Curso de Ciência da Computação
Universidade Estadual de Mato Grosso do Sul

GERAÇÃO DOS BRICKS CÚBICOS PM – COMPACTOS ATRAVÉS DA OPERAÇÃO DE COLAGEM

Denilson Higino da Silva

Prof. MSc. Delair Osvaldo Martinelli Junior (orientador)

Dourados – MS
2020

GERAÇÃO DOS BRICKS CÚBICOS PM – COMPACTOS ATRAVÉS DA OPERAÇÃO DE COLAGEM

Denilson Higino da Silva

Este exemplar corresponde à redação final da monografia da disciplina Projeto Final de Curso por Denilson Higino da Silva e aprovada pela Banca Examinadora, como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Dourados, 30 de novembro de 2020.

Prof. MSc. Delair Osvaldo Martinelli Junior (orientador)

H541g Higino, Denilson Silva
Geração dos bricks cúbicos PM – compactos através da operação de colagem / Denilson Higino da Silva. – Dourados, MS: UEMS, 2020.
50p.

Monografia (Graduação) – Ciência da Computação – Universidade Estadual de Mato Grosso do Sul, 2020.
Orientador: Prof. MSc. Delair Osvaldo Martinelli Junior.

1. Algoritmo 2. Teoria de grafos 3. Experimento I. Martinelli Junior, Delair Osvaldo II. Título

CDD 23. ed. - 005.1

Curso de Ciência da Computação
Universidade Estadual de Mato Grosso do Sul

GERAÇÃO DOS BRICKS CÚBICOS PM – COMPACTOS ATRAVÉS DA OPERAÇÃO DE COLAGEM

Denilson Higino da Silva

Novembro de 2020

Banca Examinadora:

Prof. MSc. Delair Osvaldo Martinelli Júnior (Presidente)
Área de Computação – UEMS

Prof. Dr. Evandro Cesar Bracht
Área de Computação – UEMS

Prof. Dr. Osvaldo Vargas Jaques
Área de Computação – UEMS

Esse trabalho é dedicado aos meus pais,
que me apoiaram desde sempre em
minha jornada pelo ensino, o que me
trouxe até aqui.

AGRADECIMENTOS

Quero começar agradecendo ao meu orientador, por ter tido a paciência de me ajudar a concluir este trabalho me auxiliando por todo o processo.

Agradeço também a todos os professores desse curso que, com cada matéria, me deram uma base preciosa de conhecimento que me ajudará a se tornar um ótimo programador no futuro.

Agradeço aos meus amigos que, com certeza, tornaram essa jornada mais simples de caminhar.

E um último agradecimento especial a um ex aluno da UEMS, Filipe Névola, que com a iniciativa Code for the Win, mudou minha vida e de outros para melhor, usando programação para abrir portas para o mundo.

RESUMO

Um objeto de grande interesse na otimização combinatória é o politopo dos emparelhamentos perfeitos de um grafo G. Neste trabalho implementamos um algoritmo para a geração da família dos bricks cúbicos PM – compactos. Este algoritmo usa a operação denominada de *colagem* de grafos para encontrar os grafos pertencentes a essa família. Ao longo do texto mostramos as definições e conceitos, assim como detalhes de implementação.

SUMÁRIO

1 INTRODUÇÃO

Esse trabalho é baseado no projeto de doutorado intitulado Brick's cúbicos PM-compactos escrito por Martinelli [1].

Um *politopo* é uma região contida em $\mathbb{R}^n$ que é resultante da intersecção de um conjunto de semiespaços. Um objeto de grande interesse na otimização combinatória é o *politopo dos emparelhamentos perfeitos* de um grafo G. Um grafo G é *PM – compacto* se a diferença simétrica entre cada par dos emparelhamentos perfeitos de G é um único *ciclo* [7]. Sendo assim, se um grafo é *PM – compacto*, quaisquer dois vértices do *politopo dos emparelhamentos perfeitos* são adjacentes. O problema tratado neste trabalho é a caracterização de *bricks cúbicos PM – compactos* através de um experimento de implementação ao qual usará as definições apresentadas adiante em sua lógica computacional para atingir o objetivo.

Um subconjunto M do conjunto de arestas E de um grafo G é chamado de *emparelhamento em G* se nenhum de seus elementos são adjacentes. Um emparelhamento M satura um vértice v, e v é dito M-saturado, se alguma aresta de M é incidente em v. Se todos os vértices de G forem M – saturados, o *emparelhamento é perfeito* [3]. Na imagem A da figura 1.1 temos um emparelhamento perfeito. Veja que M(A) = {(1, 2), (4, 5), (0, 3)} incide em todos os vértices de G sem ser adjacente.

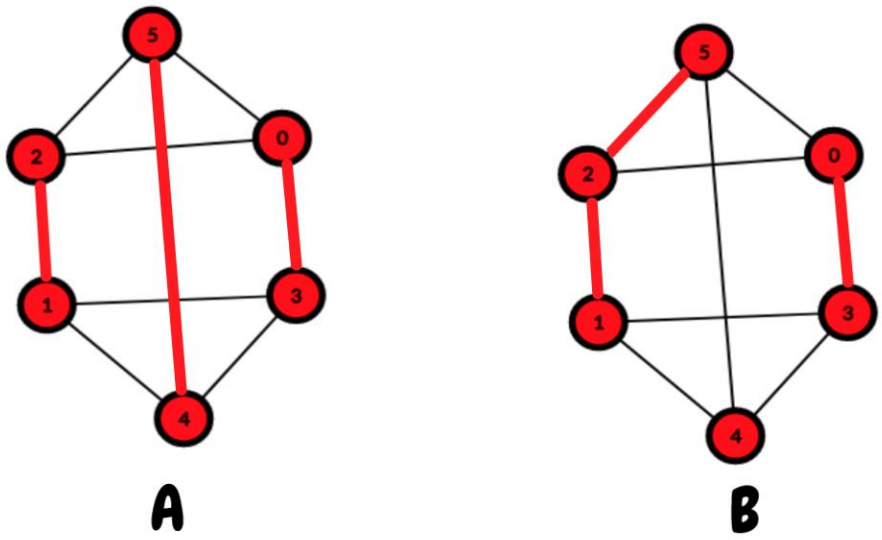


Figura 1.1 - No grafo A, as arestas em vermelho mostram um *emparelhamento perfeito* M(A) = {(1, 2), (4, 5), (0, 3)}. No grafo B não temos um emparelhamento perfeito já que as arestas (1, 2) e (2, 5) são adjacentes.

Um grafo é *coberto por emparelhamentos* se ele possui pelo menos uma aresta e todas as arestas do grafo estão contidas em ao menos um *emparelhamento perfeito*. Por conta disso, podemos restringir o estudo dos grafos PM – compactos aos grafos cobertos por emparelhamentos.

Existe um procedimento denominado decomposição em cortes justos [6], que decompõe um grafo coberto por emparelhamentos em grafos cobertos por emparelhamentos menores chamados de *bricks* e *braces*. Seja G um grafo livre de cortes justos não triviais, se G é bipartido então ele é chamado de *brace*, senão ele é um *brick.* B*ricks* e *braces* são peças fundamentais para a construção de quaisquer grafos cobertos por emparelhamentos.

Essa decomposição se aplica aos grafos PM – compactos. Ou seja, a caracterização de *bricks* e *braces* PM – compactos podem ser estendida a uma caracterização de qualquer grafo PM – compacto.

Wang et al. [5] caracterizam os grafos PM – compactos bipartidos e uma outra classe denominada quase bipartidos. Foi provado por eles que o único grafo bipartido, PM – compacto, simples e com grau mínimo três é o brace K_{33} (figura 1.2), e, por consequência direta, o K_{33} é também o único brace PM – compacto.

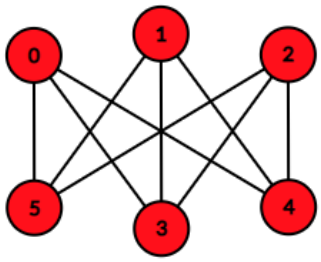


Figura 1.2 - Grafo K_{33}

Existem duas subclasses para os bricks chamadas de sólidos e não sólidos. Um brick é *sólido* se não possui cortes separadores não triviais (capítulo 2). Foi concluído por *Carvalho et al.* [2] que os únicos bricks sólidos PM – compactos são o grafo S_8 (figura 1.3 A) e a família infinita denominada roda ímpar. Uma roda *ímpar* W_K onde *k* é ímpar, é o grafo composto por um ciclo C de tamanho *K* e um vértice, denominado *hub*, adjacente a todos os vértices de C. Na figura 1.3 B temos a roda W_5, onde seu *hub* é o vértice 0 e seu ciclo é representado em vermelho.

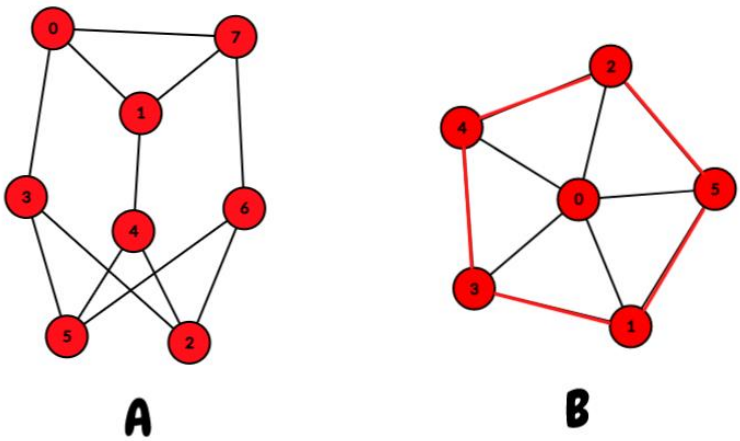


Figura 1.3 - O grafo A representa Grafo S_8. O grafo B, chamado de W_5, representa uma roda de 6 vértices, tendo seu ciclo mostrado em vermelho.

O conjunto de todos os bricks PM-compactos é infinito, basta observar a família das rodas impares, que é infinita e pertence a este conjunto. Nosso objetivo principal é implementar o algoritmo que encontrará o subconjunto dos *bricks cúbicos PM – compacto.* Acreditamos ser uma família finita com exatamente 27 grafos.

Este trabalho está divido em 3 partes. Primeiramente discutiremos aspectos teóricos a respeito do problema. No capítulo 3 mostraremos detalhes de implementação do algoritmo. E por fim, no capítulo 4 mostraremos os dados processados e os grafos gerados.

2 CONCEITOS E DEFINIÇÕES

Seja G um grafo e $X \subseteq V(G)$. Chamamos $C(X)$ o *corte* associado à X, ou seja, o conjunto das arestas que possuem um de seus vértices em X e o outro em seu complemento $\overline{X} = V(G) - X$.

Digamos agora que G é um grafo coberto por emparelhamentos e C é um corte de G. Uma *C - contração* de G é um grafo obtido a partir de G pela contração de X ou $\overline{X}$ a um único vértice. Assim, um corte C possui duas *C - contrações*. Um corte C de G é dito *separador* se as duas C – contrações resultar em grafos cobertos por emparelhamentos. A figura 2.1 ilustra um exemplo de contração. No grafo **A** podemos ver as arestas que fazem parte do corte de **A** pela linha que atravessa essas arestas. O grafo **B** é o resultado da contração do lado direito do grafo A.

