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-

49 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 Macedo 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 um determinado jogo. Aplicando suas teorias em um jogo de tabuleiro eletrônico, este trabalho propõe analisar o jogo *Big Points* a partir de um determinado estado da partida e, como resultado, identificar as melhores heurísticas para os jogadores e uma possível inteligência artificial.

Palavras-chaves: Teoria dos Jogos, Análise Combinatória de Jogos.

Abstract

Key-words: Game Theory, Combinatorial Game Theory.

Lista de ilustrações

Lista de tabelas

Lista de Códigos Fonte

Lista de símbolos

Símbolos para conjuntos e operações matemáticas

$\varnothing$	O conjunto vazio
$\{\ \}$	Delimita conjunto, de forma que $S = \{\}$ é um conjunto vazio
$\forall$	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

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 $i \in \{1, \dots, a\}$
τ_j	Estratégias pura do jogador J_2, com $j \in \{1, \dots, b\}$
$P_n(\sigma_i, \tau_j)$	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. No final, suas ações como um todo levará a uma certa situação chamada resultado. Os membros deste grupo são chamados de jogadores e as regras que eles concordaram em obedecer constitui um jogo. Estes conceitos são exemplos das ideias utilizadas em análises baseadas na teoria dos jogos.

Objetivos

O objetivo principal deste trabalho é realizar uma análise *minimax* nas versões reduzidas do jogo *Big Points*. O jogo foi reduzido em relação à quantidade e tipo de algumas peças, visto que para o jogo completo seria 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? Para responder esta pergunta, primeiro foi necessário fazer uma análise de viabilidade do cálculo computacional de todos os jogos. Após chegar à conclusão que seria impossível calcular um jogo inteiro, foi escrito um programa para percorrer todas as possibilidades de jogadas de um jogo reduzido de *Big Points* e, por fim, interpretar os resultados para responder se o jogo (reduzido) é ou não balanceado.

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 para esta 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 abstrada sobre jogo estratégico e estratégia mista.

Em 1928, John von Neumann provou o teorema *minimax*, no qual 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, o que leva à mesma situação, as escolhas de cada jogador é escondida de seu oponente. Em qualquer um dos dois casos, o jogador deve escolher sua jogada levando em consideração a possível jogada do outro (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** são dois ou mais seres racionais que possuem um mesmo objetivo e, para alcançar esse objetivo, cada jogador possui um conjunto 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 jogador tem uma **função utilidade**, que atribui um ***payoff***, ou **ganho**, para cada situação do jogo. Quando essa informação 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.

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.

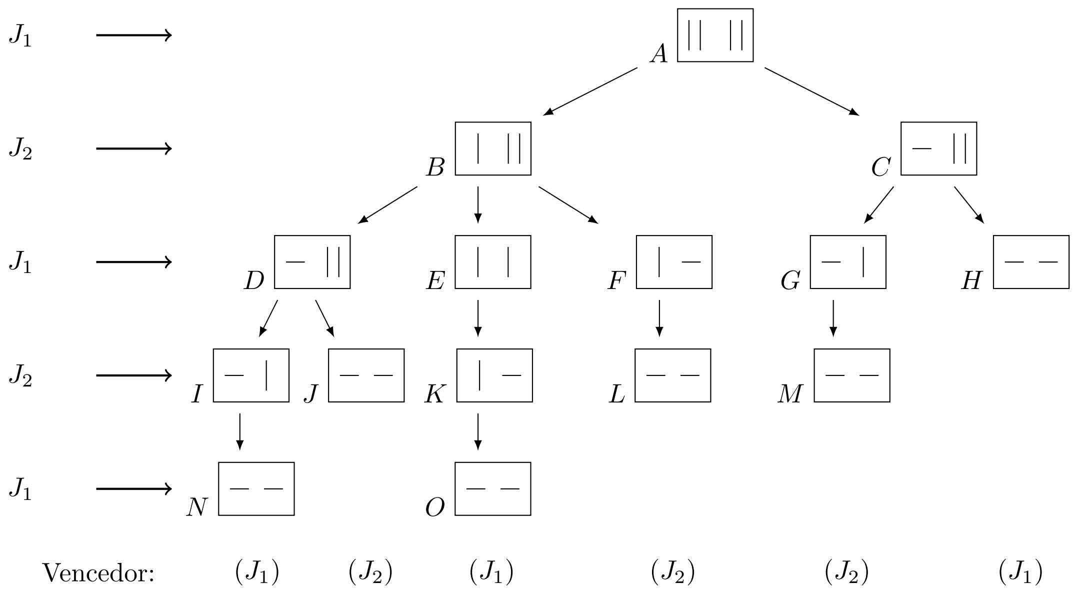


Figura 1 – Árvore do jogo *Nim*

Cada aresta representa uma jogada válida para o jogador que vai realizar o moviment (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. As últimas jogadas foram omitidas da árvore do jogo por serem simétricas às outras duas primeiras. 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 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,

[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.

[5] Movimento que garante a vitória.

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 termina em uma folha. 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 pura do jogador J_1 para o jogo *Nim*

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	$A \to C$	–	–

Tabela 2 – Estratégias pura do jogador J_2 para o jogo *Nim*

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

[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).

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 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.

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 es-

Tabela 3 – Forma Normal para o jogo *Nim*

		$\mathbf{J_2}$					
		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

tratégia 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*

		$\mathbf{J_2}$					
		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

Dessa forma, pode-se ver na Tabela 4 que a estratégia τ_4 sempre garante a vitória para J_2 independente da estratégia do jogador J_1.

