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

Coorientador:

Brasília, DF

2016

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

2016

Mateus Medeiros Furquim Mendonça

Big Points: Uma Análise Baseada na Teoria dos Jogos/ Mateus Medeiros Furquim Mendonça. – Brasília, DF, 2016-

55 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 , 2016.

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 2016:

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
2016

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 Combinatorial de Jogos.

Abstract

Key-words: Game Theory, Combinatorial Game Theory.

Lista de ilustrações

Lista de tabelas

Lista de símbolos

Símbolos para conjuntos e operações matemáticas

$\varnothing$	Um conjunto sem elementos, conjunto vazio
$\{\ \}$	Delimita conjunto, de forma que $S = \{\}$ é um conjunto vazio
$\forall$	Para cada elemento
$x \in S$	Elemento x pertence ao conjunto S
$x \notin S$	Elemento x não pertence ao conjunto S
$S \subseteq T$	Conjunto S é um subconjunto de T, significa que se $x \in S$ então $x \in T$
$S \cup T$	União entre dois conjuntos $\{x\ ; x \in S \text{ or } x \in T\}$
$S \cap T$	Inteseção entre dois conjuntos $\{x\ ; x \in S \text{ and } x \in T\}$
$S_1 \times \ldots \times S_n$	Produto cartesiano $\{(x_1, \ldots, x_n)\ ; x_i \in S_i (1 \leq i \leq n)\}$
$\sum_{i=1}^{n} x_i$	Somatório de x_1 até x_n de maneira que $\sum_{i=1}^{n} x_i = x_1 + x_2 + \ldots + x_n$
$\prod_{i=1}^{n} x_i$	Produto de x_1 até x_n de maneira que $\prod_{i=1}^{n} x_i = x_1 \cdot x_2 \cdot \ldots \cdot x_n$
$A_{p,q}$	Arranjo de p elementos tomados de q a q calculado $A_{p,q} = \frac{p!}{(p-q)!}$
$\binom{p}{q}$	Combinação de p elementos tomados de q a q calculado $\binom{p}{q} = \frac{p!}{q! \cdot (p-1)!}$

Para jogos de soma zero com dois jogadores

σ, τ	Estratégias puras
x	Estratégia mista para o jogador 1
X	Conjunto de todas as estratégias mistas para o jogador 1
y	Estratégia mista para o jogador 2
Y	Conjunto de todas as estratégias mistas para o jogador 2
$P(x, y)$	Ganho do jogador 1

Para jogos não cooperativos com n jogadores

σ_i	uma estratégia pura para o jogador i
S_i	Conjunto de todas as estratégias puras para o jogador i
x_i	uma estratégia mista para o jogador i
X_i	Conjunto de todas as estratégias mistas para o jogador i
$P_i(x_1, \ldots, x_n)$	Ganho do jogador i
$x \| x'_i$	Considerando $x = (x_1, \ldots, x_n)$ o conjunto com todas as estratégias dos n jogadores, jogador i substitui a estratégia x_i pela estratégia x'_i

Sumário

1 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 em como um todo levará a uma certa situação chamada **resultado**. Os membros deste grupo são chamados de **jogadores** e as regras que eles concordaram a obedecer constitui um **jogo**. A área da teoria dos jogos engloba estes conceitos para que seja possível realizar suas análises.

A proposta deste trabalho foi realizar uma análise de um jogo de tabuleiro, já existente, chamado *Big Points*. Utilizando-se de conceitos da teoria dos jogos e programação dinâmica, foi escrito um programa para exaurir todas as possibilidades de jogadas, e de todas as condições iniciais distintas, de uma quantidade reduzida de peças do jogo. Os resultados finais sugerem a possibilidade do jogo ser desbalanceado[1], dando ao primeiro jogador uma maior chance de vencer o jogo.

A motivação que levou à realização deste trabalho foi identificar uma heurística na qual tem-se uma maior chance de ganhar. Dessa forma, seria possível programar uma IA[2] com diferentes dificuldades para jogar contra o jogador. No entanto, devido à complexidade estimada do trabalho, seu objetivo principal foi encontrar o *winning move*[3] para que dê insumo a trabalhos futuros, como a implementação desta IA.

