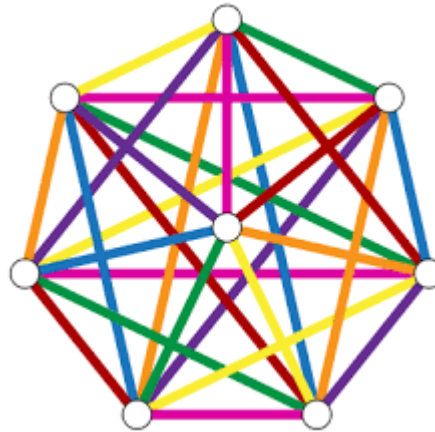
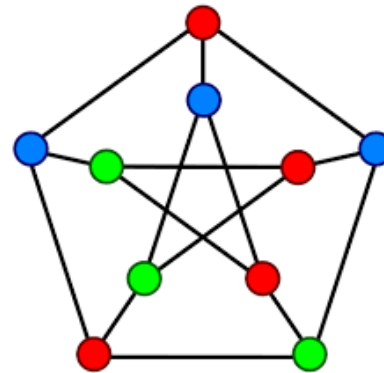
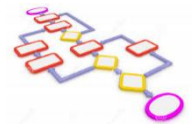


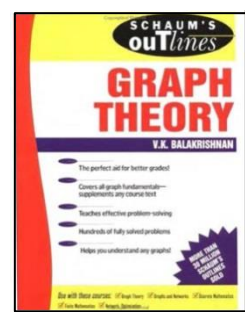
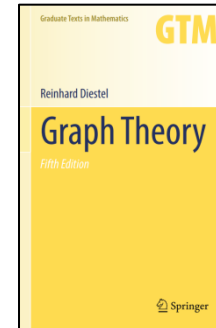
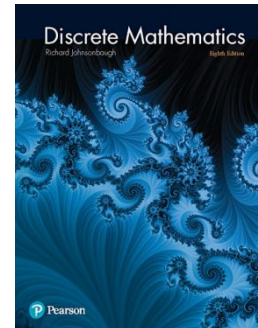
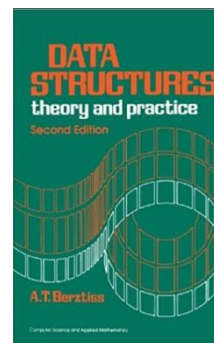
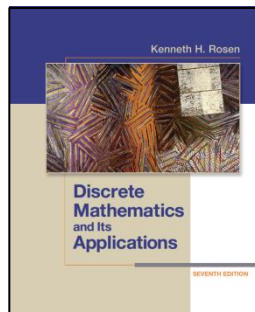
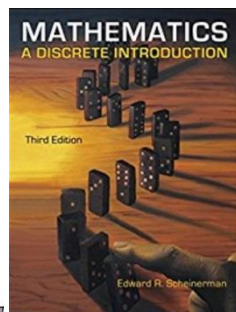
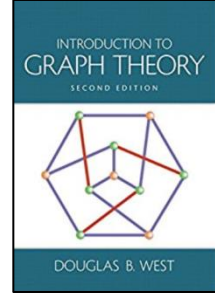
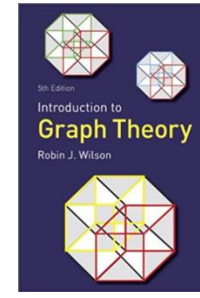
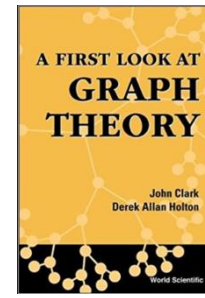
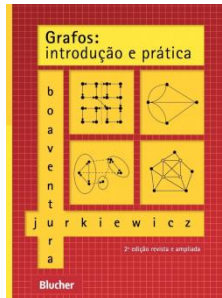
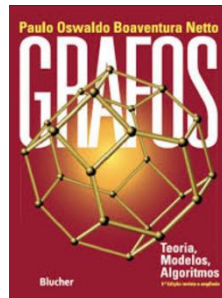
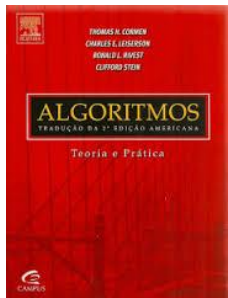
# Unidade 21– Algoritmos em Grafos





# Bibliografia

- Algoritmos Teoria e Prática – Cormen – 2ª edição – Editora Campus
- Fundamentos da Teoria dos Grafos para Computação – M.C. **Nicoletti**, E.R. **Hruschka Jr.** 3ª Edição - LTC
- Grafos – Teoria, Modelos, Algoritmos – Paulo Oswaldo **Boaventura Netto**, 5ª edição
- Grafos – Conceitos, Algoritmos e Aplicações – Marco **Goldberg**, Elizabetj Goldberg, Editora Campus
- A first look at Graph Theory – John **Clark**, Derek Allan **Holton** – 1998, World Cientific
- Introduction to Graph Teory – Robin J. **Wilson** – 4<sup>th</sup> Edition – Prentice Hall – 1996
- Introduction to Graph Theory – Douglas **West** – Second Edition 2001 – Pearson Edition
- Mathematics – A discrete Introduction – Third Edition – Edward R. **Scheinerman** – 2012
- Discrete Mathematics and its Applications – Kenneth H. **Rosen** – 7<sup>th</sup> edition – McGraw Hill – 2012
- Data Structures – Theory and Practice – A. T. **Bertziss** - New York – Academic Press – 1975 – Second Edition
- Discrete Mathematics – R. **Johnsonbaugh** – Pearson – 2018 – Eighth Edition
- Graph Theory – R. **Diestel** – Springer – 5<sup>th</sup> Edition – 2017
- Graph Theory – Theory and Problems of Graph Theory – V. **Balakrishnan** –Schaum's Outline – McGraw Hill - 1997



# Tipo abstrato de Dado Grafo



- ✓ Um grafo é uma coleção de **vértices** e **arestas**;
- ✓ Pode-se modelar a abstração por meio de uma combinação de três tipos de dados: **Vértice**, **Aresta** e **Grafo**;
- ✓ Um vértice (**VERTEX**) pode ser representado por um objeto que armazena a informação fornecida pelo usuário, por exemplo, informações de um aeroporto;
- ✓ Uma aresta (**EDGE**) armazena relacionamentos entre vértices, por exemplo: número do voo, distâncias, custos, etc.
- ✓ A ADT **Grafo** deve incluir diversos métodos para se operar com grafos;



# Tipo abstrato de Dado Grafo



- ✓ A ADT **Grafo** pode lidar com grafos **direcionados** ou **não-direcionados**.
- ✓ Uma aresta  $(u,v)$  é dita **direcionada** de  $u$  para  $v$  se o par  $(u,v)$  é ordenado, com  $u$  precedendo  $v$ ;
- ✓ Uma aresta  $(u,v)$  é dita **não direcionada** se o par  $(u,v)$  **não** for ordenado.

