LEONARDO STEFAN

COMPARAÇÃO ENTRE NEO4J E NETWORKX

(*versão pré-defesa, compilada em 18 de setembro de 2022*)

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Renato Carmo.

CURITIBA PR

2022

RESUMO

Neste trabalho foram feito experimentos com as ferramentas de grafos Neo4j e NetworkX. São executados algoritmos em cenários onde o uso de memória é o limitante na execução, são comparados e analisados o desempenho e eficácia de cada ferramenta. Por fim, sob a perspectiva do autor é comparado as diferenças na sintaxe de implementação, documentação e configurações das ferramentas.

Palavras-chave: Neo4j, NetworkX, Grafos, Comparação, Ferramentas, Rede, Banco-de-Dados

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS-FONTE

LISTA DE ACRÔNIMOS

DINF	Departamento de Informática
UFPR	Universidade Federal do Paraná
BD(s)	Banco(s) de dados
SGDB	Sistema Gerenciador de Banco de Dados
GDS	Graph Data Science (plugin Neoj4)
APOC	Awesome Procedures On Cypher
ACID	Atomicidade, Consistência, Isolação e Durabilidade (propriedades conceituais desejadas de um banco de dados)
IDE	Integrated Development Environment

SUMÁRIO

1 INTRODUÇÃO

A teoria dos grafos busca descrever diversos problemas matemáticos do cotidiano através de conjuntos de nodos (vértices) e relações (arestas). Grafos têm sido aplicados para modelar quatro principais domínios da ciência: estruturação de informação, redes tecnológicas, eventos biológicos e ciências sociais. Exemplo disso são as redes sociais que reúnem informações de bilhões de usuários e como se relacionam, o que evidencia sua importância. Algoritmos que processam grafos podem ter alta complexidade computacional tanto em tempo quanto em espaço, fazendo com que o uso dos recursos computacionais (processamento e memória) cresça acima da proporção do crescimento do grafo a ser processado.

Atualmente, existem diversas ferramentas para executar algoritmos e lidar com estruturas de dados em grafos. Algumas são *frameworks* que processam o grafo mantendo tudo em memória como é o caso do NetworkX[1], Graphviz[2] e Igraph[3]. Outras são Sistemas Gerenciadores de Banco de Dados (SGBD) orientados a grafos, focados no problema de se armazenar dados em nodos e relações para ficarem acessíveis para consulta através de algoritmos de grafos, como é o caso do Neo4j[4] ArangoDB[5] e o Amazon Neptune[6].

Os capítulos seguintes estão organizadas da seguinte forma: o Capítulo 2 expõe o método definido e utilizado para a execução do trabalho, assim como decisões de projeto; o Capítulo 3 exibe e analisa os resultados dos experimentos e o processo de implementação do projeto; por fim, o Capítulo 4 reúne as conclusões a partir dos resultados obtidos, aponta limitações do trabalho e possíveis trabalhos futuros.

1.1 OBJETIVO

Com a necessidade atual de se processar grafos cada vez maiores, chegando a possuírem bilhões de nodos e relações para serem processados, o uso da memória vem sendo o limitante no processamento destes grafos. Pensando nestes cenários, é proposta a hipótese de que a ferramenta Neo4j seja capaz de processar grafos maiores que a ferramenta NetworkX, uma vez que o Neo4j seria capaz de aproveitar a memória secundária para reduzir o uso da memória primária em tempo de execução. São utilizadas as configurações padrão de cada ferramenta, evitando o uso de recursos avançados que melhorariam seus desempenhos. É definido os seguintes objetivos.

1. Comparar experimentalmente as ferramentas Neo4j e NetworkX;

[1] networkx.org/
[2] graphviz.org/
[3] igraph.org/
[4] neo4j.com/
[5] arangodb.com/
[6] aws.amazon.com/pt/neptune/

2. Documentar as principais diferenças durante a implementação de cada ferramenta, avaliando qualitativamente o processo utilizado, pela perspectiva do autor, desde a configuração até a utilização das ferramentas.

É importante notar que as ferramentas são capazes de atribuir informações descritivas aos nodos e relações como cores, rótulos, nomes, pesos, etc. Não faz parte do objetivo deste trabalho observar o uso destes atributos.

1.2 SOBRE AS FERRAMENTAS

1.2.1 Neo4j

Neo4j(Inc, 2007) é um SGBD orientado a grafos implementado em Java. Ser orientado a grafos significa que tanto sua linguagem de consulta quando estruturação dos dados são elaborados em grafos, diferente dos bancos de dados relacionais comumente utilizados, assim o Neo4j pode ter um desempenho melhor que banco de dados relacionais como demonstrado em Batra e Tyagi (2012). As transações do Neo4j cumprem com as propriedades de Atomicidade, Consistência, Isolação e Durabilidade (ACID) garantindo a robustez dos dados. Sua linguagem de consulta Cypher foi pensada para ser simples e intuitiva para consultas de nodos e relações. Além de ser possível realizar consultas diretamente em Cypher, o Neo4j conta também com plugins com algoritmos prontos e otimizados para realizar consultas mais complexas em um grafo, uma delas é o Graph Data Science (GDS) que dispõe algoritmos para computar métricas, buscas de caminhos e similaridade, outro é o Awesome Procedures On Cypher (APOC) que possui diversas funções de tratamento de dados e gerenciamento de consultas otimizando o uso do Neo4j.