A estrutura do trabalho foi dividida em cinco capítulos, sendo o primeiro esta introdução. O capítulo seguinte (2), 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 3, referente à análise e ao desenvolvimento do projeto até sua conclusão, e no capítulo 4 os resultados desta análise são discutidos. Por último, o capítulo 5 onde é feita as considerações finais do trabalho e são citados alguns possíveis trabalhos futuros em cima do trabalho atual.

[1] É dito um jogo balanceado aquele que a chance dos jogadores de ganhar é a mesma.

[2] Inteligência Artificial.

[3] *Winning move* é aquele movimento que um jogador faz que lhe garante à vitória, independente das jogadas restantes dos outros jogadores.

2 Fundamentação Teórica

Para um bom entendimento das análises realizadas no jogo *Big Points* é preciso ter 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 2.2 explica um pouco sobre os conceitos da teoria dos jogos, mas apenas o necessário para este trabalho. Em seguida, tem-se a seção 2.2.1 que explica alguns métodos de solucionar um jogo. Na seção 2.3.2, os conceitos sobre programação são explicados e, por fim, na seção 2.4 as regras do jogo *Big Points* são explicadas.

2.1 Histórico da Teoria dos Jogos

Pode-se dizer que a análise de jogos é praticada desde o séculco 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, p. 2). No século seguinte, o matemático e filósofo Augustin Cournot fez uso da teoria dos jogos para estudos relacionados à política. Mais recentemente, em 1913, Ernst Zermelo publicou o primeiro teorema matemático da teoria dos jogos (SARTINI et al., 2004, p. 2).

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 quatro artigos sobre jogos estratégicos (PRAGUE, 2004, p. 2), introduzindo uma noção abstrada sobre jogo estratégico e estratégia mista[1]. Em 1928, John von Neumann demonstrou que todo jogo finito[2] de soma zero[3] com duas pessoas possui uma solução em estratégias mistas. Em 1944, Neumann publicou um trabalho junto a Oscar Morgenstern introduzindo a teoria dos jogos na área da economia e matemática aplicada (SARTINI et al., 2004, p. 2–3).

[1]Estratégia mista é um conjunto de estratégias puras associadas a uma distribuição de probabilidade (FIGUEIREDO, 2001).

[2]Jogos finitos são aqueles onde cada participante se depara com um conjunto finito de escolhas, ou seja, eles escolhem suas estratégias dentro de um conjunto finito de alternativas (FIGUEIREDO, 2001).

[3]Um jogo soma zero é um jogo no qual a vitória de um jogador implica na derrota do outro.

2.2 Teoria dos Jogos

A Teoria dos Jogos pode ser definida como a teoria dos modelos matemáticos que estuda a escolha de decisões ótimas[4] sob condições de conflito[5]. Os elementos básicos de um jogo são o conjunto de **jogadores**, onde cada jogador possui um conjunto de **estratégias** e, a partir das escolhas de estratégias de cada jogador, temos uma **situação** ou **perfil**. Para cada perfil do jogo, tem-se um resultado no final do jogo. 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***. Em outras palavras, 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 como mostra a tabela table 1. Além disso o ganho dos jogadores é representado como uma tupla (ou par) de valores, sendo que o primeiro é o ganho do primeiro jogador e o segundo valor, o do segundo jogador.

Tabela 1 – Matriz de ganho.

$P_2 \setminus P_1$	E_{11}	E_{12}
E_{21}	(1,0)	(2,3)
E_{22}	(3,4)	(0,2)

Dessa forma, o primeiro jogador, que é representado por P_1, possui as estratégias E_{11} e E_{12}. Semelhante ao primeiro jogador, tem-se o segundo jogador sendo representado por P_2 e com as estratégias E_{21} e E_{22}. Os valores que se encontram na interseção da estratégia de P_1 e P_2 são os ganhos dos dois jogadores, dessa forma se as estratégias escolhidas forem E_{12} e E_{21}, o primeiro jogador teria perdido com 3 pontos e o segundo jogador venceria com 4 pontos.

De uma forma matemática mais genérica, tem-se o jogador $j \in 1, 2$

2.2.1 Soluções de um jogo

Uma solução de um jogo é uma prescrição ou previsão sobre o resultado do jogo. Dois métodos importantes para encontrar a solução de um estado do jogo são **dominância** e **equilíbrio de Nash**.

