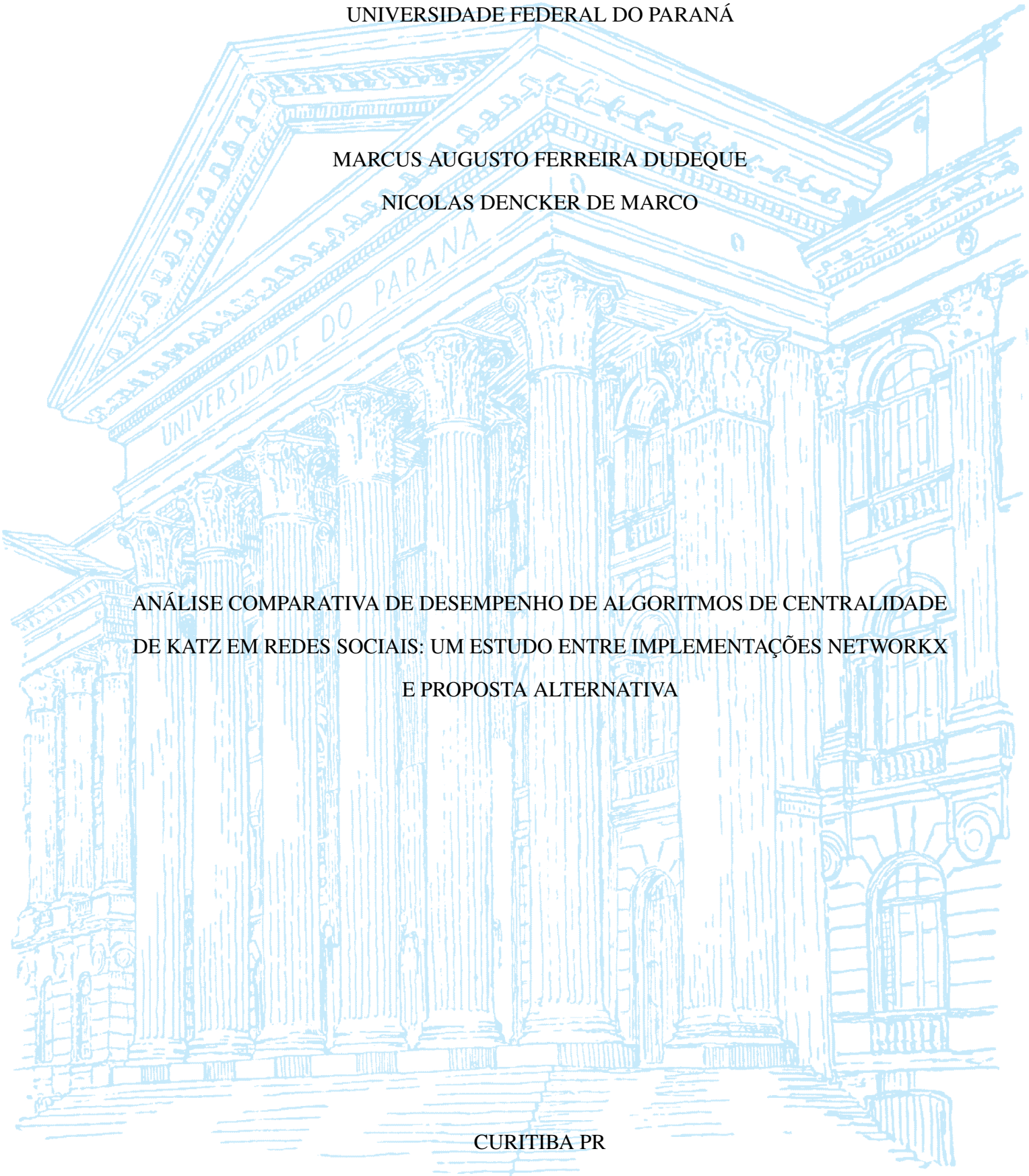
UNIVERSIDADE FEDERAL DO PARANÁ

MARCUS AUGUSTO FERREIRA DUDEQUE

NICOLAS DENCKER DE MARCO



ANÁLISE COMPARATIVA DE DESEMPENHO DE ALGORITMOS DE CENTRALIDADE DE KATZ EM REDES SOCIAIS: UM ESTUDO ENTRE IMPLEMENTAÇÕES NETWORKX E PROPOSTA ALTERNATIVA

CURITIBA PR

2024

MARCUS AUGUSTO FERREIRA DUDEQUE
NICOLAS DENCKER DE MARCO

ANÁLISE COMPARATIVA DE DESEMPENHO DE ALGORITMOS DE CENTRALIDADE DE KATZ EM REDES SOCIAIS: UM ESTUDO ENTRE IMPLEMENTAÇÕES NETWORKX E PROPOSTA ALTERNATIVA

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: *Computação*.

Orientador: André Luís Vignatti.

CURITIBA PR

2024

RESUMO

Este trabalho explora a aplicação de conceitos matemáticos e computacionais para análise de redes sociais usando a centralidade de Katz. A centralidade de Katz foi escolhida por seus resultados confiáveis e amplo uso de matrizes. Comparamos o desempenho do Algoritmo de Centralidade de Katz da biblioteca Python NetworkX com um algoritmo desenvolvido por nós. Utilizando os Grafos Barabási-Albert e GNP Random Graph para os testes, descobrimos que o algoritmo Barabási-Albert é, em média, 3,6 vezes mais rápido que o GNP para cálculos de centralidade de Katz. O algoritmo do NetworkX superou o nosso em 120% para Barabási-Albert e 25% para GNP. Apesar disso, nosso algoritmo mostra desempenho aceitável para grafos mais densos e tem potencial para melhorias.

Palavras-chave: Grafos. Centralidade. Katz.

ABSTRACT

This work explores the application of mathematical and computational concepts for analyzing social networks using Katz centrality. Katz centrality was chosen for its reliable results and extensive use of matrices. We compare the performance of the Katz Centrality Algorithm from the Python NetworkX library with a self-developed algorithm. Using Barabási-Albert and GNP Random Graphs for testing, we found that the Barabási-Albert algorithm is, on average, 3.6 times faster than GNP for Katz centrality calculations. NetworkX's algorithm outperformed ours by 120% for Barabási-Albert and 25% for GNP. Despite this, our algorithm shows acceptable performance for denser graphs and has potential for improvement.

Keywords: Graphs. Centrality. Katz.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O presente trabalho se propõe a explorar a aplicação de conceitos matemáticos e computacionais para análise de redes sociais, mais especificamente através do cálculo do grau de centralidade. Para isso escolhemos analisar mais profundamente o Algoritmo de Centralidade de Katz, principalmente por suas propriedades que envolvem tanto nós mais próximos ao nó objetivo, quanto nós mais distantes, tornando seu resultado mais confiável em relação a outros algoritmos similares. Outro motivo crucial na opção por Katz, foi a possibilidade de utilização da fórmula mencionada na seção 2.3, que tem ampla utilização de matrizes que, por sua vez, apresentam diversas propriedades que tornam sua exploração mais simples e direta.

Sabendo disso, definimos conceitos matemáticos básicos para melhor compreensão do trabalho, tais como *matrizes simétricas, traço, vetores e matrizes ortogonais*, entre outros. Passamos a definir conceitos e propriedades relevantes para o desenvolvimento do trabalho, como *matrizes singulares*, *determinante de Laplace*, *decomposição LU*, *autovalores e autovetores*, *polinônio e equação característicos* e *raio espectral*. Tudo isso para chegar à definição da *função resolvente*, item essencial para construção da fórmula que escolhemos utilizar para desenvolvimento do nosso trabalho.

Como objetivo, decidimos realizar testes comparativos de desempenho entre o Algoritmo de Centralidade de Katz da biblioteca Python NetworkX e um algoritmo desenvolvido por nós mesmos. Os grafos utilizados para as baterias de testes foram gerados por dois algoritmos de geração de grafos com lógicas análogas, a fim de compreendermos uma maior gama de possibilidades e termos um trabalho mais diverso. Os algoritmos utilizados foram Barabási-Albert, que é conhecido por sua capacidade de criar redes com propriedades semelhantes às encontradas em muitas redes sociais reais, contando comumente com a presença de hubs ou nós altamente conectados, e além dele, GNP Random Graph, que é conhecido por gerar grafos mais densos e ambíguos em termos de centralidade.

O NetworkX apresenta duas funções distintas para cálculo da centralidade de Katz. Uma delas se utiliza de força bruta e, portanto, tem desempenho inferior em relação à outra, que foi nossa escolhida para comparação. Esta se utiliza da mesma estrutura matemática que utilizamos para construção do nosso próprio algoritmo, tornando a comparação justa e válida. Os testes foram realizados em baterias de 100 repetições, com incremento de 1000 nós a cada bateria, começando em 1000 nós e comparando o tempo médio de execução entre os algoritmos.

Observamos uma grande diferença entre os tempos de execução dos tipos de geração de grafos. O algoritmo de Barabási-Albert é, em média, 3,6 vezes mais rápido que o algoritmo GNP para cálculo de centralidade de Katz utilizando qualquer dos dois algoritmos testados, utilizando o número de nós como base comparativa. Enquanto isso, observamos uma vantagem do algoritmo NetworkX sobre o nosso, sendo em Barabási-Albert 120% e em GNP 25% mais rápidos que o nosso.

Conclui-se que, apesar de sofrer leve desvantagem, nosso algoritmo apresenta desempenho aceitável, principalmente para grafos mais densos. E, pensando num trabalho futuro, com certos ajustes, uma melhora é perfeitamente possível.

