UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

INSTITUTO DE MATEMÁTICA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RODRIGO TOSCANO NEY

UMA IMPLEMENTAÇÃO DO ALGORITMO DE RECONHECIMENTO DE GRAFOS DE DISCO UNITÁRIO

RIO DE JANEIRO

2019

RODRIGO TOSCANO NEY

UMA IMPLEMENTAÇÃO DO ALGORITMO DE RECONHECIMENTO DE GRAFOS DE DISCO UNITÁRIO

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Vinícius Gusmão Pereira de Sá

RIO DE JANEIRO

2019

CIP - Catalogação na Publicação

Toscano Ney, Rodrigo
T568i UMA IMPLEMENTAÇÃO DO ALGORITMO DE RECONHECIMENTO DE GRAFOS DE DISCO UNITÁRIO / Rodrigo Toscano Ney. - Rio de Janeiro, 2019.
50 f.

Orientador: Vinícius Gusmão Pereira de Sá.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2019.

1. Grafo de disco unitário. 2. Geometria de distâncias. 3. Discretização. 4. Trigrafo. I. Gusmão Pereira de Sá, Vinícius, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

RODRIGO TOSCANO NEY

UMA IMPLEMENTAÇÃO DO ALGORITMO DE RECONHECIMENTO DE GRAFOS DE DISCO UNITÁRIO

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 05 de AGOSTO de 2019.

BANCA EXAMINADORA:

Prof. Vinícius Gusmão Pereira de Sá

Prof. Silvana Rossetto

Prof. Mitre Costa Dourado

AGRADECIMENTOS

Aos meus pais, João Luiz Rodrigues Ney e Katia Cabral Toscano, por todo suporte emocional e educação que me proporcionaram, além do amor incondicional. Às minhas avós, Nazaré Cabral (in memorian), Dinar Rodrigues Ney (in memorian) e aos meus avôs Reynaldo Toscano e João Ney pelo incentivo e amor que sempre me impulsionaram a ser uma pessoa melhor.

Ao meu orientador, Vinícius Gusmão, pela paciência e dedicação, e tantos outros professores do curso de Ciência da Computação da UFRJ que me inspiraram desde o início da graduação.

Por fim, a minha esposa e grande amor Fernanda Ribeiro Gomes Silvino Ney, que esteve ao meu lado durante todos os momentos difíceis e que nunca hesitou em me apoiar a superar meus obstáculos.

RESUMO

O trabalho apresentado descreve um algoritmo para solucionar o problema de reconhecimento de um grafo de disco unitário. Um grafo de disco unitário (GDU) é representado por discos de diâmetro unitário no plano euclidiano em que, para nós vizinhos, os discos se sobrepõem, ou seja, os nós são retratados por discos e as arestas são definidas pela distância entre o centro desses discos. No caso de nós vizinhos a distância euclidiana é inferior ou igual a uma unidade de medida. O reconhecimento deste conjunto de grafos é conhecidamente um problema NP-Difícil. Um algoritmo que se utiliza do paradigma de computação concorrente para solução computacional deste problema é aqui proposto. A solução se baseia em discretizar o plano, aproximando qualquer posição de nós para a coordenada encontrada no canto inferior esquerdo de cada célula de uma malha. Os nós de um grafo são então exaustivamente posicionados nesta malha, respeitando as regras de vizinhança estabelecidas pela distância entre os nós. São testados diferentes tamanhos de malhas aonde o algoritmo tem boa chance de concluir se um grafo é GDU.

Palavras-chave: Grafo de disco unitário. Geometria de distâncias. Discretização. Trigrafo.

ABSTRACT

The work presented here describes an algorithm for solving the problem of recognition of unit disk graphs. A unit disk graph (UDG) is represented by disks of unit diameter in the Euclidean plane where, for neighboring nodes, the disks overlap. In other words, the nodes are portrayed by disks and the edges are defined by the distance between the center of these disks. In case of neighboring nodes the Euclidean distance is less than or equal to one unit of measure. The recognition of this set of graphs is known to be NP-Hard. An algorithm that takes advantage of concurrent computing methods for finding a computational solution of this problem is proposed. The solution is based on discretizing the plane, approximating any node position to the coordinate found in the lower left corner of each cell of the mesh. The nodes of a graph are then positioned in this mesh, respecting the rules of neighborhood established by the distance between the points. Different mesh sizes are tested and the algorithm can conclude whether a graph is UDG or if no possible realization is possible.

Keywords: Unit disk graph. Distance geometry. Discretization. Trigraph.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

GDU	Grafo de Disco Unitário

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

O estudo baseado na importância da distância entre dois pontos, par a par, ou seja, na geometria da distância entre pontos de um conjuto, que vem recebendo aplicações em diversas áreas. Geometria de distância se baseia em encontrar posições para pontos que sejam compatíveis com um conjunto de distâncias, para um determinado espaço euclidiano [8]. Um exemplo concreto se encontra em redes de sensores sem fio, onde a disposição dos sensores afeta diretamente a capacidade de comunicação dessa rede. Sensores muito próximos podem sofrer interferência e, no caso de grandes distâncias, os sensores podem simplesmente não obter comunicação por falta de alcance. O estudo de redes de sensores sem fio não é o único exemplo de aplicação da teoria. A geometria de distâncias também pode ser aplicada na astronomia, analisando a posição entre planetas, na biologia, observando a comunicação entre baleias, além de diversos outros exemplos.

1.1.1 Grafos

Um grafo G é composto por:

$$\begin{gathered}\text{Um conjunto de vértices, ou nós V(G)} \\ \text{Um conjunto de arestas E(G)}\end{gathered} \tag{1.1}$$

Um grafo pode ser classificado como **direcionado** ou **não direcionado** [13]. Grafos direcionados conseguem representar fluxos mais limitados entre os vértices, aonde os mesmos só acontecem na direção declarada. Uma aresta xy, em um grafo **não direcionado**, é a representação de uma conexão entre o vértice x e o vértice y; em um grafo **direcionado**, xy representa uma seta que descreve um fluxo que começa no vértice x chegando em y. Algoritmos podem ser utilizados em conjunto com um grafo para se obter diversas respostas, como por exemplo o algoritmo de Dijkstra que consegue encontrar o menor caminho a se percorrer entre dois nós de um grafo com pesos.

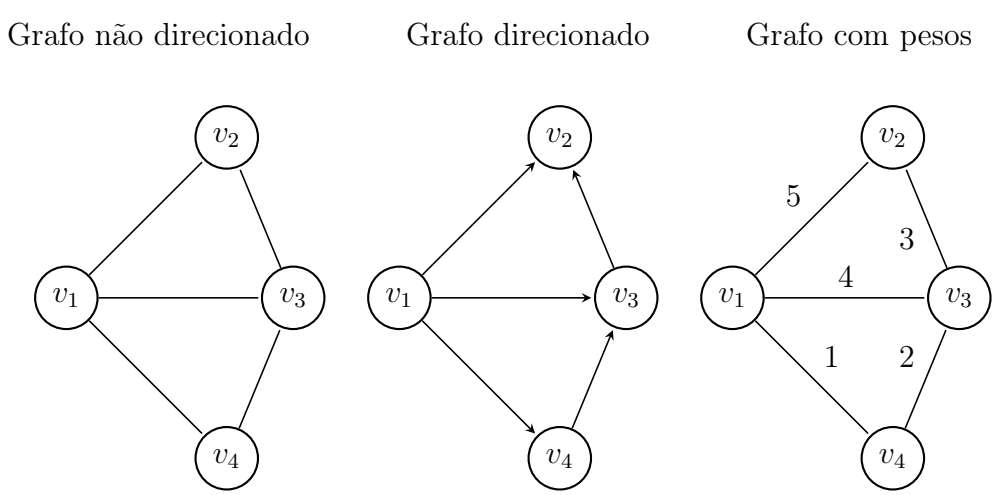

Figura 1: Exemplo de grafo não direcionado, grafo direcionado e grafo com pesos

Em geral, sensores têm um alcance que pode ser representado como disco. Um grafo de disco unitário pode ser usado para retratar o posicionamento e o alcance desses sensores em uma rede, garantindo uma melhor performance baseada nesta distribuição.

1.1.1.1 Grafos de disco unitário (GDU)

Um grupo importante de grafos são os grafos de disco unitário (GDU), que podem ser utilizados para representar problemas de distâncias relacionados a alcances a partir de pontos, como por exemplo interferência entre antenas.