1.2.2 NetworkX

(NetworkX, 2005). Em uma tradução de Hagberg et al. (2008): “NetworkX é um pacote de linguagem Python para exploração e análise de redes e algoritmos de redes. O pacote principal fornece estruturas de dados para representar diversas redes ou grafos, incluindo grafos simples, grafos direcionados e grafos com arestas paralelas com laços [...] a flexibilidade torna o NetworkX ideal para representar redes encontradas em diversos campos científicos. Além das estruturas de dados básicas, muitos algoritmos de grafos são implementados para calcular propriedades de rede e medidas de estrutura: caminhos mais curtos, centralidade de intermediação, agrupamento e distribuição de graus e muito mais [...] A facilidade de uso e flexibilidade da linguagem de programação Python, juntamente com a conexão com as ferramentas SciPy, tornam o NetworkX uma ferramenta poderosa para cálculos científicos.”

1.2.3 Trabalhos Relacionados

O desempenho das ferramentas NetworkX, Gephi, IGraph, e Pajek foi comparado anteriormente em Akhtar (2014). Na comparação são explicitados os recursos disponíveis em cada ferramenta, como algoritmos implementados, formatos de arquivos capazes de processar e tipos de grafos capazes de processar. O artigo também apresenta a comparação da complexidade dos algoritmos implementados em cada ferramenta e uma comparação experimental do desempenho das ferramentas com um único *dataset*.

Não foi possível encontrar nenhuma comparação direta do Neo4j com outras ferramentas de grafos, porém em Huang e Dong (2013) é analisado a arquitetura e paradigma do Neo4j, e é realizado testes mensurando o desempenho da ferramenta e seus componentes.

2 MÉTODO

O foco da comparação é realizar experimentos onde as ferramentas são utilizadas para executar os algoritmos em condições análogas, verificando o maior grafo que cada uma é capaz de processar e o tempo de execução em cada ferramenta. Para isso é definido: o ambiente de testes; os recursos e ferramentas usados para executar os roteiros de testes e registrar os resultados; os *datasets* com os grafos a serem processados; e os algoritmos testados e avaliados.

2.1 AMBIENTE DE TESTES

Para realizar uma comparação experimental das ferramentas que evitasse interferências do sistema operacional ou quaisquer outras ferramentas no ambiente de testes foi definido a utilização da virtualização do ambiente através da plataforma Docker, o que também facilita a reprodutibilidade dos testes devido sua estrutura de contêiner configurado por arquivo. O computador utilizado para os testes possui as seguinte configuração.

- **Sistema Operacional** : Linux Mint 20.2;
- **Processador** : Intel i3 12100F (4 núcleos, 8 threads, 3.3GHZ, cache 12MB);
- **Memória primária** : 32MB DDR4, 3000 MHz;
- **Memória secundária**: SSD 120GB sata.

O contêiner Docker processou sob a seguinte configuração de virtualização.

- 4 Processos;
- 8GB memória primária;
- 16GB de memória secundária;
- 516MB de swap.

Para os testes com o NetworkX é utilizada a imagem Docker oficial do Python (tag: `Python:3.10-slim`), instalando a biblioteca do NetworkX.

Para os testes com o Neo4j é utilizada a imagem Docker oficial do Neo4j (tag: `neo4j:4.3`), instalando o Python versão 3.10 para a automação dos scripts de teste.

2.2 *DATASETS* UTILIZADOS

A fim de verificar o comportamento de cada ferramenta processando grafos de tamanhos distintos, foram selecionados *datasets* (conjunto de dados) variados tamanhos conforme mostrado na Tabela 2.1. Todos os *datasets* foram obtidos pelo livro *Complex Networks*(Vito Latora, 2017) que os utiliza em seus exemplos.

Datasets	**Nome nos testes**	**Nodos**	**Relações**
C.elegans neural network	`c_elegans_undir`	279	2287
Internet AS network, 1997	`AS-19971108`	3015	5156
Internet AS network, 1998	`AS-19981002`	4180	7768
Internet AS network, 1999	`AS-19991002`	5861	11313
NCSTRL coauthorship	`ncstrl`	11994	20395
Arxiv physics coauthorship	`astro_ph`	16706	121251
Astrophysics coauthorship	`arxiv_phy`	49983	245300
World Wide Web NotreDame	`www_notre_dame`	325729	1469680
Medline coauthorship	`medline`	1449111	11803060

Tabela 2.1: Datasets utilizados

2.3 ALGORITMOS TESTADOS

Foram selecionados oito algoritmos para a realização dos testes, sendo: dois deles de baixa complexidade computacional; quatro intermediários, com complexidade computacional maior, porém o uso de memória não cresce tanto quanto o uso do processador; e por fim dois algoritmos com alta complexidade computacional e que utilizam mais o recurso de memória.

- Menos complexos
    1. Contagem de nodos
    2. Contagem de relações
- Intermediários
    3. Cálculo do grau de cada nodo
    4. Contagem de componentes
    5. Contagem de triângulos
    6. Cálculo do diâmetro
- Mais complexos
    7. Cálculo de caminho mínimo entre dois nodos
    8. Cálculo de todos os pares de caminhos mínimos

Para o cálculo de diâmetro, neste trabalho é considerado que se o grafo não é conexo (possui mais de um componente), diâmetro é o maior diâmetro entre os componentes do grafo.

2.4 RECURSOS E FERRAMENTAS PARA A EXECUÇÃO DOS TESTES

Como o NetworkX é uma biblioteca para o Python, e para o Neo4j possui uma interface com Python (além de outras plataformas) para executar as consultas, foi decidido utilizar o Python para executar o roteiro de testes em uma mesma base de dados, e utilizado um *shell script* que executa cada cenário de teste percorrendo os *datasets* testados. Foi desenvolvido o módulo *TimeMarker* em Python para registrar o tempo de execução de cada algoritmo.