2 CONCEITOS PRELIMINARES

2.1 DEFINIÇÕES BÁSICAS

Redes sociais podem ser representadas como matrizes, e ao longo deste trabalho vamos explorar essa característica. Utilizaremos várias definições e propriedades de matrizes que nos ajudarão a melhor compreender o comportamento de redes sociais. Estas definições e propriedades serão descritas a seguir.

2.1.1 Matrizes

Uma matriz A qualquer pode ser tanto quadrada (pertencendo a $\mathbb{R}^{n\times n}$) quanto retangular (pertencendo a $\mathbb{R}^{m\times n}$). Sua (i,j)-ésima entrada será denotada por $a_{i,j}$ e sua transposta é a matriz A^T cuja (i,j)-ésima entrada é $a_{j,i}$. Se $A = A^T$ então chamamos a matriz A de *simétrica*. O *traço* de uma uma matriz quadrada, denotado por tr(A), é definido como a soma dos elementos da diagonal principal da matriz A.

Vetores de tamanho n podem ser descritos como matrizes $n \times 1$ (se forem colunas) ou 1 $\times$ n (se forem linhas). O produto interno de dois vetores **x** e **y** de mesma dimensão é o valor escalar $\mathbf{x}^T\mathbf{y}$. Se o produto interno for zero, então os vetores **x** e **y** são ditos *ortogonais* entre si. Uma *matriz ortogonal* é uma matriz quadrada na qual todas suas colunas são mutuamente ortogonais. Se A é ortogonal, então $A^TA = I$, isto é, a multiplicação da matriz A pela sua transposta resulta na matriz identidade. (Anton e Rorres, 2011b)

[Matrizes Singulares] Na matemática chamamos de **singular** uma matriz quadrada quando esta não admite uma inversa. Uma matriz é **singular** se e somente se seu determinante é nulo. Isto é, se uma matriz quadrada apresentar ao menos uma linha ou coluna nula, terá determinante zero, caracterizando uma **matriz singular**. (Anton e Rorres, 2011b)

[Determinante de Laplace] Seja A $\in \mathbb{R}^{n\times n}$ então sua **determinante**, escrita $det(A)$ é definida por

$$det(A) = \begin{cases} A, & \text{n = 1,} \\ \sum_{j=1}^{n}(-1)^{i+j}a_{ij}det(A_{ij}), & \text{n > 1,} \end{cases} \tag{2.1}$$

para qualquer i fixo, onde A_{ij} denota a submatriz formada através da deleção das i-ésima linha e j-ésima coluna de A.

Os determinantes das matrizes $(n-1)\times(n-1)$ usados na definição são conhecidos como *menores*. Em particular, uma $k \times k$ *menor principal* é o determinante de uma submatriz principal $k \times k$ de A formada pela interseção de k linhas de A com k colunas. (Iezzi, 1977)

O determinante tem uma série de usos teóricos que decorrem do fato de que diversas identidades podem ser estabelecidas, como por exemplo det(AB) = det(A) det(B) e det(A^T) = det(A). A propriedade teórica mais útil de um determinante é que ele pode ser usado para caracterizar matrizes singulares: uma matriz quadrada A é singular se e somente se det(A) = 0, vide definição 2.1.1. (Anton e Rorres, 2011b)

Exemplos

1. Se A é uma matriz 2×2 então, sendo $i = 1$ na definição,

$$det(A) = a_{11}a_{22} - a_{12}a_{21}. \tag{2.2}$$

E se $A \in \mathbb{R}^{3\times3}$,

$$det(A) = a_{11}(a_{22}a_{33} - a_{23}a_{32}) - a_{12}(a_{21}a_{33} - a_{23}a_{31}) + a_{13}(a_{21}a_{32} - a_{22}a_{31})$$
$$= a_{11}a_{22}a_{33} + a_{21}a_{32}a_{13} + a_{31}a_{12}a_{23} - a_{13}a_{22}a_{31} - a_{23}a_{32}a_{11} - a_{33}a_{12}a_{21}. \quad (2.3)$$

2. Seja
$$A = \begin{bmatrix} 1 & 3 & 1 \\ 2 & 1 & 1 \\ 0 & 3 & 1 \end{bmatrix}. \quad (2.4)$$
Então $\det(A) = 1 + 6 + 0 - 0 - 3 - 6 = 2$. Existem três menores principais 2×2, $\det(A_{11}) = 1 - 3 = -2$, $\det(A_{22}) = 1 - 0 = 1$, e $\det(A_{33}) = 1 - 6 = -5$.

3. Suponha que G é uma rede simples conectada com três nós. Então sua matriz de adjacência é
$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}. \quad (2.5)$$
Ou uma permutação de
$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}. \quad (2.6)$$
A primeira matriz representa K_3, um triângulo, e o segundo é a matriz de adjacência de P_2. O determinante da primeira matriz é 2 enquanto o da segunda é 0. Nesse caso simples específico podemos caracterizar redes conectadas através de seus determinantes.

2.1.2 Decomposição LU (Anton e Rorres, 2011a)

Para explicar a Decomposição LU, vamos exemplificar uma decomposição de uma matriz 2×2.

$$\begin{bmatrix} 4 & 3 \\ 6 & 3 \end{bmatrix}$$

Queremos encontrar duas matrizes L e U tal que $A = LU$, onde L é uma matriz triangular inferior e U é uma matriz triangular superior.

Passos para a Decomposição LU

1. Inicialize L e U: Começamos com as formas gerais das matrizes L e U:

$$L = \begin{bmatrix} 1 & 0 \\ l_{21} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix}$$

2. Defina os elementos de U: - O elemento u_{11} é simplesmente o primeiro elemento da matriz A:

$$u_{11} = a_{11} = 4$$

- O elemento u_{12} é o segundo elemento da primeira linha de A:

$$u_{12} = a_{12} = 3$$

3. Calcule o elemento de L: - O elemento l_{21} é calculado como o fator necessário para zerar o elemento abaixo de u_{11}:

$$l_{21} = \frac{a_{21}}{u_{11}} = \frac{6}{4} = 1.5$$

4. Calcule o elemento restante de U: - O elemento u_{22} é calculado após subtrair a contribuição da matriz L:

$$u_{22} = a_{22} - l_{21} \cdot u_{12} = 3 - 1.5 \cdot 3 = 3 - 4.5 = -1.5$$

Portanto, as matrizes L e U são:

$$L = \begin{bmatrix} 1 & 0 \\ 1.5 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 4 & 3 \\ 0 & -1.5 \end{bmatrix}$$

Vamos verificar a decomposição multiplicando L e U:

$$LU = \begin{bmatrix} 1 & 0 \\ 1.5 & 1 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 0 & -1.5 \end{bmatrix} = \begin{bmatrix} 1 \cdot 4 + 0 \cdot 0 & 1 \cdot 3 + 0 \cdot (-1.5) \\ 1.5 \cdot 4 + 1 \cdot 0 & 1.5 \cdot 3 + 1 \cdot (-1.5) \end{bmatrix} = \begin{bmatrix} 4 & 3 \\ 6 & 3 \end{bmatrix}$$

Como esperado, obtemos a matriz A. E, portanto, a decomposição LU da matriz A foi realizada com sucesso, resultando nas matrizes L e U. Esse método é eficaz para resolver sistemas de equações lineares, calcular determinantes e inverter matrizes.

2.1.3 Autovalores e Autovetores

Para qualquer matriz A quadrada de tamanho n existem valores escalares (pelo menos 1 e no máximo n) e vetores **x** (pelo menos um pra cada valor de λ) tais que

$$A\mathbf{x} = \lambda\mathbf{x}. \tag{2.7}$$

Qualquer valor de λ que satisfaça esta equação é chamado de **autovalor**. Qualquer vetor **x** não nulo que satisfaça essa equação é chamado de **autovetor**.

Para todos os valores de λ temos $A0 = \lambda 0$. O vetor zero não é considerado um autovetor, entretanto. Temos

$$A\mathbf{x} = \lambda\mathbf{x} \longleftrightarrow (A - \lambda I)\mathbf{x} = 0, \tag{2.8}$$

o que significa que a matriz $A - \lambda I$ é uma matriz singular, isto é, que tem determinante nulo. Com isso estabelecemos o resultado de que os autovalores de λ são as raízes da equação $\det(A - \lambda I) = 0$, um polinômio de grau n. (Anton e Rorres, 2011c)

O polinômio

$$det(A - \lambda I) = b_0 + b_1\lambda + b_2\lambda^2 + \cdots + b_n\lambda^n \tag{2.9}$$

é conhecido como **polinômio característico** de A. A equação $det(A - \lambda I) = 0$ é conhecida como a **equação característica**.

Uma vez que sabemos os autovalores, podemos fatorar o polinômio característico de forma que

$$det(A - \lambda I) = (\lambda_1 - \lambda)^{p1}(\lambda_2 - \lambda)^{p2} \ldots (\lambda_k - \lambda)^{pk} \tag{2.10}$$

onde os elementos λ_i são únicos. Geralmente $k = n$ e $p_i = 1$ para todo i. Mas pode acontecer de encontrarmos *autovalores repetidos* nos quais $p_i > 1$. O valor de p_i é conhecido como *multiplicidade algébrica* de λ_i. Para cada autovalor distinto há entre 1 e p_i autovetores

linearmente independentes. O número de autovetores associados com um autovalor é chamado de *multiplicidade geométrica*.