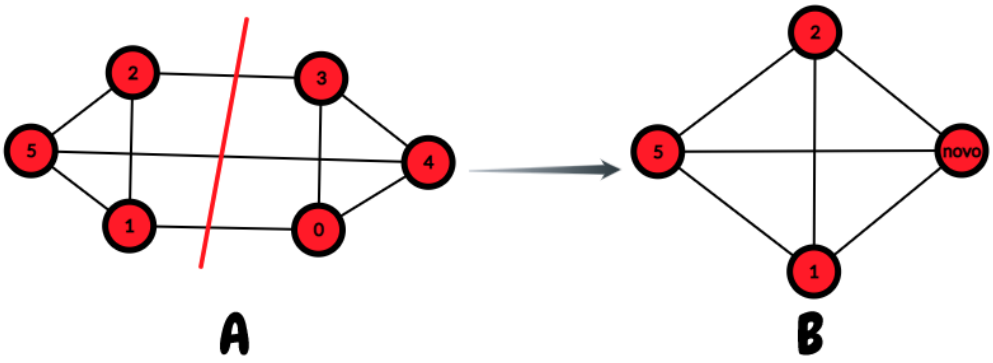


Figura 2.1 - Contração em A.

Como é possível perceber na figura 2.1, convergimos todos os vértices de um lado do grafo, digamos o $\overline{X}$, para um único novo vértice, e então, conectamos todas as arestas que fazem parte do corte neste novo vértice.

Seja G um grafo, chamamos um corte C, de G, de *justo* se $|C \cap M| = 1$, para todo emparelhamento perfeito M de G. É fácil notar que todo corte justo, de G, é um corte separador de G, mas o contrário não é verdadeiro. Assim, se um corte justo C é não trivial, as duas C – contrações são grafos cobertos por emparelhamentos com menos vértices do que G. Se uma das C – contrações resultar em algum grafo onde exista um corte justo não trivial, então podemos repetir a operação obtendo um par de grafos, cobertos por emparelhamentos, ainda menor.

Uma colagem é um processo inverso a uma contração. Por Martinelli [1], um grafo G é uma colagem de dois grafos G_1 e G_2 se possui um corte C onde G_1 e G_2 são isomorfos às duas C – contrações de G. Essa operação é representada por $G = (G_1 \Theta G_2)_{u,v}$, onde *u* é o vértice de G_1 resultante da contração de $G_2 - v$, em G, e *v* é o vértice de G_2 resultante da contração de $G_1 - u$, em G. Nesse caso dizemos que *u* e *v* são os vértices de contração de G_1 e G_2, respectivamente.

Todos os grafos PM – compactos e cobertos por emparelhamentos podem ser obtidos da colagem de grafos PM – compactos de tamanho menor. O Lema 1 nos mostra isto.

Lema 1. (*Carvalho et al.* [5]). *Seja G um grafo coberto por emparelhamentos, e seja C um corte justo, em G. Se G é PM – compacto. Então as duas C – contrações também são grafos PM – compactos.*

O Lema 1 também nos mostra que para caracterizarmos os grafos PM – compactos e cobertos por emparelhamentos é necessário caracterizarmos os grafos PM – compactos e cobertos por emparelhamentos livres de cortes justos.

Seja G um grafo coberto por emparelhamentos livre de cortes justos não triviais. Se G é bipartido então ele é chamado de *brace*, senão, ele é um *brick.* Um brick é *sólido* se não possui cortes separadores não triviais, Martinelli [1].

O Lema 2 nos mostra que podemos construir bricks não sólidos e PM – compactos através da colagem de bricks sólidos e PM – compactos.

Lema 2. (*Carvalho et al.* [5]). *Seja G um grafo coberto por emparelhamentos, e seja C um corte separador, em G. Se G é PM – compacto, então as duas C – contrações também são grafos PM – compactos.*

O Lema 2 é a principal propriedade que o algoritmo explora: Colagens sucessivas entre bricks PM – compactos, ou o brace K_{33}, para encontrarmos a família de bricks cúbicos PM – compactos.

2.1 Algoritmo de colagem de grafos e algoritmo de classificação de PM – Compactos

Dois algoritmos são fundamentais para este projeto: O algoritmo de colagem e o algoritmo que verifica se um grafo é PM – compacto. Nesta sessão, discutiremos ambos os algoritmos e então detalharemos suas implementações no capítulo 3.

Dado dois grafos para a função de colagem, G e H, a primeira ação executada é selecionar um vértice de cada grafo para executarmos a colagem sobre tais vértices. A única restrição no momento de escolher tais vértices é que eles tenham o mesmo grau. Digamos que o vértice v_1 seja o vértice selecionado em G e v_2 seja o vértice selecionado em H. O próximo passo é remover o vértice selecionado de cada grafo e após a remoção criamos um novo grafo F a partir de G e H, sem os vértices selecionados. No novo grafo F, F.G representa o conjunto de vértices e arestas em F que originalmente formam o grafo G. e F.H representa o conjunto de vértices e arestas em F que originalmente formam o grafo H. Então, no novo grafo, conectamos um dos vértices de F.G que tem uma aresta a menos, em relação ao G original, com um dos vértices de F.H que tem uma aresta a menos, em relação ao H original. Repetimos essas conexões para todos os vértices de F.G e F.H que tenham uma aresta a mesmo. Chamamos estas novas arestas criadas de arestas de corte. Abaixo, temos o Pseudocódigo 1 que representa o processo descrito.

```
algoritmo Colagem(G, H):
        v1 =  seleciona_vértice(G)
        v2 = seleciona_vértice(H)
        se v1.grau != v2.grau então
            retorne [ ]
        vértices_com_menos_uma_aresta_em_G = remove_vértice(G, v1)
        vértices_com_menos_uma_aresta_em_H = remove_vértice(H, v2)

        F = gerar_novo_grafo(G, H)
        index = 0
         para cada vértice v em vértices_com_menos_uma_aresta_em_G faça:
            F.conecte_vértices(v, vértices_com_menos_uma_aresta_em_H[index])
            index += 1
```

Pseudocódigo 1 – Exemplo em alto nível da primeira parte de uma colagem.

Neste ponto temos o primeiro passo da colagem concluído. Geramos o primeiro grafo colado a partir de outros dois e sabemos quais são nossas arestas de corte. O passo seguinte é gerar novos grafos permutando todas arestas de corte e então eliminando todos os grafos que são isomorfos aos gerados anteriormente. Seja P um grafo gerado por uma das permutações. Se P não é isomorfo a nenhum dos grafos já aceitos e é PM – compacto, o aceitamos como um novo grafo que será usado para ser colado com outros grafos e que potencialmente é um dos grafos que estamos procurando, se caso ele for 3 – regular (todos os vértices dele tem grau 3, caracterizando um grafo cúbico).

```
algoritmo Colagem(G, H):
    [ ... ]
    grafos_da_permutação = F.permutar_arestas_corte()

    grafos_aceitos = []
    para cada grafo P em grafos_da_permutação faça:
        se P é isomorfo a qualquer grafo em grafos_aceitos então:
            continue
        se verifica_PM_compacto(P) é verdadeiro então:
            grafos_aceitos.insere(P)
  retorna grafos_aceitos
```

Pseudocódigo 2 – Exemplo em alto nível da segunda parte de uma colagem.

Agora podemos discutir o segundo algoritmo mais importante para projeto, sendo ele o algoritmo que verifica se um grafo é PM – compacto ou não, no Pseudocódigo 2 chamado de **verifica_PM_compacto**. Dado um grafo para a função **verifica_PM_compacto**, o primeiro passo é adquirir um *array* com todos os emparelhamentos do grafo. Uma vez em que temos todos os

emparelhamentos do grafo, o próximo passo é contar quantos ciclos cada par de emparelhamentos possui. Se todos os pares de emparelhamento possuem um, e apenas um ciclo, este grafo é PM – compacto. No Pseudocódigo 3 temos a representação desse algoritmo.

```
algoritmo verifica_PM_compacto(G):
    emparelhamentos = adiquirir_emparelhamentos(G)
    para emparelhamento e1 em emparelhamentos faça:
        para emparelhamento e2 em emparelhamentos faça:
            se e1 == e2 então:
                continue
            se quantidade_ciclos(e1, e2) != 1 então:
                retorne falso
    retorne verdadeiro
```

Pseudocódigo 3 – Exemplo em alto nível do algoritmo que verifica se um grafo é PM – compacto.

Com o Pseudocódigo 3 temos a representação do algoritmo que verifica se um grafo é PM – compacto. No capítulo 3 iremos discutir em mais detalhes esta função olhando como encontramos os emparelhamentos e como contamos os ciclos.

3 DESENVOLVIMENTO

Utilizamos como ferramenta computacional a linguagem de script Python 3 [9]. Logo, utilizaremos as estruturas definidas nesta linguagem, como Listas, Dicionários, Tuplas, etc.

3.1 BRICKS SÓLIDOS E O K_{33}

Os grafos são armazenados em memória conforme vão sendo gerados. A estratégia foi colocar os grafos agrupados, em listas, por número de vértices. Uma das primeiras funções chamadas no código é a função que inicia um dicionário de listas. São colocadas nesse dicionário 14 chaves numeradas de 4 a 30, iterando de 2 em 2, todos os grafos iniciados automaticamente são marcados como sólidos. Abaixo, temos uma representação deste dicionário:

```
Listas = {
        4: [todos grafos gerados com 4 vértices],
        6: [todos grafos gerados com 6 vértices],
        ...
        30: [todos grafos gerados com 30 vértices],
}
```

Cada uma dessas listas é iniciada com pelo menos um *brick ou brace (no caso do K_{33}) PM-compacto*. Das listas de número 6 até a 30 nos iniciamos com uma roda ímpar com o número de vértices baseado no tamanho de cada lista.

Um exemplo de uma roda ímpar, além do da figura 1.3 B, é a roda ímpar W_7 com 8 vértices mostrado na figura 3.1.

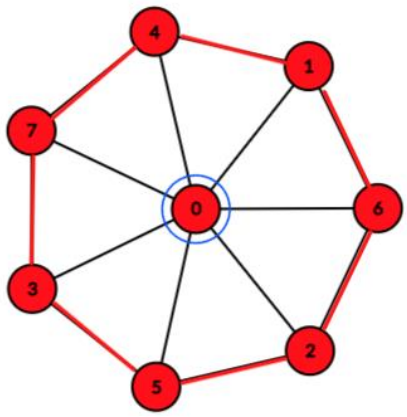


Figura 3.1 - Roda ímpar W_7 de 8 vértices. Em vermelho é representado o ciclo presente na roda e, circulado em azul, temos o *hub* da roda.

Todas elas têm um padrão que foi explorado no algoritmo para gerar uma para cada lista. Para criar uma de 8 vértices, por exemplo, basta colocar 7 vértices com grau 2 e um vértice conectado a todos os outros. Abaixo temos a função que cria uma roda:

```
def createWheel(size):
        g = {"numberOfVertex": size, "bindVertex": 0, "isSolid": True, "graph": {}}
        for i in range(1, size):
                nextV = i + 1
                if nextV == size:
                        nextV = 1
                        beforeV = i - 1
                if beforeV == 0:
                        beforeV = size – 1
                # conectando o vértice i ao vértice 0
                connectVertex(g["graph"], (i, 0))
                # conectando o vértice i ao vértice próximo
                connectVertex(g["graph"], (i, nextV))
                # conectando o vértice i ao vértice anterior
                connectVertex(g["graph"], (i, beforeV))
        return getGraph(g)
```

Trecho de código 1 – Função responsável por criar uma roda.

Existem mais quatro grafos usados para iniciar as listas. Para a lista de 4 vértices temos o K_4 (figura 3.2), para a de 6 temos o K_{33} (figura 1.2), para a de 8 temos o S_8 (figura 1.3), e para a de 12 temos o grafo de Wang (figura 3.3).