O script Python que realiza o teste começa executando os algoritmos dos mais leves até os mais complexos, registrando o tempo de execução e salvando os resultados a cada algoritmo testado. Desta forma se uma ferramenta parar de funcionar ao executar um teste para um determinado *dataset*, os seguintes algoritmos que seriam testados para o mesmo *dataset* são abortados, pois admite-se que parariam de funcionar também.

3 RESULTADOS

Nesta seção são mostrados e analisados os resultados dos experimentos. A legenda dos gráficos apresentados nesta seção seguem todos o mesmo padrão: as linhas são uma interpolação do tempo medido em cada *dataset*, cada linha representa uma ferramenta (Neo4j ou NetworkX). Os pontos são os resultados obtidos de um *dataset* usando a ferramenta da linha contida. É importante notar que a escala não é a mesma em todos os gráficos, havendo alguns gráficos em escala logarítmica para o eixo das ordenadas, e outros em escala logarítmica para o eixo das abcissas. Cada gráfico está posicionado logo após o parágrafo que discute suas informações.

3.1 RESULTADO DOS EXPERIMENTOS

Começando pelos algoritmos de baixa complexidade, tanto para a contagem de nodos quanto para a contagem de relações, o desempenho de ambas as ferramentas executam independente do tamanho do grafo, o que sugere que ambas aproveitam processamento da leitura dos grafos para armazenar tais informações. Para esses algoritmos, os tempos de execução do NetworkX foram menores que os do Neo4j. Tal diferença possivelmente se deve ao acesso à memória secundária no Neo4j demorar mais que o simples acesso à memória primária do NetworkX. Outra possível causa é o *overhead* no tempo da requisição que o Python precisa para manter a conexão com Neo4j. Este comportamento observado nos demais experimentos.

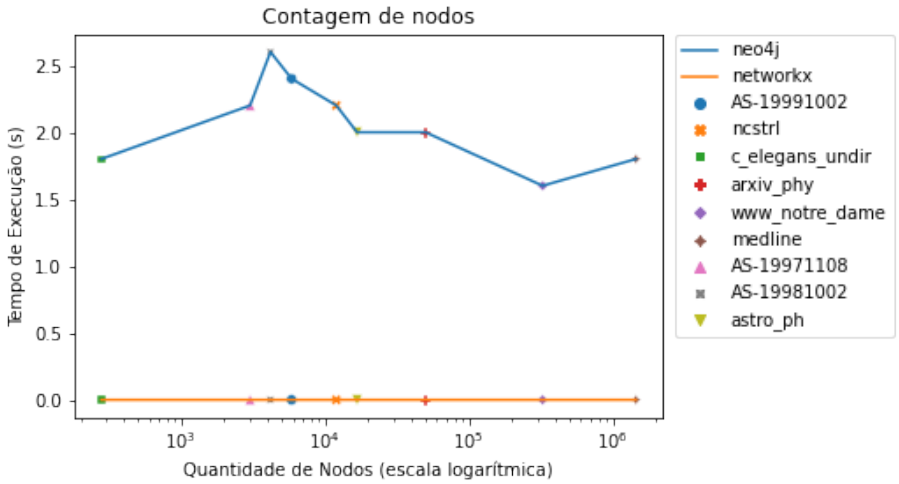


Figura 3.1: Tempo da contagem de nodos (abscissas em escala logarítmica)

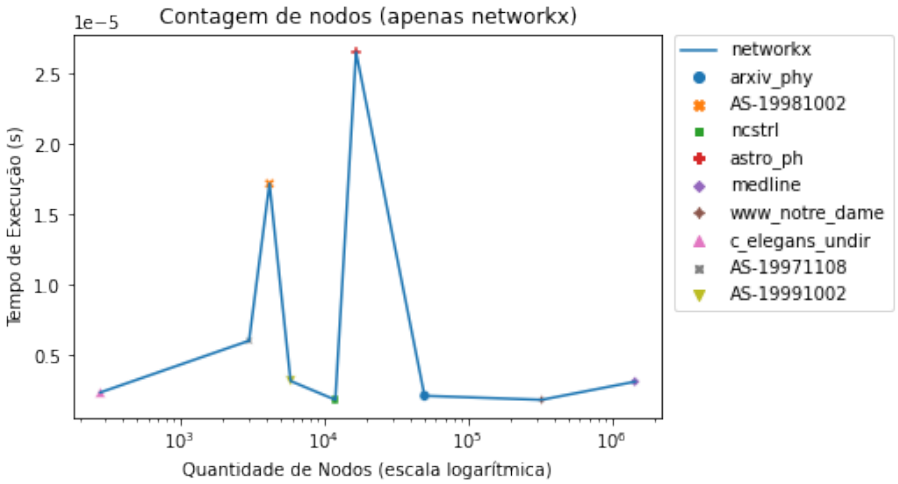


Figura 3.2: Tempo da contagem de nodos, apenas NetworkX (abscissas em escala logarítmica)

Dataset	Nodos	Tempo Neo4j	Tempo NetworkX
$c_elegans_undir$	279	1.2e+01	5.7e-05
AS-19971108	3015	1.1e+01	4.2e-04
AS-19981002	4180	1.1e+01	8.6e-04
AS-19991002	5861	1.2e+01	8.8e-04
ncstrl	11994	1.3e+01	1.9e-03
$astro_ph$	16706	1.4e+01	4.8e-03
$arxiv_phy$	49983	9.4e+00	1.5e-02
www_notre_dame	325729	9.0e+00	6.8e-02
medline	1449111	1.1e+01	7.2e-01

Tabela 3.1: Tempo da contagem de nodos

No cálculo de graus dos nodos ambas ferramentas apresentam o mesmo padrão de crescimento no tempo de execução. Porém o Neo4j possui um de execução tempo muito superior. Novamente, a possível causa seria o tempo de leitura do Neo4j à memória secundária ou por conta do *overhead* de tempo da requisição. O mesmo comportamento observado na contagem de graus foi observado na contagem de componentes e no cálculo do diâmetro.