Um grafo de disco unitário (GDU) consiste de um mapeamento de pontos em um plano que podem ser vistos como centro de discos [7]. Suas interseções são representadas por uma distância euclidiana igual a uma unidade de medida ou menos. Isso significa que são grafos onde todos os nós que são vizinhos são representados por pontos distribuídos no plano com uma distância euclidiana limite, e todos os pontos não-vizinhos necessariamente estão dispostos no plano com uma distância superior a esse limite [2]. Se definirmos este limite como 1, a distância entre dois vértices x e y como $d(x, y)$ teremos, para um grafo UDG:

(a) (b)

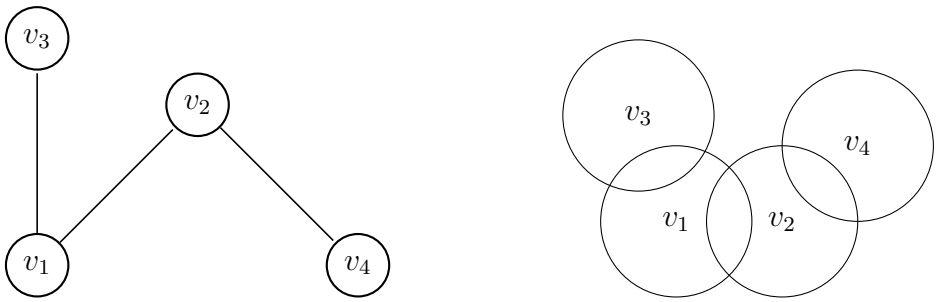

Figura 2: (a) Grafo de disco unitário G; (b) Modelo de discos congruentes para G

$$\begin{aligned} 0 \leq d(x,y) \leq 1 \text{ se } xy \in E(G), \\ 1 < d(x,y) \text{ se } xy \notin E(G) \end{aligned} \tag{1.2}$$

1.2 OBJETIVO

Reconhecer se um grafo se classifica como um grafo de disco unitário é conhecidamente um problema NP-difícil [12]. O objetivo do trabalho aqui apresentado é propor uma solução computacional para o problema de reconhecimento de grafos de disco unitário. O estudo foi baseado em soluções anteriores [3], guiando a abordagem para a otimização de algoritmos já conhecidos através da aplicação de técnicas de programação concorrente. A maior dificuldade encontrada em soluções anteriores é em relação a velocidade para encontrar uma resposta conclusiva. Este trabalho propõe conseguir uma melhor performance através do uso de *threads*.

A seção 2.1 apresentará o conceito de Trigrafos, que será utilizado para motivar a construção do algoritmo apresentado na seção 2.2. Resultados sobre a eficácia do algoritmo serão então apresentados na seção 2.3.

2 ANÁLISE

2.1 TRIGRAFO

Neste trabalho, uma definição importante para a construção da solução do problema de indentificar grafos de disco unitário, são os trigrafos. Trigrafos e suas realizações são o foco do algoritmo proposto [3] e se definem pela escolha de posições para vértices em um plano, onde suas distâncias se encontram dentro de um intervalo predefinido ou se encontram com uma pequena variação, acordada, deste intervalo.

Podemos notar que esta solução seria, na verdade, uma solução um pouco mais abrangente que a de grafo de disco unitário (GDU). Os trigrafos serão aqui usados então, como uma ferramenta para a descoberta de grafos que não são, em hipótese alguma, grafos de disco unitário. Existindo um intervalo relaxado de distâncias entre pontos, conseguimos provar que se, para um dado intervalo relaxado, nenhuma realização de trigrafo é encontrada, então mesmo diminuindo constantemente este intervalo, não existirá uma distribuição de pontos em que este grafo demonstre ser um grafo de disco unitário. Assim, dado um conjunto de pontos, se não existir uma realização de trigrafo que satisfaça as restrições definidas, podemos assumir que para qualquer outro conjunto de restrições não será possível encontrar uma distribuição de posições que respeite as regras de um grafo de disco unitário - ou seja, este grafo não é GDU.

Se dividirmos o espaço 2D em uma malha de intervalos constantes ϵ, onde $\epsilon \in \mathbb{Q}$, podemos considerar um conjunto dessas malhas variando em ϵ como $\mathbf{Q}_\epsilon = \{x \in \mathbb{Q} : x = d\epsilon, d \in \mathbb{Z}\}$ tanto no eixo das abscissas como no eixo das ordenadas. Ao chamar esse conjunto de $\mathbf{C}_\epsilon$, ou seja, para o caso de $\epsilon = 0.7$, teremos uma malha $\mathbf{C}_{0.7}$, formando vários quadrados de lado 0.7.

Como podemos ver nas Figuras 3 e 4, o algoritmo procura uma representação relaxada dos nós dentro da malha através de uma discretização. É como se estivéssemos puxando o nó para o canto inferior esquerdo da malha. A seguir será

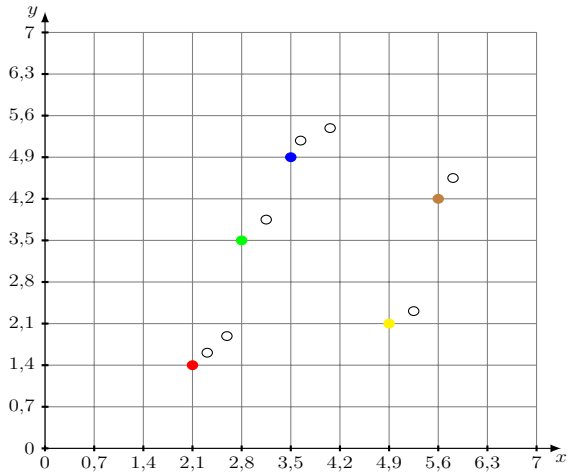

Figura 3: Exemplo de malha com $\epsilon = 0.7$

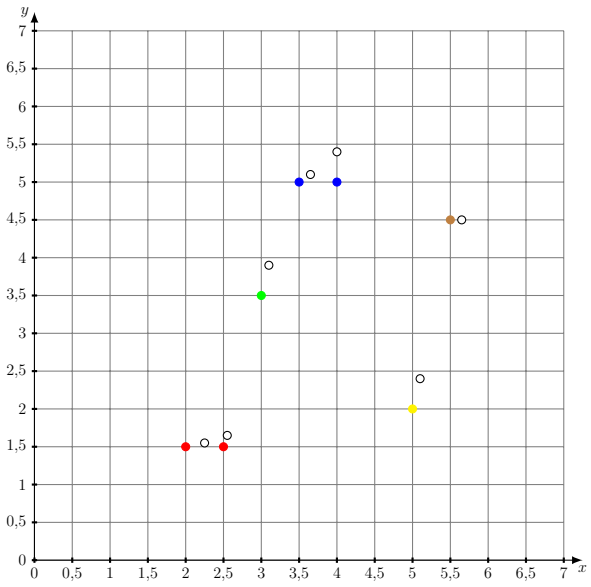

Figura 4: Exemplo de malha com $\epsilon = 0.5$

apresentada uma sequência de definições e colorários de trigrafos que foram apresentados em [3]. Essas definições são de suma importância para provar a eficácia deste método.

2.1.1 Definição 1

Sendo **G** um grafo e $\mathbf{C}_\epsilon$ uma malha, uma realização de trigafo definida nesta malha para este grafo é uma função ψ_ϵ: $V(G) \rightarrow \mathbf{C}_\epsilon$, de tal forma que para cada vértice u,v $\in V(G)$, temos:

$$\begin{aligned} d(\psi_\epsilon(u), \psi_\epsilon(v)) < 1 + \epsilon\sqrt{2} \text{ se } uv \in E(G); \\ d(\psi_\epsilon(u), \psi_\epsilon(v)) > 1 - \epsilon\sqrt{2} \text{ se } uv \notin E(G) \end{aligned} \tag{2.1}$$

Uma vez que uma realização de trigrafo se baseia em um conjunto de intervalos mais abrangentes que de um grafo de disco unitário, podemos notar que, necessariamente, todo grafo de disco unitário satisfaz as definições de trigrafo, porém, nem todo trigafo satisfaz as definições de um grafo de disco unitário. Se analisarmos este intervalo de distâncias de um trigrafo e compararmos com o os de um grafo de disco unitário, podemos subdividir os resultados em 3 categorias:

- é uma adjacência mandatória, $d(\psi_\epsilon(u), \psi_\epsilon(v)) \leq 1 - \epsilon\sqrt{2}$;
- é uma adjacência proibida, $d(\psi_\epsilon(u), \psi_\epsilon(v)) \geq 1 + \epsilon\sqrt{2}$;
- é uma adjacência opcional, $1 + \epsilon\sqrt{2} < d(\psi_\epsilon(u), \psi_\epsilon(v)) < 1 + \epsilon\sqrt{2}$.