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*. No Código 1, 2, 3 e 4 está escrito um programa simples para resolver este problema.

Take a problem, split in subproblems, solve the subproblems and reuse

Dependo da implementação do problema, o tempo de processamento para chegar no resultado desejado pode crescer exponencialmente. Nos Códigos 1, 2, 3 e 4

Código Fonte 1 – Funcao main de Fibonacci

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

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

```

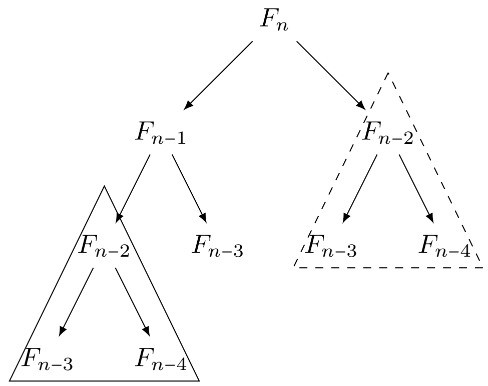


Figura 2 – Árvore de Fibonacci em P.D.

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

    return 0;
}
```

Código Fonte 2 – Fibonacci Iterativo

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

    // A sequência de fibonacci começa em: 1 e 1
    int a_0 = 1;
    int a_1 = 1;
    for (int i = 1; n > n; n++) {
        // a_n é igual a soma dos dois termos anteriores
        fib_number = a_0 + a_1;

        // Atualiza os termos
        a_0 = a_1;
        a_1 = fib_number;
    }

    return fib_number;
}
```

Código Fonte 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;

}
```

