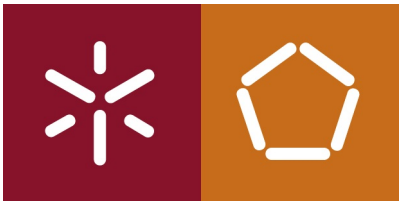

Universidade do Minho

Departamento de Informática

Scripting no Processamento de Linguagem Natural

NetworkX

Francisco Oliveira (a78416)
Raul Vilas Boas (a79617)

22 de Abril de 2019

Conteúdo

1 Introdução

O *NetworkX* é um ferramenta para a criação, manipulação e estudo de estruturas de dados.

Os casos de usos onde este tipo de ferramenta pode ser aplicado é muito vasto, portanto, é nosso objetivo num capítulo inicial explorar todas estas aplicações, bem como dar a conhecer a ferramenta em si e identificar o seu potencial.

Numa segunda parte do relatório, tendo-se descrito a ferramenta, dar-se-á início a uma vertente mais prática, onde exemplificaremos duas possível utilizações da ferramenta, uma mais simples e outra com base no processamento de linguagem natural.

2 Descrição da Ferramenta

O *NetworkX* é um pacote de *Python* desenvolvido com o objetivo de criar, manipular e estudar redes complexas. O público alvo deste ferramenta centra-se mais em utilizadores das áreas das ciências, como a matemática, física a biologia mas também por engenheiros informáticos. [1]

Este pacote providencia várias ferramentas para o estudo de estruturas dinâmicas tanto de conteúdo social, biológico e de redes de infraestruturas. Para além disto, contém uma interface e uma implementação de grafos capaz de ser utilizada por várias aplicações. Isto permite assim ter a habilidade de com um menor esforço ser possível trabalhar com uma grande quantidade de *datasets* não normalizados.

Num formato mais conciso, o *NetworkX* tem a capacidade para:

- Carregar e guardar redes em diferentes formatos;
- Gerar muitos tipos aleatórios e clássicos de redes:
- Analisar a estrutura da rede;
- Construir modelos da rede;
- Conceção de novos algoritmos para as redes;
- Desenhar redes

3 Funcionalidades

O *NetworkX* providencia de funcionalidades básicas para visualizar grafos, mas o seu principal objetivo é a análise dos grafos. Por esse motivo, é necessário utilizar outras ferramentas que se dediquem a esta tarefa, das quais, a que foi usada foi o *Matplotlib*. Para isso temos de converter o grafo gerado no *NetworlX* para um formato que seja adequado para ser lido pela ferramenta de desenho.

3.1 Instalação

Depois de instalar a biblioteca usando, por exemplo, o comando `pip install networkx` e para além disso, para ser possível desenhar os grafos também é preciso o *matplotlib*, que se pode instalar com o comando `pip install matplotlib`.

Depois é apenas necessário as importar e estão prontas a serem usadas. De forma a simplificar o código o pacote do *networkx* é abreviado para nx.

```
import networkx as nx
import matplotlib.pyplot as plt
```

3.2 Criar um grafo

Um grafo é uma coleção de nodos, ou vértices, que se ligam a outros nodos a partir de arestas. No *NetworkX* os nodos podem ser qualquer tipo de objeto, como por exemplo, uma *String*, uma imagem, um objeto XML, entre outros. As classes do *python* providenciam com a possibilidade de criar 4 tipos de grafos:

- *Graph* Esta classe implementa um grafo sem direção que ignora multiplas arestas entre dois nodos. Permite um *loop* entre si.
- *DiGraph* Grafo com arestas direcionadas.
- *MultiGraph* Grafo que permite várias arestas não direcionadas entre pares de nodos.
- *MultiDiGraph* Grafo direcionado que permite várias arestas entre pares de nodos.

Sendo assim, para criar um grafo vazio usamos a função `Graph()` disponibilizada pela biblioteca. Para criar os outros tipos de grafos, é necessários usar a sua função correspondente. Durante a criação do grafo pode se lhe atribuir atributos passando esta informação como argumento na função, estes atributos podem ser modificados posteriormente.

```
G=nx.Graph()
Gx = nx.Graph(dia="Segunda") # o atributo dia tem o valor de 'Segunda'
Gx.graph['dia'] = "Quarta" # o valor do atributo dia é alterado para 'Quarta'
G_DI=nx.DiGraph()
G_Multi=nx.MultiGraph()
G_MultiDi=nx.MultiDiGraph()
```

3.3 Adicionar componentes

Após o grafo ter sido criado, é necessário especificar que tipo de nodos e arestas vamos usar. Os nodos podem conter uma *string* ou um inteiro, ou pode conter também um identificador único para o representar. As arestas podem conter informação associada a elas, como por exemplo o peso se for o caso de um grafo pesado, isto é, cada aresta tem um inteiro que representa o valor do seu peso.

