57º SBPO
SIMPÓSIO BRASILEIRO DE PESQUISA OPERACIONAL
Gramado • RS
DE 5 A 9 DE OUTUBRO DE 2025

Abordagens Computacionais para o *Gold Grabbing Game*

Heitor Melo de Lucas Brandão
INF/UFG
Instituto de Informática, INF-UFG, Câmpus Samambaia, Goiânia/GO
heitormelo26@discente.ufg.br

Hebert Coelho da Silva
INF/UFG
Instituto de Informática, INF-UFG, Câmpus Samambaia, Goiânia/GO
hebert@ufg.br

RESUMO

Grabbing Game é um jogo em um grafo ponderado nos vértices (pesos em ouro), onde dois jogadores (Alice e Bob) revezam em turnos retirando vértices sem desconectar o grafo, com Alice iniciando o jogo, ganha o jogo quem acumular a maior soma dos pesos nos vértices escolhidos. Na literatura é demonstrado que o *Grabbing Game* é um problema que pertence à classe PSPACE-completo. Porém, a maioria dos estudos são abordagens matemáticas focadas na garantia da vitória, este trabalho objetiva cobrir essa lacuna investigando abordagens computacionais considerando tanto a vitória simples quanto a maximização do ouro. Primeiro, é proposta uma estratégia ótima por força bruta com sua análise algorítmica, destacando seu alto custo computacional, para então propor otimizações com programação dinâmica (PD) para grafos caminho e bipartidos completos. Também são exploradas abordagens gulosas em grafos completos, bipartidos completos e caminhos e de bicoloração em grafos caminho e ciclo.

PALAVRAS CHAVE. ***Gold Grabbing*, Estratégia vencedora, Estratégia ótima, Programação dinâmica.**

Tópicos (Estratégia Ótima,Abordagem Ótima com PD)

ABSTRACT

Grabbing Game is a game played on a vertex-weighted graph (weights in gold), where two players (Alice and Bob) take turns removing vertices without disconnecting the graph, with Alice starting the game. The winner is the player who accumulates the highest total weight from the chosen vertices. In the literature, it is shown that the Grabbing Game is a problem belonging to the PSPACE-complete class. However, most studies present mathematical approaches focused on gold maximization. This work aims to fill that gap by investigating computational approaches that consider both simple victory and gold maximization. First, an optimal brute-force strategy is proposed along with its algorithmic analysis, highlighting its high computational cost. Then, optimizations using dynamic programming (DP) are proposed for path graphs and complete bipartite graphs. Greedy approaches are also explored in complete graphs, complete bipartite graphs, and paths, as well as bicoloring strategies in path and cycle graphs.

KEYWORDS. Gold Grabbing, Winning strategy, Optimal strategy, Dynamic programming.

Paper topics (Optimal Strategy, Optimal Approach with DP)

DOI: 10.59254/sbpo-2025-212396
https://proceedings.science/p/212396?lang=pt-br

1. Introdução

Gold Grabbing Game ou *Grabbing Game* é um jogo para dois jogadores em um grafo ponderado nos vértices, cujos pesos representam quantidade em ouro. Os dois jogadores são apelidados de Alice e Bob, na versão original o jogo sempre começa pela Alice que deve retirar um vértice do grafo e acumular seu peso em ouro, depois alternadamente os jogadores retiram vértices até que o grafo fique vazio, objetivando acumular a maior quantidade de ouro. Este jogo possui uma única regra: depois de cada jogada, o grafo precisa continuar conexo, ou seja, não se pode retirar um vértice de corte.

Originalmente, em sua primeira versão, o jogo foi proposto por [Winkler, 2004], em que se questiona como Alice poderia garantir pelo menos metade das moedas (com valores diferentes) sob uma mesa pegando-as alternadamente com o Bob. Neste artigo, foi provado que Alice garante pelo menos metade do ouro em qualquer grafo caminho com número par de vértices, partindo da ideia de que basta Alice escolher a sequência de vértices, ímpares ou pares, que acumula mais ouro.

Posteriormente, o jogo foi generalizado para grafos, conforme definido anteriormente, com a restrição de que, após cada jogada, o grafo permaneça conexo. Essa versão generalizada, denominada *Graph-Grabbing Game* por [Micek e Walczak, 2011], estabelece que Alice garante ao menos $1/4$ do total do ouro em qualquer árvore com número par de vértices. Os autores ainda conjecturam que, sob as mesmas condições, Alice pode assegurar pelo menos metade do ouro disponível. Sendo esta conjectura provada em [Seacrest e Seacrest, 2012] cujos autores conjecturam que Alice ganha o jogo em todo grafo bipartido par. No artigo [Egawa et al., 2018] os autores provam que Alice ganha o jogo na classe $K_{m,n}$-árvore com número par de vértices, esta classe generaliza a classe dos grafos bipartidos completos, e em [Boriboon e Kittipassorn, 2020] provam que o problema pode ser resolvido nas classes bipartidas $B(T)$-árvore e a árvore $B(C_{2n})$ ambas com número par de vértices.