Código Fonte 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;
}
```

Os valores da sequência de *Fibonacci* foram conferidos no site da enciclopédia online das sequências de números inteiros[7].

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

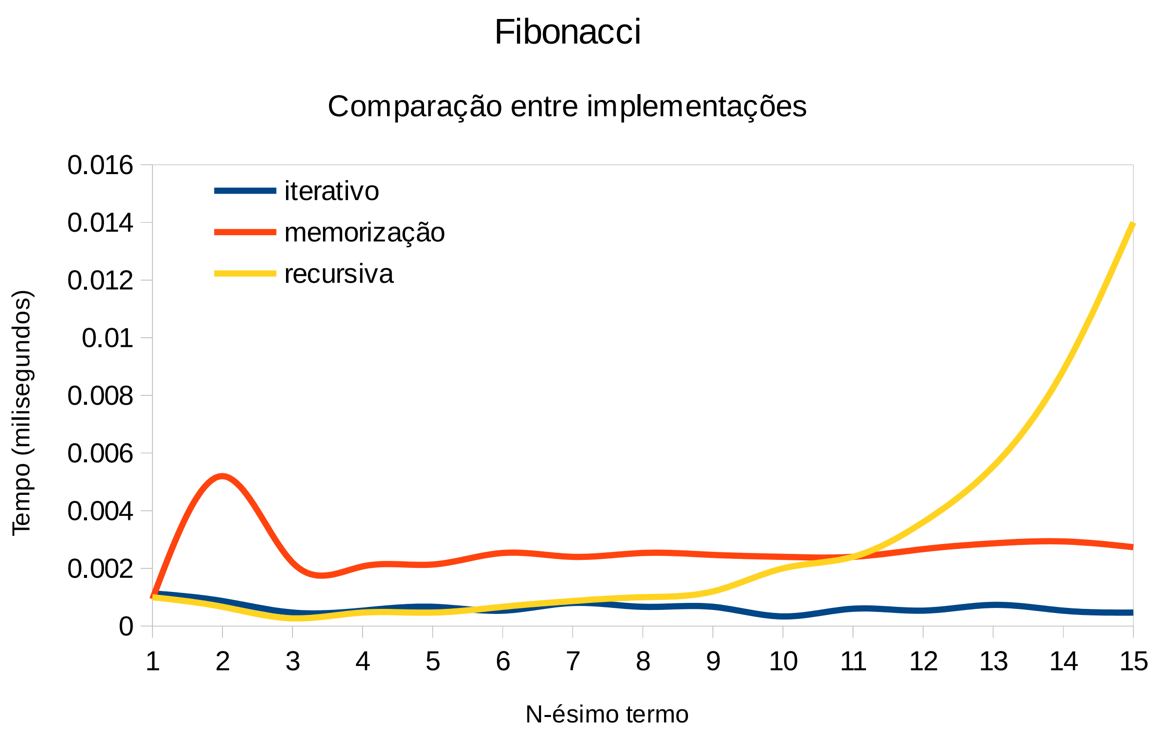


Figura 3 – Comparação entre implementações de *Fibonacci*

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 se torne

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. São 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.

O jogo é composto por cinco peões, como demonstrado na Figura, 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, como mostrado na Figura 8a, (totalizando cinquenta discos) denominados **discos comuns**, e cinco discos das cores branca e preta (totalizando dez discos) denominados **discos especiais**.

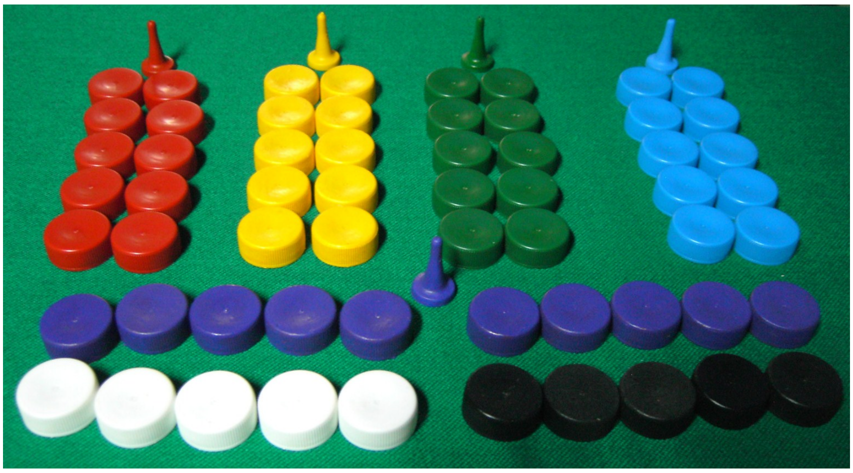


Figura 4 – Conteúdo do jogo *Big Points*

Por fim, há um pódio (ou 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. No caso de um jogo com menos de cinco peões, a seguinte fórmula se aplica: $S = N_c - P_{pos}$, onde S é a pontuação daquela determinada cor, N_c é o número de discos comuns e P_{pos} é a posição do peão no pódio.

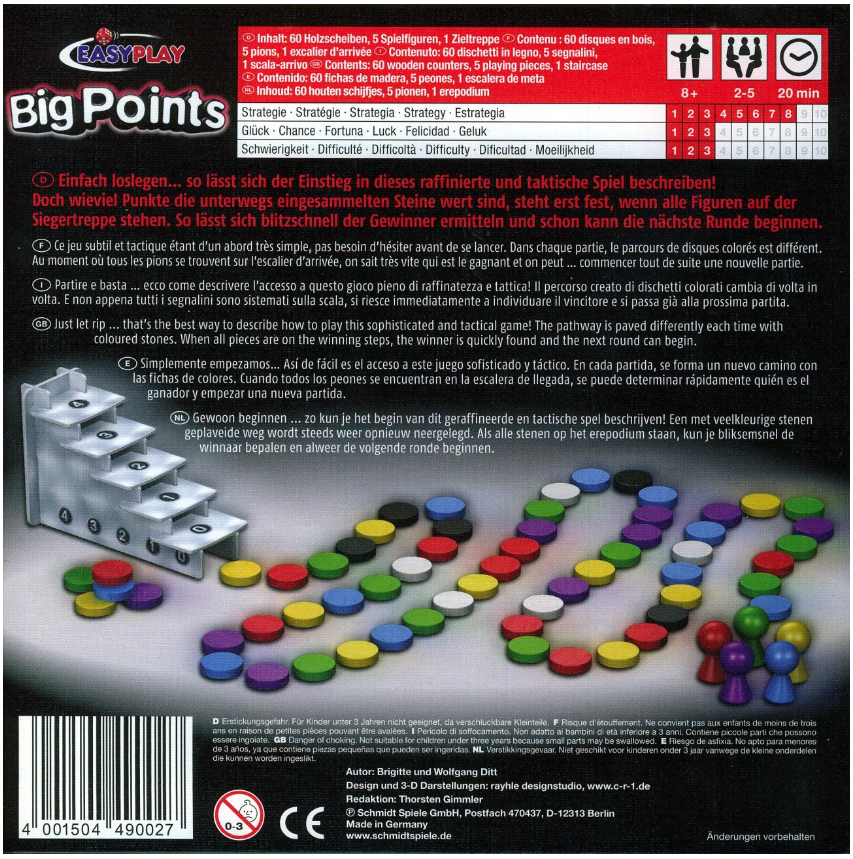


Figura 5 – Preparação do jogo *Big Points*

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 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 8b.

Após preparar o jogo, deve-se escolher o primeiro jogador de forma aleatória. Em sua vez, cada jogador deve escolher um peão, que não esteja na escada, para movê-lo até o disco à frente mais próximo de sua cor. Caso não haja um disco de sua cor para movê-lo, o peão sobe na escada para a posição mais alta que não esteja ocupada e coleta o disco daquela cor que está ao lado da escada. Em seguida, o jogador escolhe pegar o primeiro disco disponível[8] à frente ou atrás da nova posição do peão. Caso o disco não esteja disponível, verifique o próximo disco até encontrar um que esteja disponível. Ao encontrar um disco possa pegar, o jogador o retira do tabuleiro e o coloca em sua mão do jogador atual. A sua vez termina e passa para o próximo escolher um peão e pegar um disco. 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: um jogador tem um disco da cor vermelha (n_r) , zero discos da cor verde (n_g), dois azuis (n_b), cinco amarelos (n_y), quatro roxos (n_p), dois brancos (n_w) e um preto (n_k). A ordem de chegada dos peões são, respectivamente, vermelho (p_r), verde (p_g), azul (p_b), amarelo (p_y) e roxo (p_p). Sua pontuação S será descrita de acordo com a Equação

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