Existem algumas informações muito úteis que podemos tirar do polinômio característico. É possível associar seus coeficientes com menores principais: $(-1)^k b_{n-k}$ é a soma de todas as $k \times k$ menores principais de A. Sabemos que $tr(A)$ é a soma das menores principais 1×1 de A, $-tr(A)$ é o coeficiente de λ^{n-1} no polinômio característico. Ao mesmo tempo, se A tem autovalores $\lambda_1, \lambda_2, \ldots, \lambda_n$ então

$$det(\lambda I - A) = (\lambda - \lambda_1)(\lambda - \lambda_2) \cdots (\lambda - \lambda_n) \tag{2.11}$$

e multiplicando o lado direito obtemos o coeficiente de λ^{n-1}

$$-\lambda_1 - \lambda_2 - \cdots - \lambda_n \tag{2.12}$$

o que significa que a soma de todos os autovalores de uma matriz é igual ao seu traço.

Para uma matriz real genérica A, as raízes de seu polinômio característico podem ser complexas. Note que se A é uma matriz real, então

$$det(A - \lambda I) = det((A - \lambda I)^*) = det(A^T - \bar{\lambda} I). \tag{2.13}$$

Então se λ é uma raiz do polinômio característico de A, $\bar{\lambda}$ também deve ser uma raiz do polinômio característico de A. Podemos concluir que A e A^T têm os mesmos autovalores. Supondo que $A\mathbf{x} = \lambda\mathbf{x}$ e $A^T\mathbf{y} = \bar{\lambda}\mathbf{y}$, temos que $\mathbf{y}^* A = \lambda\mathbf{y}^*$. Então $\mathbf{x}$ é o autovetor à direita de A (correspondente a λ) e $\mathbf{y}^*$ é o autovetor à esquerda. (Anton e Rorres, 2011c)

2.1.4 Raio Espectral

O **espectro** de uma matriz é o conjunto de seus autovalores e é escrito como $\sigma(A)$ (Estrada e Knight, 2015a). Isto é

$$\sigma(A) = \{\lambda \colon det(A - \lambda I) = 0\}. \tag{2.14}$$

O **raio espectral** de A é definido pelo módulo de seu maior autovalor e é escrito como ρ (A). Isto é,

$$\rho(A) = max_{\lambda \in \sigma(A)} |\lambda| \tag{2.15}$$

O raio espectral de uma matriz pode ser uma medida particularmente útil para as propriedades de uma matriz.

Suponha $A\mathbf{x} = \lambda\mathbf{x}$. Então $A^k\mathbf{x} = A^{k-1}(A\mathbf{x}) = \lambda A^{k-1}\mathbf{x}$. Essa identidade forma a base de uma prova indutiva que se λ é um autovalor de A então λ^k é um autovalor de A^k. Consequentemente $\rho(A^k) = \rho(A)^k$. Se $\rho(A) < 1$ então $\rho(A^k) \to 0$ já que $k \to \infty$, podemos concluir que $A^k \to O$ também. Se $\rho(A) > 1$ podemos mostrar que A^k diverge. Se $\rho(A) = 1$ então A^k pode ou não convergir.

Já sabemos que para matrizes reais, $\rho(A^T) = \rho(A)$. Para matrizes complexas sabemos que λ é um autovalor de A se e somente se $\bar{\lambda}$ é um autovalor de A^* e, novamente, $\rho(A^*) = \rho(A)$. Além disso, se A é uma matriz regular e $A\mathbf{x} = \lambda\mathbf{x}$ então $A^{-1}\mathbf{x} = \lambda^{-1}\mathbf{x}$, e podemos concluir que

$$\rho(A^{-1}) = max_{\lambda \in \sigma(A)} \frac{1}{|\lambda|}. \tag{2.16}$$

2.2 FUNÇÃO RESOLVENTE

1. Se $z \in \mathbb{C}$ e $z \neq 1$ então

$$1 + z + z^2 + \cdots + z^p = \frac{1 - z^{p+1}}{1 - z}. \tag{2.17}$$

Se $|z| < 1$ então

$$\sum_{i=0}^{\infty} z^i = (1 - z)^{-1}. \tag{2.18}$$

As equações 2.17 e 2.18 são, respectivamente, as equações de progressão geométrica finita e infinita, e, por isso, conhecemos seus resultados. A função $f(z) = (1 - z)^{-1}$ é uma *continuação analítica* de $\sum_{i=0}^{\infty} z^i$ dentro do conjunto $\mathbb{C} - \{1\}$.

2. É interessante notar que os resultados de progressão geométrica também se aplicam para o caso onde as variáveis são matrizes. Por exemplo,

$$(I + A + A^2 + \cdots + A^p)(I - A) = I - A^{p+1} \tag{2.19}$$

Se $(I - A)^{-1}$ existe, então podemos escrever

$$I + A + A^2 + \cdots + A^p = (I - A^{p+1})(I - A)^{-1} = (I - A)^{-1}(I - A^{p+1}). \tag{2.20}$$

$(I - A)$ é invertível contanto que 1 não seja um autovalor de A.

3. Se $\rho(A) < 1$ então, desde que $\lim_{p\to\infty} A^p = O$,

$$\sum_{i=0}^{\infty} A^i = (I - A)^{-1}. \tag{2.21}$$

Se $|s| < 1/\rho(A)$ então $\rho(sA) < 1$ e daí

$$s \sum_{i=0}^{\infty} (sA)^i = s(I - sA)^{-1} = (zI - A)^{-1}, \tag{2.22}$$

onde $z = 1/s$.

A matriz $(zI - A)^{-1}$ é definida desde que $(zI - A)$ não tenho autovalores zero, o que é o caso desde que $z \notin \sigma(A)$. Dada uma matriz A, chamamos

$$f(z) = (zI - A)^{-1} \tag{2.23}$$

sua *função resolvente*. A função resolvente é uma continuação analítica de uma série infinita de potências de matriz até (quase) todo o plano dos complexos. Tem um grande número de aplicações em teoria de redes sociais, mas também é útil para entender funções de matriz (apesar de ela própria não ser uma função de matriz). (Estrada e Knight, 2015a)

2.3 FÓRMULA

2.3.1 Fórmula Original

O grau do nó i conta o número de caminhos de comprimento um partindo de i para qualquer outro nó da rede. Isto é, $k_i = (A\mathbf{e})_i$. Katz estendeu essa ideia para contar caminhos com comprimento maior que um. A partir disso, pode se inferir que os vizinhos mais próximos tenham mais influência sobre o nó i que os mais distantes. Quando combinados os caminhos de todos os comprimentos, foi possível introduzir o fator de atenuação (α) para que mais peso fosse atribuído aos caminhos mais curtos em detrimento dos caminhos mais longos (Estrada e Knight, 2015b). Isso fez com que Katz chegasse ao seguinte:

$$K_i = [\alpha^0 A^0 + \alpha A + \alpha^2 A^2 + \cdots + \alpha^k A^k + \cdots)\mathbf{e}]_i = \left[\sum_{k=0}^{\infty} (\alpha^k A^k)\mathbf{e}\right] \tag{2.24}$$

E como visto na seção 2.2, essa série é relacionada com a função resolvente $(zI - A)^{-1}$, e converge desde que $\alpha < \rho(A)$, nos levando assim à próxima fórmula.

2.3.2 Fórmula Final

Por sua clareza e mais fácil compreensão, decidimos por utilizar a seguinte fórmula, dentre as possibilidades para Centralidade de Katz:

$$K_i = [(I - \alpha A)^{-1}\mathbf{e})]_i \tag{2.25}$$

O funcionamento desta equação é muito simples:

- Primeiro aplica-se o fator de atenuação à matriz de adjacência A, através da multiplicação de α por A;
- Em seguida, subtrai-se o resultado da multiplicação realizada no passo anterior da matriz identidade;
- Inverte-se o resultado desta subtração;
- E então multiplica-se esse resultado por **e**, que nada mais é que um vetor preenchido com números 1.

Desta forma temos a construção de um vetor com todos os coeficientes de centralidade de Katz para cada um dos nós do grafo. (Estrada e Knight, 2015b)

3 DISCUSSÃO

3.1 ALGORITMO NETWORKX

3.1.1 Introdução

NetworkX é um pacote Python para exploração e análise de redes sociais e algoritmos relacionados. O pacote principal fornece estruturas para representar diversos tipos de redes sociais e grafos (simples, direcionados, com arestas paralelas ou laços). Os nós no NetworkX podem ser quaisquer objetos Python (hasháveis), enquanto arestas podem conter dados arbitrários; essa flexibilidade faz do NetworkX ideal para representar redes sociais provenientes de diversos campos da ciência. Além das estruturas básicas, há a implementação de diversos algoritmos de grafos para cálculos estruturais e de propriedades, como por exemplo: caminho mais curto, centralidade de intermediação, clusterização, distribuição de grau, entre muitos outros. O NetworkX pode ler e escrever diversos formatos de grafos, além de fornecer geradores para vários tipos de grafos clássicos e populares, como o Erdős–Rényi, Pequeno Mundo e Barabási-Albert. A facilidade no uso e a flexibilidade da linguagem Python aliados às ferramentas SciPy tornam o NetworkX uma ferramenta muito poderosa para computação na área das ciências. (WIKIPEDIA, b)

3.1.2 Geração de Grafos - Barabási-Albert

O NetworkX disponibiliza inúmeras funções de geração de grafos, cada uma com sua respectiva estratégia. Entre elas decidimos usar os algoritmos de Barabási–Albert e GNP Random Graph. Explicamos o porquê a seguir.

