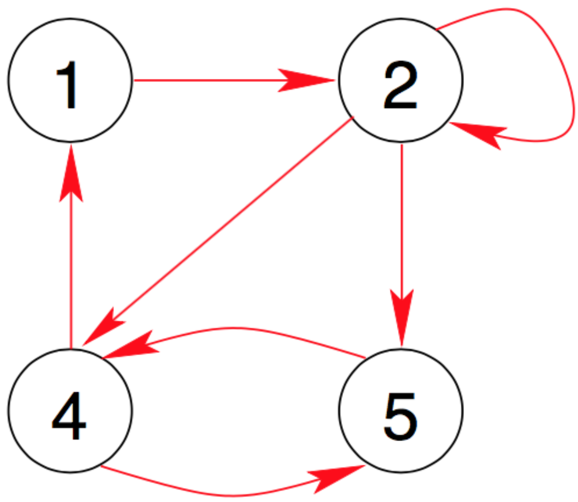
Grafos Orientados

Copiar link

Os grafos são objectos matemáticos que são também utilizados como estruturas de dados em programação.

Um *grafo orientado* é um par (V, E) com V um conjunto finito de *vértices* ou *nós* e E uma relação binária sobre V – o conjunto de *arestas* ou *arcos* do grafo.

Exemplo:
$V = \{1, 2, 3, 4, 5, 6\}$
$E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$



Grafo orientado G1

- Se $(i, j) \in E$, j diz-se *adjacente* a i
  i e j são respectivamente os vértices de *origem* e *destino* da aresta (i, j)

- Uma aresta (i, i) designa-se por **anel**

As aplicações dos grafos orientados são inúmeras; além da evidente modelação de redes (por exemplo de estradas), que discutiremos mais à frente, estes grafos têm aplicação por exemplo em *gestão de projectos*, fazendo corresponder os vértices a **tarefas.** As arestas podem exprimir uma relação de precedência: se $(t_1, t_2) \in E$, então a tarefa t_1 deve sempre ser realizada antes da tarefa t_2.

Existe uma relação óbvia entre grafos orientados e **autómatos de estados finitos**! Há sempre um grafo subjacente a um autómato:

- Os vértices do grafo são os estados do autómato
- A relação de adjacência é induzida pela relação de transição: se existe uma transição do estado p para o estado q, então o vértice q será adjacente ao vértice p

Caminhos em grafos orientados

Num grafo (V, E), um *caminho* do vértice v_0 para o vértice v_k é uma sequência de vértices

$$\langle v_0, v_1, \ldots v_k \rangle$$

tais que $v_i \in V$ para todo o $i \in \{0, \ldots, k\}$, e $(v_i, v_{i+1}) \in E$ para todo o $i \in \{0, \ldots, k-1\}$

Alternativamente, este caminho pode ser visto como uma sequência de arestas

$$(v_0, v_1), (v_1, v_2), \ldots (v_{k-1}, v_k)$$

O *comprimento* deste caminho é o número k de arestas nele contidas.

Um vértice v é **alcançável** a partir do vértice s se existe um caminho de s para v. Num grafo orientado, isto não implica que s seja alcançável a partir de v.

Um **ciclo** é um caminho de comprimento ≥ 1 com início e fim no mesmo vértice. (Note-se que existe sempre um caminho de comprimento 0 de um vértice para si próprio, que não se considera ser um ciclo)

Um grafo diz-se *acíclico* se não contém ciclos. Um **grafo orientado acíclico** é usualmente designado por **DAG**, de *Directed Acyclic Graph*.

Voltando à correspondência entre autómatos e grafos, temos que:

- a aceitação de uma palavra corresponde a um caminho entre o estado inicial do autómato e um dos seus estados finais;
- um ciclo corresponde a uma sub-palavra que pode ser repetida um número arbitrário de vezes (estrela de Kleene)

Representação em Computador por Listas de Adjacências

Existem várias forma de representar grafos; uma muito popular consiste em guardar, para cada vértice, uma lista (possivelmente ordenada) dos seus vértices adjacentes. Em Python será muito conveniente representar um grafo por um *dicionário*, que associa a cada vértice a sua lista de adjacências.

O grafo acima poderia ser representado assim:

```
g = {}

g[1] = [2]
g[2] = [2,4,5]
g[3] = []
g[4] = [1,5]
g[5] = [4]
g[6] = [3]

```

Travessia de Grafos

Um algoritmo de **travessia** de um grafo (V, E), a partir de um vértice inicial $s \in V$ dado, visita **todos os vértices alcançáveis** a partir de s segundo uma determinada estratégia, **não passando mais do que uma vez por cada vértice.** Em particular *as travessias não percorrem ciclos*, caso existam.

Os caminhos percorridos durante a travessia constituem um sub-grafo de (V, E) que é de facto uma árvore, designada por **árvore de travessia** do grafo. Diferentes estratégias produzirão árvores diferentes.

