Programação II
+
Estruturas de Dados para Bioinformática

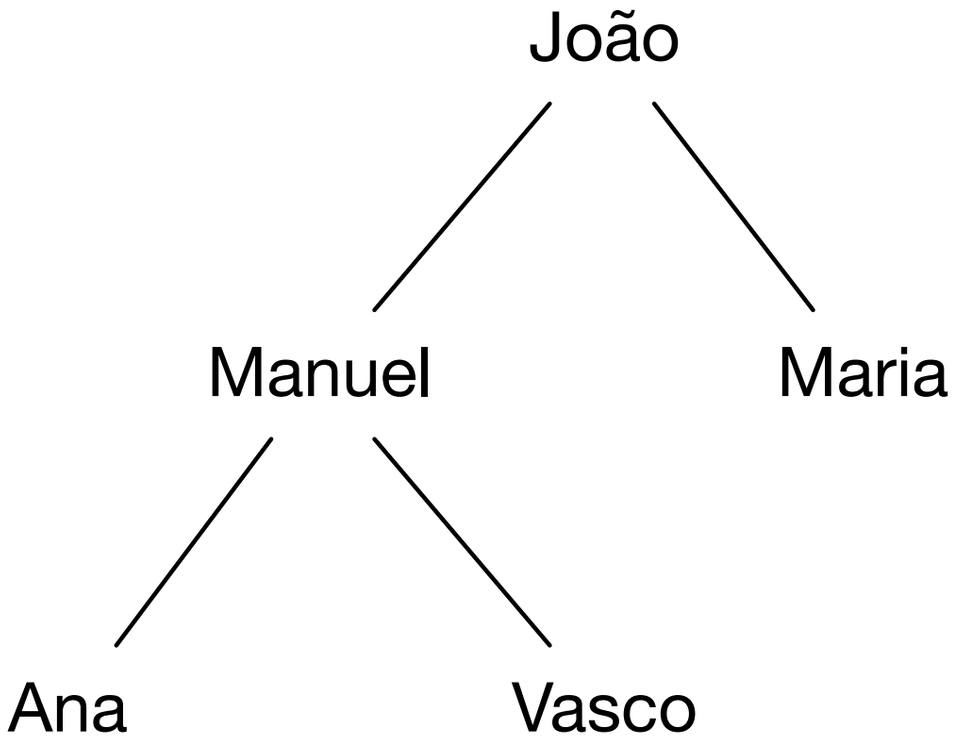
Análise de grafos (NetworkX)

Hugo Pacheco

DCC/FCUP
22/23

Árvores

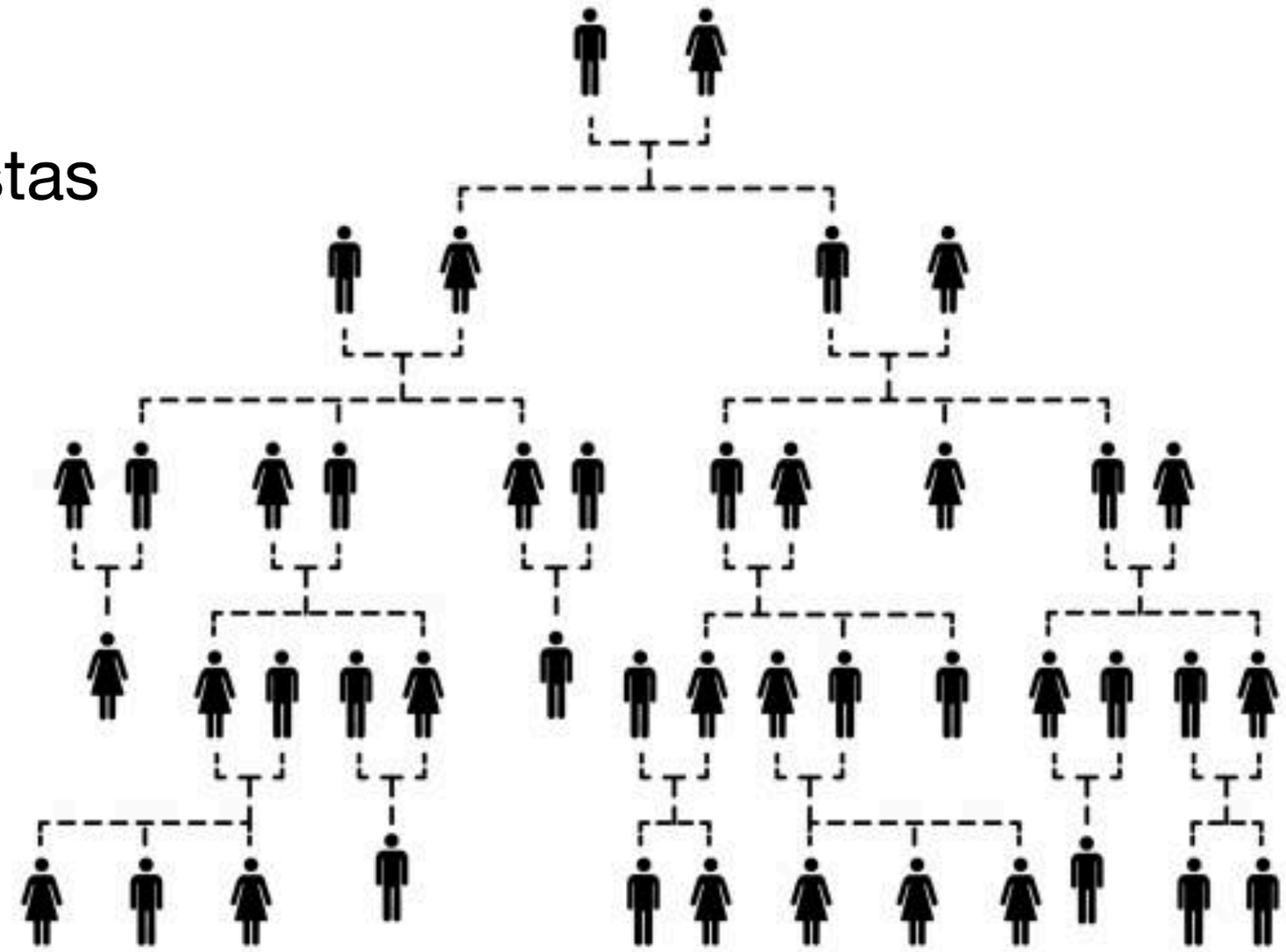
- Já vimos vários exemplos de dados hierárquicos / em árvore (e.g., JSON)
  - Pai guarda uma lista descendente de filhos, e assim consecutivamente




```
{
    name: "João",
    children: [{
        nome: "Manuel",
        children: [{
            name: "Ana",
            children: []
        }, {
            name: "Vasco",
            children: []
        }]
    }, {
        nome: "Maria",
        children: []
    }]
}
```