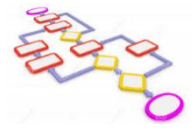


# Grafos direcionados (Digrafos)

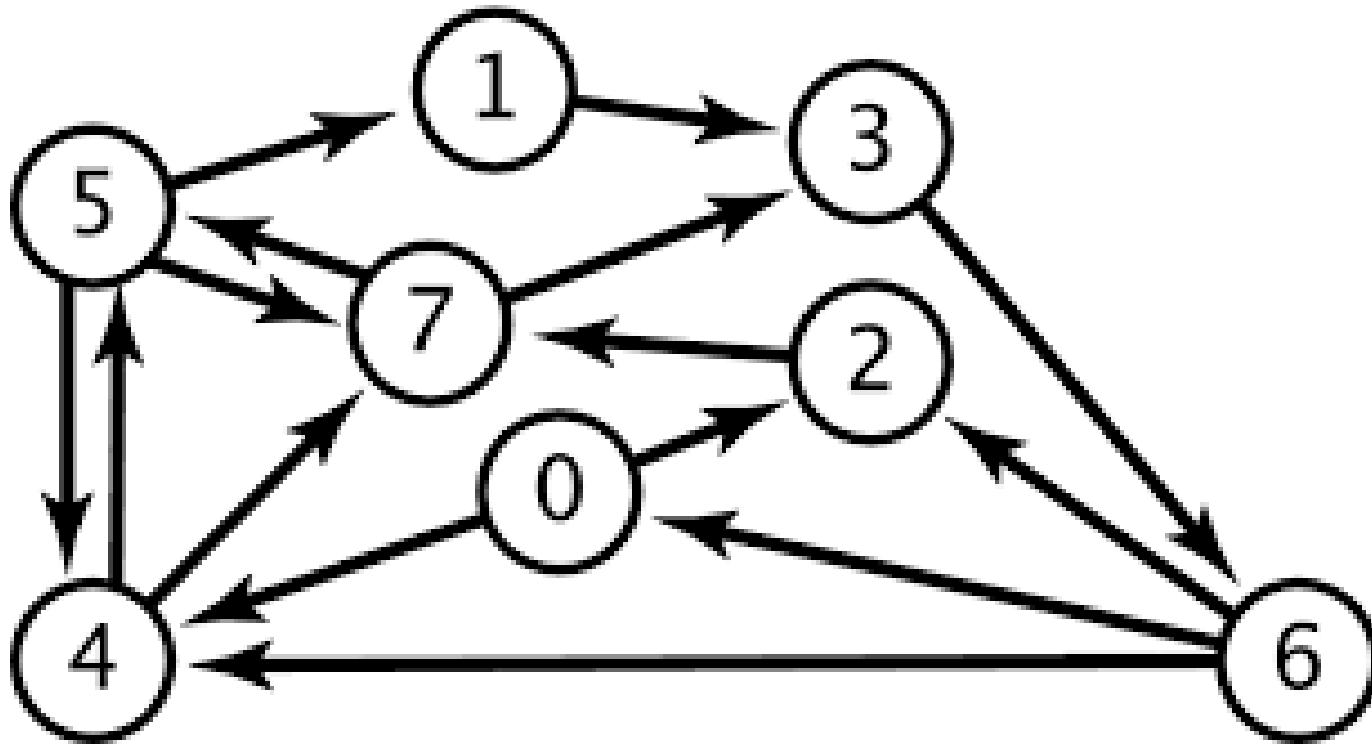


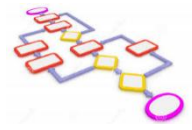
- ✓ Um **digrafo**, ou grafo **dirigido**, ou grafo **direcionado**, é um grafo com **orientação (flexa)** nas arestas;
- ✓ **Digrafos** são mais gerais que grafos;
- ✓ Num certo sentido, **digrafos** são objetos mais naturais que grafos;
- ✓ A maioria dos cursos de Teoria dos Grafos trata apenas de grafos **não direcionados (não dirigidos)**;
- ✓ Isso geralmente ocorre, pois as propriedades dos grafos não direcionados são mais facilmente aprendidas.





# Grafos direcionados (Digrafos)

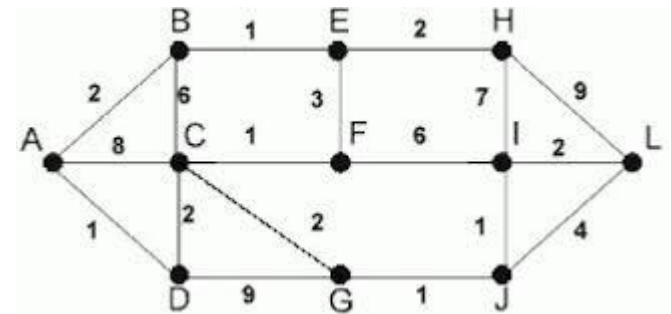




# Métodos – TAD Grafo – Exemplos



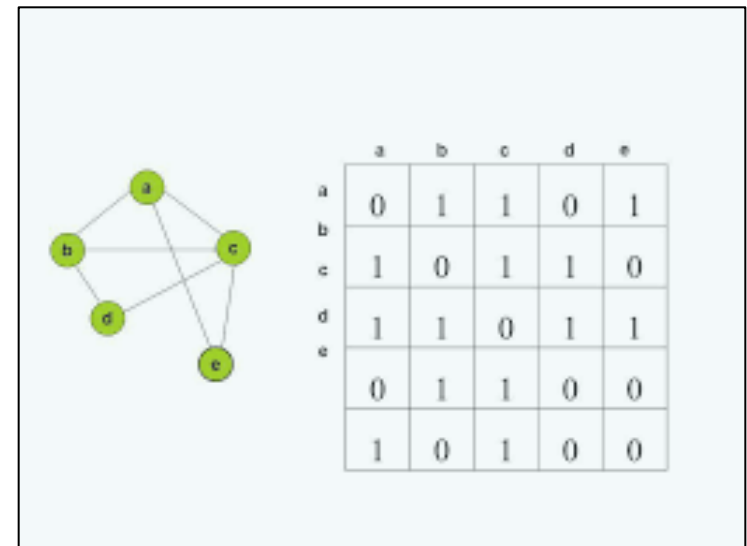
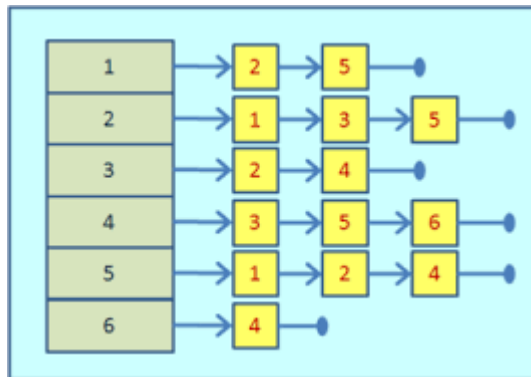
- ✓ **Retorna** uma lista dos vértices **incidentes** de um vértice qualquer;
- ✓ **Remove** uma **aresta** do grafo;
- ✓ **Remove** um **vértice** do grafo;
- ✓ **Adiciona** uma **aresta** do grafo;
- ✓ **Imprime** as **arestas** do grafo.

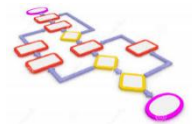


# Estruturas de Dados para Grafos



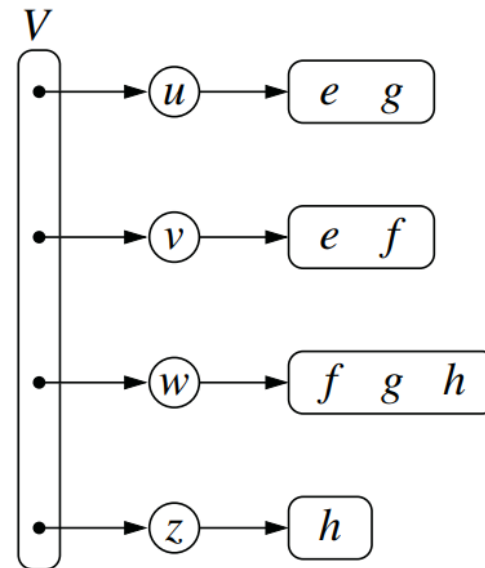
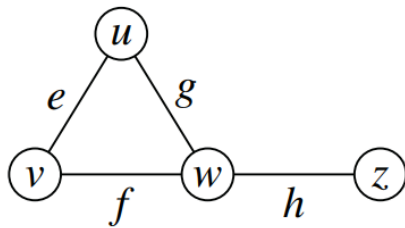
- ✓ Duas abordagens são geralmente aplicadas:
  - ✓ Lista de Adjacências
  - ✓ Matriz de Adjacências

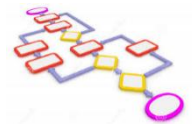




# Lista de Adjacências

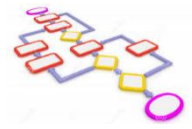
- ✓ Emprega-se uma **lista** de **vértices**, no qual cada **vértice** aponta para uma outra lista com as **arestas** incidentes ao vértice;





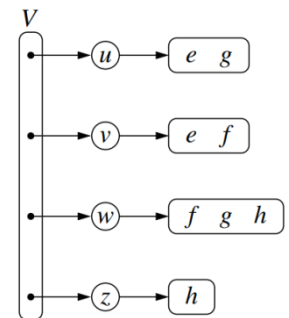
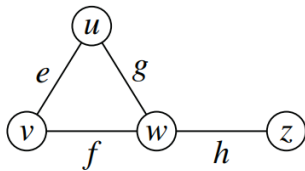
# Qual o desempenho dos algoritmos que usam Listas de Adjacência?