Nessa subdivisão, podemos notar que a grande diferença se encontra na adjacência opcional. Portanto, uma aresta de um grafo é representada em um trigafo como estando no intervalo de adjacência mandatória ou no intervalo de adjacência opcional. No entanto, em um grafo de disco unitário, só existe uma divisão de intervalo onde uma aresta de um grafo pode estar, já que não existe o intervalo *opcional* neste caso. Dessa forma, em um trigrafo, para todo vértice uv ∈ V(G) temos:

- se uv ∈ E(G), então é uma adjacência mandatória ou adjacência opcional em um trigafo;
- se uv ∉ E(G), então é uma adjacência proibida ou adjacência opcional em um trigafo;

2.1.2 Definição 2

A discretização de um grafo G em uma malha C_ϵ, segue a função $\mathfrak{f}_\epsilon : \mathbb{R}^2 \to C_\epsilon$:

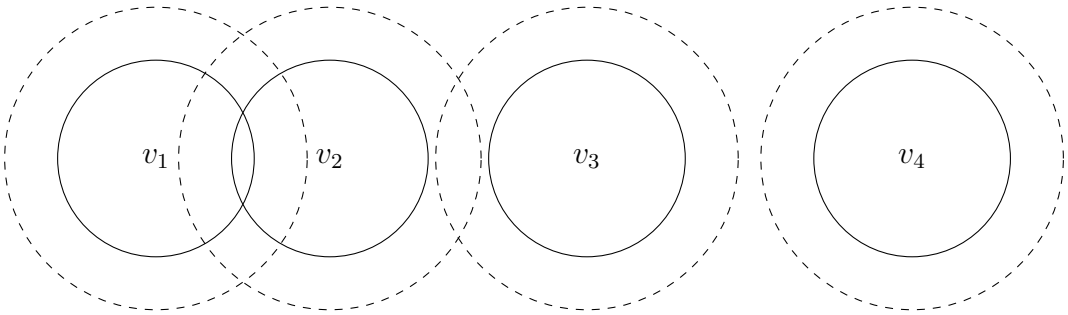

Figura 5: Exemplo de um trigrafo aonde **v1** e **v2** estão em uma adjacência mandatória, **v2** e **v3** estão em uma adjacência opcional e **v3** e **v4** estão em uma adjacência proibida.

$$\mathfrak{f}_\epsilon(x, y) : (\left\lfloor \frac{x}{\epsilon} \right\rfloor \epsilon, \left\lfloor \frac{y}{\epsilon} \right\rfloor \epsilon) \tag{2.2}$$

Basicamente, a função $\mathfrak{f}_\epsilon$ aproxima as coordenadas para a parte inferior à esquerda de um campo da malha C_ϵ, levando pontos que se encontravam em campos adjacentes na malha a compartilharem uma distância mínima de ϵ ou uma distância máxima de $\epsilon\sqrt{2}$ (diagonal). Serão então apresentados a seguir os lemas que definem o escopo de uma realização trigrafo.

2.1.3 Lema 1

Dado um grafo G e uma realização trigrafo do mesmo, se todas as adjacências opcionais desta realização trigrafo representarem um par de vizinhos no grafo G, ele então é um Grafo de Disco Unitário (GDU). O mesmo acontece no caso de todas as adjacências opcionais desta realização de trigrafo representarem não vizinhos no grafo.

Prova. Se encontrarmos uma realização trigrafo que não contém nenhuma adjacência opcional, teremos encontrado um GDU. Para provar o lema acima, transformaremos realizações de trigrafos com adjacências opcionais de um mesmo tipo (todas de vizinhos ou todas de não vizinhos) em realizações trigrafo sem adjacências opcionais, por meio de um reajuste nas distâncias entre esses vértices. Se todas as

adjacências opcionais representarem um par de vizinhos no grafo original, podemos simplesmente dividir todas as coordenadas desta realização pela maior distância encontrada em uma adjacência opcional. Já no caso de todas as adjacências opcionais representarem um par de não vizinhos no grafo original, podemos simplesmente dividir todas as coordenadas desta realização pela maior distância encontrada de uma adjacência mandatória. Independente da forma, reajustaremos o trigrafo para representar um GDU.

2.1.4 Lema 2

Se G é GDU, então existe uma realização trigrafo de G para todo $\epsilon \in \mathbb{Q}$ positivo.

Prova. Para um grafo G, sendo θ sua realização GDU, para uma malha C_ϵ de tamanho ϵ, aonde $\epsilon \in \mathbb{Q}$, sendo $\mathfrak{f}_\epsilon$ a função que discretiza θ nesta malha, teremos ψ_ϵ que mapeará V(G) em C_ϵ, ou seja, $\psi_\epsilon = \theta$ o $\mathfrak{f}_\epsilon$. ψ_ϵ é uma realização trigrafo de G. Se definirmos as distâncias entre dois pontos u, v $\in G$ nas duas realizações, com $r = \parallel \theta(u) - \theta(v) \parallel$ e $r' = \parallel \psi_\epsilon(u) - \psi_\epsilon(v) \parallel$, teremos em $\mid r' - r \mid$ um valor absoluto menor do que a $\epsilon\sqrt{2}$ (diagonal de uma célula de C_ϵ), tendo em vista que os pontos em questão foram discretizados sempre na posição inferior esquerda de sua célula na malha de tamanho ϵ. Diretamente conseguimos notar que se uv são vizinhos em G, então $r \leq 1$ e $r' < 1 + \epsilon\sqrt{2}$, e, no caso de uv não vizinhos, $r > 1$ e $r' > 1 + \epsilon\sqrt{2}$, o que categoriza uma realização trigrafo conforme definição.

2.1.5 Lema 3

Sendo G um GDU, existe uma realização GDU de G em que nenhum par de discos se tangencia (distância entre vértices exatamente igual a 1) e nenhum par de discos coincide (distância entre vértices exatamente igual a 0).

Prova. Para um grafo G, com θ sendo sua realização GDU com vértices de distância exatamente igual a 1 entre eles, e $r = 1 + \lambda$ sendo a definição da menor distância entre dois vértices não vizinhos deste grafo, e, para todo vértice u $\in$ V(G), definiremos que $\theta'(u) = \theta(u)/(1 + \lambda')$ para algum positivo $\lambda' < \lambda$, tere-

mos que o resultado de $\theta' : V(G) \rightarrow \mathbb{R}^2$ é GDU sem conter nenhum par de discos se tangenciando. Todo o par de vizinhos desda realização GDU tem distância de máximo $1/(1+\lambda') < 1$ e todo par de não vizinhos tem uma distância mínima de $(1+\lambda)/(1+\lambda') > 1$.

Para o caso de um grafo G, com θ sendo sua realização GDU não contendo nenhum par vértices com distância exatamente igual a 1, sendo $\lambda = \min(|\ 1- \parallel \theta(u) - \theta(v) \parallel|)$ para todo u, v $\in$ V(G). Se esta realização GDU conter discos coincidentes, tais que seus vértices w, z $\in$ V(G) se definem por $\theta(w) = \theta(z)$ (distância entre vértices exatamente igual a 0), podemos pertubar minimamente as coordenadas de um destes vértices de forma a fazer com que estes discos não se coincidam e que as relações de vizinhos e não vizinhos do grafo original G se mantenham respeitadas. Conseguimos esta perturbação mínima através de $\theta'(w) = \theta(w) + (\lambda', 0)$ para um λ' positivo, aonde $\lambda' < \lambda$, tendo como resultado um posicionamento de w que não coindice com z e é GDU pois a distância máxima entre dois vizinhos se torna $1 - \lambda + \lambda' < 1$ e a distância mínima entre dois não vizinhos $1 + \lambda - \lambda' > 1$. Podemos repetir a tranformação para todo vértice w, z $\in$ V(G) coincidentes.

2.1.6 Corolário 1

Para um grafo G, um positivo $\epsilon \in \mathbb{Q}$ e qualquer par de vértices u, v $\in G$ de um GDU, existe uma discretização para trigrafo ψ_ϵ aonde $\psi_\epsilon(u) \neq \psi_\epsilon(v)$.

Prova. Pelo lema 3, sabemos que se G é GDU, então existe uma realização θ aonde $\theta(u) \neq \theta(v)$. Basta então aplicarmos o lema 2, com uma malha pequena o bastante para que $\theta(u)$ e $\theta(v)$ caiam em células separadas, podendo assim então aplicar a função de discretização.

2.2 ALGORITMO

Neste seção será apresentado o pseudo-código do algoritmo para reconhecimento de grafo de disco unitário, uma explicação sobre a escolha do ϵ e a solução final implementada em java.

2.2.1 Pseudo Código

O algoritmo se divide em 3 funções:

- UDGRecognition Função inicial que itera em diversos tamanhos de malhas, tentando encontrar uma realização GDU, TRIGAFO ou dizer que não existe realização possível para este grafo;
- hasDiscreteRealization Função intermediária que inicia o processo de tentiva de colocar nós na malha;
- placeNextNode Função recursiva que tenta colocar nós na malha, respeitando as limitações de vizinhança do grafo original. Executa *backtracking* para testar todas as possibilidades de posições.

