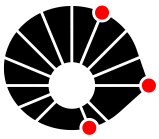
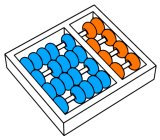




Algoritmo para o Teste de Isomorfismo de Grafos

P. T. F. Kaneko *E. C. Xavier*

Relatório Técnico - IC-PFG-17-23
Projeto Final de Graduação
2017 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Algoritmo para o Teste de Isomorfismo de Grafos

Pedro Tadahiro Furtado Kaneko* Eduardo Candido Xavier†

Resumo

Detecção de isomorfismo de grafos é um problema conhecido em Teoria de Grafos que consiste em verificar se dois grafos possuem a mesma estrutura morfológica, isto é, se existe uma bijeção entre os conjuntos de vértices com preservação de arestas. A definição de qual classe de complexidade o problema pertence ainda está em aberto pois não foi encontrada uma solução polinomial assim como também não foi provado que pertence a classe de problemas NP-Difícil.

Para este estudo foi escolhido um algoritmo exato de *backtracking* proposto por Mittal [11], para implementar, executar e comparar resultados com outros algoritmos. A comparação foi feita em relação aos resultados encontrados por Mattos [10] para o algoritmo de Weisfeiler e Lehman [15].

Os resultados mostraram que o algoritmo é superior ao método de "Força Bruta", no qual são testadas todas as permutações de vértices, e é mais eficiente que o algoritmo de Weisfeiler e Lehman para grafos do tipo *Regular Mesh*.

Mesmo o algoritmo implementado tendo a complexidade fatorial de pior caso, todos os casos de testes foram resolvidos dentro de um *timeout* de 5 min, com exceção das instâncias difíceis. Para a grande maioria dos testes, o resultado foi gerado rapidamente em tempos inferiores a 1 min.

1 Introdução

A modelagem através de grafos é uma ferramenta poderosa para a resolução de problemas na área da computação. Desta forma, grafos com propriedades semelhantes podem indicar soluções semelhantes para os problemas aos quais eles representam. O problema que buscamos resolver é determinar se dois grafos aparentemente distintos são morfologicamente iguais.

O problema de determinar a existência de um isomorfismo entre dois grafos dados é importante, pois tem aplicação em vários campos da ciência, como por exemplo em teoria de redes, recuperação de informações e química. Uma forma de se resolver o problema é pelo método *Força Bruta*, testando todas as $N!$ permutações dos N vértices de um grafo, associando cada vértice da permutação com cada vértice do outro grafo e verificando se a relação das arestas permanecem verdadeiras. Porém, este método é extremamente ineficiente e só pode ser usado na prática para grafos pequenos.

*Graduando, Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP

†Professor Associado, Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP

Apesar de já existirem tentativas de se encontrar uma função matemática que identifique grafos isomorfos [6] [13] e se ter descoberto um algoritmo de complexidade *quasipolinomial* [2], nenhuma função que consiga computar em tempo polinomial foi descoberta. Além disso, não se sabe se o problema é NP-completo [9].

O algoritmo escolhido para implementação e análise de resultados foi o proposto por Mittal em [11] que se baseou no algoritmo de Schmidt e Druffel [12]. Um dos motivos da escolha foi a ausência de testes para a verificação da eficiência do algoritmo, já que o autor afirma que o método é rápido mas não apresenta evidências práticas sobre isso.

2 Fundamentação Teórica e Notações

Nesta seção apresentamos as notações utilizadas no texto e definimos formalmente o problema de isomorfismo em grafos.

Um grafo $G(V, E)$ é uma estrutura composta por um conjunto de vértices V e um conjunto $E \subset V \times V$ de pares denominados arestas. Quando mais de um mesmo tipo de estrutura for mencionado, utilizaremos o índice superior para identificá-los, como por exemplo $G^1(V^1, E^1)$ e $G^2(V^2, E^2)$.

Chamaremos de $N = |V|$ o número de vértices de G.

Para uma aresta (v_i, v_j), dizemos que a aresta sai de v_i e entra em v_j. Chamamos de grau de entrada de um vértice v_i, o número de arestas entrando em v_i. Analogamente, define-se grau de saída de v_i como o número de arestas saindo em v_i.

A matriz de adjacência de G é a matriz $A_{N,N}$ onde:

$$a_{ij} = \begin{cases} 1 & \text{se } (v_i, v_j) \in E \\ 0 & \text{caso contrário} \end{cases} \tag{1}$$

Um passeio de v_i a v_j $(v_i, v_j \in V)$ de tamanho k é definido como a sequência de vértices $(v_i = u_0, u_1, ..., u_{k-1}, u_k = v_j)$ tal que $(u_{n-1}, u_n) \in E,\ \forall n \in \{1, 2, ..., k\}$. Um caminho de v_i a v_j $(v_i, v_j \in V)$ de tamanho k é definido como um passeio de v_i a v_j de tamanho k sem repetição de vértices.

