Complexidade Assintótica de Componentes Fortemente Conectados: Uma Análise

Giulliano Lyra Paz[1]

[1]Mestrado Profissional em Engenharia de Software (MPES)
Universidade Federal do Pampa (UNIPAMPA)

giullianopaz.aluno@unipampa.edu.br

Resumo. *Grafos são um dos assuntos mais estudados pela Ciência da Computação. Seus componentes fortemente conectados são cruciais para diversos algoritmos de grafos. Este trabalho tem como objetivos medir os tempos reais de execução dos principais algoritmos para encontrar componentes fortemente conectados e comparar com sua complexidade assintótica $O(V + E)$, além de comparar suas principais implementações. Como metodologia, os algoritmos foram executados para um conjunto crescente de vértices e um conjunto crescente de probabilidades de conexões entre vértices, possibilitando o teste dos algoritmos com uma gama variada de grafos. Como resultado, pôde-se concluir que o tempo de execução linear de $O(V + E)$ confere com os tempos de execução reais.*

1. Introdução

A Teoria dos Grafos é um dos assuntos mais estudados na Ciência da Computação. A abstração que os grafos proporcionam é utilizada nas mais diversas aplicações, como Análise de Redes Sociais, Computação Gráfica e Redes de Computadores, por exemplo [Skiena 2011].

Um dos principais algoritmos para grafos são os que se propõem a encontrar os *componentes conectados* de um grafo. Encontrar *componentes conectados* é o passo principal de vários outros algoritmos de grafos. Por exemplo, *componentes conectados* são utilizados para encontrar grupos de vértices, em algoritmos de *clustering* e para verificar a integridade de grafos [Skiena 2011, Cormen et al. 2009].

Embora *componentes conectados* sejam muito úteis, há ainda uma classe destes algoritmos que é mais desafiador: os algoritmos para encontrar *componentes* ***fortemente*** *conectados*. Este tipo de *componentes* é semelhante ao anterior, porém são encontrados em um tipo diferente de grafos, nos grafos *direcionados*. *Componentes fortemente conectados* são utilizados em algoritmos para GPS, em coletores de lixo em linguagens de programação e em Análise de Redes Sociais [Aho et al. 1983, Skiena 2011].

Estes e outros algoritmos são a parte mais duradora e importante da Ciência da Computação. Com isso, é importante haver técnicas para comparar a eficiência destes algoritmos. A principal técnica é a análise assintótica, a qual utiliza notações, como a *big-O*. Esta notação expressa a complexidade da utilização de memória ou de tempo de execução de um algoritmo no seu pior caso, ou seja, expressa as limitações de um algoritmo [Skiena 2011, Cormen et al. 2009].

Os algoritmos de grafos para encontrar *componentes conectados* e *fortemente conectados* possuem uma complexidade $O(V + E)$, onde V é a quantidade de vértices e E é a quantidade de arestas de um grafo qualquer. Este trabalho tem como objetivo analisar e verificar esta complexidade, executando os principais algoritmos para os casos em questão e verificando seus tempos de execução reais. Este trabalho optou por focar esforços nos algoritmos para encontrar *componentes fortemente conectados* por este ser mais desafiador [Skiena 2011, Cormen et al. 2009].

Este trabalho está organizado da seguinte forma: na seção 2 são apresentados termos e assuntos pertinentes ao escopo do trabalho; na seção 3 é apresentado como o trabalho foi desenvolvido; na seção 4 são apresentados os resultados; e na seção 5 são feitas as considerações finais.

2. Referencial Teórico

Nesta seção serão apresentados os conceitos e auxílio teórico para uma melhor compreensão deste trabalho. Na subseção 2.1 é apresentado o conceito e principais aplicações dos grafos; na subseção 2.2 são apresentados os *componentes conectados*; e na subseção 2.3 são apresentados os *componentes fortemente conectados*.

2.1. Grafos

A Teoria dos Grafos é um dos assuntos mais estudados pela Ciência da Computação. A abstração que os grafos proporcionam é utilizada para resolver os mais variados problemas computacionais, sendo largamente utilizados em Análises de Redes Sociais, em Computação Gráfica e em Redes de Computadores, apenas para citar alguns [Skiena 2011].

Grafos são estruturas de dados constituídas de vértices e arestas. Vértices são "nós" que representam um objeto ou um agente, enquanto que as arestas representam os relacionamentos entre os vértices. Grafos podem ser representados de forma visual, utilizando listas de adjacências, como matrizes e utilizando notações matemáticas. Formalmente, um grafo pode ser representado como $G = (V, E)$, onde V é o conjunto de vértices e E é o conjunto de arestas. Concomitantemente, $|V|$ e $|E|$ representam o tamanho do conjunto de vértices e o tamanho do conjunto de arestas, respectivamente [Cormen et al. 2009].