1.1, onde n_c é o número de cores distintas, com exceção da cor branca.

$$
\begin{aligned}
S &= n_r \cdot p_r + n_g \cdot p_g + n_b \cdot p_b + n_y \cdot p_y + n_p \cdot p_p + n_w \cdot n_c \\
S &= 1 \cdot 4 + 0 \cdot 3 + 2 \cdot 2 + 5 \cdot 1 + 4 \cdot 0 + 2 \cdot 5 \\
S &= 23
\end{aligned}
\tag{1.1}
$$

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, a Seção **??** trata sobre a implementação da programação dinâmica e a verificação e validação do programa.

2.1 Fluxo de Trabalho

O *framework Scrum* é ideal para o desenvolvimento de projetos complexos no qual a produtividade e a criatividade são essenciais para a entrega de um produto de alto valor (SCHWABER; SUTHERLAND, 2016). Inicialmente, tal método de organização e gerenciamento do projeto foi aplicado para o desenvolvimento do sistema em questão. 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.

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

[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>`

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*[5] 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 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 P 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.

$$\begin{aligned}
P &= (J-1)\binom{d_t}{n_w}\binom{d_{l1}}{n_k}\binom{d_{l2}}{n_r}\binom{d_{l3}}{n_g}\binom{d_{l4}}{n_b}\binom{d_{l5}}{n_y}\binom{d_{l6}}{n_p} \\
P &= 4\binom{55}{5}\binom{50}{5}\binom{45}{9}\binom{36}{9}\binom{27}{9}\binom{18}{9}\binom{9}{9} \\
P &= 560.483.776.167.774.018.942.304.261.616.685.408.000.000 \\
P &\approx 5 \times 10^{41}
\end{aligned} \tag{2.1}$$

Na Equação 2.1, 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.

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 Sub-Seção 2.4. Isso gera uma árvore

[5] O tipo arquivo *csv* (*comma separated value*) possui seu conteúdo separado por vírgula.

onde cada nó possui quatro filhos (jogadas possíveis) e altura média de quatro (número de turnos), totalizando uma quantidade de estados de aproximadamente $\sum_{h=0}^{4} f^h \approx 341$, com $f = 4$.

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 $\sum_{h=0}^{55} f^h \approx 3 \times 10^{71}$ estados possíveis[6].

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).

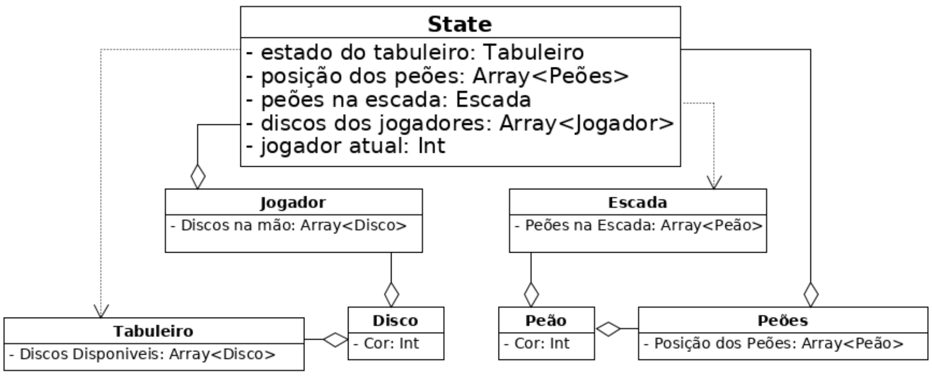


Figura 6 – Diagrama UML da Classe *State*

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, uma a classe 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.

[6] $\sum_{h=0}^{55} f^h \approx 3792504949364628210526315789473684210526315789473684210526315789473684 21$.

Código Fonte 5 – Função de Codificação e Decodificação

