)
    dequeue(Q)
    c[u] ← black
```

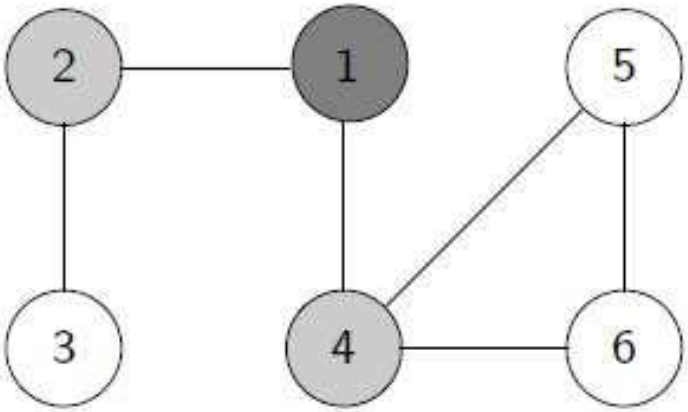
Busca em Largura



	1	2	3	4	5	6
d	0	∞	∞	∞	∞	∞
π	\	\	\	\	\	\
c	g	w	w	w	w	w
Q	1					

```
for each u ∈ V[G]
  c[u] ← white
  d[u] ← ∞
  π[u] ← NULL
c[s] ← gray
d[s] ← 0
Q ← {s} //Queue
```

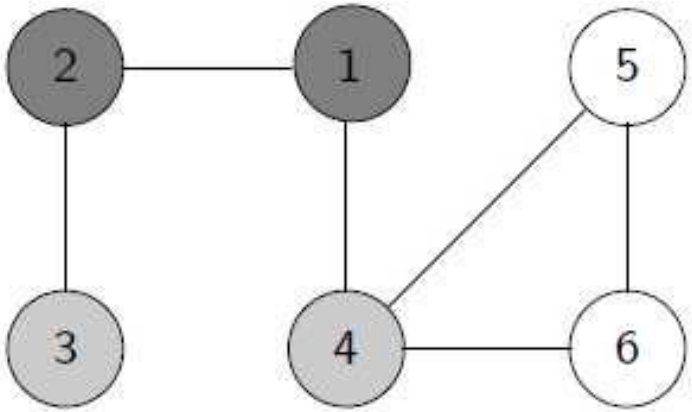
Busca em Largura



	1	2	3	4	5	6
d	0	1	∞	1	∞	∞
π	\	1	\	1	\	\
c	b	g	w	g	w	w
Q	2	4				

```
u ← head[Q]
for each v ∈ Adj[u]
  if c[v] = white
    c[v] ← gray
    d[v] ← d[u] + 1
    π[v] ← u
    enqueue(Q,v)
dequeue(Q)
c[u] ← black
```

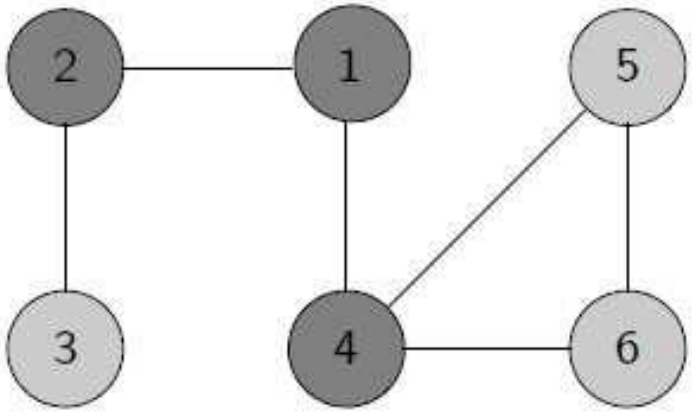
Busca em Largura



	1	2	3	4	5	6
d	0	1	2	1	∞	∞
π	\	1	2	1	\	\
c	b	b	g	g	w	w
Q	4	3				

```
u ← head[Q]
for each v ∈ Adj[u]
  if c[v] = white
    c[v] ← gray
    d[v] ← d[u] + 1
    π[v] ← u
    enqueue(Q,v)
dequeue(Q)
c[u] ← black
```

Busca em Largura



	1	2	3	4	5	6
d	0	1	2	1	2	2
π	\	1	2	1	4	4
c	b	b	g	b	g	g
Q	3	5	6			