Decidimos pelo uso do Algoritmo de Barabási-Albert considerando que ele gera redes livres de escala usando mecanismo de conexão preferencial, isto é, a cada nó gerado no grafo ele adiciona um determinado número de arestas, preferencialmente a nós com maior grau de centralidade.

Listing 3.1: Algoritmo de Bárabasi-Albert do NetworkX (NETWORKX, a)

```
def barabasi_albert_graph(n, m, seed=None, initial_graph=None):
   if m < 1 or m >= n:
      raise nx.NetworkXError(
         f"BarabásiAlbert network must have m >= 1 and m < n, m = {m}, n = {n}"
      )

   if initial_graph is None:
      # Default initial graph : star graph on (m + 1) nodes
      G = star_graph(m)
   else:
      if len(initial_graph) < m or len(initial_graph) > n:
         raise nx.NetworkXError(
            f"BarabásiAlbert initial graph needs between m={m} and n={n} nodes"
         )
      G = initial_graph.copy()

   # List of existing nodes, with nodes repeated once for each adjacent edge
   repeated_nodes = [n for n, d in G.degree() for _ in range(d)]
   # Start adding the other n - m0 nodes.
   source = len(G)
```

```
    while source < n:
        # Now choose m unique nodes from the existing nodes
        # Pick uniformly from repeated_nodes (preferential attachment)
        targets = _random_subset(repeated_nodes, m, seed)
        # Add edges to m nodes from the source.
        G.add_edges_from(zip([source] * m, targets))
        # Add one node to the list for each new edge just created.
        repeated_nodes.extend(targets)
        # And the new node "source" has m edges to add to the list.
        repeated_nodes.extend([source] * m)

        source += 1
    return G
```

Diversos sistemas naturais ou artificiais, incluindo a Internet, são vistos por serem aproximadamente livres de escala e certamente contém alguns nós com um incomum alto grau comparado a outros nós da rede. O modelo de Barabási–Albert tenta replicar a existência desses nós, nos ajudando a proporcionar testes com dados mais próximos da realidade em que vivemos. (WIKIPEDIA, a)

3.1.3 Geração de Grafos - GNP Random Graph

A escolha do algoritmo GNP partiu de uma resposta à utilização do Algoritmo Barabási-Albert. Os grafos G(n, p) são excelentes para *benchmarking*, isto é, são ótimas fontes de comparação para outros estudos similares (Erdős e Rényi, 1959). E como seu funcionamento é complementar ao de Barabási-Albert, decidimos por sua utilização para termos uma análise mais diversificada e menos enviesada.

Listing 3.2: Algoritmo GNP Random Graph do NetworkX (NETWORKX, b)

```
def gnp_random_graph(n, p, seed=None, directed=False):
    if directed:
        edges = itertools.permutations(range(n), 2)
        G = nx.DiGraph()
    else:
        edges = itertools.combinations(range(n), 2)
        G = nx.Graph()
    G.add_nodes_from(range(n))
    if p <= 0:
        return G
    if p >= 1:
        return complete_graph(n, create_using=G)

    for e in edges:
        if seed.random() < p:
            G.add_edge(*e)
    return G
```

3.1.4 Centralidade de Katz - Força Bruta

Utilizando os grafos gerados a partir dos algoritmos de Barabási–Albert e GNP Random Graph, buscamos definir a centralidade de cada nó para então podermos afirmar qual nó é mais central numa rede gigante e complexa, para isso o NetworkX possui duas funções que calculam e retornam a centralidade de Katz a partir de um determinado grafo *G* (NETWORKX, c),

`1. katz_centrality() e 2. katz_centrality_numpy()`

A primeira função, como podemos ver no código a seguir, percorre todos os nós de G somando o grau de centralidade do nó iterado aos graus de centralidade dos seus nós vizinhos, sempre usando o fator atenuante α para facilitar os cálculos computacionais evitando *overflows*. Ao fim do algoritmo cada nó possui seu grau de centralidade e podemos dizer que o nó com maior grau é o nó mais central, tendo sua centralidade baseada no grau de centralidade dos nós com os quais ele é conectado.

Listing 3.3: Algoritmo de Centralidade de Katz do NetworkX (NETWORKX, c)

```
@not_implemented_for("multigraph")
@nx._dispatchable(edge_attrs="weight")
def katz_centrality(
    G,
    alpha=0.1,
    beta=1.0,
    max_iter=1000,
    tol=1.0e-6,
    nstart=None,
    normalized=True,
    weight=None,
):

    if len(G) == 0:
        return {}

    nnodes = G.number_of_nodes()

    if nstart is None:
        # choose starting vector with entries of 0
        x = {n: 0 for n in G}
    else:
        x = nstart

    try:
        b = dict.fromkeys(G, float(beta))
    except (TypeError, ValueError, AttributeError) as err:
        b = beta
        if set(beta) != set(G):
            raise nx.NetworkXError(
                "beta dictionary must have a value for every node"
            ) from err

    # make up to max_iter iterations
    for _ in range(max_iter):
        xlast = x
        x = dict.fromkeys(xlast, 0)
        # do the multiplication y^T = Alpha * x^T A + Beta
        for n in x:
            for nbr in G[n]:
                x[nbr] += xlast[n] * G[n][nbr].get(weight, 1)
        for n in x:
            x[n] = alpha * x[n] + b[n]

        # check convergence
        error = sum(abs(x[n] - xlast[n]) for n in x)
        if error < nnodes * tol:
            if normalized:
                # normalize vector
```

```
50              try:
51                  s = 1.0 / math.hypot(*x.values())
52              except ZeroDivisionError:
53                  s = 1.0
54          else:
55              s = 1
56          for n in x:
57              x[n] *= s
58          return x
59      raise nx.PowerIterationFailedConvergence(max_iter)
```

A forma como a centralidade é calculada nessa função é percorrendo todos os nós e todos os seus vizinhos pelo menos uma vez.

O algoritmo ao calcular a centralidade de Katz em um grafo tem uma complexidade exponencial, o que o torna impraticável para grafos de tamanho médio a grande, pensando principalmente no uso de memória. Vamos detalhar o raciocínio:

1. Número de iterações (‘max_iter‘): O algoritmo itera um número fixo de vezes determinado pelo parâmetro ‘max_iter‘. Portanto, a complexidade é $O(\text{max\_iter})$. Nos nossos testes estávamos utilizando o valor padrão, 1000 iterações.

2. Operações por iteração: O código itera sobre todos os nós e, para cada nó, sobre todos os seus vizinhos. Para cada vizinho, há uma operação de soma ponderada e uma multiplicação por um fator atenuante (‘alfa‘). Em termos de complexidade, no pior caso, em que cada nó tem conexão com todos os outros nós, isso seria $O(n^2)$, onde n é o número de nós no grafo.

3. Convergência: O critério de convergência é baseado em um valor de tolerância (‘tol‘). Em cada iteração, é calculado o erro entre o vetor atual e o vetor anterior. Isso envolve uma soma sobre todos os nós, o que é $O(n)$.

Portanto, somando esses pontos, a complexidade total do algoritmo de Katz Centrality no pior caso pode ser descrita como:

$$O(\text{max\_iter} \times n^2)$$

onde n é o número de nós no grafo e ‘max_iter‘ é o número máximo de iterações especificado. Esta é uma estimativa grossa e pode variar dependendo da estrutura do grafo (densidade, distribuição de conexões, etc.), mas fornece uma boa indicação do esforço computacional esperado. Ao tentarmos executar testes utilizando este algoritmo nos deparamos com grande limitação no número de nós, tornando inviável sua utilização para efeito comparativo, já que para por volta de 50 nós o algoritmo já estourava erro e não conseguia efetuar o cálculo devido ao alto uso de memória que os *dicts* consomem para essa quantidade de iterações.

3.1.5 Centralidade de Katz - NumPy

A segunda função é a que decidimos utilizar para comparação de testes com o algoritmo desenvolvido por nós. A centralidade de Katz nesta função é calculada através de cálculos matemáticos pautados em operação de matrizes como podemos ver no listing 3.4. Primeiramente, o algoritmo gera a matriz de adjacência A e define seu tamanho como n, respectivamente, nas linhas 21 e 22.

Então na linha 23, ele atenua por α a matriz de adjacência A do grafo G, depois a subtrai da matriz identidade I de tamanho n, e por último realiza o cálculo de um sistema de equações tendo como parâmetros o resultado da subtração anterior com a matriz B de apenas uma coluna de tamanho n com todos seus valores sendo β (parâmetro passado na chamada da

função, usado por nós com valor 1.0). Os valores resultantes do sistema de equações são os graus de centralidade de Katz de cada nó.

Na linha 26 ele normaliza o vetor de centralidade, caso o parâmetro tenha sido recebido como True. E por fim retorna um dicionário com o formato de número do nó como identificador e o valor da centralidade resultante como valor do respectivo nó.