As funções **refineGranularity**, **permuteBreathFirst** e **isUDG** não são apresentadas em pseudo-código, pois podem ser implementadas de formas diferentes. No caso deste trabalho, a função **refineGranularity** apenas divide o tamanho da malha por 2 a cada iteração. Já no caso da função **isUDG**, ela testa as distâncias dos nós colocados na malha para descobrir se o algoritmo encontrou uma realização GDU do grafo dado. A função **permuteBreathFirst** apenas organiza os nós em ordem por uma busca em largura.

2.2.2 Escolha do epsilon

Para as escolhas de *epsilon*, foi necessário decidir valores para as váriaveis $epsilon_{initial}$ e $epsilon_{min}$.

A escolha para o valor de $epsilon_{initial}$ foi de $0.7 \approx \frac{1}{\sqrt{2}}$ (um pouco menor), pois como temos pela definição 1, uma adjacência mandatória precisa estar no intervalo $d(\psi_\epsilon(u), \psi_\epsilon(v)) \leq 1 - \epsilon\sqrt{2}$, ou seja, qualquer $\epsilon \geq \frac{1}{\sqrt{2}}$ faria com que a malha seja permissiva demais, não existindo nenhuma adjacência mandatória, apenas adjacências opcionais e proibidas.

Algorithm 1 UDG Recognition

Entrada: $G \leftarrow$ um grafo conectado, carregado a partir de uma matriz de adjacências;

$\epsilon \leftarrow$ granularidade inicial;

Saída: **SIM**, se G é um grafo de disco unitário;

NÃO, se G não é um grafo de disco unitário;

TRIGRAFO, Se o algoritmo não conseguiu concluir se o grafo pode ser descrito como grafo de disco unitário. Usualmente por ter parado em um critério de parada estabelecido por um ϵ_{min}.

procedure UDGRecognition(G, $\epsilon_{initial}$)

$\epsilon \leftarrow \epsilon_{initial}$

result $\leftarrow$ TRIGRAFO

$bf_{ids} \leftarrow$ permuteBreathFirst(G)

while result == TRIGRAFO **do**

if $\epsilon = \epsilon_{min}$ **then return** result

result $\leftarrow$ hasDiscreteRealization(G, ϵ, bf_{ids})

$\epsilon \leftarrow$ refineGranularity(ϵ)

return result

Algorithm 2 HasDiscreteRealization

Entrada: $G \leftarrow$ um grafo conectado, carregado a partir de uma matriz de adjacências;

$\epsilon \leftarrow$ granularidade da malha;

$bf_{ids} \leftarrow$ array com id dos nós ordenados por busca em largura;

Saída: **SIM**, se G é um grafo de disco unitário;

NÃO, se G não é um grafo de disco unitário;

TRIGRAFO, Se apenas realizações trigrafo foram encontradas para essa granularidade.

procedure hasDiscreteRealization(G, ϵ, bf_{ids})

placedNodes $\leftarrow$ empty array sized by number of nodes

return placeNextNode(G, ϵ, bf_{ids}, 0, placedNodes)

Algorithm 3 PlaceNextNode

Entrada: $G \leftarrow$ um grafo conectado, carregado a partir de uma matriz de adjacências;
$\epsilon \leftarrow$ granularidade da malha;
$bf_{ids} \leftarrow$ array com índice dos nós ordenados por busca em largura;
$nextNodeIndex \leftarrow$ índice do próximo nó a ser colocado na malha;
$placedNodes \leftarrow$ array com todos os nós já colocados na malha e suas posições em coordenadas cartesianas;

Saída: **SIM**, se G é um grafo de disco unitário;
NÃO, se G não é um grafo de disco unitário;
TRIGRAFO, Se apenas realizações trigrafo foram encontradas para essa granularidade.

procedure PLACENEXTNODE(G, ϵ, bf_{ids}, $nextNodeIndex$, $placedNodes$)
currentNode $\leftarrow G(bf_{ids}(nextNodeIndex))$
if currentNode is not the first **then**
for all neighbors already placed of currentNode **do**
possiblePositions $\leftarrow$ positions in the mesh that guarantee neighborhood with the neighbor node: $distance < 1 + \epsilon\sqrt{2}$
for all non-neighbors already placed of currentNode **do**
forbiddenPositions $\leftarrow$ forbidden positions in the mesh that breaks non-neighborhood rule: $distance \leq 1 - \epsilon\sqrt{2}$
possiblePositions $\leftarrow$ possiblePositions $-$ forbiddenPositions
if currentNode is the first **then**
possiblePositions $\leftarrow$ Pos(0,0)
else if currentNode is the second **then**
possiblePositions $\leftarrow$ possiblePositions with $ordinate = 0, abscissa > 0$
else if currentNode is the third **then**
possiblePositions $\leftarrow$ possiblePositions with $ordinate \geq 0$

foundTrigraph ← false
for all points p in possiblePositions **do**
placedNodes[bf_{ids}(nextNodeIndex)] ← p
if nextNodeIndex == $e|V(G)|$ **then**
foundTrigraph ← true
if ISUDG(G, ϵ, placedNodes) **then return** YES
else
result ← PLACENEXTNODE(G, ϵ, bf_{ids}, $nextNodeIndex$ + 1, $placedNodes$)
if result == YES **then return** YES
if result == TRIGRAPH **then**
foundTrigraph ← true
if foundTrigraph == false **then return** NO
return TRIGRAPH

Para $epsilon_{min}$, foi utilizado um valor arbitrário 0.02, pois a máquina utilizada para executar o algoritmo não teria como calcular o posicionamento na malha com um ϵ muito menor em tempo hábil. Existe, porém, um $epsilon_{min}$ matemático, calculado por McDiarmid and Müller [9], aonde é provado que para um grafo GDU de tamanho n, a realização inteira de todas as coordenadas são de no mínimo $2^{2^{\Theta(n)}}$.

2.2.3 Código Final

Nesta seção será apresentado o algoritmo, com código escrito em Java referênciado no apêndice, da solução do problema anteriormente apresentado. A decisão de arquitetura do código escrito foi baseada em um modelo MVC [4], que subdivide o código em três camadas. Para esta versão da solução, a camada de visualização contém apenas uma breve demonstração do grafo a ser testado, já que o *framework* foi aplicado para uma aplicação não *web*.

Após a demostração da solução sequencial de como iterar em malhas de diferentes tamanhos com código java sequencial, uma solução aplicando computação

concorrente será proposta na subseção seguinte.

O resultado esperado no fim da execução do algoritmo é:

- SIM Se o o grafo apresentado tiver uma realização GDU confirmada;
- NÃO Se, garantidamente, o grafo apresentado não for GDU;
- MODELO TRIGRAFO Se, após executar o algoritmo com diversas malhas de tamanhos diferentes, a aplicação atingir o tamanho mínimo predefinido de malha (condição de parada da aplicação) e apenas realizações de trigrafos forem encontradas (neste caso não conseguimos responder se o grafo é GDU ou não, e paramos com resposta inconclusiva).

O pseudocódigo que originou a solução do problema de reconhecimento de grafos GDU pode ser encontrada em [3]. Neste trabalho será apresentada a releitura do mesmo em Java. O código é aberto e pode ser encontrado no github [1].

2.2.3.1 Model–View–Controller

Em meados dos anos 70, foi criada o padrão de arquitetura MVC [4] envisionada por Trygve Reenskaug em Smalltalk. De acordo com ele, "O objetivo essencial do MVC é fazer a ponte entre o modelo mental do usuário humano e o modelo digital existente no computador" [11].

O MVC surge como uma maneira de isolar diferentes funcionalidades de um código, garantindo que o mesmo consiga ser alterado em uma das camadas sem que se tenha o conhecimento do que acontece nas outras camadas. Um exemplo seria uma modificação de uma aplicação web, aonde apenas alguma alteração na parte gráfica é requerida, com Model-View-Controller, pode-se alterar apenas o código da camada *View*, sem necessariamente precisar conhecer o código das outras duas camadas.

[1] `https://github.com/rodrigoney/udg-with-ranges`

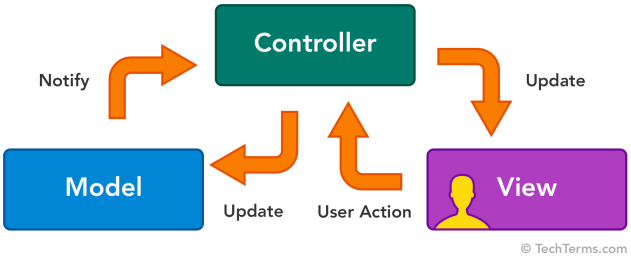

Figura 6: Modelo MVC. [2]

Além da facilidade de manutenção do código existente, a metodologia MVC permite fácil adição de novas funcionalidades. Programadores podem pensar diretamente em um modelo, que contém informações específicas daquele domínio e "plugar"o mesmo a uma visualização sem afetar outras funcionalidades já existentes [5].