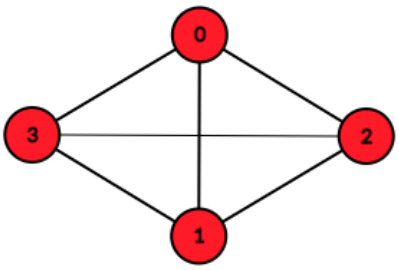


Figura 3.2 - grafo K_4

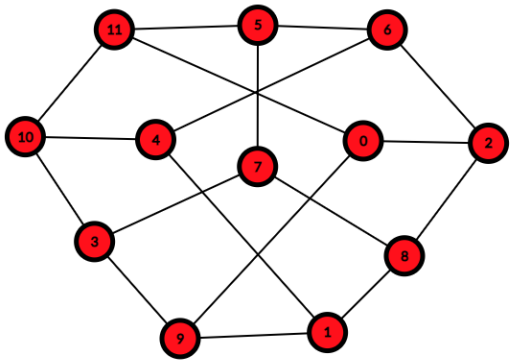


Figura 3.3 - grafo de Wang

3.2 SELECIONANDO LISTAS PARA COLAGEM

Para descobrir qual lista colar com qual, temos uma simples lógica implementada. Como exemplo, usaremos a lista de 10 vértices. Iremos precisar usar as listas cuja a soma menos 2 é igual ao número de vértices da lista que estamos calculando. Por exemplo 4 + 8 – 2 = 10 e 6 + 6 – 2 = 10. O valor -2 nessa equação se dá pelo fato de que quando estamos colando 2 grafos, 2 vértices são excluídos no processo.

Em código, essa lógica está implementada como no Trecho de código 2. Primeiro geramos um *array* com os valores de 4 a 8, que são as três listas precedentes a de 10:

```
# calculating = 10

rangeValues = list(range(4, calculating, 2))

# > [ 4, 6, 8 ]
```

Trecho de código 2 – Gerando array com os números de vértices das listas que serão usadas na colagem.

Uma vez gerado este *array*, podemos sistematizar a lógica acima para colar as listas. Começamos selecionando o primeiro e o último item do *array*, e se caso o número de itens for ímpar (como é este caso), na última iteração teremos que colar a lista desse item com ela mesma:

```
rangeValues = list(range(4, calculating, 2))
while len(rangeValues):
    if len(rangeValues) == 1:
        solo = rangeValues.pop(0)
        permuteQueues(queues[solo], queues[solo], [...])
        continue
    first = rangeValues.pop(0)
    last = rangeValues.pop(-1)
    permuteQueues(queues[first], queues[last], [...])

# iteração 1: first = 4 e last = 8
# iteração 2: solo = 6
```

Trecho de código 3 – Usando o array com os números dos vértices que serão usados para permutar as listas.

Com os pares de listas selecionados, podemos seguir com a colagem em entre os grafos de lista.

3.3 COLANDO GRAFOS

Dada duas listas, colamos os grafos sólidos de uma lista com todos os grafos da outra. No momento de filtrar uma das listas para selecionar apenas os sólidos, sempre ignoramos o K_{33}. Pelo fato de o K_{33} ser um brace, ele precisa ter um processo especial em uma colagem. Iremos discutir este caso na sessão 3.3.1.

Vamos usar um exemplo para ficar mais fácil de entender a lógica. Digamos que recebemos as listas 4 e 6 para calcular os grafos da lista 8:

```
def permuteQueues(first, lastQueue, queues, alreadyAcceptedGraphs,
calculating):
        # filtramos todos os sólidos da lista 4 (nesse caso apenas o K4)
        firstSolids = [x for x in first if x.isSolid and not x.isK33]
        computeLists(firstSolids, lastQueue, queues,
        alreadyAcceptedGraphs, calculating)

        # filtramos todos os sólidos da lista 6
        # Aqui teremos o K33 e o W5 (roda ímpar de 6 vértices)
        lastSolids = [x for x in lastQueue if x.isSolid and not x.isK33]
        computeLists(lastSolids, first, queues, alreadyAcceptedGraphs,
        calculating)
```

Trecho de código 4 – Selecionando grafos sólidos para a colagem.

O próximo será a iteração sobre as listas de sólidos e não sólidos, que foram selecionados no trecho de código 4:

```
1 def computeLists(solids, notSolids, queues, alreadyAcceptedGraphs, calculating):
2       calculateK33(solids[0], queues, alreadyAcceptedGraphs, calculating)
3       for g in solids:
4               for h in notSolids:
5                       if g.isK33 or h.isK33:
6                               continue
7                       selfPermute([g, h], queues, alreadyAcceptedGraphs)
```

Trecho de código 5 – Iterando sobre as listas de grafos para computar a colagem.

É fácil representar graficamente o que acontecerá quando o algoritmo executar o código entre as linhas 3 e 7 do Trecho de código 5. Vamos usar nosso exemplo das listas 4 e 6. Para a primeira chamada da função **computeLists** dentro da função **permutateQueues**, teremos a colagem entre os grafos das listas 4 e 6, mostradas na figura 3.4. Na figura 3.4 é mostrado um grafo que não citamos ainda. O grafo $\overline{C6}$ é um grafo gerado da colagem de dois K_4 e é o primeiro brick cúbico PM-compacto gerado quando executamos o algoritmo. Dentro da função **computeLists** é chamado a função **calculateK33** que será explicada sessão 3.3.1.

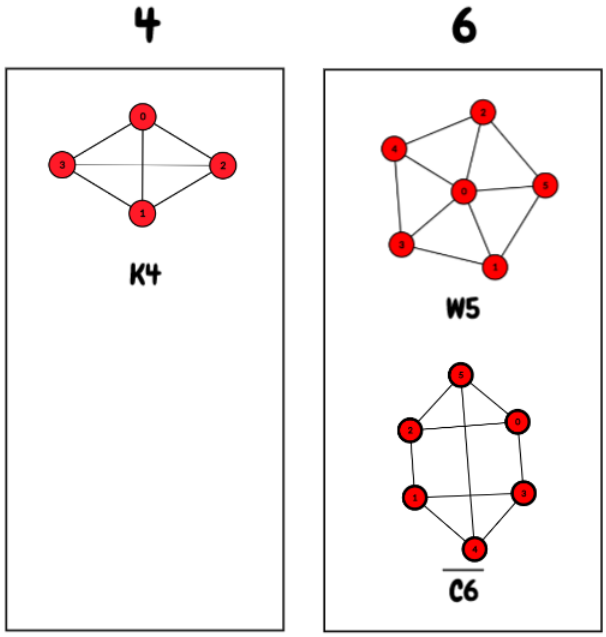


Figura 3.4 - Grafos das listas 4 e 6 que serão usados para gerar a lista 8.

Selecionado os dois grafos para a colagem, podemos chamar a função **selfPermute.** Dado dois grafos para esta função, a primeira ação tomada por ela realizar a colagem entre os grafos. Esse processo é separado em uma outra função chamada **bindByOrbits**.

Para colar dois grafos, normalmente teríamos que colar todos os vértices de um grafo com todos os vértices do outro. Isso tem um grande custo computacional já que para colar um grafo de 10 vértices, por exemplo, teríamos 10! colagens (ou seja, chamaríamos a função de colagem 3.628.800 vezes) e a maior parte dos grafos gerados na colagem são isomorfos uns aos outros. Para contorna esta situação, usamos uma função especial da biblioteca pynauty [4] chamada **autgrp**. Dado um grafo para essa função, dentre os vários valores retornados, ela retornará um *array* que classifica cada um dos vértices no grafo em grupos, baseado grau de cada vértice. Esse *array* é chamado de órbitas (*orbits*).

Vamos usar como exemplo a roda impar da figura 3.1. Quando chamamos a função **autgrp** para aquele grafo, nosso retorno será:

Vértices: [0, 1, 2, 3, 4, 5, 6, 7]

Órbitas: [0, 1, 1, 1, 1, 1, 1, 1]

Como podemos ver na relação acima, apenas o vértice 0 tem um número de órbita diferente. Isso porque ele é o único no vértice que se conecta a 7 outros vértices, todos os outros se conectam a apenas 3 vértices. Com esta informação, já sabemos quais vértices usar. Basta pegar um vértice de cada grupo. Então nesse caso, ao invés de colarmos em 8 vértices, precisamos colar em apenas 2.

Uma vez que temos nossas órbitas, e sabemos quais vértices de cada grafo usar, podemos discutir sobre a colagem em si. Graficamente uma colagem é representada na figura 3.5:

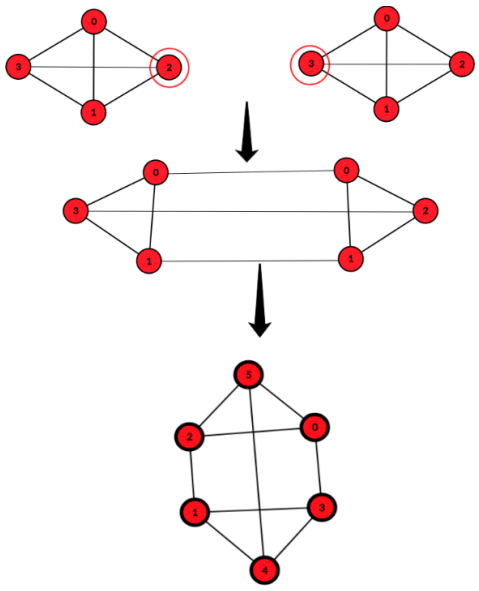


Figura 3.5 - Colagem entre dois K_4 sem as devidas permutações.

Como podemos ver, graficamente, é um processo simples. Quando colando dois K_4, escolhemos dois vértices, nesse caso o vértice 2 do grafo do lado esquerdo e o vértice 3 do grafo do lado direito. O próximo passo é remover ambos os vértices de ambos grafos e então podemos conectar os dois grafos, da forma representada na figura 3.5, e por fim renomeamos os vértices, gerando um grafo que já citamos acima, o $\overline{C6}$.

No código é preciso replicar esse comportamento. O primeiro passo é adquirir as adjacências dos grafos envolvidos na colagem e filtrar, das adjacências, os vértices que estamos removendo:

```
# pegando as adjacências
graphAdjacency1 = graph1.getAllVertexAdjacency()
graphAdjacency2 = graph2.getAllVertexAdjacency()
# removendo os vértices dos grafos
keyList1 = filterList(graphAdjacency1.keys(), graph1.bindVertex)
keyList2 = filterList(graphAdjacency2.keys(), graph2.bindVertex)
```

Trecho de código 6

Depois mapeamos os dois grafos que estamos colando, G e H, em um grafo um novo grafo que terá (**G.número_vértices + H.número_vértices - 2**) vértices:

```
newGraphSize = graph1.vertexAmount + graph2.vertexAmount - 2
for i in range(newGraphSize):
        if i < graph1.vertexAmount - 1:
                elem = keyList1.pop()
                map1[i] = elem
        else:
                elem = keyList2.pop()
                map2[i] = elem
```

Trecho de código 7

Esse laço garante que no novo grafo vamos manter as conexões originais (sem os vértices que foram removidos). Com o novo grafo, podemos seguir para o último passo e realizar as conexões que estão faltando para completarmos nosso grafo.

A ação tomada é descobrir quais vértices estão com o grau menor do que ele possuía em seu grafo original. Se estivéssemos colando dois K_4 como na figura 3.5, os vértices marcados do grafo da esquerda seriam o 0, 3 e 1, e do grafo da direita seriam o 0, 1, e 2. Uma vez que temos esses vértices, apenas conectamos os vértices de um lado com os vértices do outro lado:

```
# Conectado os vértices que tem o grafo menor que seu original
for key, degree in gDegrees.items():
        if key < graph1.vertexAmount - 1:
                if degree < graph1Degrees[map1[key]]:
                        diffs[key] = graph1Degrees[map1[key]] – degree
        else:
                if degree < graph2Degrees[map2[key]]:
                diffs2.append(key)
# Conectado os vértices que tem o grafo menor que seu original
for key, diff in diffs.items():
        if key < graph1.vertexAmount - 1:
                p = diffs2.pop()
                for i in range(diff):
                        g.connect_vertex(key, [p] + g.adjacency_dict[key])
```

Trecho de código 8 – Mapeando grafo.

