UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

JOÃO MARCOS LIMA MEDEIROS

COMPRESSÃO DE ARESTAS DE GRAFOS UTILIZANDO ÁRVORE DE SEGMENTOS

CAMPINA GRANDE - PB

2022

JOÃO MARCOS LIMA MEDEIROS

COMPRESSÃO DE ARESTAS DE GRAFOS UTILIZANDO ÁRVORE DE SEGMENTOS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador : Professor Dr. Rohit Gheyi.

CAMPINA GRANDE - PB

2022

JOÃO MARCOS LIMA MEDEIROS

COMPRESSÃO DE ARESTAS DE GRAFOS UTILIZANDO ÁRVORE DE SEGMENTOS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Rohit Gheyi
Orientador – UASC/CEEI/UFCG

Professora Dra. Patrícia Duarte de Lima Machado
Examinador – UASC/CEEI/UFCG

Professor Dr. Francisco Vilar Brasileiro
Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 02 de Setembro de 2022.

CAMPINA GRANDE - PB

ABSTRACT

In the context of graph theory, it is common to find optimizations centered on the “inside of the algorithm”, to try to reduce its asymptotic complexity. However, in certain cases, it is not possible to reduce the complexity of the algorithm, but this does not define the minimum number of operations to be performed to solve a given problem. This happens because manipulations can be made on the graph received as input, allowing a specific algorithm to run more efficiently. In this context, we present an optimization that allows creating edges from a vertex to all vertices of a given interval in a graph or even creating edges between all vertices of two intervals, with a cost equivalent to adding just O (log(u) + log (v)) edges, with u and v being the size of the first and second intervals respectively. This can be done by adding two segment trees to the graph, preserving its distances and connectivity. After that, we verified the edge reduction level obtained by this algorithm in dense graphs, where there may be several edges between vertex intervals, even if it happens randomly. With this optimization, we were able to reduce the number of edges by about 50% in graphs with at least 80% of the possible edges present.

Compressão de arestas de grafos utilizando Árvore de Segmentos

João Marcos Lima Medeiros
joao.medeiros@ccc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba

Rohit Gheyi
rohit@dsc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba

RESUMO

No contexto de teoria dos grafos, é comum encontrarmos otimizações centradas no "interior do algoritmo", de maneira a tentar reduzir a complexidade assintótica do mesmo. Porém, em determinados casos, não é possível reduzir a complexidade do algoritmo, mas isso não define o mínimo de operações a ser feitas para resolver determinado problema. Isso acontece pois podem ser feitas manipulações no próprio grafo recebido como entrada, possibilitando que um algoritmo específico possa executar de maneira mais eficiente. Nesse contexto, apresentamos uma otimização que permite criar arestas de um vértice para todos os vértices de determinado intervalo num grafo ou até criar arestas entre todos os vértices de dois intervalos, com um custo equivalente ao de adicionar apenas $O(log(u) + log(v))$ arestas, sendo u e v o tamanho do primeiro e segundo intervalos respectivamente. Isso pode ser feito através da adição de duas árvores de segmentos no grafo, preservando as distâncias e a conectividade do mesmo. Após isso, verificamos o nível de diminuição de arestas obtido através desse algoritmo em grafos densos, onde podem existir várias arestas entre intervalos de vértices, ainda que de maneira aleatória. Com essa otimização, conseguimos diminuir a quantidade de arestas em cerca de 50% em grafos com pelo menos 80% das arestas possíveis presentes.

CCS CONCEPTS

• **Theory of computation** → **Graph algorithms analysis**.

REPOSITÓRIO

https://github.com/JoaoMLima/TCC-codes

1 INTRODUÇÃO

A representação comprimida de um grafo pode ser visto como uma estrutura de dados criada para representá-lo, de modo que certos algoritmos ou consultas possam executar de maneira mais rápida quando adaptados para executar nessa versão [5]. Para este tipo de compressão, existem diferentes tipos de propriedades que podem ser mantidas do grafo original, podendo ou não haver perda de outras informações.

Neste trabalho será apresentada uma técnica que permite com que sejam adicionadas arestas de mesma distância de uma sequência de vértices consecutivos para outra sequência de vértices consecutivos, com custo de tempo e espaço logarítmico na quantidade de vértices do grafo original. Isso vai ser realizado com a ajuda de duas árvores de segmentos que são adicionadas no grafo, permitindo com que qualquer intervalo seja referenciado por no máximo $O(log(N))$ vértices, onde N é o tamanho do grafo originalmente.

