Implementação do Jogo Polícia e Ladrão em Grafos

João Lucas Melo[a], Kaio Rodrigo[b], Thiago Andrade[c]

[a]*Bacharelado de Ciência da Computação, Universidade Federal da Bahia, Salvador, Bahia, joaollm@ufba.br*
[b]*Bacharelado de Ciência da Computação, Universidade Federal da Bahia, Salvador, Bahia, kaio.rodrigo@ufba.br*
[c]*Bacharelado de Ciência da Computação, Universidade Federal da Bahia, Salvador, Bahia, thiago.vieira@ufba.br*

Abstract

Cops and Robbers é um jogo modelado sobre um grafo onde dois jogadores, Cop e Robber, se movimentam sobre vértices do grafo até que ocorra a captura de Robber por Cop (ambos ocupem o mesmo vértice). Um grafo que garante a possibilidade de captura é dito cop-win (2).

Tendo como referência o trabalho de Gena Hahn, esse estudo possui como objetivo a implementação em python do teorema proposto pelo autor que garante a propriedade cop-win ou robbers-win de um grafo.

Keywords: Grafo, Cop-Win, Robbers-Win, Cops-and-Robber,

1. Introdução

O escopo de estudo de Teoria dos Grafos é vasto e complexo. O modelo matemático de grafos pode ser empregado em aplicações para além da área de conhecimento das exatas, utilizado como ferramenta para modelar sistemas férreos, conexões lógicas entre entidades e até jogos.

Cops and Robbers é um jogo modelado sobre um grafo conexo reflexivo e finito. Dois jogadores, um policial e um ladrão, se movimentam de um vértice v a vértices vizinhos de v em turnos alternados. O objetivo do jogo é alcançar um estado onde o policial ocupe o mesmo vértice do ladrão. Se isso acontecer, o ladrão é capturado e o grafo é dito $cop - win$, que condiciona a vitória ao jogador policial (1).

Em 2007, Gena Hahn, pesquisador e professor titular da Universidade de Montreal, publicou o artigo Cops, Robbers and Graphs, dedicado a estudar

matematicamente uma caracterização para um grafo $cop-win$. Hahn fundamenta a caracterização em um teorema, onde estabelece que todo grafo pode ser dito $cop-win$ se existe no grafo uma ordenação de vértices $v_1, v_2, ..., v_n$ tal que para todo $1 \leq i < n$ existe um j em que $i < j \leq n$ tal que o conjunto dos vértices vizinhos de v_i intersectados aos demais vértices seguintes da sequência sejam subconjunto do conjunto dos vértices vizinhos de v_j intersectados aos demais vértices seguintes a i na sequência.

O objetivo desse trabalho é estudar o teorema proposto por Hahn e desenvolver um algoritmo que implemente a ideia do artigo e a tese do teorema, traduzindo o mesmo em termos computacionais. Através da implementação desse algoritmo, dada qualquer entrada de um grafo G de acordo com as especificações requisitadas, será possícel definir se G é ou não $cop-win$.

A implementação consiste em um código em python, cuja entrada é dada em um conjunto de inteiros representando os vértices e uma lista de duplas de inteiros representando as arestas. O algoritmo retorna o inteiro 1 caso o grafo G dado como entrada seja $cop-win$ e retorna 0 caso contrário.

A fim de comparação, será ainda desenvolvido um algoritmo de simulação dos movimentos em um grafo G. Por força bruta, o algoritmo buscará movimentar o policial e o ladrão pelo grafo a fim de buscar uma posição de captura. Se isso ocorrer, o grafo será dito $cop-win$ e o algoritmo retornará o inteiro 1. Caso contrário, como o programa só termina com a captura do ladrão, enquanto este fugir do policial o programa rodará indefinidamente.

Serão apresentadas ainda análises do comportamento de ambos os algoritmos, buscando entender as propriedades e condições de funcionamento bem como a análise do tempo de execução. Contrapostas as análises do algoritmo baseado no teorema e o de força bruta, será possível comparar com mais sobriedade o desempenho de ambos os programas e entender suas particularidades.