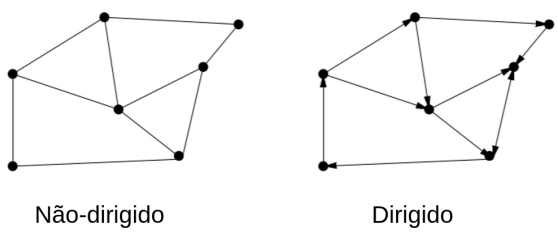

Figura 1. Propriedades dos Grafos

Os grafos possuem várias propriedades, onde algumas são: *ponderado* ou *não-ponderado*; *cíclico* ou *acíclico*; e *incorporado* ou *topológico*. Entretanto, as propriedades ilustradas pela Figura 1 (representação visual dos grafos) são suficientes para o escopo deste trabalho. Grafos *não-dirigidos* representam uma relação mútua entre os vértices,

como uma relação de amizade, por exemplo. Grafos *dirigidos* representam relações unidirecionais, como, por exemplo, a relação entre ruas de um único sentido em um mapa [Skiena 2011].

Um dos mais fundamentais problemas da Teoria dos Grafos é visitar todos os vértices e arestas de uma forma sistemática [Skiena 2011]. Há dois principais algoritmos de percorrimento em grafos: busca em profundidade[1] (DFS *do ingles Depth First Search*) e busca em largura[2] (BFS *do inglês Breadth First Search*). A estratégia do algoritmos DFS é ir o mais "fundo" possível no grafo (*i.e* o mais longe possível). Com isso, DFS explora os vértices que estão nas extremidades do grafo primeiro. O algoritmos BFS, por outro lado, explora o grafo de forma uniforme, visitando todos os vértices vizinhos antes de expandir os demais vértices. Os dois algoritmos são implementados, normalmente, de forma recursiva [Cormen et al. 2009]. Com o auxílio destes dois algoritmos, é possível executar vários outros algoritmos, como: encontrar o menor caminho entre dois vértices; encontrar ciclos em um grafo; ordenação topológica; e – o foco deste trabalho – encontrar componentes conectados e componentes fortemente conectados [Skiena 2011].

2.2. Componentes Conectados

Um grafo *não-dirigido* é dito *conectado* se, e somente se, houver um "caminho" entre todos os vértices do grafo, ou seja, se for possível chegar a qualquer vértice de qualquer outro vértice. Um *componente conectado* (CC *do inglês Connected Component*) de um grafo *não-dirigido* é o maior conjunto de vértices que possuem um caminho entre todos os vértices, ou seja, dado um grafo *não-dirigido* G, um CC é uma "parte" separada (um subgrafo) de G [Skiena 2011]. A Figura 2 ilustra dois CCs de um grafo *não-dirigido* qualquer.

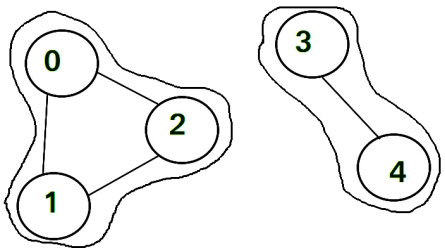

Figura 2. Componentes Conectados

CCs são utilizados quando é necessário encontrar relações entre grupos de vértices. Um exemplo seria encontrar grupos de amigos em uma rede social. Além disso, verificar se um grafo é *conectado* é um passo essencial executado por vários outros algoritmos, visando evitar erros e verificar a integridade do grafo [Skiena 2011]. Esta verificação da-se após a execução de um algoritmo para encontrar CCs. Se o algoritmo retornar apenas um *componente*, significa que o grafo em si é *conectado*. CCs podem ser encontrados utilizando tando DFS quanto BFS, uma vez que a ordem dos vértices não importa [Skiena 2011]. Abaixo encontra-se um algoritmo para encontrar CCs utilizando DFS[3].

[1]Disponível em `http://bit.ly/dfs-for-a-graph`
[2]Disponível em `http://bit.ly/bfs-for-a-graph`
[3]Disponível em `http://bit.ly/cc-in-a-ugraph`

```
Entrada: Um grafo nao-dirigido G = (V, E)
Saida: Lista de Componentes Conectados

[1]     Inicializa todos os vertices de V como nao visitados
[2]     Para cada vertice v de V, faca:
[2.1]       Se v ainda nao foi visitado, entao:
[2.1.1]         Chama DFS(v)
[2.2]        Escreva '\n' na tela

        DFS(v):
[a]         Marca v como visitado
[b]         Mostra v na tela
[c]         Para cada vertice adjacente u de v, faca:
[c.1]           Se u ainda nao foi visitado, entao:
[c.1.1]             Chama recursivamente DFS(u)
```

A complexidade de tempo para encontrar CCs é linear, sendo $O(|V| + |E|)$ [Skiena 2011]. Analisando o algoritmo acima, facilmente nota-se os dois laços, um que percorre os elementos de V**[2]** e outro que percorre a lista de adjacências de cada elemento v de V**[c]**. Cada vértice adjacente u de v representa uma aresta entre estes dois vértices. Em suma, o primeiro laço possui uma complexidade $O(|V|)$ e o segundo laço possui uma complexidade $O(|E|)$, resultando em $O(|V| + |E|)$ [Skiena 2011]. Por convenção, este trabalho tratará $O(|V| + |E|)$ como $O(V + E)$.