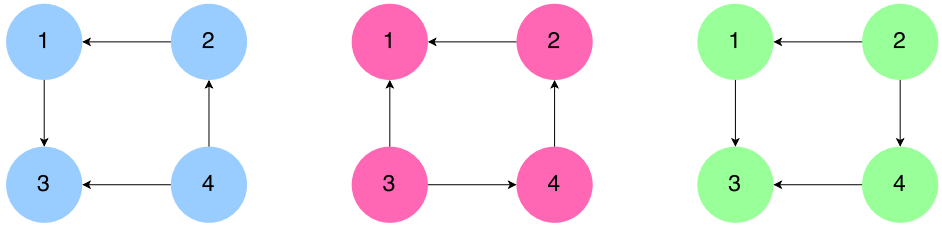
Definimos a distância $d(v_i, v_j)$ como o tamanho do menor caminho de v_i para v_j. Quando $v_i = v_j$ temos que $d(v_i, v_i) = 0$. Se não existir um caminho de v_i para v_j, matematicamente definimos $d(v_i.v_j) = \infty$, porém, por motivos computacionais, a definição $d(v_i, v_j) = N + 1$ é mais conveniente.

A matriz de distância de G é a matriz $D_{N,N}$ onde:

$$d_{ij} = d(v_i, v_j) \tag{2}$$

Dois grafos $G^1(V^1, E^1)$ e $G^2(V^2, E^2)$ são ditos isomorfos quando existe uma bijeção $f : V^1 \to V^2$ tal que $\forall (v_i^1, v_j^1) \in E^1 \implies (f(v_i^1), f(v_j^1)) \in E^2$. Denominaremos essa bijeção de mapeamento. Uma observação é que o mapeamento pode não ser único, isto é, dois grafos isomorfos podem ter seu isomorfismo representado por mais de um mapeamento.

No exemplo abaixo, os grafos azul e rosa são isomorfos, enquanto que o verde não é isomorfo com o azul nem com o rosa. A tabela 1 exibe a bijeção do isomorfismo entre os grafos azul e rosa.



(a) Grafo Azul (b) Grafo Rosa (c) Grafo Verde

Figura 1: Exemplo de problema de isomorfismo

Azul	Rosa
1	2
2	4
3	1
4	3

Tabela 1: Bijeção Azul/Rosa

Dados dois grafos, G^1, com conjunto de vértices V^1 e arestas E^1, e G^2, com conjunto de vértices V^2 e arestas E^2, deseja-se decidir se os grafos são isomorfos ou não.

Uma das grandes dificuldades deste problema é a grande variedade de grafos existentes, por isso muitas soluções são propostas para apenas certas categorias de grafos, como é o caso de grafos planares onde Hopcroft e Tarjan [7], Hopcroft e Wong [8] e Weinberg [14] todos eles desenvolveram métodos que testam em tempo polinomial.

Apesar da existência de soluções em tempo polinomial para categorias de grafos específicas, este problema em sua forma geral está em aberto pois não foi provado se ele pertence a classe dos problemas NP-Difícil mas também não foi encontrada uma solução polinomial.

3 Descrição do Algoritmo

O algoritmo escolhido para implementação e teste foi o descrito por Mittal [11], que é uma adaptação do algoritmo proposto por Schmidt e Druffel [12]. Em ambos, o algoritmo é dividido em duas partes: um particionamento inicial e o *backtrack*.

A idéia do algoritmo é classificar os vértices dos dois grafos, de modo que se os grafos forem isomorfos, os vértices de uma classe c em G^1 serão mapeados para vértices também da classe c em G^2. Idealmente, gostaria-se de classificar os vértices em exatamente N classes distintas para fazer o mapeamento direto, porém, como nem sempre isso acontece, o algoritmo faz uso do *backtrack* para fazer o resto do mapeamento.

Para a classificação dos vértices, utiliza-se as propriedades morfológicas de grau de entrada e de saída, além das distâncias entre os vértices.

São utilizados dois vetores, C e K, para cada grafo. O vetor de classes $C = [c_1, c_2, ..., c_N]$, tem como valores c_i a classe a qual o vértice v_i pertence. O vetor de contagem $K = [k_1, k_2, ..., k_N]$, tem como valores k_j o tamanho da classe j, ou seja, quantos vértices pertencem àquela classe. Uma condição necessária para que ambos grafos sejam isomorfos é que $K^1 = K^2$ sempre que as classes forem atualizadas, onde K^1 é o vetor para G^1 e K^2 para G^2.

O mapeamento de um vértice v_i^1 para um vértice v_r^2 é *consistente*[12] se:

1. $d_{ij}^1 = d_{rs}^2$ e $d_{ji}^1 = d_{sr}^2$ $\forall j, s$ tais que v_j^1 foi mapeado para v_s^2 e;
2. d_{ik}^1 (v_k^1 não mapeado anteriormente) possui um d_{rp}^2 (v_p^2 não mapeado anteriormente) correspondente tal que $c_k^1 = c_p^2$.

O objetivo do algoritmo é encontrar um mapeamento de todos os vértices em G^1 para todos os vértices em G^2 que seja consistente pela definição acima. Caso consiga, os grafos são isomorfos e o mapeamento é dado. Caso contrário, os grafos não são isomorfos.