```
u ← head[Q]
for each v ∈ Adj[u]
  if c[v] = white
    c[v] ← gray
    d[v] ← d[u] + 1
    π[v] ← u
    enqueue(Q,v)
dequeue(Q)
c[u] ← black
```

Busca em Largura – Análise

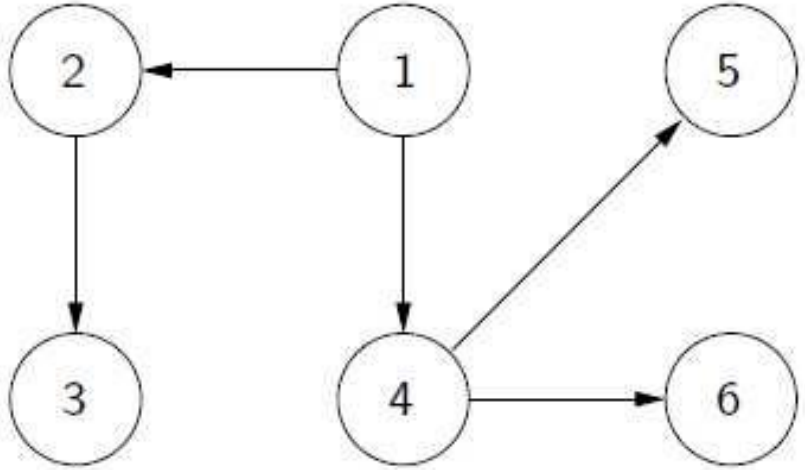
```
BuscaEmLargura(G, s)
  for each u ∈ V[G]
    c[u] ← white
    d[u] ← ∞
    π[u] ← NULL
  c[s] ← gray
  d[s] ← 0
  Q ← {s} //Queue
  while Q ≠ ∅
    u ← head[Q]
    for each v ∈ Adj[u]
      if c[v] = white
        c[v] ← gray
        d[v] ← d[u] + 1
        π[v] ← u
        enqueue(Q,v)
    dequeue(Q)
    c[u] ← black
```

- Cada vértice de V é colocado na fila Q no máximo uma vez: O(V);
- A lista de adjacência de um vértice qualquer de *u* é percorrida somente quando o vértice é removido da fila;
- A soma de todas as listas de adjacentes é O(A), então o tempo total gasto com as listas de adjacentes é O(A);
- Enfileirar e desenfileirar tem custo O(1);
- Complexidade: **O(V + A)**

Busca em Largura

- A partir de π é possível reconstruir a **árvore da busca em largura**:

	1	2	3	4	5	6
d	0	1	2	1	2	2
π	\	1	2	1	4	4
c	b	b	g	b	g	g
Q	3	5	6			



Busca em Profundidade

- A estratégia é buscar o vértice mais profundo no grafo sempre que possível:
  - As arestas são exploradas a partir do vértice *v* mais recentemente descoberto que ainda possui arestas não exploradas saindo dele.

- Quando todas as arestas adjacentes a *v* tiverem sido exploradas a busca anda para trás para explorar vértices que saem do vértice do qual *v* foi descoberto (**backtraking**).

- O algoritmo é a base para muitos outros algoritmos importantes, tais como verificação de grafos acíclicos, ordenação topológica e componentes fortemente conectados.

Busca em Profundidade

- **Algoritmo:**
  - Para controlar a busca, o algoritmo da Busca em Largura pinta cada vértice na cor branca, cinza ou preto.

  - Todos os vértices iniciam com a cor branca e podem, mais tarde, se tornar cinza e depois preto.
    - **Branca:** não visitado;
    - **Cinza:** visitado;
    - **Preta:** visitado e seus nós adjacentes visitados.

Busca em Profundidade

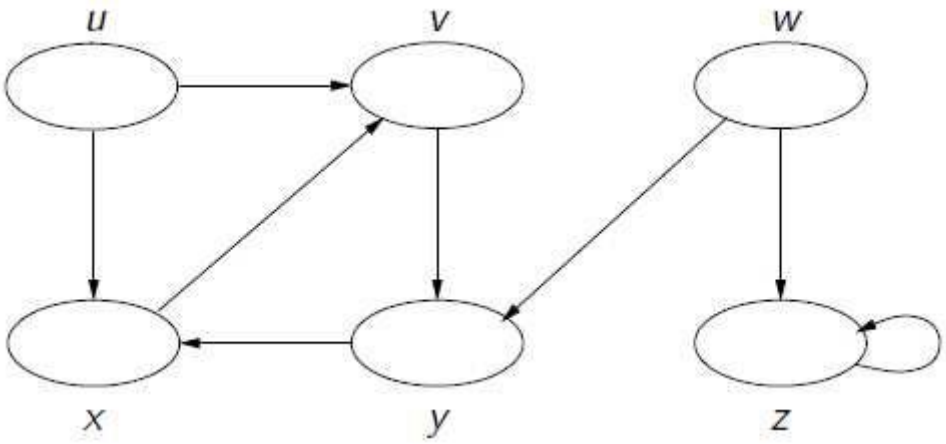
- **Algoritmo:**
  - A busca em profundidade também marca cada vértice com um *timestamp*.
  - Cada vértice tem dois *timestamps*:
    - **d[v]:** indica o instante em que *v* foi visitado (pintado com cinza);
    - **f[v]:** indica o instante em que a busca pelos vértices na lista de adjacência de *v* foi completada (pintado de preto).

Busca em Profundidade