2.3. Componentes Fortemente Conectados

Embora CCs sejam muito úteis, o que acontece se o grafo for um grafo *dirigido*? Grafos *dirigidos* têm um tipo diferente de *componentes*, os *componentes* ***fortemente*** *conectados* (SCCs *do inglês Strongly Connected Components*). Um grafo é *fortemente conectado* se este é um grafo *dirigido* e se há um "caminho" *dirigido* entre todos os seus vértices. Ou seja, para cada par de vértices $v \in V$ e $u \in V$, há um "caminho" dirigido de v para u e há um caminho dirigido de u para v [Skiena 2011, Cormen et al. 2009].

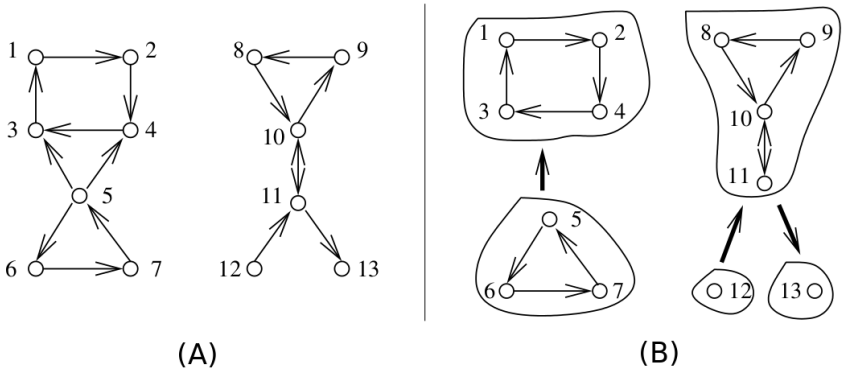

Figura 3. (A) Grafo Dirigido, (B) Componentes Fortemente Conectados

A Figura 3 ilustra graficamente os SCCs. (A) ilustra um grafo *dirigido* com 13 vértices ($|V| = 13$) e (B) ilustra os 5 SCCs. Analisando os *componentes* do lado (B) da Figura 3, pode-se verificar a relação entre seus vértices. Focando no primeiro *componente* (vértices $1, 2, 3$ e 4), há um "caminho" entre todos os vértices em questão. Por exemplo,

há um "caminho" do vértice 1 até o vértice 4 e também há um "caminho" do vértice 4 até o vértice 1.

Encontrar SCCs é relativamente mais complicado do que encontrar CCs. Há dois principais algoritmos para isso: o *algoritmo de Kosaraju* e o *algoritmo de Tarjan* [Skiena 2011, Cormen et al. 2009].

2.3.1. Algoritmo de Kosaraju

O algoritmo mais simples para encontrar SCCs em tempo linear é o *algoritmo de Kosaraju* [Skiena 2011]. O algoritmo utiliza uma pilha auxiliar e duas DFS, além de um grafo reverso extra. O *algoritmo de Kosaraju* é apresentado abaixo.

```
Entrada: Um grafo dirigido G = (V, E)
Saida: Lista de Componentes Fortemente Conectados

[1]     Inicializa todos os vertices de V como nao visitados
[2]     Inicializa pilha vazia L
[3]     Para cada vertice v de V, faca:
[3.1]       Se v ainda nao foi visitado, entao:
[3.1.1]         Chama DFS_Preencha(v)
[4]     Inverta o Grafo G, criando G`
[5]     Marque todos os vertices de V` como nao visitados
[6]     Enquanto houver vertices na pilha L, faca:
[6.1]       Retire v` de L
[6.2]       Se v` ainda nao foi visitado, entao:
[6.2.1]         Chama DFS_Mostra(v`)
[6.2.2]         Mostra '\n' na tela

        DFS_Preencha(v):
[a]         Marca v como visitado
[b]         Para cada vertice adjacente u de v, faca:
[b.1]           Se u ainda nao foi visitado, entao:
[b.1.1]             Chama recursivamente DFS_Preencha(u)
[c]         Adiciona v na pilha L

        DFS_Mostra(v):
[a]         Marca v como visitado
[b]         Mostra v na tela
[c]         Para cada vertice adjacente u de v, faca:
[c.1]           Se u ainda nao foi visitado, entao:
[c.1.1]             Chama recursivamente DFS_Mostra(u)
```

Suponto um grafo $G = (V, E)$, o *algoritmo de Kosaraju* realiza uma DFS em G, preenchendo uma pilha L. Cada vértice *v* é adicionado à pilha L cada vez que este não tem mais vértices adjacentes para expandir. Ao fim do percorrimento de G, é criado um grafo G', onde $G' = G^T$ (*i.e* o grafo transposto de G). Por fim, os vértices são retirados da pilha L, executando outra DFS para cada um destes vértices, mostrando os SCCs [Skiena 2011, Cormen et al. 2009, Aho et al. 1983].

