Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de *Software*

Big Points: Uma Análise Baseada na Teoria dos Jogos

Autor: Mateus Medeiros Furquim Mendonça
Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF
2017

Mateus Medeiros Furquim Mendonça

Big Points: Uma Análise Baseada na Teoria dos Jogos

Monografia submetida ao curso de graduação em Engenharia de *Software* da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de *Software*.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2017

Mateus Medeiros Furquim Mendonça
Big Points: Uma Análise Baseada na Teoria dos Jogos/ Mateus Medeiros Furquim Mendonça. – Brasília, DF, 2017-
63 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2017.

1. Teoria dos Jogos. 2. Análise Combinatória de Jogos. I. Prof. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. *Big Points*: Uma Análise Baseada na Teoria dos Jogos

CDU 02:141:005.6

Mateus Medeiros Furquim Mendonça

Big Points: Uma Análise Baseada na Teoria dos Jogos

Monografia submetida ao curso de graduação em Engenharia de *Software* da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de *Software*.

Trabalho aprovado. Brasília, DF, 7 de julho de 2017:

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Prof. Dr. Fábio Macêdo Mendes
Convidado 1

Prof. Dra. Carla Silva Rocha Aguiar
Convidado 2

Brasília, DF
2017

Resumo

A Teoria dos Jogos estuda as melhores estratégias dos jogadores em uma determinada situação de conflito. Este trabalho faz uso do teorema *minimax* para solucionar versões reduzidas do jogo *Big Points* com o propósito de investigar o balanceamento do jogo, que foi reduzido em relação ao tipo e quantidade de certas peças. Utilizando-se técnicas de memorização, são implementadas duas funções para separar a lógica do jogo da lógica da programação dinâmica. Os resultados após a escrita do código, a execução do programa e compilação dos dados em um gráfico de barras tridimensional, sugerem que o jogo de *Big Points* completo seja desbalanceado.

Palavras-chaves: Teoria dos Jogos, Análise Computacional dos Jogos, Programação Dinâmica, Teorema Minimax, Balanceamento de Jogos.

Abstract

The Game Theory field studies the best strategies of players where there is a conflict situation. This paper utilizes the minimax theorem to solve some reduced versions of a game called *Big Points*. Its goal is to investigate whether the game is well balanced. The game was simplified regarding its pieces' quantities and some specific types of pieces.. Using the memoization technique, we implemented two functions to separate the game's logic from the dynamic programming's logic. After writting the code, execute it, and compile the results into a tridimensional bar plot, the results suggest that the complete game of *Big Points* might not be balanced after all.

Key-words: Game Theory, Computational Game Theory, Dynamic Programming, Theorem Minimax, Game Balance.

Lista de ilustrações

Lista de tabelas

Lista de símbolos

	Símbolos para conjuntos e operações matemáticas
$\varnothing$	O conjunto vazio
$\{\ \}$	Delimita conjunto, de forma que $S = \{\}$ é um conjunto vazio
$[a, b]$	Intervalo fechado, inclui a e b
(a, b)	Intervalo aberto, não inclui a e b
$\forall$	Para cada elemento
$x \in S$	x pertence ao conjunto S
$x \notin S$	x não pertence ao conjunto S
$\sum_{i=1}^{n} x_i$	Somatório de x_1 até x_n
$\prod_{i=1}^{n} x_i$	Produtório de x_1 até x_n
$A_{p,q}$	Arranjo de p elementos tomados de q a q
$\binom{p}{q}$	Combinação de p elementos tomados de q a q
$\lceil x \rceil$	Arredonda o valor de x para o menor valor inteiro maior ou igual a x
$\lfloor x \rfloor$	Arredonda o valor de x para o maior valor inteiro menor ou igual a x

	Símbolos para jogos de soma zero com dois jogadores
J_1	Primeiro Jogador
J_2	Segundo Jogador
a	Quantidade de estratégias pura do jogador J_1
b	Quantidade de estratégias pura do jogador J_2
σ_i	Estratégias pura do jogador J_1, com $i \in \{1, \ldots, a\}$
τ_j	Estratégias pura do jogador J_2, com $j \in \{1, \ldots, b\}$
$P_n(\sigma_i, \tau_j)$	Ganho do jogador J_n, com $n \in \{1, 2\}$

Sumário

Introdução

Imagine que um grupo de pessoas concordam em obedecer certas regras e agir de forma individual, ou em grupos menores, sem violar as regras especificadas. Suas ações como um todo, no final, levarão a uma situação chamada resultado. Os membros do grupo são chamados de jogadores e as regras que concordaram em obedecer constituem um jogo. Imagine agora que, devido às regras do jogo,

Objetivos

O objetivo principal deste trabalho é realizar uma análise *minimax* nas versões reduzidas do jogo *Big Points*. O jogo foi reduzido em relação ao tipo e quantidade de certas peças, pois para analisar o jogo completo exigiria um trabalho computacional imenso.

Justificativa

A pergunta que motivou o desenvolvimento deste projeto foi a questão do balanceamento do jogo *Big Points*. Isto é, se os jogadores jogarem de forma ótima, a chance de vitória é a mesma para todos os jogadores? A partir da análise investigativa do balanceamento de um jogo aparentemente simples como o *Big Points*, pode-se fornecer recursos para a construção de programas ou modelos para análise de balanceamento de estruturas mais complexas, aplicáveis também a áreas de Teoria dos Jogos como biologia, política e economia.

Organização do Trabalho

Este trabalho foi dividido em quatro capítulos. O primeiro capítulo, Fundamentação Teórica, relata um pouco sobre a história da teoria dos jogos, esclarece alguns conceitos relevantes para o entendimento do trabalho, e explica as regras do próprio jogo. Em seguida, tem-se o Capítulo 2, referente à análise e ao desenvolvimento do projeto até sua conclusão, e no Capítulo 3 os resultados desta análise são discutidos. Por último, o Capítulo 4 onde são feitas as considerações finais do trabalho e são citados alguns possíveis trabalhos futuros a partir do trabalho atual.

1 Fundamentação Teórica

Para um bom entendimento da análise realizada no jogo *Big Points* é preciso um conhecimento básico sobre teoria dos jogos e programação dinâmica. A primeira seção deste capítulo conta brevemente sobre a história da Teoria dos Jogos, com alguns nomes icônicos desta área. A Seção 1.2 explica um pouco sobre os conceitos da Teoria dos Jogos, mas apenas o necessário para o entendimento deste trabalho. Na Seção 1.3, são apresentados os conceitos sobre programação dinâmica e, na última seção, as regras do jogo *Big Points* são explicadas.

1.1 Histórico da Teoria dos Jogos

Pode-se dizer que a análise de jogos é praticada desde o século XVIII, tendo como evidência uma carta escrita por James Waldegrave ao analisar uma versão curta de um jogo de baralho chamado *le Her* (PRAGUE, 2004). No século seguinte, o matemático e filósofo Augustin Cournot fez uso da teoria dos jogos para estudos relacionados à política (COURNOT, 1838 apud SARTINI et al., 2004).

Mais recentemente, em 1913, Ernst Zermelo publicou o primeiro teorema matemático da teoria dos jogos (ZERMELO, 1913 apud SARTINI et al., 2004). Outros dois grandes matemáticos que se interessaram na teoria dos jogos foram Émile Borel e John von Neumann. Nas décadas de 1920 e 1930, Emile Borel publicou vários artigos sobre jogos estratégicos (BOREL, 1921 apud PRAGUE, 2004) (BOREL, 1924 apud PRAGUE, 2004) (BOREL, 1927 apud PRAGUE, 2004), introduzindo uma noção abstrata sobre jogo estratégico e estratégia mista.

Em 1928, John von Neumann provou o teorema *minimax*, que diz que há sempre uma solução racional para um conflito bem definido entre dois indivíduos cujos interesses são completamente opostos (NEUMANN, 1928 apud ALMEIDA, 2006). Em 1944, Neumann publicou um trabalho junto a Oscar Morgenstern introduzindo a teoria dos jogos na área da economia e matemática aplicada (NEUMANN; MORGENSTERN, 1944 apud SARTINI et al., 2004). Além destas contribuições, John von Neumann ainda escreveu trabalhos com grande impacto na área da computação, incluindo a arquitetura de computadores, princípios de programação, e análise de algoritmos (MIYAZAWA, 2010).

Um dos principais nomes da história da Teoria dos Jogos é John Forbes Nash Junior, um matemático estadunidense que conquistou o prêmio Nobel de economia em 1994. Foi formado pela Universidade de Princeton, em 1950, com a tese *Non-Cooperative Games* (Jogos Não-Cooperativos, publicada em 1951) (NASH, 1950b apud ALMEIDA, 2006). Nesta tese, Nash provou a existência de ao menos um ponto de equilíbrio em jogos de estratégias para múltiplos jogadores, mas para isso é necessário que os jogadores se comportem racionalmente (ALMEIDA, 2006).

O equilíbrio de Nash era utilizado apenas para jogos de informação completa. Posteriormente, com os trabalhos de Harsanyi e Selten, foi possível aplicar este método em jogos de informação incompleta. A partir de então, surgiram novas técnicas de solução de jogos e a teoria dos jogos passou a ser aplicada em diferentes áreas de estudo, como na economia, biologia e ciências políticas (ALMEIDA, 2006).

Entre 1949 e 1953, Nash escreveu mais artigos ligados à solução de jogos estratégicos: *The Bargaining Problema* (O Problema da Barganha, 1949) (NASH, 1950a apud ALMEIDA, 2006) e *Two-Person Cooperative Games* (Jogos Cooperativos de Duas Pessoas, 1953) (NASH, 1953 apud ALMEIDA, 2006). Também escreveu artigos de matemática pura sobre variedades algébricas em 1951, e de arquitetura de computadores em 1954 (ALMEIDA, 2006).

Mais recentemente, dois trabalhos se destacaram na área de Teoria dos Jogos: o livro de Thomas Schelling, publicado em 1960, que se destacou em um ponto de vista social (SCHELLING, 1960 apud CARMICHAEL, 2005); e um livro de dois volumes de Elwyn Berlekamp, John Conway e Richard Guy que se tornou uma referência na área da teoria dos jogos combinatorial por explicar os conceitos fundamentais para a teoria dos jogos combinatorial (BERLEKAMP; CONWAY; GUY, 1982 apud GARCIA; GINAT; HENDERSON, 2003).

1.2 Teoria dos Jogos

A Teoria dos Jogos pode ser definida como a teoria dos modelos matemáticos que estuda a escolha de decisões ótimas[1] sob condições de conflito[2]. Atualmente, o campo da teoria dos jogos divide-se em três áreas: Teoria Econômica dos Jogos, que normalmente analisa movimentos simultâneos (Definição 1) de dois ou mais jogadores; Teoria Combinatória dos Jogos, no qual os jogadores fazem movimentos alternadamente, e não faz uso de elementos de sorte, diferente da Teoria Econômica dos Jogos que também trata desse fenômeno; e Teoria Computacional dos Jogos, que engloba jogos que são possíveis resolver por força bruta ou inteligência artificial (GARCIA; GINAT; HENDERSON, 2003), como jogo da velha e xadrez, respectivamente. Nestre trabalho serão utilizados alguns conceitos da Teoria Econômica dos Jogos para analisar um jogo de movimentos alternados, a ser resolvido computacionalmente.

---

1 É considerado que os jogadores são seres racionais e possuem conhecimento completo das regras.

2 Condições de conflito são aquelas no qual dois ou mais jogadores possuem o mesmo objetivo.

Definição 1. Em jogos com **movimentos simultâneos**, os jogadores devem escolher o que fazer ao mesmo tempo ou as escolhas de cada jogador são escondida de seu oponente. Em qualquer um dos dois casos, o jogador deve escolher sua jogada levando em consideração a possível jogada do adversário (CARMICHAEL, 2005).

Os elementos básicos de um jogo são: o conjunto de jogadores; o conjunto de estratégias para cada jogador; uma situação, ou perfil, para cada combinação de estratégias dos jogadores; e uma função utilidade para atribuir um *payoff*, ou ganho, para os jogadores no final do jogo. Os **jogadores** J são dois ou mais seres racionais que possuem um mesmo objetivo e, para alcançar esse objetivo, cada jogador possui um conjunto S de **estratégias**. A partir das escolhas de estratégias de cada jogador, tem-se uma **situação** ou **perfil** e, no final do jogo, um **resultado** para cada perfil (SARTINI et al., 2004). Em outras palavras, os jogadores escolhem seus movimentos simultaneamente, como explicado na Definição 1, o que levará a vitória de algum deles no final do jogo, ou a um empate.

Em termos matemáticos é dito que um jogo Γ é composto por um conjunto J de jogadores, que por sua vez possuem um conjunto de estratégia S. Além disso, cada jogador tem uma **função utilidade** P_i, que atribui um ***payoff***, ou **ganho**, para cada situação do jogo, como mostra a Equação 1.

$$\begin{aligned} \Gamma &= < J, S, P > \\ J &= \{j_1, j_2\} \\ S_i &= \{s_{i1}, s_{i2}, ..., s_{ij}\}, \\ \forall j_i \in J, &\ \exists\ s_{ij} \in S_i \\ P_i &: S_i \rightarrow \mathbb{R} \end{aligned} \tag{1}$$

Quando essa informação do ganho é inserida em uma matriz, tem-se uma **matriz de *payoff*** (SARTINI et al., 2004). Em outras palavras, a matriz de ganho é a representação matricial dos *payoffs* dos jogadores, onde as estratégia de um jogador estão representadas por cada linha e as de seu oponente estão representadas pelas colunas.