2. Contexto

É importante estabelecer algumas definições e condições, tanto a respeito do grafo quanto das características do jogo Cops and Robbers, para entender como um dado grafo G pode ser definido como $cop-win$.

Definição 1. Um grafo é um modelo matemático representado por um conjunto de entidades, os vértices, cujas relações são estabelecidas por arestas entre um par de vértices. A relação pode ser expressa matematicamente por

um grafo G, $V(G)$ conjunto de vértices do grafo e $E(G)$ conjunto de arestas do grafo onde $G = (V(G), E(G))$ (4).

Definição 1. Se para todo vértice $v \in V(G)$ não há arestas de v para ele mesmo nem múltiplas arestas (arestas com mesmo par de vértices), chamamos o grafo G de grafo simples. Caso haja arestas do vértice v para ele mesmo para todo vértice v, o grafo é dito reflexivo (3).

Definição 1. Se para todas as arestas $e \in E(G)$ não há noção de direção, ou seja, a aresta conecta dois vértices u, v sem noção de um vértice de saída e vértice de chegada, o grafo G é dito não direcionado (3).

Definição 1. Sendo $e \in E(G)$ uma aresta de G entre os vértices u, v, podemos representá-la por $e = uv$. Além disso, dizemos que u e v são vértices vizinhos, ou adjacentes. Denotamos o conjunto de todos os vizinhos de um vértice v no grafo G por $N_G(v)$ (3).

Definição 1. Um caminho é um grafo simples cujos vértices possam ser ordenados tal que dois vértices são adjacentes um ao outro (há uma aresta entre eles) se, e somente se, eles são consecutivos na ordenação (3).

Definição 1. Um grafo G é dito conexo se todo par de vértices $v, u \in V(G)$ pertencem a um caminho. Caso contrário, G é desconexo (3).

O teorema apresentado por Hahn para a caracterização de um grafo $cop-win$ trabalha sobre a condição de que o grafo deva ser reflexivo, não direcionado e conexo. Uma vez estabelecidas as definições dessas características, nos resta esclarecer as condições do jogo de Cops and Robbers.

Sobre um grafo reflexivo, conexo e não direcionado, dois jogadores, Cop e Robber, se posicionam sobre dois diferentes vértices c e r, respectivamente. Cada jogador possui direito a uma ação, que consiste em se movimentar pelas arestas do vértice atual até um vértice adjacente. Ao se movimentar, o jogador cede a vez ao seu adversário.

O jogo começa sempre com o movimento de Cop, que sai de c a um vértice adjacente qualquer u, cedendo sua vez a Robber, que se movimenta de r a outro vértice adjacente qualquer v. Cop sempre se moverá tentando se aproximar de Robber, ou seja, buscando se posicionar no vértice cuja distância a Robber seja a menor entre todas as possibilidades de movimentação. Robber, por sua vez, busca se movimentar para o vértice mais distante possível

de Cop. Ambos os jogadores apenas usam as arestas para se locomover, suas posições finais e iniciais sempre serão vértices.

O jogo termina se Cop captura Robber, ou seja, se ambos ocupam o mesmo vértice. Caso isso ocorra, o grafo onde o jogo ocorreu é dito $cop-win$. Se não houver possibilidade de captura do Robber em um dado grafo G, ele é então dito $robber - win$.

Para o entendimento do teorema proposto pelo autor, cabe ainda comentar sobre homomorfismo e retração de um grafo.

Definição 1. Um homomorfismo de um grafo $G = (V, E)$ para um grafo $G' = (V', E')$ é um mapeamento $f : V \to V'$ tal que $\{f(u), f(V)\} \in E'$ se, e somente se, $\{u, v\} \in E$ (6).

Definição 1. O resultado da retração de um grafo G é um subgrafo H tal que existe um homomorfismo $r : G \to H$, chamado retração, onde $r(x) = x$ para todo $x \in V(H)$ (6).