Listing 3.4: Algoritmo de Centralidade de Katz Numpy do NetworkX (NETWORKX, c)

```
1  @not_implemented_for("multigraph")
2  @nx._dispatchable(edge_attrs="weight")
3  def katz_centrality_numpy(G, alpha=0.1, beta=1.0, normalized=True, weight=None):
4
5  import numpy as np
6
7     if len(G) == 0:
8        return {}
9     try:
10       nodelist = beta.keys()
11       if set(nodelist) != set(G):
12          raise nx.NetworkXError("beta dictionary must have a value for every node")
13       b = np.array(list(beta.values()), dtype=float)
14    except AttributeError:
15       nodelist = list(G)
16       try:
17          b = np.ones((len(nodelist), 1)) * beta
18       except (TypeError, ValueError, AttributeError) as err:
19          raise nx.NetworkXError("beta must be a number") from err
20
21    A = nx.adjacency_matrix(G, nodelist=nodelist, weight=weight).todense().T
22    n = A.shape[0]
23    centrality = np.linalg.solve(np.eye(n, n) - (alpha * A), b).squeeze()
24
25    # Normalize: rely on truediv to cast to float, then tolist to make Python numbers
26    norm = np.sign(sum(centrality)) * np.linalg.norm(centrality) if normalized else 1
27    return dict(zip(nodelist, (centrality / norm).tolist()))
```

Podemos notar que o networkX utiliza a função numpy.linalg.solve() (Walt et al., 2011). A função ‘numpy.linalg.solve‘ é utilizada para resolver sistemas lineares de equações do tipo $Ax = b$, onde A é uma matriz de coeficientes e b é um vetor constante. A complexidade computacional dessa função é principalmente determinada pela decomposição LU (decomposição de Cholesky em casos de matrizes simétricas e positivas definidas) e pela subsequente solução dos sistemas triangulares resultantes.

Em termos de complexidade:

1. Decomposição LU (ou Cholesky para matrizes simétricas positivas definidas): A decomposição LU de uma matriz A de tamanho $n \times n$ tem complexidade $O(n^3)$. Para matrizes simétricas positivas definidas, a decomposição de Cholesky tem complexidade $O(n^3)$.

2. Solução dos sistemas triangulares: Depois da decomposição, resolver os sistemas triangulares (*forward e backward substitution*) tem complexidade $O(n^2)$.

Portanto, a complexidade total da função ‘numpy.linalg.solve‘ para resolver um sistema linear $Ax = b$, onde A é uma matriz de tamanho $n \times n$, é geralmente $O(n^3)$. Isso reflete o custo computacional associado à decomposição da matriz e à subsequente resolução do sistema resultante (Trefethen e III, 1997).

3.2 NOSSO ALGORITMO

Nosso algoritmo tem funcionamento em acordo com a fórmula descrita anteriormente na seção 2.3. Nesta seção descreveremos o código, explicando porquê utilizamos determinadas bibliotecas e funções, além de descrever seus devidos funcionamentos.

3.2.1 Código

Inicialmente nós utilizamos a função de geração de grafo G do NetworkX baseada no algoritmo Barabási-Albert discutido na seção 3.1.2. Definimos n sendo o número de nós do Grafo G gerado.

Com o grafo G como base, montamos sua matriz de adjacência A de tamanho n através da função do NetworkX e uma matriz identidade I de mesmo tamanho. Com as matrizes A e I temos tudo que precisamos para então realizar o cálculo. E então calculamos da seguinte forma:

Listing 3.5: Algoritmo de Centralidade de Katz do Projeto

```
1 def katz_centrality(G, alfa, node_quantity):
2     A = nx.adjacency_matrix(G)
3     A = A.todense()
4     e = [[1.0]] * node_quantity
5     alfaA = A*alfa #(1)
6     M = np.subtract(np.identity(node_quantity), alfaA) #(2)
7     invM = np.linalg.inv(M) #(3)
8     K = invM * e #(4)
```

1. Multiplicamos a Matriz de Adjacência A pelo fator atenuante α, usado por nós com valor padrão de 0,1.

2. Diminuimos a matriz identidade I pela matriz A atenuada.

3. Realizamos a inversão da matriz que resulta dessa subtração.

4. Multiplicamos a matriz invertida pelo e.

No fim temos uma matriz M de tamanho n, assim como a matriz de adjacência inicial, portanto definimos cada indíce $K[i] = M[i][i]$ sendo K um array de tamanho n contendo o valor da centralidade de Katz do nó número i.

Em termos de complexidade computacional, a principal função que afeta o algoritmo é o numpy.linalg.inv() (Walt et al., 2011). A função 'numpy.linalg.inv()' é usada para calcular a inversa de uma matriz quadrada A, ou seja, uma matriz A^{-1} tal que $A \cdot A^{-1} = I$, onde I é a matriz identidade.

A complexidade computacional da função 'numpy.linalg.inv()' pode variar dependendo do algoritmo utilizado para calcular a inversa da matriz. As principais abordagens incluem:

1. Método de Eliminação de Gauss-Jordan: O método direto de Eliminação de Gauss-Jordan tem complexidade $O(n^3)$ (Sudrastawa et al., 2022), onde n é a ordem da matriz A. Este método é direto e geralmente usado para matrizes pequenas ou quando a precisão numérica é crítica.

2. Decomposição LU, descrita anteriormente.

Portanto, a complexidade da função 'numpy.linalg.inv()' para calcular a inversa de uma matriz quadrada A de ordem $n \times n$ geralmente é $O(n^3)$. Isso reflete o custo computacional associado à decomposição da matriz e à subsequente construção da matriz inversa.

3.2.2 Bibliotecas

Numpy (Walt et al., 2011), que significa Numerical Python, é uma poderosa biblioteca da linguagem de programação Python. Consiste em objetos chamados de arrays (matrizes), que são multidimensionais. Mas mais importante, o Numpy vem com uma coleção de rotinas e funções para processar esses arrays com maior facilidade.

O Numpy fornece um grande conjunto de funções e operações de biblioteca que ajudam os programadores a executar facilmente cálculos numéricos, sendo nesse caso, as funções focadas em geração e cálculo de matrizes.

A biblioteca Math concede acesso às funções matemáticas definidas por padrão em C.

A descrição do NetworkX vimos na seção 3.1

3.3 TESTES DEFINIDOS E RESULTADOS

Para executar os testes definimos então que usaríamos o nosso algoritmo baseado na Centralidade de Katz em acordo com a fórmula apresentada na seção 2.3 e compará-lo com o algoritmo de Centralidade de Katz do NetworkX. Nosso objetivo era comparar o tempo médio de execução de ambos os algoritmos.

Para execução dos testes, utilizamos os geradores de grafos descritos nas seções 3.1.2 e 3.1.3. A fim de manter isonomia nos testes, utilizamos sempre os mesmos parâmetros de execução.

O código foi feito da seguinte forma:

Listing 3.6: Geração dos grafos

```
   node_quantity = 1000 #Exemplo
      edge_quantity = 3 #Exemplo
      G = nx.barabasi_albert_graph(node_quantity, edge_quantity)
   #G = nx.gnp_random_graph(node_quantity, 0.25)
```

Segue a comparação abaixo:

Listing 3.7: Comparação dos tempos médios por algoritmo

```
for i in range(0, repetitions):
      G = nx.barabasi_albert_graph(node_quantity, edge_quantity)
      # G = nx.gnp_random_graph(node_quantity, 0.25)

      start_time_alg = time.time()
      katz_centrality(G, alfa, node_quantity)
      end_time_alg = end_time = time.time()
      elapsed_time_alg = end_time_alg - start_time_alg
      total_elapsed_time_alg += elapsed_time_alg

      start_time_nx_numpy = time.time()
      nx.katz_centrality_numpy(G, alfa, 1)
      end_time_nx_numpy = end_time = time.time()
      elapsed_time_nx_numpy = end_time_nx_numpy - start_time_nx_numpy
      total_elapsed_time_nx_numpy += elapsed_time_nx_numpy

print("Tempo médio do nosso alg: ", total_elapsed_time_alg/repetitions)
print("Tempo médio do networkX numpy: ", total_elapsed_time_nx_numpy/repetitions)
```

Objetivando mantermos cenários justos, geramos os grafos com cada dupla de parâmetros 100 vezes, incrementando a quantidade de nós em 1000 a cada nova rodada de testes, independente do algoritmo de geração de grafo. Para cada grafo novo, executamos o algoritmo desenvolvido por

nós e o algoritmo numpy do networkX, coletando o tempo de cada execução. Ao fim, definimos o tempo médio colhendo as informações dessas 100 execuções. Além de alterar o número de nós a cada bateria, testamos com os números de arestas de 2 até 5, incrementando em 1 a cada bateria, para os grafos gerados pelo algoritmo de Barabási-Albert. E setamos como $0,25$ o valor do parâmetro p, a probabilidade da inserção de uma nova aresta, para os grafos gerados pelo algoritmo GNP. Os resultados podem ser vistos nos gráficos.

3.3.1 Comparação entre Grafos Barábasi-Albert

Como podemos ver na Figura 2, comparando os tempos médios executados entre nosso algoritmo de centralidade de Katz e o do networkX a partir de grafos gerados pelo Barabási-Albert, podemos identificar que em média o algoritmo do networkX é 120% mais rápido que o nosso algoritmo, tendendo a ser mais rápido ao aumentar o número de nós do grafo gerado na maioria dos casos, independente do aumento de número de arestas passadas como parâmetro.

Além disso, podemos perceber que apesar do aumento de número de arestas passado como parâmetro na geração dos grafos de Barabási-Albert, os tempos de execução são muito similares comparando os tempos médios de cada algoritmo entre as figuras 2, 3, 4 e 5, logo podemos concluir que no escopo de testes realizados nesse trabalho, o aumento do número de arestas passado por parâmetro no algoritmo de Barabási-Albert não afeta o tempo de execução dos algoritmos de centralidade de Katz testados.