Árvores

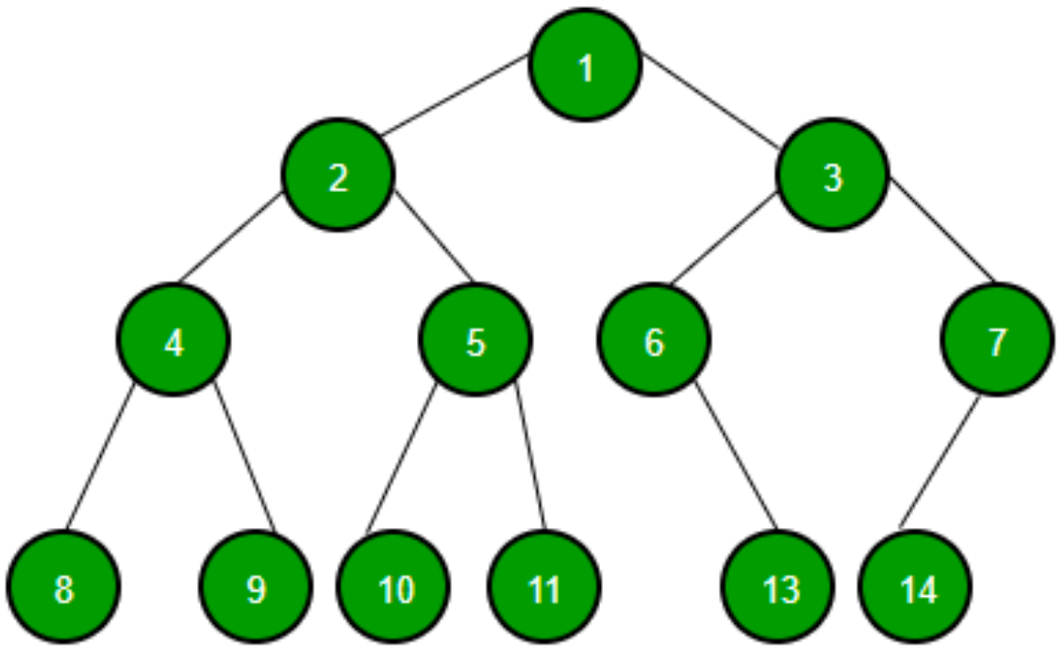
- E para árvores mais complicadas?
  - Relações pai/mãe
  - Padrastos/madrastas
  - …



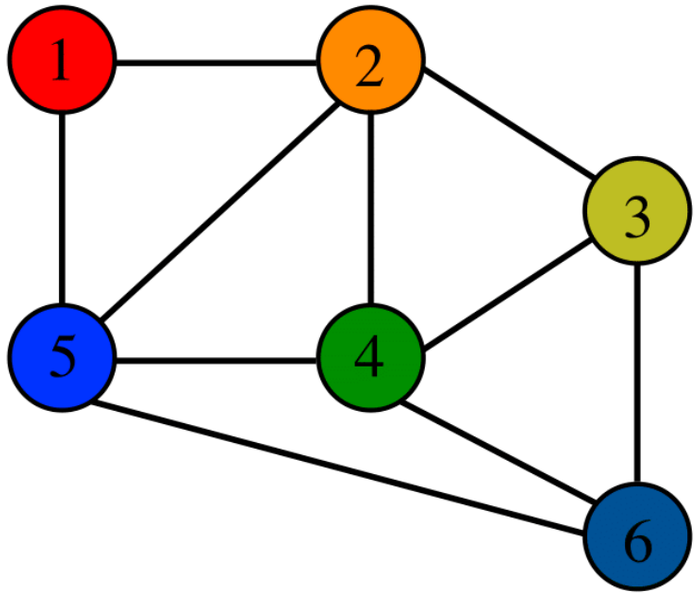
Grafos

- Um *grafo* é uma coleção de *nodos* interligados
- Tipicamente usado para representar redes
- Uma generalização de árvores