Para adicionar os nodos e as arestas no grafo vamos usar as funções `add_node()` e `add_edge()`, respetivamente. Dependendo dos parâmetros fornecidos os nodos podem ser adicionados apenas um por vez ou vários usando uma lista. O mesmo se aplica para a criação das arestas. Assim como é possível adicionar atributos ao grafo também se pode adicionar aos vértices e ás arestas passando esta informação como argumento nas seguintes funções. No caso dos vértices existe um atributo especial *weight* que é um valor necessários para usar determinados algoritmos.

- `add_node()` - adiciona apenas um nodo:

  ```
  G.add_node(1)
  ```

- `add_nodes_from()` - adiciona uma lista de nodos:

```
G.add_nodes_from([2, 3])
# outra alternativa
G.add_nodes_from(range(1,10))
```

- `path_graph()` - adiciona qualquer conjunto de nodos e pode se adicionar atributos ao nodo se necessário:

```
H = nx.path_graph(5)
```

- `add_edge()` - adiciona uma aresta entre nodos:

```
G.add_edge(1, 2)
```

- `add_edges_from` - adiciona uma lista de arestas:

```
G.add_edges_from([(1, 2), (1, 3)])
```

Assim, com base nestas funções apresentadas, criou.se um pequeno exemplo em que adicionamos vértices e arestas a um grafo obtendo o resultado observado na imagem 1.

```
G = nx.Graph()

# adicionar vértices
G.add_node('exemplo')
G.add_nodes_from(range(1,4))

# adicionar arestas
G.add_edge(1,'exemplo')
G.add_edges_from([(1, 2),(1,3)])

nx.draw_networkx(G,node_color = 'b',node_size=1600,with_labels = True)
limits=plt.axis('off')
```

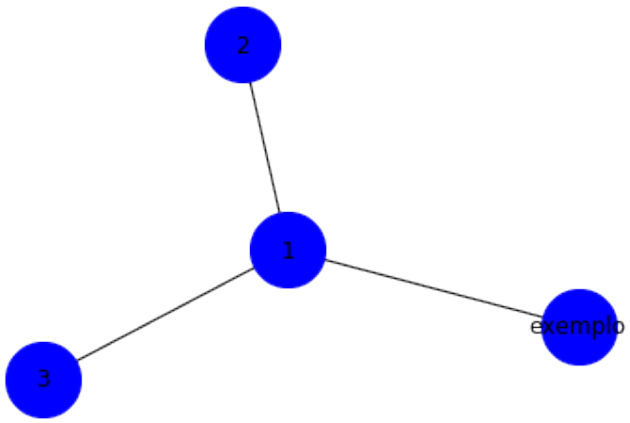

Fig. 1: Exemplo da criação de vértices e arestas.

Após os serem criados os nodos e as arestas existe a possibilidade de os conseguir examinar com base em 4 propriedades dos grafos. Estas propriedades são `G.nodes`,

`G.edges`,`G.adj` e `G.degree`. Estas propriedades permitem observar o conteúdo dos nodos, das arestas, os vizinhos de um determinado nodo e descobrir o número de arestas que incidem num determinado vértice. No entanto, estas funções apenas fornecem um visão do grafo em formato de texto.

Assim como é possível adicionar vértices e arestas, também existe a possibilidade de os remover utilizando as funções `remove_node()`, `remove_nodes_from()`, `remove_edge()` e `remove_edges_from` que são muito parecidas ás funções de adicionar.

Como base no exemplo da figura 1 ao utilizar algumas das funções retratadas obtemos os seguintes outputs.

```
G.nodes()
>>> NodeView(('exemplo', 1, 2, 3))
```

```
G.edges()
>>> EdgeView([('exemplo', 1), (1, 2), (1, 3)])
```

```
G.adj[2]
>>> AtlasView({1: {}})
```

```
G.degree[1]
>>> 3
```

3.4 Grafos direcionados

No caso dos grafos direcionados a sua classe *DiGraph* possui mais propriedades adicionais para ser possível especificar as arestas, como por exemplo, o `out_edges()` e `in_edges()` que calculam o número de arestas que tem como origem o vértice e o número de arestas que tem esse vértice como destino. Para além disto também é possível descobrir os seus sucessores e antecessores (`successors()` e `predecessors()`). Caso seja necessário tratar um grafo direcionado como um não direcionado pode se o transformar usando a função `to_indirected()` dando o grafo objetivo como argumento.