[4] É considerado que os jogadores são seres racionais e que possuem conhecimento completo das regras do jogo. Às vezes o jogador também possui informação completa sobre o estado atual e do histórico de jogadas do jogo.

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

É dito que uma determinada estratégia é uma **estratégia dominante** quando esta é a única estratégia restante após aplicar a técnica de **dominância estrita iterada**. O encontro das estratégias dos jogadores é chamado de **equilíbrio de estratégia dominante**.

Dominância estrita iterada nada mais é do que um processo onde se eliminam as estratégias que são estritamente dominadas. Obs.: faltou explicar o que é uma estratégia dominada.

Solução estratégica ou **Equilíbrio de Nash** é um conjunto de estratégias para cada jogador onde cada um deles não tem incentivo de mudar sua estratégia se os demais jogadores não o fizerem.

Zero-sum game: a vitória de um jogador implica na derrota do outro. No Big Points, o jogador com maior pontuação vence. Pode-se dar pontuação 1 caso o jogador em questão é o vencedor, e -1 para o jogador que perdeu. Caso haja mais de um jogador com a maior pontuação do jogo, é dado 0 para o payoff dos dois jogadores.

Outra maneira, mais refinada, de demonstrar a vitória e derrota entre os jogadores é calcular a difereça da pontuação entre eles. O jogador com a maior pontuação mantém sua pontuação, e o restante tem sua pontuação subtraída daquela maior pontuação do jogo (dando um resultado negativo).

Backward Induction - As long as every player take turns you can start at the end of the game and make your way to the begin. - One strategy for every decision node

Game Theory the study of strategic interaction among rational decision makers players: people playing the game; each player has a set of strategies strategies: what they will do, how they'll respond payoffs: result of the interaction of strategies

strategy is a set with what decision you will make for every decision making situation in the game

each players is chosen an strategy, these strategies interact, and the game plays out to its conclusion.

rationality and common knowledge

Teoria dos jogos é o estudo do comportamento estratégico interdependente[6], não apenas o estudo de como vencer ou perder em um jogo, apesar de às vezes esses dois fatos coincidirem. Isso faz com que o escopo seja mais abranjente, desde comportamentos no qual as duas pessoas devem cooperar para ganhar, ou as duas tentam se ajudar para ganharem independente ou, por fim, comportamento de duas pessoas que tentam vencer individualmente (SPANIEL, 2011).

[6]Estratégia interdependente significa que as ações de uma pessoa interfere no resultado da outra, e vice-versa.

2.3 Conceitos Relevantes

Alguns ceonceitos fundamentais para o entendimento da análise realizada em cima do jogo *Big Points* são *zero-sum game* e *minimax*.

Como o jogo não possui nenhum elemento dependente da sorte, não serão usados estratégias mistas. O *winning move* não foi analizado devido à complexidade da implementação da análise atual.

2.3.1 Minimax

2.3.2 Programação dinâmica

2.4 Regras do 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 1, um de cada uma das seguintes cores, denominadas **cores comuns**: vermelha, verde, azul, amarela e violeta. Para cada cor de peão, tem-se dez discos, como mostrado na figura 2a, (totalizando cinquenta discos) denominados **discos comuns**, e cinco discos das cores branca e preta (totalizando dez discos) denominados **discos especiais**. 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[7], o segundo peão, três pontos e assim por diante, até o último valer zero pontos.

No final da preparação, o jogo ficará parecido com as peças na figura 2b. 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 coisa é retirar um disco de cada cor comum e posicioná-los ao lado da escada, estes serão os discos coletados pelo jogador que subir o peão da sua cor para a escada. Em seguida, deve-se embaralhar todos os 55 discos restantes[8] e formar uma fila até a escada, estes são os

[7] No caso de um jogo com menos de cinco peões, a seguinte fórmula se aplica: $Score = N_c - P_{pos}$, onde *Score* é a pontuação daquela determinada cor, N_c é o número de discos comuns e P_{pos} é a posição do peão no pódio.

[8] 9 discos de cada uma das 5 cores comuns mais 5 discos de cada uma das 2 cores especiais resultando

Figura 1 – Caixa do jogo **Big Points**

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 fique oposto à escada.

Após preparar o jogo, deve-se escolher o primeiro jogador de forma aleatória. Na 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 para pegar o primeiro disco disponível[9] 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 que o jogador possa pegar, retire-o do tabuleiro e coloque-o na 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.

