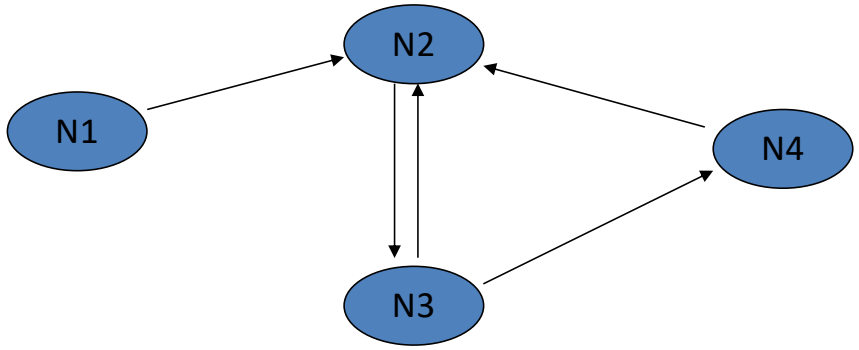
Grafos

Conceitos, representações, algoritmos

Definição de grafo

Um **grafo G=(V,E)** é uma estrutura matemática composta por:

- Um conjunto **V** de vértices ou **nós**
- Um conjunto **E** de pares de nós, designados por ramos, conexões, **ligações** ou **arcos**, cada um ligando dois nós. Estes arcos podem ter uma orientação (pares ordenados) ou não.



V = { N1, N2, N3, N4 }

E = { (N1,N2), (N2, N3), (N3,N2), (N4,N2), (N3,N4) }

Exemplo de um grafo orientado

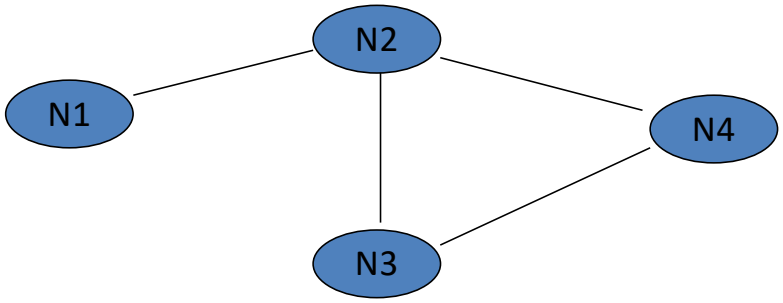
Grafos orientados e não orientados

Grafos **orientados**:

Grafos onde as ligações entre os nós (arcos) têm um sentido (elementos de E são pares ordenados)

Grafos **não orientados**:

Grafos onde os arcos não têm um sentido definido (elementos de E são pares não ordenados)



Exemplo de um grafo não orientado

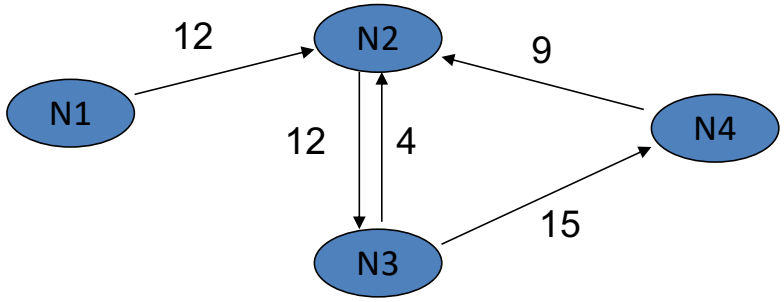
V = { N1, N2, N3, N4 }

E = { (N1,N2), (N2,N3), (N2,N4), (N3,N4) }

Grafos pesados

Como estruturas de dados, é comum os grafos terem associados aos nós e/ ou aos arcos informação de vários tipos

No caso mais comum, os arcos podem ter associados valores numéricos, comummente chamados de pesos (grafos **pesados**), sendo os elementos de E tripletos incluindo como último elemento o peso