```
// Inicialização da classe
State state = new State();

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

2.4 Verificação dos estados

Para garantir a implementação correta da PD, foram escritos em *post-its* os estados, e suas transições, do menor jogo possível, como mostrado na Figura **??**.

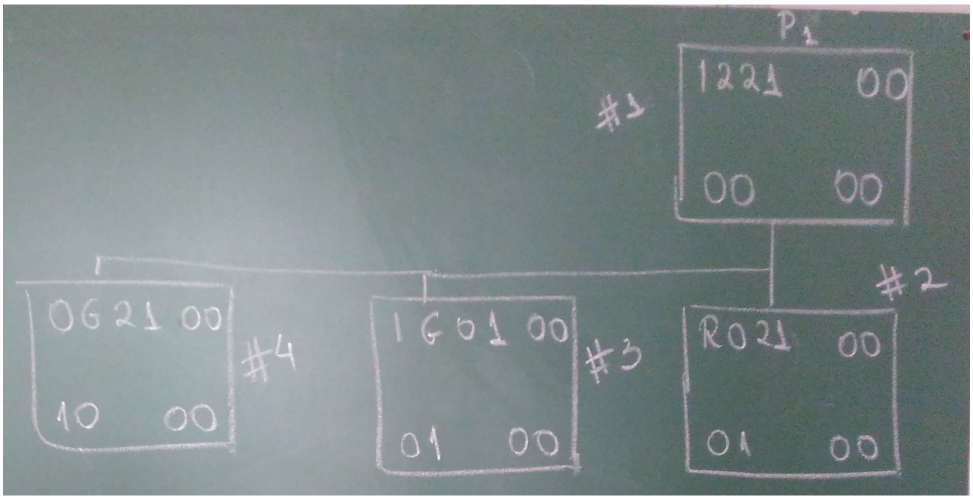
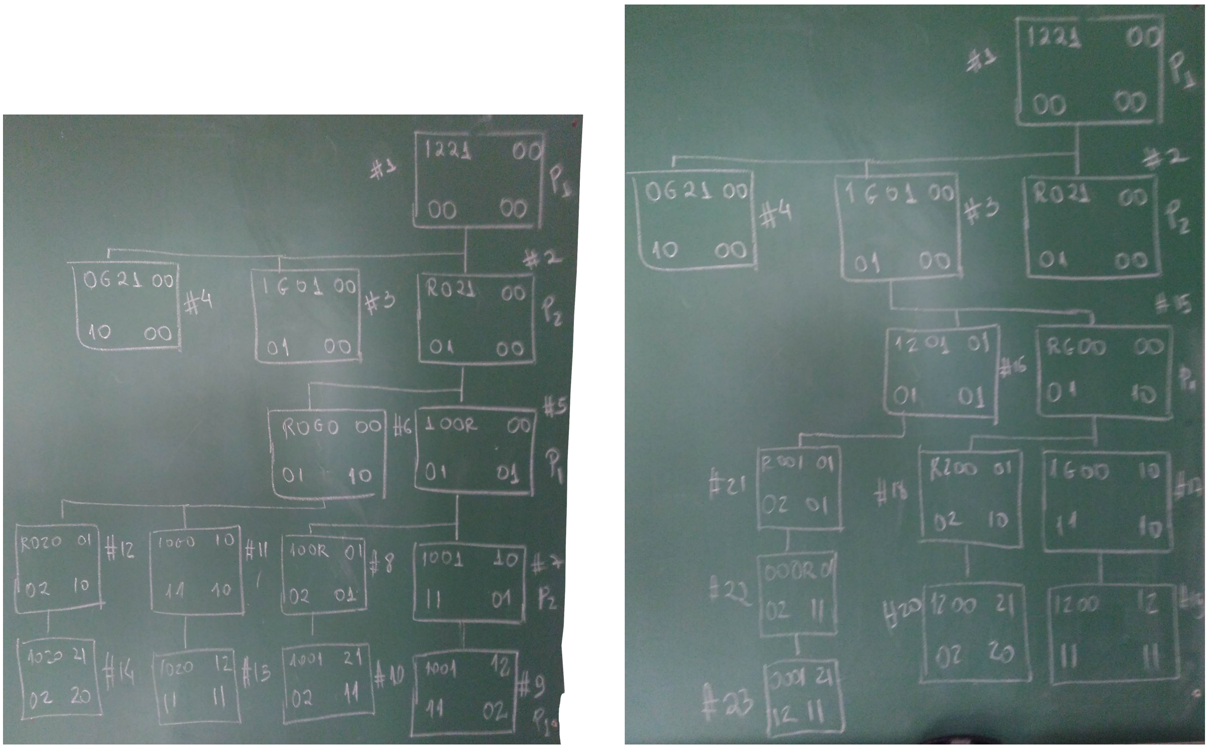


Figura 7 – Resultados ordenado por número de cores



(a) Conteúdo do jogo **Big Points**

(b) Preparação do jogo **Big Points**

Figura 8 – Organização do jogo **Big Points**

3 Resultados

Após identificar quais características representam um estado no jogo, e qual a melhor abordagem para escrever o código, foi feito um cálculo para saber quanto de memória foi necessário para armazenar um estado do jogo. Neste capítulo, são relatados os passos necessários para implementar a PD do jogo e seus resultados.

3.1 Implementação da Estrutura de Armazenamento

Dentro da estrutura `State` foram declaradas duas estruturas anônimas[1] utilizando *bit fields*. As duas estruturas servem para garantir a utilização correta dos *bits* quando as variáveis chegarem próximo ao limite da sua capacidade. Essas estruturas possuem variáveis do tipo `unsigned long long int`, 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. Como o comportamento do armazenamento é indeterminado quando a variável é ultrapassada, e para garantir consistência no armazenamento, foram utilizadas duas *structs* com, tamanho máximo igual uma variável `unsigned long long int` (64 *bits*).

A estrutura `State` possui cinco membros: `_tabuleiro`, no qual pode armazenar informações sobre um tabuleiro até 20 discos[2]; `_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 no jogo e p_i é o peão da cor i[3]; `_escada`, que indica as posições dos peões na escada, sendo a p_i-*ésima* posição de `_escada` é a posição do peao p_i; `_jogadores`, possui informações sobre os discos coletados dos dois jogadores; e por fim, a variável `_atual` que representa o jogador que fará a jogada. Esta estrutura está apresentada no Código 9.

3.2 Projeto 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, caso este estado apareça

1 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`.

2 Cinco cores e quatro discos.

3 As cores de peão seguem a ordem RGBYP começando do 0, onde **R***ed* = 0, **G***reen* = 1, **B***lue* = 2, **Y***ellow* = 3, e **P***urple* = 4.

novamente. A memorização dos estados do jogo *Big Points* foi feita em uma *hash*, com a chave sendo o estado do jogo e o valor armazenado, a pontuação máxima dos dois jogadores a partir daquele nó.

a melhor jogada para ganhar maximizar seus pontos. Caso não Na vez de cada Caso a quantidade de jogos vencidos pelo primeiro jogador seja aproximadamente 50%

Para analizar o jogo, é preciso exaurir todas as jogadas possíveis a partir de um jogo inicial. Como

utilizando programação dinâmica[^dynamic_programing] onde os estados são armazenados em uma *hash*, tem-se que o número de estados distintos varia entre 17 e 25.

Devido ao imenso número de jogadas possíveis ao longo do do jogo, decidiu-se utilizar a programação dinâmica para - Duas funções para melhor entendimento da DP e regras do jogo

A função `dp` possui os casos base para retornar a função,

Código Fonte 6 – 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.