Analisando o algoritmo apresentando acima, é possível verificar que sua complexidade resulta de: $O(V)$**[3]** + $O(E)$**[3.1.1]** + $O(L)$**[6]** + $O(E)$**[6.2.2]**. Neste caso,

desconsidera-se passos secundários, como **[1]**, **[4]** e **[5]**. Como $O(L)$ é igual a $O(V)$, podemos simplificar a complexidade para $O(2V) + O(2E)$. Como a multiplicação por um escalar não afeta a complexidade final, podemos simplificar mais ainda para $O(V + E)$. Por fim, podemos concluir que o *algoritmo de Kosaraju* executa em uma complexidade de tempo $O(V + E)$, assim como o algoritmo para encontrar CCs [Skiena 2011, Cormen et al. 2009, Aho et al. 1983].

2.3.2. Algoritmo de Tarjan

Embora o *algoritmo de Kosaraju* resolva bem o problema de encontrar SSCs, ele utiliza duas DFSs, além da pilha e do grafo extra. O *algoritmo de Tarjan* [Tarjan 1971] foi o primeiro criado para encontrar SCCs em um tempo linear e é implementado utilizando apenas uma DFS [Skiena 2011]. A biblioteca *Boost Graph Library* da linguagem de programação *C++* utiliza o *algoritmo de Tarjan* em sua implementação para encontrar SCCs[4]. Abaixo segue o pseudocódigo do algoritmo.

```
    Entrada: Um grafo dirigido G = (V, E)
    Saida: Lista de Componentes Fortemente Conectados

    [1]     Inicializa todos os vertices v de V como nao visitados
    [2]     Inicializa pilha vazia L
    [3]     Inicializa valores T de primeira visita de cada vertice v
    [4]     Inicializa valores minimos M de cada vertice v
    [5]     Inicializa indice I
    [6]     Para cada vertice v de V, faca:
    [6.1]       Se v ainda nao foi visitado, entao:
    [6.1.1]         Chama DFS(v)

            DFS(v):
    [a]         Marca v como visitado
    [b]         Faz v.T := I
    [c]         Faz v.M := I
    [d]         Incrementa I
    [e]         Adiciona v na pilha L
    [f]         Para cada vertice adjacente u de v, faca:
    [f.1]           Se u ainda nao foi visitado, entao:
    [f.1.1]             Chama recursivamente DFS(u)
    [f.1.2]             Faz v.M := Min(v.M, u.M)
    [f.2]           Senao Se u estiver na pilha L, entao:
    [f.2.1]             Faz v.M := Min(v.M, u.T)
    [g]         Se v.M == v.T, entao:
    [g.1]           Repita:
    [g.1.1]             Retite w da pilha L
    [g.1.2]             Mostre w na tela
                    Enquando w != v
    [g.2]           Mostre '\n' na tela
```

No *algoritmo de Tarjan* cada vértice possui dois índices, um que registra a ordem que o vértice foi visitado pela primeira vez (T) e o índice que registra o menor índice de

[4]Disponível em `http://bit.ly/scc-boost-graph-library`

visita dentre seus componentes (M). Além disso, o algoritmos utiliza uma pilha (L) para ir armazenando vértices que já foram visitados e só retira estes vértices da pilha quando o *componente* for definido. A Figura 4 ilustra o resultado final da execução do *algoritmo de Tarjan*, iniciando no vértice 1 e finalizando no vértice 2 , onde vértices que pertencem ao mesmo *componente fortemente conectado* terminam a execução do algoritmo com o mesmo valor M.

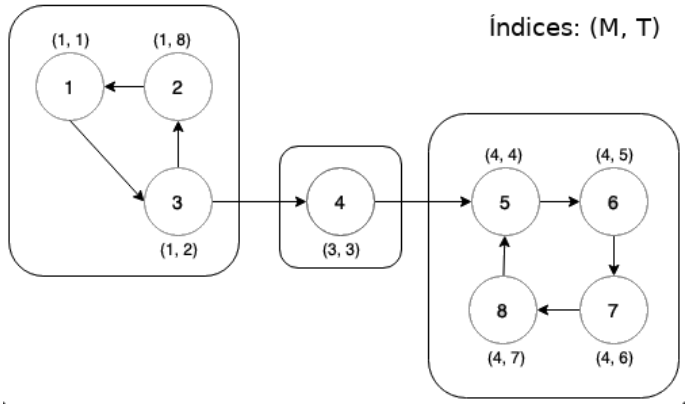

Figura 4. Exemplo da Execução do Algoritmo de Tarjan

A complexidade do *algoritmo de Tarjan* é a mesma dos demais apresentados anteriormente, $O(V + E)$ [Skiena 2011, Cormen et al. 2009, Aho et al. 1983]. Entretanto, este algoritmo é ligeiramente mais eficiente do que o *algoritmo de Kosaraju*, necessitando de apenas uma execução da DFS. A diferença entre estes dois algoritmos ficarão mais evidentes ao fim deste trabalho.

3. Desenvolvimento

Nesta seção será abordado o processo de desenvolvimento deste trabalho, como a metodologia (subseção 3.1) e as implementações (subseção 3.2).