Para um melhor entendimento destes conceitos, será utilizado uma versão curta do jogo *Nim*. Esta versão simplificada do jogo começa com quatro palitos e dois montes (com dois palitos cada monte). Cada um dos dois jogadores joga alternadamente retirando quantos palitos quiser, mas de apenas um dos montes. O jogador que retirar o último palito do jogo perde (JONES, 1980).

Começando com o conceito de abstração e representação de um jogo, existe uma maneira de fazê-la chamada forma extensiva, a qual é descrita na Definição 2. De acordo com esta definição, a árvore do jogo *Nim* simplificado é representada na Figura 1.

Definição 2. É dito que um jogo está representado na sua **forma extensiva** se a árvore do jogo reproduzir cada estado possível, junto com todas as possíveis decisões que levam a este estado, e todos os possíveis resultados a partir dele (JONES, 1980, grifo nosso). Os nós são os estados do jogo e as arestas são as possíveis maneiras de alterar aquele estado, isto é, os movimentos permitidos a partir daquele estado.

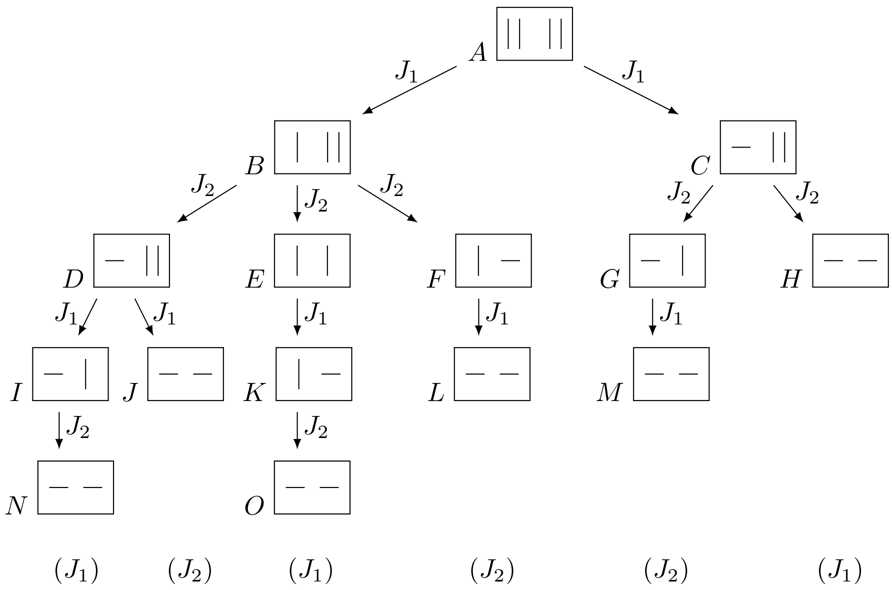


Figura 1 – Árvore do jogo *Nim*. Fonte: (JONES, 1980)

A ordem dos jogadores está sendo indicada ao lado esquerdo da figura, de forma que o jogador J_1 é o primeiro a realizar um movimento, o J_2 é o segundo, o terceiro movimento é do J_1 e assim por diante. O estado do jogo é representado por cada nó da árvore, sendo que os quatro palitos estão divididos em dois montes dentro do retângulo. Cada aresta representa uma jogada válida para o jogador que vai realizar o movimento (jogador atual). Ao analisar a primeira jogada, percebe-se que J_1 possui quatro jogadas possíveis: retirar um palito do primeiro monte; retirar dois palitos do primeiro monte; retirar um palito do segundo monte; e retirar dois palitos do segundo monte. Por serem simétricas às duas primeiras, as últimas duas jogadas foram omitidas da árvore do jogo Na aresta (A, B)[3], o primeiro jogador pega apenas um palito de um dos montes de palito, enquanto a aresta (A, C) representa o movimento de pegar todos os dois palitos de um monte. Da mesma maneira, as arestas (B, D), (B, E), (B, F), (C, G) e (C, H) são os movimentos de J_2 em resposta às jogadas de J_1.

No final da Figura 1, há uma representação para cada folha[4] para indicar o vencedor no final daquela série de movimentos. Nos nós terminais N, O e H, o jogador J_2

[3] A aresta representada como (A, B), sai do nó A ao nó B. Uma notação alternativa seria $\overrightarrow{B}$, sendo a aresta que incide em B (ADELSON-VELSKY; ARLAZAROV; DONSKOY, 1988).

[4] Um nó é considerado folha (ou nó terminal) quando não possuir nenhum filho.

retirou o último palito do jogo, resultando na vitória de J_1. Para as folhas J, L e M, a vitória é do segundo jogador.

Olhando para a árvore de baixo pra cima, o jogador J_1 ganha na folha N. Na verdade, ele já havia ganhado no nó anterior (I), pois o jogador J_2 só tem uma jogada a fazer. Como a decisão de chegar no nó I é de escolha do primeiro jogador ao realizar a jogada (D, I), pode-se dizer que essa jogada é um *winning move*[5].

Ao mesmo tempo que J_1 é um jogador inteligente que tenta sempre jogar da melhor maneira possível, o segundo jogador também fará as melhores jogadas que puder. Sabendo que o nó D garante sua derrota, J_2 fará de tudo para escolher outras jogadas. De fato, ao observar essa árvore com cuidado, o jogador J_2 sempre irá vencer, pois há sempre um nó no qual, a partir dele, lhe garante à vitória. Para entender melhor o por quê do jogador J_2 sempre ganhar, será utilizado uma análise partindo do conceito de estratégia pura (Definição 3).

Definição 3. Estratégia pura é definida como um conjunto de decisões a serem feitas para cada ponto de decisão no jogo (JONES, 1980, grifo nosso).

As estratégias pura do jogador J_1 são nomeadas σ_i com $i \in \{1, \ldots, a\}$ e as do jogador J_2 são representadas por τ_j com $j \in \{1, \ldots, b\}$, onde a e b são a quantidade de estratégias pura de J_1 e J_2, respectivamente. A estratégia pura também pode ser vista como um caminho[6] único na árvore, que tem origem no primeiro nó de decisão do jogador e vai até o último nó de decisão do mesmo jogador. No caso do jogador J_1, o caminho começa na raíz, e no caso do jogador J_2, o caminho pode começar em B ou em C. Devido à isso, J_2 deve considerar os dois casos e decidir de antemão o que fazer. A partir da Definição 3, tem-se as estratégias de ambos os jogadores nas Tabelas 1 e 2.

Tabela 1 – Estratégias puras σ_i de J_1 para o jogo *Nim*. Fonte: (JONES, 1980)

Estratégia	**1º Turno**	**2º Turno**	
		Se em	Vá para
σ_1	$A \to B$	D	I
σ_2	$A \to B$	D	J
σ_3	$A \to C$	–	–

Na Tabela 1, os movimentos de J_1 estão separadas em dois turnos. O primeiro turno é o nó raiz (A). A partir deste estado, o jogador possui duas escolhas (A, B) ou (A, C), representados na tabela como as estratégias pura σ_1 e σ_3. Mas além dessa informação, ainda deve-se representar a próxima decisão a ser feita após escolher (A, B). Se J_2 escolher

[5] Movimento que garante a vitória.

[6] Uma sequência de arestas onde o nó no final de uma aresta coincide com o nó no começo da próxima aresta, é chamado de **caminho** (ROSENTHAL, 1972, grifo nosso).

Tabela 2 – Estratégias puras τ_j de J_2 para o jogo *Nim*. Fonte: (JONES, 1980)

Estratégia	**1º Turno**	
	Se em	Vá para
τ_1	B C	D G
τ_2	B C	E G
τ_3	B C	F G
τ_4	B C	D H
τ_5	B C	E H
τ_6	B C	F H

certos movimentos que chegue no D, o primeiro jogador ainda tem mais uma escolha a fazer. Essa segunda escolha está representada nas colunas: *Se em*, no caso se o jogador estiver naquele nó; e *Vá para*, que são as possíveis jogadas a serem feitas a partir do nó em questão. Então, a diferença de σ_1 e σ_2 é apenas nesta segunda escolha. Ao chegar em um nó terminal, acaba também a descrição de uma estratégia pura.

Definição 4. Considere um jogo no qual o jogador J_1 move primeiro e, a partir de então, ambos os jogadores alternam as jogadas. Ao chegar em um nó terminal, tem-se uma função para atribuir um valor ao jogador J_1 naquela folha. Essa sequência de movimento é chamado de **jogo**, e o valor na folha é chamado **resultado do jogo** (ADELSON-VELSKY; ARLAZAROV; DONSKOY, 1988, p. 2).

De acordo com a definição de um jogo (Definição 4), a versão reduzida do *Nim* possui dezoito jogos no total, de forma que a quantidade de jogos pode ser calculado como $ab = 18$, com $a = 3$ e $b = 6$. Alguns exemplos são mostrados a seguir:

$$\sigma_1 \text{ e } \tau_1 \text{ resultam no jogo } A \to B \to D \to I \to N,$$
$$\sigma_2 \text{ e } \tau_1 \text{ resultam no jogo } A \to B \to D \to J,$$
$$\sigma_3 \text{ e } \tau_2 \text{ resultam no jogo } A \to C \to G \to M.$$

Olhando para a tabela do jogador J_2 (Tabela 2), sua primeira jogada já depende da jogada do jogador J_1. Por isso, cada estratégia τ_j com $j \in \{1, \cdots, b\}$ descreve duas possibiliades de movimento. Observando τ_1, no primeiro turno seu movimento será (B, D) se estiver em B, caso contrário, jogará (C, G).

Definição 5. A **forma normal** é a representação do resultado do jogo a partir das escolhas de estratégia pura dos jogadores, onde, ciente das regras do jogo, cada jogador seleciona uma estratégia pura sem saber a escolha do outro.

Ao escolher suas estratégias pura, os jogadores percorrem a árvore até chegar a uma folha. Essa sequência de movimentos (a escolha de uma estratégia pura σ_i e uma τ_j) é chamada de jogo. Para cada combinação de estratégias de estratégias de J_1 e J_2, tem-se um jogo diferente. Esses diferentes jogos são representados pela análise da forma normal (Definição 5) na Tabela 3.

Tabela 3 – Forma Normal para o jogo *Nim*. Fonte: (JONES, 1980)

				$\mathbf{J_2}$			
		τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
	σ_1	N	O	L	N	O	L
$\mathbf{J_1}$	σ_2	J	O	L	J	O	L
	σ_3	M	M	M	H	H	H

Nesta tabela, as estratégias dos jogadores estão nas linhas e colunas, e as letras representam as folhas, que são os resultados de caminhos tomados a partir de cada estratégia σ_i e τ_j. Cada linha é uma estratégia pura de J_1 ($\sigma_i, i \in \{1,2,3\}$) e, cada coluna, uma estratégia de J_2 ($\tau_j, j \in \{1,2,3,4,5,6\}$). Para transformar esta tabela em uma matriz de *payoff*, basta substituir os nós terminais pelo ganho do jogo. Se o primeiro jogador ganhar, seu ganho é 1, e se o segundo jogador vencer, o resultado para J_1 é -1.

Tabela 4 – Matriz de Ganho para o jogo *Nim*. Fonte: (JONES, 1980)

				$\mathbf{J_2}$			
		τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
	σ_1	1	1	-1	1	1	-1
$\mathbf{J_1}$	σ_2	-1	1	-1	-1	1	-1
	σ_3	-1	-1	-1	1	1	1

Após a representação pela matriz de ganho, é escrito no final de cada coluna, o seu maior valor, que indica o melhor caso para J_1. Desses melhores casos, o objetivo de J_2 é diminuir o ganho de J_1. De forma similar, é escrito no final de cada linha, o seu menor valor, indicando o pior caso para J_1. De seus piores casos, o jogador 1 quer maximiziar sua pontuação.

Para encontrar a solução do jogo, encontre o valor mínimo da linha *máximo das colunas* (*minimax*) e o valor máximo da coluna *mínimo das linhas* (*maximin*). Tanto o *maximin* quanto o *minimax* são valores negativos, o que leva sempre à vitória de J_2. Dessa forma, pode-se ver na Tabela 5 que a estratégia τ_3 sempre garante a vitória para

Tabela 5 – Análise *minimax* para o jogo *Nim*

		$\mathbf{J_2}$						
		τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	mínimo das linhas
	σ_1	1	1	-1	1	1	-1	-1
$\mathbf{J_1}$	σ_2	-1	1	-1	-1	1	-1	-1
	σ_3	-1	-1	-1	1	1	1	-1
máximo das colunas		1	1	**-1**	1	1	1	

J_2 independente da estratégia do jogador J_1. Isso torna o jogo desbalanceado, pois para um jogo ser balanceado não deve ter uma estratégia dominante, que em outras palavras quer dizer que não deve haver uma estratégia que um jogador sempre ganha.

1.3 Programação dinâmica

Programação dinâmica é uma técnica de programação capaz de reduzir significantemente o tempo de processamento de um problema no qual os estados possam se repetir (CORMEN et al., 2001). Um exemplo clássico é o programa de para calcular os números da sequência de *Fibonacci*, onde os dois primeiros elementos valem um ($F_1 = F_2 = 1$) e os próximos elementos são calculados com a soma dos dois anteriores, de forma que $F_3 = F_2 + F_1 = 2$ e assim por diante. Dependo da implementação do problema, o tempo de processamento para chegar no resultado desejado pode crescer exponencialmente.