Árvore: hierarquia

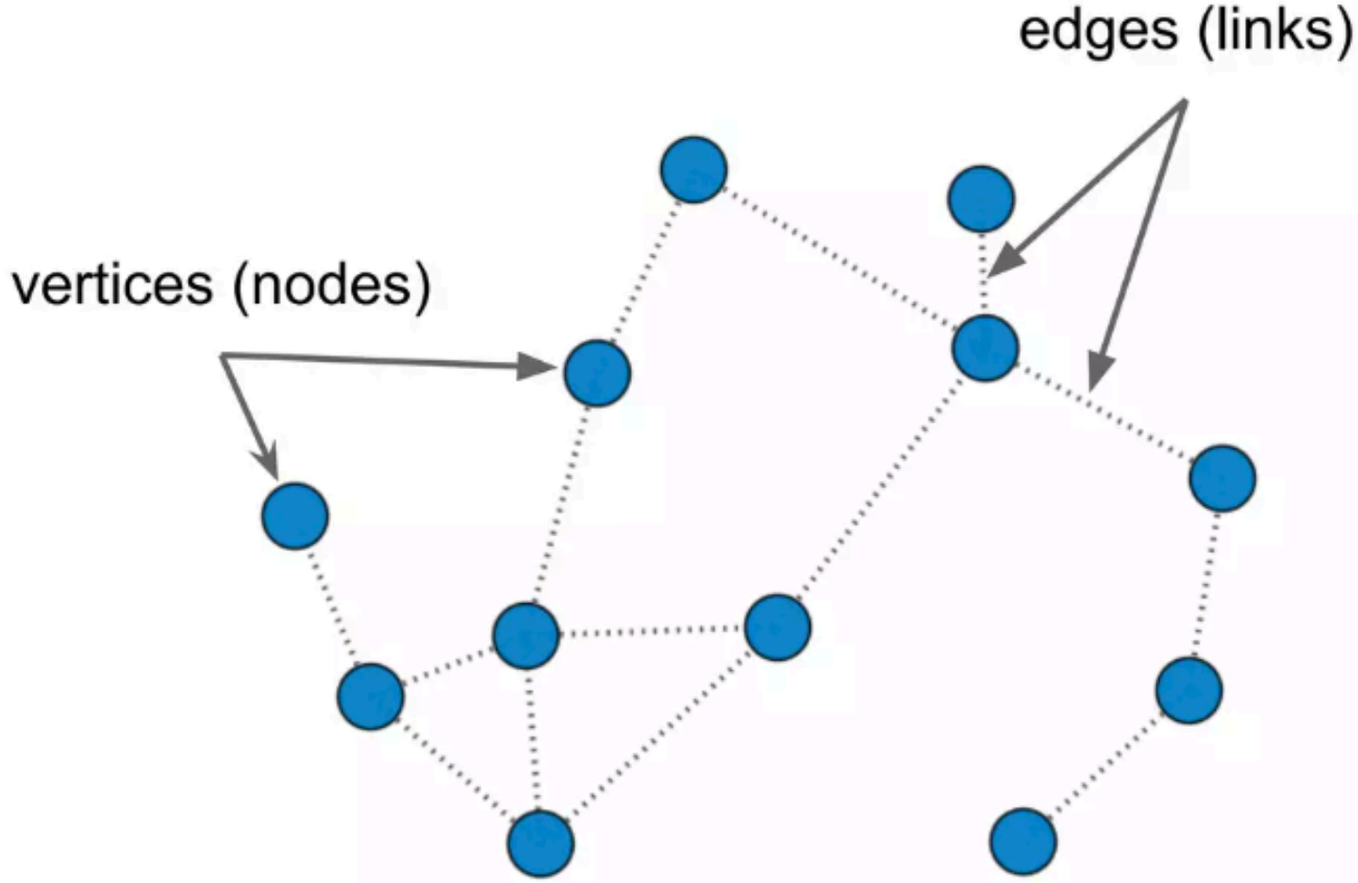


Grafo: qualquer ligação



Grafos

- Um conjunto de vértices e um conjunto de arestas
  - Vértices têm identificadores únicos no grafo
  - Arestas ligam vértices



Grafos (*NetworkX*)

- Podemos construir um grafo a partir de uma sequência de arestas: nodos implicitamente criados

- Por defeito grafo não-direcionado: arestas em ambos os sentidos

```
import networkx as nx

# lista de arestas
g1 = nx.Graph([(1,2),(2,3)])
# dicionário de adjacências
g2 = nx.Graph({1:[2,3],2:[1,4]})

print(g1.nodes()) # [1, 2, 3]
print(g2.edges()) # [(1, 2), (1, 3), (2, 4)]
print(g2.has_edge(2,1)) # True
```

Nodos podem ser qualquer valor *hashable*.
Intuição: qualquer valor de um tipo imutável!

Grafos (*NetworkX*)

- Grafos direcionados: arestas num sentido específico
- Uma ligação nos dois sentidos pode ser modelada explicitamente por duas arestas

```
import networkx as nx

# lista de arestas
g1 = nx.DiGraph([(1,2),(2,3)])
# dicionário de adjacências
g2 = nx.DiGraph({1:[2,3],2:[1,4]})

print(g1.nodes()) # [1, 2, 3]
print(g2.edges()) # [(1, 2), (1, 3), (2, 4)]
print(g2.has_edge(2,1)) # True
print(g2.has_edge(3,1)) # False
```

NetworkX (nodos)

- Podemos adicionar ou remover nodos de um grafo

```
import networkx as nx
g = nx.Graph()

g.add_node('a')
g.add_nodes_from([True,(4.5,'c')])
print(g.nodes()) # ['a', True, (4.5, 'c')]

g.remove_node('c') # fails, node does not exist
g.remove_nodes_from(['a',(4.5,'c')])
print(g.nodes()) # [True]
```

NetworkX (nodos)

- Nodos podem ter atributos associados (uma espécie de dicionário)

```
import networkx as nx
g = nx.Graph()

g.add_node(1,color="blue")
print(g.nodes()) # [1]
print(g.nodes()[1]) # {'color': 'blue'}

g.nodes()[1] = {'color':'red'} # fails, not assignable
g.nodes()[1]['color']='red'
print(g.nodes()[1]) # {'color': 'red'}

g.add_nodes_from([(4, {"color": "red"}),\
                  (5, {"color": "green"})])
print(g.nodes()) # [1, 4, 5]
print(g.nodes()[4]) # {'color': 'red'}
```

NetworkX (arestas)

- Podemos adicionar ou remover arestas de um grafo

```
import networkx as nx
g = nx.Graph()

g.add_edge(1, 2)
print(g.nodes(),g.edges()) # [1,2] [(1,2)]

g.add_edges_from([(2,3),(3,4)])
print(g.nodes(),g.edges()) # [1, 2, 3, 4] [(1, 2), (2, 3), (3, 4)]

g.remove_node(2)
print(g.nodes(),g.edges()) # [1, 3, 4] [(3, 4)]

g.remove_edge(3,4)
print(g.nodes(),g.edges()) # [1, 3, 4] []
```