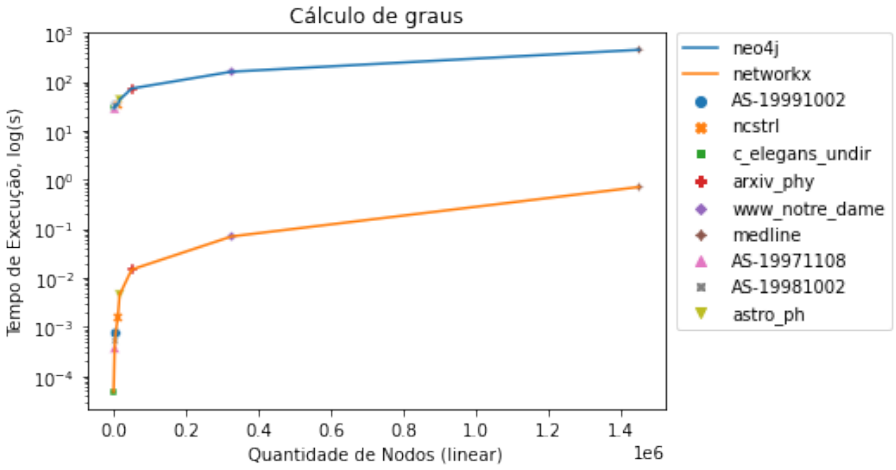


Figura 3.3: Cálculo de grau dos nodos (ordenadas em escala logarítmica)

Dataset	Nodos	Tempo Neo4j	Tempo NetworkX
$c_elegans_undir$	279	2.9e+01	4.8e-05
AS-19971108	3015	2.9e+01	3.8e-04
AS-19981002	4180	3.3e+01	5.2e-04
AS-19991002	5861	3.4e+01	7.5e-04
ncstrl	11994	3.4e+01	1.6e-03
$astro_ph$	16706	4.3e+01	4.5e-03
$arxiv_phy$	49983	7.3e+01	1.5e-02
www_notre_dame	325729	1.6e+02	7.0e-02
medline	1449111	4.5e+02	7.2e-01

Tabela 3.2: Tempo da contagem de graus

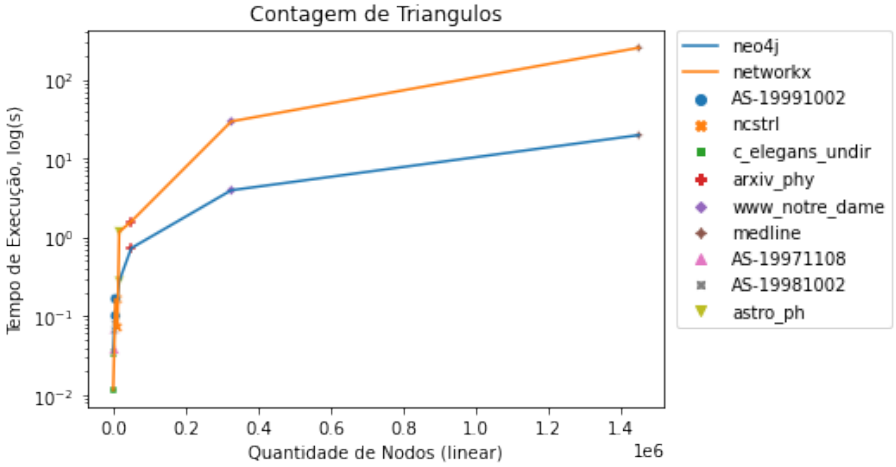


Figura 3.4: Contagem de componentes (ordenadas em escala logarítmica)

Para o cálculo do diâmetro, foram necessárias implementações distintas para realizar os te. Os em cada ferramenta. O Neo4j não possui nenhum algoritmo pronto para esta tarefa, sendo realizado o cálculo de caminhos mínimos entre todos os nodos utilizando a colheita (*yield*) a maior

Dataset	Nodos	Tempo Neo4j	Tempo NetworkX
$c_elegans_undir$	279	1.6e+01	2.9e-04
AS-19971108	3015	2.2e+01	1.8e-03
AS-19981002	4180	2.3e+01	3.7e-03
AS-19991002	5861	2.8e+01	4.7e-03
ncstrl	11994	2.3e+01	9.7e-03
$astro_ph$	16706	3.2e+01	2.7e-02
$arxiv_phy$	49983	3.6e+01	8.4e-02
www_notre_dame	325729	5.0e+01	4.9e-01
medline	1449111	2.3e+02	6.1e+00

Tabela 3.3: Tempo da contagem de componentes

distância. Já o NetworkX, possui o rotina de cálculo do diâmetro nativamente implementado, porém não é capaz de processar grafos não conexos, sendo necessário implementar a colheita do processamento para cada componente.