Código Fonte 7 – Função Play

```
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);

    // 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));
    }

    // 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';
        }
    }

    // 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);

    // 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();
```

```
        }
    }

    // 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;
    }

    // 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();
        }
    }

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

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

- Explicação da DP e da função Play (função para realizar as jogadas)

3.3 Implementação do Minimax

Código Fonte 8 – 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;
            }
        }
    };

    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;
```

```
            }
        }
    };


    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];
```

long long int, de forma que cada estado ocupasse apenas 64 *bits* na memória.

Na figura 9

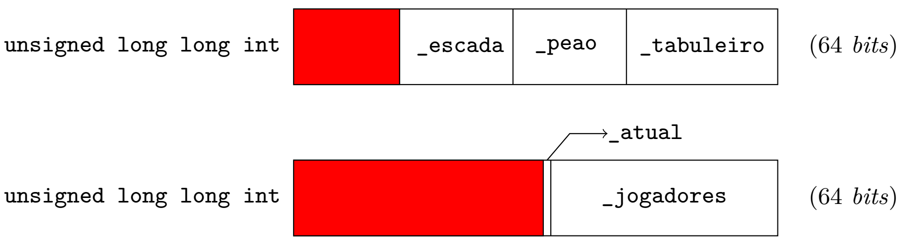


Figura 9 – Diagrama da `struct State`

Código Fonte 9 – 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;
    };
```

O cálculo para determinar os *bits* necessários para armazenar as informações de cada variável foi realizado será explicado a seguir.

O cálculo de bits do atributo `tabuleiro` é apresentado na equação 3.1.

$$
\begin{aligned}
\texttt{\_tabuleiro} &= n_c \cdot n_d \\
\texttt{\_tabuleiro} &= 5 \cdot 4 \\
\texttt{\_tabuleiro} &= 20 \ \textit{bits}
\end{aligned}
\tag{3.1}
$$

Na Equação 3.1, n_c e n_d são o número de cores e o número de discos do jogo, respectivamente. Seus valores são no máximo $n_c = 5$ e $n_d = 4$.

O cálculo de bits do atributo `peao` é apresentado na equação 3.2.

$$
\begin{aligned}
\texttt{\_peao} &= \lceil \log_2(n_d + 1) \rceil \cdot n_p \\
\texttt{\_peao} &= \lceil \log_2(5 + 1) \rceil \cdot 4 \\
\texttt{\_peao} &= 3 \cdot 4 \\
\texttt{\_peao} &= 15 \ \textit{bits}
\end{aligned}
\tag{3.2}
$$

Na Equação 3.2, o valor de n_d é o número de discos e n_p é o número de peões do jogo, que por sua vez é igual a n_c (número de cores comuns). Cada peão pode estar: fora do tabuleiro, com $peao(p_i) = 0$; em cima de um disco da sua cor, com $peao(p_i) \in \{1, 2, ..., n_d\}$; e na escada, com $peao(p_i) = n_d + 1$.

O cálculo de bits do atributo `escada` é apresentado na equação 3.3.

$$
\begin{aligned}
\texttt{\_escada} &= \lceil \log_2(n_p + 1) \rceil \cdot n_p \\
\texttt{\_escada} &= \lceil \log_2(6) \rceil \cdot 5 \\
\texttt{\_escada} &= 15 \ \textit{bits}
\end{aligned}
\tag{3.3}
$$

A Equação 3.3 possui as variáveis n_p e n_c com $n_p, n_c \in \{2, 3, 4, 5\}$ e $n_p = n_c$. Cada peão tem um local na escada, que armazena a posição dele de forma que $0 \leqslant escada(p_i) \leqslant n_c$. As situações possíveis são: $escada(p_i) = 0$ quando o peão não estiver na escada; e $escada(p_i) \in \{1, 2, 3, 4, 5\}$ sendo a ordem de chegada do peão na escada[4].

[4] O primeiro peão p_i a chegar na escada é indicado com $escada(p_i) = 1$.

O cálculo de bits do atributo `jogadores` é apresentado na equação 3.4.

$$
\begin{aligned}
\texttt{\_jogadores} &= \lceil \log_2(n_d + 1) \rceil \cdot n_c \cdot n_j \\
\texttt{\_jogadores} &= \lceil \log_2(4 + 1) \rceil \cdot 5 \cdot 2 \\
\texttt{\_jogadores} &= 3 \cdot 5 \cdot 2 \\
\texttt{\_jogadores} &= 30 \ bits
\end{aligned}
\tag{3.4}
$$

A capacidade da variável `_jogadores` é de 30 *bits*, como demonstrado na equação **??**. As variáveis utilizadas nessa equação são: n_d, o número de discos $n_d \in \{1, 2, 3, 4, 5\}$; n_c, o número de cores $n_c \in \{1, 2, 3, 4, 5\}$; e n_j, o número de jogadores $n_j = 2$. A informação armazenada na mão dos jogadores, para cada disco, vai até o número máximo de discos mais um, pois o jogador pode pegar todos os discos no tabuleiro e o disco adquirido ao mover o peão para a escada. Para armazenar o número seis, são necessários $\lceil \log_2(6) \rceil = 3 bits$

O cálculo de bits do atributo `atual` é apresentado na equação 3.5.

$$
\begin{aligned}
\texttt{\_atual} &= \lceil \log_2(2) \rceil \\
\texttt{\_atual} &= 1 \ bit
\end{aligned}
\tag{3.5}
$$

3.4 Funções de acesso da estrutura `State`

A estrutura possui um construtor que atribui valores às variáveis através de RAII[5], dessa forma não se faz necessário nenhuma extra implementação. Todas as variáveis possuem um valor padrão, verdadeiro para qualquer tamanho de tabuleiro t_i, onde $4 \leqslant t_i \leqslant 20$.

Código Fonte 10 – 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)
{
}
```

cpp programing language criador do c++

Atributo `tabuleiro`

Código Fonte 11 – Funções de acesso ao atributo `tabuleiro`

```
int tabuleiro (int pos) const {
```