Ademais, há trabalhos sobre o *Gold Grabbing* cujas propostas estão em desenvolver um novo modo de jogo com novas regras, como em [Matsumoto et al., 2022] cuja modificação consiste em ter vértices e arestas ponderados, de modo a obter o ouro pelo vértice escolhido e também pelas arestas incidentes a este. Em [Chaplick et al., 2016] propuseram uma modificação na definição do turno da jogada, deixando de ser jogadas alternadas com Alice começando, para que o turno seja jogado pelo jogador que estiver com menos ouro, e nestas condições provaram que Alice pode assegurar $1/3$ do ouro. Sob a perspectiva da complexidade de algoritmos, de acordo com [Cibulka et al., 2013], o *Gold Grabbing* pertence à classe PSPACE-completo, isto é, pertence ao conjunto de todos os problemas de decisão que podem ser resolvidos por uma máquina de Turing determinística usando uma quantidade polinomial de espaço (memória).

Assim, o problema do *Gold Grabbing*, mesmo tendo sua primeira aparição em [Winkler, 2004], continua como um problema ativo e em aberto para determinadas conjecturas, como a de [Seacrest e Seacrest, 2012]. Além disso, há uma lacuna na literatura no que se refere às abordagens computacionais e aos algoritmos aplicáveis a esse problema. Diante disso, propõe-se o estudo de diferentes abordagens algorítmicas de estratégias vencedoras e estratégias ótimas, isto é, Alice garantir pelo menos metade do ouro e acumular o máximo de ouro possível, respectivamente.

Neste trabalho, apresenta-se a discussão da seguinte maneira: na Seção 2, apresentamos uma estratégia ótima para o *Grabbing Game* baseada em força bruta e analisamos sua complexidade algorítmica; na Seção 3, apresentamos essa estratégia usando programação dinâmica e, também, com otimizações de memória específicas para grafos $K_{m,n}$ e P_n; na Seção 4, investigamos o uso de abordagens gulosas, analisando se elas funcionam como estratégias vencedoras ou ótimas nas classes K_n, $K_{m,n}$ e P_n; e na Seção 5, estendemos essa investigação para abordagens por bicoloração nas classes P_n e C_n. Por fim, a Seção 6 apresenta as conclusões e recomendações para trabalhos

DOI: 10.59254/sbpo-2025-212396
https://proceedings.science/p/212396?lang=pt-br

futuros.

2. Estratégia Ótima

No estudo de [Egawa et al., 2018], os autores demonstram que Alice ganha o jogo em todo grafo $K_{m,n}$-árvore com número par de vértices, isto corrobora com a conjectura de [Seacrest e Seacrest, 2012]. Uma $K_{m,n}$-árvore G, com $m, n \geq 1$, é um grafo bipartido obtido a partir de um grafo bipartido completo $K_{m,n}$ e de árvores $T_1, T_2, \ldots$, por meio da identificação de exatamente um vértice de $K_{m,n}$ com exatamente um vértice de T_i, para $1 \leq i \leq m + n$.

Dado um grafo G e $v \in V(G)$, se um jogador retira v é dito que foi feito o "movimento v" e este jogador obtém o peso $w(v)$. Um movimento v é viável se o grafo $G - v$ permanece conexo e este é ótimo se ele maximiza o ganho do jogador e minimiza os ganhos do outro jogador, resultando na definição de estratégia ótima, ou seja, uma abordagem do *Gold Grabbing* que fornece maximização no ouro obtido, por [Egawa et al., 2018]:

$$\text{val}(G) = \begin{cases} 0, \text{ se } V(G) = \emptyset, \\ \max\{\omega(v) - \text{val}(G - v) \mid v \text{ é viável}\}, \quad \text{se } V(G) \neq \emptyset, \end{cases} \tag{1}$$

$$\text{Um movimento } v \text{ é } \textit{ótimo} \iff \text{val}(G) = \omega(v) - \text{val}(G - v) \tag{2}$$

2.1. Complexidade Algorítmica

Utilizando a Equação (1) como base, implementa-se o Algoritmo 1 de força bruta que calcula os movimentos ótimos em um grafo G, atribuindo valores, $val(G)$ em cada decisão de escolha de vértice a ser removido. Dessa forma, em cada instante do grafo G tem-se qual é o movimento ótimo, enunciado na Fórmula 2, para aquela situação, o que representa qual vértice dentre os viáveis naquele momento trará maior vantagem ao jogador. Não obstante, observa-se que esse algoritmo é computacionalmente oneroso por trabalhar com todas as possibilidades de jogos disponíveis, apresentando complexidade exponencial para um grafo qualquer, e pode ser representado pela seguinte recorrência:

$$\text{T}(n) = \begin{cases} 1, \text{ se } n = 1, \\ v_i * T(n-1) + (|E| + |V|), \quad \text{se } n > 1 \end{cases} \tag{3}$$

Sendo v_i os vértices viáveis para aquela iteração, $T(n-1)$ a chamada recursiva para a próxima iteração e $(|E| + |V|)$ o custo da operação de verificar se o vértice é viável, mediante verificação de conectividade do grafo após a remoção simulada do vértice a partir de um BFS. Desse modo, observa-se que a complexidade do algoritmo, apesar de estar limitada a no máximo $O({v_i}^n)$, varia de acordo com o tipo do grafo, visto que, dependendo da estrutura do grafo, a quantidade de vértices viáveis varia.