3.1. Metodologia

Como metodologia escolhida, optou-se por encontrar algoritmos que resolvam o problema de SCCs e, então, executá-los, mensurando seu tempo de execução para verificar se sua complexidade temporal $O(V + E)$ mantém-se. Os dois principais algoritmos para encontrar SCCs são o *algoritmo de Kosaraju* e o *algoritmo de Tarjan* [Skiena 2011, Cormen et al. 2009]. Seguindo o conselho de [Skiena 2011], o algoritmo da biblioteca *Boost Graph* da linguagem de programação *C++* foi escolhido. Este algoritmo implementa sua versão do *algoritmo de Tarjan*. Além deste, duas outras implementações foram utilizadas: uma do *algoritmo de Tarjan*[5] e outra do *algoritmo de Kosaraju*[6]. Estas duas implementações são do site GeeksforGeeks[7], site que possui várias implementações dos mais variados algoritmos da Ciência da Computação.

Todos estes algoritmos recebem como entrada um grafo *dirigido* e retornam uma lista de *componentes fortemente conectados*. Para a criação do conjunto de testes foi

[5]Disponível em `http://bit.ly/scc-geeksforgeeks-tarjan`
[6]Disponível em `http://bit.ly/scc-geeksforgeeks-kosaraju`
[7]Disponível em `https://www.geeksforgeeks.org`

utilizada a seguinte estratégia:

```
[1]     Inicializa quantidade de testes nT
[2]     Inicializa conjunto de tamanhos de vertices W
[3]     Inicializa conjunto de probabilidades P
[4]     Enquanto t < nT, faca:
[5]         Para cada p de P, faca:
[6]             Para cada w de W, faca:
[7]                 Inicializa grafo G = (V, E)
[8]                 Inicializa |V| := w
[9]                 Para cada v1 de V, faca:
[10]                    Para cada v2 de V, faca:
[11]                        Se Rand(0.0, 1.0) < p, faca:
[12]                            Add_Aresta(v1, v2, G)
[13]                Inicializa contador de tempo
[14]                Encontra_SCCs(G)
[15]                Finaliza contador de tempo
[16]                Escreve tempo de execucao no arquivo
[17]        Incrementa t
```

Inicialmente é definida a quantidade de testes que serão feitos (amostras), sendo representada por nT **[1]**. Em **[2]** é inicializado o conjunto de quantidades de vértices, por exemplo, $W = \{1k, 2k, ..., 10k\}$. Então, em **[3]** é inicializado o conjunto de probabilidades P, por exemplo, $P = \{0.1, 0.2, ..., 1.0\}$. Os algoritmos são executado para cada $w \in W$, para testá-los com grafos de diferentes quantidades de vértices. Isto é repetido para cada $p \in P$, com isso, para cada quantidade de vértices, há uma probabilidade p de haver arestas entre cada par de vértice. Em suma, cada algoritmo é testado com diferentes quantidades de vértices e com diferentes quantidades de arestas. Por fim, estes passos são executados nT vezes, visando amenizar possíveis *pontos fora da curva*. Ao fim dos testes, são utilizadas as seguintes fórmulas:

$$\overline{x} = \frac{\sum_{i=1}^{nT}(x_i)}{nT} \tag{1}$$

$$s = \sqrt{\frac{\sum_{i=1}^{nT}(x_i - \overline{x})^2}{nT - 1}} \tag{2}$$

onde $\overline{x}$ (Equação 1) é a média aritmética simples e s (Equação 2) é o desvio padrão dos nT testes. Após os cálculos, os resultados são mostrados em forma de gráficos, os quais serão apresentados na seção de resultados.

3.2. Implementações

Em seu livro, Skiena [Skiena 2011] cita algumas implementações de algoritmos para estruturas de dados em grafos. Como opção principal, é citada a biblioteca *C++ Boost Graph Library*, a qual possui vários algoritmos para grafos escritos em *C/C++*. Este trabalho optou por esta implementação devido à familiaridade com as linguagens de programação *C* e *C++*. Sendo o objetivo deste trabalho a realização de uma análise da complexidade assintótica de algoritmos para SCCs e não suas implementações em diferentes linguagens de programação, a escolha da biblioteca não é restritiva. Entretanto,

é possível acessar todas as implementações de algoritmos para grafos citadas pelo autor em sua página web[8].

Durante buscas por implementações do *algoritmos de Kosaraju*, foi encontrado o site *GeeksforGeeks*, um portal que possui diversas implementações dos mais variados problemas da Ciência da Computação. Neste portal, foram encontradas implementações tanto do *algoritmos de Kosaraju*, como do *algoritmos de Tarjan*. Assim, para manter um padrão e realizar uma rápida comparação entre implementações, foram definidas três implementações para realizar os testes: uma do *algoritmos de Kosaraju* do portal GeeksforGeeks; e duas implementações do *algoritmos de Tarjan*, sendo uma do portal GeeksforGeeks e outra da biblioteca *Boost Graph Library*.

Enquanto que a implementação da biblioteca *Boost Graph Library* manteve-se intacta, as implementações do portal *GeeksforGeeks* foram ligeiramente modificadas. Foi necessário implementar o *destrutor* da classe *Graph*, a qual não estava liberando memória corretamente. Além disso, foram implementados testes para verificar a *corretude funcional* das implementações, ou seja, para verificar se as implementações, de fato, resultam no resultado correto. Para a criação dos gráficos, foi utilizada a biblioteca *Matplotlib*[9] da linguagem de programação *Python* e, para a execução de teste automatizados, foi utilizado o interpretador de comandos *Bash*[10]. As implementações, testes e demais artefatos podem ser acessados no GitHub[11] do autor deste trabalho.

