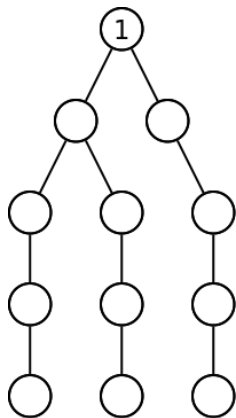


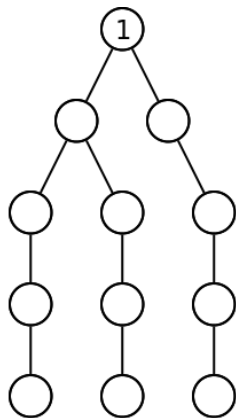
Pesquisa em Grafos

- Uma das tarefas mais importantes é saber **percorrer** um grafo, ou seja **passar por todos os nós** usando para isso as **ligações entre eles**
- Chama-se a isto fazer uma **pesquisa** no grafo
- Existem dois tipos básicos de pesquisa que variam na **ordem em que percorrem os nós**:
 - ▶ **Pesquisa em Profundidade** (*Depth-First Search - DFS*)
Pesquisar todo o grafo ligado a um nó adjacente antes de entrar no nó adjacente seguinte
 - ▶ **Pesquisa em Largura** (*Breadth-First Search - BFS*)
Pesquisar os nós por ordem crescente da sua distância em termos de número de arestas para lá chegar

Pesquisa em Grafos

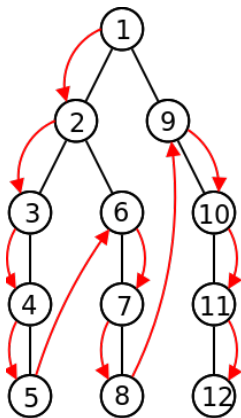


Pesquisa em Profundidade

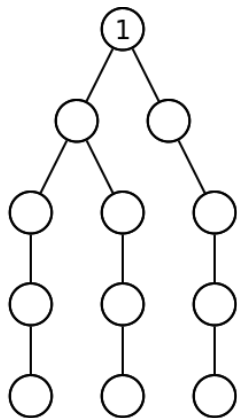


Pesquisa em Largura

Pesquisa em Grafos

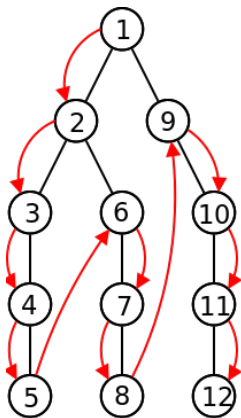


**Pesquisa em
Profundidade**

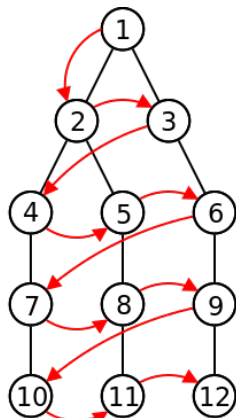


**Pesquisa em
Largura**

Pesquisa em Grafos



**Pesquisa em
Profundidade**



**Pesquisa em
Largura**

- Na sua essência, DFS e BFS fazem o "mesmo":
percorrer todos os nós
- Quando usar um ou outro depende do problema e da **ordem em que nos interessa percorrer os nós**
- Vamos ver como **implementar** ambos e dar exemplos de várias aplicações

Pesquisa em Profundidade

O "esqueleto" de uma pesquisa em profundidade:

DFS (versão recursiva)

dfs(nó v):

marcar v como visitado

Para todos os nós w adjacentes a v **fazer**

Se w ainda não foi visitado **então**

dfs(w)

Complexidade:

- Temporal:
 - ▶ Lista de Adjacências: $\mathcal{O}(|V| + |E|)$
 - ▶ Matriz de Adjacências: $\mathcal{O}(|V|^2)$
- Espacial: $\mathcal{O}(|V|)$

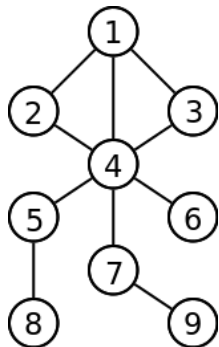
Pesquisa em Profundidade

Vamos ver mesmo um programa exemplo a ser feito:
(programa feito na aula, terão código equivalente no guião #08)

Imagine que um grafo (não dirigido) é dado como:

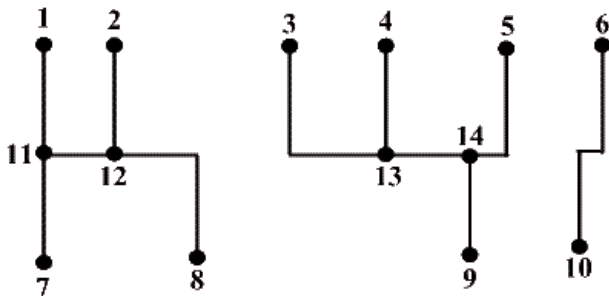
```
num_nos  
num_arestas  
origem1 fim1  
origem2 fim2  
...  
origem|E| fim|E|
```

```
9  
10  
1 2  
1 3  
1 4  
2 4  
3 4  
4 5  
4 6  
4 7  
5 8  
7 9
```