3.3.2 Comparação entre Grafos GNP

Comparando os tempos médios executados entre nosso algoritmo de centralidade de Katz e o do networkX a partir de grafos gerados pelo GNP, podemos identificar que em média o algoritmo do networkX é 25% mais rápido que o nosso, e assim como nos testes com grafos Barabási-Albert, tende a ser cada vez mais rápido ao aumentar o número de nós do grafo gerado na maioria dos casos. Como vemos a seguir:

3.3.3 Comparação entre Grafos Barabási-Albert e GNP

Na figura 8 comparamos os resultados obtidos ao testar grafos Barábasi-Albert com os obtidos ao testar grafos GNP igualando o número de nós, a primeira comparação que vemos é que ambos algoritmos levam mais tempo para calcular a centralidade de Katz em grafos GNP do que em grafos Barábasi-Albert.

Ao comparar apenas o tempo médio de execução do nosso algoritmo entre os dois tipos de Grafos, podemos observar que o nosso algoritmo performa em média cerca de 3,6 vezes melhor ao calcular a centralidade de Katz em grafos Barabási-Albert que em grafos GNP. Já quando falamos do algoritmo do networkX, ele performa em média 6,25 vezes melhor em grafos Barabási-Albert que em Grafos GNP.

3.3.4 Complexidade e Tempo de Execução

Algo interessante notar ao analisar as tabelas a seguir acompanhadas das figuras 6 e 7, é que a complexidade $O(n^3)$, onde n é a ordem da matriz de ambos os algoritmos, é confirmada analisando os resultados de tempo de execução dos algoritmos, independente de qual grafo estamos falando. Digamos que a proporção entre o aumento de número de nós é Pn, e a proporção do aumento do tempo médio dos algoritmos de centralidade de Katz é Ptm, em toda a bateria de testes que fizemos o Ptm é sempre limitado superiormente por Pn^3, nunca ultrapassando esse

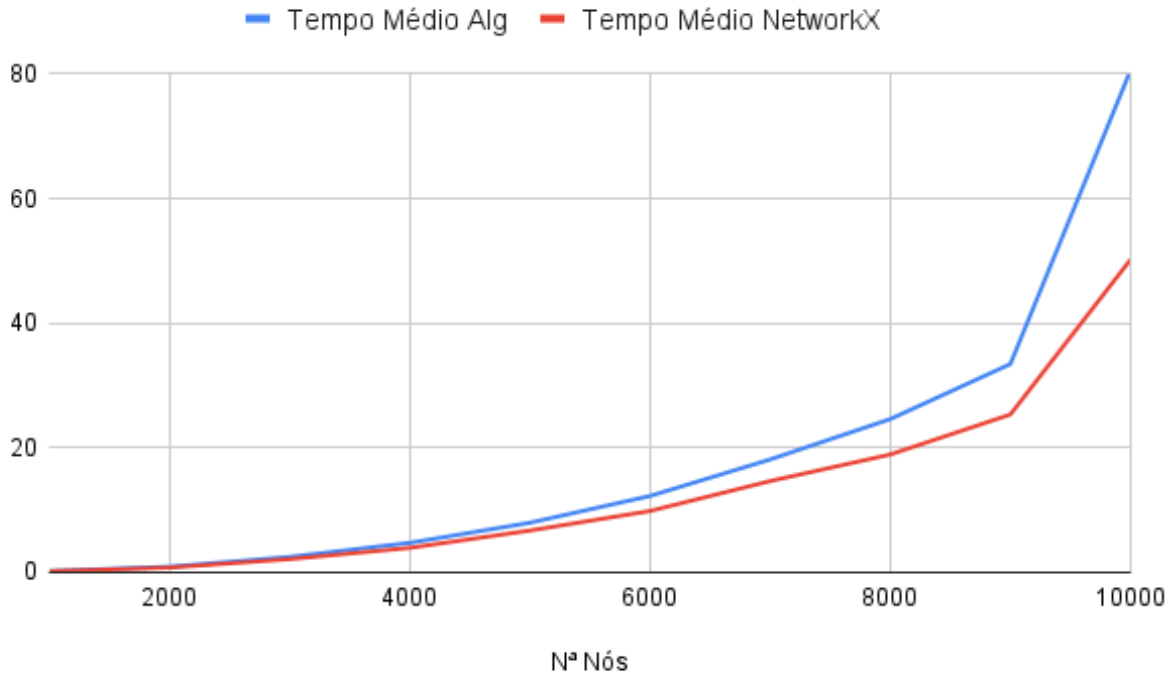


Figura 3.1: Tempo Médio em Segs de Grafos GNP c/ p=0.25.

valor de Pn^3. Além disso, a diferença entre Pn^3 e Ptm parece tender a aumentar ao incremento do número de nós de cada grafo.

Por exemplo, ao comparar os testes do gráfico 1, de 1000 nós com 2000 nós, temos: $Pn = 2000/1000 = 2$, e $Ptm = 0,2334/0,0371 = 6,2911$. Sendo $Pn^3 = 2^3 = 8 > 6,2911$, logo $Pn^3 > Ptm$.

Tabela 3.1: Proporção de Aumento de Nós com Aumento de TM p/ Grafos Barabási.

Pn	**Pn^3**	**Ptm Alg**	**Ptm NetworkX**
1	1	1	1
2	8	6,2890	7,4890
3	27	18,7932	21,2189
4	64	37,9884	40,7091
5	125	60,3581	58,0749
6	216	103,1042	89,7075
7	343	148,7221	130,0460
8	512	218,9245	199,9885
9	729	292,4327	257,2111
10	1000	384,6897	385,2846

Tabela 3.2: Proporção de Aumento de Nós com Aumento de TM p/ Grafos GNP.

Pn	**Pn^3**	**Ptm Alg**	**Ptm NetworkX**
1	1	1	1
2	8	4,4371	4,3912
3	27	12,1835	12,1878
4	64	23,3048	22,5866
5	125	39,1850	38,5749
6	216	60,3432	56,6315
7	343	89,2382	84,2883
8	512	121,1657	108,8434
9	729	164,7270	145,7928
10	1000	397,0117	288,9267

4 CONCLUSÃO

Após analisarmos os resultados deste estudo, podemos afirmar que tanto o algoritmo de Centralidade de Katz desenvolvido por nós, fundamentado no Teorema (Estrada e Knight, 2015b), quanto o implementado no NetworkX, demonstraram ser significativamente mais rápidos ao calcular a centralidade em grafos de grande escala, especialmente os gerados pelo modelo Barabási-Albert, que se aproximam mais das redes reais, tornando a identificação dos nós centrais menos desafiadora para ambos os algoritmos. Em contrapartida, grafos mais densos gerados aleatoriamente, como os GNP, apresentam uma distribuição mais uniforme de centralidade de nós, o que acaba dificultando essa identificação.

Observamos que em redes reais e altamente escaláveis, onde há uma variação extrema no número de conexões dos nós, os grafos Barabási-Albert permitem que os algoritmos de centralidade identifiquem mais facilmente os nós mais centrais. Isso se deve à presença de alguns nós altamente conectados e muitos nós com poucas conexões, facilitando o cálculo e a identificação da centralidade.

Além disso, nosso algoritmo, embora simples e baseado em uma fórmula teórica direta, conseguiu competir em tempo de execução com o robusto algoritmo numpy do NetworkX, especialmente em grafos densos. Por outro lado, o algoritmo do NetworkX mostrou-se mais eficiente ao calcular a centralidade em grafos Barabási-Albert, superando nosso método em termos de desempenho na maioria dos casos, sendo cerca de 2 vezes mais veloz.

Em resumo, a escolha do algoritmo de centralidade de Katz mais adequado depende da natureza específica do grafo em questão, sendo essencial considerar a estrutura e a densidade do grafo para otimizar o desempenho computacional e a precisão na identificação dos nós centrais.

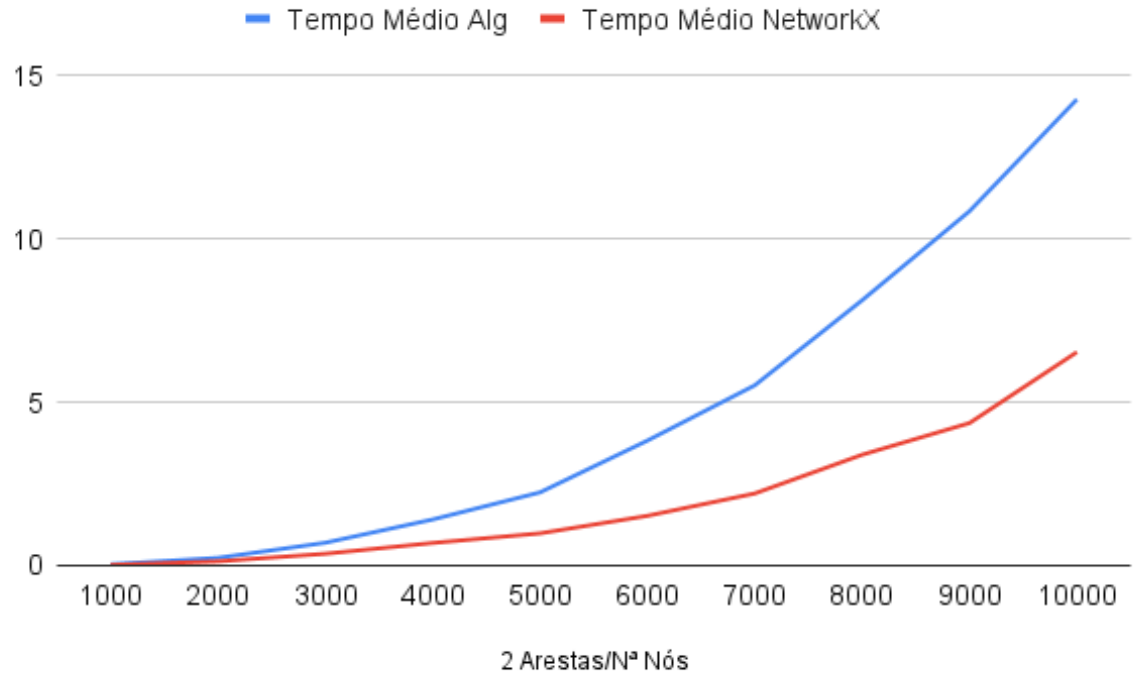


Figura 4.1: Tempo Médio em Segs de Grafos Barabási-Albert c/ Edge=2.