Em grafos muito densos, espera-se que surjam essas "arestas entre intervalos" naturalmente, que podem então ser comprimidos com o auxílio de uma árvore de segmentos. Por isso, foi testado o quanto essa técnica pode ser efetiva também na compressão de grafos aleatórios, onde não foram adicionadas de maneira direta arestas entre intervalos, e não são diretamente o foco dessa otimização. Nesses casos, observou-se uma taxa de compressão para 50% do tamanho original do grafo quando este possui uma densidade de pelo menos 80%.

Este documento é organizado da seguinte forma: A Seção 2 explica de maneira breve os conhecimentos abordados no decorrer do trabalho. A Seção 3 apresenta o algoritmo utilizado na criação de arestas entre dois intervalos de vértices. Na Seção 4 nós apresentamos nossos experimentos e resultados. Na Seção 5, tiramos nossas conclusões e quais trabalhos podem ser feitos futuramente como continuidade a esta pesquisa.[1]

2 BACKGROUND

Na Seção 2.1 apresentamos grafos e os seus conceitos que serão abordados neste documento. A Seção 2.2 define a importância de compressões em grafos e porque isso pode ser útil em diferentes algoritmos. Por último, a Seção 2.3 apresenta a estrutra árvore de segmentos e suas importantes características.

2.1 Grafos

Um grafo é uma estrutura desenvolvida para abstrair a relação entre objetos de maneira matemática. Um grafo $G = (V, A)$ é definido a partir de um conjunto de vértices V e um conjunto de arestas A. Os vértices representam os elementos que serão relacionados, e podem guardar propriedades diversas destes elementos. Já as arestas, expressam uma relação unidirecional ou bidirecional entre dois objetos. Essa relação também pode guardar algumas propriedades pré-determinadas [10].

Neste artigo serão abordados grafos direcionados e ponderados, isso significa que as arestas não necessariamente são bidirecionais e que elas guardam um número correspondente ao seu peso.

Um caminho no grafo G é uma sequência finita não-nula $C = v_0a_1v_1a_2v_2 \cdots a_kv_k$, cujos termos são anternadamente vértices e arestas, tal que, para $1 \leq i \leq k$, a_i representa uma aresta de v_{i-1} para v_i. Podemos também dizer que este caminho conecta o vértice v_0 ao vértice v_k, com uma distância de $D = P_{a_1} + P_{a_2} + P_{a_3} + \cdots + P_{a_k}$, onde P_{a_i} é o peso da aresta a_i [3]. Os conceitos de distância e conectividade entre dois vértices serão utilizados durante todo o decorrer deste documento.

2.2 Compressão de grafos

No contexto de algoritmos, muitas vezes pode ser útil comprimir um grafo de forma que ainda seja possível obter algumas propriedades do grafo original a partir do novo grafo. Isso pode ser útil principalmente para executar de maneira mais rápida algoritmos que se baseiam em algumas propriedades específicas do grafo original.

Por exemplo, se queremos apenas fazer consultas de conectividade num grafo não-direcionado (composto apenas por arestas bidirecionais), normalmente não precisamos guardar todas as arestas do mesmo. Ao guardarmos apenas uma floresta geradora do grafo original, obteremos um grafo com no máximo $O(N)$ arestas, onde N é a quantidade de vértices do grafo, que contém toda a informação necessária para qualquer tipo de consulta de conectividade. Esse exemplo pode ser visto na Figura 1.

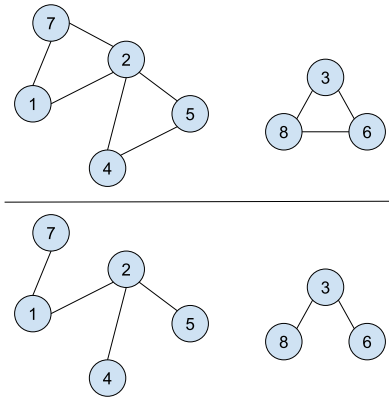

Figura 1: Exemplo de grafo e sua representação que preserva a propriedade de conectividade.