3. Abordagem Ótima com PD

Nessa seção, aborda-se o uso de programação dinâmica (PD) no Algoritmo 1, em decorrência da complexidade algorítmica apresentada por este, elucidada anteriormente na Subseção 2.1, além disso, desenvolvemos otimizações para esta abordagem no consumo de memória em grafos $K_{m,n}$ e P_n. Nesse sentido, inicialmente, propomos o Algoritmo 2, cuja diferença em relação ao Algoritmo 1 é a inserção da programação dinâmica (PD) para memoizar os resultados de estados (vértices restantes) anteriormente já calculados. O que permite adicionar a verificação se determinado subproblema já foi solucionado anteriormente para então reaproveitá-lo ou, caso contrário, resolver tal subproblema e armazenar seu respectivo $val(G)$. Com o objetivo de otimizar o desempenho da abordagem ótima, isto é, da abordagem que maximiza o ganho de Alice.

Para a realização dos experimentos mostrados abaixo nas Tabelas 2 e 3, foi utilizada a seguinte especificação computacional:

DOI: 10.59254/sbpo-2025-212396
https://proceedings.science/p/212396?lang=pt-br

- **CPU**: AMD Ryzen 7 5700X 8-Core, 16 threads;
- **GPU**: NVIDIA GeForce RTX 4060, 8GB;
- **Memória**: 32 GB DDR4 RAM;
- **Sistema Operacional**: Windows 10 64-bit; e
- **Ambiente de Desenvolvimento**: Jupyter Notebook 7.2.2, Python 3.12.7.

3.1. Grafos $K_{m,n}$

Os Algoritmos 1 e 2 resolvem o *Gold Grabbing* gerando a solução ótima utilizando a Equação (1). Nesta seção, serão submetidos grafos bipartidos completos $K_{m,n}$ como entrada dos algoritmos para realizar testes computacionais e, por fim, melhorias na versão de programação dinâmica. Essa versão, implementada pelo Algoritmo 2, teve seus tempos de execução reduzidos substancialmente para $K_{m,n}$ em relação à sua versão original, conforme mostra a Tabela 2. Cuja coluna "Sem PD" mostra os tempos para o Algoritmo 1, que utiliza apenas força bruta, enquanto que na "Sem R(t)" o Algoritmo 2, o qual implementa a mesma lógica, mas utilizando programação dinâmica.

Em contrapartida, deve ser ressaltado que o uso desta programação dinâmica, envolve um *trade-off* entre tempo computacional e uso de memória, pelo fato de economizar tempo de execução na resolução dos problemas ao reaproveitar soluções anteriormente armazenadas em memória. Acarretando, assim, no uso exponencial desta conforme a Fórmula 4, que representa a quantidade de índices na tabela para um grafo $K_{m,n}$, desconsiderando subproblemas de tamanho 1, visto que o $val(G)$ destes é o próprio peso do vértice.

$$E(m,n) = (2^m - 1) \times (2^n - 1) \tag{4}$$

É importante destacar que esta fórmula quantifica a quantidade de subproblemas presentes em um $K_{m,n}$ por representar a combinação entre todos os subconjuntos não vazios dos conjuntos M e N, o que é equivalente a mapear todas as combinações de jogos para o problema proposto. Consequentemente, por todos os subproblemas estarem sendo mapeados pela expressão, é intuitivo assumir que para a classe $K_{m,n}$ tem-se como limite inferior $\Omega(2^{m+n})$.

Portanto, fica nítido que ambos algoritmos, apesar de serem ótimos, não são eficientes, isto é, o Algoritmo 1 é exponencial em tempo, enquanto que o Algoritmo 2 é exponencial em memória. Em decorrência disso, propõe-se uma otimização para a abordagem por programação dinâmica, cujo objetivo é balancear o gasto em memória ao deletar estados que não serão mais reutilizados e identificar aqueles que nunca serão reutilizados. Essa otimização é modelada da seguinte maneira: Dado um grafo $K_{m,n}$, seja $R(t)$ o número máximo de reutilizações de um estado de tamanho t ao longo da execução do Algoritmo 2, temos que:

$$\mathrm{R}(t) = \begin{cases} m + n - t - 1, & \text{se } 2 \leq t \leq m + n - 2 \\ 0, \text{caso contrário} \end{cases} \tag{5}$$

$K_{3,3}$	t	2	3	4
	$R(t)$	3	2	1

Tabela 1: Exemplo de reutilizações dos estados para $K_{3,3}$.

DOI: 10.59254/sbpo-2025-212396
https://proceedings.science/p/212396?lang=pt-br

Esse padrão de reuso de estados elucidado pela Fórmula 5, acontece devido ao número de caminhos recursivos que levam a um estado de tamanho t. Visto que, pelo Algoritmo 2, um estado E qualquer de tamanho t surge após a remoção de $(m+n)-t$ vértices anteriores à sua recursão, tendo em mente que quando uma recursão em $G = K_{m,n}$ com $|G| = n+m$ acontece, uma nova instância do problema (subproblema) é criada com $|G| = n+m-1$. Nessa lógica, um subproblema (estado) de tamanho t só será alcançado após $(m+n)-t$ remoções de vértices.