NetworkX (arestas)

- Arestas podem ter atributos associados (uma espécie de dicionário)

```
import networkx as nx
g = nx.Graph()

g.add_edge(1, 2, weight=4.7 )
print(g.edges()) # [(1, 2)]

g.add_edges_from([(3, 4), (4, 5)], color='red')
print(g.edges()) # [(1, 2), (3, 4), (4, 5)]
print(g.edges()[3,4]) # {'color': 'red'}

g.add_edges_from([(1, 2, {'color': 'blue'}), (1, 3, {'weight': 8})])
print(g.edges[1,2]) # {'weight': 4.7, 'color': 'blue'}

g.edges[3, 4]['weight'] = 4.2
print(g.edges[3,4]) # {'color': 'red', 'weight': 4.2}
```

NetworkX (grau)

- O grau de cada nodo é o número de arestas ligadas a esse nodo
- Para grafos direcionados, temos grau de entrada e de saída

```
import networkx as nx

g1 = nx.Graph([('a','b'),('a','c'),('b','d'),('d','a')])
print(g1.degree)
# [('a', 3), ('b', 2), ('c', 1), ('d', 2)]
print(g1.degree['b'])
# 2

g2 = nx.DiGraph([('a','b'),('a','c'),('b','d'),('d','a')])
print(g2.in_degree)
# [('a', 1), ('b', 1), ('c', 1), ('d', 1)]
print(g2.out_degree)
# [('a', 2), ('b', 1), ('c', 0), ('d', 1)]
```

NetworkX (adjacências)

- Um grafo é essencialmente uma lista de adjacências
- Podemos aceder às adjacências de um vértice

Grafos não direcionados: arestas duplicadas
Grafos direcionados: apenas arestas de saída

```
g1 = nx.Graph([('a','b'),('a','c'),('b','d'),('d','a')])
print(g1.adj) # {'a': {'b': {}, 'c': {}, 'd': {}}, 'b': {'a':
{}, 'd': {}}, 'c': {'a': {}}, 'd': {'b': {}, 'a': {}}}
print(g1.adj.items())
print(g1['a']) # {'b': {}, 'c': {}, 'd': {}}
g1['b']['a']['color'] = 'red'

g2 = nx.DiGraph([('a','b'),('a','c'),('b','d'),('d','a')])
print(g2.adj) # {'a': {'b': {}, 'c': {}}, 'b': {'d': {}}, 'c':
{}, 'd': {'a': {}}}
print(g2['a']) # {'b': {}, 'c': {}}
print(g2.succ) # igual a g2.adj
print(g2.pred) # arestas invertidas
```

NetworkX (operações)

- Formalmente, grafos capturam relações binárias entre conjuntos
- Podemos definir operações matemáticas sobre grafos
  - Subgrafo

```
def is_subgraph(g1,g2):
    return set(g1.nodes()).issubset(set(g2.nodes()))
       and set(g1.edges()).issubset(set(g2.edges()))
```

  - União

```
g12 = nx.compose(g1,g2) # união (sets de nodos arbitrários)
g12 = nx.union(g1,g2) # união disjunta (sets de nodos distintos)
```