V = { N1, N2, N3, N4 }

E = { (N1,N2,12), (N3,N2,4),
(N2,N3,12), (N4,N2,9), (N3,N4,15) }

Exemplo de um grafo pesado

Representações computacionais de grafos

Os grafos podem ser representados computacionalmente de várias formas distintas

A escolha depende do tipo de aplicação, do tipo de grafos, da densidade de ligações e dos algoritmos que se pretendem implementar sobre estes

Representações mais comuns:

- Matrizes
- Listas de adjacência

Matrizes de adjacência

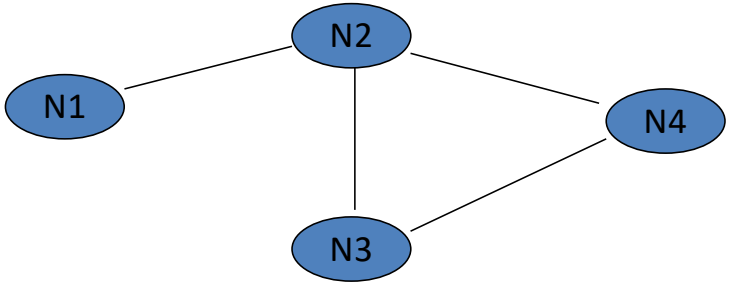
As matrizes de adjacência representam grafos como **matrizes** em que

- as linhas correspondem aos nós origem das ligações
- as colunas correspondem aos nós destino
- se existe ligação entre nó i e j, então o valor na matriz (M[i][j]) será 1; caso contrário, será 0
- em grafos não orientados, a matriz é simétrica podendo apenas representar-se uma matriz triangular
- no caso dos grafos pesados, os pesos nas ligações podem ser representados como valores na matriz

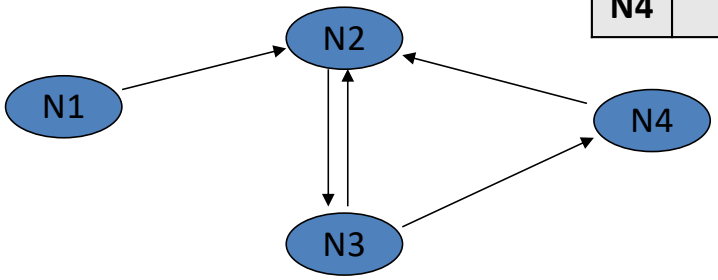
Esta representação traz vantagens em casos em que a densidade de ligação do grafo é grande, i.e. há um grande nº médio de ligações por nó e quando o nº de nós é baixo

Em casos onde a densidade é baixa e o nº de nós elevado, a matriz torna-se pesada em termos de memória