Por exemplo, considere um grafo $K_{3,3}$ composto por duas partições $X = \{x_1, x_2, x_3\}$ e $Y = \{y_1, y_2, y_3\}$. Para que o estado de tamanho quatro $E = \{x_1, x_2, y_1, y_2\}$, apareça, foi necessária a remoção de dois $(3+3-4)$ vértices, $\{x_3, y_3\}$, consequentemente, tal estado E só pode surgir a partir de uma das sequências de remoção $\{(x_3, y_3)\text{e}(y_3, x_3)\}$, sendo que a primeira que acontecer será a responsável por criar o estado E e a segunda irá reutilizá-lo. Portanto, nesse exemplo, e como mostra a Tabela 1, o estado E será reutilizado uma única vez, isto é, $m+n-t-1$ vezes.

Desse modo, conforme mostra a Tabela 2 pela coluna "Tabela PD(Bytes)" que mostra o grau máximo de ocupação da tabela PD, quando "Pico Nº Estado" ocorre, o uso de $R(t)$ reduz o número de estados utilizados praticamente pela metade, o que é um ganho significativo em relação à abordagem tradicional. Mediante o gráfico presente na Figura 1b, que mostra a execução de ambas as versões do algoritmo para um $K_{m,n}$, fica nítido como ao longo da execução há um balanceamento na carga da tabela PD para a versão que utiliza $R(t)$, destacada pela cor verde, enquanto que a tradicional, em azul, cresce até o final da execução. Ademais, em decorrência da diminuição da tabela PD, o tempo de execução do algoritmo também diminui proporcionalmente, visto que há maior economia nas manipulações da estrutura.

$K_{m,n}$	**Sem PD**	**Sem R(t)**			**Com R(t)**		
	Tempo(s)	**Tempo(s)**	**Pico Nº Estados**	**Tabela PD (Bytes)**	**Tempo(s)**	**Pico Nº Estados**	**Tabela PD (Bytes)**
2x2	0.0010	0.0031	9	2488	0.0021	4	1344
2x3	0.0040	0.0128	21	5376	0.0088	12	3490
3x3	0.0240	0.0695	49	13072	0.0452	26	7280
3x4	0.3730	0.3324	105	27896	0.2261	58	15496
4x4	0.1600	1.7087	225	60112	1.1565	118	35272
4x5	12.0917	9.0948	465	125240	6.1743	246	75776
5x5	131.3322	40.7927	961	263568	27.3714	494	157144
5x6	1406.0846	193.8483	1953	547672	131.2134	1006	325088

Tabela 2: Comparação de desempenho entre Algoritmo 1 e Algoritmo 2 com e sem $R(t)$

3.2. Grafos P_n

Analogamente, esse *trade-off* também está presente para grafos caminho em que a execução do Algoritmo 2 acarreta no uso polinomial de memória, como mostra a Fórmula 6, que mostra quantos estados são gerados para um P_n, obtida pela combinação simples tomada à dois, desconsiderando subproblemas de tamanho 1, visto que o $val(G)$ destes é o próprio peso do vértice.

$$E(n) = \frac{(n^2 - n)}{2} \tag{6}$$

Dessa forma, a fim de diminuir esse *trade-off* de tempo por memória, as seguintes otimizações foram implementadas: Verificação do estado composto apenas por vértices inicialmente internos e a deleção do estado após reuso.

A primeira otimização consiste em memoizar apenas estados que são um caminho interno, isto é, dado um estado, este deve ser formado apenas por vértices inicialmente de grau dois. Por exemplo, na Figura 3 o estado $E_1 = \{v_2, v_3, v_4\}$ seria armazenado na tabela da PD, enquanto que

DOI: 10.59254/sbpo-2025-212396
https://proceedings.science/p/212396?lang=pt-br

os estados $E_2 = \{v_1, v_2, v_3\}$ e $E_3 = \{v_3, v_4, v_6\}$ não seriam armazenados pois não apresentam somente vértices internos. E a segunda consiste em deletar o estado que foi reusado, pois ele não será mais usado.

Ambas as otimizações são válidas, pois, em um grafo caminho, para qualquer estado (conjunto de vértices restantes) $V' \subseteq V(P_n)$, o subgrafo induzido $G' = P_n[V']$ é um subcaminho de P_n. Nesse subcaminho, existem dois vértices $v, u \in V'$ que formam o (v, u)-caminho (ou (u, v)-caminho) de tamanho $|V'|$, sendo esses os únicos momentos em que o estado aparece durante o jogo. Essa lógica é a análoga à explicação da Fórmula 5 na subseção anterior 3.1.

Assim, quando o estado não contém nenhum vértice de grau dois do grafo P_n, ele está necessariamente entre dois caminhos, e, consequentemente, surgirá uma vez como (v, u)-caminho e outra como (u, v)-caminho, sendo este o único caso de reuso do estado. Por outro lado, estados cuja extremidade contém pelo menos um vértice de grau um em P_n não apresentam esse fenômeno de duplicidade.

Com o uso dessas duas otimizações, é possível aprimorar a implementação do Algoritmo 2, evitando desperdício de espaço ao alocar apenas os estados realmente necessários e liberando-os logo após seu uso. Essa estratégia contribui significativamente para a redução do tamanho da tabela de programação dinâmica (PD). Sendo isso feito, sem perder nenhum reaproveitamento, como pode ser visto pelas colunas "Nº Reusos Dif" que mostram quantos estados distintos foram reutilizados.