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.

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

(a) Conteúdo do jogo **Big Points** (b) Preparação do jogo **Big Points**

Figura 2 – Organização do jogo **Big Points**

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: se o primeiro jogador tiver dois discos vermelhos, um disco verde e três azuis e a ordem de chegada deles for azul em primeiro lugar, verde logo em seguida e depois o vermelho, sua pontuação será descrita de acordo com a equação , onde n_c é o número de cores do jogo, n_r, n_g e n_b são as quantidades de discos vermelhos, verdes e azuis, respectivamente, que o jogador possui e p_r, p_g e p_b são as posições dos peões vermelho, verde e azul, respectivamente, na escada.

$$
\begin{aligned}
Pontuacao &= n_r \cdot (n_c - p_r) + n_g \cdot (n_c - p_g) + n_b \cdot (n_c - p_b) \\
Pontuacao &= 2 \cdot (3 - 3) + 1 \cdot (3 - 2) + 3 \cdot (3 - 1) \qquad \text{(e.q. Exemplo de pontuação)} \\
Pontuacao &= 7
\end{aligned}
$$

3 Metodologia

3.1 *Scrum*

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. Inicialmente, tal método de organização e gerenciamento do projeto foi aplicado para o desenvolvimento do sistema em questão (SCHWABER; SUTHERLAND, 2016). O *kanban* do waffle.io foi utilizado para registrar tarefas devido à sua integração com as *issues* do github. 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 *branchs* 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[1]. Por fim, para trabalhar em outra *branch*, foi sempre necessário atualizá-la em relação à mestre[2].

3.2 Análise do jogo *Big Points*

Para analizar o jogo *Big Points*, é preciso realizar todas as jogadas de todos os jogos possíveis. Cada jogador, na sua vez, deve escolher uma jogada na qual lhe garanta a vitória, se houver mais de uma, escolha a que tiver a maior pontuação. Caso não tenha uma jogada para vencer, o jogador deve minimizar a pontuação do adversário. Após fazer isso para um jogo inicial, os resultados são escritos em um arquivo *csv* para análise. Esse procedimento é repetido para *cada* organização possível do tabuleiro inicial.

Exaurir todas as possibilidades de jogadas é um trabalho computacional 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. Isso gera uma árvore onde cada nó possui quatro filhos e a altura média dessa árvore é quatro, totalizando uma quantidade de estados de aproximadamente $\sum_{h=0}^{4} 4^h \approx 341$. Ao final do cálculo deste jogo reduzido, temos que o número de estados distintos varia entre 17 e 25, dependendo do

[1] `$ git checkout <to-branch>; git merge <from-branch>`

[2] `$ git rebase <from-branch> <to-branch>`

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 table 2.

Tabela 2 – Pontuação utilizando Minimax.

Jogo	**Pontuação**	**#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

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.

3.2.1 Quantidade de partidas

$$\begin{aligned}
Partidas &= (\#J - 1) \cdot \binom{\#D_T}{\#D_W} \cdot \binom{\#D_{L1}}{\#D_K} \cdot \binom{\#D_{L2}}{\#D_R} \cdot \binom{\#D_{L3}}{\#D_G} \cdot \binom{\#D_{L4}}{\#D_B} \cdot \binom{\#D_{L5}}{\#D_Y} \cdot \binom{\#D_{L6}}{\#D_V} \\
Partidas &= 4 \cdot \binom{55}{5} \cdot \binom{50}{5} \cdot \binom{45}{9} \cdot \binom{36}{9} \cdot \binom{27}{9} \cdot \binom{18}{9} \cdot \binom{9}{9} \\
Partidas &= 560'483'776'167'774'018'942'304'261'616'685'408'000'000 \\
Partidas &\approx 5 \times 10^{41}
\end{aligned}$$

(e.q. Quantidades de Partidas Distintas)

3.3 Estrutura de dados

Devido à enorme quantidade de estados de um jogo reduzido de *Big Points*, foi implementado duas funções para codificar e decodificar a *struct State* para um *long long int*, de forme que ocupe apenas 64 *bits* na memória. Após testar nos limites da capacidade da variável, percebeu-se um erro quando executado com quatro cores e cinco discos, o que levou à implementação por *bit fields*.