Componentes Conexos

- Descobrir o número de **componentes conexos** de um grafo G
- **Exemplo:** o grafo seguinte tem **3 componentes conexos**



Componentes Conexos

O "esqueleto" de um programa para resolver:

Descobrir componentes conexos

contador \leftarrow 0

marcar todos os nós como **não visitados**

Para todos os nós v do grafo **fazer**

Se v ainda não foi visitado **então**

contador \leftarrow contador + 1

dfs(v)

escrever(*contador*)

Complexidade temporal:

- Lista de Adjacências: $\mathcal{O}(|V| + |E|)$
- Matriz de Adjacências: $\mathcal{O}(|V|^2)$

Grafos Implícitos

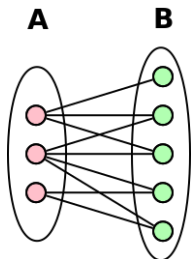
- Nem sempre é necessário guardar explicitamente o grafo.
- **Exemplo:** descobrir o número de "blobs" (manchas conexas) numa matriz. Duas células são adjacentes se estiverem ligadas vertical ou horizontalmente.

#.##..##		1.22..33
#.....##		1.....33
...##...	--> 4 blobs -->	...44...
...##...		...44...

- Para resolver basta fazer um $dfs(x, y)$ para visitar a posição (x, y) e onde os adjacentes são $(x + 1, y)$, $(x - 1, y)$, $(x, y + 1)$ e $(x, y - 1)$
- Chamar um DFS para ir "colorindo" as componentes conexas é conhecido como fazer um **Flood Fill**.

Grafos Bipartidos

- Um **grafo bipartido** é um grafo onde é possível dividir os nós em dois grupos A e B tal que cada aresta liga um nó de A a um nó de B:
 - ▶ Não podem existir arestas de A para A
 - ▶ Não podem existir arestas de B para B

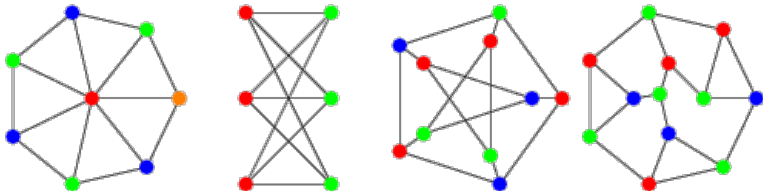


- Muitos grafos reais são deste tipo. Alguns exemplos:
 - ▶ Produtos e Compradores
 - ▶ Filmes e Actores
 - ▶ Livros e Autores
 - ▶ ...

Grafos Bipartidos

Colorindo Grafos

- O problema de **graph coloring** implica descobrir uma alocação de cores aos nós de um grafo tal que nunca aconteça que dois nós vizinhos tenham a mesma cor



- Dado um grafo qual o menor número de cores que precisamos?
(o *chromatic number* de um grafo)
 - ▶ Para um grafo geral este problema é muito complicado e não existem soluções polinomiais.
(este é um dos 21 problemas NP-completos originais)

Grafos Bipartidos

Algoritmo com DFS

- Saber se um grafo é bipartido é um caso particular de coloração
- Grafo bipartido \leftrightarrow **é possível colorir com duas cores?**
- Podemos adaptar um *dfs* para resolver:

Algoritmo para testar se um grafo é bipartido

Fazer um dfs a partir de um nó u e colorir esse nó com uma cor

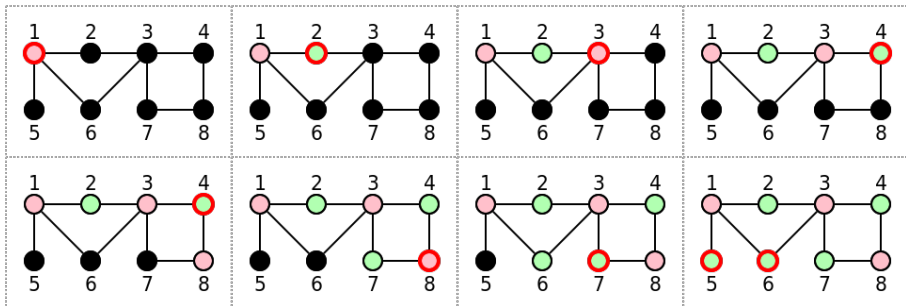
Para cada nó w vizinho de u :

- Se w não foi visitado, fazer $dfs(w)$ e pintar w com cor diferente de v
- Se w já foi visitado, verificar se cor é diferente
 - ▶ Se cor for igual, grafo não é bipartido!