Inicialmente, classificam-se os vértices a partir dos graus de entrada e de saída. Após isso, as classes são subdivididas agrupando seus elementos que possuem uma mesma distância P (para $P = 1, 2, ..., N-1$) entre si. Nesta subdivisão, toma-se o cuidado para que este subconjunto seja máximo e único na propriedade da distância P, pois caso contrário poderia ser subdividido em mais de uma forma e a classificação poderia não ficar consistente entre G^1 e G^2. Esta observação não foi descrita em [11], porém ela é necessária para garantir o correto funcionamento do algoritmo.

Após essas classificações iniciais, todas as classes de tamanho 1 são mapeadas.

Utilizaremos um segundo índice inferior para representar o nível do *backtrack*, deste modo $C_t = [c_{1t}, c_{2t}, ..., c_{Nt}]$ e $K_t = [k_{1t}, k_{2t}, ..., k_{Nt}]$ são, respectivamente, os vetores de classes e de contagem no nível t do *backtrack*.

Na parte de *backtrack*, a cada nível, se $v_i \in V^1$ foi mapeado para $v_r \in V^2$, as novas classes para os vértices $v_j \in V^1$ são classificadas a partir das triplas:

1. A classe do nível anterior de v_j : $c_{j(t-1)}^1$;
2. A distância d_{ij} e;
3. A distância d_{ji}

Analogamente, as novas classes para os vértices $v_s \in V^2$ são classificadas a partir das triplas:

1. A classe do nível anterior de v_s : $c_{s(t-1)}^2$;
2. A distância d_{rs} e;
3. A distância d_{sr}

Atualizando os vetores de contagem K^1 e K^2, se $K^1 = K^2$ então o mapeamento é consistente e o nível do *backtrack* aumenta, caso contrário, busca-se um novo mapeamento consistente até um isomorfismo ser determinado ou os vértices se esgotarem.

3.1 Complexidade de Tempo

O particionamento inicial possui complexidade de no mínimo $\Omega(N^3)$, devido ao cálculo de todas as distâncias de pares através do algoritmo de Floyd-Warshall[4].

A análise de complexidade da parte *backtrack* é mais complexa[12], sendo limitada inferiormente por $\Omega(N^2)$ e superiormente por $O(N \times N!)$. Sugerindo que no pior caso, o algoritmo ainda é semelhante ao de *Força Bruta*.

3.2 Algoritmo

Parte I: Particionamento Inicial

1. Usando matrizes de adjacência, encontre os graus de entrada e os graus de saída de cada vértice dos grafos.

2. Particione os vértices de G^1 onde vértices com a mesma combinação de graus de entrada e de saída são dadas as mesmas classes; instancie o vetor de classes C^1 e o vetor de contagem K^1. Usando as classes de G^1, construa o vetor de classes C^2 e o vetor de contagem K^2. Se $K^1 \neq K^2$, os grafos não são isomorfos e o algoritmo termina.

3. Usando algoritmo de Floyd-Warshall [4], encontre as matrizes de distância $D^1 = [d^1_{ij}]$ e $D^2 = [d^2_{ij}]$ para G^1 e G^2 respectivamente.

4. Considere a partição correspondente às classes c^1_i e c^2_i de G^1 e G^2, $i = 1, 2, ..., N$. Para $P = 1, 2, ..., N-1$, vértices da classe c^1_i que estão a uma distância P entre si formam uma nova classe (tomando cuidado com a observação destacada na descrição do algoritmo) e o restante forma uma nova classe. O mesmo processo deve ser feito para G^2.

5. Mapeie os vértices de G^1 pertencendo a classes de tamanho 1 com vértices de G^2 em suas classes correspondentes.

6. Faça $P = 1$; escolha o próximo vértice já mapeado de G^1; se não houver mais vértices mapeados vá para o passo **(11)**.

7. Se $P = N$, vá para o passo **(6)**; Para o vértice mapeado v^1_i de G^1, teste se existe um e somente um vértice não mapeado v^1_j de G^1 tal que v^1_i e v^1_j estejam a uma distância P um do outro. Se nenhum ou mais de um existir, vá para o passo **(10)**.

8. Para o vértice mapeado v^2_r de G^2 (mapeado de i no passo **(7)**) encontrar o vértice não-mapeado v^2_u de G^2 tal que v_u e v_r estejam a uma distância P um do outro. Se nenhum ou mais de um existir, os grafos não são isomorfos e o algoritmo termina.

9. Mapeie o vértice v^1_j de G^1 para o vértice v^2_u de G^2. Se v^1_j (logo, v^2_u) pertencer a uma classe de tamanho 2 e se v^1_k é o outro vértice pertencente a mesma classe de v^1_j, então mapeie v^1_k para o vértice v^2_v de G^2 a qual pertence a mesma classe de v^2_u. Atualize os

vetores de classes e de contagem de G^1 e G^2. Se $K^1 \neq K^2$, os grafos não são isomorfos e o algoritmo termina.

10. Se ainda há um vértice não mapeado de G^1, então $P = P + 1$, vá para o passo **(7)**.

11. Se todos os vértices já foram mapeados, então os grafos são isomorfos e o algoritmo termina. Caso contrário, encontre vértices não mapeados de G^1 e G^2.