Estabelecidas as definições, comportamentos e pressupostos dos conceitos abordados pelo teorema propsto por Hahn, podemos agora entender como o autor constitui a caracterização de um grafo $cop - win$.

3. Teorema

Partiremos do pressuposto que o jogo Cops and Robbers ocorrerá sobre um grafo reflexivo, não direcionado e conexo. Ambos os jogadores alternam movimentos, que consistem em ocupar um vértice vizinho do atual. O comportamento de Cop é dado sempre buscando aproximação de Robber e de Robber sempre buscando distanciamento de Cop. O jogo termina quando Cop captura Robber, ou seja, ambos ocupem o mesmo vértice. Se isso ocorre, o grafo é dito $cop - win$.

Em seu artigo, Hahn estabelece uma definição antes de apresentar seu teorema:

Definição 1. Suponha os vértices de um grafo finito G são enumerados da forma $v_1, ..., v_n$. Nessa ordem fixa, definimos para um vértice $v \in V(G)$, $N_i(v) = N(v) \cap \{v_j; i \leq j \leq n\}$. Similarmente, denotamos $N_i[v]$ (1).

O autor então segue com a definição e demonstração do seu teorema caracterizando um grafo $cop - win$:

Teorema 1. *Caracterização de um grafo cop-win: Um grafo finito, não direcionado e reflexivo G é $cop-win$ se, e somente se, seus vértices podem ser ordenados linearmente da forma $v_1, v_2, ..., v_n$ tal que para todo $i, 1 \leq i < n$, existe um $j, i < j \leq n$ tal que $N_i[v_i] \subseteq N_i[v_j]$ (1).*

Proof. A prova depende fortemente da reflexividade do grafo. Como é uma boa aplicação da ideia de uma retração, nós provaremos a proposição essencial à ela. Se lembre que um homomorfismo de um grafo G a um grafo H é um mapeamento $h : V(G) \to V(H)$ tal que se $uv \in E(G)$, então $h(u)h(v) \in E(H)$. Nós escrevemos $h : G \to H$, ou simplesmente $G \to H$, para indicar que h é um homomorfismo de G a H (ou simplesmente atestar a existência de tal homomorfismo). Uma retração é um homomorfismo ρ de G em um subgrafo H de G tal que ρ restrito a H é a identidade de H (1). □

De forma simplificada, o teorema afirma que um grafo G é dito $cop-win$ caso exista uma ordenação de seus vértices onde os vizinhos de um dado vértice v, intersectados com os vértices sucessores da sequência, estejam contidos no conjunto de vértices vizinhos de algum sucessor na ordenação. Esse processo de comparação deve ser verdadeiro para todos os vértices da sequência, independente da posição de Cop ou Robber no grafo.

Esse trabalho possui como objetivo interpretar o teorema proposto por Hahn em termos computacionais, traduzindo-o em um algoritmo que reconheça em uma representação de um grafo e decida se ele é ou não $cop-win$.

4. Explicação do Algoritmo

Com base no teorema proposto por Hahn em seu artigo Cops, Robbers and Graphs, foi desenvolvido um algoritmo que reconhece se um dado grafo G é ou não $cop-win$, em função da caracterização dada pelo autor.

O programa foi desenvolvido utilizando a linguagem de programação python. Ele recebe dois objetos como entrada, sendo eles uma lista de inteiros (representando os vértices do grafo) e uma lista de duplas de inteiros (representando o conjunto de arestas entre todos os pares de vértices do grafo). Após as devidas operações, o algoritmo retorna 1 caso o grafo passado como entrada seja $cop-win$ e 0 caso contrário.

Ao receber o conjunto de vértices e arestas, o algoritmo itera sobre ambos, registrando para cada $v \in V(G)$ seu respectivo conjunto de vértices vizinhos. Dessa forma, é possível acessar com facilidade $N(v_i)$ para qualquer $i \leq n$ de alguma sequência de vértices $v_1, v_2, ..., v_n$.