Tabela 1. Ambiente de Testes

Item	**Configuração**
Máquina	Notebook Acer Aspire ES1-572-51NJ
CPU	Intel Core i5-7200U
Quantidade de Núcleos	4
CPU Cache L1	32 KB
CPU Cache L2	256 KB
CPU Cache L3	3 MB
Clock da CPU	2.5 GHz (Turbo de 3.1 GHz)
Memória RAM	12 GB DDR4
Sistema Operacional	Ubuntu 18.04.01 - 64 Bits
Kernel Linux	4.15.0
Compilador *C/C++*	*G++* 7.4.0
Interpretador *Python*	*Python* 3.6.8 - GCC 8.3.0
Shell Scripts	GNU Bash 4.4.20

4. Resultados

Nesta seção, serão apresentados os resultados dos testes realizados, visando analisar a complexidade assintótica dos algoritmos em questão. O ambiente de testes utilizado é apresentado na Tabela 1. Cada implementação foi executada com $nT = 10$, $W = \{1k, 2k, ..., 10k\}$ e $P = \{0.1, 0.2, ..., 1.0\}$. Com isso, cada implementação foi executada $nT \times |W| \times |P|$ vezes, ou seja, 10^3 vezes. Os gráficos foram gerados utilizando

[8]Disponível em `http://bit.ly/cc-skiena`
[9]Disponível em `https://matplotlib.org/`
[10]Disponível em `https://www.gnu.org/software/bash/`
[11]Disponível em `http://bit.ly/scc-complexity-analysis`

a biblioteca *matplotlib* da linguagem de programação *Python*. Após a execução dos algoritmos utilizando as implementações em *C/C++*, os resultados foram sendo salvos em arquivos de texto, os quais foram utilizados posteriormente na geração do gráficos. Cada ponto nos gráficos refletem a média aritmética simples das nT amostras (Equação 1) e as barras verticais refletem os desvios padrão das amostras (Equação 2).

A seção está organizada da seguinte forma: na subseção 4.1 é apresentada a comparação entre as implementações utilizadas; na subseção 4.2 são apresentados os resultados do *algoritmo de Kosaraju* utilizando a implementação do portal *GeeksforGeeks*; na subseção 4.3 são apresentados os resultados do *algoritmo de Tarjan* utilizando a biblioteca *Boost Graph Library*; e, por fim, na subseção 4.4 são apresentados os resultados da implementação do portal *GeeksforGeeks* para o *algoritmo de Tarjan*.

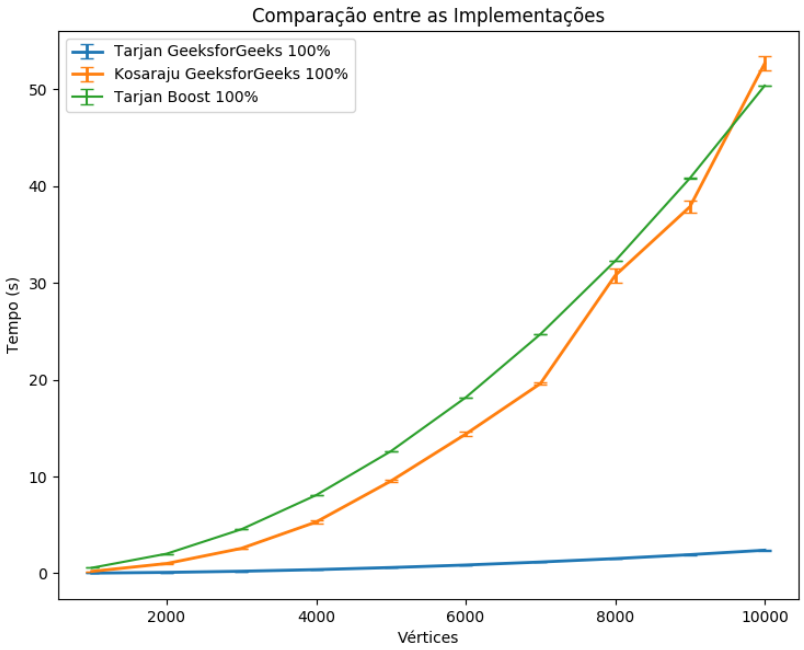

Figura 5. Comparação entre as Implementações

4.1. Comparação entre as Implementações

Dentre as três implementações utilizadas, a implementação do *algoritmo de Tarjan* do portal *GeeksforGeeks* mostrou-se consideravelmente melhor do que as demais (Figura 5). Embora os algoritmos possuam uma mesma complexidade linear $O(V + E)$, os resultados mostram uma diferença notável nos tempos de execução. Atendo-se à diferença entre as duas diferentes implementações do *algoritmo de Tarjan*, o tempo de execução da implementação do *GeeksforGeeks* mostrou-se, em média, 20 vezes menor do que a implementação da biblioteca *Boost Graph Library*. Inicialmente foram levantadas hipóteses sobre uma possível deficiência na implementação do *GeeksforGeeks*, porém foram executados testes comparando os resultados e quantidades de *componentes conectados* encontrados pelas duas implementações do *algoritmos de Tarjan*; nenhuma diferença nos resultados foi encontrada. A única diferença entre as duas implementações, contudo, é o tempo de execução. Nas próximas subseções, serão apresentados os resultados de execuções para cada implementação.