Grafos Bipartidos

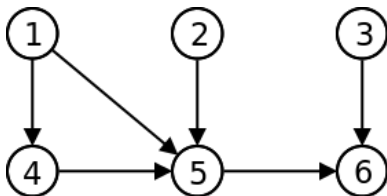
Exemplo de funcionamento do algoritmo com DFS

- Nó preto: não visitado
- Nó vermelho: grupo A
- Nó verde: grupo B



Ordenação Topológica

- Dado um grafo dirigido e acíclico G , descobrir uma ordenação dos nós, tal que nessa ordem u vem antes de v se e só se não existe uma aresta (v, u)
- **Exemplo:** Para o grafo de baixo, uma possível ordenação topológica seria: 1, 2, 3, 4, 5, 6 (ou 1, 4, 2, 5, 3, 6 - existem ainda outras ordenações topológicas possíveis)



Um exemplo clássico de aplicação é decidir por qual ordem executar tarefas que têm precedências.

Ordenação Topológica

- Como resolver este problema com DFS? Qual a relação da ordem em que um DFS visita os nós com uma ordenação topológica?

Ordenação Topológica - $\mathcal{O}(|V| + |E|)$ (lista) ou $\mathcal{O}(|V|^2)$ (matriz)

ordem ← lista vazia

marcar todos os nós como **não visitados**

Para todos os nós v do grafo **fazer**

Se v ainda não foi visitado **então**

dfs(v)

escrever(ordem)

dfs(nó v):

marcar v como visitado

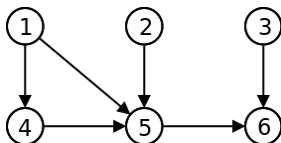
Para todos os nós w adjacentes a v **fazer**

Se w ainda não foi visitado **então**

dfs(w)

*adicionar v ao início da lista *ordem**

Ordenação Topológica



Exemplo de execução:

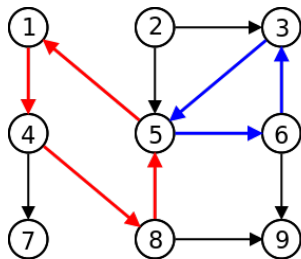
- $ordem = \emptyset$
- entra dfs(1) | $ordem = \emptyset$
- entra dfs(4) | $ordem = \emptyset$
- entra dfs(5) | $ordem = \emptyset$
- entra dfs(6) | $ordem = \emptyset$
- sai dfs(6) | $ordem = 6$
- sai dfs(5) | $ordem = 5, 6$
- sai dfs(4) | $ordem = 4, 5, 6$
- sai dfs(1) | $ordem = 1, 4, 5, 6$
- entra dfs(2) | $ordem = 1, 4, 5, 6$
- sai dfs(2) | $ordem = 2, 1, 4, 5, 6$
- entra dfs(3) | $ordem = 2, 1, 4, 5, 6$
- sai dfs(3) | $ordem = 3, 2, 1, 4, 5, 6$
- $ordem = 3, 2, 1, 4, 5, 6$

Ordenação Topológica

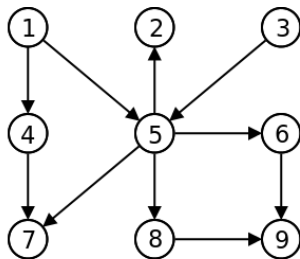
- A complexidade é $\mathcal{O}(|V| + |E|)$ (com lista de adjacências) porque só passamos uma vez por cada nó e por cada aresta.
- Se implementarem (de forma "naive") a lista com arrays, a **operação "inserir no início"** pode custar $\mathcal{O}(|V|)$, fazendo com que o algoritmo passe a demorar $\mathcal{O}(|V|^2)$
- Uma alternativa para usar arrays é **adicionar ao final** (em $\mathcal{O}(1)$) e depois basta imprimir a **ordem inversa** da calculada!
- Um algoritmo sem DFS seria, de forma **greedy**, procurar um nó com grau de entrada igual a zero, adicioná-lo à ordem e depois retirar as suas arestas, repetindo depois o mesmo algoritmo para selecionar o próximo nó.
 - ▶ Uma implementação "naive" deste algoritmo demoraria $\mathcal{O}(|V|^2)$ ($|V|$ vezes procurar um mínimo entre todos os nós ainda não adicionados)

Detecção de Ciclos

- Descobrir se grafo (dirigido) G é **acíclico** (não contém ciclos)
- **Exemplo:** o grafo da esquerda contém um ciclo, o grafo da direita não



Grafo com Ciclos



Grafo acíclico

Detecção de Ciclos

Vamos usar 3 "cores":

- **Branco** - Nó não visitado
- **Cinzento** - Nó a ser visitado (ainda estamos a explorar descendentes)
- **Preto** - Nó já visitado (já visitamos todos os descendentes)

Detecção de Ciclos - $\mathcal{O}(|V| + |E|)$ (lista) ou $\mathcal{O}(|V|^2)$ (matriz)

$\text{cor}[v \in V] \leftarrow \text{branco}$

Para todos os nós v do grafo **fazer**

Se $\text{cor}[v] = \text{branco}$ **então**
 $\text{dfs}(v)$

dfs(nó v):