```
BuscaEmProfundidade(G)
  for each u ∈ V[G]
    c[u] ← white
    π[u] ← NULL
  time ← 0
  for each u ∈ V[G]
    if c[u] = white
      visita(u)
```

```
visita(u)
  c[u] ← gray
  d[u] ← time ← time + 1
  for each v ∈ Adj[u]
    if c[v] = white
      π[v] ← u
      visita(v)
  c[u] ← black
  f[u] ← time ← time + 1
```

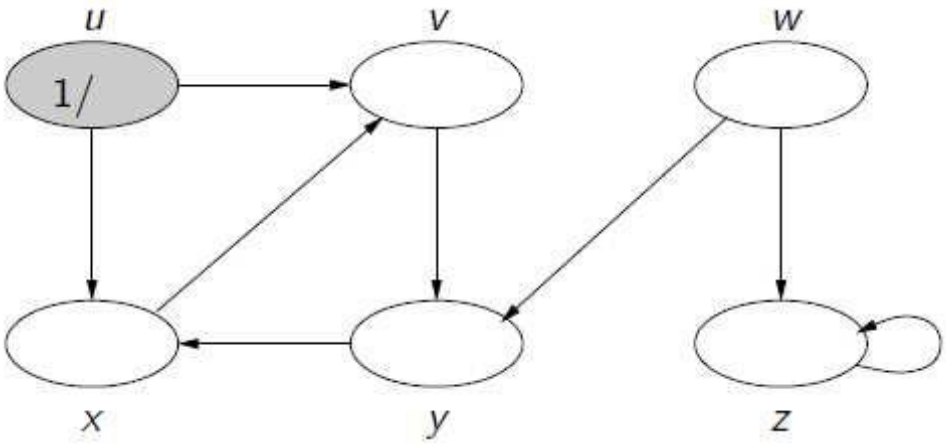
Busca em Profundidade




```
for each u ∈ V[G]
  c[u] ← white
  π[u] ← NULL
time ← 0
for each u ∈ V[G]
  if c[u] = white
    visita(u)
```

	u	v	y	x	w	z
c	w	w	w	w	w	w
π	/	/	/	/	/	/
d						
f						

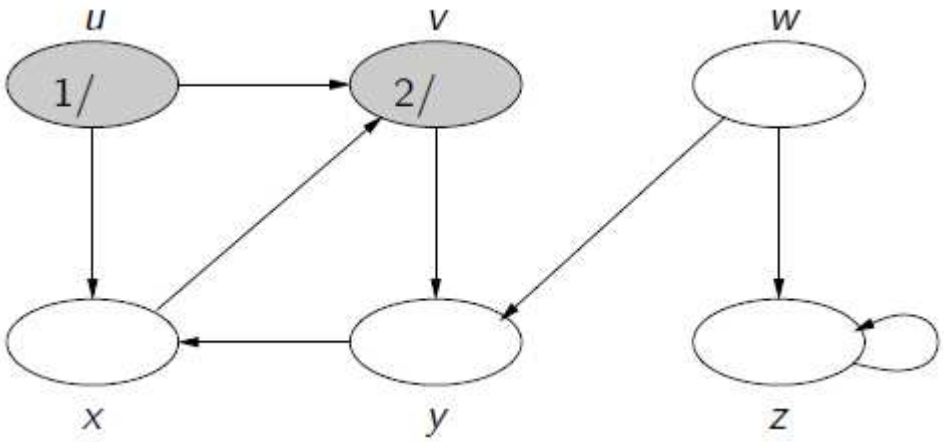
Busca em Profundidade