  - Interseção

```
g12 = nx.intersection(g1,g2) # ignora atributos
```

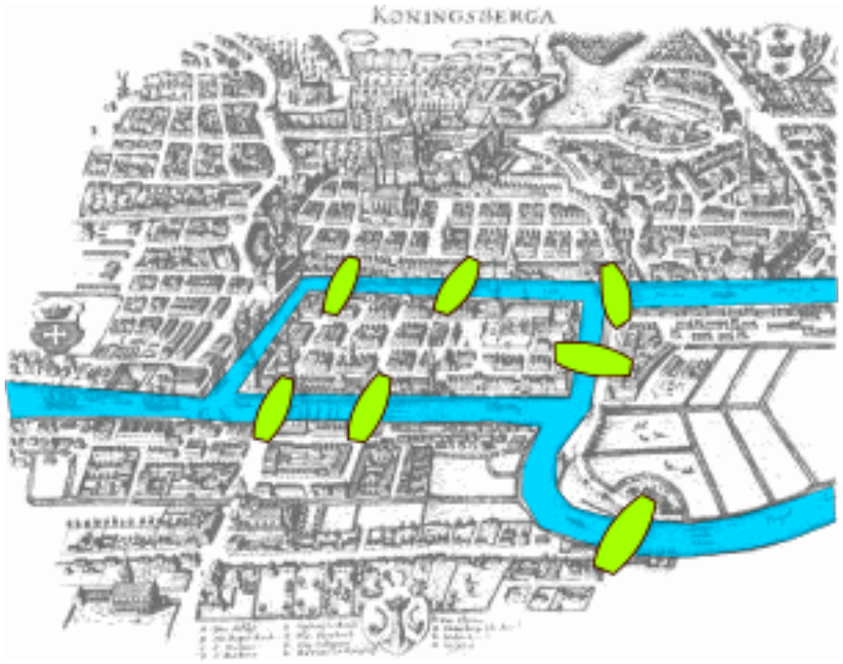
  - Diferença

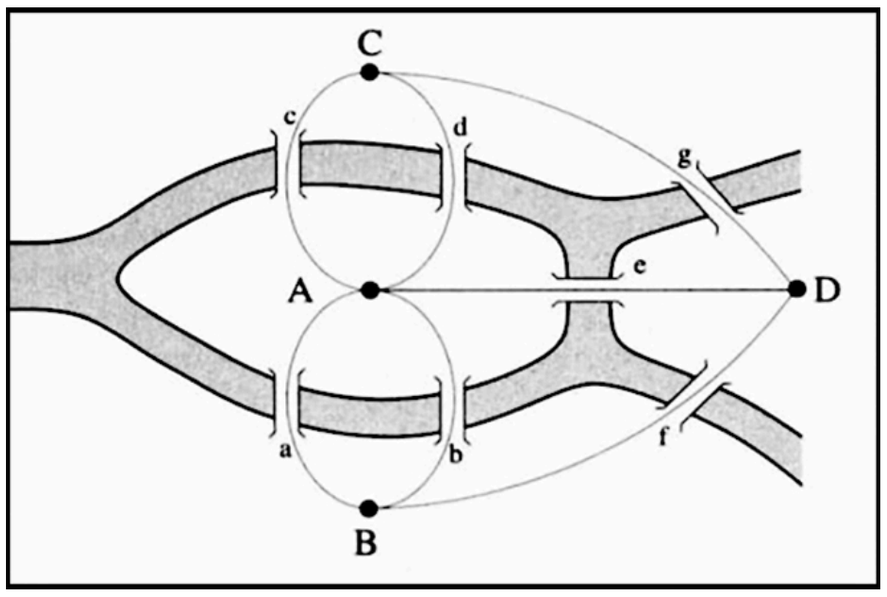
```
g12 = nx.difference(g1,g2) # nodos têm que ser os mesmos; ignora atributos
```

Teoria de Grafos

- Muitos problemas computacionais são descritos e resolvidos com grafos
- Exemplo clássico: 7 pontes de Königsberg

“Existe uma travessia que passe por todas as pontes exatamente uma vez?”





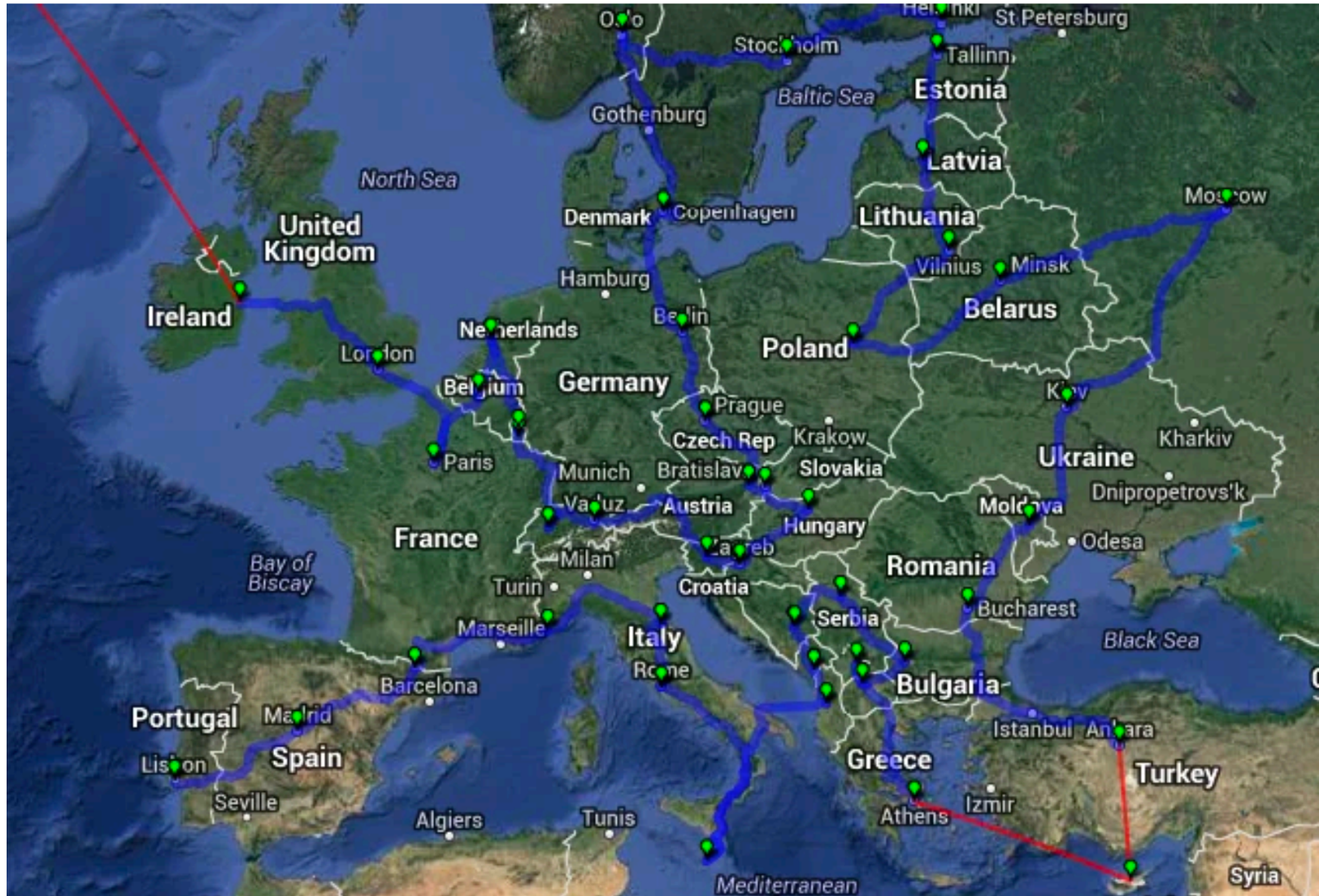
Solução matemática: se e só se existem no máximo dois vértices com grau ímpar

Teoria de Grafos

- Muitos problemas computacionais são descritos e resolvidos com grafos
- Exemplo clássico: caixeiro viajante

“Qual o itinerário mais curto que passa por todas as cidades exatamente uma vez?”