Parte II: Backtrack

12. Seja m o número de vértices já mapeados no particionamento inicial; Faça $t = m$ e $p_i = 0$ $(m \leq i \leq N)$ (p_t é o vértice no grafo G^1 escolhido no nível t)

13. Seja C^1_m e C^2_m os vetores de classe C^1 e C^2 obtidos anteriormente. Seja K^1_t e K^2_t os vetores de contagem do nível t

14. Se $t = N$, conclua que os grafos são isomorfos e o algoritmo termina

15. Faça $i = p_t$

16. Se $i \neq 0$, vá para **(18)**

17. Escolha um inteiro i para o qual v^1_i ainda não tenha sido mapeado. Se existe uma classe c^1_{it} com um único vértice não-mapeado, então escolha i. Faça $p_t = i$

18. Escolha um r tal que $c^1_{it} = c^2_{rt}$ e para o qual ainda não tenha sido mapeado o v^2_r

19. Se existir um r, vá para **(21)**

20. Faça $t = t - 1$. Se $t \leq m$, conclua que os grafos não são isomorfos; caso contrário, vá para **(18)**

21. Componha um vetor de triplas X_1 onde $x^1_j = (c^1_{jt}, d^1_{ij}, d^1_{ji})$, gere o vetor C^1_{t+1} a partir de X_1 substituindo por um número único para cada termo único de X_1. Componha o vetor de triplas X_2 onde $x^2_j = (c^2_{jt}, d^2_{ij}, d^2_{ji})$ e gere o vetor C^2_{t+1} a partir de X_2 substituindo por um número único para cada termo único de X_2.

22. Calcule os vetores de contagem K^1_{t+1}, K^2_{t+1}

23. Se $K^1_{t+1} \neq K^2_{t+1}$, vá para **(18)**

24. Incremente t

25. Vá para **(14)**

4 Ambiente de Teste

4.1 Equipamento e Ferramentas

Os testes foram executados no seguinte ambiente:

- Sistema Operacional Microsoft® Windows 7 Ultimate (Service Pack 1) de 64 bits;
- Processador Intel® Core™ i5-4460 com 4 cores;
- 8 GB de memória RAM DDR3 1600MHz de velocidade em dual-channel;
- 1TB de HD com velocidade de 7200RPM;
- GCC(rubenvb 4.6.3) versão 4.6.3 com as flags -O3 -fexceptions -g -std=c++11 -Wextra -Wall -ansi -pedantic

4.2 Dados de Entrada

Para fins de comparação, a realização do experimento foi feita com os mesmos *datasets* de [10]. Os grafos de entrada consistem em três categorias:

- Instâncias aleatórias: 270 instâncias (135 isomorfas e 135 não-isomorfas) para grafos de 10 a 90 vértices [3];
- Instâncias difíceis: 39 instâncias (8 isomorfas e 31 não-isomorfas) para grafos de 3 a 77 vértices [1];
- Instâncias categóricas [5]:
  - Grafos conectados aleatórios: 1000 instâncias isomorfas para grafos de 20 a 1000 vértices;
  - Regular Mesh: 2300 instâncias isomorfas para grafos de 16 a 1296 vértices;
  - Modified Mesh: 2300 instâncias isomorfas para grafos de 16 a 1296 vértices;

Maiores detalhes de cada *dataset* utilizado pode ser encontrado em [10].

5 Resultados

Os resultados foram comparados em relação ao seu comportamento e tempo de execução obtidos em [10], porém, vale ressaltar que diferenças na otimização das implementações e configuração das máquinas podem alterar as medidas do tempo de execução e, portanto, deve-se dar mais importância no comportamento de cada categoria testada.

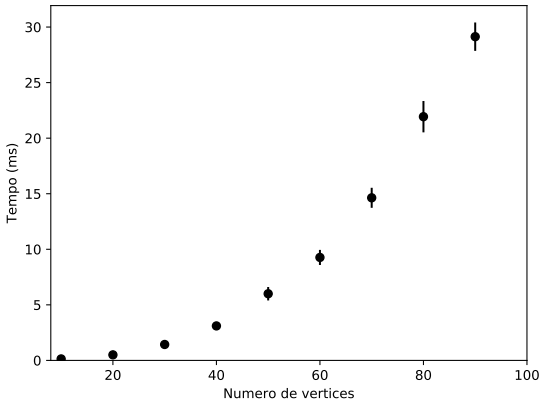


Figura 2: Resultados para instâncias formadas por grafos aleatórios isomorfos

5.1 Instâncias Aleatórias

Pela execução do algoritmo para grafos aleatórios isomorfos, obtivemos o gráfico da figura 2:

A execução para grafos aleatórios não-isomorfos foi concluída para todas as instâncias com tempo inferior a 1 ms e não foi gerado o gráfico.

Pela curva da figura 2, é possível notar que o tempo de execução cresce rapidamente com o aumento do número de vértices. Mesmo assim, o algoritmo conseguiu resolver rapidamente cada instância em tempo inferior a 40 ms com exceção de uma instância (grafo_0040_14) que teve tempo de execução de 10103 ms e não foi colocada no gráfico da figura 2 para melhor visualização da curva de crescimento.