Uma vez tendo acesso ao conjunto de vértices vizinhos de cada vértice do grafo, o algoritmo testará a condição do teorema para cada possível ordenação de vértices. Se em alguma das ordenações a condição do teorema for satisfeita, então o algoritmo para a execução e retorna 1. Caso contrário, continua a execução até a última possível ordenação de vértices e retorna 0 se em nenhuma ordenação foi possível satisfazer a condição do teorema.

Para identificar se uma ordenação de vértices $v_1, v_2, ..., v_n$ satisfaz a condição do teorema, o algoritmo busca para todo $1 \leq i < n$ um $j, i < j \leq n$ tal que os vizinhos de v_i (com interseção dos posteriores vértices da sequência) sejam subconjunto dos vizinhos de v_j. Encontrado um j para todos os i da ordenação, o algoritmo identifica o grafo como $cop - win$ e retorna 1. Caso contrário, o algoritmo identifica o grafo como $robber - win$ e retorna 0.

5. Implementação do algoritmo *copWin*

O algoritmo foi implementado em python. Foram usadas estruturas nativas da linguagem para a representação, definição, armazenamento e manipulação das estruturas dos grafos.

A lista de vértices foi representada por uma *lista* de inteiros e a lista de arestas por uma *lista* de *tuplas*, onde cada *tupla* contém dois inteiros representando o par de vértices em que a aresta incide. Foi usado ainda um *dicionário* para o armazenamento de conjuntos de vértices vizinhos a todo vértice v do grafo. A chave do *dicionário* corresponde a um inteiro, representando um vértice, e seu valor atribuído corresponde a uma *lista* de inteiros, representando os vértices vizinhos ao vértice da chave.

A implementação foi modularizada em diferentes funções, cuja função principal é $copWin()$, cujo pseudocódigo segue abaixo:

Algorithm 1 CopWin(vertices,edges)

```
for all iterations i in vertices do
    if TheoremCondition(i,edges) == True then
        return True
    end if
end for
return False
```

A função recebe como argumento as entradas descritas anteriormente, uma lista de vértice e uma lista de arestas.

Ao receber as entradas, a função cria um dicionário cujas chaves são os vértices do grafo e os valores são uma lista contendo todos seus vértices adjacentes. O dicionário, portanto, armazena todos os vizinhos de cada vértice v do grafo, como descrito anteriormente. O dicionário é atribuído a variável $neighbors$.

A função então invoca $permutationWrapper()$ para testar a condição do teorema para todas as possíveis ordenações de vértices do grafo. Caso alguma ordenação satisfaça a condição, $copWin()$ retorna 1, sinalizando que o grafo é cop-win. Caso nenhuma ordenação de vértices satisfaça a condição, $copWin()$ retorna 0, sinalizando que o grafo não é cop-win.

A função $permutationWrapper()$, por sua vez, realiza a checagem da condição do teorema para todas as ordenações possíveis entre os vértices. Para isso, ela instancializa todas as possíveis permutações de sequência de vértices através de uma abordagem recursiva. Invoca a função $theoremCondition()$ passando o dicionário de vizinhos e uma das possíveis ordenações de vértices como argumentos. Dessa forma, o algoritmo garante a checagem da condição do teorema para todas as possibilidades de ordenação.

Algorithm 2 TheoremCondition(orderedVertices, edges)

```
for i in orderedVertices do
    for j in orderedVertices where j ← i + 1 do
        subsetV1 ← NeighbourIntersection(i.index, i, orderedVertices, edges)
        subsetV2 ← NeighbourIntersection(i.index, j, orderedVertices, edges)
        if not SubSetComparison(subsetV1,subsetV2) then
            return False
        end if
    end for
end for
return False
```

$theoremCondition()$, por sua vez, traduz o teorema em termos computacionais. Busca para todo i na sequência $1 \leq i < n$ (onde n representa o tamanho do conjunto de vértices do grafo), um $j, i < j \leq n$ tal que $N_i[v_i] \subseteq N_i[v_j]$.