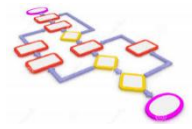




# Qual o desempenho dos algoritmos com a Lista de Adjacência?

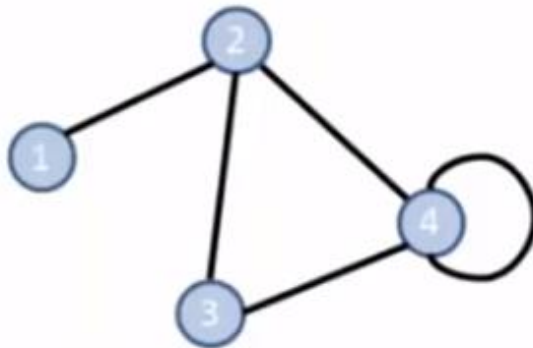
- ✓ A **pesquisa** de um determinado vértice na lista de vértices, tem complexidade  **$O(n)$** ;
- ✓ Igualmente, a pesquisa de uma determinada **aresta** na lista de arestas, tem complexidade  **$O(n)$** ;
- ✓ O método que **insere** um novo **vértice** na lista de vértices, tem complexidade  **$O(1)$** ;
- ✓ O método que **insere** uma nova **aresta** na lista de arestas também tem complexidade  **$O(1)$** ;
- ✓ O método que retorna os vértices na lista de vértices tem complexidade  **$O(n)$** ;
- ✓ O método que retorna as arestas na lista de arestas tem complexidade  **$O(n)$** ;
- ✓ O método que remove um vértice tem complexidade  **$O(\deg(v))$** .





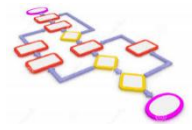
# Matriz de Adjacências

- ✓ Utiliza-se uma matriz  $n \times n$  para o armazenamento do grafo;
- ✓ Sendo  $n$  o número de vértices do grafo;
- ✓ Uma aresta é representada por uma “**marca**” (um determinado valor) na posição  $(i,j)$  da matriz;
- ✓ Aresta liga o vértice  $i$  ao vértice  $j$ ;
- ✓ Para muitos vértices e poucas arestas, desperdiça-se espaço.



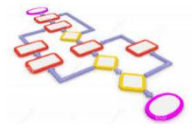
	1	2	3	4
1	0	1	0	0
2	1	0	1	1
3	0	1	0	1
4	0	1	1	1





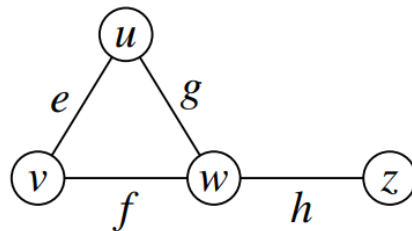
# Qual o desempenho dos algoritmos com a estrutura Matriz de Adjacências?





# Qual o desempenho dos algoritmos com Matriz de Adjacências ?

- ✓ A maior vantagem de uma matriz de adjacências é que qualquer aresta pode ser acessada no pior caso em tempo  **$O(1)$** ;
- ✓ Entretanto, diversas outras operações são **menos** eficientes com o emprego de matriz de adjacências;
- ✓ Por exemplo, para se encontrar as arestas incidentes a um vértice  $V$ , deve-se examinar todas as  $n$  entradas associadas com  $V$ ; Lembre-se que em listas de adjacência pode-se localizar arestas em  **$O(d(V))$** .



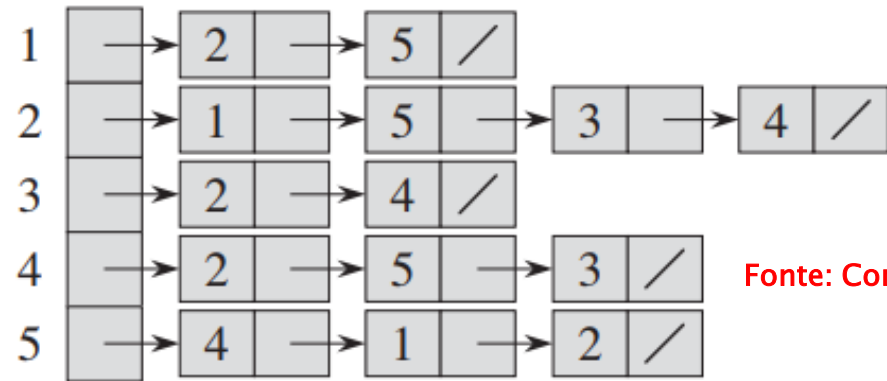
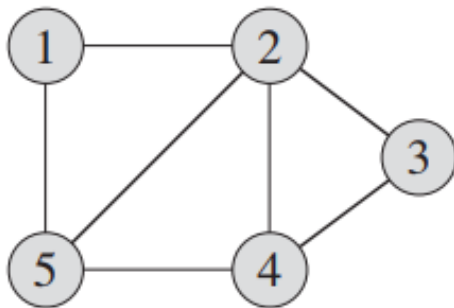
	0	1	2	3
$u \longrightarrow$	0	$e$	$g$	
$v \longrightarrow$	1	$e$	$f$	
$w \longrightarrow$	2	$g$	$f$	$h$
$z \longrightarrow$	3		$h$	



# Implementação com Listas de Adjacências



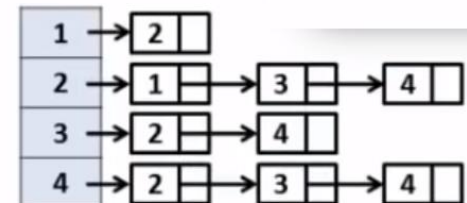
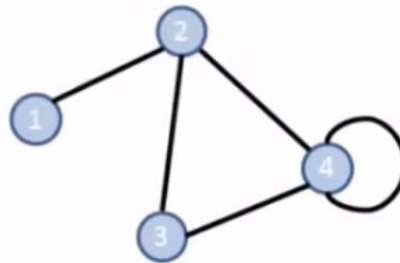
- ✓ Fornece uma forma compacta de representar grafos **esparços** – aqueles para os quais  $|E|$  é muito menor que  $|V|^2$ . Assim, essa implementação é usualmente o método mais escolhido;



Fonte: Cormen

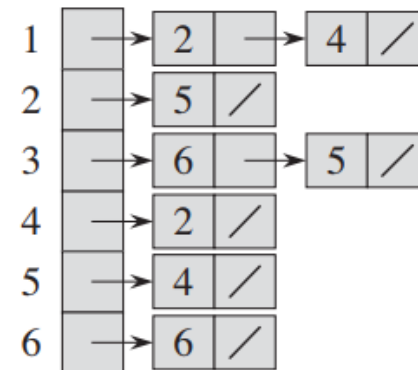
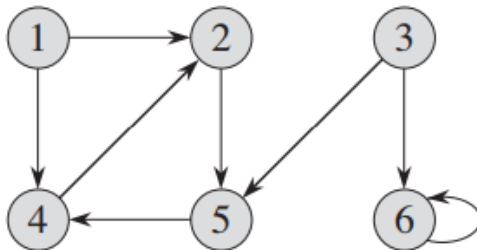


GRAFO



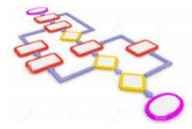
# Implementação com Listas de Adjacências