A efetividade dessa otimização pode ser observada na Tabela 3, ao comparar as colunas "Pico Nº Estados" e "Nº Reusos". A primeira indica o número máximo de estados simultaneamente ocupados na tabela PD, enquanto a segunda mostra quantos desses estados foram efetivamente reutilizados. Essa abordagem transforma o crescimento do "Pico Nº Estados" de polinomial (na versão tradicional) para linear em relação a n, refletindo uma melhoria substancial na eficiência do algoritmo. Além disso, a redução no uso de memória também impacta diretamente o tempo de execução, como mostra a coluna "T(s)", já que a menor ocupação da tabela resulta em menos operações de leitura e escrita o que acelera o processamento.

A liberação de espaço na tabela PD fica nítida pela Figura 1a que mostra a evolução da ocupação desta em ambas as versões do algoritmo para um P_{30}. Nesse gráfico, a linha em azul representa a execução sem as otimizações, "PD Padrão", e a em verde a execução com ambas as otimizações,"PD Otimizado". Esta controla melhor o tamanho da tabela ao longo da execução, além de regredir na ocupação a partir do pico de 27 estados, enquanto que a "PD Padrão" cresce até o final do experimento.

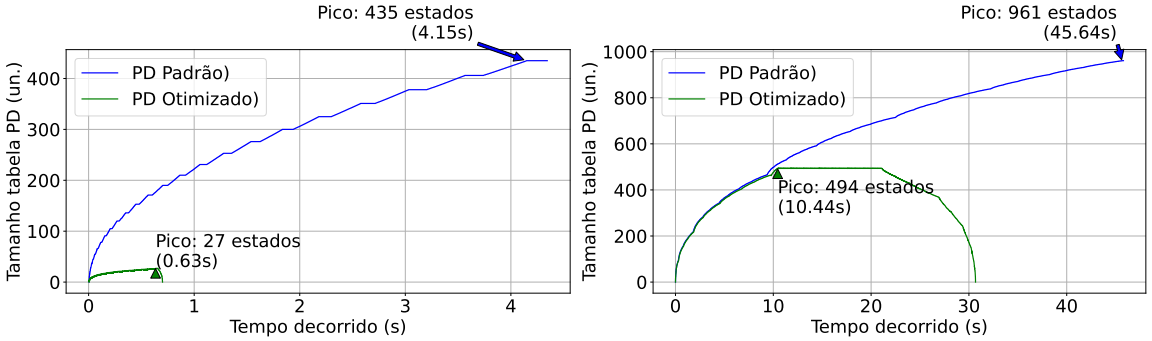
N	**valG(P_n)**					**valG(P_n) Otimizado**				
	T(s)	Nº Reusos	Nº Reusos Dif	Pico Nº Estados	Tabela PD (Bytes)	T(s)	Nº Reusos	Nº Reusos Dif	Pico Nº Estados	Tabela PD (Bytes)
5	0.003	3	10	10	2512	0.0020	3	10	2	704
10	0.055	28	45	45	12816	0.016	28	45	7	2296
15	0.21	78	105	105	31928	0.05	78	105	12	4472
20	0.73	153	179	179	65032	0.157	153	179	17	7528
25	1.94	253	300	300	104888	0.34	253	300	22	9968
30	4.4	756	435	435	168920	0.678	756	435	27	14352
50	52	1128	1225	1225	588752	5.07	1128	1225	47	33448
75	352.8	2628	2775	2775	1780272	22.5	2628	2775	72	63120
100	1425	4753	4950	4950	3755144	79	4753	4950	97	107504
150	9595	10878	11175	11175	11665312	343	10878	10878	147	216208

Tabela 3: Comparação de desempenho e uso de memória entre PD sem otimização e PD otimizado.

4. Abordagem Gulosa

Esta seção tem como objetivo explorar soluções gulosas para o problema do *Gold Grabbing*, isto é, elucidar a respeito do comportamento desse tipo de abordagem mais simplista para tipos específicos de grafos. Concluindo que essa não é uma estratégia vencedora em grafos caminho e é uma estratégia ótima para grafos completos, como K_n e $K_{m,n}$.

DOI: 10.59254/sbpo-2025-212396
https://proceedings.science/p/212396?lang=pt-br



(a) P_{30} (b) $K_{5,5}$

Figura 1: Comparação da evolução da ocupação da tabela PD para P_{30} e $K_{5,5}$.

4.1. Grafos P_n

Considere um grafo P_n, é intuitivo que para esse tipo de grafo Alice e Bob apresentam sempre duas jogadas possíveis, exceto quando há somente 1 vértice, podendo, assim, escolherem entre os vértices das extremidades. Dessa maneira, se Alice, como joga primeiro, faz uma escolha gulosa entre dois vértices $\{a, b\} \in V(P_n)$ e escolhe o vértice a, pois $w(a) \geq w(b)$, disponibiliza para Bob um vértice $c \in V(P_n)$ que pode ter peso maior que o de a. Logo, não há garantia de Alice sempre vencer a rodada de jogadas, para então vencer o jogo. Isso fica evidente na Figura 2, onde Alice escolheria os vértices $\{v_1, v_3, v_5\}$, enquanto Bob escolheria os vértices $\{v_2, v_4, v_6\}$, totalizando 101 e 201 pontos, respectivamente. Dessa forma, é possível afirmar que a abordagem gulosa não é vitoriosa.