Para todo i na condição estabelecida, a função busca um j tal que a condição dos vizinhos seja satisfeita. A checagem dessa condição é feita ao invocar a funções $neighbourIntersection()$ e $subsetComparison()$. A primeira

define os subconjuntos a serem comparados e a segunda efetivamente checa a condição de um estar ou não contido no outro.

Caso para todo i exista o j desejado tal que valha a condição do teorema, a função retorna True. Caso contrário, retorna False.

Algorithm 3 NeighbourIntersection(posicaoNaLista,v,vertices,edges)

```
Vizinhos = edges[v]
Conj = [ ]
for v in vertices do
    if posicaoNaLista > 0 then
        posicaoNaLista -= 1
        continue
    end if
    if v in Vizinhos then
        Conj.append(v)
    end if
end for
return Conj
```

A função *neighbourIntersection*() cria os conjuntos de vizinhos a serem comparados em *theoremCondition*(). Recebe quatro parâmetros de entrad, um inteiro i (referente ao índice do vértice v_i na ordenação iterada), um inteiro vi (referente ao vértice v_i da ordenação iterada), uma lista de inteiros $listV$ (referente à ordenação de vértices) e um dicionário $listE$ (referente aos vértices vizinhos de todos os v vértices do grafo).

A função armazena em uma lista todos os vértices vizinhos a vi que estão à sua frente na ordenação $listV$ e a retorna ao final da operação.

Algorithm 4 SubSetComparison(set1,set2)

```
for i in set1 do
    if not i in set2 then
        return False
    end if
end for
return True
```

Por fim, *subsetComparison*() exerce papel de função auxiliar para comparação dos elementos de duas listas. Recebe como parâmetro duas listas,

$list1$ e $list2$, e retorna True caso todos os elementos de $list1$ estejam em $list2$ e False caso contrário.

6. Análise da complexidade do algoritmo *copWin*

Para começar a análise do algoritmo, precisamos considerar o ponto de entrada, a função $CopWin$. A função $CopWin$ começa com uma chamada para todas as iterações i dos vértices, com uma condição de parada no meio do código. Por haver uma chamada para $TheoremCondition$, precisamos examina-la antes de continuar nossa análise da função $CopWin$.

$TheoremCondition$ começa com dois blocos de "for" em cima dos vértices ordenados: um que começa de i primeiro elemento, e outro começa de $j = i + 1$. Dentro dos loops, há duas chamada para $NeighbourIntersection$, que não muda de acordo com a entrada e sempre leva $O(n)$ passos para ser completa. Ainda, chamamos $SubSetComparison$ que, no melhor caso, é $\Theta(1)$ (Quando falha no primeiro elemento) e em seu pior caso, é $O(n^2)$ (Quando ele verifica todos os elementos e subset1 é subconjunto de subset2). Então, em seu pior caso, $TheoremCondition$ demora $n^2 + 2n + n^2 = O(n^2)$. Em seu melhor caso, $TheoremCondition = n^2 + 2n + 1 = \Theta(n^2)$.

Usando a análise de $TheoremCondition$, podemos continuar a análise sabendo que $TheoremCondition = O(n^2) = \Theta(n^2)$. Pela condição de parada da função $CopWin$ ser dependente de $TheoremCondition$, podemos dizer que o melhor caso de $CopWin$ é $\Theta(n^2)$, quando a primeira iteração que nós testamos é suficiente para provar o teorema. No pior caso, $CopWin$ testa todas as permutações possíveis sem achar uma que valide o Teorema, $n! * n^2 = O(n!n^2)$.

6.1. Analisando a complexidade do problema de decidir se um grafo é cop-win

Na seção 3.3 Complexity, Hahn nos afirma que não é difícil (isto é, é polinomial) identificar se um grafo é cop-win. Porém, para que isso fique mais claro, podemos analisar da seguinte forma: O autor faz uma citação [Goldstein e Reingold] onde é proposto o teorema:

O problema de determinar se k policiais podem capturar um ladrão, dada as posições iniciais, é EXPTIME-completo.