Ao aplicar as funções descritas no grafo da imagem 2 obtemos os seguintes resultados.

```
list(G.successors(2))
>>> [3]
```

```
list(G.predecessors(2))
>>> [1]
```

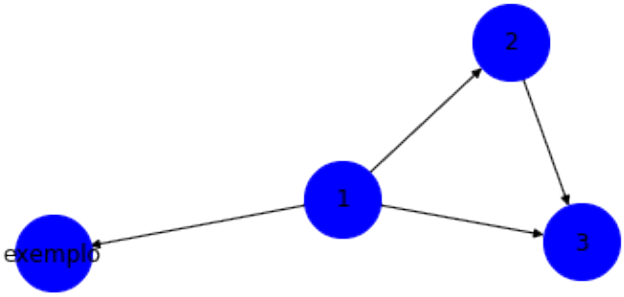

Fig. 2: Grafo direcionado do exemplo 1.

3.5 Multigrafos

Para além destes, o *NetworkX* também providencia da possibilidade de criar multigrafos como foi explicado anteriormente. Esta classe permite a adição de vários vértices entres os mesmo nodos possibilitando diferentes tipos de informação em cada uma das arestas. Isto pode ser bastante útil em algumas situações, no entanto alguns algoritmos não estão preparados para estes tipos de grafos.

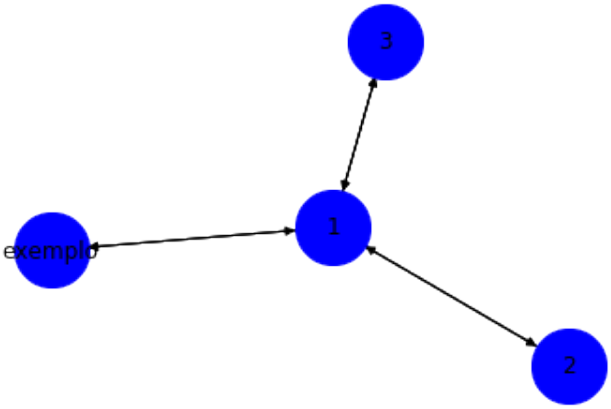

Fig. 3: Exemplo de um grafo multi direcionado.

3.6 Geradores de grafos

Outra estratégia para gerar grafos sem se ter que adicionar um nodo e aresta de cada vez é usando os geradores de grafos fornecidos na API do *NetworkX*. Primeiro começamos por mostrar as operações básicas de grafos que podem ser aplicadas:

- `subgraph(G,nbunch)` - cria um subgrafo a partir de um grafo com base no nodos fornecidos;
- `union(G1,G2)` - une dois grafos;
- `disjoint_union(G1,G2)` - une dois grafos assumindo que todos os nodos são diferentes;

- `cartesian_product(G1,G2)` - retorna o produto cartesionado dos grafos;
- `compose(G1,G2)` - junta os grafos identificando os nodos em comum;
- `complement(G)` - inverte o grafo;
- `create_empty_copy(G)` - retorna uma cópia vazia da mesma classe do grafo;
- `to_undirected(G)` - retorna a representação não direcionada do grafo;
- `to_directed(G)` - retorna a representação direcionada do grafo.

Outra alternativa é usar um dos grafos que existem, como por exemplo o grafo de *Petersen*, o grafo de *Tutte* e também se pode usar um gerador de grafos estocásticos ou ler um grafo que tenha sido guardado num ficheiro. Os formatos usados são *GML*, *GraphML*, *pickle*, entre outros.