Em comparação com a execução feita em [10], com exceção da instância *grafo_0040_14* os tempos de execução foram semelhantes, porém é notável que a curva realizada por esta implementação é mais acentuada.

5.2 Instâncias Difíceis

Para as instâncias difíceis, somente 12 das 39 instâncias foram resolvidas em tempo inferior ao *timeout* de 5 min. As instâncias das quais foram obtidas as soluções encontram-se na Tabela 2.

Para essas instâncias, é possível notar grande diferença do resultado obtido em [10]. 8 das 12 instâncias resolvidas foram determinadas em menos de 150 ms, porém as outras 4 foram com tempos superiores a 10.000 ms. Visto que o tamanho dos grafos são pequenos, este não é um resultado satisfatório, uma vez que na referência [1] indicou-se um tempo de execução muito menor.

Ambos os algoritmos resolveram dentro do tempo de *timeout* as instâncias 10 e 31, porém, enquanto que nesta execução foram necessários mais de 10.000 ms, em [10] foram resolvidas em menos de 60 ms. De forma análoga, para a instância 33 que foi resolvida em tempo inferior a 1 ms, em [10] foi necessário um tempo de execução maior que 40.000 ms.

Tabela 2: Resultados para Instâncias Difíceis

Instância	Número de Vértices	Tempo(ms)
1	10	<1
2	12	<1
3	17	<1
4	20	15
5	30	<1
10	11	10405
31	18	14071
33	20	<1
35	40	11419
36	40	11434
38	8	140
39	3	<1

5.3 Instâncias Categóricas

Devido a quantidade e diversidade dos grafos de entrada, a análise foi feita para cada categoria separadamente.

5.3.1 *Regular Mesh*

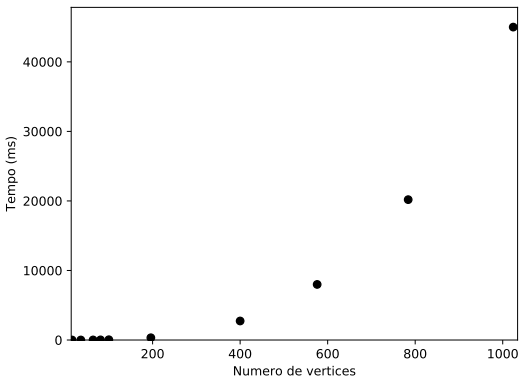
Para os resultados apresentados na figura 3, podemos verificar a curva de crescimento com o aumento do número de vértices e observar que o algoritmo conseguiu fornecer a resposta dentro do limite de *timeout* para todos os casos, mesmo para um grande número de vértices. As instâncias com mais de 1000 vértices obtiveram resposta em menos de 50.000 ms (1/6 *timeout*), o que indica que grafos ainda maiores poderiam ser testados dentro do limite de tempo estabelecido.

Outra observação é a de que o tempo de execução diminui do conjunto 2D para o 3D e o mesmo acontece do conjunto 3D para o 4D. Um estudo mais aprofundado utilizando mais conjuntos e instâncias poderia ser realizado para observar se há alguma relação nesta diminuição do tempo com a variação do tipo do conjunto.

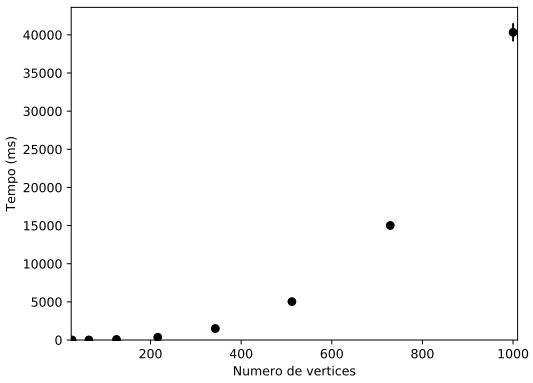
Enquanto que em [10] não se obteve resultados dentro do *timeout* estabelecido, o algoritmo de Mittal aqui testado foi mais eficiente.

5.3.2 *Modified Mesh*

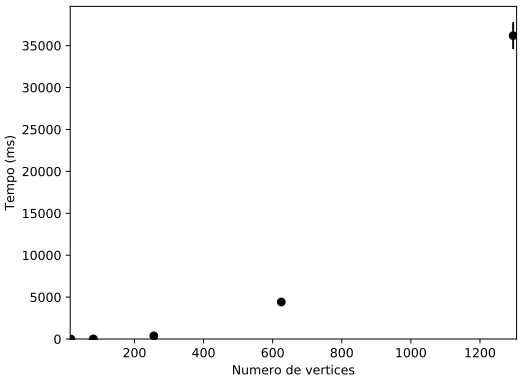
Analisando os resultados para o *Modified Mesh 2D* que se encontram na Figura 4, podemos observar que, assim como ocorreu nos testes da categoria *Regular Mesh*, o algoritmo resolveu todas as instâncias dentro do *timeout* estabelecido e suas maiores instâncias foram resolvidas em tempo inferior a 50.000 ms. Ao aumentar o valor de ρ, nota-se um pequeno aumento