3.3.1 TRATANDO O GRAFO K_{33}

A função **calculateK33** é a função responsável por tratar todas as colagens com o grafo K_{33}. Como foi discutido acima na sessão 3.3, o grafo K_{33} é um grafo especial que precisa de uma colagem diferente. Pelo fato do grafo K_{33} ser um Brace, se colarmos um grafo sólido em uma das partições do K_{33}, o grafo resultante não será um brick, pois haverá um corte justo e por tanto não é um brick ou brace, como podemos constatar na figura 3.6. Mas se um outro grafo sólido for colado na partição contrária ao primeiro grafo colado, voltamos a ter um brick.

Quando chamamos a função **calculateK33** são necessárias 3 informações: Saber qual o número da lista que estamos calculando, saber qual grafo vamos colar na primeira partição, e por fim, qual grafo iremos colar na segunda partição. A única informação que não temos, e que precisamos descobrir, é a última. Para tal, colocamos os valores que conhecemos na seguinte equação:

$$\mathbf{N = 6 + X + Y - 4}$$

Onde **N** é número da lista que estamos gerando, **6** é a quantidade de vértices do K_{33}, **X** é o número de vértices do grafo que vamos colar na primeira partição, **Y** é o número de vértices do grafo que estaremos colando na segunda partição (e é o grafo que não conhecemos ainda), e por fim, **- 4** representa número de vértices que estamos perdendo no processo.

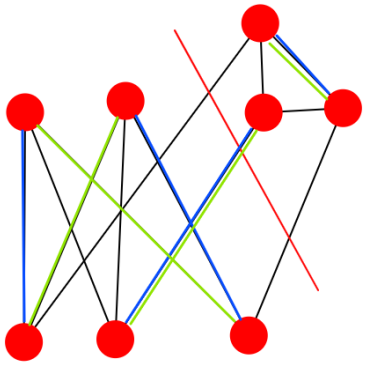


Figura 3.6 - O grafo acima mostra a colagem de um K_4 (parte superior da linha vermelha) em um K_{33} (parte inferior da linha vermelha). Em azul e verde temos a representação de dois emparelhamentos perfeitos. A linha em vermelho caracteriza um corte justo, já que seu conjunto de arestas possui uma aresta de cada emparelhamento perfeito, ou seja, o grafo acima não é um brick.

Usando um exemplo, vamos assumir que estamos calculando os grafos para a lista de 14 vértices e o nosso primeiro item na lista de sólidos é o S_8 (figura 1.3). Quando chamamos a função temos:

14 = 6 + 8 + Y – 4

Resolvendo a equação, encontramos que o valor de **Y** é 4. Tendo esse valor em mãos, nosso próximo passo é colar o grafo S_8 na primeira partição do K_{33} e depois colar todos os grafos sólidos da lista 4 nesse novo grafo gerado pela colagem do S_8 com o K_{33}.

Podemos ver esse mesmo exemplo a seguir no trecho de código simplificado dessa função:

```
def calculateK33(g, queues, alreadyAcceptedGraphs, calculating):
        # g = S8 então X = 8
        X = g.vertexAmount
        # Y = 14 - 6 + 4 – 8 => Y = 4
        Y = calculating - 6 + 4 – X
        # Colagem entre o K33 e o S8
        firstGraph = bindGraph(K33_GRAPH, g)
        # Para todos os grafos sólidos na lista 4, colamos esse novo grafo
        # gerado
        for graph in queues[Y]:
                if graph.isSolid:
                        bondedGraph = bindGraph(firstGraph, deepGraph)
                        selfPermute([bondedGraph, deepGraph], […])
```

Trecho de código 9 – Versão simplificada da função calculateK33

3.4 PERMUTANDO VÉRTICES DE UM GRAFO

A colagem entre dois grafos pode gerar vários outros. Para cada um desses novos grafos, vamos executar permutações entre todas as arestas envolvidas nas conexões dos dois grafos colados. A figura 3.7, mostra quais as arestas que serão permutadas do grafo que colamos na figura 3.5, ou seja, iremos permutar as arestas de corte do grafo.

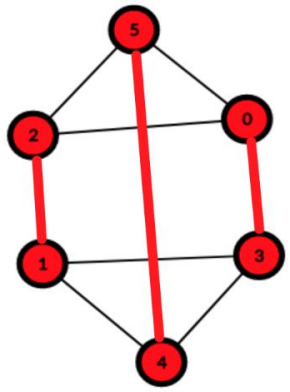


Figura 3.7 - Arestas de corte do grafo $\overline{C6}$

A função responsável pelas permutações é a **twistEdges**. Começamos criando um *array* de permutações com os vértices de uma das extremidades das arestas de corte do grafo. Uma vez que o *array* é adquirido, o próximo passo será gerar novos grafos baseados nas permutações e então verificar se esse grafo é PM compacto e, se sim, verificamos se ele não é isomorfo a nenhum dos outros grafos já aceitos. Caso ele não seja isomorfo a nenhum dos outros grafos já aceitos para aquela lista, o aceitamos.

```
def twistEdges(graph, alreadyAccepted):
        newAcceptedGraphs = []
        connections = graph.connections
        # baseado nas conexões nós fazemos a permutação
        keys = list(connections.keys())
        # baseado nas conexões nós fazemos a permutação
        perm = list(permutations(connections.values()))
        # pra cada conjunto de vértices permutados nós criamos um
        # novo grafo
        for vertices in perm:
                g = deepcopy(graph)
                for k, v in zip(keys, vertices):
                        adj = g.getAllVertexAdjacency()
                        g.connect_vertex(...)
                # verificamos se o novo grafo é PM compacto
                if not isPMCompact(g):
                        continue
                # verificamos se o novo grafo já isomorfo a algum já aceito
                # caso não, nós o aceitamos
                for ag in alreadyAccepted + newAcceptedGraphs:
                        if isomorphic(g, ag):
                                break
                else:
                        newAcceptedGraphs.append(g)
        return newAcceptedGraphs
```

Trecho de código 10 – Função twistEdges

A função usada para criar a lista de vértices permutados é a *permutations*, de uma das bibliotecas padrões do Python, itertools [8]. A função que checa se um grafo é isomorfo a outro é a **isomorphic**, outra função da biblioteca pynauty [4].

Nessa função, por conta das permutações, é gera um número extremamente grande, passando da casa dos trilhões já que a quantidade de permutações resultante é fatorial ao número de vértices. A permutação de um grafo com 22 vértices, por exemplo, resulta em 1.1240007e+21 combinações. Por esse motivo é difícil passa de 22 vértices, isso poderia levar meses.

Uma vez que um grafo é aceito nessa função, significa que achamos um dos grafos que estávamos procurando. Mas antes fecharmos o assunto temos que discutir como reconhecemos um grafo PM – compacto.

3.5 VERIFICICANDO SE UM GRAFO É PM – COMPACTO

Seja G um grafo, se G é PM - compacto então para cada par de emparelhamentos perfeitos temos um e apenas um ciclo. Explorando esta propriedade, a função responsável por verificar se um grafo é PM – compacto foi criada. Na sessão 3.5.1 encontramos todos os emparelhamentos em um grafo. Na sessão 3.5.2 contamos os ciclos existentes em cada par de emparelhamentos. Caso seja 1, classificamos o grafo como PM – compacto.

3.5.1 ENCONTRANDO EMPARELHAMENTOS PERFEITOS

Um grafo não possui apenas um emparelhamento perfeito, e precisamos de todos eles. Para isso foi usado uma estratégia recursiva. Vamos usar o $\overline{C6}$ para esse exemplo. Os passos descritos a seguir poderão ser vistos na figura 3.8. Começamos fixando um vértice, vamos dizer o vértice 0 do nosso grafo de exemplo. Depois selecionamos uma das arestas desse vértice. Agora com a aresta selecionada, apenas a salvamos, removemos os dois vértices relacionados a ela e seguimos fixando mais vértices, selecionando mais arestas, as salvando e removendo os vértices relacionados as arestas até que não tenhamos mais vértices. Quando os vértices acabam, voltamos recursivamente olhando para o último vértice fixado e selecionando a próxima aresta conectado a ele.

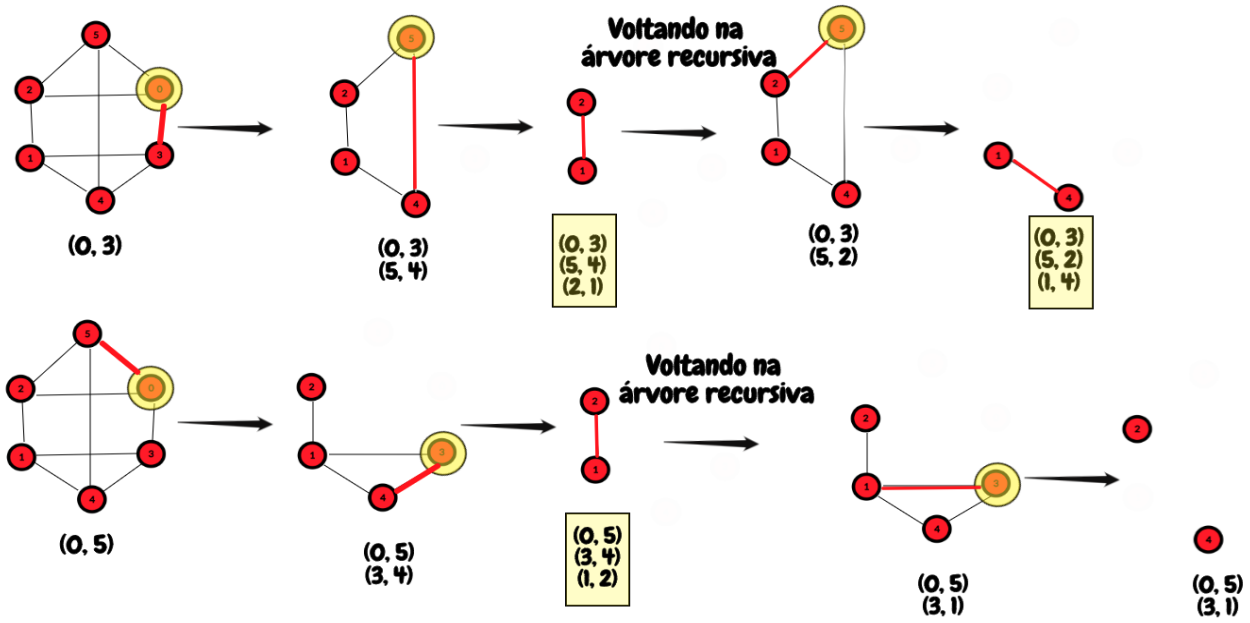


Figura 3.8 - Encontrando emparelhamentos.

Repassando o que está acontecendo na figura 3.8, começamos fixando o vértice 0 e selecionando a aresta (0, 3), salvamos essa aresta e removemos os dois vértices dela do grafo. No próximo passo, fixamos o vértice 5, selecionamos a aresta (5, 4), a salvamos junto com a anterior e removemos seus vértices do grafo. Restando apenas dois vértices no grafo, não precisamos fixar ninguém, apenas salvamos a aresta a nossa pilha e teremos nosso primeiro emparelhamento perfeito encontrado. Então voltamos um nível na nossa árvore recursiva, onde o vértice fixo é o 5. Olhamos para a próxima aresta do vértice 5 que não foi usada

ainda, (5, 2), salvamos ela em uma nova pilha contendo apenas a aresta (0, 3) já que uma outra aresta nesse nível da árvore significa outro emparelhamento. Removemos os vértices dessa última aresta salva e vamos para o próximo passo onde novamente temos apenas dois vértices. Salvamos a aresta desses dois vértices e descobrimos um outro emparelhamento perfeito.