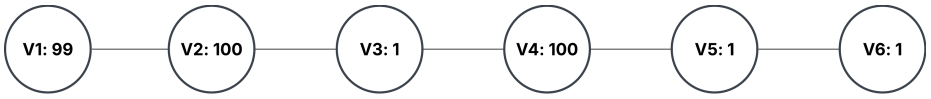


Figura 2: Contraexemplo para Guloso

4.2. Grafos K_n

Seja $G = K_n$ um grafo completo de n vértices. Por G ser completo, é conhecido que todos os seus vértices não são vértices de corte, pois qualquer par de vértices $(v, u) \in V(G)$ são adjacentes entre si. Desse modo, após cada i-ésima rodada, restarão $n - i$ vértices viáveis, até que todos os n vértices sejam removidos, visto que o subgrafo induzido $G' = K_n - v$ também é um grafo completo.

Logo, a abordagem gulosa para essa classe de grafos é vitoriosa e ótima, pois Alice, ao jogar primeiro, sempre poderá escolher o vértice de maior peso restante de G', e isso é sempre possível porque G' é um K_{n-i}. Como os vértices são removidos em ordem decrescente de peso, Alice obterá a soma dos maiores pesos disponíveis entre todos os vértices, enquanto Bob ficará com os menores. Assim, a gulosa maximiza a pontuação de Alice e garante pelo menos seu empate, no caso em que todos os vértices tenham o mesmo peso.

4.3. Grafos $K_{m,n}$

Agora, considere $G = K_{m,n}$ um grafo bipartido completo com partições A ($|A| = m$) e B ($|B| = n$), onde $m, n \geq 1$, tal que $m + n$ é par quando m ou n é igual a um, ou seja, quando

DOI: 10.59254/sbpo-2025-212396
https://proceedings.science/p/212396?lang=pt-br

não forma um caminho ímpar. Assim, utilizar uma abordagem gulosa, significa que Alice sempre pegará o vértice v de maior peso $w(v)$ caso este não seja o último vértice de sua partição, ou seja, não desconecta o grafo. Para provar que essa estratégia é vencedora, considere o seguinte lema:

Lema 1. *Considere um par de jogadas consecutivas, onde Alice remove o vértice v_A e Bob remove o vértice v_B em seguida. Então,*

$$w(v_B) \leq w(v_A).$$

Para provar o lema, segue-se que antes da jogada de Alice, o grafo está conexo e possui pelo menos um vértice em cada partição. Portanto, Alice pode escolher qualquer vértice, pois este não é de corte e pela abordagem gulosa Alice escolherá: $v_A = \max w(v) \mid v \in V$, assim, após a remoção de v_A, há duas situações:

- Se nenhuma partição ficou com exatamente um vértice, então todos os vértices remanescentes são válidos para remoção, e Bob escolherá o vértice de maior peso restante. Nesse caso, como v_A era o maior peso inicial, o maior peso remanescente será no máximo $w(v_A)$.
- Se uma partição (digamos A) ficou com exatamente um vértice e a outra (B) com pelo menos dois, então o único vértice proibido é o único remanescente de A. Então, Bob escolherá o maior peso dentre os vértices restantes em $V' \setminus A$, que ainda é no máximo $w(v_A)$.

Em ambos os casos, temos $w(v_B) \leq w(v_A)$, provando o Lema 1. Agora, considere que o número total de jogadas é $2k$ (pares de movimentos de Alice e Bob) ou $2k+1$ (se Alice realiza um movimento extra no final). Aplicando o lema a cada par de jogadas, obtemos

$$\sum_{i=1}^{k} w(v_{B,i}) \leq \sum_{i=1}^{k} w(v_{A,i}).$$

Se Alice ainda realiza um movimento adicional $v_{A,k+1}$, adiciona um peso positivo ou nulo à sua soma. Portanto, a soma total dos pesos coletados por Alice é maior ou igual à soma dos pesos coletados por Bob.

5. Abordagem por Bicoloração

Nessa seção do trabalho, aborda-se a utilização da bicoloração que é uma técnica baseada em colorir dois conjuntos disjuntos de vértices de um grafo, de modo a criar duas partições cujas adjacências acontecem apenas entre vértices de partições distintas, ou seja, vértices da mesma cor não são adjacentes. Devido a isso, caso seja possível dividir um grafo em duas partições de mesmo tamanho, torna-se viável aplicar uma estratégia vencedora para o *Gold Grabbing* com o seguinte intuito: Alice sempre pega vértices da cor que não perde, visto que obrigatoriamente alguma das cores acumula pelo menos metade dos pesos. A partir disso, conclui-se que essa abordagem é uma estratégia vitoriosa para grafos P_n e C_n.

5.1. Grafos P_n

Nesse sentido, considere um grafo caminho P_n par, tal que seus vértices são rotulados de v_1 a v_n e particionados em dois conjuntos $A = \{v_i \in A \mid 1 \leq i \leq n, \text{com i par}\}$ e $B = \{v_i \in B \mid 1 \leq i \leq n, \text{com i ímpar}\}$, de tal forma que a paridade funcione como cores distintas, obedecendo à regra de vizinhança para bicoloração.