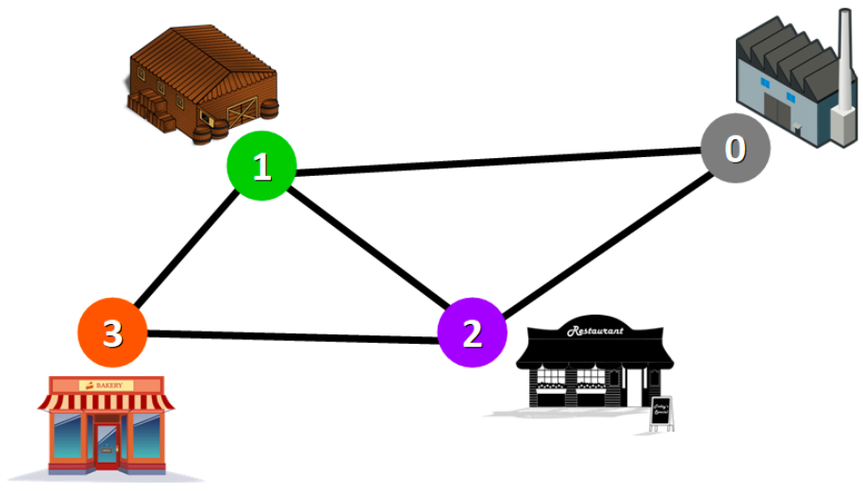
Problema complexo, soluções aproximadas, e.g., Capitais Europeias



NetworkX (algoritmos)

- Implementar algoritmos de grafos é complexo
  - Solução básica é ineficiente, e mesmo assim não trivial

Sabem codificar este grafo em Python e implementar uma função que decide se existe caminho entre dois pontos?



  - Muitas optimizações heurísticas na prática

- A biblioteca NetworkX oferece implementações de vários algoritmos de grafos: documentação

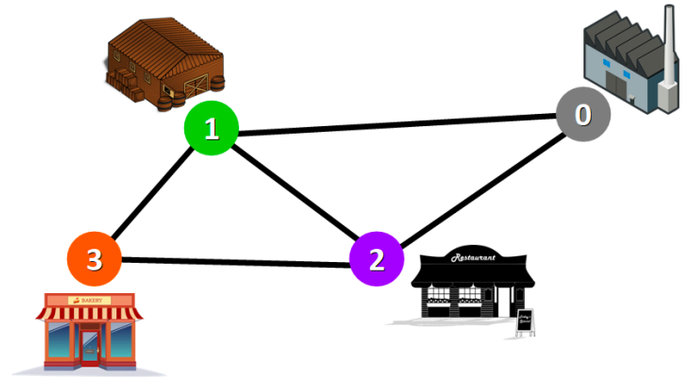
NetworkX (caminhos)

- Pergunta mais frequente em grafos: existe caminho entre dois nodos?
- Um caminho é uma ligação direta ou indireta entre nodos

```
g = nx.Graph()
g.add_nodes_from([(0,{'name':'factory'}),(1,{'name':'farm'}),(2,
{'name':'restaurant'}),(3,{'name':'bakery'})])
g.add_edges_from([(0,1),(0,2),(1,2),(1,3),(2,3)])
gns = g.nodes(data=True) # [(0, {'name': 'factory'}), ...]

def findName(name):
    return [id for id,att in gns if att['name']==name][0]
bakery = findName('bakery') # 3
factory = findName('factory') # 0

print(g.has_edge(bakery,factory)) # False
print(nx.has_path(g,bakery,factory)) # True
```



Exemplo (Gene Ontology)

- Criar um grafo de relações entre identificadores GO:id

```
import networkx as nx

g = nx.Graph()
with open('../../../dados/PZ.annot.txt','r') as f:
    linhas = f.read().splitlines()

pzs = {}; names = {}
for linha in linhas:
    pz,go,name = linha.split('\t')
    pzs[pz] = pzs.get(pz, []) + [go]
    g.add_node(go)
    names[name] = names.get(name, []) + [go]

for pz,gos in pzs.items():
    g.add_edges_from([(go1,go2,{'same':'pz'})\
                      for go1 in gos for go2 in gos if go1!=go2])
for name,gos in names.items():
    g.add_edges_from([(go1,go2,{'same':'name'})\
                      for go1 in gos for go2 in gos if go1!=go2])
```

Exemplo (Gene Ontology)