Como segundo exemplo, no caso de possuírmos uma compressão que mantém as propriedades de distância entre vértices, poderemos também executar algoritmos como o algoritmo de menor caminho de Dijkstra [4] no grafo derivado da compressão e obter informações relacionadas ao grafo original.

Neste artigo será abordada uma compressão que, a grosso modo, pode transformar o que seria apenas uma aresta em um caminho que possui uma distância igual ao peso desta aresta. Desta forma, é possível manter as propriedades de distância e de conectividade em grafos direcionados (e por consequência em grafos não-direcionados também).

2.3 Árvore de Segmentos

Uma árvore de segmentos é uma estrutura de dados primeiramente introduzida por Bentley [2] que armazena um conjunto de intervalos. É útil quando informação adicional precisa ser armazenada para cada intervalo [7]. Essa estrutura é uma árvore enraizada em que a raiz representa todo um intervalo e as folhas representam posições unitárias do intervalo, como pode ser visto na Figura 2.

Cada vértice não-folha possui arestas direcionadas para dois vértices filhos, um filho esquerdo que representa a primeira metade do intervalo que o mesmo representa, e um filho direito que representa a segunda metade deste intervalo. Esta estrutura possui algumas

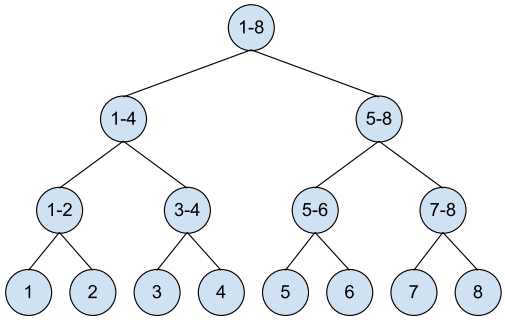

Figura 2: Representação de uma árvore de segmentos para 8 posições. Em cada vértice, é indicado o intervalo de posições que ele representa.

características que são interessantes para resolver problemas de intervalos eficientemente. A seguir são apresentadas algumas dessas características:

2.3.1 Profundidade da árvore de ordem logarítmica. Para simplificarmos a prova, vamos assumir que $N = 2^k$ e k é um número inteiro. Pois caso N não seja uma potência de 2, basta utilizar uma árvore de segmentos que armazena $2^{\lceil log(N) \rceil}$ posições e só aproveitar as N primeiras. Já que $N = 2^k$, ao percorrermos a árvore da raíz até uma das folhas, sempre vamos dividir o tamanho do intervalo representado no vértice atual por 2 até chegarmos a um intervalo unitário (que não possui vértices-filhos). Então só podemos descer na árvore no máximo $log(N)$ vezes.

2.3.2 Quantidade total de vértices e arestas de ordem linear. Novamente vamos assumir que $N = 2^k$ e k é um número inteiro. Nesse caso, é interessante notar que a quantidade de vértices sempre dobra a cada nível de profundidade na árvore. Isso acontece até chegarmos no nível dos vértices folhas, que possui N vértices. Logo, o total de vértices se dá pela equação:

$$\sum_{k=0}^{log(N)} 2^k = 2^{log(N)+1} - 1 = 2N - 1 \tag{1}$$

Cada vértice (com exceção da raiz) recebe uma aresta que possui origem no seu vértice-pai. Logo, a quantidade total de arestas é igual a $2N - 2$.

2.3.3 Qualquer intervalo pode ser decomposto em $O(log(N))$ vértices. Para mostrarmos isso, vamos partir do vértice com maior profundidade que cobre certo intervalo $I = [L, R]$ por completo. Este intervalo pode ser então quebrado em duas partes:

(1) Sufixo do intervalo coberto pelo filho esquerdo(vamos chamar o intervalo de $J = [L, M]$);
(2) Prefixo do intervalo coberto pelo filho direito (vamos chamar o intervalo de $K = [M + 1, R]$ e o filho direito de V).

Para a prova, tentaremos mostrar que o intervalo K pode ser decomposto em no máximo $O(log(N))$ vértices. Caso K seja igual ao intervalo coberto por V, esse é o único vértice retornado, então assumiremos que K está estritamente contido em V.