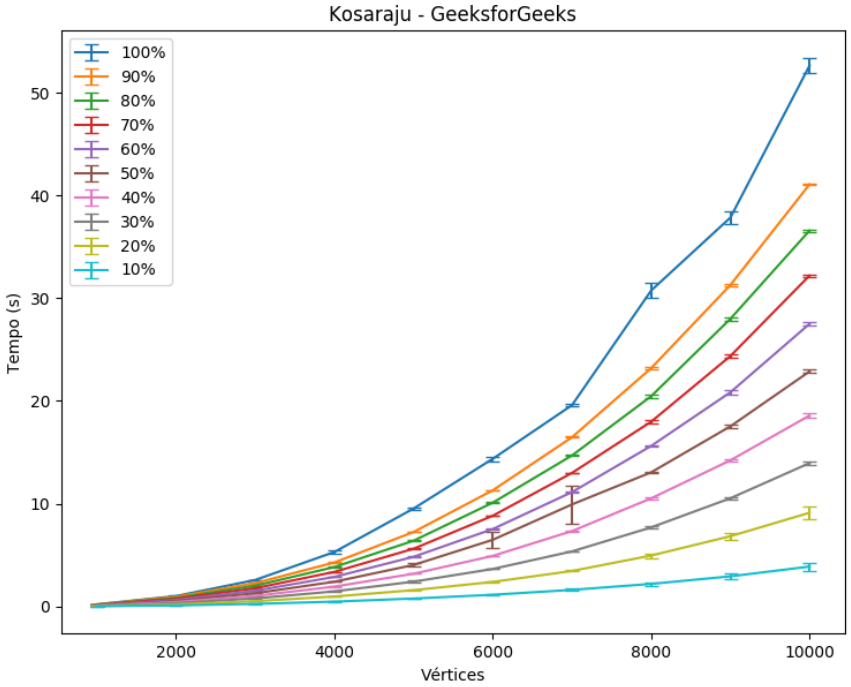

Figura 6. Algoritmo de Kosaraju - GeeksforGeeks

4.2. Algoritmo de Kosaraju do GeeksforGeeks

A Figura 6 mostra os resultados obtidos da implementação do *algoritmo de Kosaraju* do portal *GeeksforGeeks*. Cada linha do gráfico representa uma porcentagem de arestas (i.e. quanto maior a porcentagem, maior a quantidade de arestas), como pode ser visto nas legendas. Para 10000 vértices, o tempo de execução variou de poucos segundos para 50 segundos no tempo de execução. Com isso, é possível argumentar que quanto maior a quantidade de arestas, maior o tempo de execução. Ou seja, a variável mais significante no tempo de execução do algoritmo é a quantidade de arestas.

4.3. Algoritmo de Tarjan da Boost Graph Library

Como já foi visto na Figura 5, a implementação do *algoritmo de Tarjan* da *Boost Graph Library* (Figura 7) teve um desempenho muito semelhante à implementação do *algoritmo de Kosaraju* do *GeeksforGeeks* (Figura 6), inclusive nos tempo máximos e mínimos. Entretanto, por definição, o *algoritmo de Tarjan* é melhor do que o de *Kosaraju*. Além disso, a implementação do *algoritmo de Tarjan* da *Boost Graph Library* apresentou um desvio padrão elevado em algumas amostras nos seus resultados, mostrando que a implementação pode variar bastante, dependendo do grafo que recebe como entrada.

4.4. Algoritmo de Tarjan do GeeksforGeeks

A implementação que se mostrou melhor nos resultados foi a implementação do portal *GeeksforGeeks* do *algoritmo de Tarjan* (Figura 8). Como mostra a Figura 5, a diferença no tempo de execução chegou a ser 20 vezes menor do que as demais implementações. Seus resultados variaram de milésimos de segundos até 2,5 segundos, em média. Analisando o gráfico, aparenta que a implementação tem um valor elevado nos seus desvios padrão. Contudo, isso se deve à escala dos gráficos. Uma análise mais justa pode ser feita na Figura 5, onde as três implementações são colocadas em uma mesma escala.