Nos Códigos 1, 2, 3 e 4 é demonstrado a diversas implementações para este problema e, na Figura 3, está representado em gráfico o tempo de cálculo do n-ésimo termo de *fibonacci*. Os valores da sequência de *Fibonacci* foram conferidos no site da enciclopédia online das sequências de números inteiros[7].

Código 1 – Funcao main de Fibonacci

```
#include <iostream> // std::cout
#include <map>       // std::map (PD)

// Protótipo (declaração) da função
int fibonacci(int);

int main()
{
    // Calcula e escreve o décimo quinto termo
    std::cout << fibonacci(15) << std::endl;

    return 0;
}
```

[7] *The Online Encyclopedia of Integers Sequences (OEIS)*, sequência A000045 no link https://oeis.org/A000045/a000045_3.txt

O Código 1 mostra a função principal do código para chamar a função de fibonacci, calculando o décimo quinto elemento da sequência. As diferentes implementações seguem nos código seguintes. No Código 2, foi implementado a sequência de *Fibonacci* com o cálculo iterativo. Os Códigos 3 e 4 utilizam recursão, mas o segundo deles faz uso da memorização dos termos para evitar calculá-los novamente.

Código 2 – Fibonacci Iterativo

```
int fibonacci(int n)
{
    // Declara e inicia a variável
    int fib_number = 0;

    // Os dois primeiros termos são iguais a 1
    int a_1 = 1;
    int a_2 = 1;
    for (int i = 1; i < n; i++) {
        // a_n é igual a soma dos dois termos anteriores
        fib_number = a_1 + a_2;

        // Atualiza os termos
        a_1 = a_2;
        a_2 = fib_number;
    }

    return fib_number;
}
```

O Código 2 calcula a sequência de forma iterativa. Tem-se os valores dos dois primeiros termos `a_1` e `a_2` e, dentro do *loop*, é calculado os próximos termos até o n-ésimo termo.

Código 3 – Fibonacci Recursivo

```
int fibonacci(int n)
{
    // Declara e inicia a variável
    int fib_number = 0;

    if (n <= 2) {
        // Os dois primeiros termos são iguais a 1
        fib_number = 1;
    }
    else {
        // Cada número em seguida são a soma dos dois anteriores
        fib_number = fibonacci(n-1) + fibonacci(n-2);
    }

    return fib_number;

}
```

O cálculo recursivo (Código 3) precisa de um caso base, onde a chamada recursiva vai parar de chamar a si mesma e retorna os valores dos dois primeiros termos. Caso `n`

`> 2`, é realizado a soma dos dois termos anteriores para descobrir o número atual (`n`) da sequência. Por fim, o valor é retornado para as funções anteriores que foram chamadas recursivamente até a função `fibonacci(15)` na função principal.

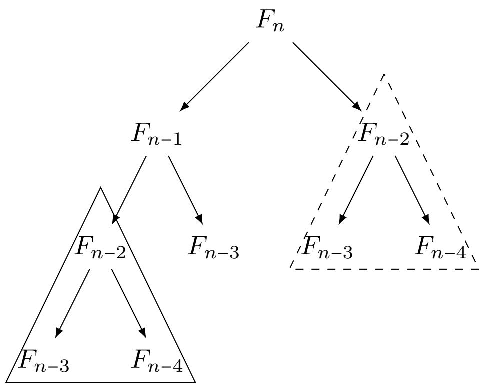


Figura 2 – Árvore de Fibonacci em P.D.

Para entender melhor o código escrito utilizando a técnica de programação dinâmica, observe a árvore recursiva na Figura 2 que calcula F_n, com $n = 5$ de forma que $F_{n-4} = F_{n-3} = 1$ são os casos base. Nesta árvore, o cálculo de F_{n-2} é realizado duas vezes como mostrado no triângulo com traço contínuo e no triângulo tracejado. A ideia da programação dinâmica é memorizar aquele valor para não desperdiçar processamento calculando novamente o seu valor. Com a técnica de memorização, o resultado do triângulo tracejado estará armazenado no `std::map` e seu valor será apenas retornado.

Código 4 – Fibonacci com Programação Dinâmica

```
std::map<int,int> memoization;

int fibonacci(int n)
{
    // Verifica se a_n já foi calculado
    auto it = memoization.find(n);
    if (it != memoization.end()) {
        return memoization.at(n);
    }

    // Declara e inicia a variável
    int fib_number = 0;

    if (n <= 2) {
        // Os dois primeiros termos são iguais a 1
        fib_number = 1;
    }
    else {
        // Cada número em seguida são a soma dos dois anteriores
        fib_number = fibonacci(n-1) + fibonacci(n-2);
    }

    // Armazena a_n para referências futuras
    memoization[n] = fib_number;
```

```
    return fib_number;
}
```

A única diferença entre o Código 4 para o Código 3 é o `std::map` utilizado para armazenar os valores já calculados. Dentro da função recursiva, antes dos casos base, é verificado com `auto it = memoization.find(n);` se aquele termo de *Fibonacci* já foi calculado. Se o valor se encontrar no mapa, apenas retorne-o. Caso contrário, calcule *Fibonacci* de forma recursiva normalmente, mas armaenando seu valor no mapa com `memoization[n] = fib_number;` antes de retornar a função.

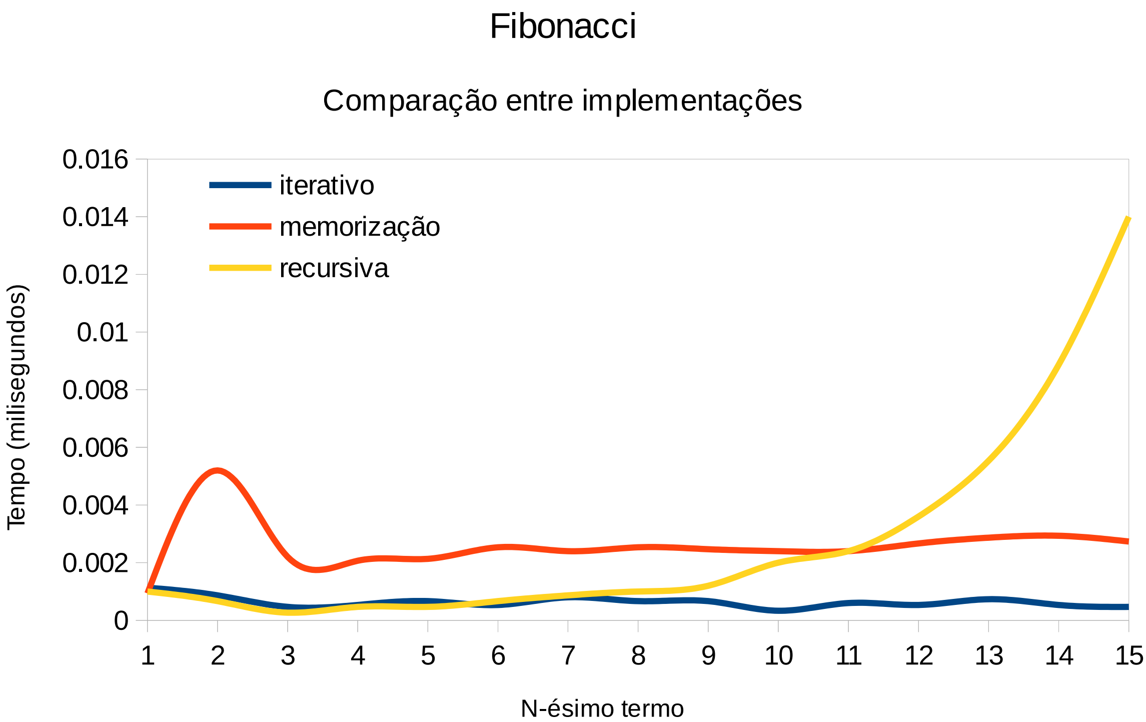


Figura 3 – Comparação entre implementações de *Fibonacci*

Na Figura 3 fica claro que a implementação recursiva do algoritmo cresce exponencialmente de acordo com o número de cálculos a ser realizado. Para tratar desse problema, a técnica de memorização armazena os valores da sequência de *Fibonacci* em um `map` e depois acessa seus valores ao invés de recalcular aquele n-ésimo termo. Isso faz com que o tempo do cálculo deixe de ser exponencial e passe a ficar mais parecido com o cálculo utilizando algoritmo iterativo.

1.4 *Big Points*

Big Points é um jogo abstrato e estratégico com uma mecânica de colecionar peças que pode ser jogado de dois a cinco jogadores. Foi criado no ano de 2008 e por um casal alemão chamado Brigitte Ditt e Wolfgang Ditt[8].

O jogo possui cinco peões de cores distintas, que podem ser usadas por qualquer jogador, para percorrer um caminho de discos coloridos até chegar à escada. Durante o percurso, os jogadores coletam alguns destes discos e sua pontuação final é determinada a partir da ordem de chegada dos peões ao pódio e a quantidade de discos adquiridos daquela cor. Ganha o jogador com a maior pontuação.

Como mostrado na Figura 4, o jogo é composto por cinco peões, um de cada uma das seguintes cores, denominadas **cores comuns**: vermelha, verde, azul, amarela e roxo. Para cada cor de peão, tem-se dez discos (totalizando cinquenta discos) denominados **discos comuns**, e cinco discos das cores branca e preta (totalizando dez discos) denominados **discos especiais**.

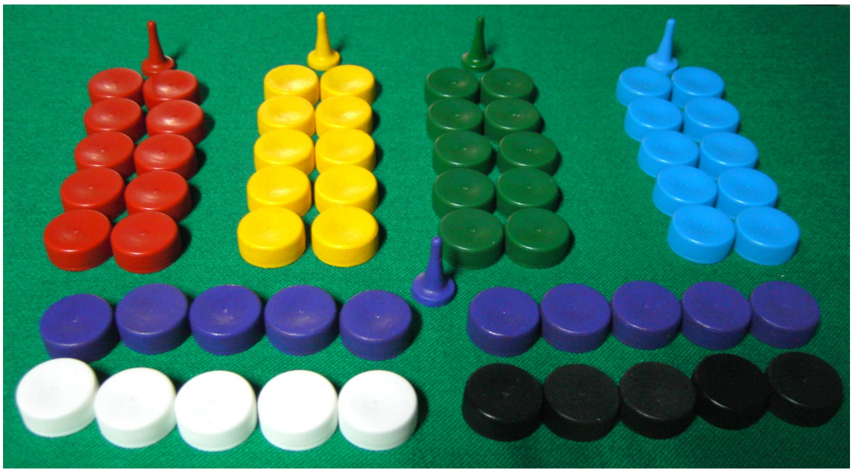


Figura 4 – Conteúdo do jogo *Big Points*

Por fim, há uma escada com um lugar para cada peão. A escada determinará a pontuação equivalente a cada disco da cor do peão, de maneira que o peão que ocupar o espaço mais alto no pódio (o primeiro a subir) fará sua cor valer quatro pontos, o segundo peão, três pontos e assim por diante, até o último valer zero ponto.

A preparação do jogo ocorre em algumas etapas, envolvendo a posição dos peões, a aleatoriedade do tabuleiro e alguns discos ao lado da escada. A primeira etapa é retirar um disco de cada cor comum e posicioná-los ao lado da escada: estes serão os discos coletados pelo jogador que levar o peão da sua cor para a escada. Com isso, restará nove discos de cada uma das cinco cores comuns mais cinco discos de cada uma das duas cores especiais resultando em $(n_{dc} - 1) \cdot n_{cc} + n_{de} \cdot n_{ce} = (10 - 1) \cdot 5 + 5 \cdot 2 = 55\ discos$, onde n_{dc} é o número de discos comuns, n_{cc} é o número de cores comuns, n_{de} é o número de discos especiais, e n_{ce} é o número de cores especiais. Em seguida, deve-se embaralhar todos os 55 discos restantes e formar uma fila até a escada: estes são os discos possíveis de serem coletados e onde os peões andam até chegar na escada. Por último, é preciso posicionar

[8] https://boardgamegeek.com/boardgame/34004/big-points

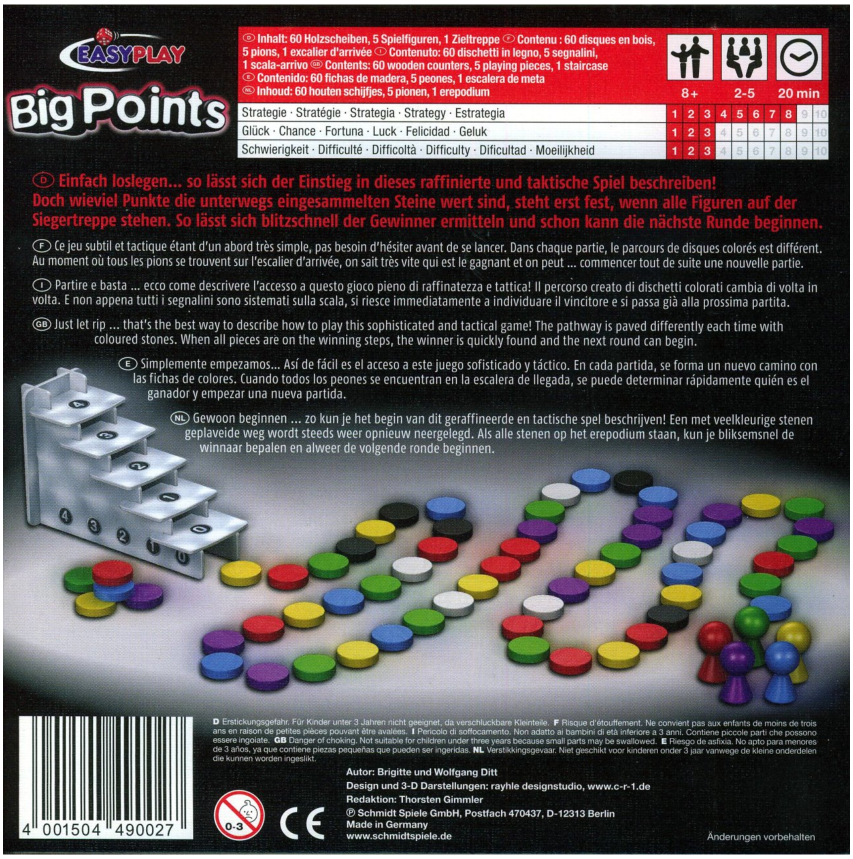


Figura 5 – Preparação do jogo *Big Points*

os peões no começo da fila de discos, de forma que fiquem opostos à escada. No final da preparação, o jogo assumirá uma forma semelhante à apresentada na Figura 5.