5 *Resource Aquisition Is Initialization* é uma técnica de programação que vincula o ciclo de vida do recurso ao da estrutura (CUBBI; MAGGYERO; FRUDERICA,).

```
        return (_tabuleiro & (1<<pos))>>pos;
    }

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

Atributo `peao`

Código Fonte 12 – 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);
    }
```

Atributo `escada`

Código Fonte 13 – 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));
    }
```

Atributo `jogador`

Código Fonte 14 – 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);
    }
```

Atributo `atual`

Código Fonte 15 – Funções de acesso ao atributo `atual`

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

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

3.5 Comparador da estrutura `State`

Código Fonte 16 – 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;
}
```

Ao final do cálculo deste jogo reduzido, temos que o número de estados distintos varia entre 17 e 25, dependendo do estado inicial do tabuleiro. Devido a este grande número de estados repetidos, escrever o algoritmo fazendo uso de programação dinâmica economizou bastante tempo e processamento.

O jogo seria um jogo balanceado se ambos os jogadores ganharem aproximadamente metade das vezes. Se existem seis jogos diferentes (combinação de duas cores com dois discos cada), o jogo é considerado balanceado se cada jogador ganhar três jogos. Neste caso, temos os jogos $j_i \in \{1122, 1212, 1221, 2112, 2121, 2211\}$, e para cada j_i temos a pontuação máxima e a quantidade de estados distintos, como demonstrado na tabela +@tbl:1.

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, na maioria das vezes. Isso torna o jogo desequilibrado.

long long int, de forma que cada estado ocupasse apenas 64 *bits* na memória.

Tabela 5 – Pontuação utilizando *minimax*

Jogo	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

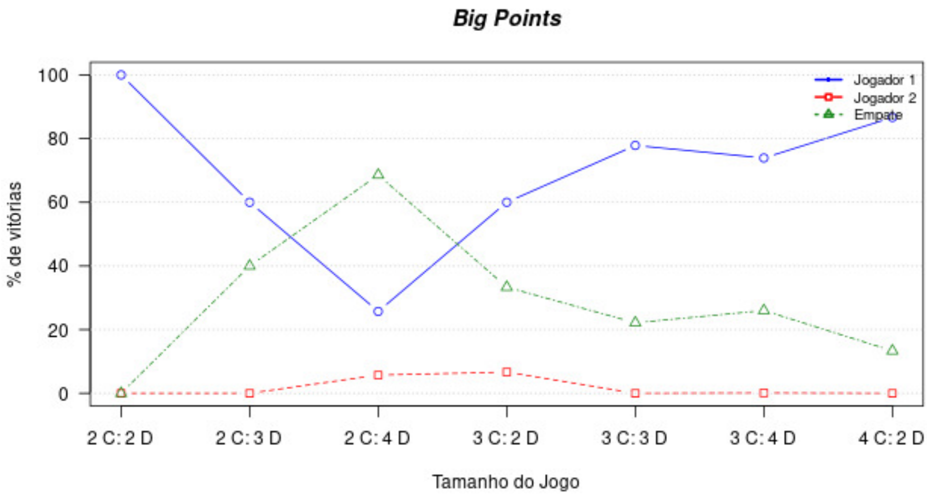


Figura 10 – Resultados ordenados por número de cores

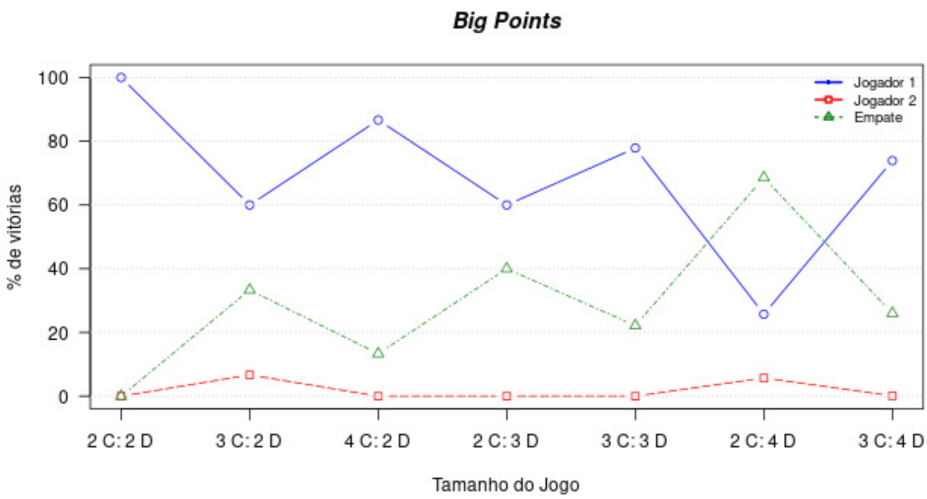


Figura 11 – Resultados ordenados por número de discos

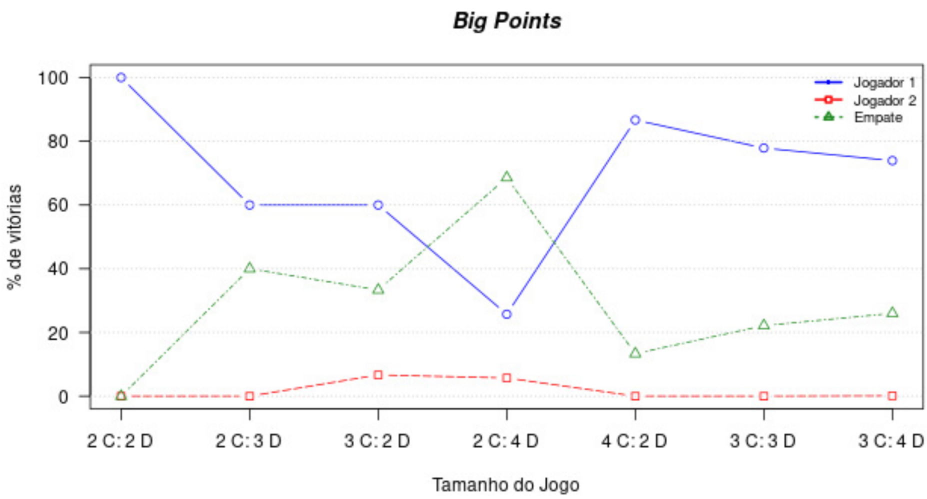


Figura 12 – Resultados ordenados por número total de peças

4 Considerações Finais

A análise utilizada para solucionar[1] o jogo neste trabalho foi o teorema *minimax*, onde cada jogador tenta aumentar sua pontuação e diminuir a pontuação do oponente. Os resultados obtidos ao final da análise computacional baseadas neste teorema sugere a possibilidade do jogo completo ser desbalanceado[2], dando ao primeiro jogador uma maior chance de vencer o jogo.

4.1 Trabalhos futuros

Desenvolvimento de uma I.A. para competir contra um jogador humano. Análise mais complexa do jogo *Big Points*, utilizando processamento paralelo e distribuído.

[1] Solucionar um jogo é percorrer todas as sua possibilidades de movimento e seus resultados.