(a) 2D

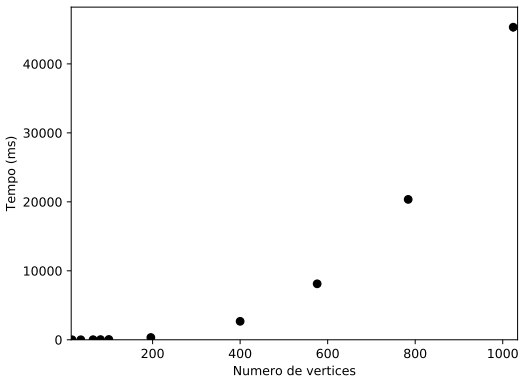


(b) 3D

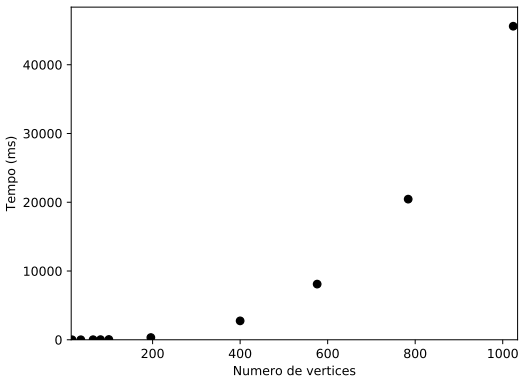


(c) 4D

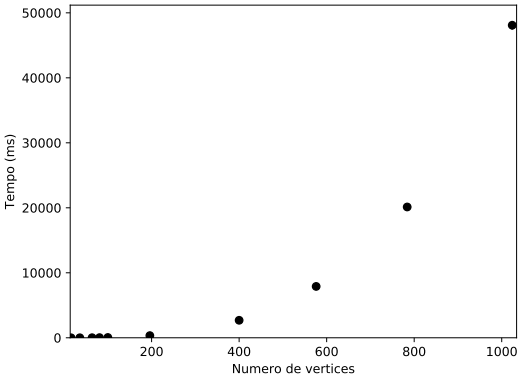
Figura 3: Resultados para instâncias *Regular Mesh*



(a) $\rho = 0.2$



(b) $\rho = 0.4$



(c) $\rho = 0.6$

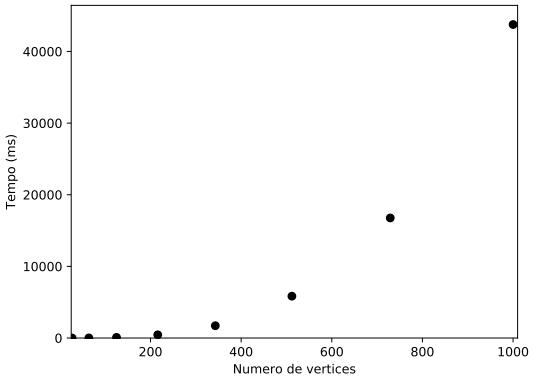
Figura 4: Resultados para instâncias *Modified Mesh 2D*

do valor médio do tempo para as maiores instâncias, porém, ao observar os casos *3D* na Figura 5 e *4D* na Figura 6, não notamos este pequeno aumento e, portanto, não aparentam estar relacionados.

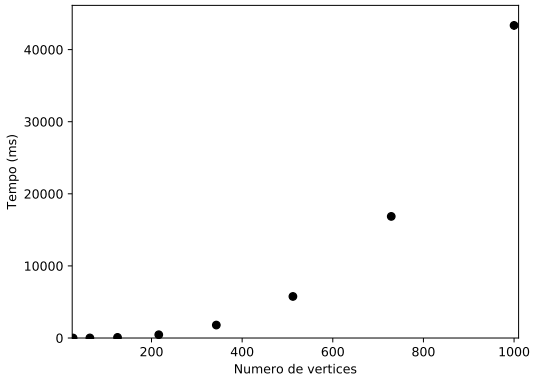
Os resultados *Modified Mesh 3D*, na Figura 5, foram muito próximos do *2D*. Sendo assim, os conjuntos *2D* e *3D* não apresentaram diferença na performance. Já em *Modified Mesh 4D* houve um aumento significativo no tempo de execução, com as instâncias mais difíceis com tempo máximo chegando a valores de 100.000 ms (mesmo assim, inferior ao *timeout* de 5 min).

As principais diferenças dos resultados deste algoritmo e dos obtidos em [10] foram:

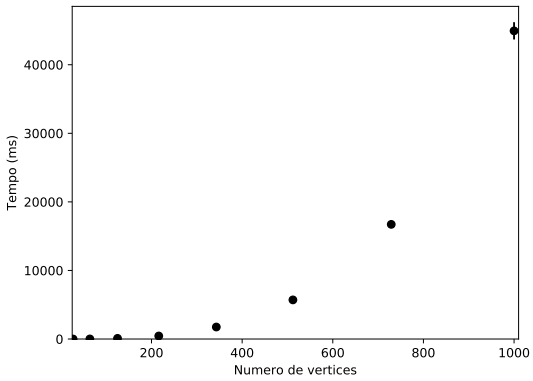
- o algoritmo de Weisfeiler e Lehman não conseguiu dar uma resposta dentro do *timeout* para grafos da categoria *Regular Mesh* enquanto que o algoritmo de Mittal obteve;
- em [10] é observado uma redução significativa do tempo de execução ao aumentar o valor de ρ enquanto que o algoritmo de Mittal não há variação;



