Grafos

Implementação

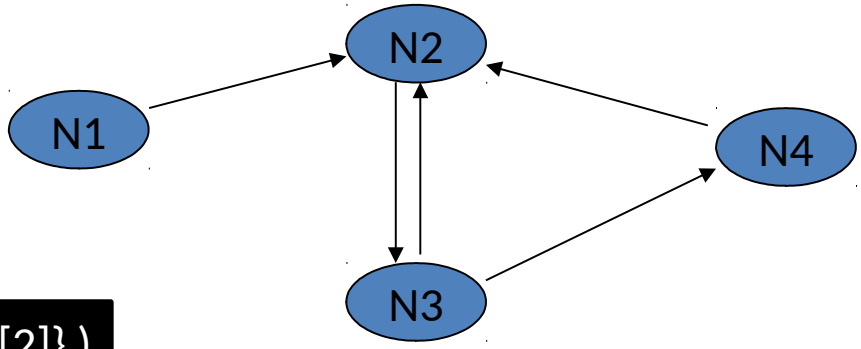
Implementando grafos

- 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

- Cria os seguintes métodos:
  - **print_graph** : imprime o grafo
  - **get_nodes**: retorna a lista de nós
  - **get_edges**: retorna a lista de arcos como lista de pares (tuplos)
  - **add_node** : adiciona o nó v ao grafo, assume que o nó não tem ligação.
  - **add_edge** – adiciona o arco *(o,d)* ao grafo
    - Se os nós *o* ou *d* não existirem adiciona-os ao grafo.

Implementando grafos




```
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)]
```

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

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

gr2.printGraph()
```

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

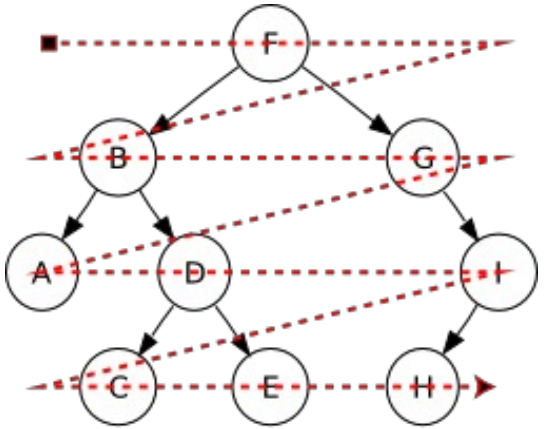
out_degree– calcula grau de saída do nó v

in_degree– calcula grau de entrada do nó v

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

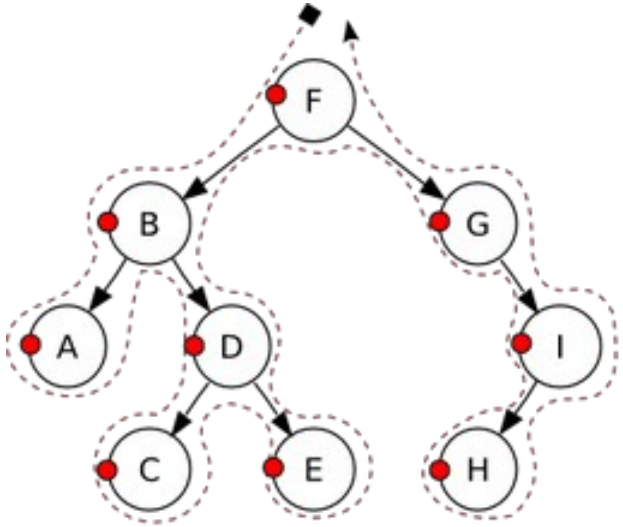
Implementando grafos: travessias

- **reachableBFS (node)** : 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



Ordem: F, B, G, A, D, I, C, E, H

- **reachableDFS (node**: 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"



Pré-ordem: F, B, A, D, C, E, G, I, H

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

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

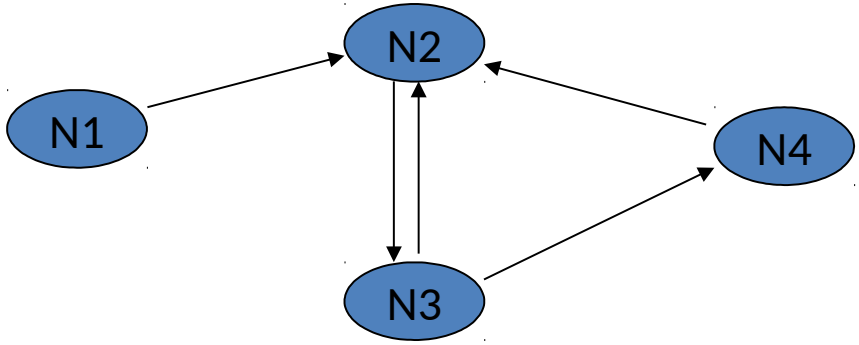
shortestPath– retorna caminho mais curto entre s e d (lista de nós por onde passa)

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

Implementando grafos: ciclos

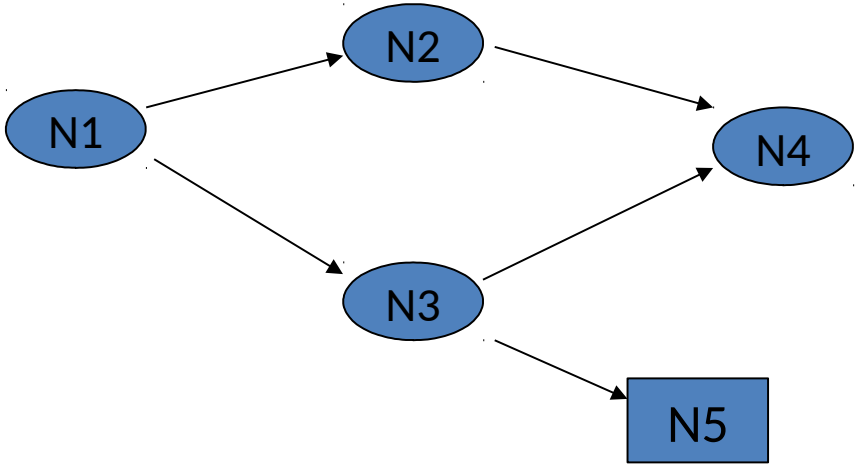
```
def node_has_cycle (self, v):
...

def has_cycle(self):
..
```




```
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

- 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). Procure informação disponível sobre o algoritmo de Dijkstra !