Código-fonte 3.1: Cálculo de diâmetro Neo4j

```
MATCH (source:AUTHOR)
CALL gds.allShortestPaths.delta.stream("myGraph",{
   sourceNode:source,
   relationshipWeightProperty: 'cost',
   delta: 3.0
})
YIELD totalCost as distance
WHERE gds.util.isFinite(distance) and distance > 1
return max(distance)
```

Código-fonte 3.2: Cálculo de diâmetro NetworkX

```
def fullDiameter(graph):
   subs = (G.subgraph(comp).copy() for comp in nx.connected_components(G))
   return max(nx.algorithms.distance_measures.diameter(sub) for sub in subs)
```

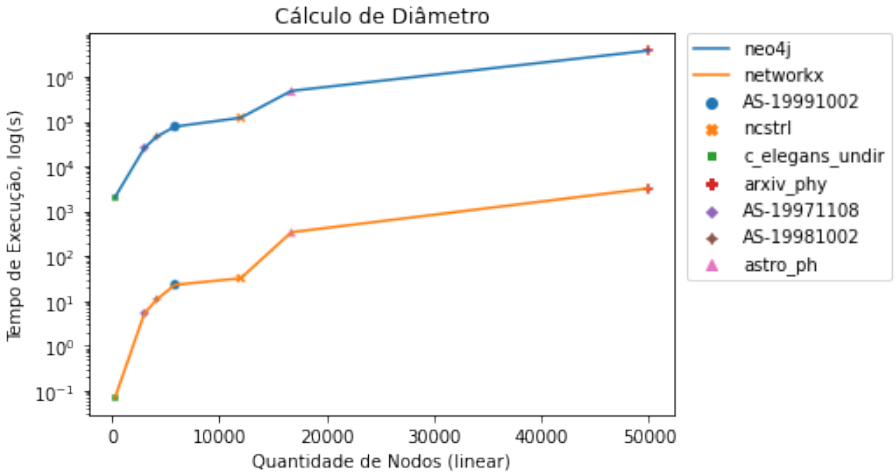


Figura 3.5: Cálculo do diâmetro do grafo (ordenadas em escala logarítmica)

Dataset	Nodos	Tempo Neo4j	Tempo NetworkX
$c_elegans_undir$	279	2.1e+03	7.0e-02
AS-19971108	3015	2.6e+04	5.3e+00
AS-19981002	4180	4.6e+04	1.1e+01
AS-19991002	5861	7.6e+04	2.3e+01
ncstrl	11994	1.2e+05	3.2e+01
$astro_ph$	16706	4.8e+05	3.4e+02
$arxiv_phy$	49983	3.8e+06	3.2e+03

Tabela 3.4: Tempo do cálculo do diâmetro

O único experimento onde o Neo4j foi mais eficiente que o NetworkX foi na contagem de triângulos. Uma hipótese para tal é a utilização de implementações distintas para cada ferramenta, e no caso do Neo4j esta implementação aproveitar melhor o fato dos grafos utilizados serem esparsos.

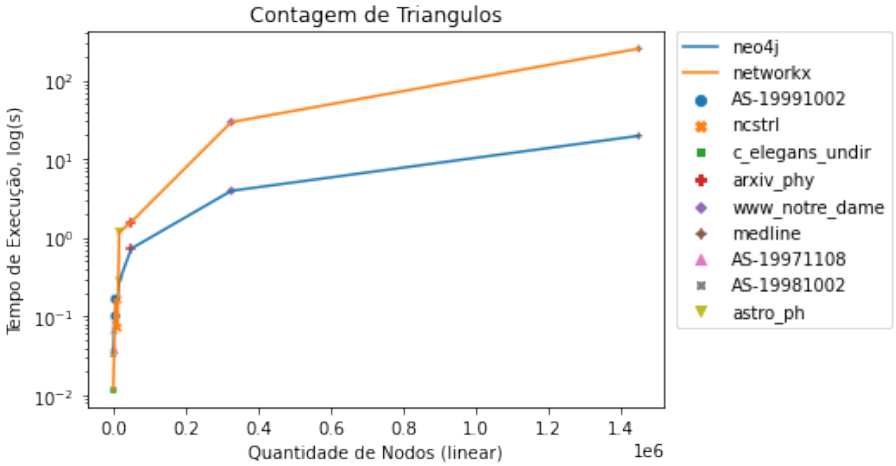
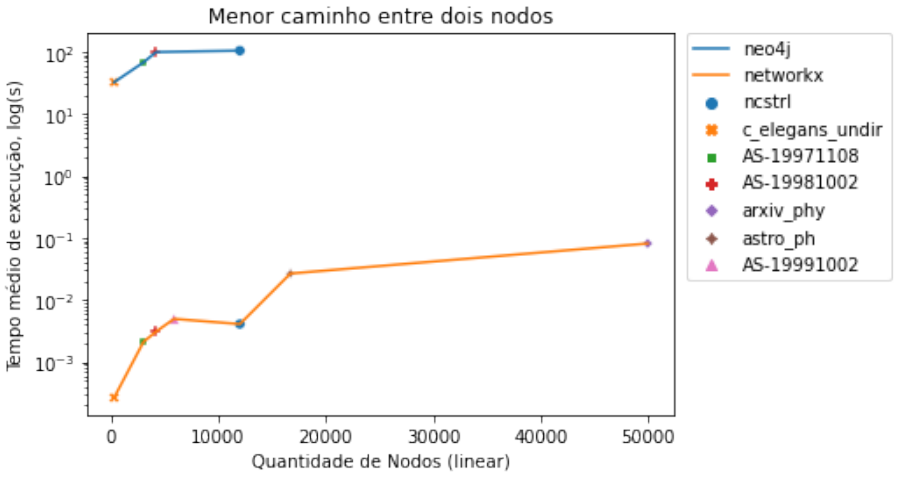


Figura 3.6: Contagem de triângulos no grafo (ordenadas em escala logarítmica)

Dataset	Nodos	Tempo Neo4j	Tempo NetworkX
$c_elegans_undir$	279	3.4e-02	1.2e-02
AS-19971108	3015	6.8e-02	3.9e-02
AS-19981002	4180	8.0e-02	6.6e-02
AS-19991002	5861	1.0e-01	1.7e-01
ncstrl	11994	1.6e-01	7.2e-02
$astro_ph$	16706	2.8e-01	1.2e+00
$arxiv_phy$	49983	7.3e-01	1.6e+00
www_notre_dame	325729	3.9e+00	2.9e+01
medline	1449111	2.0e+01	2.5e+02