Após preparar o jogo, deve-se escolher o primeiro jogador de forma aleatória. Em sua vez, cada jogador escolhe um peão para movêlo até o próximo disco de sua cor. Levando em consideração a Figura 5, caso o jogador queira mover o peão roxo, os primeiros quatro discos serão ignorados e o peão colocado no quinto disco (o primeiro disco da cor roxa).

Em seguida, o jogador escolhe pegar um disco disponível[9] à frente ou atrás do peão. Ainda fazendo referência à Figura 5, após mover o peão roxo, o jogador pode pegar o disco verde à frente do local onde o peão roxo terminou seu movimento, ou o disco verde que se encontra atrás dele. Após pegar o disco, se o jogador começou seu turno com um disco preto na mão, ele poderá descartá-lo para realizar uma outra jogada, caso contrário, ele passa sua vez.

Ao realizar uma segunda jogada descartando o disco preto, o jogador pode escolher qualquer peão que possua um movimento válido. Dessa vez, o movimento do peão pode ser para trás. O jogador, então, coleta o disco normalmente, à frente ou atrás do peão movido.

Eventualmente, não terá mais disco à frente dos peões para realizar seus movimen-

9 É dito disponível aquele disco presente no tabuleiro, e que não possui um peão em cima.

tos. Neste caso, o peão deve ser movido para o espaço desocupado mais alto na escada. O jogador que realizar este movimento pega o disco da cor do peão que se encontra ao lado da escada.

Caso aconteça de um peão terminar seu movimento próximo a outro peão, é dito que aquele disco é indisponível para coletar. Então o jogador deve escolher próximo disco disponível, depois do peão vizinho. Dessa forma ele ainda possui duas escolhas: o disco de um lado sem peão; e, do outro lado, o próximo disco disponível após o peão vizinho.

O jogo segue desta maneira até que todos os peões se encontrem na escada. No final do jogo, conta-se os pontos e ganha o jogador que tiver a maior pontuação.

A pontuação do jogo é dependente da ordem de chegada dos peões na escada e da quantidade de discos de cada cor que o jogador tiver. O primeiro peão que chegou na escada faz com que cada disco de sua cor valha quatro pontos. Os jogadores devem então multiplicar a quantidade de discos daquela cor pelo valor da ordem de chegada do peão da sua cor na escada.

Exemplo: No final do jogo, o jogador tem três disco da cor vermelha, quatro discos da cor verde, quatro azuis, um amarelo, três roxos, um branco e um preto. A ordem de chegada do primeiro ao último peão são, respectivamente, vermelho, verde, azul, amarelo e roxo. Sua pontuação é demonstrada na Tabela 6.

Tabela 6 – Exemplo da Pontuação do Jogo *Big Points*

Cor do Disco	**#Discos**	**Valor**	**Total**
Vermelha	3	4	12
Verde	4	3	12
Azul	4	2	8
Amarela	1	1	1
Roxa	3	0	0
Branca	1	6	6
Preta	1	0	0
Pontuação			39

2 Metodologia

Após o entendimento dos conceitos de Teoria dos Jogos, Programação Dinâmica e das regras do jogo *Big Points*, serão explicados a metodologia seguida para a construção do projeto. A primeira seção explica como foram as reuniões com o orientador e a organização das tarefas. Na Seção 2.2, são feitas as análises da quantidade de jogos distintos e das jogadas para exaurir todas as possibilidades do jogo. Em seguida, na Seção 2.3, é explicado como os estados do jogo foram armazenados para ocupar o menor espaço possível. Por último, as Seções 2.3 e 2.4 que tratam, respectivamente, sobre o projeto da programação dinâmica e a verificação e validação do programa.

2.1 Fluxo de Trabalho

Inicialmente, foi utilizado o *kanban* do *waffle.io*[1] foi utilizado para registrar tarefas devido à sua integração com as *issues* do *GitHub*[2]. Reuniões com o orientador foram realizadas para discutir aspectos técnicos do jogo, como as estruturas de dados a serem utilizadas para reduzir os dados armazenados, e alguns métodos importantes para agilizar o processamento.

Porém, ao longo do tempo, o esforço para manter a rastreabilidade das tarefas tornou-se muito alto em relação à complexidade do projeto, e ao tamanho da equipe. As tarefas passaram a ser *branches* locais com nomes significativos, representando a funcionalidade a ser desenvolvida. Após a conclusão da tarefa, testes simples e manuais foram aplicados para então unir à *branch* mestre[3]. Por fim, para trabalhar em outra *branch*, sempre foi necessário atualizá-la em relação à mestre[4] para garantir a consistência do trabalho.

Outra ferramenta utilizada ao longo do desenvolvimento deste trabalho foi uma ferramenta de rastreamento de tempo chamada wakatime[5]. Esta ferramenta, além de monitorar o tempo gasto em determinado projeto, ela também registra o tempo gasto nos arquivos. No link do projeto no wakatime[6], o tempo registrado no trabalho foi de quase 140 horas em 17 semanas.

---

[1] https://waffle.io/mfurquim/tcc
[2] https://github.com/mfurquim/tcc
[3] `$ git checkout <to-branch>; git merge <from-branch>`
[4] `$ git rebase <from-branch> <to-branch>`
[5] https://wakatime.com/@mfurquim/
[6] https://wakatime.com/@mfurquim/projects/vchkyvvbrq?start=2017-03-06&end=2017-07-03

2.2 Análise do jogo *Big Points*

Para analisar o jogo *Big Points*, foram rastreadas todas as jogadas de todos os jogos possíveis. Em seguida, foi feita uma simulação onde cada jogador, na sua vez, escolheria uma jogada que lhe garantisse a vitória ou, se houver mais de uma possibilidade, escolhe a que resultasse em maior pontuação. Caso não existisse uma jogada que levasse à vitória, o jogador deveria minimizar a pontuação de seu adversário. Após fazer isso para um jogo escolhido, os resultados foram escritos em um arquivo *csv*[7] para análise. Esse procedimento foi repetido para *cada* combinação possível do tabuleiro inicial.

2.2.1 Quantidade de partidas

Para estudar a viabilidade de solucionar[8] o jogo, foi preciso calcular a quantidade de partidas distintas do jogo *Big Points*. A característica do jogo que muda de uma partida para outra são a quantidade de jogadores e o arranjo dos discos formando o tabuleiro. Para a quantidade J de jogadores, tem-se $J \in [2, 5]$. Agora, para a organização dos discos, faz-se uma combinação de cada cor, com a quantidade restante de discos.

$$\begin{aligned} P &= (J-1)\binom{d_t}{n_w}\binom{d_{l1}}{n_k}\binom{d_{l2}}{n_r}\binom{d_{l3}}{n_g}\binom{d_{l4}}{n_b}\binom{d_{l5}}{n_y}\binom{d_{l6}}{n_p} \\ P &= 4\binom{55}{5}\binom{50}{5}\binom{45}{9}\binom{36}{9}\binom{27}{9}\binom{18}{9}\binom{9}{9} \\ P &\approx 5 \times 10^{41} \end{aligned} \tag{1}$$

Na Equação 1, a quantidade de discos de uma determinada cor é indicado por n, então para a quantidade de discos da cor vermelha, verde, azul, amarela, roxa, branca, e preta são, respectivamente, n_r, n_g, n_b, n_y, n_p, n_w, e n_k. Para encurtar o cálculo, foi utilizado variáveis auxiliares para indicar a quantidade total de discos d_t e a quantidade restante dos discos após a combinação anterior (d_{l1}, d_{l2}, d_{l3}, d_{l4}, d_{l5} e d_{l6}).

O valor total d_t de peças é igual a $d_t = n_r + n_g + n_b + n_y + n_p + n_w + n_k$ que valem, como dito na Seção 1.4, $n_w = n_k = 5$ para as cores especiais, e $n_r = n_g = n_b = n_y = n_p = 9$ para as cores comuns. As outras variáveis restantes após as combinações são: $d_{l1} = d_t - d_w$, para a combinação dos discos totais com os discos brancos; $d_{l2} = d_{l1} - d_k$, para a combinação dos discos restantes da combinação passada com os discos pretos; e assim por diante. No final, tem-se que a quantidade de partidas distintas[9] é aproximadamente $P \approx 5 \times 10^{41}$

[7] O tipo arquivo *csv* (*comma separated value*) possui seu conteúdo separado por vírgula.
[8] Solucionar um jogo é percorrer todas as sua possibilidades de movimento e seus resultados.
[9] P = 560.483.776.167.774.018.942.304.261.616.685.408.000.000.

2.2.2 Quantidade de jogadas

O próximo passo é exaurir todas as possibilidades de jogadas. Porém, o trabalho computacional é imenso e cresce exponencialmente de acordo com o tamanho do jogo. Para um jogo pequeno, com apenas dois discos e duas cores comuns (sem especiais), as jogadas possíveis são: mover o peão vermelho e pegar o disco da direita, ou da esquerda; e mover o peão verde e pegar o disco da direita ou da esquerda. Um jogo deste tamanho termina, em média, no quarto turno, como será mostrado na Seção 2.4. Isso gera uma árvore onde cada nó possui um número médio f de quatro filhos (jogadas possíveis) e uma altura média h de quatro (número de turnos), totalizando uma quantidade de estados de aproximadamente $\sum_{n=0}^{h} f^n \approx 341$, com $f = 4$ e $h = 4$.

Seguindo esta linha de raciocínio, um jogo completo (com 55 discos e todas as cores disponíeveis) teriam as possibilidades de jogada: mover os peões vermelho, verde, azul, amarelo, ou roxo; pegar o disco da direita ou da esquerda; e utiliziar, ou não, o disco preto para jogar novamente. Com 5 peões, 2 opções para pegar os discos (esquerda ou direita) e a opção de usar ou não a peça preta, totaliza $5 \cdot 2 \cdot 2 = 20$ possibilidades de jogada. Como o jogo possui 55 discos, pode-se estimar que o jogo irá terminar no quinquagésimo quinto turno, totalizando $\sum_{n=0}^{h} f^n \approx 3 \times 10^{71}$ estados possíveis[10], com $h = 55$.

Para solucionar o jogo *Big Points* é preciso, então, calcular todas as jogadas de todas as partidas distintas, totalizando $T = P \cdot E \approx 10^{113}$ estados.

2.3 Representação e Codificação dos Estados

Para escrever a rotina de programação dinâmica capaz de otimizar o processamento recursivo, foi necessário identificar as variáveis do jogo que representam um **estado**. Um estado do jogo, como mostrado na Figura 6, depende dos discos do tabuleiro, dos peões que estão na escada, da mão dos jogadores, e do jogador atual (o jogador que fará a próxima jogada).

State
- estado do tabuleiro: bitset<num_discos>
- posição dos peões: Array<pos_peão>
- peões na escada: Array<int>
- discos dos jogadores: Array<Jogador>
- jogador atual: Bit

Figura 6 – Diagrama da estrutura `State`

[10] $\sum_{n=0}^{h} f^n \approx 379.250.494.936.462.821.052.631.578.947.368.421.052.631.578.947.368.421.052.631.578.947.368.421$ com $h = 55$.

Devido à enorme quantidade de estados do jogo *Big Points*, se fez necessário armazenar essas informações na menor quantidade de *bits*. Para isso foi proposto uma função para codificar, e outra para decodificar, cada estado em uma variável, como mostrado no Código 5, com o objetivo de reduzir o espaço ocupado na memória. Após implementar e testar nos limites da capacidade da maior variável disponível (`unsigned long long int`), percebeu-se um erro quando o cálculo utilizava quatro cores e cinco discos, o que levou a outra solução: a implementação dos estados por *bit fields*, implementado no capítulo seguinte.

Código 5 – Função de Codificação e Decodificação

```c
// Inicialização da classe
struct State state();

// Protótipo das funções
unsigned long long int codificacao(struct State);
struct State decodificacao(unsigned long long int);
```

2.4 Verificação dos Estados e Validação da Programação Dinâmica

Para garantir a implementação correta da PD, foram escritos em *post-its* os estados e as transições do menor jogo possível. Para a versão digital do trabalho, os mesmos estados foram desenhados no terminal (Figura 7).