De seguida, mostramos alguns exemplos de grafos que podem ser gerados pelo *NetworkX*. Como por exemplo o grafo de *Peterson* que consiste num grado não orientado com 10 vértices e 15 arestas usado como exemplo útil e contra-exemplo para muitos problemas em teoria de grafos.

```
G = nx.petersen_graph()
nx.draw_networkx(G,node_color = 'b',node_size=1600,with_labels = True)
limits=plt.axis('off') # turn of axis
```

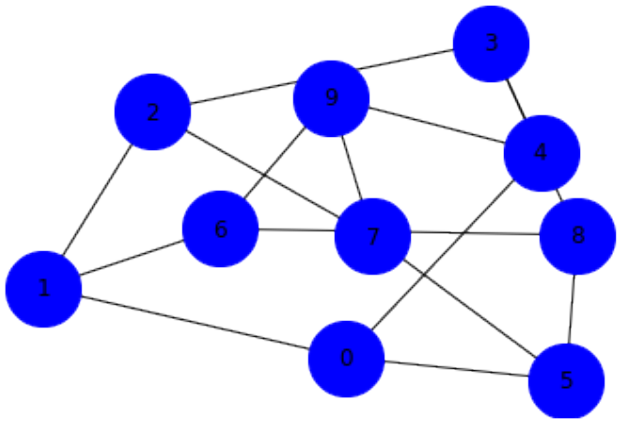

Fig. 4: Grafo de *Petersen*.

Para além disto, também é possível gerar grafos aleatórios, deste modo, utilizou-se o modelo de *Erdõs-Rényi* para demonstrar esta possibilidade, pois este +e um dos dois modelos estritamente relacionados para gerar grafos aleatórios que inclui o limite entre cada par de nós com igual probabilidade independentemente das extremidades.

O função `erdos_renyi_graph()` disponibilizada pelo *NetworkX* pode receber dois argumentos, dois quais o primeiro é o número de vértices que serão criados e o outro é a probabilidade de se criar uma aresta entre dois determinados vértices.

```
G=nx.erdos_renyi_graph(10,0.4)
nx.draw(G,node_color = 'b',node_size=1000,with_labels=True)
```

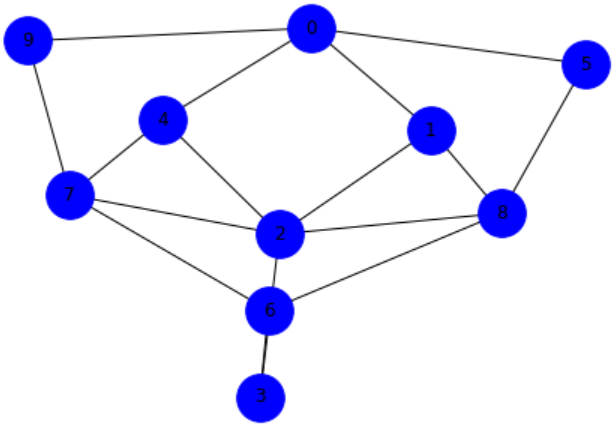

Fig. 5: Grafo de *Erdõs-Rényi*.

4 Exemplo de Utilização

Com o intuito de demonstrar um pequeno exemplo da utilização da ferramenta *NetworkX* decidiu-se criar um algoritmo para criar um grafo. O exemplo consiste na criação de um árvore binária com base num *array* de inteiros. Para isso vai se percorrendo o *array* e criando arestas entre o vértice atual e o seus respetivos pais que estão nas posições obtidas a partir da multiplicação do índice atual e a soma de um ou dois dependendo de qual o pai que se pretende obter.

Para desenhar o grafo utilizou-se umas das funções disponibilizadas pelo *NetworkX* denominada `draw_kamada_kawai()` que desenha dependendo da força direcional do grafo. O resultado obtido é apresentado na figura 7.

```
def create_edges(lista):
    tam = len(lista)-1
    for i in range(tam):
        if 2*i+1 <= tam:
            G.add_edge(lista[i],lista[2*i+1])
        if 2*i+2 <= tam:
            G.add_edge(lista[i],lista[2*i+2])

G = nx.Graph()
lista = range(30)
G.add_nodes_from(lista)
create_tree(lista)
nx.draw_kamada_kawai(G,node_color = 'lime',node_size=500,with_labels = True)
```

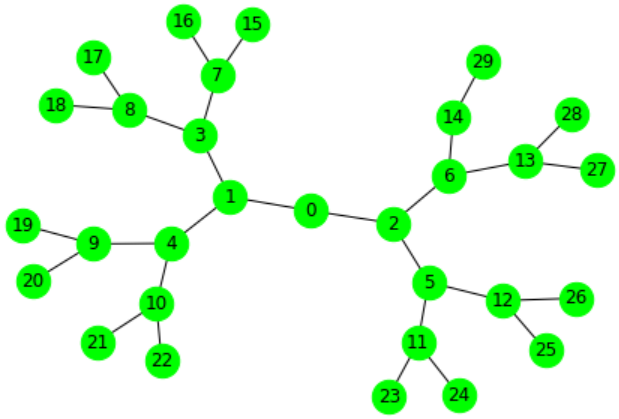

Fig. 6: Grafo exemplo.