Note-se que não se trata aqui de um algoritmo para a resolução de um problema específico, mas antes de uma família de algoritmos. Para **resolver um problema específico** haverá que:

1. seleccionar a estratégia de travessia adequada para esse problema

2. adaptar o esquema geral da travessia para a resolução do problema em causa

Travessia em Profundidade

Esta estratégia de travessia caracteriza-se pelo seguinte:

> *Todos os vértices adjacentes a um vértice u são visitados, por ordem, imediatamente a seguir a u*

Quer isto dizer que quando um vértice v adjacente a u é visitado, são em seguida visitados todos os que são alcancáveis a partir de v. Os outros adjacentes a u, "irmãos" de v, só serão visitados depois de todos estes, apesar de estarem mais próximos de u — ou seja, os vértices não são visitados por ordem crescente de distância à origem.

A implementação típica desta estratégia de travessia é *recursiva.*

Durante a execução de uma travessia, cada vértice de um grafo pode encontrar-se num de três estados, a que associaremos um código de cores:

- ainda não alcançado pela travessia [**BRANCO**]
- já alcançado, mas alguns dos seus vértices adjacentes ainda não alcançados [**CINZENTO**]
- já processado (todos os seus adjacentes foram alcançados) [**PRETO**]

Observe-se que

> *Os vértices cinzentos constituem uma fronteira entre os que já foram completamente tratados pela travessia e os que não foram ainda alcançados.*

Para o controlo da travessia, um algoritmo deve guardar esta informação de estado (num *array* indexado pelos vértices do grafo).

```
color = {}
for v in g:
    color[v] = 'WHITE'

def visit(g, s):
    color[s] = 'GRAY'
    for v in g[s]:
        if color[v] == 'WHITE':
            visit(g, v)
```

```
    color[s] = 'BLACK'

```

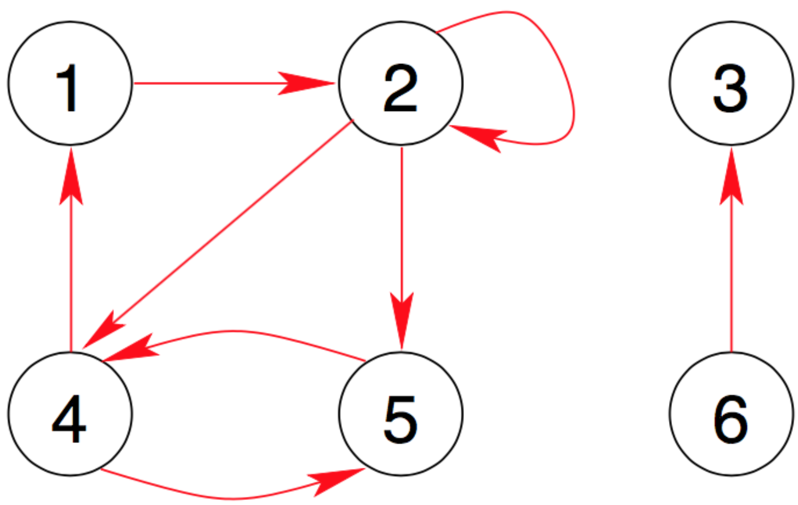
Note que:

- quando é encontrado um vértice adjacente cinzento, isso significa que foi detectado um ciclo durante a travessia

- quando é encontrado um vértice adjacente preto, isso significa que a travessia chegou a um vértice já visitado, mas isto não aconteceu ao percorrer um ciclo

Dependendo da aplicação, pode não ser importante distinguir entre os vértices CINZENTOS e PRETOS; neste caso a informação de estado será simplesmente Booleana (**não visitado** / **visitado**).

Exercício
Execute à mão a travessia do grafo apresentado em cima, iniciando no vértice 1. Repita o exercício iniciando a travessia no vértice 5.



Grafo orientado G1

Travessia em Profundidade de Todos os Caminhos

A noção tradicional de travessia, vista em cima, evita que o mesmo vértice possa ser visitado mais do que uma vez, mesmo não seja num ciclo. Por exemplo no grafo acima, visitamos 5 a partir de 4, e quando tentamos visitar 5 directamente a partir de 2 isso já não acontecerá, uma vez que 5 já foi visitado (e *pintado*).

Pode no entanto ser importante em aplicações concretas enumerar todos os caminhos sem ciclos de um grafo, e fazer uma travessia, por exemplo no grafo acima, procedesse pela seguinte ordem:
1 2 4 5 5 4.

Para isto basta voltar a pintar os vértices de branco, depois de visitados todos os seus adjacentes. Continua a ser possível detectar ciclos através da cor cinzento.

```
color = {}
for v in g:
    color[v] = 'WHITE'

def allPaths(g, s):
    color[s] = 'GRAY'
    print (s)
    for v in g[s]:
        if color[v] == 'WHITE':
            allPaths(g, v)
    color[s] = 'WHITE'

```