$\text{cor}[v] \leftarrow \text{cinzento}$

Para todos os nós w adjacentes a v **fazer**

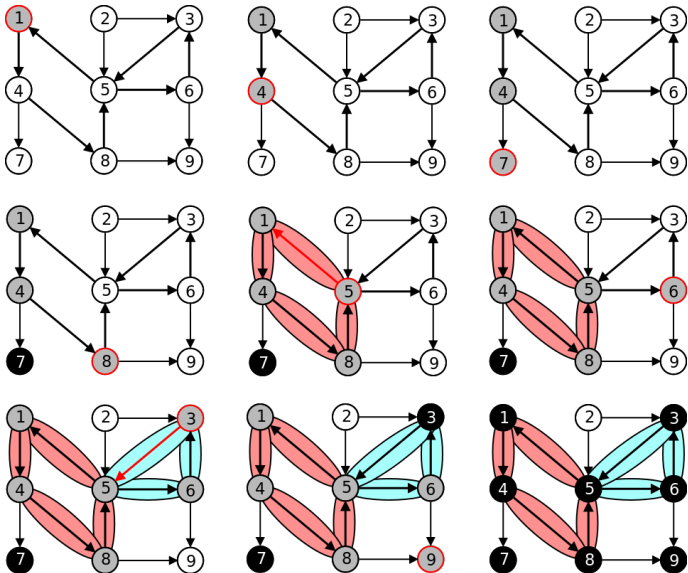
Se $\text{cor}[w] = \text{cinzento}$ **então**
 escrever("Ciclo encontrado!")

Senão se $\text{cor}[w] = \text{branco}$ **então**
 $\text{dfs}(w)$

$\text{cor}[v] \leftarrow \text{preto}$

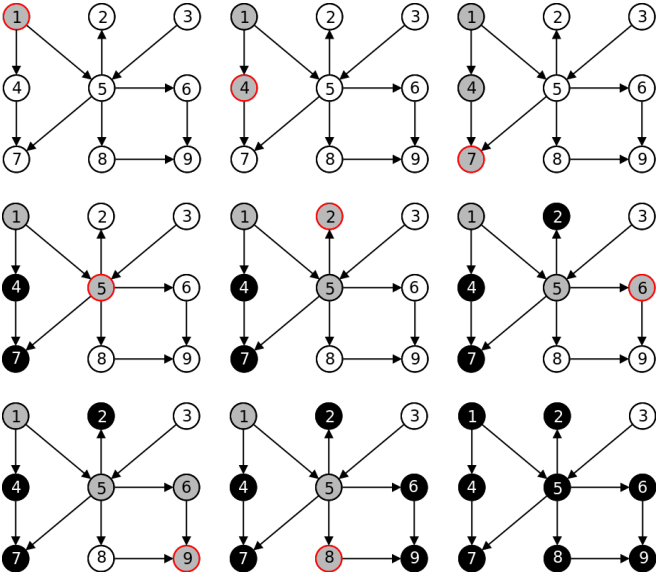
Deteção de Ciclos

Exemplo de Execução (começando no nó 1) - Grafo com 2 ciclos



Deteção de Ciclos

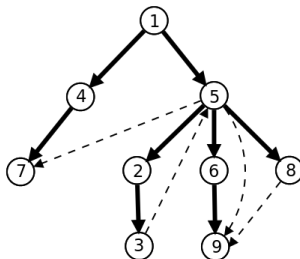
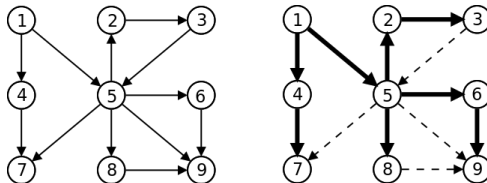
Exemplo de Execução (começando no nó 1) - Grafo acíclico



Classificação de Arestas por um DFS

Uma outra "visão" de DFS

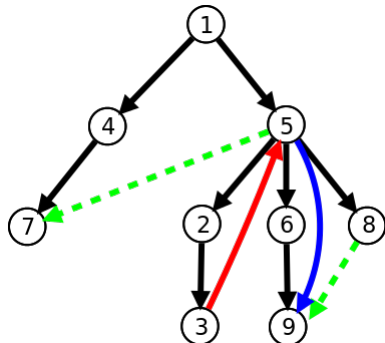
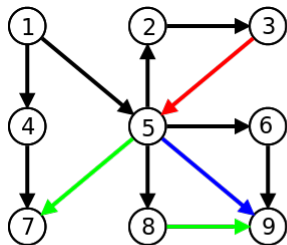
- Uma pesquisa em profundidade cria implicitamente uma **árvore de pesquisa**, que corresponde às arestas que levaram à exploração de nós



Classificação de Arestas por um DFS

Uma outra "visão" de DFS

- Uma visita por DFS classifica as arestas em 4 categorias
 - ▶ **Tree Edges** - Arestas da árvore de DFS
 - ▶ **Back Edges** - Aresta de um nó para um antecessor na árvore
 - ▶ **Forward Edges** - Arestas de um nó para um seu sucessor na árvore
 - ▶ **Cross Edges** - Todas as outras (de um ramo para outro ramo)



Classificação de Arestas por um DFS

Uma outra "visão" de DFS

- Um exemplo de aplicação: descobrir ciclos é descobrir... **Back Edges!**
- Perceber os tipos de arestas ajuda a resolver problemas!
- Nota: um grafo não dirigido apenas tem **Tree Edges** e **Back Edges**.