(a) $\rho = 0.2$

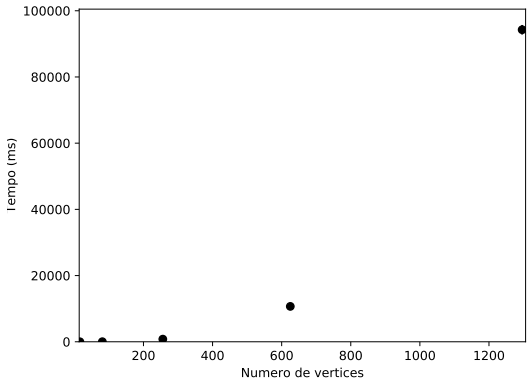


(b) $\rho = 0.4$

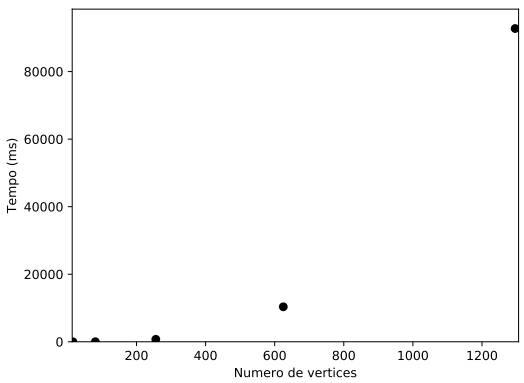


(c) $\rho = 0.6$

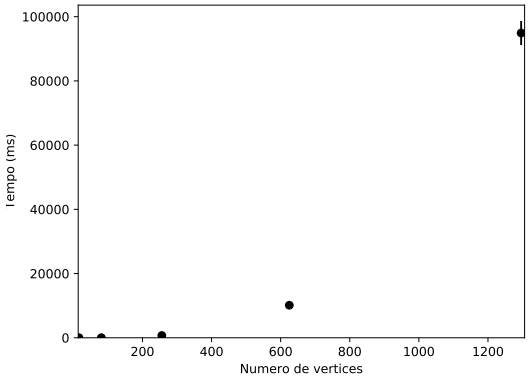
Figura 5: Resultados para instâncias *Modified Mesh 3D*



(a) $\rho = 0.2$



(b) $\rho = 0.4$



(c) $\rho = 0.6$

Figura 6: Resultados para instâncias *Modified Mesh 4D*

- o maior tempo de execução para uma instância foi próximo de 12.000 ms em [10], já neste experimento foi próximo de 100.000 ms;

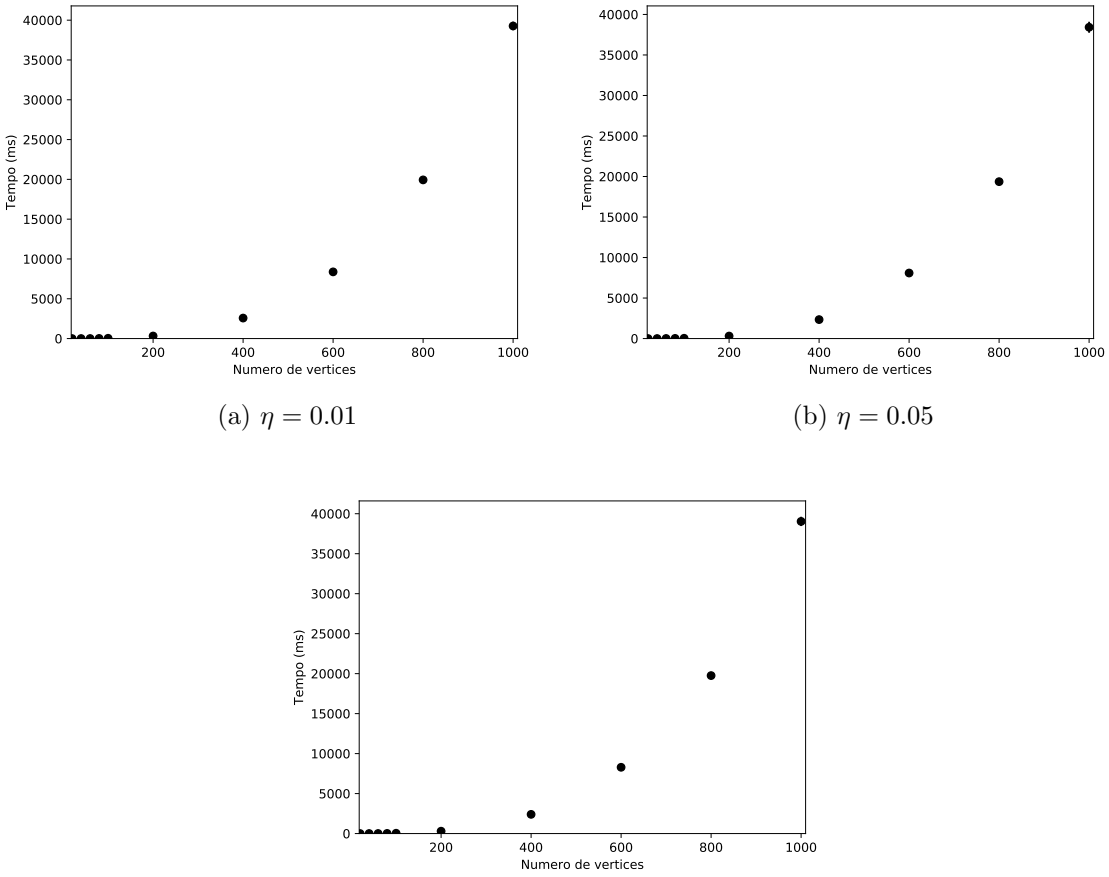
Desta forma podemos concluir que o algoritmo de Weisfeiler e Lehman é muito mais eficiente do que o de Mittal para grafos da categoria *Modified Mesh* enquanto que para *Regular Mesh* ocorre o oposto.