```
c[u] ← gray
d[u] ← time ← time + 1
for each v ∈ Adj[u]
  if c[v] = white
    π[v] ← u
    visita(v)
c[u] ← black
f[u] ← time ← time + 1
```

	u	v	y	x	w	z
c	g	w	w	w	w	w
π	/	/	/	/	/	/
d	1					
f						

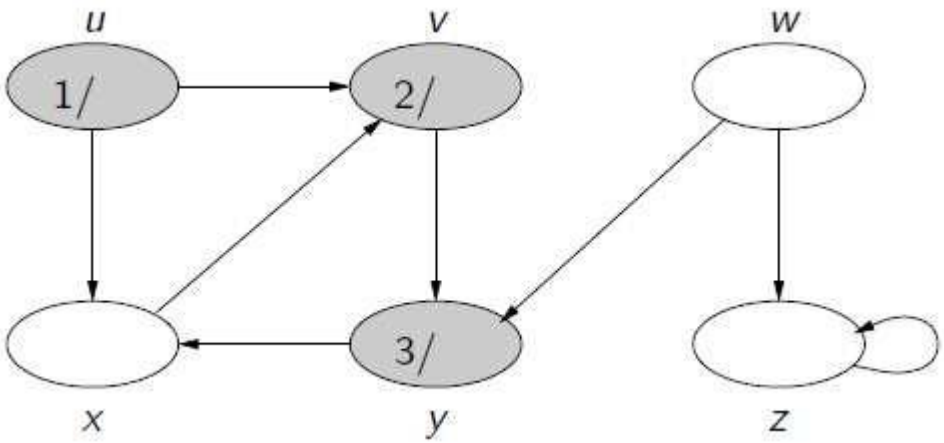
Busca em Profundidade




```
c[u] ← gray
d[u] ← time ← time + 1
for each v ∈ Adj[u]
  if c[v] = white
    π[v] ← u
    visita(v)
c[u] ← black
f[u] ← time ← time + 1
```

	u	v	y	x	w	z
c	g	g	w	w	w	w
π	/	u	/	/	/	/
d	1	2				
f						

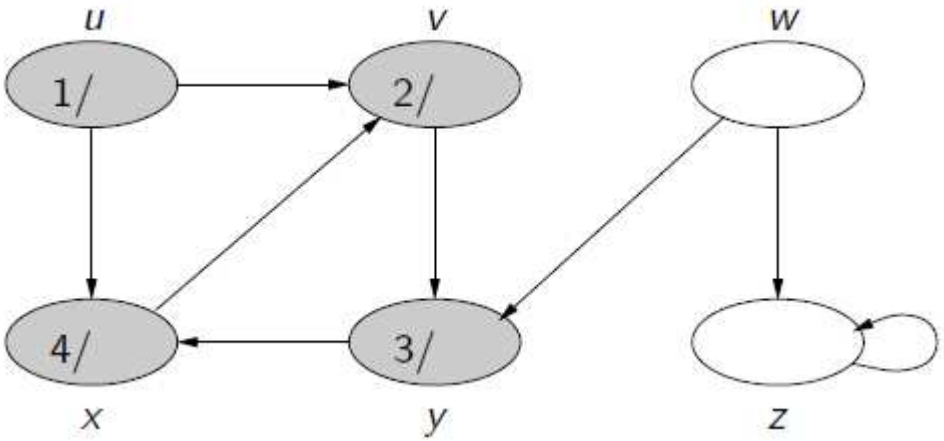
Busca em Profundidade