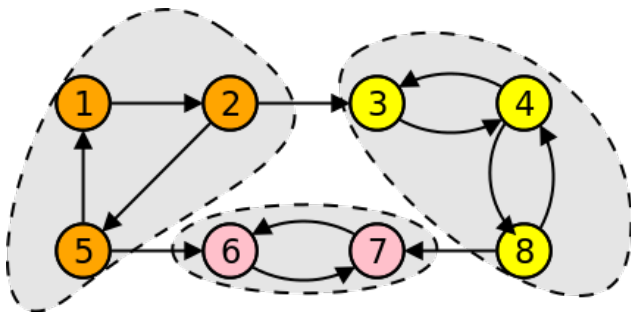
Componentes Fortemente Conexos

Uma aplicação mais elaborada de DFS

- Decompor um grafo nos seus **componentes fortemente conexos**

Um **componente fortemente conexo** (CFC) é um subgrafo maximal onde existe um caminho (dirigido) entre quaisquer pares de nós do grafo.

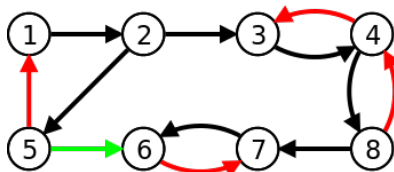
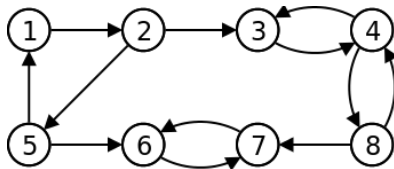
Um exemplo de um grafo e os seus três CFCs:



Componentes Fortemente Conexos

Uma aplicação mais elaborada de DFS

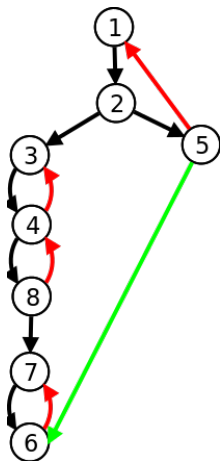
- Como calcular os componentes fortemente conexos?
- Vamos tentar usar as nossas noções de arestas para nos ajudar:



Componentes Fortemente Conexos

Uma aplicação mais elaborada de DFS

- Vamos olhar bem para a árvore gerada:

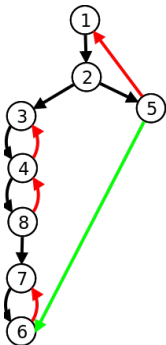


- Qual é o antecessor de um nó que é atingível por ele?
 - ▶ 1: é o próprio 1
 - ▶ 2: é o 1
 - ▶ 5: é o 1
 - ▶ 3: é o próprio 3
 - ▶ 4: é o 3
 - ▶ 8: é o 3
 - ▶ 7: é o próprio 7
 - ▶ 6: é o 7
- *Et voilà!* Aqui estão os nossos CFCs!

Componentes Fortemente Conexos

Uma aplicação mais elaborada de DFS

- Vamos acrescentar mais 2 propriedades aos nós numa visita dfs:
 - ▶ **num(i)**: ordem em que i é visitado
 - ▶ **low(i)**: menor $num(i)$ atingível pela subárvore que começa em i .
É o mínimo entre:
 - ★ $num(i)$
 - ★ menor $num(v)$ entre todos os back edges (i, v)
 - ★ menor $low(v)$ entre todos os tree edges (i, v)



i	$num(i)$	$low(i)$
1	1	1
2	2	1
3	3	3
4	4	3
5	8	1
6	7	6
7	6	6
8	5	4

Componentes Fortemente Conexos

Uma aplicação mais elaborada de DFS

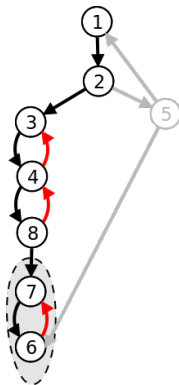
Ideia do **Algoritmo de Tarjan** para descobrir CFCs:

- Fazer um **DFS** e em cada nó i :
 - ▶ Ir colocando os nós numa **pilha S**
 - ▶ Calcular e guardar os valores de **num(i)** e **low(i)**.
 - ▶ Se à saída da visita a i tivermos um **num(i) = low(i)**, então i é a "raíz" de um CFC. Nesse caso retirar tudo o que está na pilha até i e reportar esses elementos como um CFC!