A aplicação do modelo MVC neste trabalho será descrita a seguir, porém é importante ressaltar que a camada *View* não foi importante para alcançar os resultados almejados, tendo assim apenas uma simples implementação na solução final.

2.2.3.2 Model

Para *Model* temos a parte da modelagem imutável do domínio. Para programação orientada a objeto, classes modelam o problema em si [4]. Podemos ver nesta camada as definições de todas as váriaveis envolvidas no problema, por exemplo, para o reconhecimento de grafos GDU, os conceitos de **grafo**, **nó** e **posição na malha** podem ser modelados sem que se entenda como a camada *Controller* irá lidar com essas classes.

A classe *DefaultConstants* (A.1) define as constanstes usadas na aplicação. Informações como os três possíveis retornos da execução (YES, NO e TRIGRAPH ONLY), ϵ inicial para a malha e $min(\epsilon)$ (critério de parada) são definidas nesta classe. Além disso, informações que podem ajudar na perfomance da execução tambem já são pré calculadas daqui, como a raiz de 2 e a precisão dos decimais.

[2] `https://techterms.com/definition/mvc`

Para o nó de um grafo, a classe *Node* (A.2) retêm informações relevantes do domínio. Um nó é identificado por um índice, e sabe exatamente quais são seus vizinhos. Uma variável de controle foi adicionada para ajudar na varredura do grafo usando o algoritmo de busca em largura, que será explicado posteriormente.

A interpretação de posição é feita pela classe *Position* (A.3). um ponto em 2d precisa de informações da coordenada x e da coordenada y. O método *equals* foi implementado para facilitar a comparação entre duas posições da malha.

A decisão de modelagem para o Grafo Conectado, exercida pela classe *ConnectedGraph* (A.4), foi pensada de maneira a facilitar qualquer alteração no mesmo. Métodos como *addNode* e *getNodes* auxiliam na adição de novos nós nos grafos e criam um laço entre o identificador do nó e o respectivo grafo. Podemos definir o grafo conectado como um mapa de nós, indexado pelo identificador do nó.

A última classe modelada representa o objeto resultante da execução do algoritmo, *UDGResult* (A.5). Apesar da informação esperada ser apenas se o grafo é GDU ou não, esta classe contém meta informação da execução, como por exemplo o ϵ em que o algoritmo conseguiu encontrar a resposta e, no caso do grafo ser GDU, a posição na malha referente aos nós deste grafo.

2.2.3.3 Controller

A camada *Controller* é a essência da aplicação. Toda a lógica de execução do algoritmo será administrada nesta camada. Ela conecta a camada *View* com a camada *Model*, sendo responsável por estabelecer o protocolo de comunicação, definindo a padronização de entrada e saída dos dados. Pode-se definir o controlador como o responsável por tratamento de eventos, conectando a interação externa a lógica interna [11].

O algoritmo de reconhecimento de grafos GDU, neste trabalho, foi implementado na camada *Controller*. Por ser um código complexo, ao invés de apresentarmos o código do controlador de uma única vez, apresentaremos o algoritmo para a solução sequencial do problema em partes menores.

A classe que representa o controlador da aplicação foi nomeada de *UDG* (A.6). É desejado que tenhamos apenas um único controlador para gerenciar o fluxo de execução. Esta classe segue o padrão de projeto *Singleton* [1], aonde apenas uma instância pode ser encontrada em memória. A variável **callableUDG** representa a classe a ser chamada para executar certas partes do algoritmo, visando posteriormente permitir várias instâncias do mesmo em diferentes *threads*, possibilitando o uso de programação concorrente. Sua definição será apresentada em breve.

A função *udgRecognition* (A.7) é o ponto inicial de execução. Entre seus parâmetros:

- GRAPH O grafo a ser testado sobre sua realização GDU;
- CONCURRENCY Se o algoritmo deve ser executado de maneira concorrente ou sequencial;
- CORES Caso a concorrência seja requisitada, o número de cores da máquina será usado para saber quantas *threads* a aplicação deve alocar.

Alguns objetos são instanciados ao início da função:

- EPSILON O valor atual de epsilon a ser usado, ele é iniciado com o valor estipulado em *DefaultConstants*;
- BREADTH FIRST PERMUTATION IDS Uma lista com os identificadores dos nós ordenados através do algoritmo de busca em largura. Esta será a ordem usada para a tentiva de inserção dos nós na malha;
- RESULT O objeto que irá conter o resultado do algoritmo, ele é iniciado com *TRIGRAPH_ONLY*;
- CALLABLE UDG O objeto que contém o retorno do algoritmo para um certo epsilon.

A **udgRecognition** pode ser processada de duas maneiras diferentes, de forma concorrente ou de forma sequencial. O pedaço de código apresentado demonstra

apenas a forma sequencial, enquanto a forma concorrente será apresentada em breve neste texto.

O algoritmo de reconhecimento GDU é executado em loop para diferentes granularidades de malha, tendo como condições de parada já ter encontrado algum resultado diferente de *TRIGRAPH_ONLY*, ou ter atingido o tamanho mínimo de malha pré estipulado. A cada iteração do loop, a **udgRecognition** chama a função **hasDiscreteRealization** que, para o epsilon corrente, tenta classificar o grafo como *GDU* ou *NÃO GDU*. O loop então continua sempre com um epsilon novo que é calculado através da função *refineGranularity* (A.8).

Para o cálculo de refinamento de granunaliradade, foi decidido utilizar apenas uma função que retorna a metade do tamanho do epsilon atual. Outras formas de refinamento podem ser aplicadas, como por exemplo, uma divisão exponencial. Tudo depende apenas do poder computacional da máquina que irá processar o algoritmo, tendo em vista que para menores tamanho de malha o custo computacional cresce (assim como sua precisão).

A funcão *hasDiscreteRealization* (A.9) representa o início de uma execução do algoritmo de *Backtracking*. Todos os estados inicias das variáveis que irão auxiliar na execução do algoritmo são assimilados dentro desta função. O retorno da mesma irá representar o resultado encontrado para o tamanho de epsilon estipulado. A função inicia um *ArrayList* vazio, sob o qual é alocado espaço apenas para o número de nós a serem posicionados na malha. É então chamada a função *placeNextNode*, posicionando assim o primeiro nó na malha.

A próxima função, *placeNextNode*, é o principal componente do algorimo de classificação. Por ser um pouco mais complexa, será apresentada em pequenas partes.

A função recebe uma lista de parâmetros (A.10) que definem cada etapa do algoritmo recursivo.

- GRAPH A representação imutável do grafo a ser testado, contém informações de nós e arestas;

- UNROUNDED EPSILON O valor do epsilon atual, não truncado;
- UNROUNDED EPSILON SQUARED2 O resultado pré computado de $\epsilon\sqrt{2}$, não truncado;
- PERMUTED ARRAY Uma *array* de inteiros ordernados pela etapa de busca em largura no grafo;
- NEXT NODE INDEX O identificador do próximo nó a ser posicionado na malha;
- PLACED NODES Um lista de todos os nós do grafo já posicionados na malha até então;

Para as váriveis que vão ser usadas no algoritmo, são aproximados os valores de *unRoundedEpsilon* e *unRoundedEpsilonSquared2* para 7 casas decimais. Outras variáveis são iniciadas, vazias, com o intuito de serem utilizadas mais a frente dentro da mesma função.

A primeira etapa do *placeNextNode* é a listagem de todas as posições permitidas (A.11) e todas as posições não permitidas, na malha, para se posicionar o nó corrente. Esta listagem é feita através de um *loop* que compara a vizinhança do nó atual com todos os outros nós do grafo recebido. No caso de nós vizinhos, são listadas todas as posições na malha que estão dentro do intervalo relaxado, como referenciado na definição 2.1.1. No caso de nós não vizinhos, são listados todos os pontos onde o nó **não** pode ser posicionado. Ao final do *loop*, excluímos todos as posições proibidas da lista de posições possíveis.

Após a análise de posições possíveis, foi implementado no código uma otimização para os três primeiros nós (A.12). Sempre posicionamos o primeiro nó na posição [0,0], o segundo nó no eixo das abscissas e o terceiro nó nos quadrantes de ordenada positiva. Vale lembrar que a ordem na qual posicionamos os nós vem de uma busca em largura, ou seja, garantimos que o nó será vizinho de outro nó ao posicioná-lo na malha.