- ✓ A representação de um grafo  $G = (V, E)$  na forma de Listas de Adjacências consiste de um **array** Adj de  $|V|$  **listas**, uma para cada vértice em  $V$ ;
- ✓ Para cada  $u \in V$ , a lista de adjacência Adj[u] consiste de todos os vértices  $v$  tais que haja uma aresta  $(u, v)$   $u \in E$ ;
- ✓ Ou seja, Adj[u] consiste de todos os vértices adjacentes a  $u$  em  $G$ ;
- ✓ Considerando que a lista de adjacências representa as arestas de um grafo, pode-se representar o grafo  $G$  com os atributos: **V**: conjunto de vértices de  $G$  e **Adj[u]**: conjunto de arestas de  $G$ , para todo  $u \in V$ ;



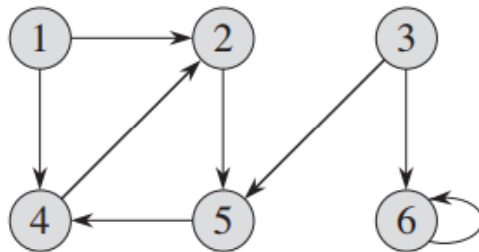
Fonte: Cormen



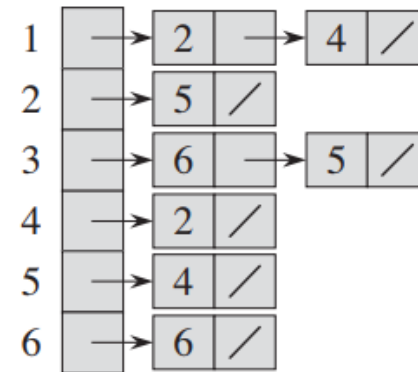


# Implementação com Listas de Adjacências

- ✓ Em um grafo **G direcionado (Digrafo)**, a soma dos nós de todas as listas de adjacências é  $|E|$ ;
- ✓ Isso ocorre, uma vez que 1 aresta na forma  $(u,v)$  é representada uma única vez em  $\text{Adj}[u]$ .

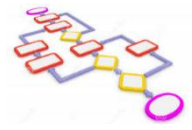


\* 8 arestas



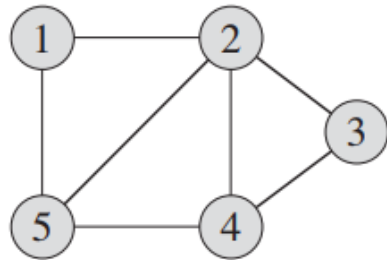
\* 8 nós



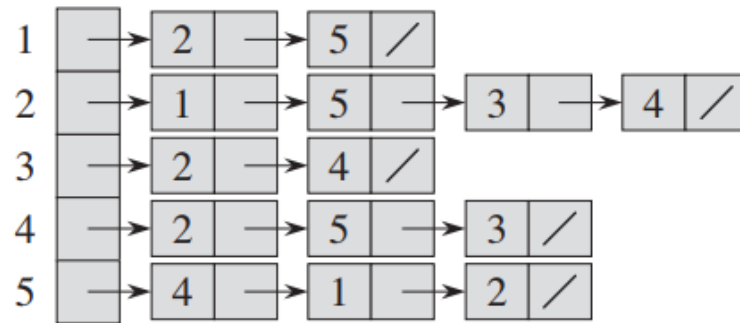


# Implementação com Listas de Adjacências

- ✓ Em um grafo **G simples não-direcionado**, a soma dos nós de todas as listas de adjacências é  $2 * |E|$ ;
- ✓ Isso ocorre, uma vez que se **(u,v)** é uma aresta não-direcionada, então u aparece em Adj[u] e vice-versa.

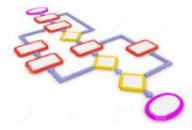


\* 7 arestas



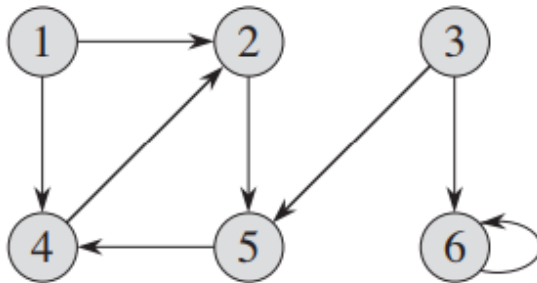
\* 14 nós



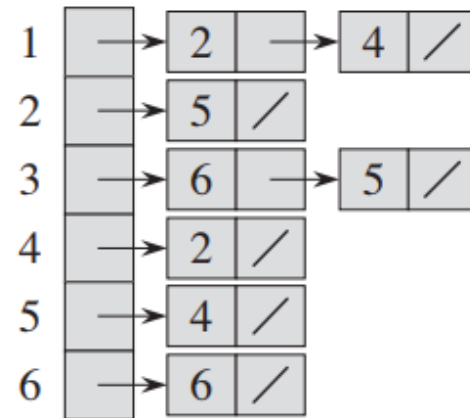


# Implementação com Listas de Adjacências

- ✓ Para grafos **direcionados**, a representação em listas de adjacências tem a desejável propriedade que a quantidade de memória necessária é  $\Theta(V + E)$ ;



- ✓ 6 vértices
- ✓ 8 arestas

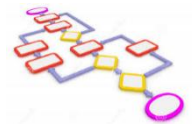


- ✓ 6 nós para vértices
- ✓ 8 nós para arestas



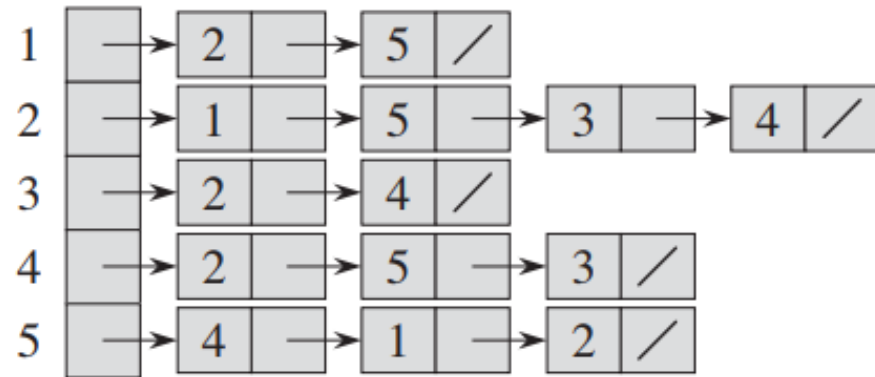
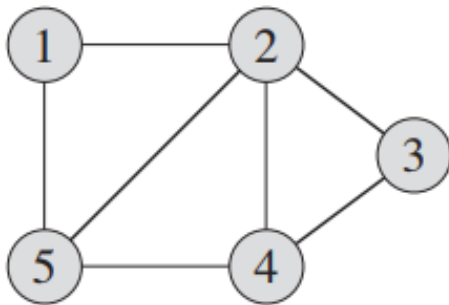
\* Total de 14 nós





# Implementação com Listas de Adjacências

- ✓ Para grafos simples **não-direcionados**, a representação em listas de adjacências tem a desejável propriedade que a quantidade de memória necessária é  $\Theta(V + 2 \cdot E)$ ;