Como olhamos todas as arestas do vértice 5, que foi o último fixado, voltamos a olhar para o vértice 0 começando selecionando a aresta (0, 5), salvando-a em uma pilha zerada (já que 0 é o vértice do topo), e removendo os dois vértices dessa aresta. Depois fixamos o vértice 3, selecionando a aresta (3, 4) a salvamos junto com a anterior e removemos seus vértices do grafo. No próximo passo, selecionamos a última aresta, a salvamos e descobrimos mais um emparelhamento perfeito. Voltamos um nível na nossa árvore, selecionamos a aresta que ainda não foi usada do vértice 3, colocamos ela na pilha e removemos os dois vértices dessa aresta. E agora restaram dois vértices que não estão conectados um ao outro, o que significa que aqui não temos um emparelhamento perfeito.

Nesse ponto voltamos na árvore recursivamente até o nível do vértice zero novamente. Como o vértice zero não tem mais nenhuma aresta para ser verificada, terminamos o processo para o zero aqui e fixaríamos o próximo vértice para ser o topo da árvore. Esse processo é repetido para todos os vértices.

3.5.2 ENCONTRANDO CICLOS

Agora, dado um par de emparelhamentos perfeitos, precisamos verificar se o grafo possui um e apenas um ciclo para esse par.

Vamos discutir o método para encontrar tais ciclos analisando a figura 3.9. Nela temos em verde o emparelhamento perfeito [(0, 3), (5, 2), (1, 4)] e em roxo temos o emparelhamento perfeito [(0, 3), (5, 4), (2, 1)]. No código, olhamos apenas os vértices que fazem sentido olhar, baseado no grau de cada um deles, logo o vértice 0 e o 3 não serão analisados. Então, fixando o vértice 5, começamos com a aresta (5, 2) em azul e vamos marcando os vértices visitados, assim temos a sequência (5, 2, 1, 4, 5). Em seguida olhamos para a próxima aresta do vértice 5, em rosa, para encontrarmos a sequência (5, 4, 1, 2, 5).

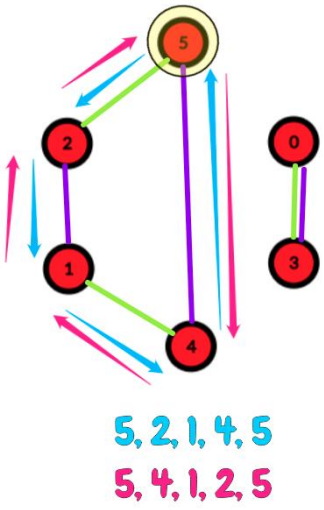


Figura 3.9 - Encontrando ciclo entre dois emparelhamentos perfeitos.

Se o primeiro número da sequência é igual ao último, temos um ciclo. Nesse caso o ciclo rosa e o azul são iguais, então consideramos apenas 1. O próximo passo será fixar outro vértice e repetir o mesmo processo até visitar todos os vértices. E é necessário olhar todos os vértices do grafo sejam fixados já que existe a chance de acharmos mais de um ciclo. Vamos analisar a figura 3.10 como exemplo.

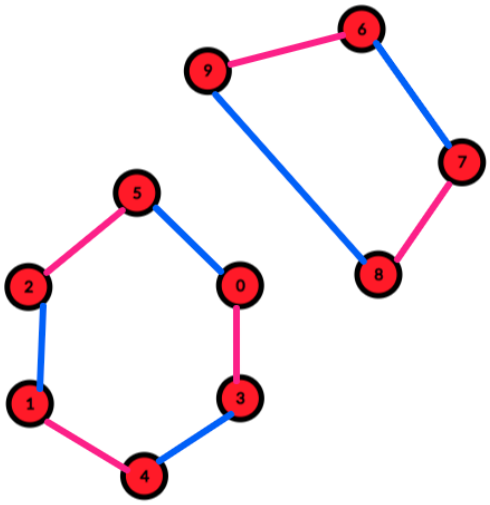


Figura 3.10 - Grafo com dois ciclos

Como podemos ver na figura 3.10, existem dois emparelhamentos perfeitos no grafo, um em rosa e o outro em azul e claramente podemos ver dois ciclos distintos no grafo, logo, esse grafo não pode ser caracterizado como PM – compacto.

Um último para aceitarmos o grafo que passa pela função **isPMCompact** é apenas verificar se o grafo é 3-regular. Essa verificação é extremamente simples, precisamos apenas certificar todos os vértices do grafo têm grau 3.

Nesse ponto, em que sabemos se o grafo é PM – compacto ou não, retornando para a função **twistEdges** e se o grafo for PM – compacto, não isomorfo a nenhum dos já aceitos e 3-regular, o registramos como um dos grafos que estamos procurando. E mesmo que ele não seja 3-regular, nos ainda o mantemos como um dos aceitos, já que este grafo pode ser usado para gerar outros grafos.

Todo o código detalhado até este ponto forma o algoritmo que usamos para encontramos os bricks cúbicos PM – compactos, mas existe muito mais do código e que é importante para que tudo funcione. Esse código pode ser encontrado no Gitub: https://github.com/denihs/tcc/blob/master/pynauty-0.6.0/tccSource/main.py.

4 DADOS COMPUTADOS

O código rodou por 25 dias, gerando e analisando trilhões de grafos. Infelizmente não conseguimos ir adiante de 22 vértices por conta da nossa capacidade computacional e da quantidade de dados gerados, mas felizmente, conseguimos retirar dados suficientes e condizentes com o que foi previsto no manuscrito de Martinelli [1]. Encontramos a família de 27 bricks cúbicos PM – compactos, variando entre 4 e 18 vértices.

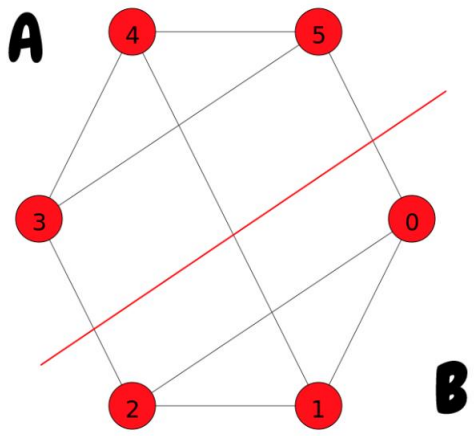
A baixo são listados os 25 grafos gerados pelo algoritmo. Os outros dois bricks cúbicos que não estão na lista são o K_4 (figura 3.2) e o grafo de Wang (figura 3.3).

Todos os dados computados também poderão ser encontrados junto ao código no GitHub (https://github.com/denihs/tcc/blob/master/pynauty-0.6.0/tccSource).

Para cada brick cúbico gerado, iremos mostrar também os dois grafos (ou 3 no caso de uma colagem envolvendo o K_{33}), que colados, o geraram. Na descrição da imagem também citamos qual a identificação interna que o grafo ganhou enquanto estava sendo gerado pelo código. O significado é simples, vamos usar de exemplo o **6-2**. O **6** representa o número de vértices do grafo, nesse caso 6, e o **2** representa quando esse grafo foi calculado quando o código estava computando para a lista com o seu número de vértices. Nesse caso o **2** significa que o grafo **6-2** foi o terceiro grafo gerado enquanto a lista de 6 vértices estava sendo computada. Terceiro porque nossa contagem dentro da lista começa do zero.

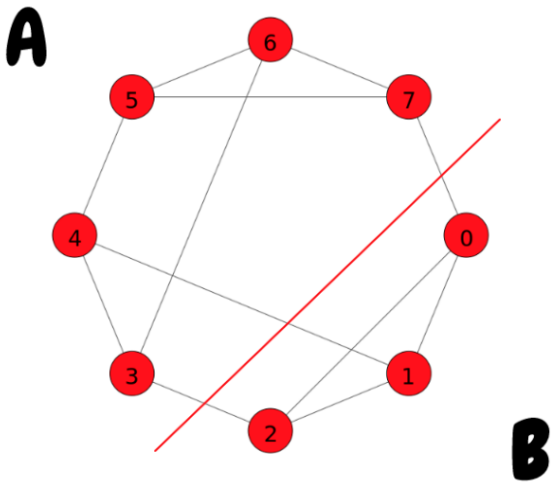
Também será indicado para cada grafo as arestas de corte, resultantes da operação de colagem, por uma linha vermelha. Desta forma ficará fácil identificar os grafos que fazem parte da colagem.

1. Da colagem de dois de dois K_4 (figura 3.2) geramos o grafo 1 ($\overline{C6}$).



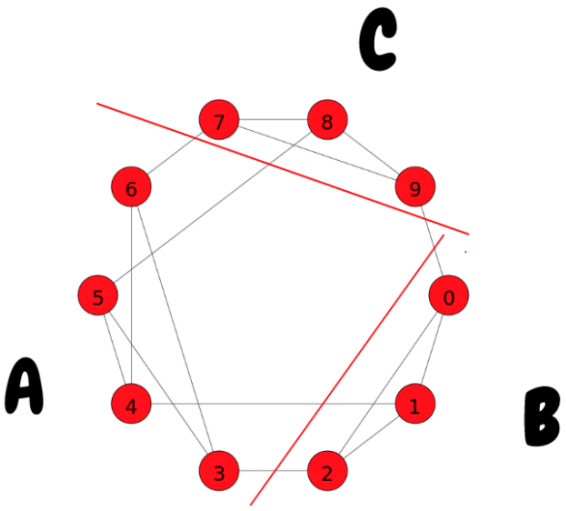
Grafo 1 – Internamente chamado de 6-2. Em cada um dos lados, A e B, temos um K4.

2. Da colagem entre um K4 e o Grafo 1, geramos o Grafo 2.



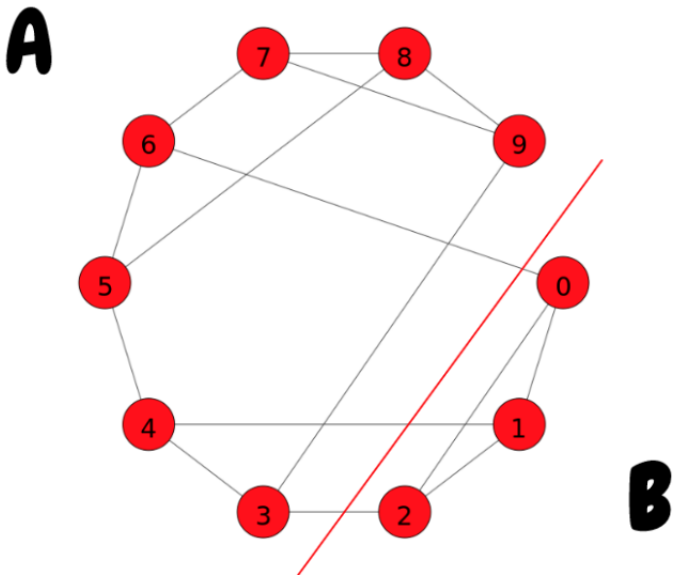
Grafo 2 – Internamente chamado de 8-3. No lado A temos o Grafo 1. No lado B temos o K_4.

3. Da colagem entre um K_{33} e dois K_4 (um em cada partição do K_{33}) geramos o Grafo 3.



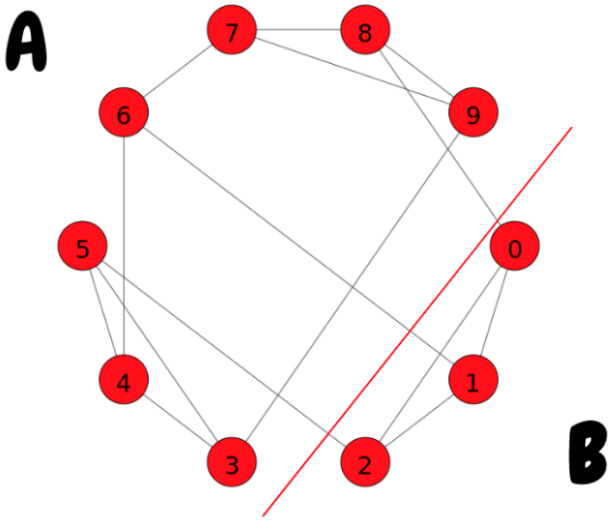
Grafo 3 - Internamente chamado de 10-1. No lado A temos o K33. No lado B temos um dos K4 e no C o outro. Perceba que cada um dos K4 está em uma partição diferente do K33.