```
c[u] ← gray
d[u] ← time ← time + 1
for each v ∈ Adj[u]
  if c[v] = white
    π[v] ← u
    visita(v)
c[u] ← black
f[u] ← time ← time + 1
```

	u	v	y	x	w	z
c	g	g	g	w	w	w
π	/	u	v	/	/	/
d	1	2	3			
f						

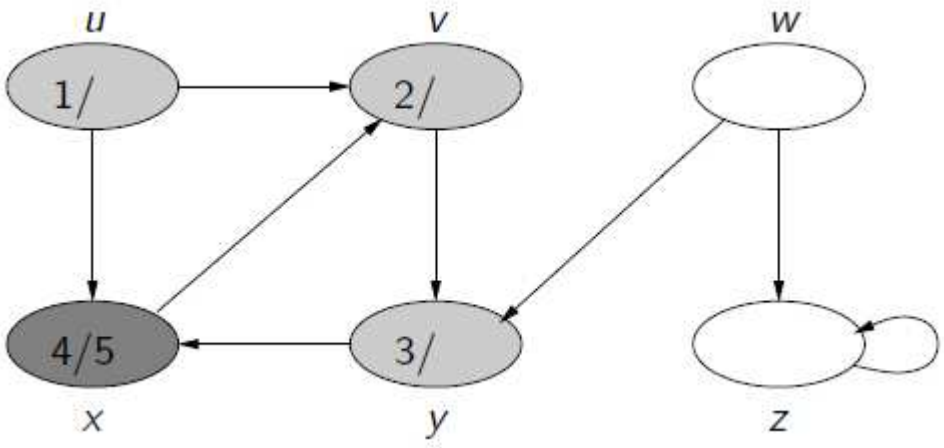
Busca em Profundidade




```
c[u] ← gray
d[u] ← time ← time + 1
for each v ∈ Adj[u]
  if c[v] = white
    π[v] ← u
    visita(v)
c[u] ← black
f[u] ← time ← time + 1
```

	u	v	y	x	w	z
c	g	g	g	g	w	w
π	/	u	v	y	/	/
d	1	2	3	4		
f						

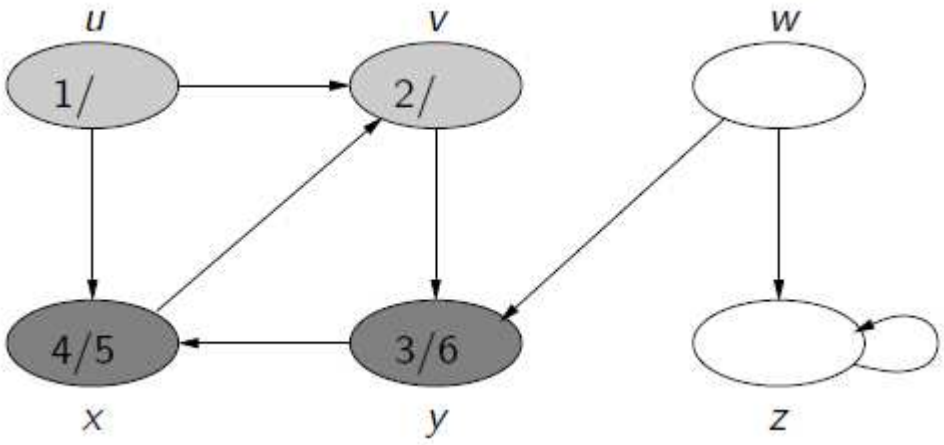
Busca em Profundidade