[2] É dito um jogo balanceado aquele que a chance dos jogadores de ganhar é a mesma.

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 15 e 17.

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 12 e 13.

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 13.

BOREL Émile. *The Theory of Play and Integral Equations with Skew Symmetric Kernels*. 1921. Citado na página 12.

BOREL Émile. *On Games that Involve Chance and the Skill of Players*. 1924. Citado na página 12.

BOREL Émile. *On Systems of Linear Forms of Skew Symmetric Determinant and the General Theory of Play*. 1927. Citado na página 12.

CARMICHAEL, F. *A Guide to Game Theory*. [S.l.: s.n.], 2005. Citado na página 13.

CORMEN, T. et al. *Introduction To Algorithms*. MIT Press, 2001. ISBN 9780262032933. Disponível em: <https://books.google.com.br/books?id=NLngYyWFl_YC>. Citado na página 19.

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 12.

CUBBI; MAGGYERO; FRUDERICA. *RAII*. <http://en.cppreference.com/w/cpp/language/raii>. Accessed May 31, 2017. Citado na página 38.

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 13.

JONES, A. J. *Game Theory*: Mathematical models of conflict. [S.l.: s.n.], 1980. Citado 2 vezes nas páginas 14 e 16.

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 12.

NASH, J. J. F. *The Bargaining Problem*. 1950. Disponível em: <https://www.econometricsociety.org/publications/econometrica/1950/04/01/bargaining-problem>. Citado na página 13.

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 12.

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 13.

NEUMANN, J. von. *Zur Theorie der Gesellschaftsspiele*. [S.l.]: Mathematische Annalen, 1928. 295–320 p. Citado na página 12.

NEUMANN, J. von; MORGENSTERN, O. *Theory of Games and Economic Behavior*. [S.l.]: Princeton University Press, 1944. Citado na página 12.

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 12.

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 16.

SARTINI, B. A. et al. *Uma Introdução a Teoria dos Jogos*. 2004. Citado 2 vezes nas páginas 12 e 14.

SCHELLING, T. *The Strategy of Conflict*. Harvard University Press, 1960. Disponível em: <https://books.google.com.br/books?id=7RkL4Z8Yg5AC>. Citado na página 13.

SCHWABER, K.; SUTHERLAND, J. *The Scrum Guide*. [S.l.]: Scrum.Org, 2016. Citado na página 25.

ZERMELO, E. F. F. *Über eine Anwendung der Mengdenlehre auf die theories des Schachspiels*. 1913. 501–504 p. Citado na página 12.

Anexos

ANEXO A – Regras Originais do Jogo *Big Points*

Big Points

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

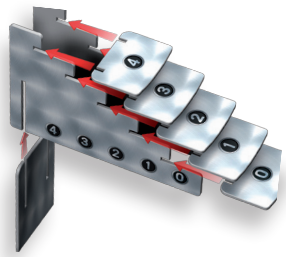
PORTUGUES

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.

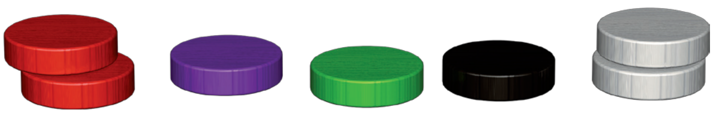
Em seguida, calculam-se os pontos:

- 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!