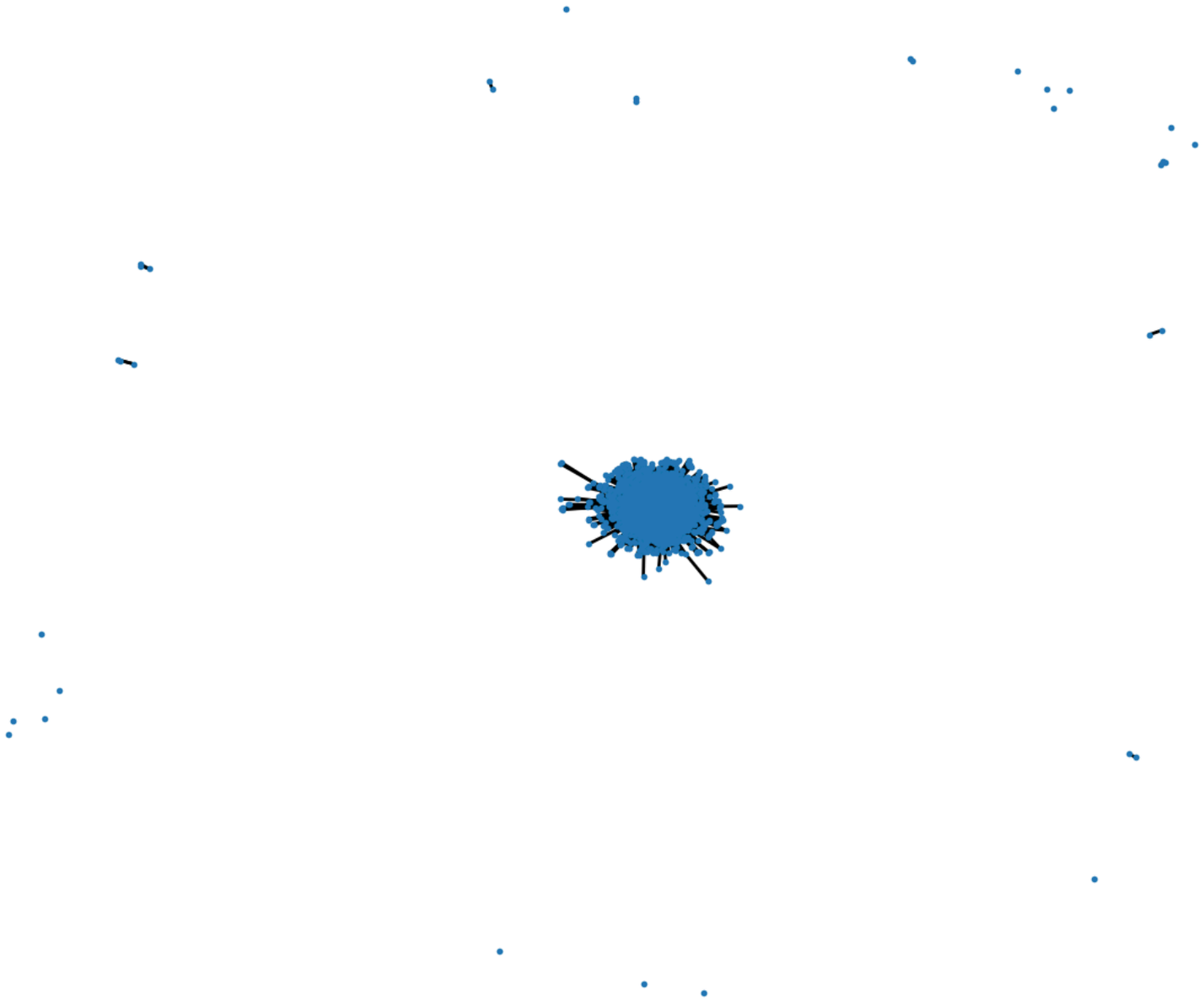
- Caminho mais curto entre dois identificadores
- Subentende uma noção de distância entre nodos (atributo *weight* em cada aresta)
    - Por defeito todas as arestas têm distância 1
    - conceito generalizável (e.g., arestas são estradas e *weight* = distância em kms)

```
import networkx as nx

p = nx.shortest_path(g,'GO:0003824','GO:0004449')
gp = nx.subgraph(g,p)
print(gp.nodes())
# ['GO:0004449', 'GO:0055114', 'GO:0003824']
print(gp.edges(data=True))
# [('GO:0004449', 'GO:0055114', {'same': 'name'}),
('GO:0055114', 'GO:0003824', {'same': 'name'})]
```

Exemplo (Gene Ontology)

- Desenhar o grafo (vamos ver mais em detalhe para a frente)