Ao analizarmos o vértice V, um ponto chave é que não é possível que os dois filhos deste vértice façam parte da resposta final, pois

nesse caso, o próprio deveria ter feito parte. Então, só precisamos verificar se o filho esquerdo de V está contido no intervalo K ou não. Caso a resposta seja positiva, poderemos adicioná-lo na resposta. Observe que nesse caso, o filho direito possui as mesmas propriedades originais de V, então podemos chamar recursivamente essa busca de resposta para esse nó. No caso de resposta negativa, essa recursão deve ser chamada para o filho esquerdo de V.

Observe que apenas um nó pode ser adicionado por passo nessa busca. Ao mesmo tempo, essa recursão só pode ter profundidade no máximo $log(N)$, que é a profundidade da árvore de segmentos. Além disso, possui apenas uma ramificação desde o vértice V até chegar numa folha.

Com base nisso, são necessários no máximo $log(N)$ vértices para formar o intervalo K. Analogamente, são necessários no máximo $log(N)$ vértices também para o intervalo J.

3 COMPRESSÃO DE ARESTAS

A otimização proposta neste artigo utiliza-se da diminuição da quantidade de arestas necessárias para ligar um intervalo de vértices a outro intervalo de vértices com arestas idênticas. Primeiramente, precisamos definir o que são arestas entre intervalos, para então demonstrar a otimização propriamente dita. Repare que quando falamos de um intervalo, estamos falando de um conjunto de vértices que suas posições na nossa lista de vértices forma um intervalo contíguo.

Os exemplos mais simples de arestas entre intervalos são demonstrados na Figura 3, que mostra um vértice V sendo ligado a todos os vértices no intervalo $[L, R]$ por uma aresta de mesma distância d. Da mesma forma é possível existir uma aresta de todos os vértices no intervalo $[L, R]$ para um vértice V.

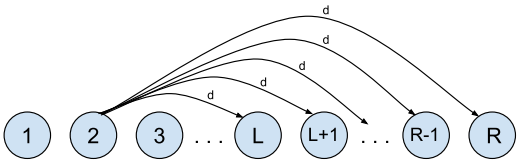

Figura 3: No exemplo acima temos o vértice 2 ligado a todos os vértices no intervalo $[L, R]$ por uma aresta de mesma distância d.

Um exemplo mais complexo é o caso em que todos os vértices do intervalo $[L, R]$ estão ligados a todos os vértices do intervalo $[X, Y]$ por arestas idênticas. Observe que esse último caso pode ser simplificado com a criação de um vértice auxiliar e com a decomposição das arestas da forma mostrada na Figura 4. Como esse vértice não representa nenhuma estrutura no grafo original, iremos nos referir a ele (e a outros vértices com essa mesma característica) como vértice *ghost*.

3.1 Algoritmo

Para conseguirmos utilizar nossa otimização, precisamos de uma estrutura baseada numa árvore de segmentos, na qual teremos vários nós *ghost* e cada nó representa um certo intervalo de nós no grafo original. Se adicionarmos arestas de distância 0 de cada vértice para seus vértices filhos (o que custaria $O(N)$ adições de aresta), será

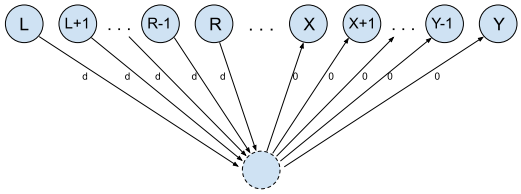

Figura 4: Simplificação de aresta entre dois intervalos com o auxílio de um vértice *ghost*.

possível adicionar uma aresta entre qualquer vértice e um intervalo de vértices $[L, R)$ em $O(log(R - L))$ passos se aproveitando dos seguintes fatos:

(1) É possível formar qualquer intervalo $[L, R)$ com $O(log(R - L))$ vértices numa árvore de segmentos.
(2) Ao ligar uma aresta ao vértice que representa certo intervalo na árvore de segmentos, indiretamente estamos ligando a todos os vértices que este intervalo cobre.

Esta forma de representar uma aresta de vértice para intervalo é demonstrada na Figura 5, onde é mostrada a criação de uma aresta entre o vértice 7 e o intervalo de vértices $[2, 5]$. Observe que o intervalo $[2, 5]$ foi decomposto em três intervalos, $[2, 2]$, $[3, 4]$ e $[5, 5]$.