4. Gerado da colagem entre o K_4 e o Grafo 2.



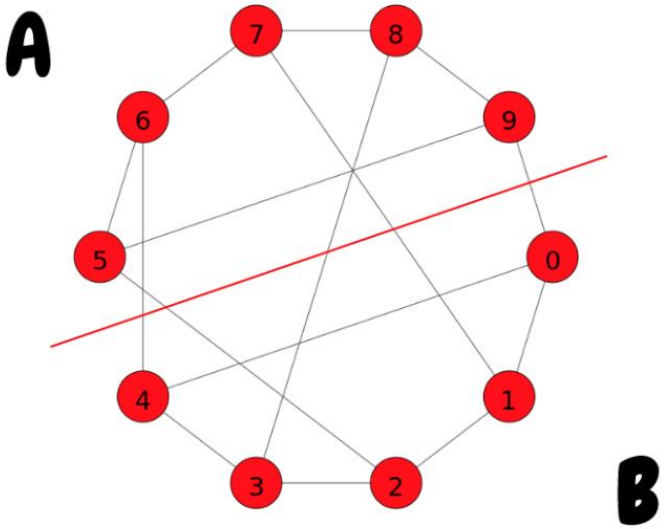
Grafo 4 - Internamente chamado de 10-7. No lado A temos o Grafo 2 e no lado B temos o K_4.

5. Gerado da colagem entre o K_4 e o Grafo 2, temos o Grafo 5. Se repararmos, o grafo 4 também foi gerado pela mesma colagem do grafo 5, mas como explicado no capitulo 3, uma colagem entre dois grafos gera muitos outros, uns isomorfos e outros não. O grafo 4 e o 5 estão nesta lista porque eles não são isomorfos um ao outro.



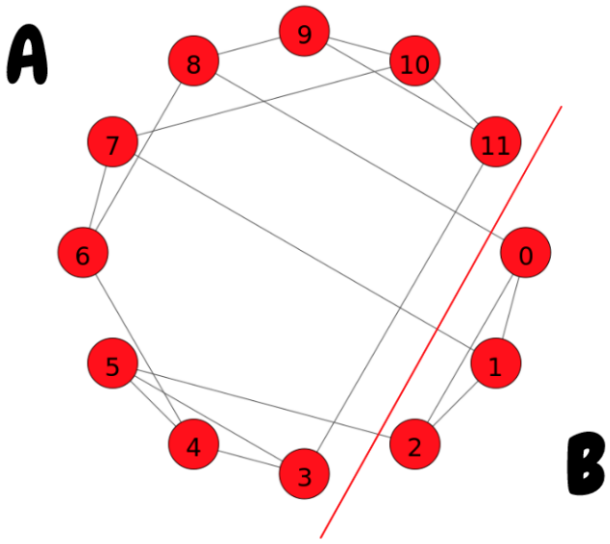
Grafo 5 - Internamente chamado de 10-8. No lado A temos o Grafo 2 e no lado B temos o K4.

6. Gerado da colagem entre dois grafos W_5 (roda ímpar 5).



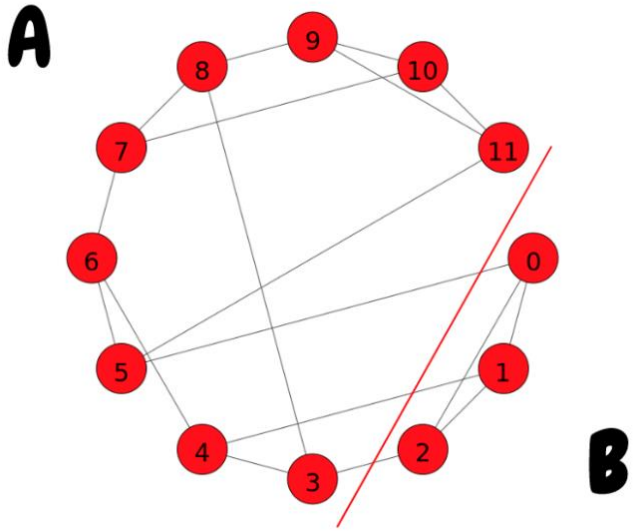
Grafo 6 - Internamente chamado de 10-9. Em cada um dos lados temos um W5.

7. Gerado a partir da colagem entre o K_4 e o Grafo 3.



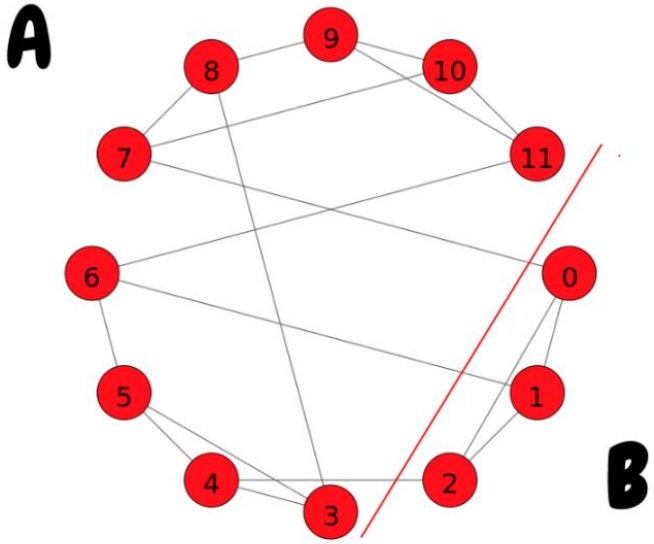
Grafo 7 - Internamente chamado de 12-3. No lado A temos o Grafo 3 e no lado B temos o K4.

8. Gerado a partir da colagem entre o K_4 e o Grafo 4.



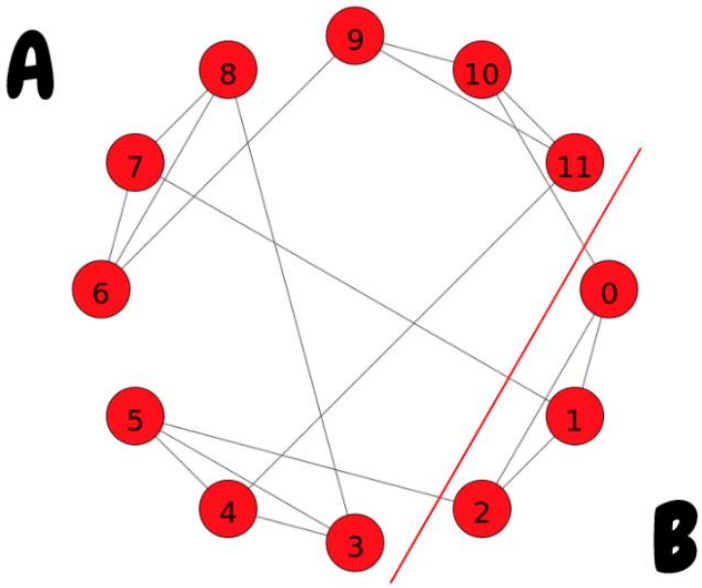
Grafo 8 - Internamente chamado de 12-13. No lado A temos o Grafo 4 e no lado B temos o K4.

9. Gerado a partir da colagem entre o K_4 e o Grafo 4.



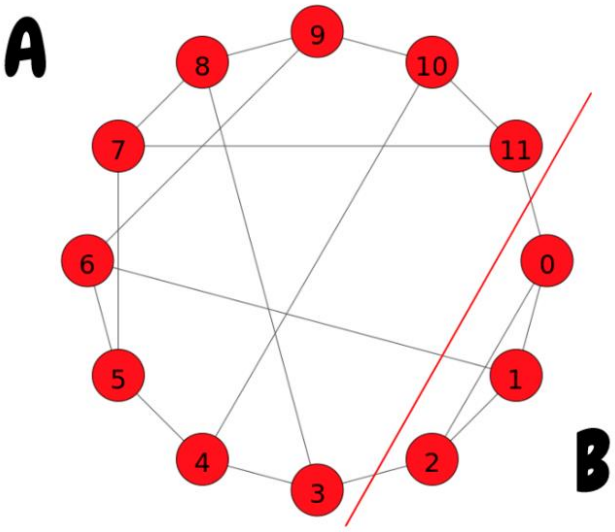
Grafo 9 - Internamente chamado de 12-14. No lado A temos o Grafo 4 e no lado B temos o K4.

10. Gerado a partir da colagem entre o K4 e o Grafo 5.



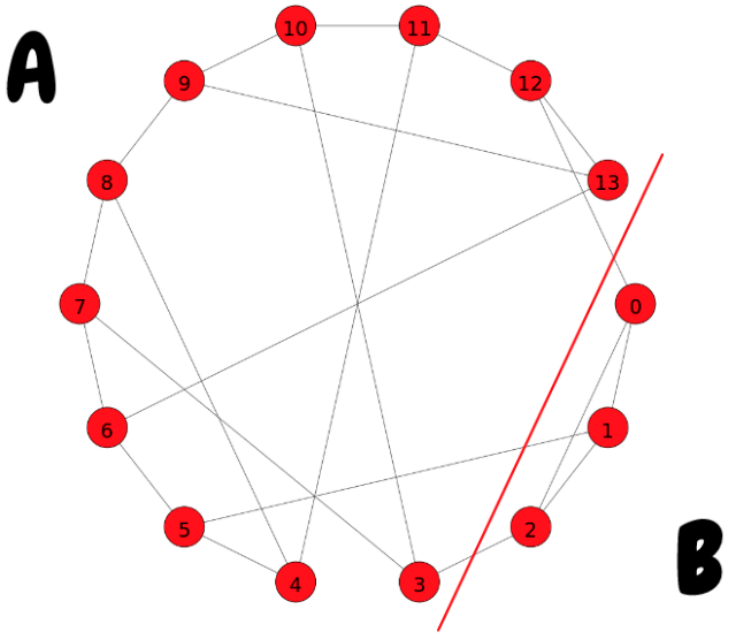
Grafo 10 - Internamente chamado de 12-15. No lado A temos o Grafo 5 e no lado B temos o K4.

11. Gerado a partir da colagem entre o K_4 e o Grafo 6.



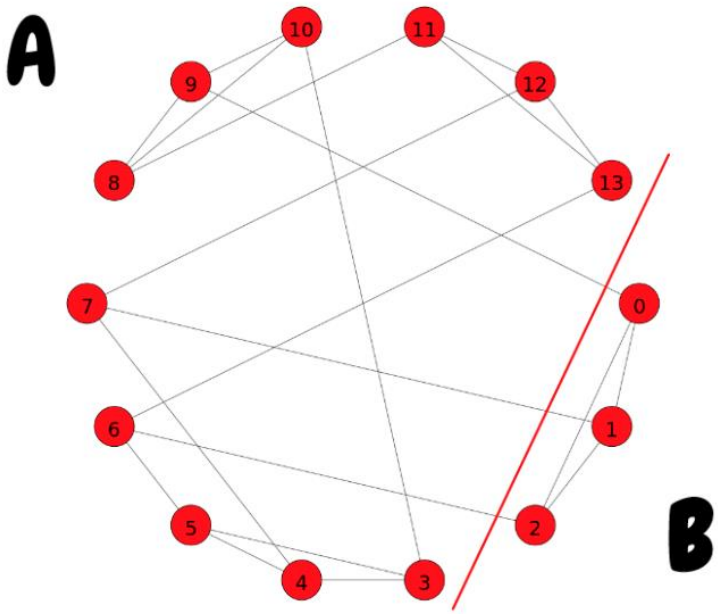
Grafo 11 - Internamente chamado de 12-16. No lado A temos o Grafo 6 e no lado B temos o K4.