Dentro desse teorema, reduzindo a prova ao nosso cenário, podemos fixar k como 1, por exemplo, e o teorema mantém sua validade. Em conclusão, dada a análise do algoritmo e a do problema supracitado, podemos dizer que o **problema** pode ser resolvido em tempo polinomial, isto é, pertence a NP.

7. Implementação do algoritmo *bruteForce*

A fim de comparação de desempenho, foi desenvolvido um segundo algoritmo, que testa por forca bruta se um dado grafo G é $cop - win$.

O algortimo, implementado em python, recebe uma lista de inteiros (representando os vértices do grafo), uma lista de duplas de inteiros (representando as arestas para todo par de vértices), e dois inteiros (posição inicial do Cop e do Robber no grafo, respectivamente).

O algoritmo simula uma partida de Cops and Robbers, onde o Cop semrpe tenta se aproximar de Robber e este tenta sempre se afastar de Cop. Ambos os jogadores alternam turnos, se movendo estre os vértices do grafo de acordo com as regras já estabelecidas. Se, em algum momento, Cop e Robber ocuparem o mesmo vértice, o algoritmo interrompe sua execução e retorna 1, sinalizando que o grafo passado como entrada do programa é $cop - win$. No entanto, enquanto não houver a captura, o algoritmo continuará a execução. Nesse caso, se o grafo é $robber - win$, ele rodará indefinidamente.

São usadas duas funções na implementação do algoritmo de força bruta, o $distances$ e $bruteForce$.

$bruteForce$ é a função principal do algoritmo. Recebe como entrada os valores descritos anteriormente, uma lista de inteiros, lista de duplas de inteiros e dois inteiros. Assim como no algoritmo principal, os vértices e arestas do grafo representados pelas listas são iterados a fim de armazenar o conjunto de vizinhos de um vértice v para todo $v \in V(G)$. Uma vez tendo acesso a essas informações, o algoritmo rodará enquanto a posição de Cop for diferente de Robber. O algoritmo então simula um movimento de Cop, checa se houve captura (e para, caso ocorra), cede a vez a Robber, checa mais uma vez se houve captura (e para, caso ocorra), e segue alternando entre movimentos dos jogadores e checagens de captura. O pseudocódigo de $bruteForce$ segue abaixo:

Algorithm 5 bruteForce(vertices,edges,cop,robber)

```
while cop ≠ robber do
    neighbours_cop ← N(cop)
    distance_to_robber ← distances(neighbours, robber)
    MoveCop ← Max
    for i in neighbours_cop do
        if distance_to_robber[i] ≤ MoveCop then
            MoveCop ← distance_to_robber[i]
            cop ← i
        end if
    end for
    if cop == robber then return True
    end if
    neighbours_robber ← N(robber)
    distance_to_cop ← distances(neighbours, cop)
    MoveRobber ← 0
    for i in neighbours_robber do
        if distance_to_cop[i] ≥ MoveRobber then
            MoveRobber ← distance_to_cop[i]
            robber ← i
        end if
    end for
end while
return True
```

Como descrito anteriormente, Cop sempre se movimentará a fim de se deslocar para o vértice mais próximo de Robber, assim como Robber se movimenta a fim de se deslocar para o vértice mais distante de Cop. A tomada de decisão é simulada pelo algoritmo através do cálculo da distância dos vértices vizinhos de um jogador a outro. Para isso, é invocada a função *distances*, que recebe *neighbors*, a estrutura contendo os vizinhos de um dado vértice $v \in V(G)$ e um inteiro *source* representando um vértice cuja distância deve ser avaliada.