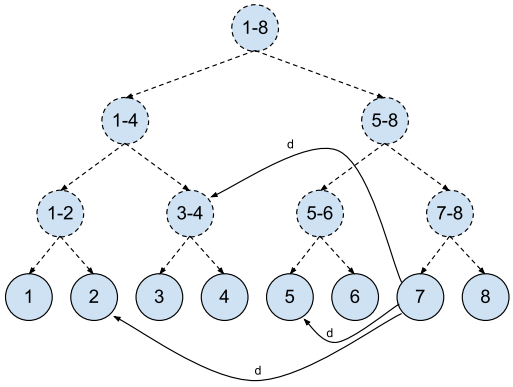

Figura 5: Grafo que mostra a criação de uma aresta entre o vértice 7 e o intervalo de vértices [2, 5]. Vértices tracejados representam vértices *ghost*. Arestas tracejadas possuem distância 0.

Para criar arestas de intervalo para nó, o raciocínio é parecido, também sendo necessária uma outra estrutura de nós *ghost* baseado numa árvore de segmentos. Porém, dessa vez, será necessário criar arestas de distância 0 dos nós filhos para seus respectivos pais.

Combinando as duas estruturas num grafo só, obteremos uma estrutura parecida com a demonstrada na Figura 6.

Para criar uma aresta de intervalo para intervalo, como dito anteriormente, precisamos de um vértice *ghost* extra. Na Figura 7 vemos como criar uma aresta de distância d saindo de todos os vértices no intervalo $[1, 3]$ para todos os vértices no intervalo $[4, 8]$. Neste caso, foram omitidos os vértices *ghost* não utilizados, para manter a legibilidade.

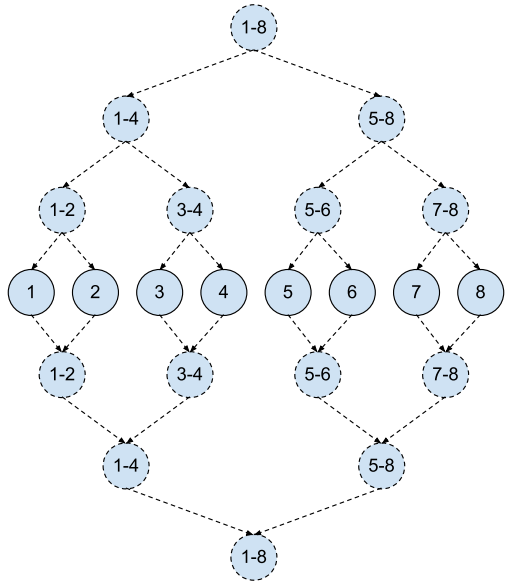

Figura 6: Grafo base para a criação de arestas de intervalo para intervalo com 8 vértices. Vértices tracejados representam vértices *ghost*. Arestas tracejadas possuem distância 0.

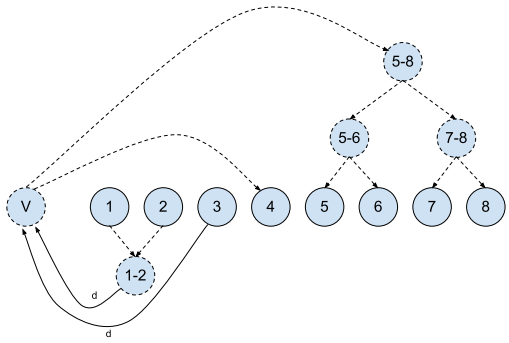

Figura 7: Grafo que mostra a criação de uma aresta entre os intervalo de vértices [1, 3] e [4, 8], usando um vértice V como vértice auxiliar. Vértices tracejados representam vértices *ghost*. Arestas tracejadas possuem distância 0.

O algoritmo agora está dividido em duas partes:

(1) Método que cria todos os vértices *ghost* iniciais e suas arestas correspondentes
(2) Método que cria aresta de intervalo para intervalo

3.2 Análise

Nesta seção será apresentada uma análise de complexidade do algoritmo dividida em duas partes. Na primeira, será abordada a complexidade do pré-processamento, que nada mais é do que a criação de um grafo preliminar dada uma quantidade N de vértices no grafo original, antes da adição de qualquer aresta. Na segunda parte, será abordada a complexidade da adição de uma aresta de intervalo para intervalo.