```
c[u] ← gray
d[u] ← time ← time + 1
for each v ∈ Adj[u]
  if c[v] = white
    π[v] ← u
    visita(v)
c[u] ← black
f[u] ← time ← time + 1
```

	u	v	y	x	w	z
c	g	g	g	b	w	w
π	/	u	v	y	/	/
d	1	2	3	4		
f				5		

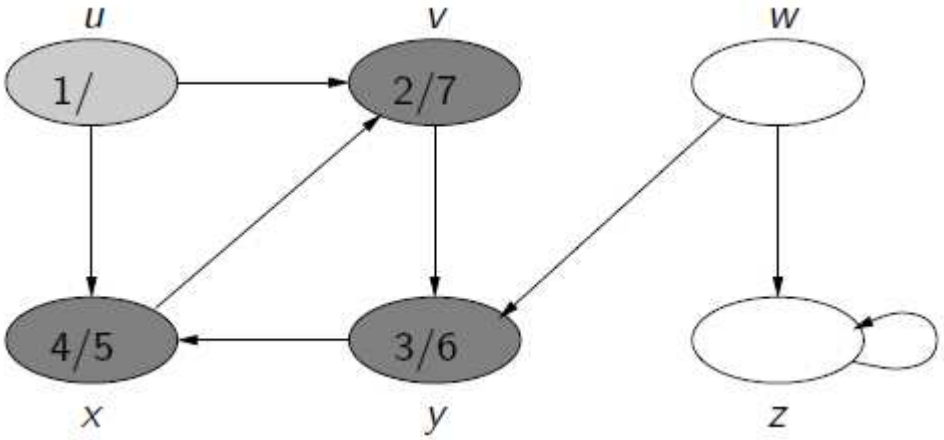
Busca em Profundidade




```
c[u] ← gray
d[u] ← time ← time + 1
for each v ∈ Adj[u]
  if c[v] = white
    π[v] ← u
    visita(v)
c[u] ← black
f[u] ← time ← time + 1
```

	u	v	y	x	w	z
c	g	g	b	b	w	w
π	/	u	v	y	/	/
d	1	2	3	4		
f			6	5		

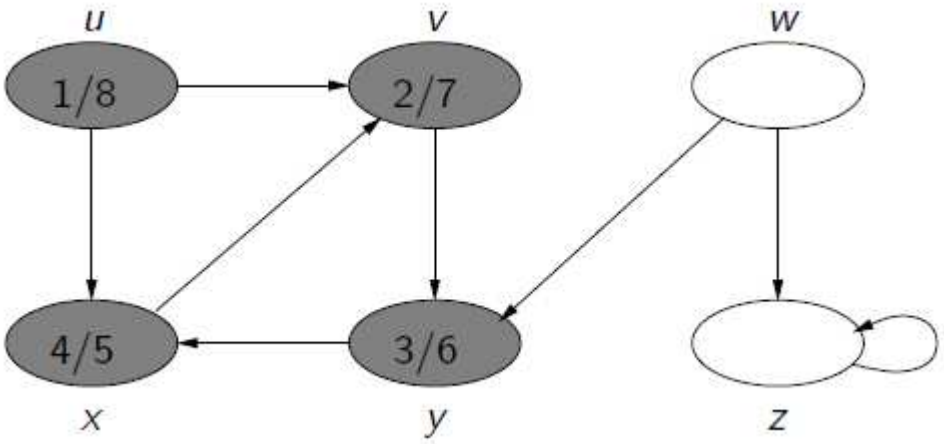
Busca em Profundidade




```
c[u] ← gray
d[u] ← time ← time + 1
for each v ∈ Adj[u]
  if c[v] = white
    π[v] ← u
    visita(v)
c[u] ← black
f[u] ← time ← time + 1
```

	u	v	y	x	w	z
c	g	b	b	b	w	w
π	/	u	v	y	/	/
d	1	2	3	4		
f		7	6	5		

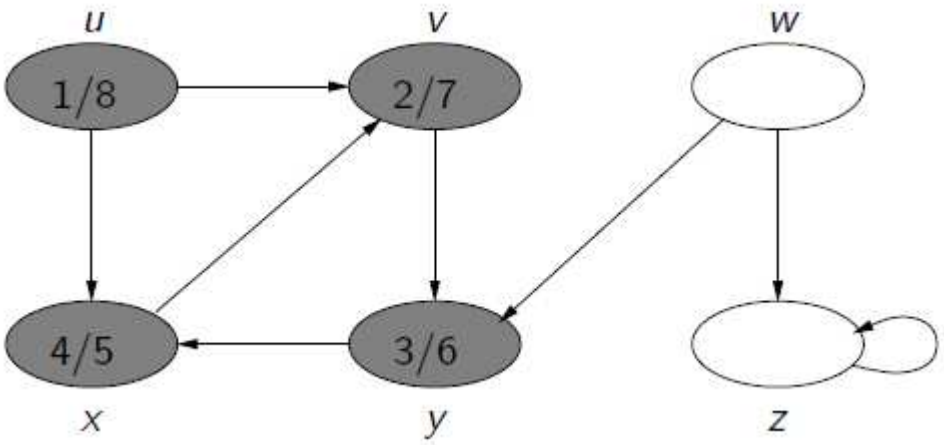
Busca em Profundidade