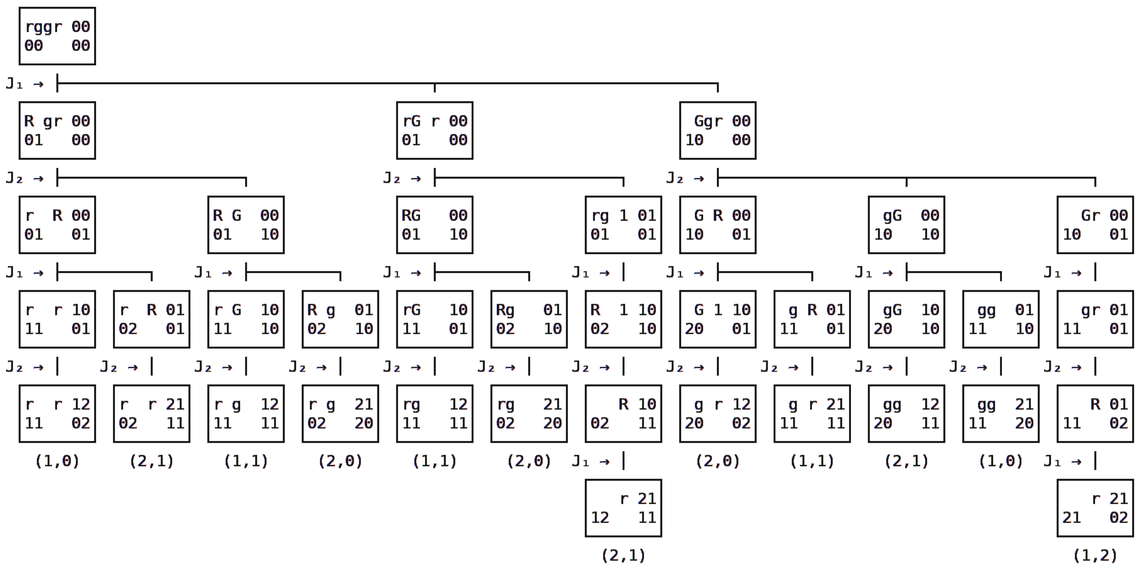


Figura 7 – Estados do menor jogo de *Big Poits* analisado

A estrutura da árvore foi escrita de cima para baixo, com cada nó representando um estado do jogo, e cada aresta representando uma transformação válida no estado do jogo (uma jogada válida). O nó raiz representa o estado inicial do jogo e o jogador a realizar a jogada está indicado ao lado da aresta. Para esta análise, foi escolhido um tabuleiro de forma arbitrário (as letras minúsculas), e a escada e a mão dos jogadores começam vazias. As letras minúsculas representam os discos da cor vermelha (`r`) e verde (`g`). De forma semelhante, as letras maiúsculas representam os peões das mesmas cores: vermelha (`R`) e verde (`G`). Logo em seguida, a um espaço de distância à direta do tabuleiro, encontra-se a escada, representada por dois números. Estes dois números representam a ordem de chegada dos peões. Seu primeiro valor é a ordem de chegada do peão `R` e, o segundo valor, a do peão `G`.

Os quatro números abaixo, dois à direita e dois à esquerda, representam os discos na mão dos jogadores J_1 e J_2, respectivamente. Seguindo a mesma lógica da escada, as mãos dos jogadores são representados por dois números, sendo o primeiro número a quantidade de discos vermelhos (`r`) e, o da direita, de discos verdes (`g`).

Foram escritos todas as possibilidades de estado e transições deste jogo reduzido para verificar os estados calculados e validar a implementação da PD. Acompanhando a primeira coluna, o primeiro jogador move o peão `R` pra cima do primeiro disco de sua cor (`r`), alterando o tabuleiro de `rggr` para `Rggr`. Após mover o peão, ele coleta o disco verde (`g`) à sua direita, incrementando assim sua mão de `00` para `01`, e modificando o tabuleiro de `Rggr` para `R gr`.

Em seguida, o segundo jogador também move o peão vermelho (`R`), e pega o disco `g` à esquerda (pois não há discos para se pegar à direita). O tabuleiro passa de `R gr` para `r  R`, e sua mão passa de `00` para `01`. J_1 faz mais um movimento, subindo o peão `R` para a escada, pois não há nenhum disco à sua frente. A escada passa de `00` para `10`, indicando que o peão vermelho foi o primeiro a chegar na escada. Aagora, cada disco `r` vale um ponto, enquanto os discos `g` valem zero, pois só há uma posição para o peão `G` ocupar na escada: a última posição.

O último movimento desta partida é feita pelo jogador J_2, que realiza a única jogada disponível: mover o peão `G` para a escada e coletar seu disco que fica ao lado da escada. Por fim, a pontuação é calculada de acordo com as Tabelas 7 e 8.

Tabela 7 – Pontuação P_1 de J_1

Cor do Disco	#Discos	Valor	Total
Vermelha	1	1	1
Verde	1	0	0
$\mathbf{P}_1$			1

Tabela 8 – Pontuação P_2 de J_2

Cor do Disco	#Discos	Valor	Total
Vermelha	0	1	0
Verde	2	0	0
$\mathbf{P}_2$			0

3 Resultados

Após identificar quais características representam um estado no jogo, e a melhor abordagem para escrever o código, seria feito um cálculo para saber quanto de memória foi necessário para armazenar um estado do jogo. Para a análise, o jogo foi reduzido para no máximo cinco cores e quatro discos, e todos os cálculos levaram em consideração essa redução. Neste capítulo, são relatados os passos necessários para realizar este cálculo, implementar o código do jogo utilizando a técnica de memorização da programação dinâmica, e os resultados do processamento deste programa.

3.1 Estrutura de Armazenamento

Foi decidido implementar a estrutura de armazenamento como uma `struct` ao invés de uma classe, devido à rapidez do acesso aos atributos e ao espaço reduzido ocupado na memória. A estrutura `State` possui cinco membros: `tabuleiro`, no qual pode armazenar informações sobre um tabuleiro de até 20 discos[1]; `peao`, que representa a posição do peão $p_i \in \{0, 1, ..., n_d, n_d + 1\}$, onde n_d é o número de discos de cores comuns[^cor_peao] no jogo e p_i é o peão da cor i; `escada`, que indica as posições dos peões na escada, sendo a *i-ésima* posição da `escada` é a posição do peao p_i; `jogadores`, que possui informações sobre os discos coletados dos dois jogadores; e, por fim, a variável `atual` que representa o jogador que fará a próxima jogada. Com isso em mente, foi realizado um cálculo para descobrir quantos *bits* serão necessários para armazenar cada uma destas informações.

3.1.1 Cálculo dos Membros da Estrutura `State`

Para os cálculos de um jogo reduzido de *Big Points* implementado neste trabalho, foram utilizados variáveis (e seus limites) de acordo com a Tabela 9.

Tabela 9 – Variáveis e seus limites utilizado no jogo reduzido de *Big Points*

Representação	**Variável**	**Limites**
#Cores	n_c	$[2, 5]$
#Peões	n_p	$n_p = n_c$
#Discos	n_d	$[2, 4]$
# Máx de Discos	d_t	$[4, 20]$

[1] Em um jogo reduzido de cinco cores de discos e quatro discos de cada cor, totaliza vinte discos no tabuleiro.

Como visto na Tabela 9, tais variáveis representam: n_c, para a quantidade de cores, com $n_c \in [2,5]$; n_p, para a quantidade de peões, com $n_c = n_p$; n_d, para a quantidade de discos, por cor, com $n_d \in [2,4]$; e d_t, para a quantiade máxima de discos, com $d_t \in [4,20]$.

Sabendo de antemão a ordem dos discos do tabuleiro, a única informação que é necessário guardar a respeito do tabuleiro é a disponibilidade do disco, ou seja, se os discos foram ou não pegos pelos jogadores. Desta forma, é preciso apenas um bit por disco no tabuleiro. A Equação 1 demonstra que só se faz necessário 20 *bits* para armazenar se um determinado disco está ou não disponível.

$$
\begin{aligned}
\texttt{tabuleiro} &= n_c \cdot n_d \\
\texttt{tabuleiro} &= 5 \cdot 4 \\
\texttt{tabuleiro} &= 20 \ \textit{bits}
\end{aligned}
\tag{1}
$$

A Equação 2 trata dos peões, o qual é necessário saber apenas em qual posição eles se encontram. Novamente com o conhecimento da configuração inicial do tabuleiro, é preciso saber apenas em qual disco de sua cor o peão se encontra, ao invés de saber a posição em relação a todos os discos do tabuleiro. A posição em relação aos discos totais no tabuleiro está no intervalo $[1, d_t]$ e a posição em relação aos discos de uma só cor está no intervalo $[1, n_c]$. Cada peão pode estar: fora do tabuleiro, com $\texttt{peao}(p_i) = 0$; em cima de um disco da sua cor, com $\texttt{peao}(p_i) \in \{1, 2, ..., n_d\}$; e na escada, com $\texttt{peao}(p_i) = n_d + 1$. Dessa forma, é preciso apenas $\log_2(1 + n_d + 1) = 3$ *bits* para cada peão, $3 \cdot n_p = 15$ *bits*.

$$
\begin{aligned}
\texttt{peao} &= \lceil \log_2(n_d + 1) \rceil \cdot n_c \\
\texttt{peao} &= \lceil \log_2(4 + 1) \rceil \cdot 5 \\
\texttt{peao} &= 3 \cdot 5 \\
\texttt{peao} &= 15 \ \textit{bits}
\end{aligned}
\tag{2}
$$

O cálculo de quantos *bits* a escada vai ocupar (Equação 3) é similar ao dos peões, mas ao invés de armazenar a quantidade dos discos de uma cor, a escada armazena a ordem de chegada de cada peão. Cada peão tem um local na escada, que armazena a sua posição de forma que $0 \leqslant \texttt{escada}(p_i) \leqslant n_c$. As situações possíveis são: $\texttt{escada}(p_i) = 0$, quando o peão não estiver na escada; e $\texttt{escada}(p_i) \in [1, n_c]$, sendo $\texttt{escada}(p_i)$ a ordem de chegada do peão na escada[2] indicando se foi o primeiro, segundo, terceiro, quarto ou

[2] O primeiro peão p_i a chegar na escada é indicado com $\texttt{escada}(p_i) = 1$.

quinto, com $n_c = 5$.

$$\begin{aligned} \texttt{escada} &= \lceil \log_2(n_p) \rceil \cdot n_p \\ \texttt{escada} &= \lceil \log_2(6) \rceil \cdot 5 \\ \texttt{escada} &= 15\ \textit{bits} \end{aligned} \tag{3}$$

O atributo `jogadores`, que corresponde aos discos na mão dos jogadores, é o que ocupa mais espaço na `struct`. A capacidade da variável `jogadores` é de 30 *bits*, como demonstrado na Equação 4. A informação armazenada na mão dos jogadores, para cada disco, vai até o número máximo de discos mais um ($\texttt{jogadores} \in [0, n_d{+}1]$), pois o jogador pode pegar todos os discos no tabuleiro além do disco adquirido ao mover o peão para a escada. Para armazenar a quantidade 5 de discos de uma determinada cor na mão de um jogador, são necessários $\lceil \log_2(5) \rceil = 3$ *bits*, que seria 101_2 em binário. Como são cinco cores e dois jogadores, faz-se necessário 30 *bits* para armazenar as informações deste atributo.

$$\begin{aligned} \texttt{jogadores} &= \lceil \log_2(n_d + 1) \rceil \cdot n_c \cdot n_j \\ \texttt{jogadores} &= \lceil \log_2(4 + 1) \rceil \cdot 5 \cdot 2 \\ \texttt{jogadores} &= 3 \cdot 5 \cdot 2 \\ \texttt{jogadores} &= 30\ \textit{bits} \end{aligned} \tag{4}$$

O cálculo mais simples é do atributo `atual`, apresentado na equação 5, que precisa indicar apenas se o próximo jogador a realizar o movimento é o J_1 ou o J_2.

$$\begin{aligned} \texttt{atual} &= \lceil \log_2(2) \rceil \\ \texttt{atual} &= 1\ \textit{bit} \end{aligned} \tag{5}$$

Somando os *bits* de todos os membros da `struct State`, tem-se a Equação 6.

$$\begin{aligned} \texttt{State} &= \texttt{tabuleiro} + \texttt{peao} + \texttt{escada} + \texttt{jogadores} + \texttt{atual} \\ \texttt{State} &= 20 + 15 + 15 + 30 + 1 \\ \texttt{State} &= 81\ \textit{bits} \end{aligned} \tag{6}$$

3.1.2 Implementação da Estrutura `State`

Para a implementação na linguagem C/C++, utilizou-se de uma `struct` com duas outras `struct`s anônimas dentro e *bit fields* nas variáveis.