É válido notar que a adição de vértices ou arestas ao nosso grafo pode ser considerada de custo $O(1)$ tanto em tempo quanto em espaço se utilizarmos a representação do grafo por meio de listas de adjacências.

3.2.1 Pré-processamento. Dado a quantidade de vértices no grafo, no pré-processamento temos que garantir a criação de uma estrutura baseada na combinação de duas árvores de segmentos para N vértices cada, como demonstrado na Figura 6.

Como já visto na Seção 2.3.2, uma árvore de segmentos por si só possui $O(N)$ vértices e $O(N)$ arestas. Ao combinar estas duas estruturas, a quantidade de vértices e arestas criados continua com complexidade linear, e a compleixade total de tempo e espaço no pré-processamento é $O(N)$.

3.2.2 Complexidade por aresta. Dados dois intervalos $I = [L, R]$ e $J = [X, Y]$, para criarmos uma aresta de peso d do intervalo I para o intervalo J, de acordo com a Figura 4, devem ser seguidos os seguintes passos:

(1) Criação de um vértice *ghost*;
(2) Ligação do intervalo I para o vértice *ghost*, por meio de arestas de distância d;
(3) Ligação do vértice *ghost* para o intervalo J, por meio de arestas de distância 0.

A criação do vértice *ghost* pode ser considerada de complexidade $O(1)$ por ser apenas a criação de um vértice.

O intervalo I, por sua vez, pode ser representado por $O(log(N))$ vértices criados durante o pré-processamento e que conseguem ser alcançados a partir dos vértices no intervalo I com distância 0, como explicado na Seção 3.1. Ao adicionarmos uma aresta de peso d de cada um desses vértices para o vértice *ghost*, todos os vértices do intervalo I alcançam o vértice *ghost* em uma distância d, mas com a criação de apenas $O(log(N))$ arestas.

De maneira análoga ao que foi feito com o intervalo I, o intervalo J pode ser representado por $O(log(N))$ vértices que alcançam os vértices no intervalo J com distância 0. Ao adicionarmos uma aresta de peso 0 do vértice *ghost* para cada um desses vértices, obteremos um resultado semelhante ao apresentado na Figura 4, com a criação de apenas $O(log(N))$ arestas adicionais.

Com isso, para cada aresta de intervalo para intervalo, será necessária a criação de 1 vértice e $O(log(N)) + O(log(N))$ arestas individuais no grafo comprimido. Isso representa uma complexidade tanto de tempo quanto de espaço de $O(1) + O(log(N)) + O(log(N)) = O(log(N))$.

3.3 Aplicações

Nesta seção enumeramos alguns algoritmos que podem ser usados em conjunto com a compressão de arestas sem nenhuma perda de informação. Em seguida, mostro uma forma de comprimir grafos muito densos utilizando-se de arestas de intervalo para intervalo.

3.3.1 Algoritmos de uso geral. Com base no fato de que a compressão de arestas de intervalos apresentada acima mantém as propriedades de conectividade e distância do grafo, qualquer algoritmo que se baseia nessas propriedades pode ser executado em um grafo que passou por esse tipo de compressão, por exemplo:

(1) Algoritmo de menor caminho [4];

(2) Algoritmo para encontrar componentes fortemente conectados [9];
(3) Algoritmo de resolução de 2-SAT [1].

Estes e muitos outros algoritmos podem ser executados em grafos que possuem arestas comprimidas.

3.3.2 Compressão de grafos densos. Como dito anteriormente, o método de compressão apresentado só é efetivo na diminuição de arestas quando existem pares de intervalos (I, J) tal que para todo vértice em $v \in I$, e todo vértive $u \in J$, existe uma aresta de v para u idêntica.

Em grafos com densidade de arestas grande, e principalmente onde não existe peso ou outro fator que diferencie as arestas além dos seus dois vértices correspondentes, é possível identificar intervalos (I, J) onde pode ser aplicada essa compressão. Na Figura 8 temos um exemplo de grafo não-ponderado com 6 vértices e densidade de arestas de 50% (15 arestas), representado através de uma matriz de adjacências. Nesta representação, as arestas de intervalo para intervalo podem ser visualizadas como um retângulo em que todos os espaços em seu interior representam a presença de uma aresta.