```
c[u] ← gray
d[u] ← time ← time + 1
for each v ∈ Adj[u]
  if c[v] = white
    π[v] ← u
    visita(v)
c[u] ← black
f[u] ← time ← time + 1
```

	u	v	y	x	w	z
c	b	b	b	b	w	w
π	/	u	v	y	/	/
d	1	2	3	4		
f	8	7	6	5		

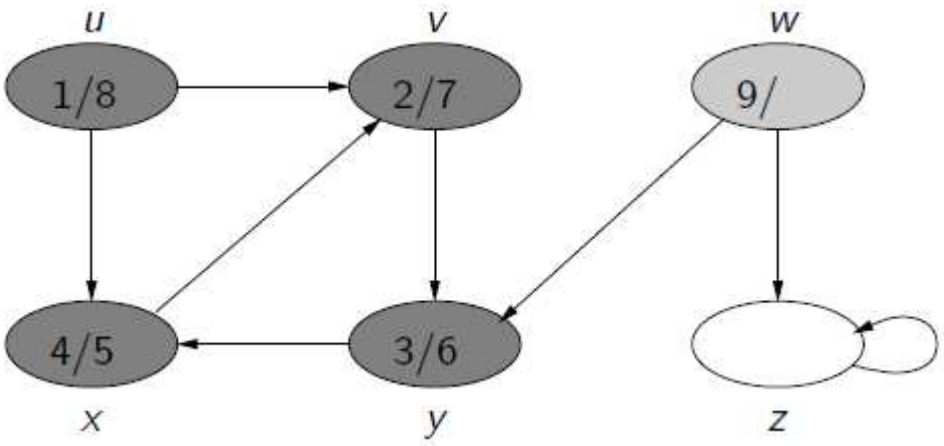
Busca em Profundidade




```
for each u ∈ V[G]
  c[u] ← white
  π[u] ← NULL
time ← 0
for each u ∈ V[G]
  if c[u] = white
    visita(u)
```

	u	v	y	x	w	z
c	b	b	b	b	w	w
π	/	u	v	y	/	/
d	1	2	3	4		
f	8	7	6	5		

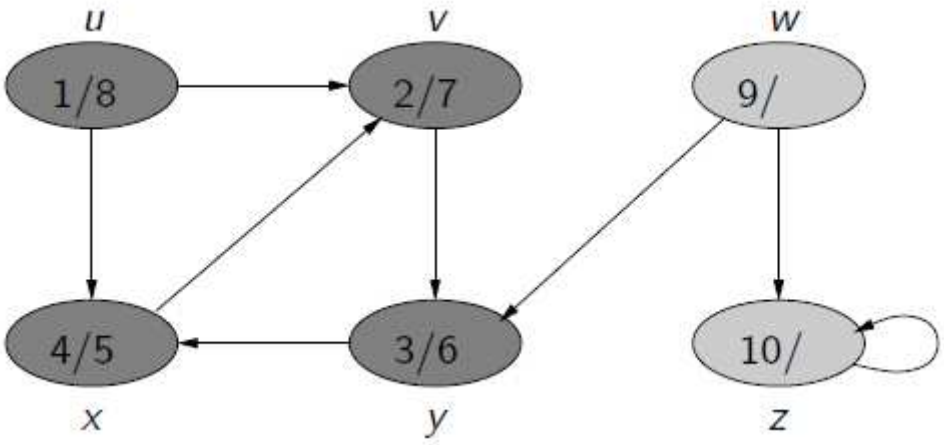
Busca em Profundidade