3.3.1 Estado do jogo

Para escrever a programação dinâmica capaz de

3.3.2 Bit fields

Dentro da estrutura `State` foi declarado duas estruturas anônimas[3] 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 da variável, é declarado a quantidade de *bits* que será utilizado para ela, 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 desconhecido quando a variável é ultrapassada, e para garantir consistência no armazenamento, foi utilizado duas *structs* com, no máximo, uma variável `unsigned long long int` (64 *bits*).

A estrutura `State` possui cinco variáveis: `_tabuleiro`, no qual pode armazenar informações sobre um tabuleiro até 20 discos[4]; `_peao`, que representa a posição $p_i \in \{0, 1, ..., n_d, n_d + 1\}$, onde n_d é o número de discos de cores comuns no jogo e p_i é o peão da cor i[5]; `_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.

```
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;

```

[3] 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`.

[4] Cinco cores e quatro discos.

[5] 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.

```
        // 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
        ll _atual :1;
    };
```

O cálculo para determinar os *bits* necessários para armazenar as informações de cada variável foi realizado da seguinte forma:

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

Na equação e.q. *bits* de _tabuleiro, 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 $n_d = 4$.

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

Na segunda equação, e.q. *bits* de _peao, 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 $peao(p_i) = n_d + 1$.

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

A equação e.q. *bits* de _escada 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[6].

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

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 $\lceil \log_2(6) \rceil = 3bits$

$$\begin{aligned} \texttt{\_atual} &= \lceil \log_2(2) \rceil \\ \texttt{\_atual} &= 1 \; bit \end{aligned} \tag{e.q. \textit{bits} de \_atual}$$

3.3.3 Funções de acesso

A estrutura possui um construtor que atribui valores às variáveis através de RAII[7], 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 $4 \leqslant t_i \leqslant 20$.

```
        int mjogadores = 0, int matual = 0) : _tabuleiro(mtabuleiro),
        _peao(mpeao), _escada(mescada), _jogadores(mjogadores),
        _atual(matual)
    {
    }
        return (_tabuleiro & (1<<pos))>>pos;
    }

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

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

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

3.3.4 Comparador

3.4 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 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 cadaCaso 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*, temos 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

3.4.1 Função dp

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

```
#include <bitset>
#include <iostream>
#include <vector>

#include "dp.h"
#include "game.h"
#include "turn.h"

using namespace std;
using ii = pair<int, int>;
using game_res = pair<bool, ii>;

game_res play(map<struct State,ii>& dp_states, struct Game game, struct State stat
{
    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]]
        }
    } 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);
}

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);
        }
    }

    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];
}

bool is_pawns_stair(struct Game& game, struct State& state)
{
    for (int i = 0; i < game.num_cores; i++) {
        if (state.escada(i) == 0) {
            return false;
        }
    }

    return true;
}

ii calculate_score(struct Game& game, struct State& state)
{
    vector<short> score(game.num_jogadores, 0);
    for (int j = 0; j < game.num_jogadores; j++) {
        for (int disc = 0; disc < game.num_cores; disc++) {
            if (state.escada(disc) != 0) {
                score[j] += state.jogador(j,disc)*(game.num_cores - state.escada(d
            }
        }
    }

    return ii(score[0],score[1]);
}

short max_in_escada(struct Game& game, struct State& state)
{
    short highest = 0;

    for (size_t i = 0; i < game.num_cores; i++) {
        highest = max(highest, (short) state.escada(i));
    }

    return highest;
}

//void print_game(ostream& out, struct Game game, struct State& state)
//{
```

```
//   out << state.jogador_atual+1 << " - ";
//   if (state.peao[0] && state.peao[0] <= game.num_discos) {
//       game.board[game.color_index[0][state.peao[0]-1]] = 'R';
//   }
//   if (state.peao[1] && state.peao[1] <= game.num_discos) {
//       game.board[game.color_index[1][state.peao[1]-1]] = 'G';
//   }
//   out << game.board << "  (";
//   for (short c = 0; c < game.num_cores; c++) {
//       if (c) out << ",";
//       out << state.escada[c];
//   }
//   out << ")  ";
//   for (short j = 0; j < game.num_jogadores; j++) {
//       if (j) out << "  ";
//       out << j+1 << ": ";
//       for (short c = 0; c < game.num_cores; c++) {
//           if (c) out << ",";
//           out << state.jogadores[j][c];
//       }
//   }
//
//   return;
//}
```

3.4.2 Função play