Código 6 – Definição da estrutura `State`

```
struct State
{
    // Cinco cores, quatro discos
    struct {
        // 5 cores * 4 discos (1bit pra cada)
        ll _tabuleiro :20;

        // 0..5 posições possíveis (3bits) * 5 peões
        ll _peao :15;

        // 0..5 posições (3bits) * 5 peões
        ll _escada :15;
    };

    struct {
        // 0..5 discos (3bits) * 5 cores * 2 jogadores
        ll _jogadores :30;

        // Jogador 1 ou Jogador 2 (quem fará a próxima jogada)
        ll _atual :1;
    };
```

Dentro da estrutura `State` (Código 6) foram utilizados *bit fields*, que ocupa apenas parte da memória da variável, e variáveis do tipo `unsigned long long int`[3], que ocupa 64 *bits*. Após a declaração de um membro da estrutura, é declarado a quantidade de *bits* que será utilizado para ele, de modo que `ll _tabuleiro :20` ocupe apenas 20 *bits* da variável `unsigned long long int`, `ll _peao :15` ocupe 15 *bits*, e assim por diante de forma que não ultrapsse os 64 *bits* da variável.

Para armazenar a `struct State` em um `std::map` com o objetivo de memorizar os cálculos, foram criadas duas estruturas anônimas[4], como mostrado na Figura 8. Estas duas estruturas servem para garantir o alinhamento correto dos *bits*, desta forma, garantindo a consistência no armazenamento. Por exemplo: ao armazenar a `struct State` com todos os valores iguais a zero, menos o valor de `_atual`, e o *bit* de `_atual` for o último, espera-se que o valor armazenado seja 2^{61}_2, e não 1_2. A área colorida na figura não foi utilizada para pois não daria tempo de calcular um jogo grande o suficiente para ocupar todo o espaço da `struct`.

3.1.3 Funções de Acesso e Comparador da Estrutura `State`

A estrutura possui um construtor e algumas funções para acessar seus membros de forma rápida, implementados com operadores *bit wise* e máscara de *bits*.

[3] No código foi utilizado `using unsigned long long int = ll;` para facilitar a implementação.

[4] Estruturas anônimas permitem acesso às suas variáveis de forma direta, como por exemplo: `state._tabuleiro` acessa a variável `_tabuleiro` dentro da estrutura anônima, que por sua vez se encontra dentro da estrutura `State`.

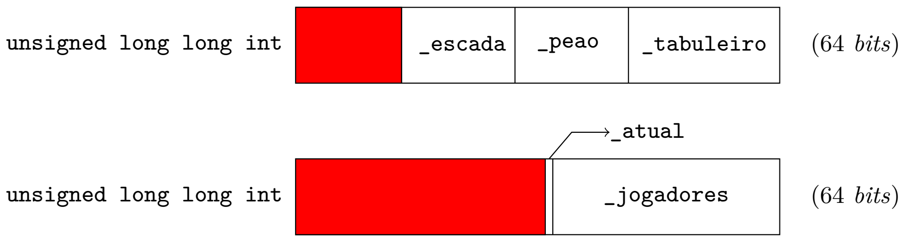


Figura 8 – Diagrama da `struct State`

A estrutura possui um construtor que atribui valores, e une o ciclo de vida das variáveis à própria estrutura, seguindo o princípio de RAII[5], dessa forma não se faz necessário nenhuma implementação extra. Todas as variáveis possuem um valor padrão, válido para qualquer tamanho de tabuleiro $n_d \in [4, 20]$.

Código 7 – Construtor da estrutura `State`

```
State(int mtabuleiro = (1<<20)-1, int mpeao = 0, int mescada = 0,
    int mjogadores = 0, int matual = 0) : _tabuleiro(mtabuleiro),
    _peao(mpeao), _escada(mescada), _jogadores(mjogadores),
    _atual(matual)
{
}
```

As funções de acesso ao atributo `_tabuleiro`, implementadas no Código 8, servem para tornar um disco no tabuleiro disponível ou indisponível. É passado um valor inteiro para ele que será utilizado como uma máscara binária, realizando um *shift* para acessar apenas aquele *bit* na variável, desta forma, sendo possível recuperar e alterar o seu valor.

Código 8 – Funções de acesso ao atributo `tabuleiro`

```
int tabuleiro (int pos) const {
    return (_tabuleiro & (1<<pos))>>pos;
}

void settabuleiro (int pos, int available) {
    _tabuleiro = (_tabuleiro & ~(1<<pos)) | ((available&1)<<pos);
}
```

O Código 9 mostra a implementação das funções para modificar o `_peao`. Como cada peão ocupa três *bits*, o acesso é realizado com o número 7 (ou 111_2 em binário). Esta máscara binária é movida até o índice da cor desejada, o operador *bit wise* `&` (*and*) é aplicado na variável para extrair apenas a informação daquele peão e, por fim, é realizado outro *shift* pra trazer os três bits para os três valores menos significativos. Para modificar o valor de `_peao`, é feito um processo semelhante ao de recuperá-lo, mas antes é preciso zerar os valores naquele índice para depois utilizar o operador *bit wise* `|` (*or*) para adicionar o novo valor.

5 *Resource Aquisition Is Initialization* é uma técnica de programação que vincula o ciclo de vida do recurso ao da estrutura (CORMEN et al., 2001).

Código 9 – Funções de acesso ao atributo `peão`

```
int peao (int cor) const {
    return (_peao & (7<<(3*cor)))>>(3*cor);
}

void setpeao (int cor, int pos) {
    _peao = (_peao&~(7<<(3*cor)))|((pos&7)<<(3*cor));
}

void movepeao (int cor) {
    setpeao(cor,peao(cor)+1);
}
```

Os processos, que foram utilizados no código do `_peao`, de dar *shift* na máscara binária, e acessar ou alterar o valor com os operadores *bit wise*, foram usados novamente no Código 10.

Código 10 – Funções de acesso ao atributo `escada`

```
int escada (int cor) const {
    return (_escada & (7<<(3*cor)))>>(3*cor);
}

void setescada (int cor, int pos) {
    _escada = (_escada&~(7<<(3*cor)))|((pos&7)<<(3*cor));
}
```

Para acessar os valores dos discos na mão dos jogadores (Código 11), foi utilizado um *shift* de tamanho `15` para selecionar apenas a mão de determinado jogador. Em seguida, o valor dos discos são acessados da mesma maneira que os peões no Código 9.

Código 11 – Funções de acesso ao atributo `jogador`

```
int jogador (int jogador, int cor) const {
    return ((_jogadores>>(15*jogador)) & (7<<(3*cor)))>>(3*cor);
}

void setjogador (int jogador, int cor, int qtd) {
    _jogadores = (_jogadores & ~(7<<(3*cor + 15*jogador) ))
        | ((qtd & 7) << (3*cor + 15*jogador));
}

void updatejogador (int player, int cor) {
    setjogador(player, cor, jogador(player, cor)+1);
}
```

Por fim, a atualização do jogador `_atual`, que basta inverter o *bit* com o operador ˆ (*xor*).

Código 12 – Funções de acesso ao atributo `atual`

```
int atual () const {
    return _atual;
}
```

```
    void updateatual () {
        _atual ^= 1;
    }
```

Para utilizar a estrutura em um mapa, foi preciso criar uma função de comparação para que o `std::map` saiba como comparar os estados e armazená-los internamente. No Código 13, os membros da estrutura `State` são apenas comparados em ordem arbitrária, caso eles sejam diferentes.

Código 13 – Comparado da estrutura `State`

```
    // Operator to use it in map
    bool operator<(const struct State& s) const {
        if (_tabuleiro != s._tabuleiro) return _tabuleiro < s._tabuleiro;
        if (_peao != s._peao) return _peao < s._peao;
        if (_escada != s._escada) return _escada < s._escada;
        if (_jogadores != s._jogadores) return _jogadores < s._jogadores;
        return _atual < s._atual;
    }
```

3.2 Implementação da Programação Dinâmica

Programação dinâmica é um método para a construção de algoritmos no qual há uma memorização de cada estado distinto para evitar recálculo (CORMEN et al., 2001). A memorização dos estados do jogo *Big Points* foi feita em um mapa, com a chave sendo o estado do jogo e com o valor armazenado sendo a pontuação máxima dos dois jogadores a partir daquele nó.

Para facilitar a implementação da programação dinâmica, o código foi separado em duas funções: a função `dp()` (Código 14), que é responsável pela chamada recursiva da programação dinâmica, e do retorno nos casos base e nos casos onde o valor já está memorizado no mapa; e a função `play()`, que realiza toda a lógica do jogo reduzido de *Big Points*.

A função `dp()` recebe como argumentos o mapa dos estados já memorizados anteriormente, uma cópia da estrutura que armazena informações relevantes a respeito do jogo, e a estrutura mais importante que é a `state`. Seu caso base é quando todos os peões se encontram na escada, que isso indica o final do jogo, e então é calculado a pontuação dos dois jogadores e retornado. A memorização ocorre no final da função `play()`, logo após a chamada da função `dp()`.

Em seguida, é verificado se o estado já foi calculado e se encontra no mapa. Caso contrário, o *loop* será executado n_c vezes e, para cada vez, chamará a função `play()` duas vezes, a primeira vez para jogar com o peão da cor `pawn` e pegar o disco da direita, e a segunda vez para jogar com o mesmo peão, mas pegando o disco da esquerda.

Os resultados das jogadas são: um valor *booleano*, verdadeiro se a jogada foi válida e falso se for inválida; e um par de inteiros com as pontuações do primeiro e do segundo jogador. Se a jogada for válida, a pontuação dos jogadores é colocada em um vetor que será ordenado posteriormente de acordo com o `state.atual()`.

Código 14 – Programação Dinâmica

```
ii dp(map<struct State, ii>& dp_states, struct Game game, struct State state)
{
    // If all pawns are in the stair
    if (is_pawns_stair(game, state)) {
        return calculate_score(game, state);
    }

    auto it = dp_states.find(state);
    if (it != dp_states.end()) {
        return dp_states[state];
    }

    vector<ii> results;
    for (short pawn = 0; pawn < game.num_cores; pawn++) {
        struct Turn right(state.atual(), pawn, true);
        struct Turn left(state.atual(), pawn, false);

        // DP após jogadas
        game_res result = play(dp_states, game, state, left);
        if (result.first) {
            results.push_back(result.second);
        }

        result = play(dp_states, game, state, right);
        if (result.first) {
            results.push_back(result.second);
        }
    }
```

A função `play` foi implementada com o objetivo de separar a lógica do jogo da lógica da programação dinâmica. No Código 15, foi escrito o começo da função com os parâmetros e a inicialização de algumas variáveis que são utilizadas pelo resto da função.

Código 15 – Função Play: Inicialização de Variáveis

```
game_res play(map<struct State, ii>& dp_states, struct Game game, struct State
    state, struct Turn turn)
{
    short player = turn.current_player;
    short pawn = turn.pawn_to_move;
    bool pick_right = turn.pick_right;
    short prev_pos = state.peao(pawn);
```

Caso o peão esteja na escada, como mostrado no Código 17, o jogador não pode realizar esse movimento e a função deve retornar imediatamente com o primeiro valor falso para indicar uma jogada inválida.

Código 16 – Função Play: Movimento Inválido

```
    // Cannot move pawn, it's already on the stair
    if (state.escada(pawn) != 0) {
        // cout << "Can't move. Pawn already on the stair" << endl;
        return game_res(false, ii(-1,-1));
    }
```

O Código 17 remove os discos indisponíveis da `string game.board` para que o jogador não possa mover com o peão pra cima daquele disco e nem coletá-lo.

Código 17 – Função Play: Remoção de Discos

```
    // Remove discs from the board according to tabuleiro
    for (size_t i = 0; i < game.board.size(); i++) {
        if (state.tabuleiro(i) == 0) {
            game.board[i] = '0';
        }
    }
```

Em seguida, no Código 18, se ainda tiver discos na frente do peão, ele é movido para o próximo disco de sua cor. Caso o disco não esteja disponível, a variável `bool available` será igual a falso e o *loop* vai continuar. Se não tiver nenhum disco disponível à sua frente, a *flag* `bool in_range` recebe o valor falso e o *loop* é quebrado.

Código 18 – Função Play: Movimento do Peão

```
    // Moving Pawn
    bool available = false;
    bool in_range = false;
    do {
        if (state.peao(pawn) <= game.num_discos) {
            state.movepeao(pawn);
        }
        in_range = state.peao(pawn) <= game.num_discos;
        if (in_range) {
            available = game.board[game.color_index[pawn][state.peao(pawn)-1]] != '0';
        }
    } while (!available && in_range);
```

Se não tiver nenhum disco à sua frente, o peão sobe a escada para a posição mais alta não ocupada (Código 19).

Código 19 – Função Play: Subir a Escada

```
    // Step in the stair
    if (!in_range) {
        if (state.escada(pawn) == 0) {
            state.setescada(pawn, max_in_escada(game, state)+1);
            state.updatejogador(player, pawn);
            state.updateatual();
        }
    }
```

Após o movimento do peão, é realizado duas atualizações no estado do jogo, escrito no Código 20. A primeira atualização é se o peão se moveu pra cima de um disco no tabuleiro, aquele disco agora se torna indisponível. E a segunda é, se o peão estava em cima do tabuleiro antes de se mover, aquele disco agora se torna disponível para coletar.