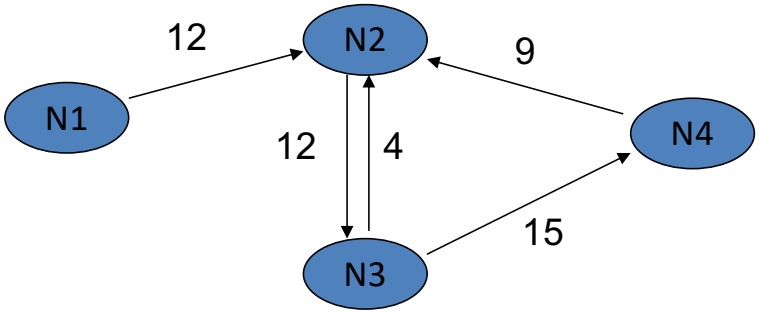
Matrizes de adjacência - exemplos



	N1	N2	N3	N4
N1	0	1	0	0
N2	0	0	1	0
N3	0	1	0	1
N4	0	1	0	0



	N1	N2	N3	N4
N1	0	1	0	0
N2		0	1	1
N3			0	1
N4				0



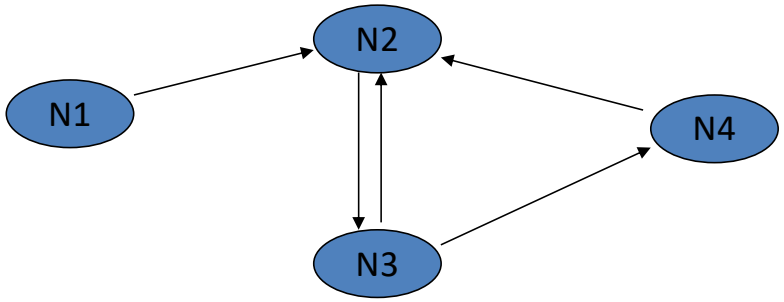
	N1	N2	N3	N4
N1	0	12	0	0
N2	0	0	12	0
N3	0	4	0	15
N4	0	9	0	0

Listas de adjacência

Apenas se representam as ligações existentes; para cada nó *i* temos uma lista com os arcos originados em *i*; representação mais compacta para grafos com número limitado de ligações.

Se o grafo é não orientado, poderá haver informação redundante representando-se ambos os sentidos da ligação

Esta representação pode ser adaptada representando-se cada nó como um objecto e as ligações como listas de referência a objetos desta classe.



Grafo G:

1 => [2]
2 => [3]
3 => [2,4]
4 => [2]

Nós adjacentes, sucessores e antecessores

Num grafo **orientado** G = (V,E):

- Um vértice *s* é **sucessor** do vértice *v* se existe em E o par ordenado *(v,s)*
- Um vértice *p* é **antecessor** do vértice v se existe em E o par ordenado *(p,v)*
- Se existe em E o par ordenado *(p,s)*, os vértices *p* e *s* dizem-se **adjacentes** (i.e. dois vértices são adjacentes se um é sucessor do outro)

Num grafo **não orientado** G=(V,E), os vértices x e y são adjacentes se existe em E o par não ordenado (x,y) (i.e. existe (x,y) ou (y,x))

Grau de um nó

O **grau** de um nó é definido como o número de ligações que ligam esse nó a outros nós, i.e. é o nº de nós adjacentes

Se grafo é **orientado**, podem definir-se:

- o grau de **entrada**: número de ligações que chegam a esse nó (nº de predecessores)
- o grau de **saída**: número de ligações que saem desse nó (nº de sucessores)

Para o grafo completo pode definir-se:

- **Grau médio <k>**: média do grau calculada sobre todos os nós
- **Distribuição do grau P(k)**: probabilidade que um nó tenha grau k, calculadas para cada valor de k = 1, ..., nº de nós do grafo

Implementando grafos

Vamos criar uma classe para representar grafos **orientados** (note-se que podemos sempre representar grafos não orientados colocando ambos os sentidos da ligação)

A representação será dada por **listas de adjacência**

Será usado um dicionário para representar o grafo onde

- as **chaves** são os identificadores dos **nós**
- os **valores** representam os **arcos**, indicando uma lista de nós que estão ligados ao nó chave

Implementando grafos

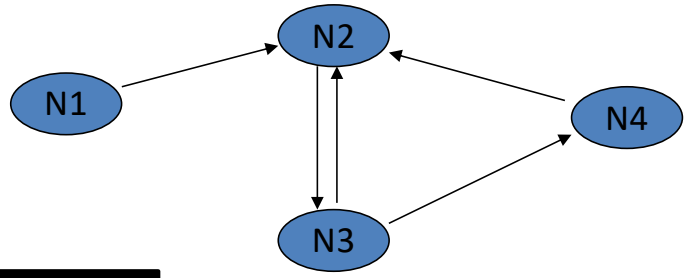
```
class MyGraph:
    def __init__(self, g = { } ):
        self.graph = g

    def print_graph(self):
        for v in self.graph.keys():
            print (v, " -> ", self.graph[v])

    def get_nodes(self):
        return list(self.graph.keys() )

    def get_edges(self):
        edges = [ ]
        for v in self.graph.keys():
            for d in self.graph[v]:
                edges.append((v,d))
        return edges
```