- ✓ 5 vértices
- ✓ 7 arestas

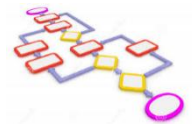


- ✓ 5 nós para vértices
- ✓ 14 nós para arestas



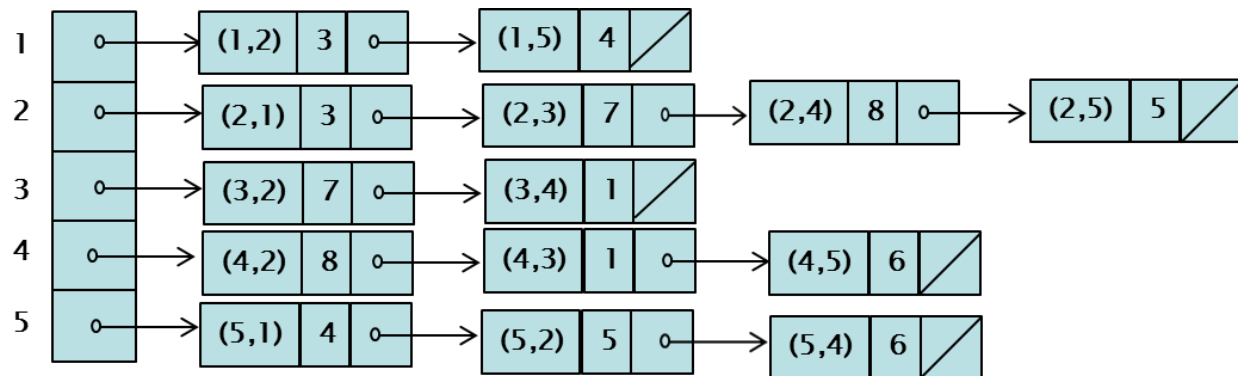
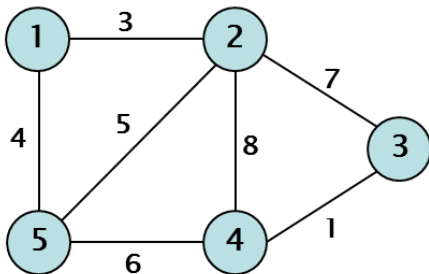
\* Total de 19 nós

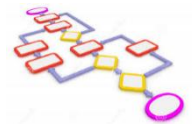




# Implementação com Listas de Adjacências

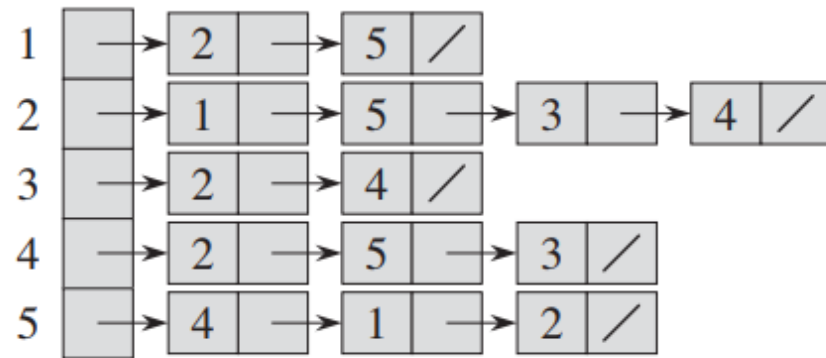
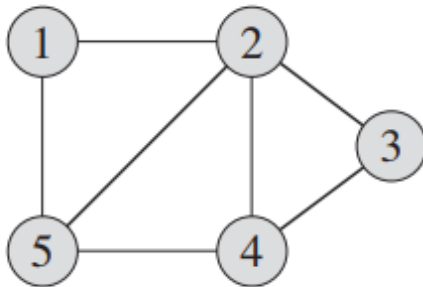
- ✓ Pode-se facilmente adaptar-se uma lista de adjacências para representar grafos com **pesos**;
- ✓ Ou seja, grafos para o qual cada aresta tem um peso associado, tipicamente dado por uma função de pesos:  $w : E \rightarrow R$ .
- ✓ Por exemplo: seja  $G = (V, E)$  um grafo com pesos com uma função de pesos  $w$ . Pode-se armazenar o valor da função  $w$  para uma aresta  $e \in E$  no nó da lista  $ajd[u]$ .

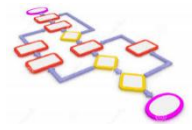




# Implementação com Listas de Adjacências

- ✓ Uma potencial desvantagem da representação por **Lista de Adjacências** é que ela **não** provê uma forma rápida de se determinar se uma determinada aresta  $(u,v)$  está presente no grafo;
- ✓ Assim, será necessário pesquisá-la na Lista de Adjacências;

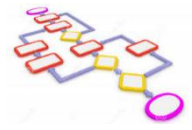




# Busca em Grafos

- ✓ Consiste em “**explorar**” um grafo;
- ✓ Processo sistemático de como **caminhar** pelos vértices e arestas do grafo;
- ✓ Diversos problemas em grafos necessitam de operações de busca em grafos;
- ✓ A operação de busca pode exigir que se visite todos os vértices do grafo para determinados problemas;
- ✓ Os principais tipos de busca em grafos são: **Busca em Profundidade, Busca em Largura e Busca pelo Menor Caminho**;

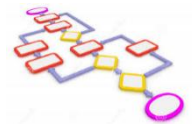




# Busca em Profundidade

- ✓ Parte-se de um vértice inicial e se explora o máximo possível cada um de seus ramos, antes de se retroceder (“**Backtracking**”);
- ✓ Explora-se todas as arestas de um determinado vértice antes de se retornar a seu antecessor;
- ✓ A estratégia seguida pela busca em profundidade (**Depth\_first search** ou **DFS**) é buscar mais fundo no grafo sempre que possível;
- ✓ A busca é encerrada quando se encontra o que se quer ou visita-se todos os vértices;





# Busca em Profundidade

## DFS – Funcionamento

**Defina um nó inicial**

**Escolha um de seus adjacentes ainda não visitados**

**Visite-o**

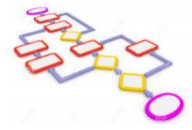
**Repita o processo até atingir o nó objetivo, ou um nó cuja adjacência já tenha sido toda visitada (nó final)**

**Se atingir um nó final que não seja objetivo:**

**Volte ao pai deste**

**Continue de um nó irmão ainda não visitado**





# Busca em Profundidade

## DFS – Reescrevendo...

**Defina um nó inicial**

**Enquanto este não for um nó objetivo ou final (nó cuja adjacência já tenha sido toda visitada)**

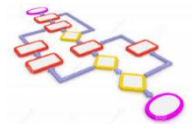
**Escolha um de seus adjacentes ainda não visitados  
Visite-o**

**Se nó final não objetivo:**

**Volte ao pai deste**

**Se houver pai, repita. Senão escolha outro nó inicial**





# Busca em Profundidade

## DFS – Funcionamento

**Defina um nó inicial**

**Enquanto este não for um nó objetivo ou final (nó cuja adjacência já tenha sido toda visitada)**

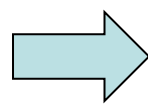
**Escolha um de seus adjacentes ainda não visitados  
Visite-o**

**Se nó final não objetivo:**

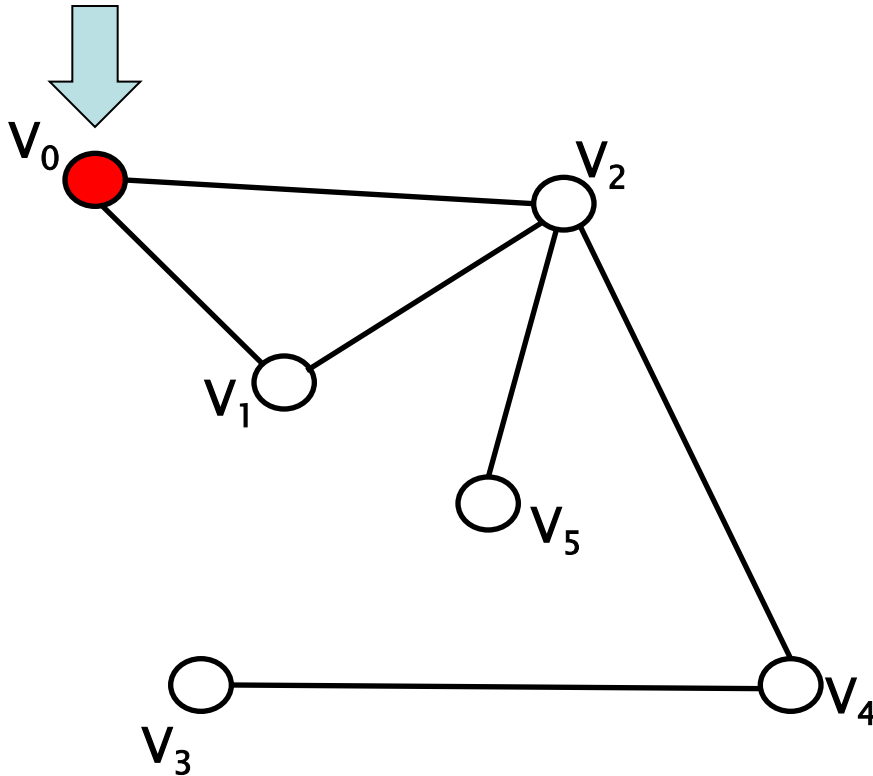
**Volte ao pai deste  
Se houver pai, repita. Senão  
escolha outro nó inicial**