```
c[u] ← gray
d[u] ← time ← time + 1
for each v ∈ Adj[u]
  if c[v] = white
    π[v] ← u
    visita(v)
c[u] ← black
f[u] ← time ← time + 1
```

	u	v	y	x	w	z
c	b	b	b	b	g	w
π	/	u	v	y	/	/
d	1	2	3	4	9	
f	8	7	6	5		

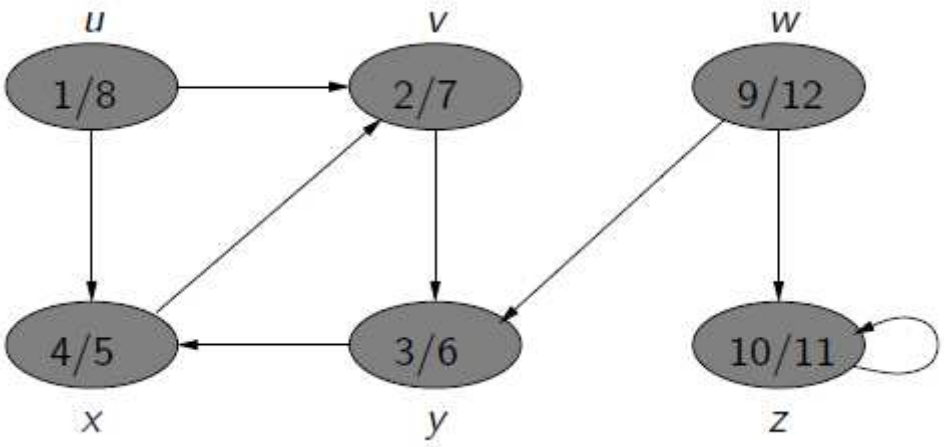
Busca em Profundidade




```
c[u] ← gray
d[u] ← time ← time + 1
for each v ∈ Adj[u]
  if c[v] = white
    π[v] ← u
    visita(v)
c[u] ← black
f[u] ← time ← time + 1
```

	u	v	y	x	w	z
c	b	b	b	b	g	g
π	/	u	v	y	/	w
d	1	2	3	4	9	10
f	8	7	6	5		

Busca em Profundidade




```
c[u] ← gray
d[u] ← time ← time + 1
for each v ∈ Adj[u]
  if c[v] = white
    π[v] ← u
    visita(v)
c[u] ← black
f[u] ← time ← time + 1
```

	u	v	y	x	w	z
c	b	b	b	b	b	b
π	/	u	v	y	/	w
d	1	2	3	4	9	10
f	8	7	6	5	11	12

Busca em Profundidade – Análise

```
BuscaEmProfundidade(G)
  for each u ∈ V[G]
    c[u] ← white
    π[u] ← NULL
  time ← 0
  for each u ∈ V[G]
    if c[u] = white
      visita(u)
```

```
visita(u)
  c[u] ← gray
  d[u] ← time ← time + 1
  for each v ∈ Adj[u]
    if c[v] = white
      π[v] ← u
      visita(v)
  c[u] ← black
  f[u] ← time ← time + 1
```

- O procedimento `visita` é chamado exatamente uma vez ara cada vértice *u* ∈ V, isso porque `visita` é chamado apenas para vértices brancos e a primeira ação é pintar o vértice de cinza: O(V);
- O loop principal de `visita(u)` tem complexidade O(A);
- Complexidade: **O(V + A)**

Busca em Profundidade

- **Classificação de Arestas:**
  - **Arestas de árvore:** são arestas de uma árvore de busca em profundidade. A aresta (u, v) é uma aresta de árvore se v foi descoberto pela primeira vez ao percorrer a aresta (u, v);
  - **Arestas de retorno:** conectam um vértice u com um antecessor v em uma árvore de busca em profundidade (inclui self-loops);
  - **Arestas de avanço:** não pertencem à árvore de busca em profundidade mas conectam um vértice a um descendente que pertence à árvore de busca em profundidade;
  - **Arestas de cruzamento:** podem conectar vértices na mesma árvore de busca em profundidade, ou em duas árvores diferentes.

Busca em Profundidade

- Classificação de arestas pode ser útil para derivar outros algoritmos.

- Na busca em profundidade cada aresta pode ser classificada pela cor do vértice que é alcançado pela primeira vez:
  - Branco indica uma **aresta de árvore**.
  - Cinza indica uma **aresta de retorno**.
  - Preto indica uma **aresta de avanço** quando u é descoberto antes de v ou uma **aresta de cruzamento** caso contrário.