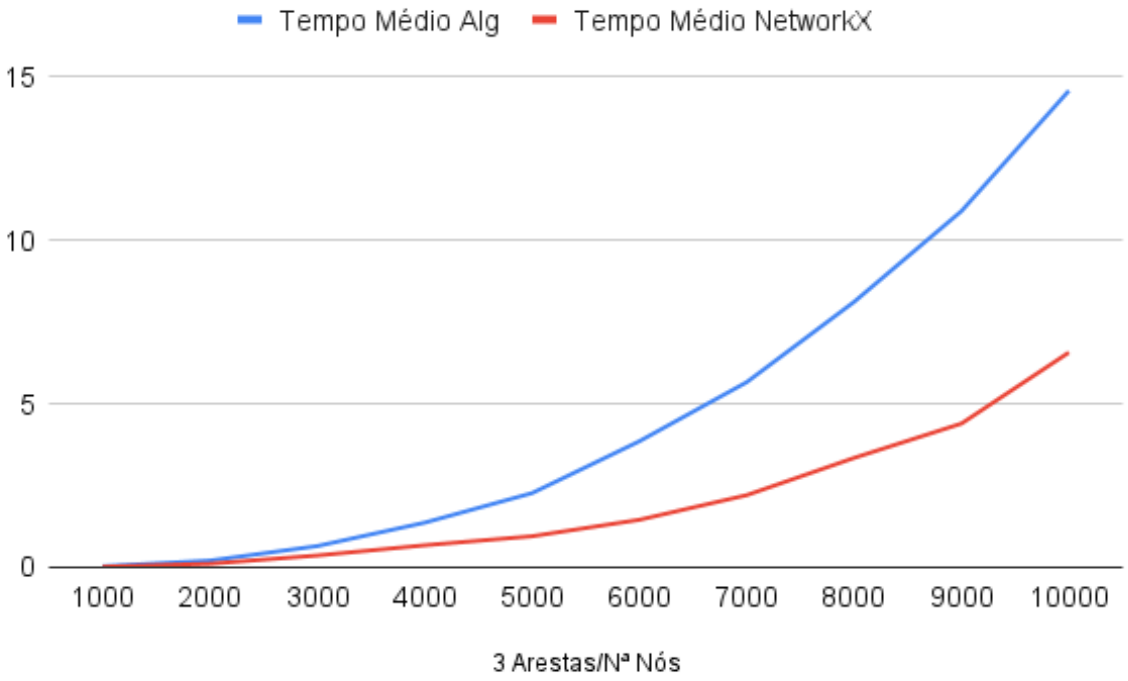


Figura 4.2: Tempo Médio em Segs de Grafos Barabási-Albert c/ Edge=3.

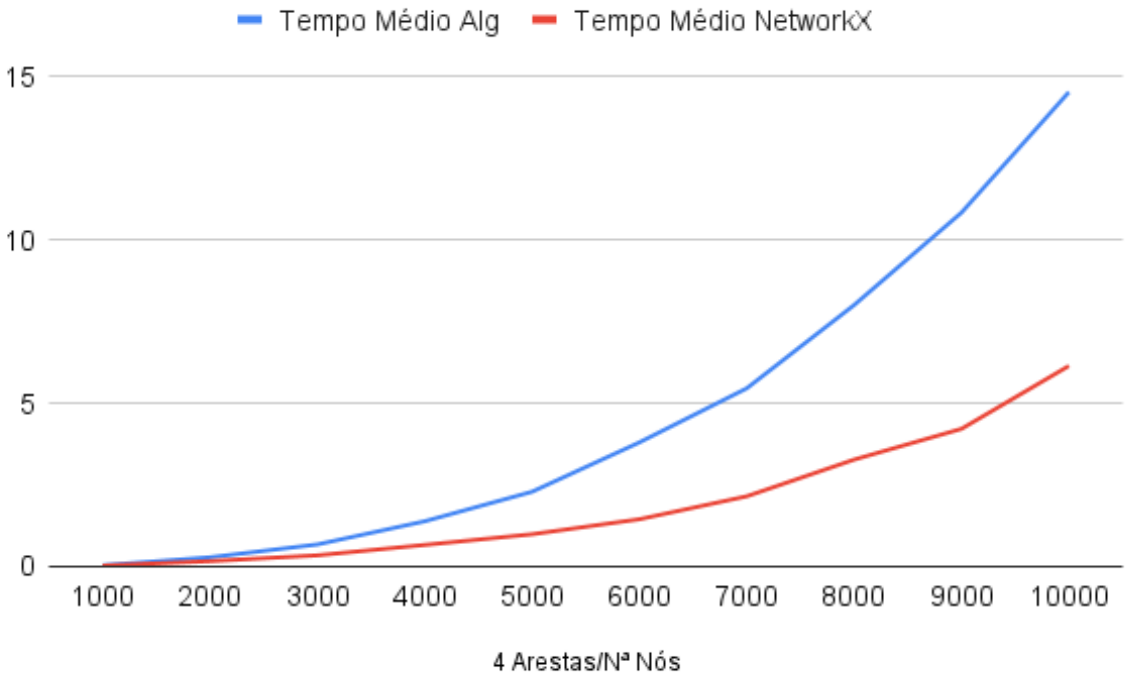


Figura 4.3: Tempo Médio em Segs de Grafos Barabási-Albert c/ Edge=4.

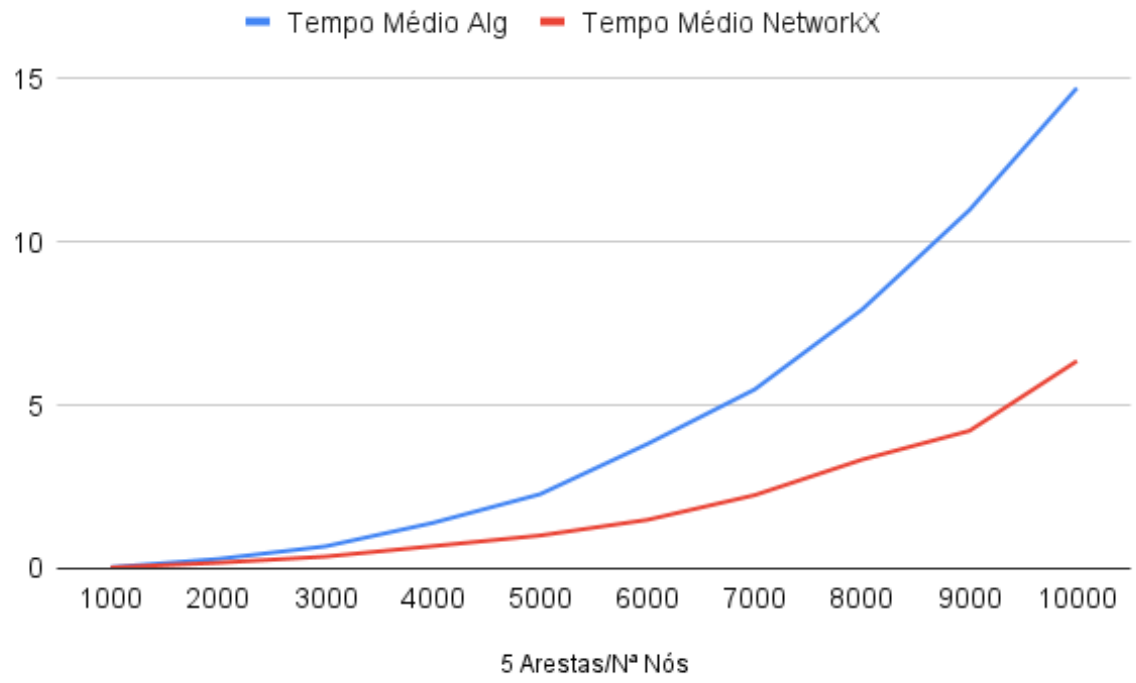


Figura 4.4: Tempo Médio em Segs de Grafos Barabási-Albert c/ Edge=5.