Único atributo:
graph – dicionário

Imprime grafo:
Cada linha: nó e seus vizinhos

Lista de arcos é retornada como lista de pares (tuplos)

Implementando grafos



test1

```
gr = MyGraph( {1:[2], 2:[3], 3:[2,4], 4:[2]} )
gr.print_graph()
print (gr.get_nodes())
print (gr.get_edges())
```

```
1 -> [2]
2 -> [3]
3 -> [2, 4]
4 -> [2]
[1, 2, 3, 4]
[(1, 2), (2, 3), (3, 2), (3, 4), (4, 2)]
```

Implementando grafos

```
def add_vertex(self, v):
    ...

def add_edge(self, o, d):
    ...
```

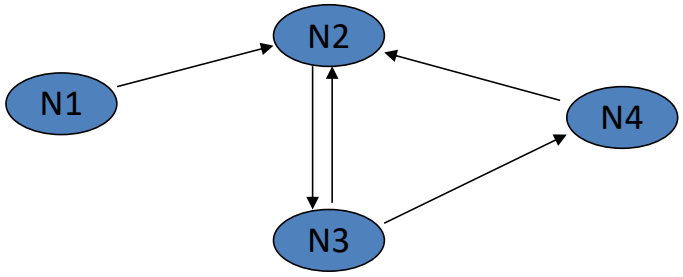
add_vertex – adiciona o nó v ao grafo

add_edge – adiciona o arco *(o,d)* ao grafo

Se os nós *o* ou *d* não existirem adiciona-os ao grafo

Implementando grafos

```
def add_vertex(self, v):
    if v not in self.graph.keys():
        self.graph[v] = [ ]

def add_edge(self, o, d):
    if o not in self.graph.keys():
        self.add_vertex(o)
    if d not in self.graph.keys():
        self.add_vertex(d)
    if d not in self.graph[o]:
        self.graph[o].append(d)
```



test2

```
gr2 = MyGraph()
gr2.add_vertex(1)
gr2. add_vertex(2)
gr2.add_vertex(3)
gr2.add_vertex(4)

gr2.add_edge(1,2)
gr2.add_edge(2,3)
gr2.add_edge(3,2)
gr2.add_edge(3,4)
gr2.add_edge(4,2)

gr2.print_graph()
```

Implementando grafos: graus

```
def get_successors(self, v)
    ...

def get_predecessors(self, v)
    ...

def get_adjacents(self, v):
    ...

def out_degree(self, v):
    ...

def in_degree(self, v):
    ...

def degree(self, v):
    ...
```

get_successors – dá lista de nós sucessores do nó v

get_predecessors – dá lista de nós antecessores do nó v

get_adjacents – dá lista de nós adjacentes do nó v

out_degree– calcula grau de saída do nó v

in_degree– calcula grau de entrada do nó v

degree– calcula grau do nó v (todos os nós Adjacentes quer percursores quer sucessores)

Implementando grafos: graus

```
def get_successors(self, v):
        return list(self.graph[v])

def get_predecessors(self, v):
        res = [ ]
        for k in self.graph.keys():
                if v in self.graph[k]:
                        res.append(k)
        return res

def get_adjacents(self, v):
        suc = self.get_successors(v)
        pred = self.get_predecessors(v)
        res = pred
        for p in suc:
            if p not in res: res.append(p)
        return res
```

```
def out_degree(self, v):
        return len(self.graph[v])

def in_degree(self, v):
        return len(self.get_predecessors(v))

def degree(self, v):
        return len(self.get_adjacents(v))
```

```
gr = MyGraph( {1:[2], 2:[3], 3:[2,4], 4:[2]} )
gr.print_graph()
print (gr.get_successors(2))
print (gr.get_predecessors(2))
print (gr.get_adjacents(2))
print (gr.in_degree(2))
print (gr.out_degree(2))
print (gr.degree(2))
```

test3

Caminhos

Num grafo orientado **G = (V,E)**, um **caminho P** entre dois nós *x* e *y* é definido como uma sequência de nós $P_1, P_2, ..., P_n$ onde $P_1 = x$, $P_n = y$ e todos os pares ordenados (P_i, P_{i+1}) são arcos pertencentes ao grafo (i.e. pertencem a E), sendo **n-1** o comprimento do caminho P