5 Aplicação no contexto NLP

No intuito de entender melhor a ferramenta que nos foi proposta criamos um forma de a usar num contexto de processamento de linguagem natural. Nesse sentido, utilizou-se o livro *Os Maias* para criar um grafo, no qual cada vértice é um nome próprio encontrado no livro e as arestas são as ligações dos nomes próprios encontradas nas mesma frase. Para além disso, as arestas vão conter um peso que representa o número de vezes que essa ligação ocorreu entre os vértices. Isto permite-nos analisar num determinado excerto de texto o número de ocorrências em que duas personagens aparecem na mesma frase, isto pode ajudar a determinar, por exemplo, quais as personagens que possuem um maior contacto ou importância no livro.

De modo a ser possível identificar os nomes próprios nas frases utilizou-se um *corpus* CETEM do jornal Público que continha palavras de notícias anotadas com a respetiva classe de palavra entre outros atributos. No nosso caso, o atributo que nos interessava era a classe da palavra. Deste modo, a partir deste ficheiro criou-se um ficheiro *tagged* que contém a palavra e o seu tipo respetivo. Para agilizar o processo e para não ser preciso estar sempre a fazer este processamento, pois é uma tarefa demorada, realizamos isto apenas um vez com uma espécie de *build*. Para correr o *build* deve-se usar o argumento `-b`. Depois o ficheiro já se encontra pronto para ser usado as vezes que forem necessárias.

De seguida, fornecemos um nome de ficheiro `txt` como argumento, com por exemplo o conteúdo de *Os Maias* e o nosso algoritmo encarrega-se de gerar o grafo com os nomes próprios e as ocorrências entre eles. Para isso, percorre-se cada linha do ficheiro e caso seja encontrado nessa linha dois ou mais nomes próprios, é criado um vértice com os respetivos nomes e uma ligação entre cada um eles.

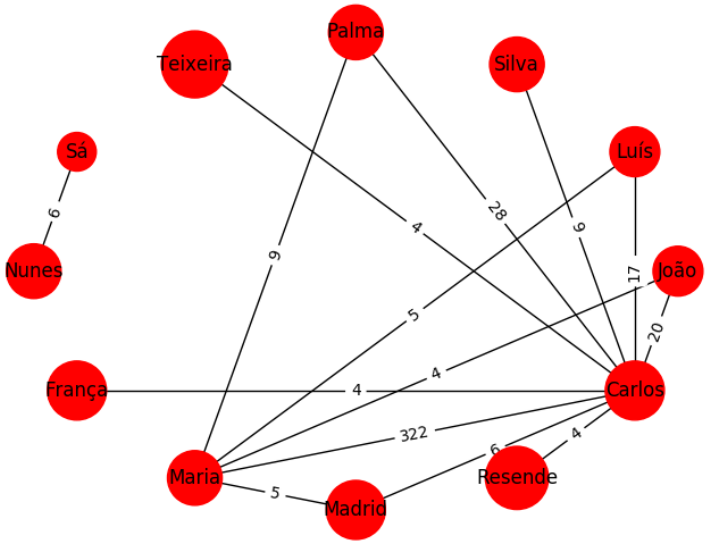

Fig. 7: Grafo exemplo dos Maias.

6 Conclusão

Neste trabalho prático, propusemos-nos a analisar com detalhe a ferramenta *NetworkX* que consiste maioritariamente numa ferramenta dedicada à análise de grafos. Foi analisada a API disponível e as capacidades relevantes para o SPLN oferecidas por esta.

Com o trabalho desenvolvido, consideramos ter atingido os objetivos propostos sendo esses o estudo da ferramenta e a criação de um exemplo em contexto de processamento de linguagem natural. Como trabalho futuro poderia-se tentar melhor a abordagem utilizada pois pode não apanhar todos os casos contidos nos ficheiros ou apanhar casos que não estejam corretos com base no contexto fornecido.

Referências

[1] https://networkx.github.io/ Acedido em 10 de Abril de 2019.

[2] https://networkx.github.io/documentation/stable/ Acedido em 10 de Abril de 2019.