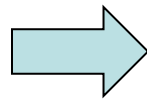
# Busca em Profundidade



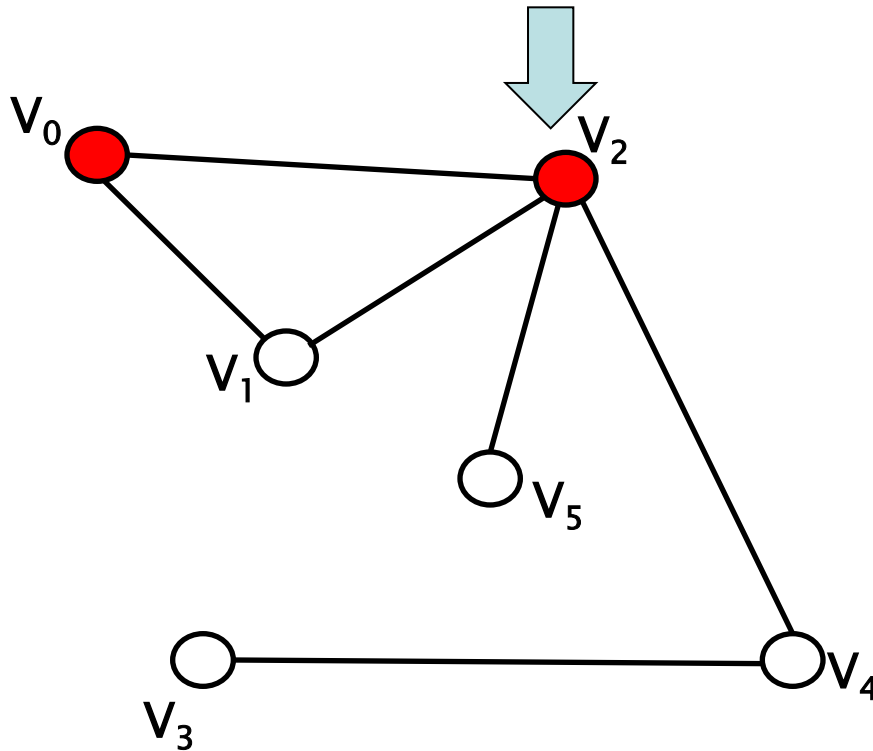
Inicia-se por exemplo, a busca em  $V_0$ .



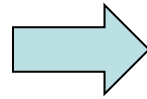
# Busca em Profundidade



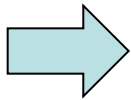
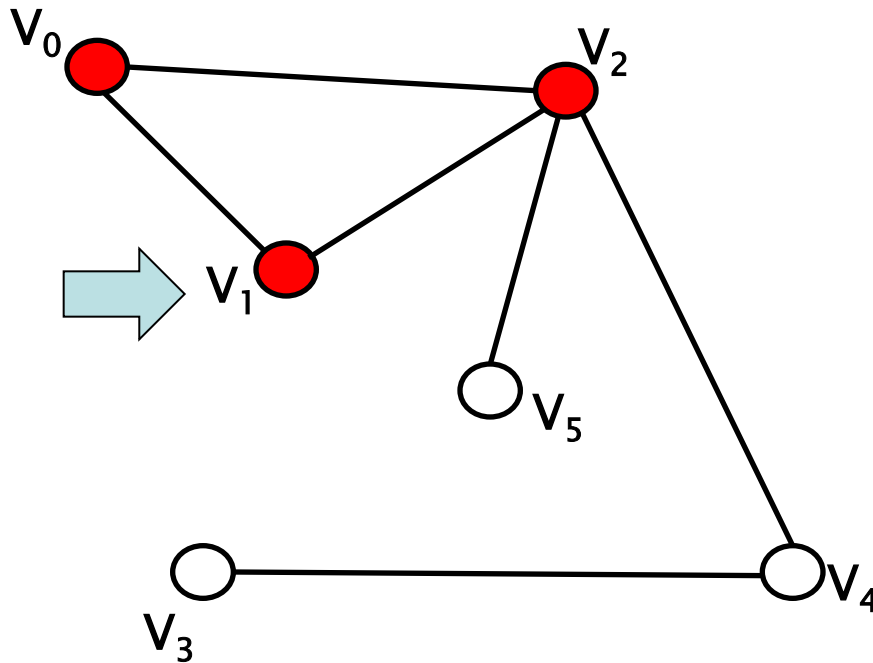
Vértices adjacentes ainda não visitados a partir de  $V_0$ :  $V_1$  e  $V_2$   
Escolho  $V_2$



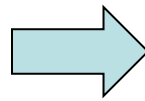
# Busca em Profundidade



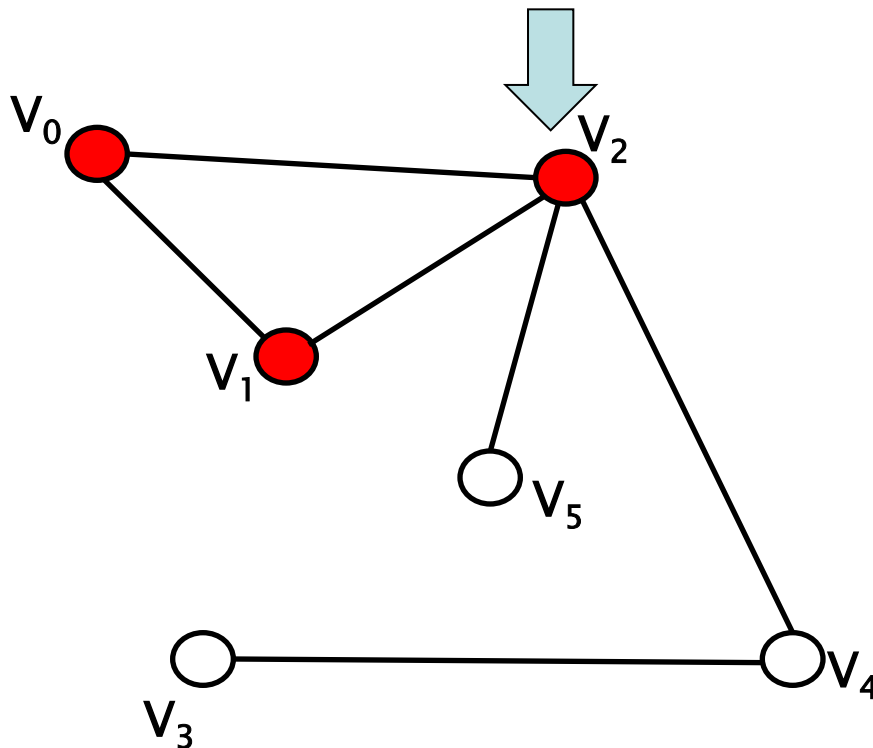
Vértices adjacentes ainda não visitados a partir de  $V_2$ :  $V_1$ ,  $V_4$ , ou  $V_5$   
Escolho  $V_1$