Tabela 3.5: Tempo da contagem de triângulos

Olhando para os algoritmos testados que possuem maior complexidade de espaço, é onde vemos acontecer o contrário do que se propunha na hipótese inicial. O NetworkX foi capaz de processar grafos maiores que o Neo4j conseguiu. No algoritmo de caminho mínimo entre dois nodos, houve três casos onde o NetworkX conseguiu processar o grafos maiores aos quais o Neo4j falhou por falta de memória (`astro_ph`, `arxiv_phy` e `AS-1991002`). Para este teste nenhum dos dois foi capaz de processar os grafos `www_nostra_damus` e `medline` devido à falta de memória. O tempo de execução do Neo4j continua sendo muito maior do que o do NetworkX, porém mantendo o mesmo padrão de crescimento.



;

Figura 3.7: Caminho mínimo entre dois nodos

Dataset	Nodos	Tempo Neo4j	Tempo NetworkX
$c_elegans_undir$	279	3.2e+01	2.7e-04
AS-19971108	3015	6.8e+01	2.1e-03
AS-19981002	4180	9.9e+01	3.1e-03
AS-19991002	5861	-	5.0e-03
ncstrl	11994	1.0e+02	4.1e-03
$astro_ph$	16706	-	2.7e-02
$arxiv_phy$	49983	-	8.1e-02

Tabela 3.6: Tempo da busca do caminho mínimo entre dois nodos

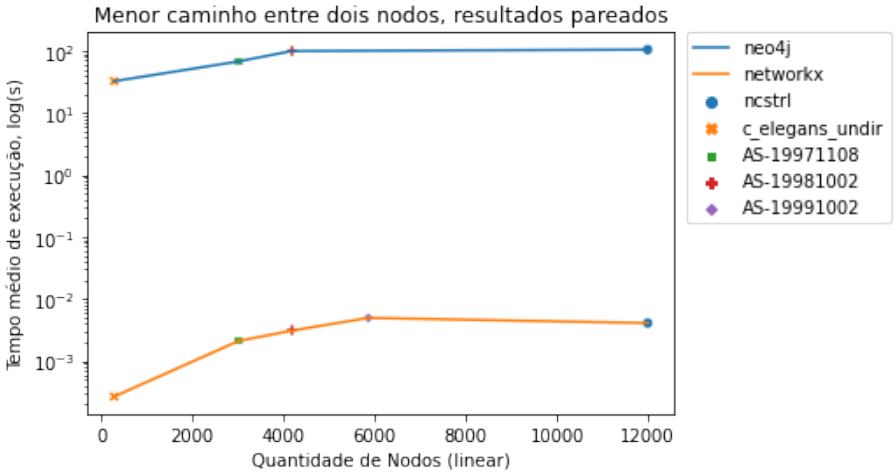


Figura 3.8: Caminho mínimo entre dois nodos, recorte inicial do tempo

No experimento dos algoritmos de todos os pares de caminhos mínimos, que demandam mais memória e processamento por precisarem armazenar os resultado do processamento. Para esse experimento, como o GDS do Neo4j possui duas implementações distintas para executar esse algoritmo, ambos foram testados, um deles utilizando o algoritmo de *Dijkstra* e outro usando o *DeltaSteping* (algoritmo por correção de rótulos, paralelizável). No caso do NetworkX apenas uma implementação estava disponível, não foi possível identificar qual implementação é a utilizada. Nestes testes o NetworkX novamente foi capaz de processar um grafo maior que o Neo4j, o `AS-1991002`. Tanto o Neo4j quanto o NetworkX não foram capazes de processar os grafos `AS-1991002`, `astro_ph`, `arxiv_phy`, `www_nostra_damus` e `medline`, por falta de memória.

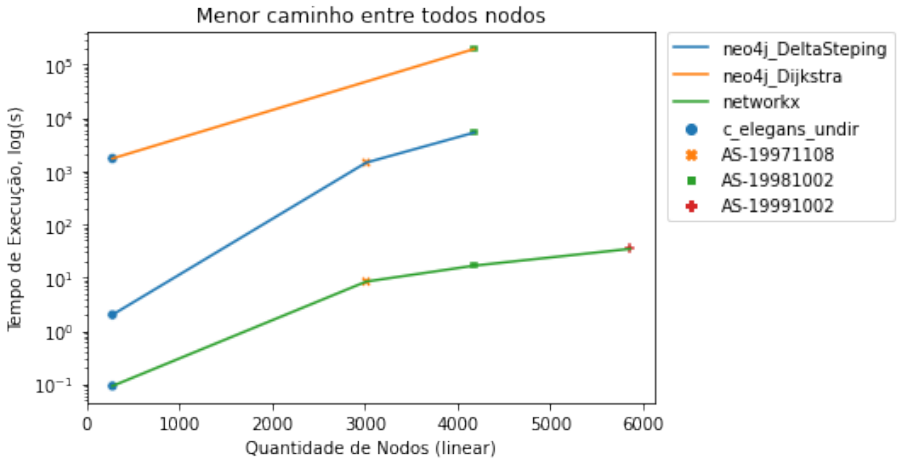


Figura 3.9: Busca do caminho mínimo entre todos nodos

Dataset	Nodos	Neo4j Dijkstra	Neo4j Delta Steping	NetworkX
$c_elegans_undir$	279	1.7e+03	2.0e+00	9.1e-02
AS-19971108	3015	-	1.4e+03	8.4e+00
AS-19981002	4180	-	5.2e+03	1.6e+01
AS-19981002	4180	1.9e+05	1.9e+05	1.6e+01
AS-19991002	5861	-	-	3.4e+01

Tabela 3.7: Tempo da busca de todos caminhos mínimos

Durante a execução dos experimentos houve alguns problemas com o Neo4j que estava salvando *logs* excessivos da execução dos algoritmos, gerando arquivos que atingiam o limite do sistema e fazendo com que o teste parasse de funcionar. Para contornar este problema foi necessário adicionar uma configuração extra para que não fossem gerados os *logs* de cada passo dos algoritmos executados pelo Neo4j. Sendo essa a única exceção feita para modificar a configuração, a fim de manter a configuração entre as ferramentas o mais semelhante o possível.