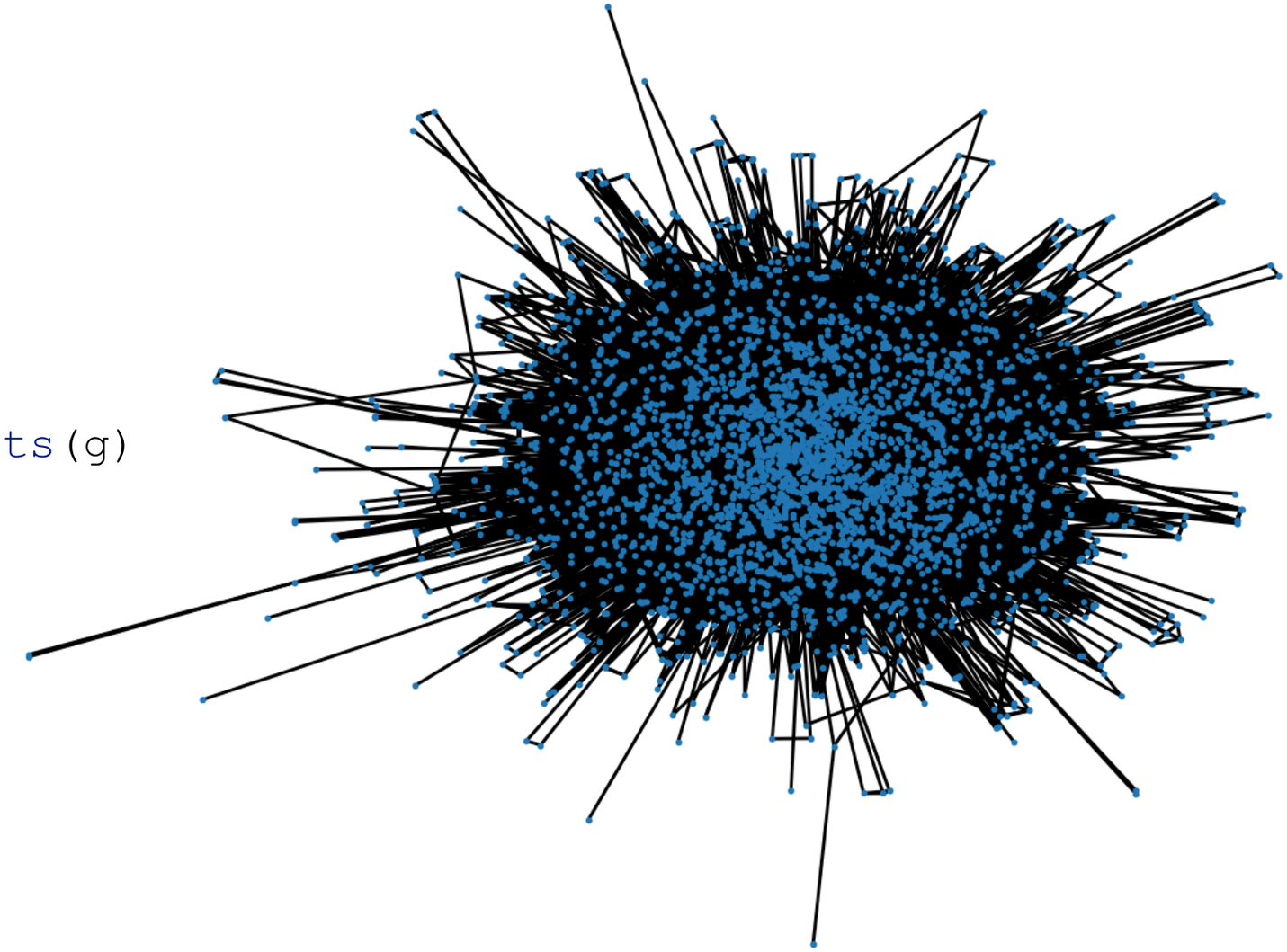


Exemplo (Gene Ontology)

- Selecionar o maior componente conexo

```
import networkx as nx

c = nx.connected_components(g)
maxc = max(c,key=len)
gc = g.subgraph(maxc)
```

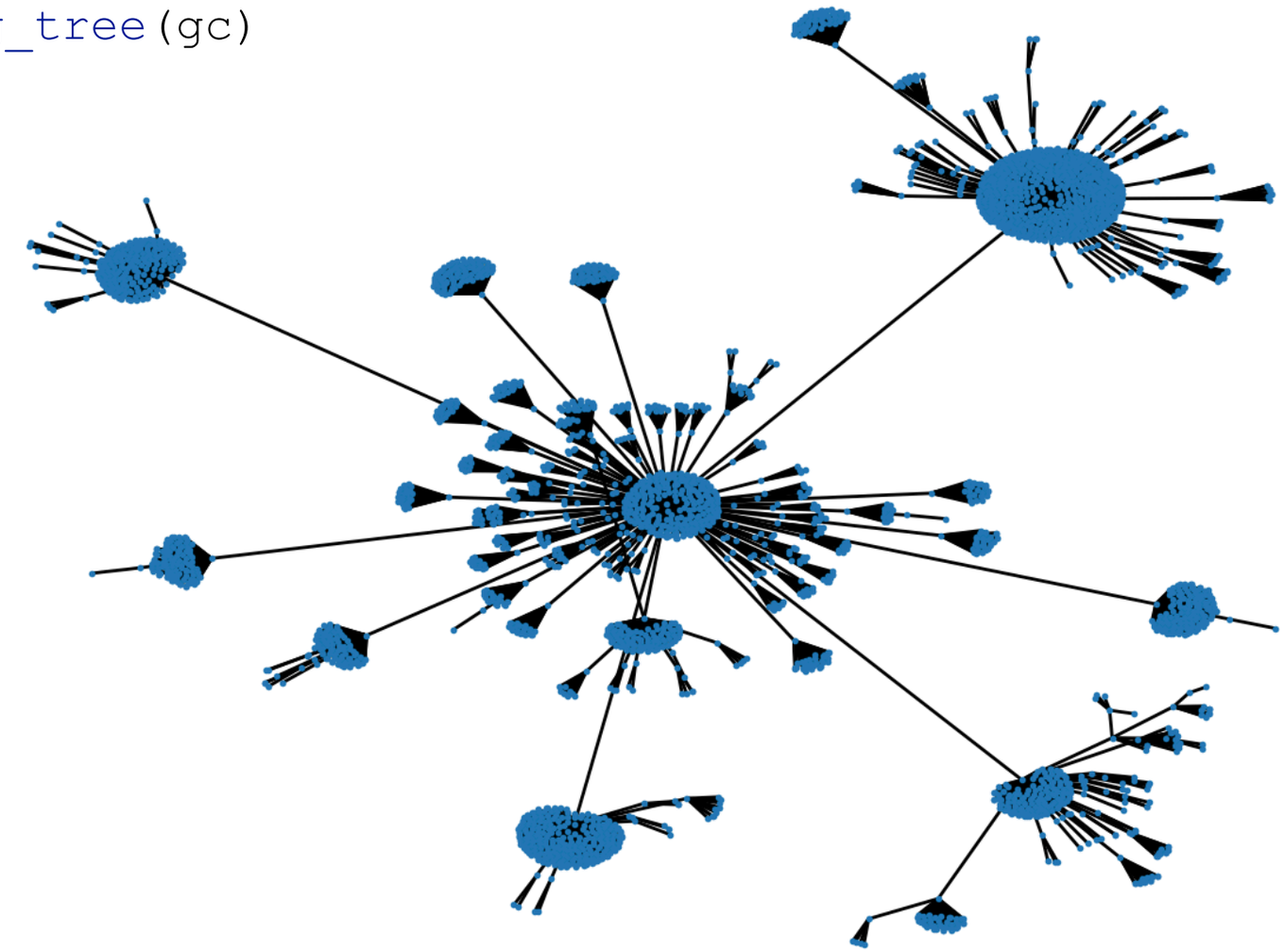


Exemplo (Gene Ontology)

- Simplificar calculando *minimum spanning tree*

```
import networkx as nx

mingc = nx.minimum_spanning_tree(gc)
```



Exemplo (Gene Ontology)

- Determinar influência de cada nodo?
- Há vários algoritmos, por exemplo *PageRank* do Google

```
import networkx as nx

ranks = nx.pagerank(g)
sranks = sorted(ranks.items(),key=lambda x: x[1])
top5 = sranks[-5:]
for r in top5:
    print(r)
# ('GO:0005829', 0.006169049326442934 5)
# ('GO:0005634', 0.007587975063755939)
# ('GO:0005737', 0.007665482985638903)
# ('GO:0005524', 0.008231140227958476)
# ('GO:0005515', 0.009864412246806 68)
```