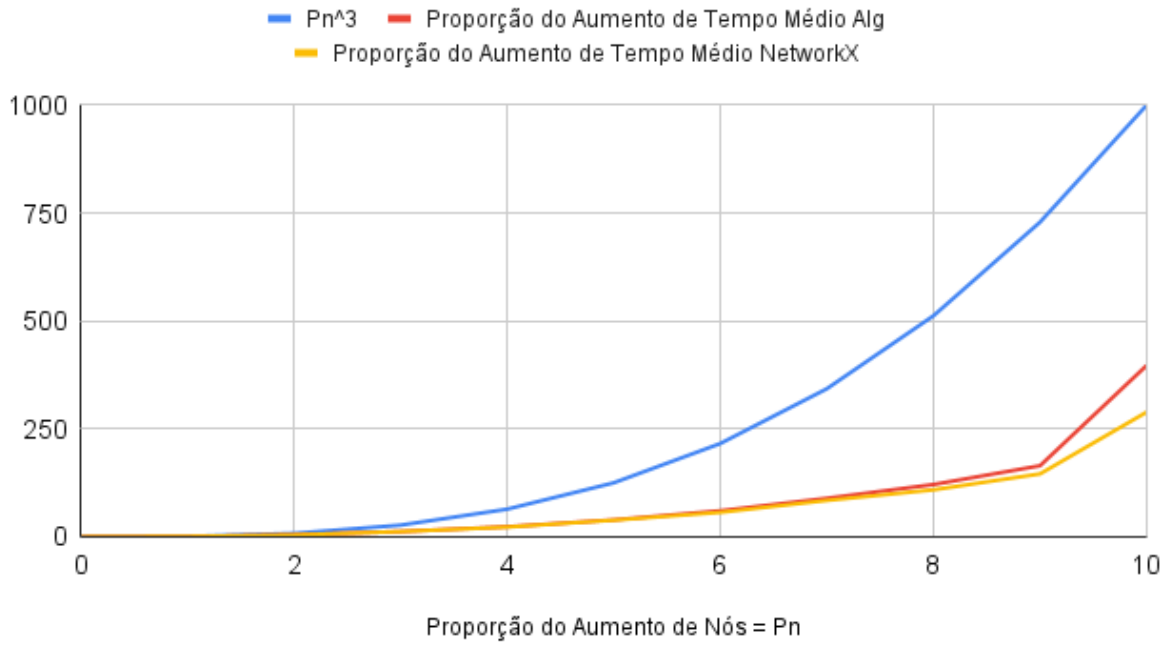


Figura 4.5: Proporção de Aumento de Nós com Aumento de TM p/ Grafos Barabási.

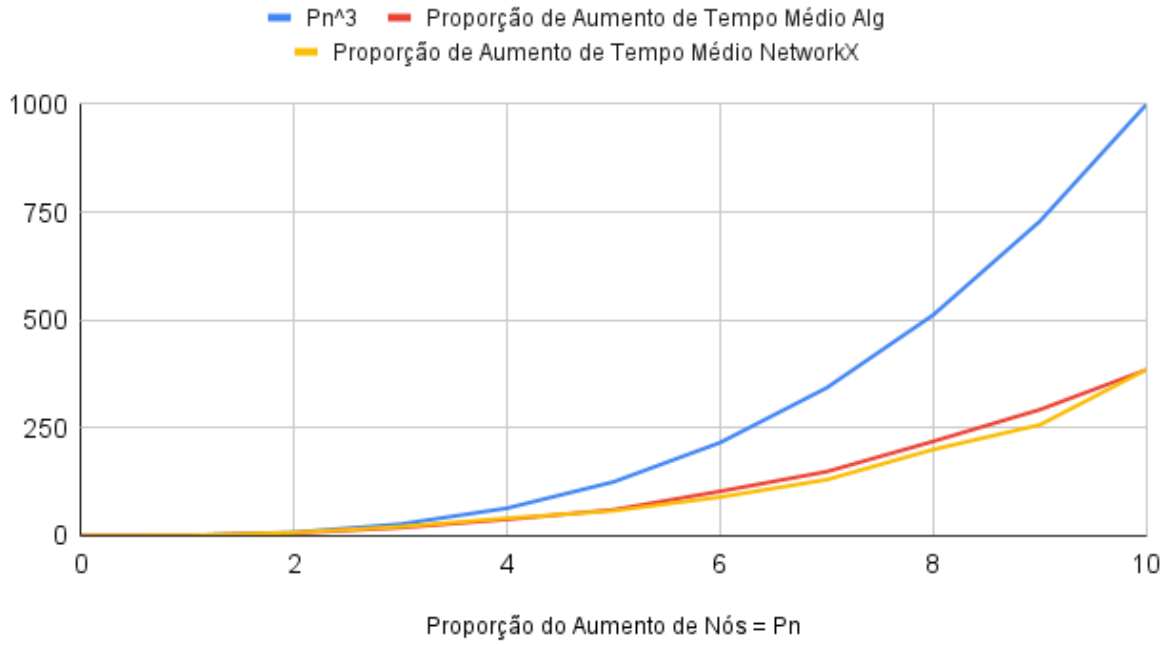


Figura 4.6: Proporção de Aumento de Nós com Aumento de TM p/ Grafos GNP.

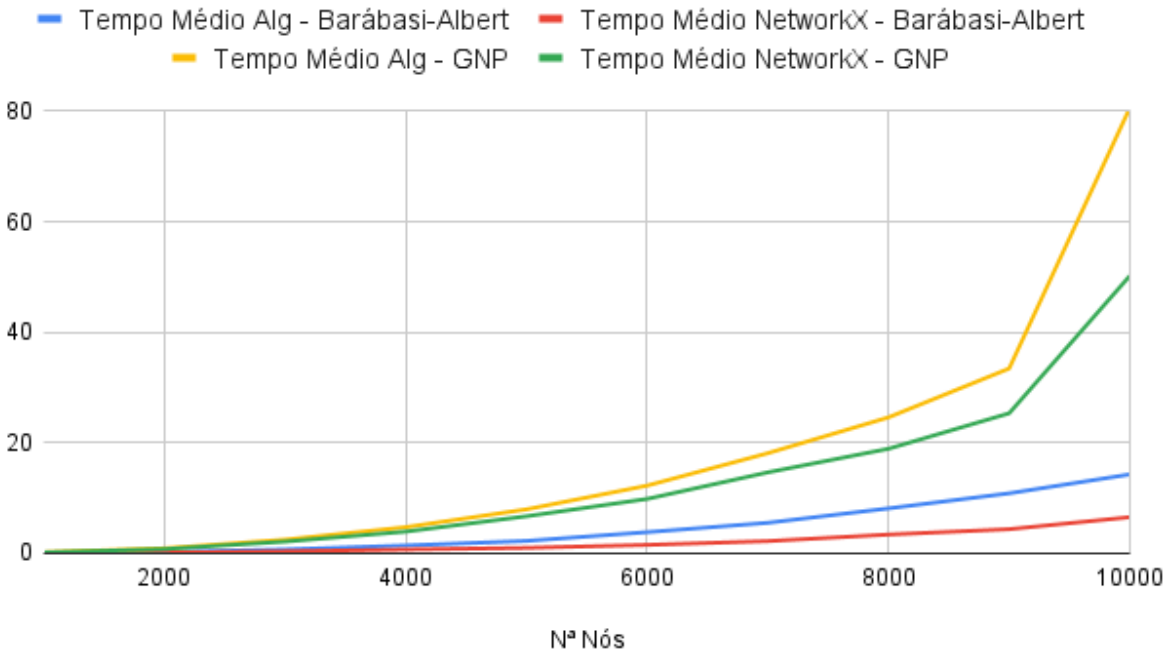


Figura 4.7: TM Comparando os Algoritmos para ambos os Grafos.

REFERÊNCIAS

Anton, H. e Rorres, C. (2011a). *Álgebra Linear com Aplicações*, capítulo: 9. Bookman, Porto Alegre, 10 edition.

Anton, H. e Rorres, C. (2011b). *Álgebra Linear com Aplicações*, capítulo: 1. Bookman, Porto Alegre, 10 edition.

Anton, H. e Rorres, C. (2011c). *Álgebra Linear com Aplicações*, capítulo: 5. Bookman, Porto Alegre, 10 edition.

Erdős, P. e Rényi, A. (1959). On random graphs. *Publicationes Mathematicae*, 6:290–297.

Estrada, E. e Knight, P. A. (2015a). *A First Course in Network Theory*, capítulo: 12. Oxford University Press, Oxford.

Estrada, E. e Knight, P. A. (2015b). *A First Course in Network Theory*, capítulo: 15. Oxford University Press, Oxford.

Iezzi, G. (1977). *Fundamentos de Matemática Elementar*, capítulo: 4. São Paulo.

NETWORKX. barabasi_albert_graph. `https://networkx.org/documentation/stable/reference/generated/networkx.generators.random_graphs.barabasi_albert_graph.html`. Acesso em: 21 out. 2024.

NETWORKX. gnp_random_graph. `https://networkx.org/documentation/stable/reference/generated/networkx.generators.random_graphs.gnp_random_graph.html`. Acesso em: 21 out. 2024.

NETWORKX. katz_centrality. `https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.katz_centrality.html`. Acesso em: 21 out. 2024.

Sudrastawa, I. P. A., Parwata, A., Saputra, M. W. A., Wirautama, I. G. A. M. e Vergantana, I. W. S. M. (2022). Conceptual and practical review of gaussian elimination and gauss-jordan reduction. *Jurnal Ilmu Komputer Indonesia (JIK)*, 7(2):19–25.

Trefethen, L. N. e III, D. B. (1997). *Numerical Linear Algebra*. SIAM, Philadelphia.

Walt, S. V. D., Colbert, S. C. e Varoquaux, G. (2011). The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30. `https://numpy.org/doc/stable/`.

WIKIPEDIA. Modelo de barabási-albert. `https://pt.wikipedia.org/wiki/Barabási_Albert_model`. Acesso em: 21 out. 2024.

WIKIPEDIA. Networkx. `https://en.wikipedia.org/wiki/NetworkX`. Acesso em 21 out. 2024.