	1	2	3	4	5	6
1	X	1	0	1	1	1
2	0	X	0	1	1	0
3	0	0	X	0	1	0
4	1	1	1	X	1	0
5	0	1	1	0	X	0
6	1	0	0	1	0	X

Figura 8: Matriz de adjacências aleatória. Retângulos como os três que foram destacados podem ser substituídos por uma aresta de intervalo para intervalo.

Para fazermos a compressão a partir da matriz de adjacências, primeiro precisamos definir alguma técnica a ser usada para escolher quais arestas de intervalo para intervalo serão usadas, de modo a tirar o melhor proveito dessa compressão e diminuir ao máximo a quantidade total de arestas no grafo.

Caso qualquer aresta entre intervalos tivesse o mesmo custo de inserção, e não um custo de ordem logarítmica com relação ao tamanho desses intervalos, encontrar a melhor forma de comprimir este grafo seria igual a encontrar a menor quantidade de retângulos necessários para cobrir todas as arestas assinaladas na matriz de adjacências. Esse problema, já conhecido como "Rectilinear Picture Compression Problem", foi estudado por Masek [6] e categorizado como NP-Hard.

Considerando isso, e considerando também que cada aresta entre intervalos possui custos diferentes, será necessário utilizar de alguma heurística para definir como "cobriremos" a matriz de adjacências. A heurística escolhida se baseia também em cobrir a matriz de adjacências com retângulos, mas não permitindo que uma célula da matriz de adjacências seja coberta por dois retângulos diferentes. Este problema também já foi estudado, e existem comprovadamente algoritmos que podem resolvê-lo em tempo polinomial, conforme descrito por Ohtsuki [8]. Iremos utilizar esse algoritmo na escolha dos retângulos a serem usados como arestas de intervalo para intervalo nos nossos testes de compressão.

Na Figura 8 podemos ver as diferenças entre as duas abordagens e como o particionamento mínimo de um polígono pode precisar gerar mais retângulos que sua cobertura.

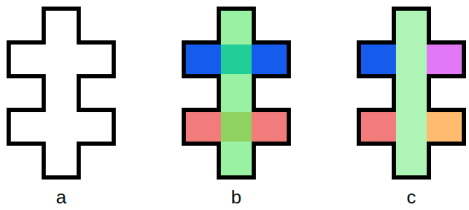

Figura 9: a) Polígono retilinear. b) cobertura mínima, com três retângulos. c) partição mínima, com cinco retângulos.

4 AVALIAÇÃO

Nosso principal objetivo na avaliação é verificar o quão bem podemos comprimir grafos utilizando a compressão de arestas entre intervalos, fazendo uso da heurística mencionada na seção anterior para escolher quais intervalos de arestas devem ser usados na compressão.

Os grafos utilizados nos testes foram gerados de forma pseudoaleatória, através de uma função que recebe como parâmentro apenas a quantidade de vértices e arestas desejados, além de uma seed. Nessa função, para cada nova aresta a ser adicionada no grafo, uma posição no produto cartesiano do conjunto de vértices que não tenha sido escolhida anteriormente é escolhida para compor a próxima aresta.

Como padrão para os testes, todos os grafos gerados possuem exatamente 100 vértices. Foram feitos testes com grafos com entre 0 e 100% de densidade, em intervalos a cada 5%.

Como pode ser visto nas Figuras 10 e 11, o algoritmo conseguiu uma taxa de compressão relevante à medida que a densidade se aproxima de 80%, podendo ser usado nesses casos. Considerando que as arestas a serem otimizadas foram escolhidas através de uma heurística, provavelmente existem melhores heurísticas que podem ser utilizadas com esse mesmo propósito.

5 CONCLUSÃO

Neste artigo, mostramos uma otimização que, com o auxílio de uma estrutura baseada em árvores de segmentos, permite inserir arestas idênticas entre dois intervalos de vértices em um grafo, com a adição de apenas $O(log(N))$ arestas, onde N é a quantidade de

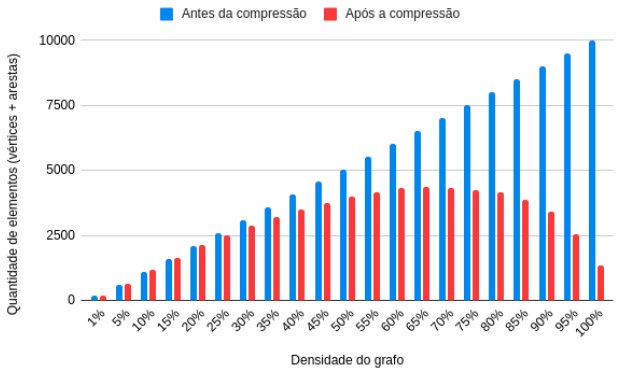

Figura 10: Quantidade bruta de elementos (vértices e arestas) nos grafos antes e depois da compressão proposta.