Algorithm 6 distances(edges,vertice)

```
Q ← Queue
Q.put(vertice)
Visited_vertice ← []
Visited_vertices.append(vertice)
distance ← {k : max for k edges}
while not Q.empty do
    vertex ← q.top
    if  thenvertex == vertice
        distance[vertice] = 0
    end if
    for u in neighbors[vertex] do
        if u not in visited_vertices then
            if distance[u] > distance[vertice] + 1 then
                distance[u] = distance[vertex] + 1
            end if
            Q.put(u)
            Visited_vertices.append(u)
        end if
    end for
end while
return distance
```

distances usa a lista *neighbors* de vértices vizinhos a um v para calcular a distância de cada vértice $n \in neighbors$ a *source*. A função então retorna uma lista de pares chave/valor, cujas chaves são os vértices $n \in neighbors$ e a eles são atribuídos valores correspondentes à sua distância a *source*.

A cada movimento de um jogador, *bruteForce* usa *distances* para saber a qual distância do oponente (passando o vértice de sua posição como parâmetro *source*) o jogador estará se mover para qualquer um de seus vértices vizinhos. Portanto, Cop sabe qual o vértice vizinho que o deixará mais perto de Robber, assim como Robber sabe qual vértice vizinho o deixará mais longe de Cop. *bruteForce* pode, então, movimentar Cop e Robber pelo grafo (ou seja, atualizar os valores dos vértices que ocupam).

Como descrito, a condição de captura será checada a cada movimento de um jogador. Se houve captura, ou seja, se Cop e Robber ocupam o mesmo vértice, o algoritmo finaliza sua execução e retorna 1, sinalizando que o grafo

é $cop - win$. Caso não ocorra, a simulação continua indefinidamente.

8. Análise da complexidade do algoritmo *bruteForce*

No início do código, temos a função $bruteForce$ que faz uma verificação em **cada vértice do grafo**, analisando **todos os seus vizinhos** e também a distância entre Cop e Robber. Sua condição de parada é o caso onde Cop e Robber ocupem o mesmo vértice. Então, esse laço de verificação, no pior caso, visita todos os vértices e calcula o conjunto vizinhança de cada um. Portanto, somente essa função levaria um tempo **exponencial** para ser executada.

9. Conclusão

O estudo se estende demosntrando algumas classes de grafos e especificações em que é possível dizer se um grafo é $cop - win$ ou não. Porém, como o foco do nosso trabalho é somente o teorema do Hahn e sua implementação, avaliamos somente a sessão do teorema e sua complexidade. Em suma, é notável pela execução do código e a comparação com a execução do brute-force, que a solução proposta pelo artigo é sofisticada, através do teorema. A ordenação de vértices, mesmo em todas as permutações possíveis, nos deu um algoritmo $n! * n^2 = O(n!n^2)$, enquanto no $bruteForce$, esse tempo é exponencial. É válido citar ainda que além da solução do problema Cop and Robbers, o artigo traz um teorema importante pro estudo de remoção de vértices em um grafo, e pode ser utilizado e/ou aplicado também com esse objetivo.

References

[1] HAHN, Gena. **COPS, ROBBERS AND GRAPHS**, 2007

[2] M. AIGNER; M. FROMME. **A GAME OF COPS AND ROBBERS** Discrete Applied Mathematics 8, 1984.

[3] WEST, Douglas B. **Introduction to Graph Theory**, 2002.

[4] MOTA, Guilherme Oliveira. **Teoria dos Grafos**, 2019.

[5] GOLDSTEIN, A. S. - REINGOLD, E. M. **The complexity of pursuit on a graph**, 1995.

[6] HOMOMORFISMO de Grafos. Wikipedia, 2022. Disponível em: https://pt.wikipedia.org/wiki/Homomorfismo_de_grafos.

[7] PROJETO do Artigo da Implementação do Jogo Polícia e Ladrão em Grafos. Overleaf, 2022. Disponível em: https://pt.overleaf.com/project/63793b9f8d699c18a25830a5.

[8] REPOSITÓRIO de Trabalho Grafos. Github, 2022. Disponível em: https://github.com/jlucasldm/trabalho-grafos.

[9] RODRIGO, Kaio. **Grafos - MATA53 - Implementação do problema Cops and Robbers**. Youtube, 2022. Disponível em: https://www.youtube.com/watch?v=XS6Q4z-4NDU.