Num grafo **não orientado**, a definição é semelhante sendo que a condição é a de que o par não ordenado (P_i, P_{i+1}) pertence ao grafo (note que num par não ordenado não interessa a ordem dos nós)

Nós atingíveis

Para um dado nó origem v, os nós **atingíveis** são aqueles para os quais existe um caminho com origem em v e final nesse mesmo nó

Para identificar todos os nós atingíveis de um nó origem é necessário realizar uma **travessia** do grafo, onde o algoritmo passa por:

- Iniciar no nó origem
- Visitar todos os sucessores do nó origem, todos os sucessores desses nós, e assim sucessivamente até visitar todos os nós atingíveis

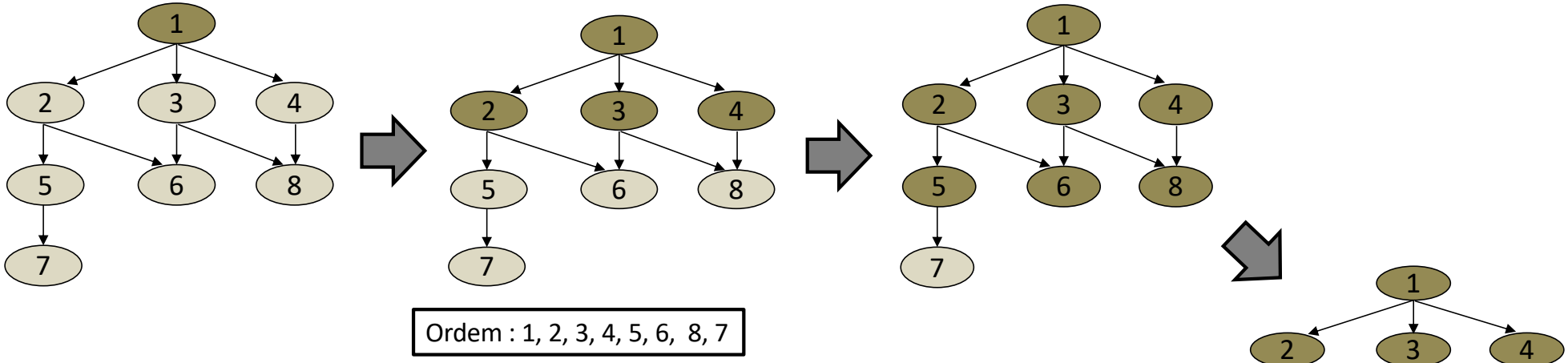
Travessia do grafo

Duas estratégias podem ser usadas para definir a ordem de exploração dos nós numa travessia:

- Em **largura**: começa pelo nó origem, depois explora todos os seus sucessores, depois os sucessores destes, e assim sucessivamente até todos os nós atingíveis terem sido explorados
- Em **profundidade**: começa pelo nó origem e explora o 1º sucessor, seguido pelo 1º sucessor deste e assim sucessivamente até não haver mais sucessores e ter que se fazer "backtracking"