Como P_n é caminho, em cada jogada (exceto a última) há dois vértices viáveis, não cortantes, sendo eles o v_i e $v_{n-(i-1)}$ e i a enumeração da jogada. Dessa forma, Alice, por ser a primeira a jogar, terá disponível na primeira jogada os vértices v_1 e v_n e caso ela escolha pegar o vértice v_1,

DOI: 10.59254/sbpo-2025-212396
https://proceedings.science/p/212396?lang=pt-br

Bob terá que necessariamente optar por dois vértices pares v_2 ou v_n. Assim ela sempre poderá escolher um vértice v_i e como ela sempre faz as jogadas de índice i ímpar ela poderá pegar todos os vértices do conjunto B. Caso ela escolha pegar o vértice v_n, Bob somente poderá pegar dois vértices ímpares v_1 ou v_{n-1} e como Alice faz as jogadas ímpares, ela poderá pegar todos os vértices pares $v_{n-(i-1)}$, pois $(i-1)$ será par e n é par. Logo $n-(i-1)$ é sempre par, resultando em Alice obtendo todo o conjunto A.

Portanto, a estratégia vencedora consiste na Alice escolher na primeira jogada o vértice que pertença ao conjunto vencedor A ou B, ou seja, escolher v_1 caso o somatório dos pesos de A seja maior que o de B e escolher v_n caso contrário. Isso acontece, pois como Alice joga primeiro sempre há um vértice par e um ímpar como opções, enquanto Bob possui apenas dois vértices da paridade contrária à última jogada de Alice, logo ela controla o jogo escolhendo o conjunto com a paridade vencedora.

Não obstante, essa abordagem não é uma estratégia ótima, isto é, não faz com que Alice acumule o máximo de ouro possível, visto que ela não se preocupa em prejudicar as jogadas de Bob, mas sim em apenas garantir pelo menos a metade do ouro. A figura 3 mostra um contraexemplo para a abordagem anterior a partir de um grafo P_6 onde Alice ficaria com os vértices $\{v_1, v_3, v_5\}$ acumulando oito de ouro e Bob acumularia seis de ouro. Contudo, caso o jogo obedecesse à estratégia ótima (Fórmula 2) ela escolheria $\{v_5, v_4, v_2\}$ totalizando nove de ouro e Bob cinco.



Figura 3: Contraexemplo para Bicoloração

5.2. Grafos C_n

A estratégia da bicoloração para grafos ciclos de tamanho n, também é uma estratégia vencedora, porém não ótima, e utiliza exatamente a mesma estratégia para grafos caminho, também sendo restrito a um grafo par. Isso acontece, pois dado um grafo $G = C_n$ por ter exatamente n vértices de grau dois e n arestas, ao ter um $v \in V(C_n)$ removido, o grafo induzido $G' = G - v$ resultante não será mais um ciclo, mas sim um P_{n-1}, pois se manterá conexo com $n-1$ vértices e $n-2$ arestas. E como v era necessariamente adjacente a vértices de cores diferentes da dele, o grafo P_{n-1} resultante tem como os dois primeiros vértices viáveis, vértices da cor oposta a v, limitando as jogadas de Bob a vértices da cor não escolhida por Alice, visto que ele será o primeiro a jogar no P_{n-1}.

Desse modo, bicolorir os vértices de um grafo C_n é uma condição suficiente para garantir a vitória de Alice, que precisa apenas na primeira jogada escolher um vértice da cor cuja partição tem maior peso atrelado, sendo, assim, uma estratégia vencedora. Analogamente à estratégia anterior para P_n, esta estratégia também não é ótima, e como, basta adicionar uma aresta entre os vértices v_1 e v_6 da Figura 3 que se obtém um contraexemplo para C_n por bicoloração.

6. Conclusão

O *Gold Grabbing Game* é um jogo em grafos com uma literatura ampla abordando aspectos matemáticos do problema, porém pouco foi estudado do ponto de vista computacional e assim existe uma lacuna na literatura neste ponto. Neste trabalho, abordamos e comparamos algumas estratégias computacionais para resolver o problema.

Estudando a Equação (1) para *Gold Grabbing Game* observamos que a árvore de recursão gerada é exponencial, assim, uma abordagem por força bruta considerando esta equação é inviável computacionalmente pois apresenta complexidade exponencial $O({v_i}^n)$. A partir disso, estudamos

DOI: 10.59254/sbpo-2025-212396
https://proceedings.science/p/212396?lang=pt-br

diversas técnicas de algoritmos para otimização computacional de tempo e espaço para solução ótima do problema usando a Equação (1) e uma que se mostrou interessante foi a Programação Dinâmica (PD). A princípio e para o caso geral, esta abordagem se mostrou eficiente em tempo, mas não em memória. Avaliando a PD conseguimos otimizar o gasto da memória quando considerados grafos caminho e bipartido completo.

Ademais, foram apresentadas as abordagens gulosa e por bicoloração, que se mostraram eficazes em classes específicas de grafos, tanto na perspectiva de estratégia vencedora quanto na busca por soluções ótimas. Observou-se que a estratégia gulosa não é vencedora em grafos do tipo caminho P_n, mas se mostrou eficiente para grafos completos K_n e bipartidos completos $K_{m,n}$, sendo inclusive ótima nestes últimos. Por outro lado, a abordagem baseada em bicoloração demonstrou garantir a vitória de Alice em grafos P_n e C_n, embora não maximize necessariamente o total de ouro obtido, mostrando-se, portanto, uma estratégia vencedora mas não ótima.