Componentes Fortemente Conexos

Uma aplicação mais elaborada de DFS

Exemplo de execução: no momento em que saímos de $dfs(7)$, descobrimos que $num(7) = low(7)$ (7 é a "raíz" de um componente fortemente conexo)



Estado da pilha **S**:

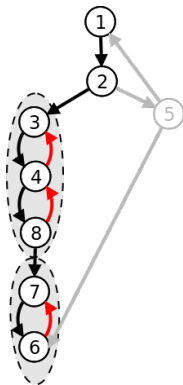
6
7
8
4
3
2
1

Retiramos tudo da pilha até ao **7**, e fazemos output do CFC: **{6, 7}**

Componentes Fortemente Conexos

Uma aplicação mais elaborada de DFS

Exemplo de execução: no momento em que saímos de $dfs(3)$, descobrimos que $num(3) = low(3)$ (3 é a "raíz" de um componente fortemente conexo)



Estado da pilha **S**:

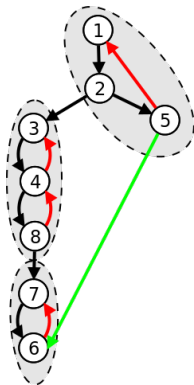
8
4
3
2
1

Retiramos tudo da pilha até ao **3**, e fazemos output do CFC: $\{8, 4, 3\}$

Componentes Fortemente Conexos

Uma aplicação mais elaborada de DFS

Exemplo de execução: no momento em que saímos de $dfs(1)$, descobrimos que $num(1) = low(1)$ (1 é a "raíz" de um componente fortemente conexo)



Estado da pilha **S**:

5
2
1

Retiramos tudo da pilha até ao **3**, e fazemos output do CFC: $\{5, 2, 1\}$

Componentes Fortemente Conexos

Algoritmo de Tarjan para CFCs

$index \leftarrow 0$; $S \leftarrow \emptyset$

Para todos os nós v do grafo **fazer**

Se $num[v]$ ainda não está definido **então**

$dfs_cfc(v)$

dfs_cfc (nó v):

$num[v] \leftarrow low[v] \leftarrow index$; $index \leftarrow index + 1$; $S.push(v)$

 /* Percorrer arestas de v */

Para todos os nós w adjacentes a v **fazer**

Se $num[w]$ ainda não está definido **então** /* Tree Edge */

$dfs_cfc(w)$; $low[v] \leftarrow \min(low[v], low[w])$

Senão se w está em S **então** /* Back Edge */

$low[v] \leftarrow \min(low[v], num[w])$

 /* Sabemos que estamos numa raiz de um SCC */

Se $num[v] = low[v]$ **então**

 Começar novo CFC C

Repetir

$w \leftarrow S.pop()$; Adicionar w a C

Até $w = v$

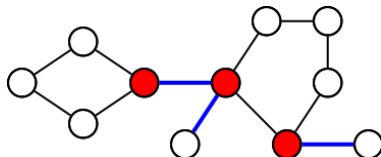
 Escrever C

Pontos de Articulação e Pontes

Um **ponto de articulação** é um **nó** cuja remoção aumenta o número de componentes conexos.

Uma **ponte** é uma **aresta** cuja remoção aumenta o número de componentes conexos.

Exemplo (a vermelho os pontos de articulação, a azul as pontes):



Um grafo sem pontos de articulação diz-se **biconexo**.

Pontos de Articulação

Uma aplicação mais elaborada de DFS

- Descobrir os **pontos de articulação** é um problema muito útil
 - ▶ Por exemplo, um grafo "robusto" a ataques não deve estar sujeito a ter pontos de articulação que se forem "atacados" o tornem desconexo.
- Como calcular? Um possível **algoritmo**:
 - 1 Fazer um DFS e contar número de componentes conexos
 - 2 Retirar do grafo original um nó e executar novo DFS, contando núm. de componentes conexos. Caso o número aumente, então o nó é um ponto de articulação.
 - 3 Repetir o passo 2 para todos os nós do grafo
- Qual seria a **complexidade** deste método? $\mathcal{O}(V(V + E))$, pois vamos ter de fazer V chamadas um DFS, e cada chamada demora $V + E$.
- **É possível fazer melhor... fazendo um único DFS!**

Pontos de Articulação

Uma aplicação mais elaborada de DFS

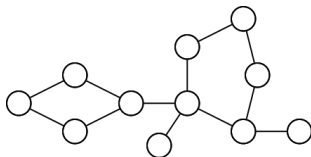
Uma ideia:

- Aplicar DFS no grafo e obter a **árvore de DFS**
- Se um **nó v tem um filho w que não tem nenhum caminho para um antecessor de v , então v é um ponto de articulação!** (pois retirá-lo desliga w do resto do grafo)
 - ▶ Isto corresponde a verificar que $low[w] \geq num[v]$
- A única exceção é a **raíz** da pesquisa. Se tiver mais que um filho... então é também ponto de articulação!

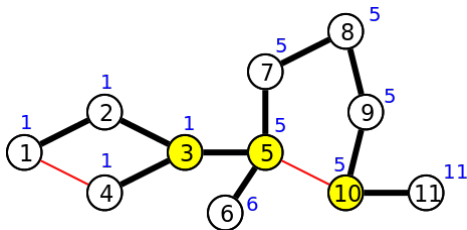
Pontos de Articulação

Uma aplicação mais elaborada de DFS

- Um grafo exemplo:

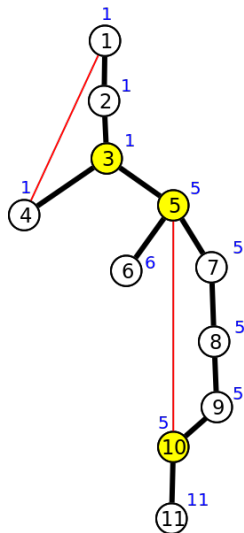


- $num[i]$ - números dentro do nó
- $low[i]$ - números a azul
- pontos de articulação: nós a amarelo