Com a lista de possíveis posições, podemos então testar todas as opções (A.13). A parte final da função itera toda as posições, adicionado o nó corrente a lista de nós

já inseridos na malha, chamando então, em recursão, a mesma função *placeNextNode*. Podemos notar que o algoritmo sempre tentará colocar nós nas posições encontradas em sequência, executando o *backtrack* caso as opções daquela linhagem não sejam indubitavelmente classificatórias. Três resultados podem ser gerados a partir desta lógica:

- Após o algoritmo conseguir posicionar todos os nós na malha, sem adjacências opcionais, o resultado será que o grafo é GDU.
- Após o algoritmo conseguir posicionar todos os nós na malha mas o resultado não ser claramente GDU, com nós vizinhos e não vizinhos em uma distância de adjacência opcional. Sendo assim, o resultado será uma realização trigrafo.
- Após, exaustivamente, o algoritmo testar todas as opção e, em nenhuma delas conseguir posicionar todos os nós, podemos afirmar que este grafo não é GDU. Vide Lemma 2 em 2.1.4.

2.2.3.4 View

A última camada aqui apresentada, *View*, é onde todo a interação com o usuário acontece. Nesta camada esta alocada a interface gráfica da aplicação em que o design e usabilidade são o foco [11]. Este projeto não teve como objetivo trabalhar em cima da camada *View*, o algoritmo sempre lê o grafo a ser testado de uma arquivo local, pré escrito em disco antes da execução. Apenas uma repretação gráfica simples do gráfico é renderizada para garantir que o grafo gerado a partir da matriz de adjacência do arquivo é realmente o desejado.

2.2.4 Concorrência

Na seção anterior foi discutida a solução do algoritmo de reconhecimento de Grafos de Disco Unitário, utilizando um algoritmo sequencial que faz uso de técnicas de *backtracking*. A maneira com que o algoritmo é executado depende diretamente de dois *loops*:

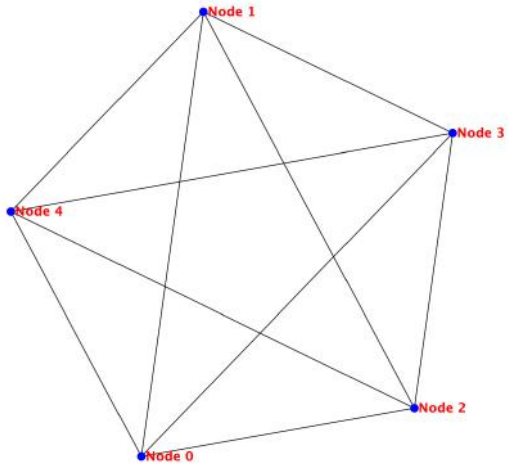

Figura 7: Representação do grafo k5

- Um *loop* principal que itera sobre diferentes tamanhos de malhas. O valor de ϵ diminui gradativamente, aumentando a chance de conseguir uma resposta positiva ou negativa quanto à classificação de GDU.

- Para uma dada malha, existe o *loop* que testará todas as combinações possíveis de posições para os nós na malha, respeitando as definições de vizinhaça do grafo.

Neste trabalho foi proposta uma otimização em relação aos diferentes valores possíveis de ϵ. Ao invés de uma execução sequencial, são instânciadas multiplas *threads*, com o objetivo de processar o posicionamento de vértices de diferentes malhas em paralelo. As combinações de posições por malha continuam sendo testadas sequencialmente, porém, desta forma, podemos identificar respostas para malhas de tamanhos menores em maior velocidade pois não precisamos esperar o resultado inconclusivo de malhas mais largas.

Esta otimização (A.14) foi aplicada na função *udgRecognition*, onde pode-se explicitamente requerer que a execução seja concorrente. Diferentemente da execução sequencial, para garantir que iremos usar o máximo de recursos possíveis, é instanciado um *pool* de *threads* com uma quantidade igual ao número de cores menos 1 (subtração de 1 devido a uma *thread* já ser alocada a *thread* principal) [10]. Esse valor pode ser modificado previamente, pois o número de cores é um parâmetro para

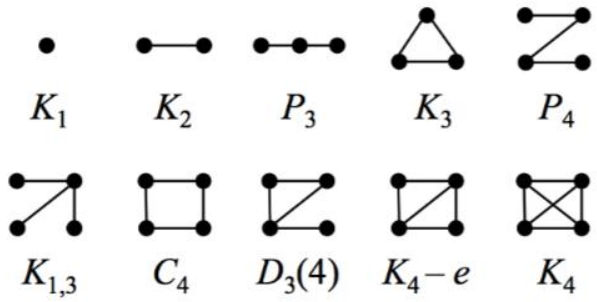

Figura 8: Grafos com menos de 5 nós

essa função. Para este projeto, foi utilizado um *hardware* com 8 cores, ou seja, 7 *threads* no *pool*.

Após o *pool* de *threads* ser instanciado, o código inicia, para cada *thread*, o processo sequencial de posicionamento de nós na malha. Cada *thread* irá executar o processo para um diferente tamanho de ϵ.

A parte final desta modificação, e que garante que não iremos esperar o término de todas as *threads*, é onde testamos o resultado de cada thread. A *main thread* fica pausada, escutando o primeiro retorno dado por alguma thread, do pool instanciado. No caso de um retorno conclusivo (GDU ou Não GDU), a main thread para imediatamente o processo, sem esperar o resultado das outras threads [6]. No caso de uma resposta inconclusiva, a main thread simplesmente loga o resultado desta thread e pausa novamente, esperando um próximo resultado.

2.3 RESULTADOS

A aplicação foi executada para a análise de diferentes tipos de grafos. O algoritmo testou se o grafo era GDU utilizando tanto o modelo sequencial de execução, como o modelo concorrente com o uso de *threads*. Para este trabalho foram testado apenas grafos de no máximo 5 nós.

Todo o código executado foi escrito em Java utilizando a abstração de *Executors* da biblioteca de concorrência *java.utils.concurrent*. Para seguir o padrão de trabalhos anteriores, a execução começou com $\epsilon = 0.7$, com a função *refineGranularity*

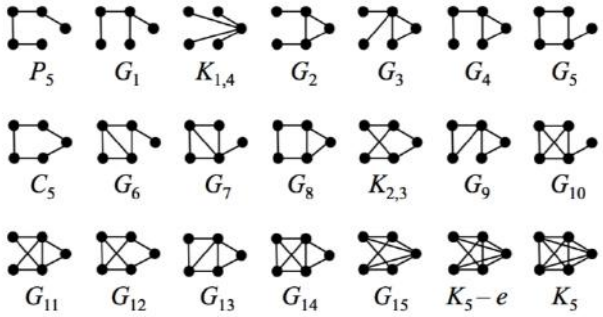

Figura 9: Grafos com 5 nós

apenas refinando o novo ϵ para o valor corrente divido por 2. O tempo obtido foi calculado apenas medindo o fluxo a partir do iníco da execução da função *udgRecognition*, nenhum tempo gasto para plotar o grafo ou escrever no stdout foi medido. O tempo medido para cada caso é o resultado da média de tempos de 10 execuções consecutivas.

Como critério de parada, o modelo de execução sequencial teve como valor mínimo $\epsilon = 0.021875$, nenhum epsilon abaixo deste valor foi executado, gerando resultados de *TRIGRAPH_ONLY* caso o algoritmo tenha atingido este limite de forma inconclusiva. Para o modelo de execução concorrente, o critério de parada é o fim de todas as threads, ou seja, relativo ao numero de core passados. Estes resultados foram gerados utilizando um computador com 8 cores, ou seja, 7 threads, gerando um ϵ mínimo possível de $0.7/2^{7-1} = 0.0109375$.

Após a obtenção dos tempos de execução ficou claro que, para grafos com poucos nós, o reconhecimento de um **Grafo de Disco Unitário** através da solução computacional é relativamente rápido. As execuções conseguiram classificar corretamente os grafos em menos de 1 segundo para a maioria dos casos, como podemos ver na Tabela 1 e na Tabela 2. A única exceção foi a classificação do grafo *g15* para o modelo de procesamento sequencial, aonde o tempo gasto foi de 1.3 segundos.

Como o algoritmo encontrou uma classificação para a maioria dos grafos em um tempo eficiente, o modelo de execução sequencial acabou conseguindo resultados de tempo melhores do que os do modelo de execução concorrente nesta amostragem. Isso aconteceu devido ao fato do resultado ter sido encontrado tão rapidamente para

Grafo	**Sequencial (ms)**	ϵ	**concorrente (ms)**	ϵ	**Resultado**
$k1$	6	0.7	11	0.7	GDU
$k2$	16	0.7	26	0.7	GDU
$k3$	25	0.7	35	0.7	GDU
$k4$	44	0.7	80	0.7	GDU
$k5$	388	0.7	492	0.7	GDU
k_{1-3}	24	0.7	37	0.7	GDU
k_{1-4}	422	0.35	83	0.175	GDU
k_{4-e}	43	0.7	82	0.7	GDU
k_{5-e}	401	0.7	441	0.04375	GDU
$p3$	25	0.7	36	0.7	GDU
$p4$	26	0.7	37	0.7	GDU
$p5$	27	0.7	41	0.7	GDU
$c4$	97	0.35	91	0.35	GDU
$c5$	82	0.7	133	0.0875	GDU
d_{3-4}	38	0.7	68	0.7	GDU

Tabela 1: Tempos em milisegundos e resultados para os grafos: k, p, c, d

esses casos que o simples fato do modelo de execução concorrente precisar instanciar objetos que administram as *threads* acabou impactando na performance do mesmo. É de se notar também que para grafos mais complexos, em que o resultado da execução sequencial é encontrado em um $\epsilon < 0.7$, o modelo de execução concorrente acaba prevalecendo sobre o modelo sequencial.