Como direções futuras, propõe-se investigar a existência de estratégias vencedoras por bicoloração em outras classes de grafos, como árvores arbitrárias. Além disso, deixamos em aberto a ampliação das otimizações por programação dinâmica para novas classes de grafos, com o objetivo de identificar padrões de reutilização de estados que permitam maior eficiência no uso de memória e tempo de execução.

7. Algoritmos

Algoritmo 1: $val(G)$

```
1  Função Valor(Grafo, Pesos):
2      se Grafo não possui nós então
3          retorna (0, [])
4      Melhor_Valor ← −∞;
5      Melhor_Escolha ← None;
6      Melhor_Caminho ← [ ];
7      para cada v em Lista_de_Nós(Grafo) faça
8          se Eh_Viavel(Grafo, v) então
9              Grafo_Copia ← cópia de Grafo;
10             Remover nó v de Grafo_Copia;
11             (Valor_Subgrafo,Sub_Melhor_Caminho) ← Valor (Grafo_Copia,
                Pesos);
12             Valor_Atual ← Pesos[v] − Valor_Subgrafo;
13             se Valor_Atual > Melhor_Valor então
14                 Melhor_Valor ← Valor_Atual;
15                 Melhor_Escolha ← v;
16                 Melhor_Caminho ← [v] + Sub_Melhor_Caminho;
17     retorna (Melhor_Valor, Melhor_Caminho)
```

DOI: 10.59254/sbpo-2025-212396
https://proceedings.science/p/212396?lang=pt-br

Algoritmo 2: $val(G)$ com Programação Dinâmica

```
1  Função Valor_DP (Grafo, Pesos, Memo) :
2      se Memo é None então
3          Memo ← dicionário vazio;
4      Estado ← tupla dos nós de Grafo;
5      se Estado está em Memo então
6          retorna Memo[Estado];
7      se Grafo não possui nós então
8          retorna (0, [ ]);
9      Melhor_Valor ← −∞;
10     Melhor_Caminho ← [ ];
11     para cada v em Lista_de_Nós(Grafo) faça
12         se Eh_Viavel(Grafo, v) então
13             Grafo_Copia ← cópia de Grafo;
14             Remover nó v de Grafo_Copia;
15             Valor_Subgrafo,Sub_Melhor_Caminho ← Valor_DP (Grafo_Copia,
                Pesos, Memo);
16             Valor_Atual ← Pesos[v] − Valor_Subgrafo;
17             se Valor_Atual > Melhor_Valor então
18                 Melhor_Valor ← Valor_Atual;
19                 Melhor_Caminho ← [v] + Sub_Melhor_Caminho;
20             se número de nós em Grafo > 1 então
21                 Memo[Estado] ← (Melhor_Valor, Melhor_Caminho);
22                 Adicionar Estado a Estados_Na_Tabela;
23     retorna (Melhor_Valor, Melhor_Caminho)
```

DOI: 10.59254/sbpo-2025-212396
https://proceedings.science/p/212396?lang=pt-br

Referências

Boriboon, S. e Kittipassorn, T. (2020). The graph grabbing game on blow-ups of trees and cycles. URL https://arxiv.org/abs/2007.11805.

Chaplick, S., Micek, P., Ueckerdt, T., e Wiechert, V. (2016). A note on concurrent graph sharing games. *Integers*, 16:G1. ISSN 1867-0652. DBLP's bibliographic metadata records provided through http://dblp.org/search/publ/api are distributed under a Creative Commons CC0 1.0 Universal Public Domain Dedication. Although the bibliographic metadata records are provided consistent with CC0 1.0 Dedication, the content described by the metadata records is not. Content may be subject to copyright, rights of privacy, rights of publicity and other restrictions.

Cibulka, J., Kynčl, J., Mészáros, V., Stolař, R., e Valtr, P. (2013). Graph sharing games: Complexity and connectivity. *Theoretical Computer Science*, 494:49–62. ISSN 0304-3975. URL https://www.sciencedirect.com/science/article/pii/S0304397512011449. Theory and Applications of Models of Computation (TAMC 2010).

Egawa, Y., Enomoto, H., e Matsumoto, N. (2018). The graph grabbing game on km,n-trees. *Discrete Mathematics*, 341(6):1555–1560. ISSN 0012-365X. Publisher Copyright: © 2018 Elsevier B.V.

Matsumoto, N., Moriyama, R., e Ota, K. (2022). Graph grabbing game on totally-weighted graphs. *Discrete Applied Mathematics*, 322:384–390. ISSN 0166-218X. URL https://www.sciencedirect.com/science/article/pii/S0166218X22003547.

Micek, P. e Walczak, B. (2011). A graph-grabbing game. *Combinatorics, Probability and Computing*, 20(4):623–629.

Seacrest, D. E. e Seacrest, T. (2012). Grabbing the gold. *Discrete Mathematics*, 312(10):1804–1806.

Winkler, P. (2004). Mathematical puzzles: A connoisseur's collection. *AK Peters*, 143:149–151.

DOI: 10.59254/sbpo-2025-212396
https://proceedings.science/p/212396?lang=pt-br