Pontos de Articulação

Uma aplicação mais elaborada de DFS



- 3 é ponto de articulação:
 $low[5] = 5 \geq num[3] = 3$
- 5 é ponto de articulação:
 $low[6] = 6 \geq num[5] = 5$
ou
 $low[7] = 5 \geq num[5] = 5$
- 10 é ponto de articulação:
 $low[11] = 11 \geq num[10] = 10$
- 1 não é ponto de articulação:
só tem um tree edge

Pontos de Articulação

Algoritmo muito parecido com CFCs, mas com DFS diferente:

Algoritmo para descobrir pontos de articulação

dfs_art(nó v):

$num[v] \leftarrow low[v] \leftarrow index$; $index \leftarrow index + 1$; $S.push(v)$

Para todos os nós w adjacentes a v **fazer**

Se $num[w]$ ainda não está definido **então** /* Tree Edge */

$dfs_art(w)$; $low[v] \leftarrow min(low[v], low[w])$

Se $low[w] \geq num[v]$ **então**

escrever($v + "$ é um ponto de articulação")

Senão se w está em S **então** /* Back Edge */

$low[v] \leftarrow min(low[v], num[w])$

$S.pop()$

Em vez da stack, podíamos usar as cores (cinzento significa que está na stack)

Pesquisa em Largura

- Uma pesquisa em largura (BFS) é muito semelhante a uma DFS. Só muda a ordem em que se visita os nós!
- Em vez de usarmos recursividade, vamos manter explicitamente uma fila de nós não visitados (q)

Esqueleto da Pesquisa em Largura - $O(V+E)$

bfs(nó v):

$q \leftarrow \emptyset$ /* Fila de nós não visitados */

$q.enqueue(v)$

marcar v como visitado

Enquanto $q \neq \emptyset$ /* Enquanto existirem nós por processar */

$u \leftarrow q.dequeue()$ /* Retirar primeiro elemento de q */

Para todos os nós w adjacentes a u **fazer**

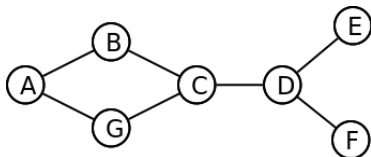
Se w ainda não foi visitado **então** /* Novo nó! */

$q.enqueue(w)$

marcar w como visitado

Pesquisa em Largura

- Um exemplo:



- 1 Inicialmente temos que $q = \{A\}$
- 2 Retiramos **A**, adicionamos vizinhos não visitados ($q = \{B, G\}$)
- 3 Retiramos **B**, adicionamos vizinhos não visitados ($q = \{G, C\}$)
- 4 Retiramos **G**, adicionamos vizinhos não visitados ($q = \{C\}$)
- 5 Retiramos **C**, adicionamos vizinhos não visitados ($q = \{D\}$)
- 6 Retiramos **D**, adicionamos vizinhos não visitados ($q = \{E, F\}$)
- 7 Retiramos **E**, adicionamos vizinhos não visitados ($q = \{F\}$)
- 8 Retiramos **F**, adicionamos vizinhos não visitados ($q = \{\}$)
- 9 q vazia, terminamos a pesquisa em largura

Pesquisa em Largura

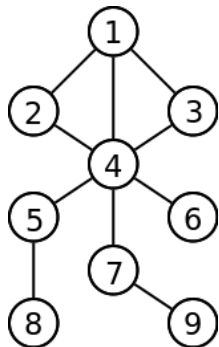
Vamos ver uma implementação a ser feita:

(programa feito na aula, código irá ser colocado aqui)

Imagine que um grafo (não dirigido) é dado como:

num_nos
num_arestas
origem₁ fim₁
origem₂ fim₂
...
origem_{|E|} fim_{|E|}

9
10
1 2
1 3
1 4
2 4
3 4
4 5
4 6
4 7
5 8
7 9



Pesquisa em Largura

Calculando distâncias

- Quase tudo o que pode ser feito com DFS também pode ser feito com BFS!
- Uma diferença importante é que na BFS visitamos os nós pela sua ordem de distância (em termos de número de arestas) ao nó inicial!
- Desse modo BFS pode ser usada para descobrir **distâncias mínimas** entre nós num **grafo não pesado** (com ou sem direção).
- Vamos ver o que realmente muda no código

Pesquisa em Largura

Calculando distâncias

- A vermelho estão as linhas que é necessário acrescentar. Em *no.distancia* fica guardada a distância ao nó *v*.