Um ponto interessante a se levantar em relação aos resultados foi a diferença de ϵ para as soluções de execução sequencial e concorrente de um mesmo grafo. Em alguns casos, a solução concorrente obteve resultados conclusivos em granularidades mais baixas e até mesmo mais rapidamente do que a solução de execução sequencial (como por exemplo para os grafos k_{1-4} e $g15$). Este fenômeno pode estar diretamente relacionado à sequência escolhida para o posicionamento dos pontos na malha. Como as tentativas de preenchimento da malha são feitas seguindo o resultado da busca em

largura, algumas granularidades de malha podem acabar encontrando um resultado conclusivo para combinações que se encontram no início da sequência e que em uma granularidade maior não encontravam um resultado conclusivo.

Grafo	**Sequencial (ms)**	ϵ	**Concorrente (ms)**	ϵ	**Resultado**
$g1$	27	0.7	44	0.7	GDU
$g2$	35	0.7	59	0.35	GDU
$g3$	33	0.7	65	0.7	GDU
$g4$	29	0.7	54	0.35	GDU
$g5$	463	0.35	382	0.04375	GDU
$g6$	166	0.7	389	0.7	GDU
$g7$	193	0.7	593	0.7	GDU
$g8$	697	0.35	159	0.0875	GDU
$g9$	174	0.7	451	0.04375	GDU
$g10$	175	0.7	380	0.7	GDU
$g11$	174	0.7	529	0.7	GDU
$g12$	786	0.35	557	0.04375	GDU
$g13$	209	0.7	384	0.04375	GDU
$g14$	206	0.7	450	0.04375	GDU
$g15$	1321	0.35	573	0.04375	GDU

Tabela 2: Tempos em milisegundos e resultados para os grafos: g

3 CONCLUSÃO

Vimos que apesar do problema de reconhecimento de grafos de disco unitário ser um problema NP-Difícil, uma prova computacional é possível em alguns casos pequenos. Diferentes tipos de grafos foram colocados em prova neste trabalho e todos obtiveram um resultado conclusivo sobre sua classicação GDU em um tempo plausível. Claramente o algoritmo proposto pode ser otimizado com o uso de computação concorrente, para casos em que o resultado necessite de uma malha com $\epsilon < 0.7$.

Uma prova computacional para grafos mais complexos não parece ser palpável para os computadores que temos atualmente. Caso o valor de ϵ necessário para se obter um resultado conclusivo seja muito pequeno, o número de combinações de posições possíveis a serem testadas será grande o bastante para que a aplicação demore dias ou meses para obter uma resposta.

3.0.1 Trabalhos Futuros

Como foi citado anteriormente na seção 2.2.4, existe outra forma de abordar o problema de paralelização do algoritmo proposto. Este trabalho aborda apenas o uso de *threads* para que diversas malhas possam ser testadas concorrentemente.

Para um trabalho futuro, uma alternativa seria fazer uso de concorrência para as linhagens de possíveis posicionamentos dos nós na malha. A proposta seria de iniciar uma *thread* sempre que algum quarto nó for posicionado, tendo em vista que os três primeiros posicionamentos de nós já foram otimizados, garantindo assim que testaremos mais posições em paralelo para uma determinada malha.

REFERÊNCIAS

[1] Aljasser, K. Implementing design patterns as parametric aspects using paraaj: The case of the singleton, observer, and decorator design patterns. *Computer Languages, Systems and Structures 45* (2016), 1 – 15.

[2] Breu, H. *Algorithmic aspects of constrained unit disk graphs.* PhD thesis, University of British Columbia, 1996.

[3] de Sá, V., e et al. On the recognition of unit disk graphs and the distance geometry problem with ranges. *Discrete Applied Mathematics 197* (2015), 3–9.

[4] Deacon, J. Model-view-controller (mvc) architecture.

[5] E. Krasner, G., e Pope, S. A cookbook for using the model - view controller user interface paradigm in smalltalk - 80. *Journal of Object-oriented Programming - JOOP 1* (01 1998).

[6] Greenhouse, A., Halloran, T., e Scherlis, W. L. Observations on the assured evolution of concurrent java programs. *Science of Computer Programming 58*, 3 (2005), 384 – 411. Special Issue on Concurrency and synchonization in Java programs.

[7] Guilherme D. da Fonseca, V. G. P. d. S., e de Figueiredo, C. M. H. Shifting coresets: Obtaining linear-time approximations for unit disk graphs and other geometric intersection graphs. *International Journal of Computational Geometry & Applications 27* (2017), 255 – 276.

[8] Liberti, L., e Lavor, C. *Open Research Areas in Distance Geometry.* Springer International Publishing, Cham, 2018.

[9] McDiarmid, C., e Muller, T. Integer realizations of disk and segment graphs. *arXiv e-prints* (Nov 2011), arXiv:1111.2931.

[10] Miller, A. Core java concurrency, 2009.

[11] Pop, D.-P., e Altar, A. Designing an mvc model for rapid web application development. *Procedia Engineering 69* (2014), 1172 – 1179. 24th DAAAM International Symposium on Intelligent Manufacturing and Automation, 2013.

[12] REU, HEINZ, K. D. G. Unit disk graph recognition is np-hard. *Computational Geometry 9* (1998), 3–24.

[13] West, D. B., e others. *Introduction to graph theory*, vol. 2. Prentice hall Upper Saddle River, 2001.

A APÊNDICE

Código A.1: Constantes

```
1  public class DefaultConstants {
2
3     public final static String CONFIRMED_UDG = "YES";
4     public final static String NOT_UDG = "NO";
5     public final static String TRIGRAPH_ONLY = "TRIGRAPH ONLY";
6     public final static double SQUARED_TWO = Math.sqrt(2);
7     public final static double INITIAL_EPSILON = 0.7;
8     public final static double MIN_EPSILON = 0.02;
9     public final static int PRECISION_SCALE = 7;
10
11 }
```

Código A.2: Nó

```
1  import java.util.HashSet;
2  import java.util.Set;
3
4  public class Node {
5
6     private Integer index;
7     private Set<Integer> neighbors;
8     private boolean visited;
9
10    public Node() {
11       super();
12       neighbors = new HashSet<Integer>();
13       visited = false;
14    }
15
16    public Integer getIndex() {
17       return index;
18    }
```

```
19
20      public void setIndex(Integer index) {
21          this.index = index;
22      }
23
24      public Set<Integer> getNeighbors() {
25          return neighbors;
26      }
27
28      public void setNeighbors(Set<Integer> neighbors) {
29          this.neighbors = neighbors;
30      }
31
32      public void addNeighbor(Integer neighbor){
33          this.neighbors.add(neighbor);
34      }
35
36      public boolean wasVisited() {
37          return visited;
38      }
39
40      public void setVisited(boolean visited) {
41          this.visited = visited;
42      }
43
44      public boolean isNeighboor(int neighboorId){
45          return neighbors.contains(neighboorId);
46      }
47
48      @Override
49      public String toString() {
50          return "Node [index=" + index + ", neighbors=" + neighbors + "]";
51      }
52  }
```

Código A.3: Posição

```
1  public class Position {
2     private double x;
3     private double y;
4
5     public Position(double x, double y) {
6        this.x = x;
7        this.y = y;
8     }
9
10    public double getX() {
11       return x;
12    }
13    public void setX(double x) {
14       this.x = x;
15    }
16    public double getY() {
17       return y;
18    }
19    public void setY(double y) {
20       this.y = y;
21    }
22
23     @Override
24     public boolean equals(Object o) {
25         if (o == this) return true;
26         if (!(o instanceof Position)) {
27             return false;
28         }
29
30         Position pos = (Position) o;
31         return this.x == pos.x && this.y == pos.y;
32     }
33
```

```
34      @Override
35      public int hashCode() {
36          int result = 17;
37          result = 31 * result + (int) (x * 100000);
38          result = 27 * result + (int) (y * 100000000);
39          return result;
40      }
41
42     @Override
43     public String toString() {
44        return "(x,y) = (" + x + ", " + y + ")";
45     }
46  }
```