5.3.3 Grafos Conectados Aleatórios

Nesta categoria, pode-se observar pela figura 7 que o tempo de execução foi inferior aos da categoria *Modified Mesh* ao se comparar instâncias com número de vértices parecidos. A variação de η não mostrou variação nos resultados, o que sugere que o algoritmo é estável em relação a densidade de arestas desta categoria.

Novamente vemos que o algoritmo testado teve desempenho inferior, o qual revelou

instâncias com tempo de execução na ordem de 40.000 ms contra os piores casos de 12.000 ms em [10]. Mesmo assim, o algoritmo testado foi capaz de resolver todas as instâncias dentro do *timeout*.



(a) $\eta = 0.01$

(b) $\eta = 0.05$

(c) $\eta = 0.1$

Figura 7: Resultados para instâncias de grafos conectados aleatórios

Como as execuções para *Força Bruta* estouravam o limite do *timeout* para instâncias maiores que 10 vértices [10], o algoritmo de Mittal aqui apresentado mostrou-se mais prático e aplicável possuindo tempos de execução pequenos para algumas categorias de grafos mesmo com número de vértices superiores a 1000.

6 Conclusão

Com a implementação, execução e análise dos resultados, foi possível concluir o objetivo de testar um algoritmo para determinação da existência de isomorfismo entre grafos.

Durante a implementação do algoritmo foi observado um detalhe importante que não

foi indicado na proposta de algoritmo de Mittal que poderia causar o malfuncionamento do mesmo. No passo **(4)** do algoritmo, a subdivisão das classes por subconjuntos de vértices com distância P entre si só pode ser efetuada quando tal subconjunto é **máximo** e **único** em relação a distância P dentro da classe, evitando assim a possibilidade de diferentes subdivisões.

A comparação dos resultados provou que o algoritmo de Mittal é muito mais eficiente que o método *Força Bruta* e que, em comparação ao algoritmo de Weisfeiler e Lehman, ele apresenta uma performance superior na categoria *Regular Mesh* e inferior para as outras. Mas mesmo com o desempenho inferior, conseguiu resolver todas as instâncias dentro de um *timeout* de 5 min, com exceção da categoria de grafos difíceis.

Referências

[1] Graphs for gi testing. http://funkybee.narod.ru/graphs.htm. Acesso: 10-11-2016.

[2] L. Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*, pages 684–697. ACM, 2016.

[3] M. Bayati, J. H. Kim, and A. Saberi. A sequential algorithm for generating random graphs. *Algorithmica*, 58(4):860–910, 2010.

[4] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[5] P. Foggia, C. Sansone, and M. Vento. A database of graphs for isomorphism and sub-graph isomorphism benchmarking. In *Proc. of the 3rd IAPR TC-15 International Workshop on Graph-based Representations*, pages 176–187, 2001.

[6] F. Harary. The determinant of the adjacency matrix of a graph. *Siam Review*, 4(3):202–210, 1962.

[7] J. E. Hopcroft and R. E. Tarjan. Isomorphism of planar graphs. In *Complexity of computer computations*, pages 131–152. Springer, 1972.

[8] J. E. Hopcroft and J.-K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 172–184. ACM, 1974.

[9] R. M. Karp. *Reducibility among Combinatorial Problems*. Springer US, Boston, MA, 1972.

[10] E. R. Mattos and E. C. Xavier. Algoritmos para Teste de Isomorfismo de Grafos. Technical Report IC-PFG-16-08, Institute of Computing, University of Campinas, December 2016.

[11] H. B. Mittal. A fast backtrack algorithm for graph isomorphism. *Information Processing Letters*, 29(2):105 – 110, 1988.

[12] D. C. Schmidt and L. E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *Journal of the ACM (JACM)*, 23(3):433–445, 1976.

[13] J. Turner. Generalized matrix functions and the graph isomorphism problem. *SIAM Journal on Applied Mathematics*, 16(3):520–526, 1968.

[14] L. Weinberg. A simple and efficient algorithm for determining isomorphism of planar triply connected graphs. *IEEE Transactions on Circuit Theory*, 13(2):142–148, 1966.

[15] B. Weisfeiler and A. Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968.