3.2 OBSERVAÇÕES QUALITATIVAS

Nesta seção são destacadas as principais diferenças observadas durante a implementação dos testes.

3.2.1 Sintaxe

A sintaxe de cada ferramenta é bem distinta, já nas operações básicas de contagem de nodos e de relações o Neo4j torna-se mais comprido, isso se deve ao seu paradigma de banco de dados, ao qual o foco é realizar consultas baseadas em propriedades como pode ser observado nos códigos-fonte a seguir.

Código-fonte 3.3: Contagem de nodos e relações, Neo4j, **sem** utilizar atributos

```
1 // Contagem de nodos
```

```
MATCH (nd) RETURN COUNT(nd)
// Contagem de relações
MATCH ()-[rel]->() RETURN COUNT(r)
```

Código-fonte 3.4: Contagem de nodos e relações, NetworkX, **sem** utilizar atributos

```
# Contagem de nodos
G.number_of_nodes()
# Contagem de relações
G.number_of_edges()
```

Os próximos dois códigos-fonte demostram como o Neo4j fica menos complexo e menor quando se utiliza atributos dos nodos.

Código-fonte 3.5: Contagem de nodos, Neo4j, **com** utilização de atributos

```
// Contagem de nodos com a propriedade cor vermelha
MATCH (nd {"color":"red"}) RETURN COUNT(nd)
```

Código-fonte 3.6: Contagem de nodos, NetworkX, **com** utilização de atributos

```
# Contagem de nodos com a propriedade cor vermelha
len(node
   for node, data
   in G.nodes(data=True)
   if data.get("color") == "red"
)
```

Não é o foco deste trabalho avaliar as ferramentas utilizando atributos de nodos e relações. Sendo essa primeira comparação apenas para que fique claro o porquê desta diferença entre as ferramentas. Não será mais demonstrado tal diferença nos algoritmos seguintes.

Para os algoritmos mais complexos, onde é necessária a utilização do *plugin* GDS para o Neo4j e é necessária uma etapa a mais preparando o subgrafo para o GDS processar. Esta etapa extra não é necessária para o NetworkX.

Código-fonte 3.7: Preparação do grafo para o GDS

```
CALL gds.graph.project(
   'myGraph', // Nome do subgrafo gerado
   'NODE_LABEL',
   { COAUTHORS: {
      orientation: 'UNDIRECTED',
      properties: 'cost'
     }
   }
)
```

Além da etapa extra no Neo4j, a chamada dos algoritmos do GDS também é muito mais comprida e com mais detalhes, uma vez que não está sendo utilizado as propriedade dos nodos e relações.

Código-fonte 3.8: Cálculo de componentes, Neo4j

```
CALL gds.wcc.stats('myGraph')
YIELD componentCount
```

Código-fonte 3.9: Cálculo de componentes, NetworkX

```
nx.number_connected_components(my_graph)
```

Código-fonte 3.10: Cálculo de grau, Neo4j

```
CALL gds.degree.stream('myGraph')
YIELD nodeId, score
RETURN nodeId AS id, score AS followers
```

Código-fonte 3.11: Cálculo de grau,NetworkX

```
my_graph.degree()
```

Código-fonte 3.12: Contagem de triângulos, Neo4j

```
CALL gds.triangleCount.stream('myGraph')
YIELD nodeId, triangleCount
RETURN nodeId AS id, triangleCount
```

Código-fonte 3.13: Contagem de triângulos, NetworxX

```
nx.triangles(my_graph)
```

Como já comentado na Seção 3.1, foi necessário realizar implementações distintas para o teste do cálculo do diâmetro, sendo a do Neo4j uma utilização do algoritmo de busca de todos os caminhos mínimos, mantendo somente a distância a cada processamento.

Código-fonte 3.14: Calculo do diâmetro, Neo4j

```
MATCH (source:AUTHOR)
   CALL gds.allShortestPaths.delta.stream("myGraph",{
      sourceNode:source,
      relationshipWeightProperty: 'cost',
      delta: 3.0
   })
   YIELD totalCost as distance
   WHERE gds.util.isFinite(distance) and distance > 1
   return max(distance)
```

Código-fonte 3.15: Calculo do diâmetro, NetworkX

```
def fullDiameter(graph):
   subs = (G.subgraph(comp).copy() for comp in nx.connected_components(G))
   return max(nx.algorithms.distance_measures.diameter(sub) for sub in subs)
```

Para o algoritmo de busca de todos os caminhos mínimos não foi diferente do descrito anteriormente, o código-fonte do Neo4j foi muito mais comprido. Neste caso, é interessante notar que devido às propriedades ACID que não permitem resultados parciais no Neo4j, sendo necessário utilizar o método `periodic.iterate` do *plugin* APOC, para que o processamento seja fracionado registrando os resultados periodicamente (*commit* de banco de dados).