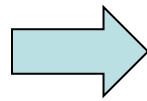
# Busca em Profundidade



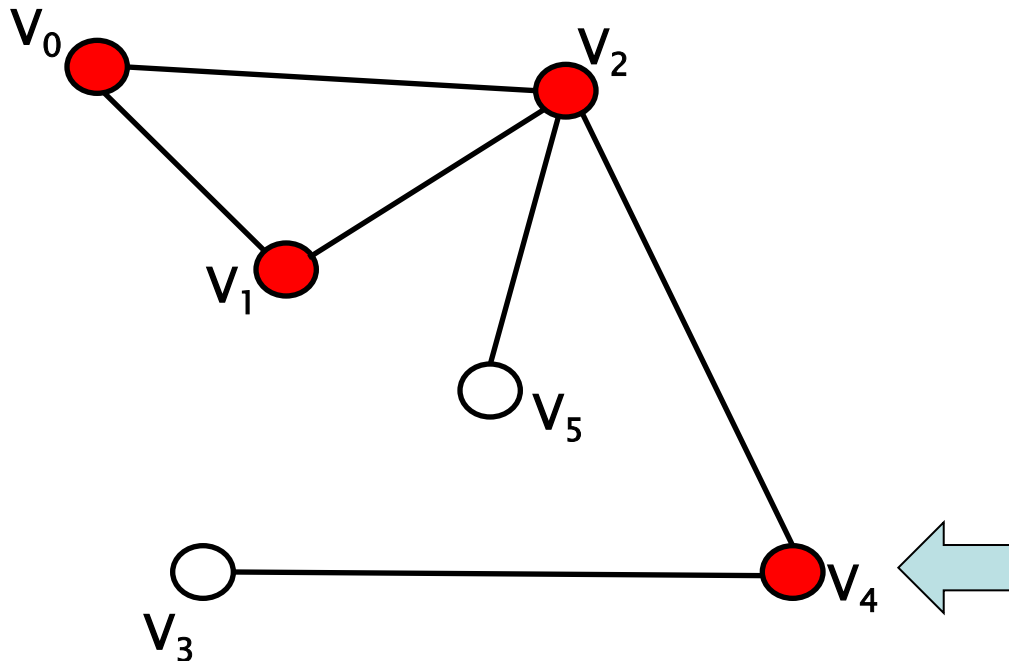
Vértices adjacentes ainda não visitados a partir de  $V_1$ : Nenhum  
Ainda há vértices a serem percorridos



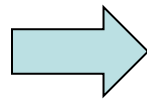
# Busca em Profundidade



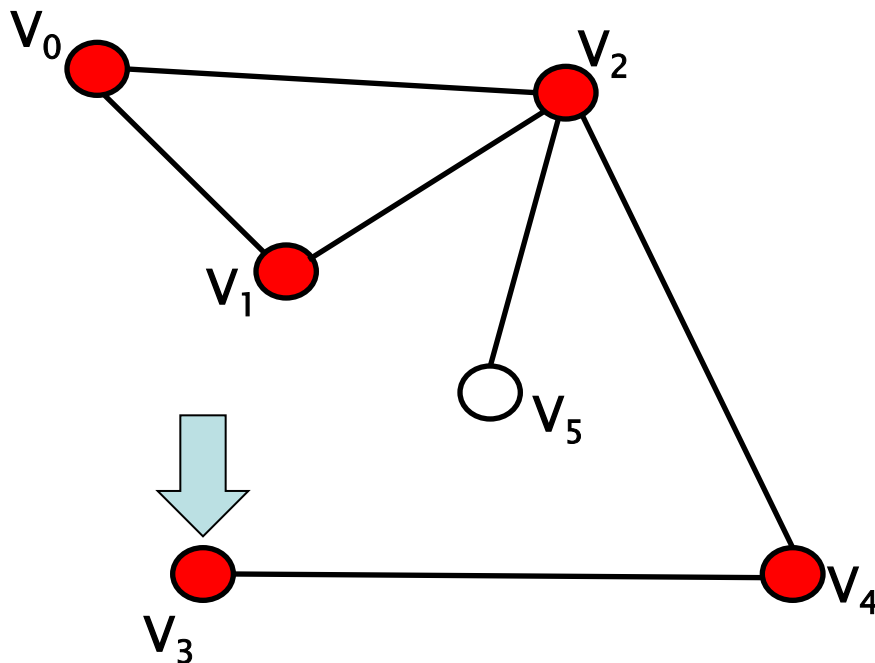
Vértices adjacentes ainda não visitados a partir de  $V_2$ :  $V_4$  ou  $V_5$   
Escolho  $V_4$



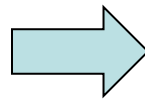
# Busca em Profundidade



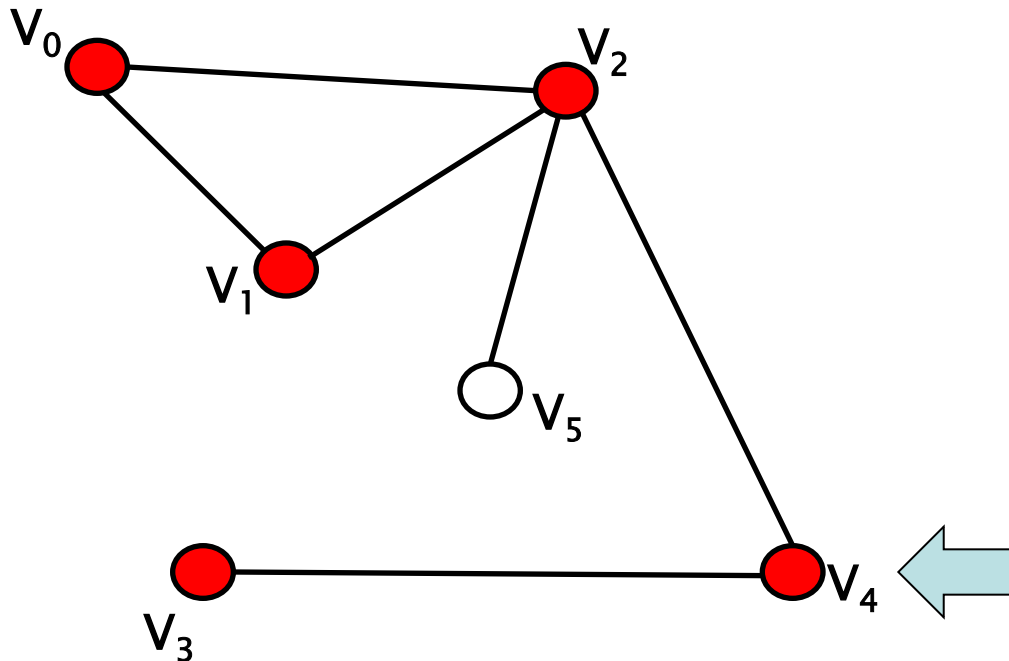
Vértices adjacentes ainda não visitados a partir de  $V_4$ :  $V_3$   
Escolho  $V_3$



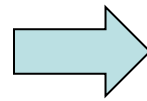
# Busca em Profundidade



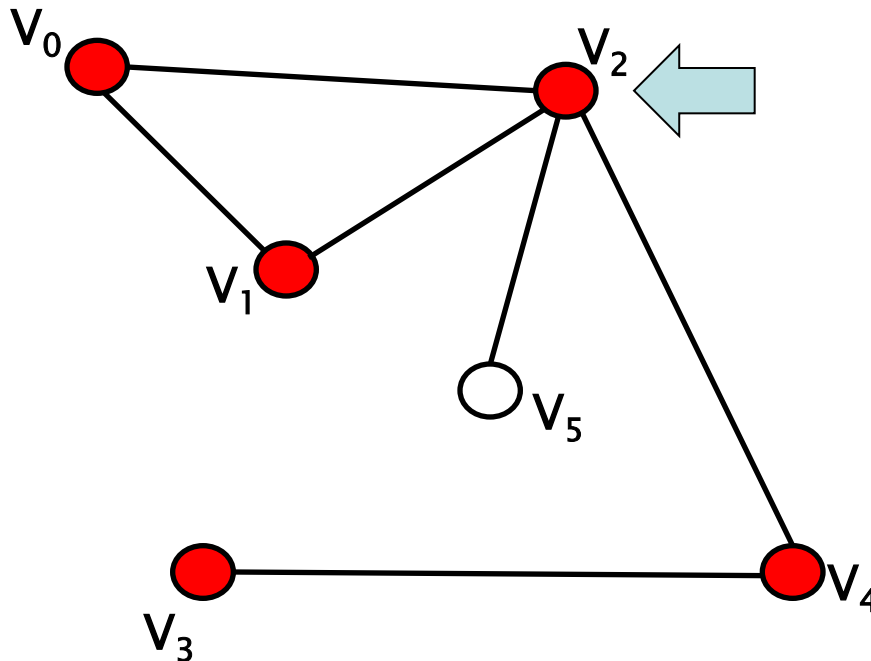
Vértices adjacentes ainda não visitados a partir de  $V_3$ : não há  
Ainda há vértices não visitados  
Volto para o Pai:  $V_4$   
Escolho  $V_4$