Código 20 – Função Play: Atualiza Tabuleiro

```
    // Update board: Discs under pawns are unavailable
    for (int color = 0; color < game.num_cores; color++) {
        // If pawn is in board
        if (state.peao(color) != 0 and state.peao(color) <= game.num_discos) {
            // Removing disc under pawns' current position
            game.board[game.color_index[color][state.peao(color)-1]] = '0';
        }
    }

    // If pawn is in board and was on the board before moving
    if (state.peao(pawn)-1 > 0 and prev_pos > 0) {
        // Replacing disc under pawn's previous position
        game.board[game.color_index[pawn][prev_pos-1]] = '1' + pawn;
    }
```

O próximo passo (Código 21) é coletar o disco de acordo com a variável `pick_right`. Se o seu valor for verdadeiro, então o *loop* vai procurando por discos disponíveis à direita do peão movido, caso contrário, irá procurar por discos disponíveis à esquerda. Se houver disco disponível pra pegar, a variável `pick` terá um valor verdadeiro, e entrará no bloco `if (pick)` para o jogador coletar o disco e pra retirá-lo disco do tabuleiro.

Código 21 – Função Play: Coleta Disco

```
    // Pick a disc if the pawn has moved within the range of the board
    bool pick = false;
    if (in_range) {
        short pawn_pos = state.peao(pawn)-1;
        short disc_pos = -1;

        for (short i = 1;; i++) {
            // Pick right
            if (pick_right) {
                disc_pos = game.color_index[pawn][pawn_pos]+i;

                // Does not pick right (out of board)
                if (disc_pos >= (short) game.board.size()) {
                    return game_res(false, ii(-1,-1));
                }
            }
            // Pick left
            else {
                disc_pos = game.color_index[pawn][pawn_pos]-i;

                // Does not pick left (out of board)
                if (disc_pos < 0) {
                    return game_res(false, ii(-1,-1));
                }
```

```
            }

            // Does not pick if disc is 0, try again
            if (game.board[disc_pos] == '0') {
                continue;
            }

            // There is a disc to be picked
            pick = true;
            break;
        }

        // If There is a disc to be picked
        if (pick) {
            char pick_char = -1;

            // Disc's char to pick
            pick_char = game.board[disc_pos];

            // Remove it from the board
            state.settabuleiro(disc_pos, 0);

            // Add it to the player's hand
            state.updatejogador(player, pick_char-'1');

            // Calculate next player
            state.updateatual();
        }
    }
```

Por fim, a função é retornada de acordo com o Código 22 com a melhor pontuação para o jogador `_atual` que é resultado da função `dp()`.

Código 22 – Função Play: Retorno da Função

```
    auto max_score = dp(dp_states, game, state);

    return game_res(true, max_score);
}
```

3.2.1 Implementação do *Minimax*

De acordo com o teorema *minimax*, o jogador `_atual` quer maximizar sua pontuação e minimizar a pontuação de seu oponente.

No Código 23, para maximizar sua pontuação, o primeiro jogador ordena suas possíveis jogadas baseada na ordem crescente das suas pontuações e em ordem decrescente da pontuação de seu adversário.

Código 23 – Implementação do *Minimax*

```
    auto p1_order = [](const ii& a, const ii& b){
        if (a.first > a.second) {
            if (b.first > b.second) {
```

```
            return a.first > b.first ? true : false;
        }
        else {
            return true;
        }
    }
    else if (a.first == a.second) {
        if (b.first > b.second) {
            return false;
        }
        else if (b.first == b.second) {
            return a.first > b.first ? true : false;
        }
        else {
            return true;
        }
    }
    else {
        if (b.first >= b.second) {
            return false;
        }
        else {
            return a.second < b.second ? true : false;
        }
    }
};
```

No Código 24, da mesma forma que o primeiro jogador, J_2 vai ordenar suas jogadas baseado na ordem crescente de suas pontuações e em ordem decrescente da pontuação de J_1.

Código 24 – Implementação do *Minimax*

```
auto p2_order = [](const ii& a, const ii& b){
    if (a.second > a.first) {
        if (b.second > b.first) {
            return a.second > b.second ? true : false;
        }
        else {
            return true;
        }
    }
    else if (a.second == a.first) {
        if (b.second > b.first) {
            return false;
        }
        else if (b.second == b.first) {
            return a.second > b.second ? true : false;
        }
        else {
            return true;
        }
    }
    else {
        if (b.second >= b.first) {
            return false;
```

```
            }
            else {
                return a.first < b.first ? true : false;
            }
        }
    };
```

No Código 25, o `if()` serve para ordenar o vetor de pontuações de acordo com qual dos dois jogadores vão jogar. Depois de ordenar o vetor, o valor da frente (o melhor para aquele jogador) é armazenado no mapa e retornado da função `dp()`.

Código 25 – Implementação do *Minimax*

```
    if (state.atual() == 0) {
        sort(results.begin(), results.end(), p1_order);
    }
    else {
        sort(results.begin(), results.end(), p2_order);
    }

    dp_states[state] = results.size() == 0 ? ii(-1, -1) : results.front();

    return dp_states[state];
}
```

3.3 Cálculo das Partidas Reduzidas

Para o menor jogo trabalhado neste projeto, foi escolhido a quantidade de cores $n_c = 2$ e a quantidade de discos $n_d = 2$. Com isso, tem-se os jogos j_i e, para cada jogo, foi executado o algorítmo implementado para descobrir a pontuação máxima de ambos os jogadores e a quantidade de estados distintos, como demonstrado na tabela 10.

Tabela 10 – Pontuação utilizando *minimax*

Jogo j_i	**J_1**	**J_2**	**#Estados**
1122	2	1	17
1212	2	0	25
1221	2	1	25
2112	2	1	25
2121	2	1	25
2211	2	0	17

Ao final do cálculo deste jogo reduzido, tem-se que o número de estados distintos varia entre 17 e 25, dependendo do estado inicial do tabuleiro. Em todos as possíveis combinações de tabuleiros iniciais, o primeiro jogador sempre ganha com dois pontos enquanto o segundo jogador consegue fazer no máximo um ponto. A partir desta informação, infere-se que o jogo completo pode ser desbalanceado, pois apenas o jogador J_1 vence.

Foi calculado todas as combinações de tabuleiro dos seguintes jogos reduzidos: duas cores e dois discos; duas cores e três discos; duas cores e quatro discos; três cores e dois discos; três cores e três discos; três cores e quatro discos; e quatro cores e dois discos. O resultado desse processamento foi compilado nas Figuras 9, 10 e 11. Cada uma destas figuras mostram, respectivamente, a porcentagem de vitórias de J_1, empate, e vitórias de J_2, nas configurações explicadas acima.

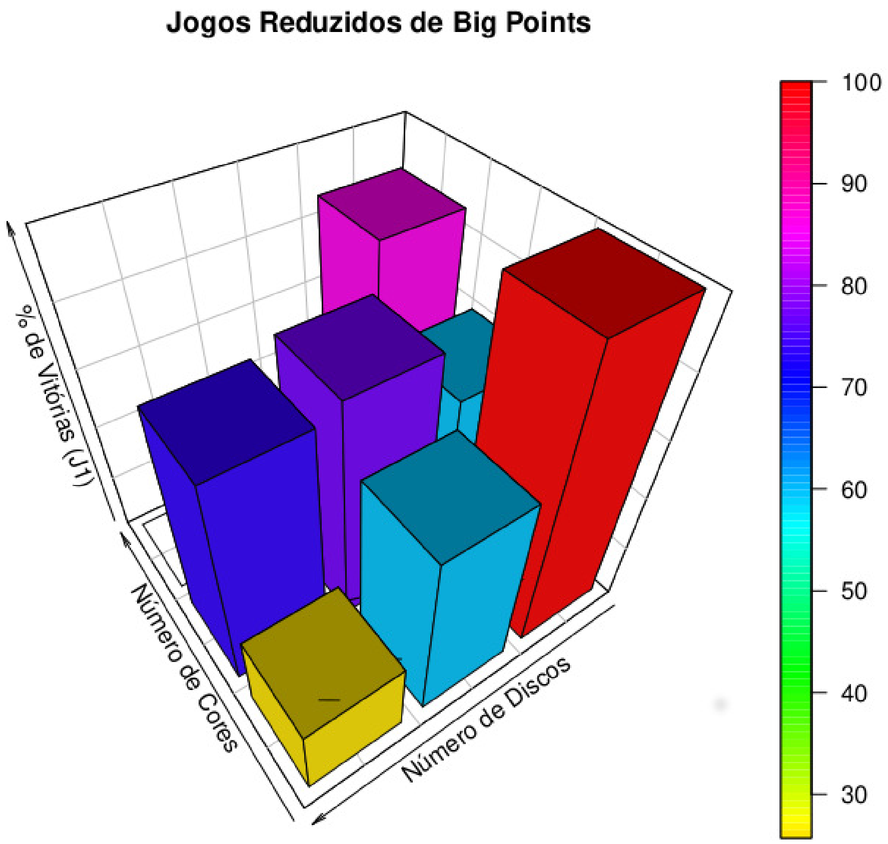


Figura 9 – Porcentagem de vitórias do Jogador 1 nos jogos reduzidos de *Big Points*

Na Figura 9, a barra mais alta (mais à direita), é a mesma configuração de jogo que a Tabela 10 e indica a vitória do Jogador 1 (100% das vezes). Este gráfico começa representando dois discos e duas cores. As barras mais à esquerda (aumentando no eixo *Número de Discos*) aumentam de um em um disco, até chegar na menor barra (mais à baixo do gráfico). Esta menor barra representa a porcentagem de vitórias do primeiro jogador em um jogo com duas cores e quatro discos.

As Figuras 10 e 11 seguem a mesma linha de raciocínio, sendo que a Figura 10 mostra a porcentagem de empates em todas as configurações de jogos reduzido, enquanto a Figura 11 demonstra a porcentagem de vitórias do jogador 2.

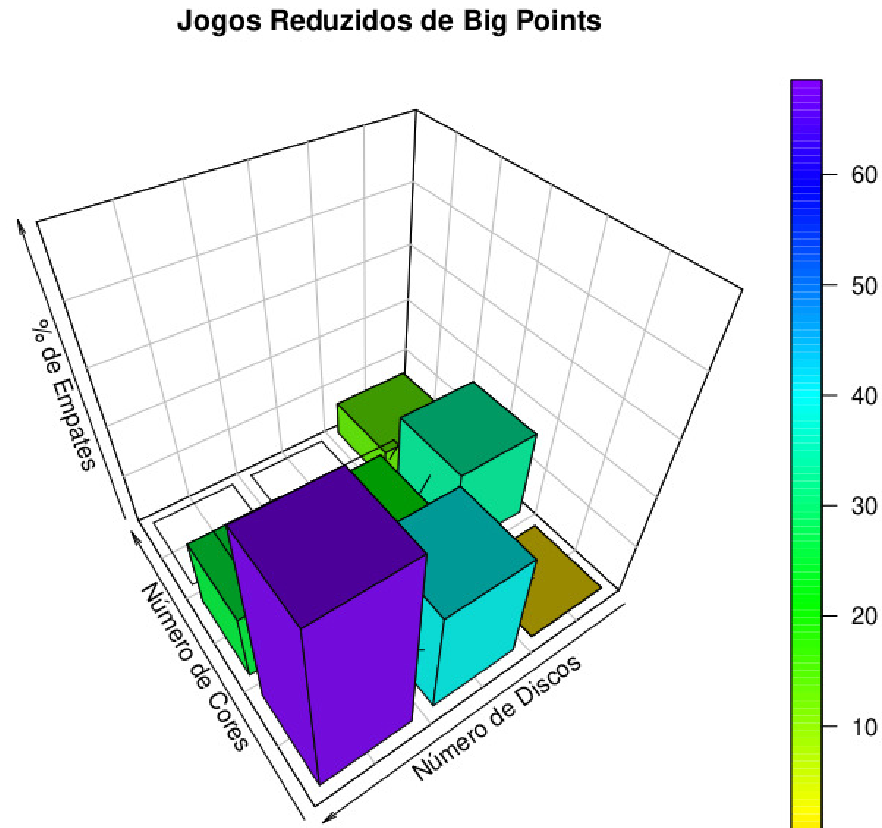


Figura 10 – Porcentagem de empates nos jogos reduzidos de *Big Points*

Observando a primeira fileira da Figura 9, é possível inferir que, aumentando a quantidade de discos, o jogo se torna mais balanceado, visto que a chance do jogador J_1 sempre ganhar continua diminuindo. No mesmo quadrante, na Figura 10, a barra é a maior, indicando a maior chance de empate em todas as configurações calculadas.

Comparando as Figuras 9 e 11, é fácil perceber quão desbalanceado é o jogo reduzido. Para a menor porcentagem de vitórias do jogador 1 tem-se, aproximadamente, 20%, enquanto que a maior porcentagem de vitórias do jogador 2 não chega nem a 10%.