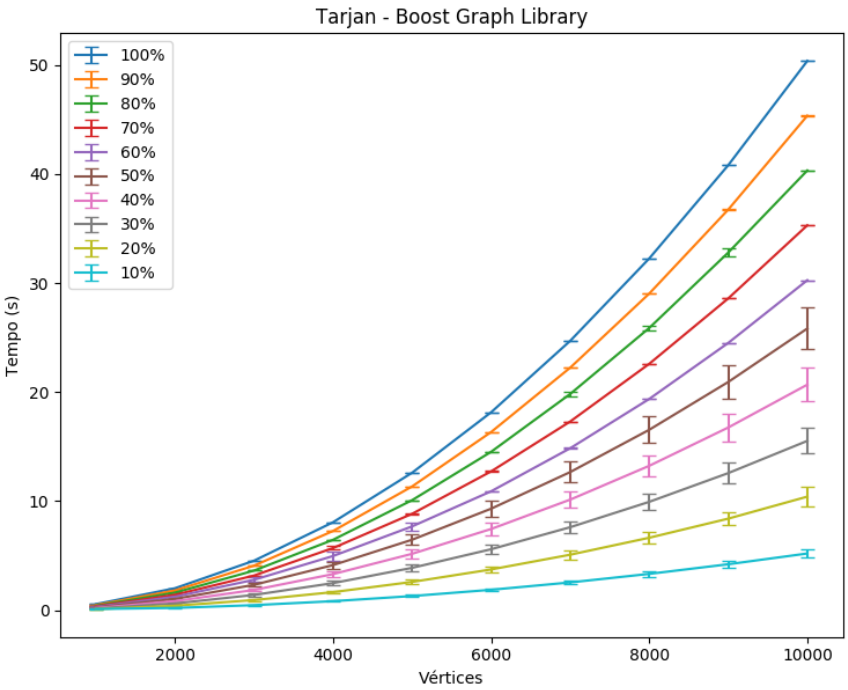

Figura 7. Algoritmo de Tarjan - Boost Graph Library

5. Conclusão

Neste trabalho foram realizadas execuções dos principais algoritmos para encontrar *componentes fortemente conectados*, visando verificar suas complexidades assintóticas e fazer uma breve comparação entre suas implementações. Foram escolhidos dois algoritmos: o *algoritmos de Kosaraju* e o *algoritmo de Tarjan*, e três implementações: a implementação do *algoritmo de Tarjan* da biblioteca *C++ Boost Graph Library*; a implementação do *algoritmo de Tarjan* do portal *GeeksforGeeks*; e a implementação do *algoritmos de Kosaraju*, também do portal *GeeksforGeeks*.

Os resultados mostraram que os tempos de execução reais conferem com a complexidade assintótica $O(V + E)$. Contudo, a comparação das implementações mostrou uma diferença considerável entre duas implementações do mesmo algoritmo. Como esta análise excede o escopo deste trabalho, está diferença será analisada em trabalhos futuros.

Como trabalhos futuros, podem ser citados: utilização de *datasets* de grafos reais, como Twitter, Reddit e Instagram; uma análise do estado da arte de algoritmos para encontrar SCCs; utilização de uma gama maior de algoritmos nos testes; a análise do uso de memória e CPU de cada algoritmo; e a verificação da diferença entre as implementações do *algoritmo de Tarjan* utilizadas neste trabalho.

Referências

Aho, A. V., Ullman, J. D., and Hopcroft, J. E. (1983). *Data Structures and Algorithms*. Addison-Wesley.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, 3th edition.

Skiena, S. S. (2011). *The Algorithm Design Manual*. Springer, 2th edition.

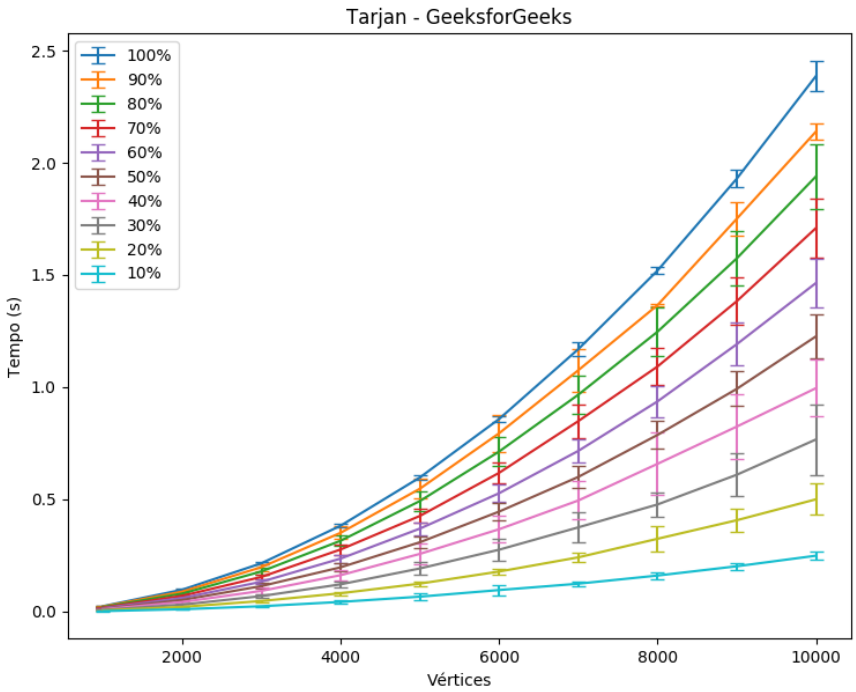

Figura 8. Algoritmo de Tarjan - GeeksforGeeks

Tarjan, R. (1971). Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory*. IEEE.