Código-fonte 3.16: Todos os pares de caminhos mínimos, Neo4j

```
CALL apoc.periodic.iterate(
   "MATCH (source:AUTHOR ) return source",
   "CALL gds.allShortestPaths.delta.stream('myGraph',{
      sourceNode:source,
      relationshipWeightProperty: 'cost',
      delta: 3.0
   })
   YIELD index, sourceNode, targetNode, totalCost, nodeIds
   with gds.util.asNode(sourceNode) as s,
       gds.util.asNode(targetNode) as t,
       totalCost,
       nodeIds
   WHERE gds.util.isFinite(totalCost) and totalCost > 1
   CREATE (s)-[:SHORTEST_PATH{cost:totalCost, nodeIds:nodeIds}]-> (t)" ,
   {batchSize:100, parallel:true}
)
```

Código-fonte 3.17: Todos os pares de caminhos mínimos, NetworkX

```
nx.all_pairs_shortest_path(G)
```

3.2.2 Documentação

Ambas as ferramentas possuem a documentação oficial em seus sites muito bem detalhada com sintaxe, explicação e exemplos de utilização de forma intuitiva. Por lidar com *plugins* e também por ser um banco de dados, o Neo4j possui separadamente a documentação sobre a utilização de *drivers* de conexão com o banco de dados e a documentação dos *plugins*, o que acaba "poluindo"a busca de uma documentação especifica, mas facilitam na implementação.

3.2.3 Visualização

Caso seja necessário uma análise visual do grafo, ambas as ferramentas são capazes de gerar visualizações do grafo. O Neo4j possui a ferramenta com mais recursos nativos, sua visualização é iterativa, permitindo ver informações ou mudar a consulta feita com poucos cliques na interface. O NetworkX gera uma imagem estática, existem outras bibliotecas adicionais que se integram ao NetworkX que fazer semelhante à visualização do Neo4j.

3.2.4 Instalação e Configuração

Sendo uma biblioteca Python, a instalação do NetworkX é simples e pratica, bastando instalar o pacote PyPi com o comando `pip install networkx` em uma maquina que já possua o Python instalado.

Por ser um banco de dados, a instalação do Neo4j possui algumas configurações a mais como senha e usuário, local de armazenamento, conexões com o banco de dados, etc. deixando-o mais complexo. Ainda se for necessário acessar o Neo4j através de outro programa é necessária a instalação dos *drivers* e bibliotecas de conexão, aumentando ainda mais a complexidade de implantação.

4 CONCLUSÃO

Tanto em eficiência sob a perspectiva do uso de memória, quanto para o tempo de execução, o Neo4j demonstrou ser desvantajoso, demorando mais para executar os algoritmos, não sendo capaz de processar grafos tão grandes quanto o NetworkX. Este resultado se limita ao cenário definido no escopo do projeto.

Uma diferença marcante para o Neo4j é a instalação e configuração mais detalhadas e a sintaxe que é pensada para a utilização de filtros. O que não é feito neste cenário experimentado e acaba poluindo o código gerando maior complexidade na implementação e implantação.

Por fim, é importante lembrar que devido a utilização da configuração padrão de cada ferramenta, exceto pela configuração de omissão de *logs* do Neo4j, sem tentar otimizar as consultas e/ou algoritmos para que utilizassem melhor os recursos da maquina

4.1 TRABALHOS FUTUROS

Devido ao tempo curto para a execução do projeto e abrangência do tema, surgem algumas propostas de trabalhos que continuariam ou se aproveitariam deste.

4.1.1 Melhorias no projeto

Este trabalho comparou a execução dos algoritmos apenas experimentalmente, sem analisar a implementação dos algoritmos de cada ferramenta. Esta análise pode trazer uma informação mais precisa sobre o uso de memória e desempenho de cada ferramenta. Ainda seria interessante comparar a arquitetura de cada ferramenta e todo o fluxo da chamada dos algoritmos apontando quais as vantagens e desvantagens de cada implementação, algo análogo ao feito em Huang e Dong (2013).

Embora os testes estejam isolados por ferramenta e por *dataset*, a separação por algoritmo não foi tão completa, sendo possível haver interferências no desempenho dos testes de acordo com o uso de memória do teste do algoritmo anterior. Sendo outra possibilidade de melhoria realizar melhor este isolamento entre a execução de cada algoritmo.

4.1.2 Variações

Como os experimentos abordam apenas o Neo4j e o NetworkX, seria interessante adicionar outras ferramentas para serem testadas nos scripts. Uma sugestão é utilizar as mesmas ferramentas testadas em Akhtar (2014) com o método seguido neste trabalho.

Grafos possuem uma grande variedade de aplicação, o cenário abordado não contempla casos como os algoritmos de coloração de grafos ou consulta de nodos a partir de seus atributos. Outra possibilidade de variação do projeto é adicionar novos algoritmos aos testes que utilizem

atributos definidos aos nodos e relações. Tais propriedades poderiam ser melhor exploradas no Neo4j tendo em vista que este foi pensado para armazenamento e processamento dessas informações.

REFERÊNCIAS

Akhtar, N. (2014). Social network analysis tools. páginas 388–392.

Batra, S. e Tyagi, C. (2012). Comparative analysis of relational and graph databases. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(2):509–512.

Hagberg, A. A., Schult, D. A. e Swart, P. J. (2008). Exploring network structure, dynamics, and function using networkx. Em Varoquaux, G., Vaught, T. e Millman, J., editores, *Proceedings of the 7th Python in Science Conference*, páginas 11 – 15, Pasadena, CA USA.

Huang, H. e Dong, Z. (2013). Research on architecture and query performance based on distributed graph database neo4j. páginas 533–536.

Inc, N. (2007). Neo4j.

NetworkX (2005). Networkx.

Vito Latora, Vincenzo Nicosia, G. R. (2017). *Complex Networks: Principles, Methods and Applications*. Número ISBN 9781107103184. Cambridge University Press.