Pesquisa em Largura - Distâncias

bfs(nó *v*):

q $\leftarrow \emptyset$ /* Fila de nós não visitados */

q.enqueue(v)

v.distancia $\leftarrow 0$ /* distância de *v* a si próprio é zero */

marcar *v* como visitado

Enquanto *q* $\neq \emptyset$ /* Enquanto existirem nós por processar */

u $\leftarrow q.dequeue()$ /* Retirar primeiro elemento de *q* */

Para todos os nós *w* adjacentes a *u* **fazer**

Se *w* ainda não foi visitado **então** /* Novo nó! */

q.enqueue(w)

marcar *w* como visitado

w.distancia $\leftarrow u.distancia + 1$

Pesquisa em Largura

Mais aplicações

- BFS pode ser aplicada em qualquer tipo de grafos
- Considere por exemplo que quer saber a **distância mínima** entre um ponto de **partida** (P) e um ponto de **chegada** (C) num labirinto 2D:

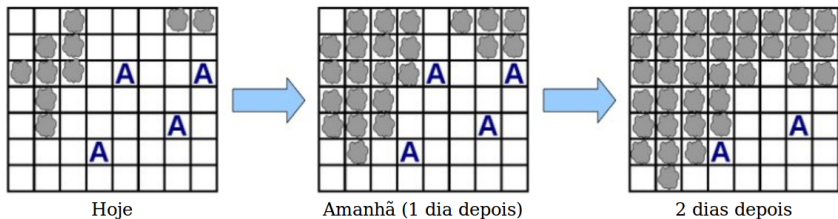
```
#####                               #####
#P.....#                           #P12345#
####.###                               #####4###
#C.....#   --->   BFS a partir de P   #876567#
#####                               #####
```

- ▶ Um nó neste grafo é a posição (x, y)
- ▶ Os nós adjacentes são $(x + 1, y)$, $(x - 1, y)$, $(x, y + 1)$ e $(x, y - 1)$
- ▶ Todo o resto da BFS fica igual! (demora $O(\text{linhas} \times \text{colunas})$)
- ▶ Para colocar na fila precisamos de saber representar uma par de coordenadas (ex: struct em C, pair ou class em C++, class em Java).

Pesquisa em Largura

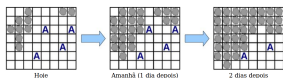
Mais aplicações

- Vamos ver um problema que saiu na qualificação das ONI'2010
- O problema foi inspirado na erupção do **vulcão Eyjafjallajökull**, cuja nuvem de cinzas tantos problemas causou no tráfego aéreo na europa.
- Imagine que a posição da **nuvem de cinzas** lhe é dada numa matriz e que em cada unidade de tempo a nuvem se expande uma quadrícula na horizontal e na vertical. Os A's são aeroportos.



Pesquisa em Largura

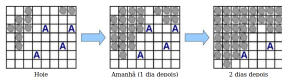
Mais aplicações



- O problema pede:
 - ▶ Qual o **primeiro aeroporto** a ser coberto pelas cinzas
 - ▶ Quanto tempo até **todos** os aeroportos estarem cobertos pelas cinzas
- Seja $dist(A_i)$ a distância do aeroporto i até uma qualquer nuvem
- O problema pede a menor $dist(A_i)$ e a maior $dist(A_i)$!
- Uma maneira seria fazer uma BFS a partir de todos os aeroportos
 $\mathcal{O}(\text{num_aeroportos} \times \text{linhas} \times \text{colunas})$
- Outra maneira seria fazer uma BFS a partir de todas as cinzas
 $\mathcal{O}(\text{num_cinzas} \times \text{linhas} \times \text{colunas})$
- Como fazer melhor, e executar apenas uma BFS?

Pesquisa em Largura

Mais aplicações



- Ideia: inicializar a lista da BFS com todas as cinzas!
- Tudo o resto fica igual.

...#...	.. 1#1 ..	. 21#12 .	321#123	321#123
..##...	. 1##1 ..	21##12 .	21##123	21##123
.####...	-> 1####1 .	-> 1####12	-> 1####12	-> 1####12
.....	11111 ..	1111 12 .	1111 123	1111123
##.....	##1	##122 ..	##1223 .	##12234

- As distâncias vão ser as que queremos.
- Cada célula só vai ser percorrida uma vez! $\mathcal{O}(\text{linhas} \times \text{colunas})$

Pesquisa em Largura

Mais aplicações

- Vamos a um último problema onde o grafo não existe "explicitamente" [*problema original das IOI'1996*]
- Considere o seguinte puzzle (uma espécie de cubo de Rubik "plano")

▶ A posição inicial do puzzle é:

1	2	3	4
8	7	6	5

▶ Em cada jogada pode fazer um de três movimentos:

★ **Movimento A:** trocar as fila superior com a inferior

8	7	6	5
1	2	3	4

★ **Movimento B:** shift do rectângulo para a direita

4	1	2	3
5	8	7	6

★ **Movimento C:** rotação (sentido do ponteiros do relógio) das 4 células do meio

1	7	2	4
8	6	3	5

▶ Quantas jogadas precisa para chegar a uma dada posição?

Pesquisa em Largura

Mais aplicações

- Pode ser resolvido... com **pesquisa em largura!**
- O nó inicial é... a posição inicial.
- Os nós adjacentes são... as posições que se podem alcançar usando movimentos A, B ou C.
- Quando atingimos a posição desejada... sabemos necessariamente a distância mínima (n° jogadas) para lá chegar!
- O (mais) difícil é... saber como representar os estados! :)