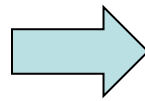
# Busca em Profundidade



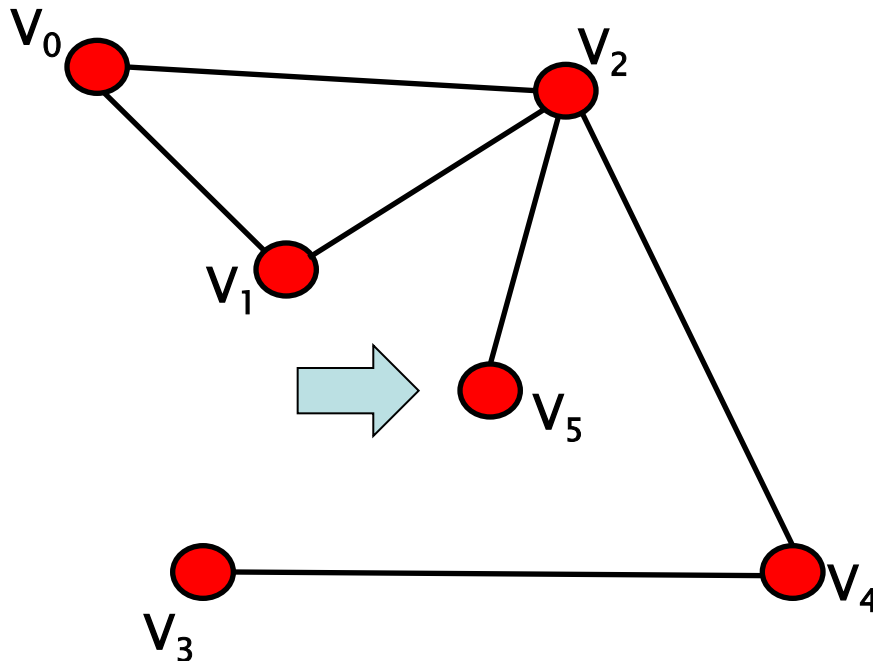
Vértices adjacentes ainda não visitados a partir de  $V_4$ : não há  
Ainda há vértices não visitados  
Volto para o Pai:  $V_2$   
Escolho  $V_2$



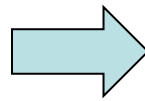
# Busca em Profundidade



Vértices adjacentes ainda não visitados a partir de  $V_2$ :  $V_5$   
Escolho  $V_5$

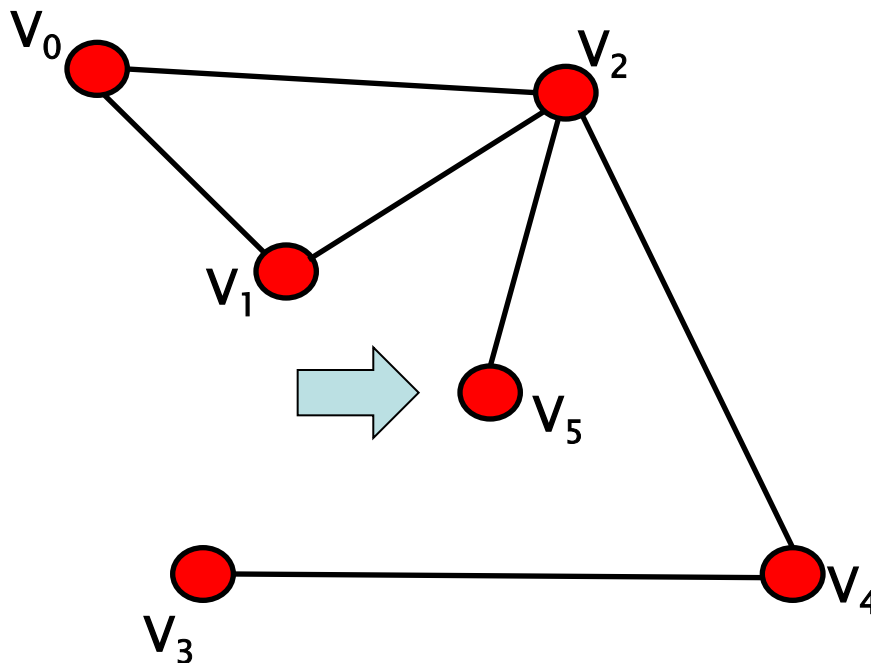


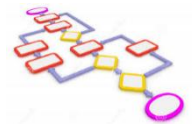
# Busca em Profundidade



Vértices adjacentes ainda não visitados a partir de  $V_5$  : não Há  
Não há mais vértices a serem visitados

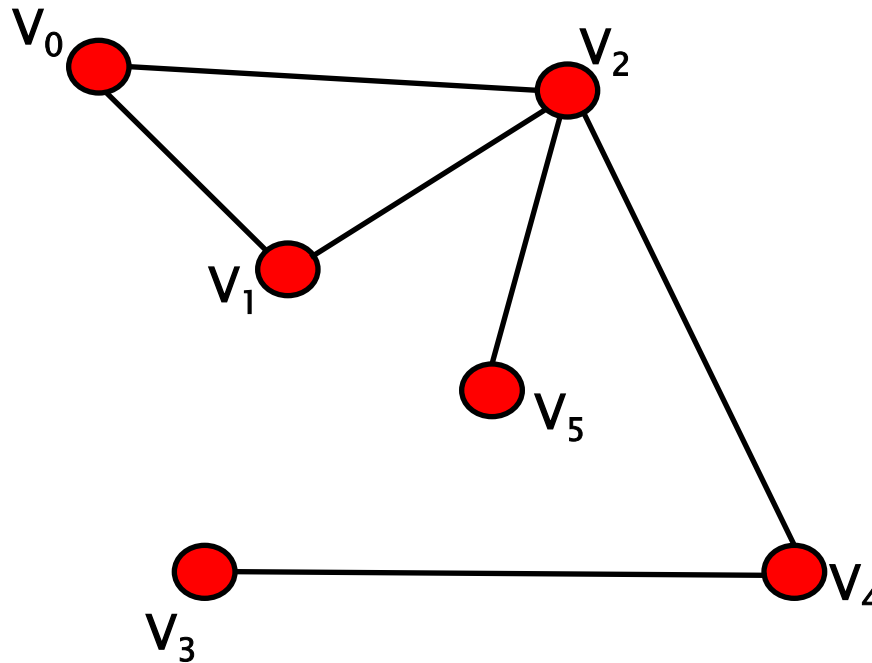
Fim da Busca em Profundidade

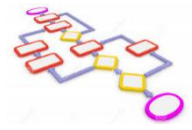




# Busca em Profundidade

Sequência gerada:  $V_0, V_2, V_1, V_4, V_3, V_5$

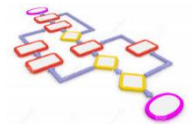




# Busca em Largura

- ✓ Parte-se de um vértice inicial e se explora todos os vértices vizinhos;
- ✓ Em seguida, para cada vértice vizinho, repete-se esse processo, visitando-se os vértices ainda não explorados;





# Busca pelo menor caminho

- ✓ Parte-se de um vértice inicial, calcula-se a menor distância deste vértice à todos os demais, desde que exista uma aresta ligando-os;
- ✓ Esse problema pode ser resolvido com o **Algoritmo de Dijkstra** para grafos direcionados ou não direcionados com arestas de peso não negativo;





# FIM