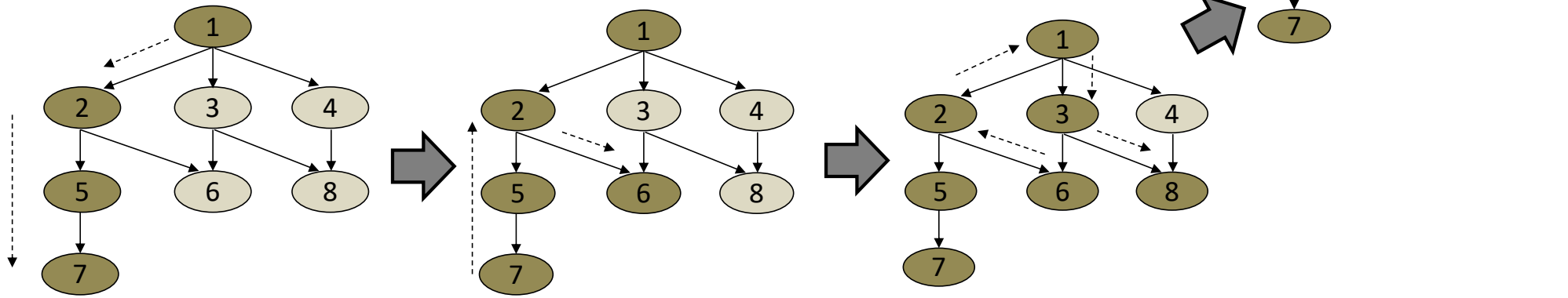
Podem ser implementadas guardando nós a explorar em estruturas de dados como *queue* (largura) e *stack* (profundidade).

BFS – em largura



Ordem : 1, 2, 3, 4, 5, 6, 8, 7

DFS – em profundidade



Ordem: 1, 2, 5, 7, 6, 3, 8, 4

Implementando grafos: travessias

```
def reachable_bfs (self, v):
    l = [v]
    res = []
    while len(l) > 0:
        node = l.pop(0)
        if node != v: res.append(node)
        for elem in self.graph[node]:
            if elem not in res and elem not in l and elem != node:
                l.append(elem)
    return res

def reachable_dfs (self, v):
    l = [v]
    res = []
    while len(l) > 0:
        node = l.pop(0)
        if node != v: res.append(node)
        for elem in self.graph[node]:
            if elem not in res and elem not in l:
                l.insert(0, elem)
    return res
```

```
gr2 = MyGraph( {1:[2,3], 2:[4], 3:[5], 4:[], 5:[]} )
print (gr2.reachable_bfs(1))
print (gr2.reachable_dfs(1))
```

Caminhos mais curtos / distâncias

O **caminho mais curto** entre dois nós define-se como o caminho P entre esses nós cujo comprimento (nº de nós) seja o menor de entre todos os possíveis caminhos entre eles

O caminho mais curto pode ser encontrado com uma travessia do grafo em largura terminada quando o nó desejado é atingido

A **distância** entre dois nós define-se como o comprimento do caminho mais curto entre ambos, i.e. o número de nós visitados (incluindo o destino); se não existir nenhum caminho a distância é considerada infinita

Implementando grafos: distância

```
def distance(self, s, d):
        …


def shortest_path(self, s, d):
        …



def reachable_with_dist(self, v):
        …
```

distance – retorna distância entre nós s e d

shortest_path– retorna caminho mais curto entre s e d (lista de nós por onde passa)

reachable_with_dist– retorna lista de nós atingíveis a partir de v com respetiva distância (lista de pares nó, distância)

Implementando grafos: travessias

```
def distance(self, s, d):
    if s == d: return 0
    l = [(s,0)]
    visited = [s]
    while len(l) > 0:
        node, dist = l.pop(0)
        for elem in self.graph[node]:
            if elem == d: return dist + 1
            elif elem not in visited:
                l.append((elem,dist+1))
                visited.append(elem)
    return None
```

Usa queue de nós, juntando valor com a distância (tuplo)

Retorna None se não é atingível

test4

```
gr = MyGraph( {1:[2], 2:[3], 3:[2,4], 4:[2]} )
print (gr.distance(1,4))
print (gr.distance(4,3))

gr2 = MyGraph( {1:[2,3], 2:[4], 3:[5], 4:[], 5:[]} )
print (gr2.distance(2,1))
print (gr2.distance(1,5))
```