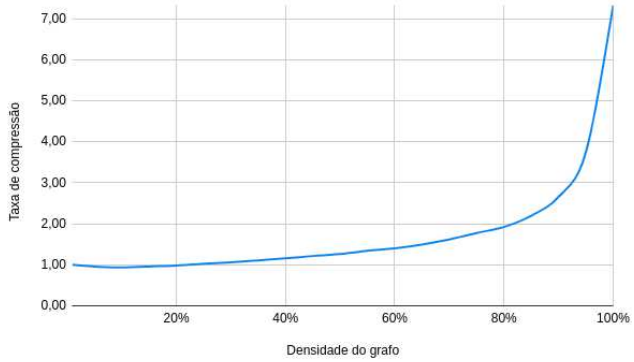

Figura 11: Taxa de compressão para cada faixa de densidade.

vértices nesse grafo. Estas arestas criadas, conseguem o mesmo efeito de manutenção das distâncias e conectividade no grafo que seria obtido de forma convencional. Demonstramos a complexidade referida acima e performamos testes para verificar se a otimização possui utilidade na compressão de grafos aleatórios que possuem uma alta densidade de arestas.

A otimização pode ser útil para que a execução de qualquer algoritmo que envolve distâncias e conectividade em grafos que possuem essas arestas de intervalo para intervalo seja feita em menos tempo. Também pode ser útil para comprimir grafos em geral, desde que o grafo possua uma densidade muito alta.

5.1 Trabalhos Futuros

Para trabalhos futuros, o estudo de uma boa heurística no agrupamento das arestas pode resultar numa diferença considerável nos resultados apresentados. Não só é necessária uma heurística que resulte num bom agrupamento, mas também que possua um tempo de execução razoável. Além disso, faz-se necessário o estudo dessa otimização para comprimir grafos provenientes de dados reais, e não puramente aleatórios, que foi o caso apresentado aqui.

AGRADECIMENTOS

Agradeço ao Dr. Rohit Gheyi por toda sua ajuda durante a preparação deste trabalho e durante todo o decorrer da graduação. Agradeço a instituição Universidade Federal de Campina Grande e a coordenação do curso de Ciência da Computação por todo o suporte que recebi. Agradeço também aos meus colegas de toda a graduação por me ajudar a seguir em frente.

REFERÊNCIAS

[1] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. 1979. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inform. Process. Lett.* 8, 3 (1979), 121–123. https://doi.org/10.1016/0020-0190(79)90002-4

[2] Jon Louis Bentley. 1977. Solutions to Klee's rectangle problems. *Unpublished manuscript* (1977), 282–300.

[3] J.A. Bondy and U.S.R. Murty. 1976. *Graph Theory with Applications.* American Elsevier Publishing Company.

[4] Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1 (1959), 269–271.

[5] Tomás Feder and Rajeev Motwani. 1995. Clique partitions, graph compression and speeding-up algorithms. *J. Comput. System Sci.* 51, 2 (1995), 261–272.

[6] William J Masek. 1979. Some NP-complete set covering problems. *Unpublished manuscript* (1979).

[7] Kurt Mehlhorn. 1984. *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry.* Springer-Verlag, Berlin, Heidelberg.

[8] Tatsuo Ohtsuki. 1982. Minimum dissection of rectilinear regions. In *Proc. 1982 IEEE Symp. on Circuits and Systems, Rome.* 1210–1213.

[9] M. Sharir. 1981. A strong-connectivity algorithm and its applications in data flow analysis. *Computers Mathematics with Applications* 7, 1 (1981), 67–72. https://doi.org/10.1016/0898-1221(81)90008-0

[10] Steven S. Skiena. 2008. *The Algorithm Design Manual* (second ed.). Springer-Verlag, USA.