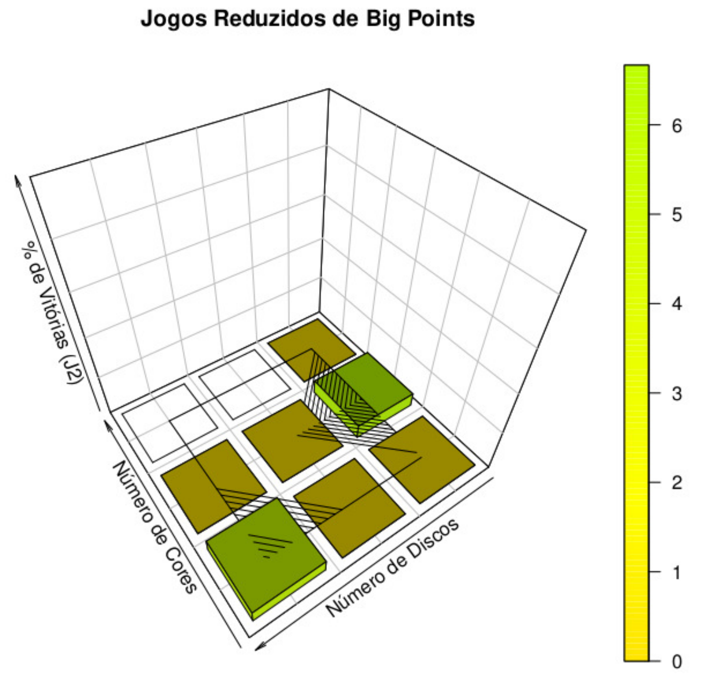


Figura 11 – Porcentagem de Vitórias do Jogador 2 nos jogos reduzidos de *Big Points*

4 Considerações Finais

A análise utilizada para solucionar o jogo neste trabalho foi o teorema *minimax*, onde cada jogador tenta aumentar sua pontuação e diminuir a pontuação do oponente. O programa escrito utilizando a técnica de memorização da programação dinâmica conseguiu rodar uma quantidade significativa de dados em um período de três semanas. Os resultados obtidos ao final da análise computacional baseadas neste teorema sugerem a possibilidade do jogo completo ser desbalanceado, dando ao primeiro jogador uma maior chance de vencer o jogo.

Este trabalho introduz uma análise da Teoria dos Jogos em um reduzido jogo de *Big Points*. Para dar continuidade à este trabalho são propostas os seguintes temas:

- Identificação de uma heurística para competir contra um jogador humano.
- Desenvolvimento de uma inteligência artificial
- Análise mais completa do jogo *Big Points*.

Devido à impossibilidade de um computador realizar todo o cálculo de um jogo inteiro de *Big Points*, uma boa alternativa seria identificar uma heurística boa para que o computador pudesse jogar em tempo hábil. Sem considerar que haja um peão na escada, dois indicadores que sugerem ser bons parâmetros são: a cor da qual possui mais discos; e a distância dessa cor para a escada. Considerando que haja peões na escada, o objetivo deve ser coletar a maior quantidade de discos da cor de peão que está mais ao topo da escada.

Para uma boa implementação deste um jogo (por ser multiplayer), o jogador deve ter a opção de jogar contra o computador. Isso pode ser alcançado fazendo uso de inteligência artificial para competir contra um jogador humano.

Em relação a realizar uma análise mais completa do jogo *Big Points*, pode-se pensar em processamento paralelo e distribuído. Dessa forma, seria possível distribuir a carga de processamento, a partir de certos estados da árvore, para diversos nós, ou clusters.

Referências

ADELSON-VELSKY, G. M.; ARLAZAROV, V. L.; DONSKOY, M. V. *Algorithms for Games*. New York, NY, USA: Springer-Verlag New York, Inc., 1988. ISBN 0-387-96629-3. Citado 2 vezes nas páginas 22 e 24.

ALMEIDA, A. N. de. Teoria dos jogos: As origens e os fundamentos da teoria dos jogos. UNIMESP - Centro Universitário Metropolitano de São Paulo, São Paulo, SP, Brasil, 2006. Citado 2 vezes nas páginas 19 e 20.

BERLEKAMP, E. R.; CONWAY, J. H.; GUY, R. K. *Winning Ways for Your Mathematical Plays, Vol. 1*. 1. ed. London, UK: Academic Press, 1982. Disponível em: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/1568811306>. Citado na página 20.

BOREL Émile. *The Theory of Play and Integral Equations with Skew Symmetric Kernels*. 1921. Citado na página 19.

BOREL Émile. *On Games that Involve Chance and the Skill of Players*. 1924. Citado na página 19.

BOREL Émile. *On Systems of Linear Forms of Skew Symmetric Determinant and the General Theory of Play*. 1927. Citado na página 19.

CARMICHAEL, F. *A Guide to Game Theory*. [S.l.: s.n.], 2005. Citado na página 20.

CORMEN, T. et al. *Introduction To Algorithms*. MIT Press, 2001. ISBN 9780262032933. Disponível em: <https://books.google.com.br/books?id=NLngYyWFl_YC>. Citado 3 vezes nas páginas 26, 43 e 45.

COURNOT, A.-A. Recherches sur les principes mathématiques de la théorie des richesses. L. Hachette (Paris), 1838. Disponível em: <http://catalogue.bnf.fr/ark:/12148/cb30280488q>. Citado na página 19.

GARCIA, D. D.; GINAT, D.; HENDERSON, P. Everything you always wanted to know about game theory: But were afraid to ask. *SIGCSE Bull.*, ACM, New York, NY, USA, v. 35, n. 1, p. 96–97, jan. 2003. ISSN 0097-8418. Disponível em: <http://doi.acm.org/10.1145/792548.611900>. Citado na página 20.

JONES, A. J. *Game Theory*: Mathematical models of conflict. [S.l.: s.n.], 1980. Citado 7 vezes nas páginas 9, 11, 21, 22, 23, 24 e 25.

MIYAZAWA, F. K. Introdução à teoria dos jogos algorítmica. UNICAMP, São Paulo, SP, Brasil, 2010. Disponível em: <http://www.ic.unicamp.br/~fkm/lectures/algorithmicgametheory.pdf>. Citado na página 19.

NASH, J. J. F. *The Bargaining Problem*. 1950. Disponível em: <https://www.econometricsociety.org/publications/econometrica/1950/04/01/bargaining-problem>. Citado na página 20.

NASH, J. J. F. *Non-Cooperative Games*. 1950. Disponível em: <http://rbsc.princeton.edu/sites/default/files/Non-Cooperative_Games_Nash.pdf>. Citado na página 19.

NASH, J. J. F. *Two-Person Cooperative Games*. 1953. Disponível em: <http://www.jstor.org/stable/1906951?seq=1#page_scan_tab_contents>. Citado na página 20.

NEUMANN, J. von. *Zur Theorie der Gesellschaftsspiele*. [S.l.]: Mathematische Annalen, 1928. 295–320 p. Citado na página 19.

NEUMANN, J. von; MORGENSTERN, O. *Theory of Games and Economic Behavior*. [S.l.]: Princeton University Press, 1944. Citado na página 19.

PRAGUE, M. H. *Several Milestones in the History of Game Theory*. VII. Österreichisches Symposion zur Geschichte der Mathematik, Wien, 2004. 49–56 p. Disponível em: <http://euler.fd.cvut.cz/predmety/game_theory/games_materials.html>. Citado na página 19.

ROSENTHAL, R. W. Some topics in two-person games (t. parthasarathy and t. e. s. raghavan). *SIAM Review*, v. 14, n. 2, p. 356–357, 1972. Disponível em: <https://doi.org/10.1137/1014044>. Citado na página 23.

SARTINI, B. A. et al. *Uma Introdução a Teoria dos Jogos*. 2004. Citado 2 vezes nas páginas 19 e 21.

SCHELLING, T. *The Strategy of Conflict*. Harvard University Press, 1960. Disponível em: <https://books.google.com.br/books?id=7RkL4Z8Yg5AC>. Citado na página 20.

ZERMELO, E. F. F. *Über eine Anwendung der Mengdenlehre auf die theories des Schachspiels*. 1913. 501–504 p. Citado na página 19.

Anexos

ANEXO A – Regras Originais do *Big Points*

Big Points

De Brigitte e Wolfgang Ditt
para 2 a 5 jogadores a partir dos 8 anos

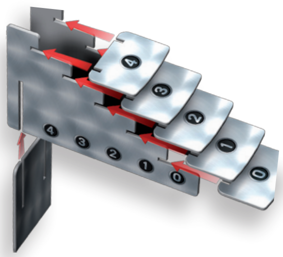
PORTUGUES

O material

- 60 discos em madeira (10 discos de cada umas das seguintes cores : azul, vermelho, amarelo, verde e violeta e ainda 5 brancos e 5 pretos)
- 5 peões : azul, vermelho, amarelo, verde e violeta
- 1 escada de chegada

Conceito do jogo

Os jogadores movem um peão qualquer para o próximo disco da mesma cor do peão. Depois, recolhem o disco situado à frente ou atrás desse peão. O valor dos discos recolhidos depende da ordem dos peões na escada de chegada no fim do jogo.



Antes do primeiro jogo, destacar cuidadosamente as peças do cartão e montar a escada de chegada como mostra a ilustração.

Os preparativos

Formar uma pilha com um disco de cada uma das cores seguintes : azul, vermelho, amarelo, verde e violeta e colocar essa pilha ao lado da escada. (Esses discos destinam-se aos jogadores que coloquem o seus peão na escada de che gada.) Misturar os discos res-tantes (e claro, os blancos e os pretos) e colocá-los como de-sejar de maneira a formar um percurso desde a base da es-cada. A ordem das cores não importa. Posicionar os peões no início do percurso (ver a ilustra ção à direita).

O desenvolvimento do jogo

Escolher um jogador inicial. Depois, joga-se à vez seguindo o sentido dos ponteiros do relógio. Na sua vez, o jogador escolhe um peão **qualquer**. Coloca-o sobre o disco seguinte cuja **cor** corresponda ao peão escolhido, em direcção à meta. Não é permitido mover um peão para trás.

Depois, o jogodor retira o dico do percurso. Ele pode escolher **entre o disco livre à frente** do peão que acabou de mover, ou **entre o disco primeiro livre atrás** do peão que acabou de mover. Os discos já ocupados não podem ser retirados do percurso. Cada jogador guarda os seus discos (escondidos) na palma da mão até ao final do jogo.

Exemplo:
O jogador move o peão azul para o disco azul seguinte. Em seguida, ele pode ficar com o disco verde que se encontra à frente do peão azul (ilustração de cima), ou com o disco preto que se encontra atrás do peão azul (ilustração de baixo).

- Nota: se, no início, não houver discos livres atrás do peão, o jogador **tem** de ficar com o disco livre seguinte **na direcção do movimento.** Esta regra também se aplica movermos um peão para um disco à frente da escada de chegada e não haja mais discos livres à frente desse peão; nesse caso, o jogador fica com o último disco livre que se encontre **atrás** do peão.

- Se não houver mais disco nenhum da cor correspondente ao peão, entre este e a escada de chegada, move-se o peão para a escada. O jogador coloca-o no degrau livre mais alto, de seguida pode retirar o disco da cor correspondente da pilha que se encontra ao lado da escada.

Os discos pretos

Se um jogador tirar um disco preto, pode utilizá-lo mais tarde para um turno suplementar:

- No momento em que o jogador decida utilizar um disco preto, ele pode - depois da sua vez - mover outro peão. Ele pode escolher o peão que acabou de mover ou outro peão. Depois, ele retira um disco - segundo as regras descritas anteriormente. Segue-se a vez do jogador seguinte.
- Durante o seu turno suplementar (e exclusivamente nesse), o jogador também pode mover um peão **para trás** colocando-o num disco da cor correspondente.

Não se pode usar mais que um disco preto no mesmo turno. Além disso, um disco preto **não pode usar-se no mesmo turno** em que foi conquistado. Ou seja, o jogador só pode usá-lo no turno seguinte à sua conquista.

Os discos pretos retiram-se do jogo depois de terem sido usados pelos jogadores e não voltam a ser utilizados.

Fim do jogo e pontuação

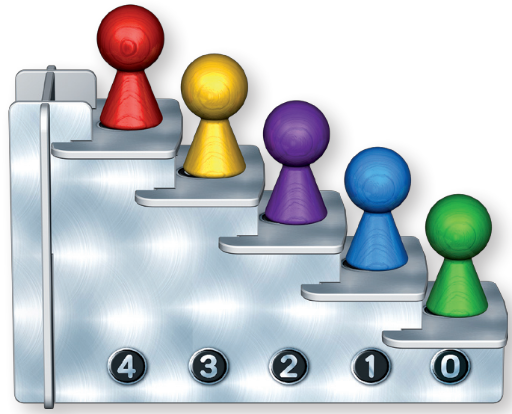
O jogo acaba quando o último peão é colocado na escada de chegada.

Em seguida, calculam-se os pontos:

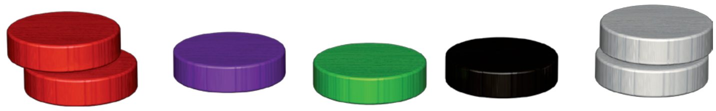
- Cada disco vale tantos pontos quantos os indicados no degrau da escada do peão da cor correspondente.
- Os discos pretos não valem nada.
- Cada disco branco vale tantos pontos quanto o número de discos de cores diferentes que o jogador possua.

Exemplo :

No fim do jogo, a escada terá um aspecto como o da ilustração do lado.



O jogador tem os seguintes discos:



A sua pontuação será:

2 x vermelhos (4 pontos cada um) = 8 points
1 x violeta (2 pontos cada um) = 2 pontos
1 x verde (0 pontos cada um) = 0 pontos
1 x preto (0 pontos cada um) = 0 pontos
2 x brancos (além do branco, o jogador possui 4 cores diferentes por isso recebe 4 pontos por cada um) = 8 pontos
No total: 18 pontos

O jogador que obtiver mais pontos ganh o jogo. Em caso de empate, há vários vencedores!

Várias partidas

Como os jogos não são muito longos, podem fazer-se várias partidas. Jogar tantas partidas como o número de jogadores. Em cada uma dessas partidas, começa um novo jogador. Adicionar os resultados das diferentes partidas. O jogador com mais pontos ganha. Em caso de empate, há vários vencedores!