```
#include <bitset>
#include <iostream>
#include <vector>

#include "dp.h"
#include "game.h"
#include "turn.h"

using namespace std;
using ii = pair<int, int>;
using game_res = pair<bool, ii>;

game_res play(map<struct State,ii>& dp_states, struct Game game, struct State
{
    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]] !=
        }
    } 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);
}

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);
        }
    }

    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];
}

bool is_pawns_stair(struct Game& game, struct State& state)
{
    for (int i = 0; i < game.num_cores; i++) {
        if (state.escada(i) == 0) {
            return false;
        }
    }

    return true;
}

ii calculate_score(struct Game& game, struct State& state)
{
    vector<short> score(game.num_jogadores, 0);
    for (int j = 0; j < game.num_jogadores; j++) {
        for (int disc = 0; disc < game.num_cores; disc++) {
            if (state.escada(disc) != 0) {
                score[j] += state.jogador(j,disc)*(game.num_cores - state.esca
            }
        }
    }

    return ii(score[0],score[1]);
}

short max_in_escada(struct Game& game, struct State& state)
{
    short highest = 0;

    for (size_t i = 0; i < game.num_cores; i++) {
        highest = max(highest, (short) state.escada(i));
    }

    return highest;
}

//void print_game(ostream& out, struct Game game, struct State& state)
```

```
//{
//   out << state.jogador_atual+1 << " - ";
//   if (state.peao[0] && state.peao[0] <= game.num_discos) {
//       game.board[game.color_index[0][state.peao[0]-1]] = 'R';
//   }
//   if (state.peao[1] && state.peao[1] <= game.num_discos) {
//       game.board[game.color_index[1][state.peao[1]-1]] = 'G';
//   }
//   out << game.board << "  (";
//   for (short c = 0; c < game.num_cores; c++) {
//       if (c) out << ",";
//       out << state.escada[c];
//   }
//   out << ")  ";
//   for (short j = 0; j < game.num_jogadores; j++) {
//       if (j) out << "  ";
//       out << j+1 << ": ";
//       for (short c = 0; c < game.num_cores; c++) {
//           if (c) out << ",";
//           out << state.jogadores[j][c];
//       }
//   }
//
//   return;
//}
```

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

3.5 Verificação dos estados

Foi escrito os estados e suas transições em _post-it_s para garantir que a *DP* foi feita corretamente. Os estados

4 Resultados

4.1 Trabalhos futuros

5 Conclusão

5.1 Trabalhos futuros

Referências

CUBBI; MAGGYERO; FRUDERICA. *RAII*. <http://en.cppreference.com/w/cpp/language/raii>. Accessed May 31, 2016. Citado na página 29.

FIGUEIREDO, R. S. *Teoria dos Jogos*: Conceitos, formalização matemática e aplicação à distribuição de custo conjunto. [S.l.]: Universidade Federal de São Carlos, 2001. 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.

SARTINI, B. A. et al. *Uma Introdução a Teoria dos Jogos*. 2004. Citado na página 19.

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

SPANIEL, W. *Game Theory 101*: The complete textbook. [S.l.: s.n.], 2011. Citado na página 21.

Anexos

ANEXO A – Regras Originais do Jogo *Big Points*

PORTUGUES

Big Points

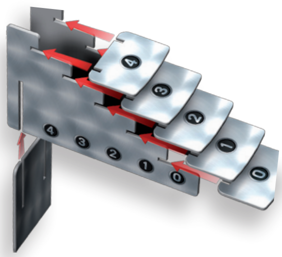
De Brigitte e Wolfgang Ditt
para 2 a 5 jogadores a partir dos 8 anos

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 restantes (e claro, os blancos e os pretos) e colocá-los como desejar de maneira a formar um percurso desde a base da escada. 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.

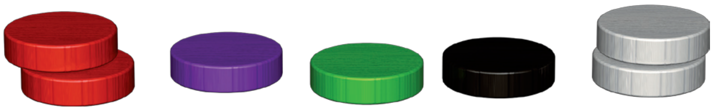
Em seguida, calculam-se os pontos:

- 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!