12. Gerado a partir da colagem entre o K_4 e o grafo de Wang.



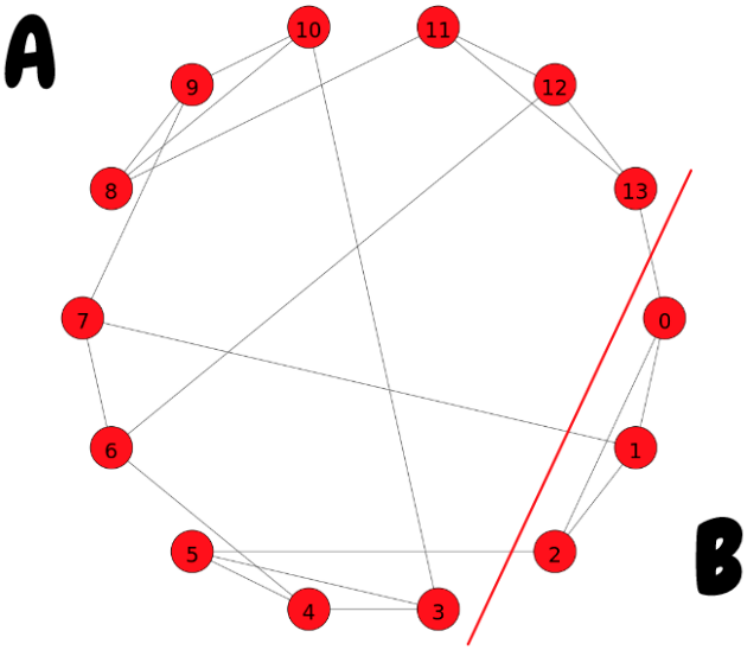
Grafo 12 - Internamente chamado de 14-1. No lado A temos o grafo de Wang e no lado B temos o K4.

13. Gerado a partir da colagem entre o K_4 e o Grafo 7.



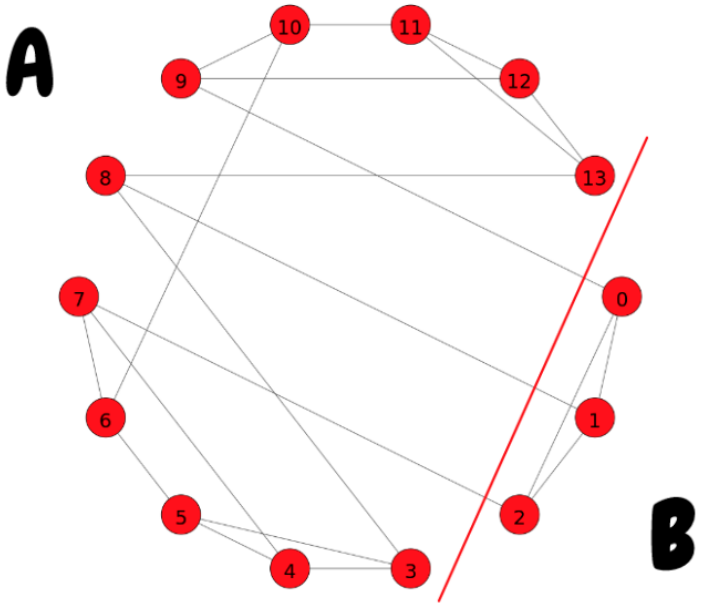
Grafo 13 - Internamente chamado de 14-8. No lado A temos o Grafo 7 e no lado B temos o K4.

14. Gerado a partir da colagem entre o K_4 o Grafo 7.



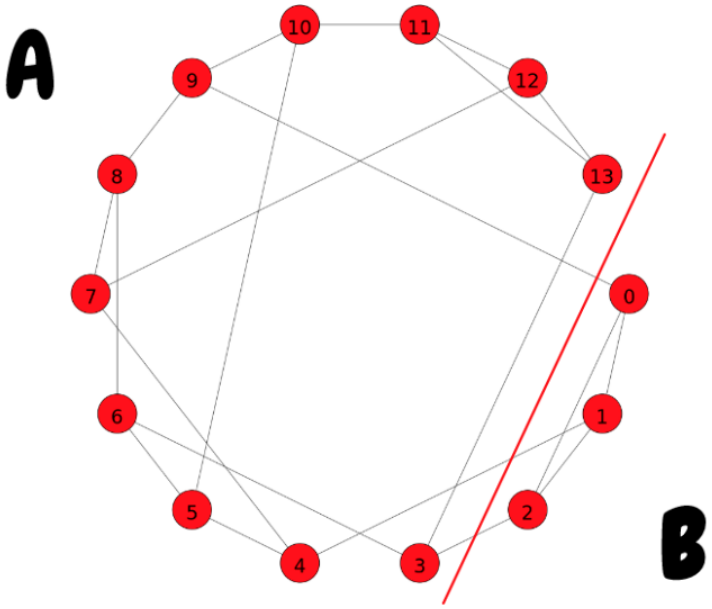
Grafo 14 - Internamente chamado de 14-9. No lado A temos o Grafo 7 e no lado B temos o K4.

15. Gerado a partir da colagem entre o K_4 e o Grafo 8.



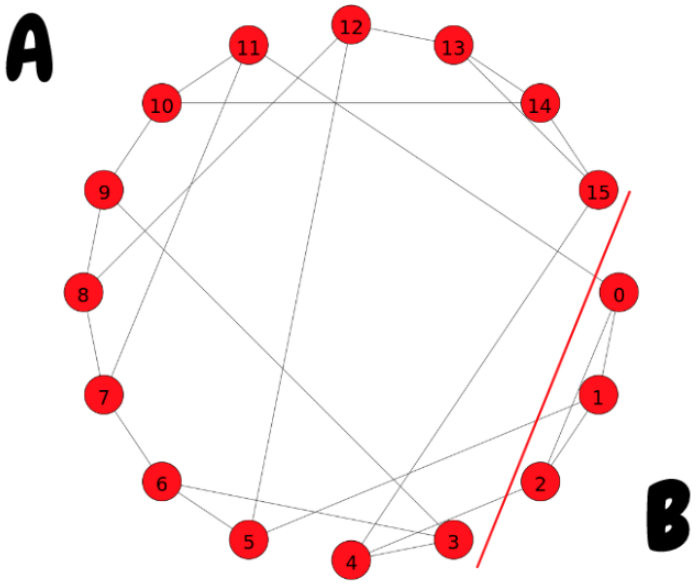
Grafo 15 - Internamente chamado de 14-25. No lado A temos o Grafo 8 e no lado B temos o K4.

16. Gerado a partir da colagem entre o K_4 e o Grafo 11.



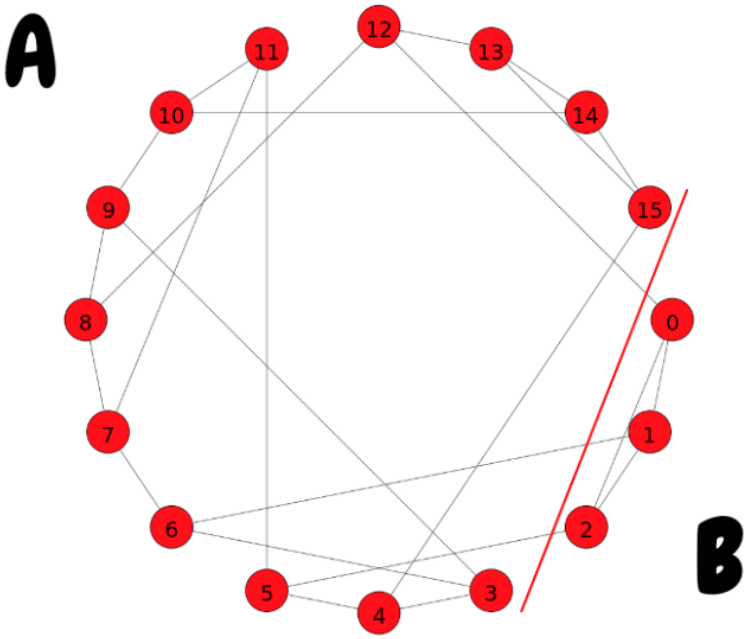
Grafo 16 - Internamente chamado de 14-26. No lado A temos o Grafo 11 e no lado B temos o K4.

17. Gerado a partir da colagem entre o K_4 e o Grafo 12.



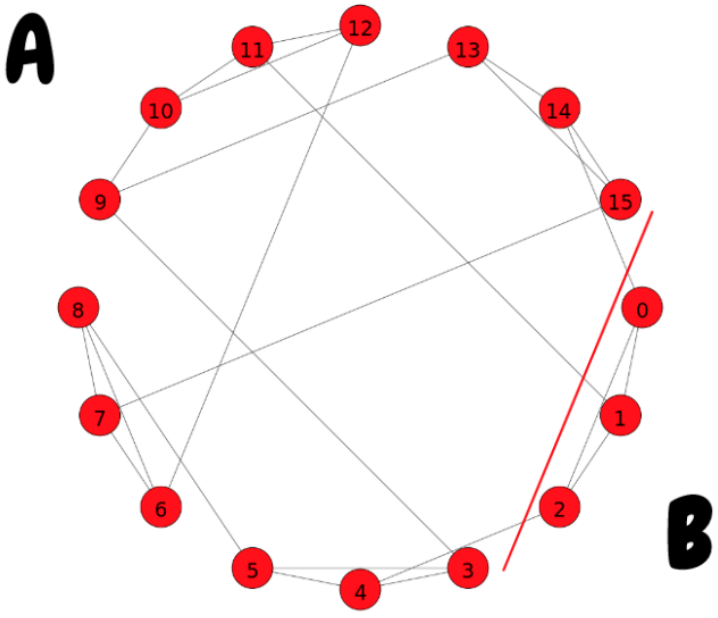
Grafo 17 - Internamente chamado de 16-2. No lado A temos o Grafo 12 e no lado B temos o K4.

18. Gerado a partir da colagem entre o K_4 e o Grafo 12.



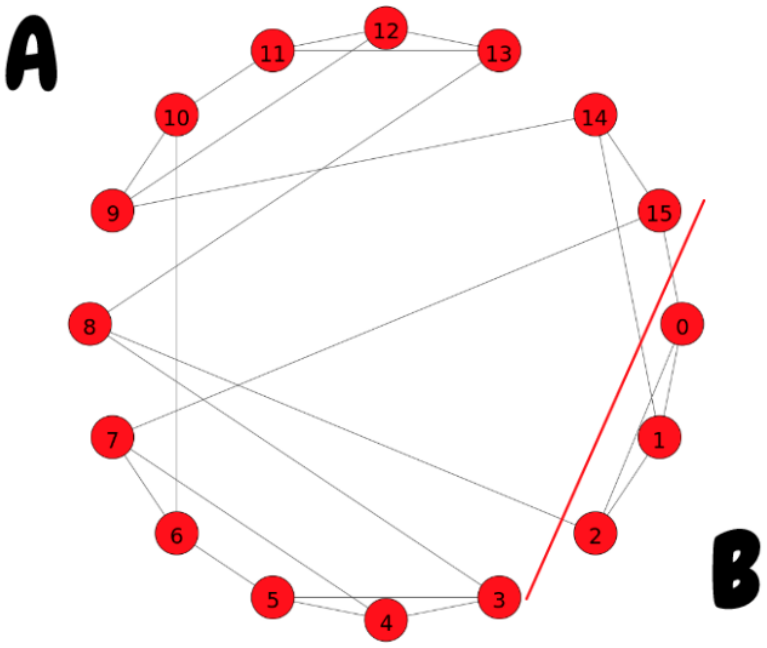
Grafo 18 - Internamente chamado de 16-3. No lado A temos o Grafo 12 e no lado B temos o K4.