Código A.4: Grafo Conectado

```
1   import java.util.HashMap;
2   import java.util.Map;
3
4   public class ConnectedGraph {
5
6      private Map<Integer, Node> nodes;
7
8      public ConnectedGraph(int numberOfNodes){
9         super();
10        nodes = new HashMap<Integer, Node>(numberOfNodes);
11     }
12
13     public void addNode(Node node){
14        nodes.put(node.getIndex(), node);
15     }
16
17     public Map<Integer, Node> getNodes() {
18        return nodes;
19     }
```

```
20
21      public Node getNode(Integer id){
22          return nodes.get(id);
23      }
24
25      public void setGraph(Map<Integer, Node> nodes) {
26          this.nodes = nodes;
27      }
28
29      @Override
30      public String toString() {
31          return "ConnectedGraph [graph=" + nodes + "]";
32      }
33  }
```

Código A.5: Resultado

```
1   import javafx.util.Pair;
2   import java.util.ArrayList;
3
4   public class UDGResult {
5
6       private String result;
7       private ArrayList<Pair<Integer,Position>> placedNodes;
8       private double epsilon;
9
10      public UDGResult(String result, ArrayList<Pair<Integer,Position>> placedNodes, double epsilon) {
11          super();
12          this.result = result;
13          this.placedNodes = placedNodes;
14          this.epsilon = epsilon;
15      }
16
17      public String getResult() {
18          return result;
```

```
19      }
20
21      public ArrayList<Pair<Integer, Position>> getPlacedNodes() {
22          return placedNodes;
23      }
24
25      public double getEpsilon() {
26          return epsilon;
27      }
28
29      public void setResult(String result) {
30          this.result = result;
31      }
32
33      public void setPlacedNodes(ArrayList<Pair<Integer, Position>> placedNodes) {
34          this.placedNodes = placedNodes;
35      }
36
37      public void setEpsilon(double epsilon) {
38          this.epsilon = epsilon;
39      }
40
41      @Override
42      public String toString() {
43          return "UDGResult [result=" + result + ", placedNodes=" + placedNodes.toString() + ",
                    epsilon=" + epsilon + "]";
44      }
45  }
```

Código A.6: Singleton

```
1   public class UDG {
2       private static UDG instance;
3       private CallableUDG callableUDG;
4
```

```
5       private UDG() {
6       }
7
8       public static synchronized UDG getInstance() {
9           if (instance == null)
10              instance = new UDG();
11          return instance;
12      }
13      ...
14  }
```

Código A.7: Função udgRecognition

```
1   public UDGResult udgRecognition(ConnectedGraph graph, boolean concurrency, int cores) {
2
3       double epsilon = CallableUDG.getDoubleWithPrecisionScale(DefaultConstants.INITIAL_EPSILON);
4       int[] breadthFirstPermutationIds = CallableUDG.permuteBreathFirst(graph, 0);
5       UDGResult result = new UDGResult(DefaultConstants.TRIGRAPH_ONLY, null, 0);
6       callableUDG = new CallableUDG(graph, epsilon, breadthFirstPermutationIds);
7
8       if (!concurrency) {
9           while (result.getResult().equals(DefaultConstants.TRIGRAPH_ONLY)) {
10              if (epsilon < DefaultConstants.MIN_EPSILON)
11                  return result;
12              result = callableUDG.hasDiscreteRealization(graph, epsilon, breadthFirstPermutationIds);
13              epsilon = CallableUDG.refineGranularity(epsilon);
14          }
15      } else {
16          ... // Concurrency - ON
17      }
18
19      return result;
20  }
```

Código A.8: Refinamento da granularidade

```
class CallableUDG implements Callable<UDGResult> {
    ...
    public static double refineGranularity(double epsilon) {
      return getDoubleWithPrecisionScale(epsilon/2);
    }
    ...
}
```

Código A.9: Função hasDiscreteRealization

```
class CallableUDG implements Callable<UDGResult> {
    ...
    public UDGResult hasDiscreteRealization(ConnectedGraph graph,
                                        double epsilon,
                                        int[] breadthFirstPermutationIds){

      Map<Integer, Node> nodes = graph.getNodes();
      ArrayList<Pair<Integer,Position>> placedNodes =
             new ArrayList<Pair<Integer,Position>>(nodes.size());

      return placeNextNode(graph,
                           epsilon,
                           epsilon*DefaultConstants.SQUARED_TWO,
                           breadthFirstPermutationIds,
                           0,
                           placedNodes);
   }
    ...
}
```

Código A.10: Função placeNextNode - Variáveis

```
private UDGResult placeNextNode(ConnectedGraph graph,
                                double unRoundedEpsilon,
                                double unRoundedEpsilonSquared2,
                                int[] permutedArray,
```

```
                            int nextNodeIndex,
                            ArrayList<Pair<Integer,Position>> placedNodes){

 // vars
double epsilon = getDoubleWithPrecisionScale(unRoundedEpsilon);
 double epsilonSquared2 = getDoubleWithPrecisionScale(unRoundedEpsilonSquared2);
 UDGResult result;
HashSet<Position> possiblePositions = new HashSet<Position>();
HashSet<Position> excludedPositions = new HashSet<Position>();
 ...
```

Código A.11: Função placeNextNode - Listagem de possibilidades

```
...
// Setting up possible positions
for(int k=0; k<nextNodeIndex; k++){
   Node kNode = graph.getNode(permutedArray[k]);
   if(kNode.isNeighboor(permutedArray[nextNodeIndex])){
      possiblePositions.addAll(findPositionsOnMeshInsideCircumference(true, 1+epsilonSquared2,
           placedNodes.get(k).getValue(), epsilon));
   } else {
      excludedPositions.addAll(findPositionsOnMeshInsideCircumference(false, 1-epsilonSquared2,
           placedNodes.get(k).getValue(), epsilon));
   }
}

possiblePositions.removeAll(excludedPositions);
...
```

Código A.12: Função placeNextNode - Otimização

```
 ...
// Special rules for initial nodes
if(nextNodeIndex == 0){
   possiblePositions.add(new Position(0, 0));
} else if(nextNodeIndex == 1){
```

```
6       for (Position position : (HashSet<Position>) possiblePositions.clone()) {
7          if(!(position.getX() > 0 && position.getY() == 0))
8             possiblePositions.remove(position);
9       }
10   } else if(nextNodeIndex == 2){
11      for (Position position : (HashSet<Position>) possiblePositions.clone()) {
12         if(!(position.getY() >= 0))
13            possiblePositions.remove(position);
14      }
15   }
16   ...
```

Código A.13: Função placeNextNode - Backtracking recursivo

```
1     ...
2    boolean foundTrigrah = false;
3    for (Position pos : possiblePositions) {
4       placedNodes.add(new Pair<Integer,Position>(Integer.valueOf(permutedArray[nextNodeIndex]),
            pos));
5       if(nextNodeIndex == graph.getNodes().size()-1){
6          foundTrigrah = true;
7
8          if(isUDGrealization(graph, epsilonSquared2, placedNodes)){
9             return new UDGResult(DefaultConstants.CONFIRMED_UDG, placedNodes, epsilon);
10         }
11      } else {
12         result = placeNextNode(graph, epsilon, epsilonSquared2, permutedArray, nextNodeIndex+1,
               placedNodes);
13         if(result.getResult().equals(DefaultConstants.CONFIRMED_UDG))
14            return result;
15         if(result.getResult().equals(DefaultConstants.TRIGRAPH_ONLY))
16            foundTrigrah = true;
17      }
18      placedNodes.remove(nextNodeIndex);
19   }
```

```
20    if(!foundTrigrah)
21      return new UDGResult(DefaultConstants.NOT_UDG, null, epsilon);
22    return new UDGResult(DefaultConstants.TRIGRAPH_ONLY, placedNodes, epsilon);
23  }
```

Código A.14: Função udgRecognition - Concorrente

```
1  public UDGResult udgRecognition(ConnectedGraph graph, boolean concurrency, int cores) {
2      ...
3
4      if (!concurrency) {
5         ... // Concurrency - OFF
6          }
7      } else {
8          final ExecutorService pool = Executors.newFixedThreadPool(cores-1);
9          final ExecutorCompletionService<UDGResult> completionService = new
               ExecutorCompletionService<>(pool);
10
11         for (int i = 0; i < (cores - 1); ++i) {
12             completionService.submit(new CallableUDG(graph, epsilon, breadthFirstPermutationIds));
13             epsilon = CallableUDG.refineGranularity(epsilon);
14         }
15
16         for (int i = 0; i < (cores - 1); ++i) {
17             try {
18                 final Future<UDGResult> future = completionService.take();
19                 final UDGResult content = future.get();
20                 if(!content.getResult().equals(DefaultConstants.TRIGRAPH_ONLY)){
21                     result = content;
22                     break;
23                 } else {
24                     System.out.println("Granularity " + content.getEpsilon() + " ended with TRIGRAPH
                           ONLY");
25                 }
26             } catch (Exception e) {
```

```
27                 System.out.println("Error: " + e.getCause());
28             }
29         }
30         pool.shutdown();
31     }
32 
33     return result;
34 }
```