Implementando grafos: travessias

```
def shortest_path(self, s, d):
    if s == d: return []
    l = [(s,[])]
    visited = [s]
    while len(l) > 0:
        node, preds = l.pop(0)
        for elem in self.graph[node]:
            if elem == d: return preds+[node,elem]
            elif elem not in visited:
                l.append((elem,preds+[node]))
                visited.append(elem)
    return None
```

Usa queue de nós, juntando lista com o caminho (tuplo)

test4

```
gr = MyGraph( {1:[2], 2:[3], 3:[2,4], 4:[2]} )
print (gr.shortest_path(1,4))
print (gr.shortest_path(4,3))

gr2 = MyGraph( {1:[2,3], 2:[4], 3:[5], 4:[], 5:[]} )
print (gr2.shortest_path(1,5))
print (gr2.shortest_path(2,1))
```

Implementando grafos: travessias

```
def reachable_with_dist(self, s):
    res = [ ]
    l = [(s,0)]
    while len(l) > 0:
        node, dist = l.pop(0)
        if node != s: res.append((node,dist))
        for elem in self.graph[node]:
            if not is_in_tuple_list (l,elem) and not is_in_tuple_list (res,elem):
                l.append((elem,dist+1))
    return res

def is_in_tuple_list (tl, val):
    res = False
    for (x,y) in tl:
        if val == x: return True
    return res
```

test4

```
gr = MyGraph( {1:[2], 2:[3], 3:[2,4], 4:[2]} )
print (gr.reachable_with_dist(1))
print (gr.reachable_with_dist(3))

gr2 = MyGraph( {1:[2,3], 2:[4], 3:[5], 4:[], 5:[]} )
print (gr2.reachable_with_dist(1))
print (gr2.reachable_with_dist(5))
```

Ciclos e grafos cíclicos/ acíclicos

Um caminho é chamado de fechado se começa e termina no mesmo nó do grafo

Um caminho fechado chama-se de **ciclo** (simples) se não tem vértices ou arcos repetidos

Num grafo orientado, qualquer caminho fechado inclui um ciclo (simples)

Um grafo que contém ciclos é designado por **cíclico**, enquanto um grafo que não contém ciclos é designado por **acíclico**

Os grafos orientados acíclicos (designados por DAGs) são bastante importantes e têm propriedades matemáticas únicas (como a ordenação topológica)

As árvores são casos particulares de grafos acíclicos em que os nós são todos ligados

Implementando grafos: ciclos

```
def node_has_cycle (self, v):
    l = [v]
    res = False
    visited = [v]
    while len(l) > 0:
        node = l.pop(0)
        for elem in self.graph[node]:
            if elem == v: return True
            elif elem not in visited:
                l.append(elem)
                visited.append(elem)
    return res

def has_cycle(self):
    res = False
    for v in self.graph.keys():
        if self.node_has_cycle(v): return True
    return res
```

test5

```
gr = MyGraph( {1:[2], 2:[3], 3:[2,4], 4:[2]} )
print (gr.node_has_cycle(2))
print (gr. node_has_cycle(1))
print (gr.has_cycle())

gr2 = MyGraph( {1:[2,3], 2:[4], 3:[5], 4:[], 5:[]} )
print (gr2. node_has_cycle(1))
print (gr2.has_cycle())
```

Exercício (avaliação contínua)

- Use o código atual da classe MyGraph como base para a implementação de uma classe que implemente grafos orientados pesados (i.e. que tenham um peso numérico associado a cada arco)
- Altere a representação para permitir representar um peso associado a cada arco (sugestão: use uma lista de tuplos onde o primeiro elemento é o nó destino e o segundo é o peso)
- Adapte os algoritmos de procura do caminho mais curto e distância, para retornarem o caminho com menor peso (sendo o peso de um caminho a soma dos pesos dos arcos que o compõem). Sugestão: use o algoritmo de Dijkstra !