19. Gerado a partir da colagem entre o K_4 e o Grafo 13.



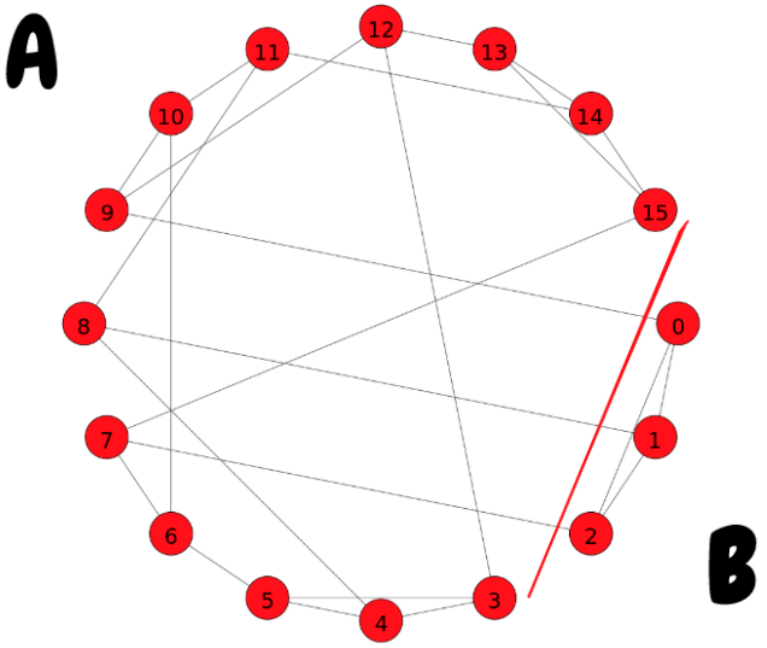
Grafo 19 – Internamente chamado de 16-21. No lado A temos o Grafo 13 e no lado B temos o K4.

20. Gerado a partir da colagem entre o K_4 e o Grafo 15.



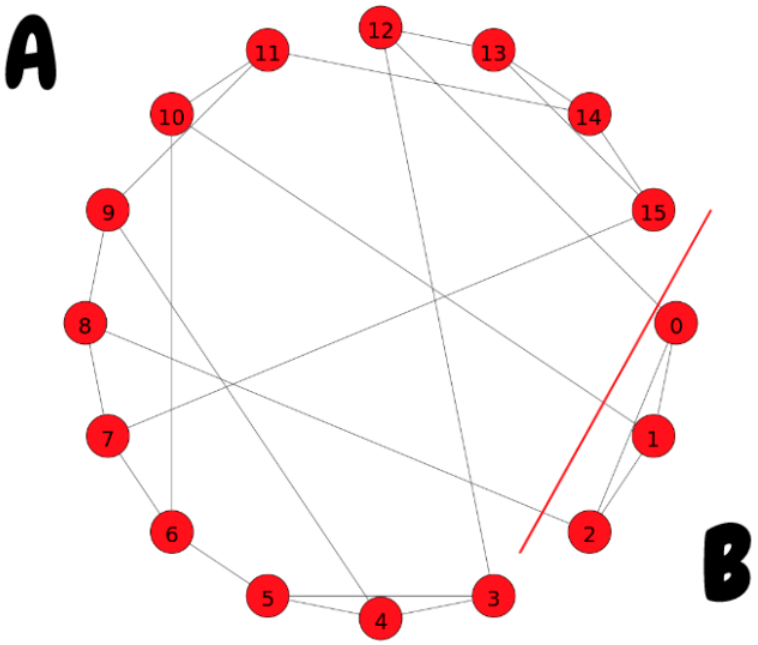
Grafo 20 – Internamente chamado de 16-45. No lado A temos o Grafo 15 e no lado B temos o K4.

21. Gerado a partir da colagem entre o K_4 e o Grafo 16.



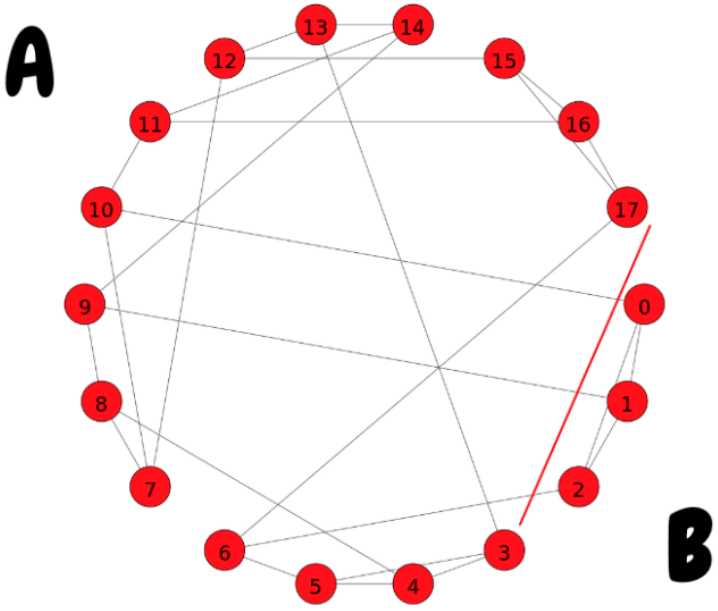
Grafo 21 – Internamente chamado de 16-46. No lado A temos o Grafo 16 e no lado B temos o K4.

22. Gerado a partir da colagem entre o K_4 e o Grafo 16.



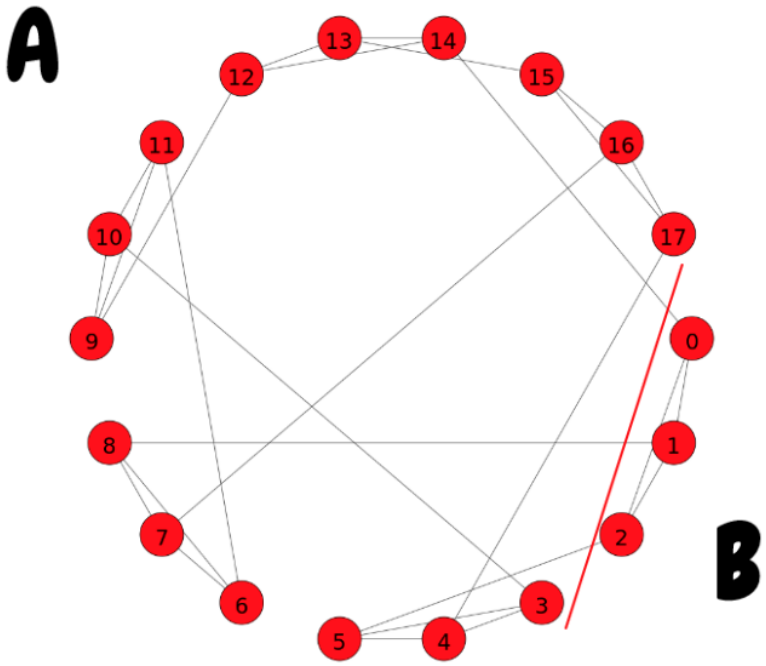
Grafo 22 – Internamente chamado de 16-47. No lado A temos o Grafo 16 e no lado B temos o K4.

23. Gerado a partir da colagem entre o K_4 e o Grafo 18.



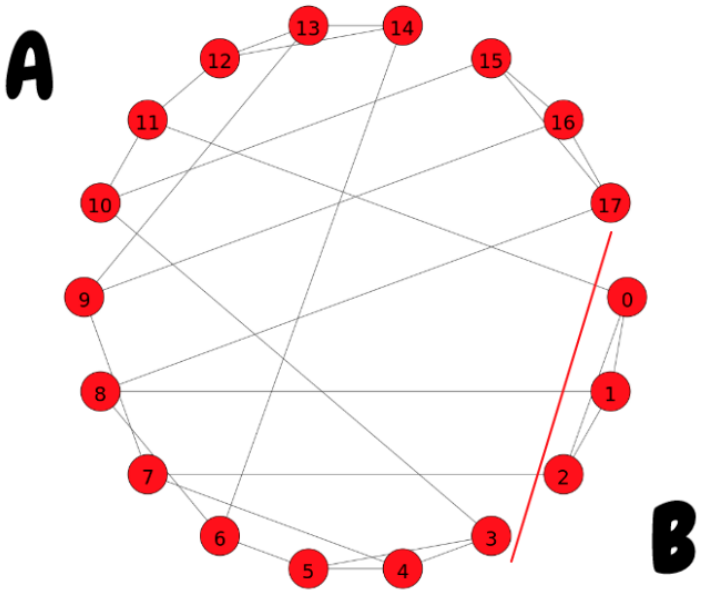
Grafo 23 – Internamente chamado de 18-9. No lado A temos o Grafo 18 e no lado B temos o K4.

24. Gerado a partir da colagem entre o K_4 e o Grafo 19.



Grafo 24 – Internamente chamado de 18-55. No lado A temos o Grafo 19 e no lado B temos o K4.

25. Gerado a partir da colagem entre o K_4 e o Grafo 21.



Grafo 25 – Internamente chamado de 18-85. No lado A temos o Grafo 21 e no lado B temos o K4.

5 CONCLUSÃO

Foi um grande desafio implementar esse algoritmo. A teoria é complexa e algumas partes precisaram ser implementadas mais de uma vez para funcionar corretamente. Muito mais dados foram gerados do que os apresentados nesse trabalho, mas não faria sentido colocar tudo aqui. Como já dito acima, o todos resultados e o código poderão ser encontrados no GitHub (https://github.com/denihs/tcc/blob/master/pynauty-0.6.0/tccSource).

Para o futuro, acredito que muitas melhorias poderiam ser aplicadas no código. Existem várias funções com um valor de complexidade alto que abrem oportunidades para melhorias. Além da melhoria no código, outro ponto que pode melhorar é o poder computacional para rodar esse algoritmo e o tempo em que poderíamos colocar esse algoritmo para rodar. Neste trabalho chegamos apenas a 22 vértices ao longo de 25 dias. Encontramos a família de 27 bricks desejada, e assim como afirmado na tese base desse trabalho, esses são todos os grafos que encontraríamos. Porém, acredito que seria interessante ir adiante de 22 vértices para constatarmos que de fato que não encontraríamos nada além da quantidade já prevista.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] D. O. Martinelli J., M.H. Carvalho. Brick´s cúbicos PM – compactos, 2019.

[2] Y. Lin M.H. Carvalho, X. Wang. Birkhoff-von neumann graphs that are pm-compact (submitted – last revised 26 jun 2019), 2019.

[3] U.S.R Murty J.A. Bondy. *Graph Theory.* Springer-Verlag, Berlin, 2008.

[4] P. Dobsan. Pynauty 0.6.0 documentation, 2015. https://web.cs.dal.ca/~peter/software/pynauty/html/guide.html

[5] M.H. Carvalho C. L. Lucchesi G. Sanjith C.H.C Little X. Wang, Y. Lin. A characterization of pm-compact bipartite and near-bipartite graphs. *Discrete Mathematics,* (313):772-783, 2013.

[6] W. R. Pullyblanks J. Edmounds, L. Lovász. Brick decomposition and the matching rank of graphs. *Combinatorica 2,* pages 247-274, 1982.

[7] V. Chvátal. On certain polytopes associated with graphs. *Jornal of Combinatorial Theory,* B(18): 138-154, 1975.

[8] *Itertools* - Functions creating iterators for efficient looping. Python Software Foundation, 2001-2020. https://docs.python.org/3/library/itertools.html

[9] *Python*. Python Software Foundation, 2001-2020. https://www.python.org/