1 Introdução

A proposta deste trabalho é realizar uma análise em cima de um jogo de tabuleiro, já existente, chamado *Big Points*. Fazendo uso 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 corroboram com a ideia de que o jogo é desbalanceado[1], dando ao primeiro jogador uma maior chance de vencer o jogo.

A estrutura do trabalho foi dividida em cinco capítulos, sendo o primeiro esta introdução. O capítulo seguinte, de 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 a conclusão 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 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. Na

2.1 2.1.1 2.2 2.3 2.3.1 2.3.2 2.4

2.1 Teoria dos Jogos

A Teoria dos Jogos pode ser definida como a teoria dos modelos matemáticos que estuda a escolha de decisões ótimas sob condições de conflito. Os elementos básicos de um jogo são: o conjunto de **jogadores**, onde cada jogador possui um conjunto de **estratégias**. A partir das escolhas de estratégias de cada jogador, temos uma **situação** ou **perfil**.

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 na matriz da **forma normal**, 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.

2.1.1 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**.

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[1], 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).

2.2 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[2]. Em 1928, John von Neumann de-

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

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

monstrou que todo jogo finito[3] de soma zero[4] 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).

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[5], o segundo peão, três pontos e assim por diante, até o último valer zero pontos.

[3]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).

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

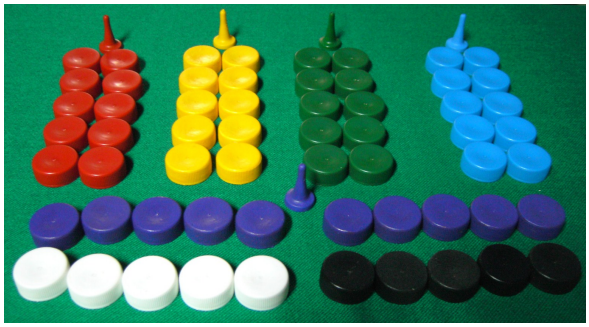
[5]No caso de um jogo com menos de cinco peões, a seguinte fórmula se aplica: $Score = N_c - P_{pos}$, onde *Score* é a pontuação daquela determinada cor, N_c é o número de discos comuns e P_{pos} é a posição do peão no pódio.

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

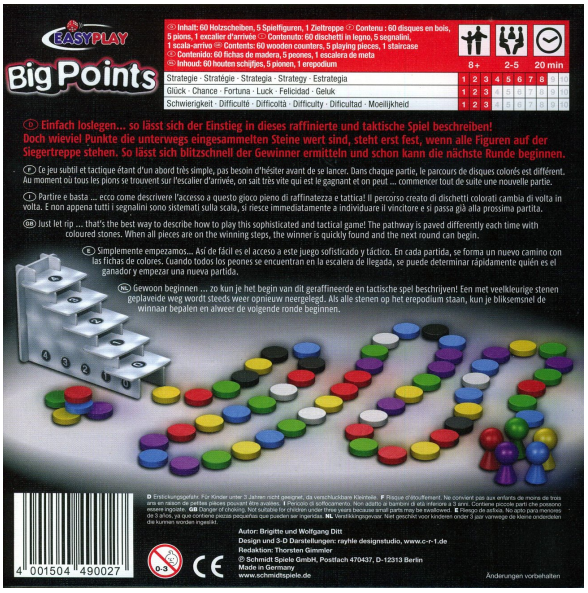
No final da preparação, o jogo ficará parecido com as peças na figura 2b. A preparação do jogo ocorre em algumas etapas envolvendo a posição dos peões, a aleatoriedade do tabuleiro e alguns discos ao lado da escada. A primeira coisa é retirar um disco de cada cor comum e posicioná-los ao lado da escada, 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[6] 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 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

[6] 9 discos de cada uma das 5 cores comuns mais 5 discos de cada uma das 2 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.



(a) Conteúdo do jogo **Big Points**



(b) Preparação do jogo **Big Points**

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

pegar o primeiro disco disponível[7] 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.

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.

$$Pontuacao = n_r \cdot (n_c - p_r) + n_g \cdot (n_c - p_g) + n_b \cdot (n_c - p_b)$$

$$Pontuacao = 2 \cdot (3 - 3) + 1 \cdot (3 - 2) + 3 \cdot (3 - 1) \qquad \text{(e.q. Exemplo de pontuação)}$$

$$Pontuacao = 7$$

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

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

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

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 table 1.

Tabela 1 – 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.

```
10 struct State
11 {
12     // Cinco cores, quatro discos
13     struct {
14         // 5 cores * 4 discos (1 bit pra cada)
15         ll _tabuleiro :20;
16
17         // 0..5 posições possíveis (3 bits) * 5 peões
18         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.

```
19
20          // 0..5 posições (3bits) * 5 peões
21          ll _escada :15;
22      };
23
24      struct {
25          // 0..5 discos (3bits) * 5 cores * 2 jogadores
26          ll _jogadores :30;
27
28          // Jogador 1 ou Jogador 2
29          ll _atual :1;
30      };
```

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

$$\begin{aligned} \texttt{\_tabuleiro} &= n_c \cdot n_d \\ \texttt{\_tabuleiro} &= 5 \cdot 4 \\ \texttt{\_tabuleiro} &= 20 \; bits \end{aligned} \qquad \text{(e.q. } bits \text{ de \_tabuleiro)}$$

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 $n_d = 4$.

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

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 $peao(p_i) = n_d + 1$.

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

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].

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

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

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

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 $4 \leqslant t_i \leqslant 20$.

```
34          int mjogadores = 0, int matual = 0) : _tabuleiro(mtabuleiro),
35          _peao(mpeao), _escada(mescada), _jogadores(mjogadores),
36          _atual(matual)
37      {
38      }

42          return (_tabuleiro & (1<<pos))>>pos;
43      }
44
45      void settabuleiro (int pos, int available) {
46          _tabuleiro = (_tabuleiro & ~(1<<pos)) | ((available&1)<<pos);
47      }
```

[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,

```
1 #include <bitset>
2 #include <iostream>
3 #include <vector>
4
5 #include "dp.h"
6 #include "game.h"
7 #include "turn.h"
8
9 using namespace std;
10 using ii = pair<int, int>;
11 using game_res = pair<bool, ii>;
12
13 game_res play(map<struct State,ii>& dp_states, struct Game game, struct
14 {
15     short player = turn.current_player;
16     short pawn = turn.pawn_to_move;
17     bool pick_right = turn.pick_right;
```

```
18      short prev_pos = state.peao(pawn);
19
20      // Cannot move pawn, it's already on the stair
21      if (state.escada(pawn) != 0) {
22          // cout << "Can't move. Pawn already on the stair" << endl;
23          return game_res(false, ii(-1,-1));
24      }
25
26      // Remove discs from the board according to tabuleiro
27      for (size_t i = 0; i < game.board.size(); i++) {
28          if (state.tabuleiro(i) == 0) {
29              game.board[i] = '0';
30          }
31      }
32
33      // Moving Pawn
34      bool available = false;
35      bool in_range = false;
36      do {
37          if (state.peao(pawn) <= game.num_discos) {
38              state.movepeao(pawn);
39          }
40          in_range = state.peao(pawn) <= game.num_discos;
41          if (in_range) {
42              available = game.board[game.color_index[pawn][state.peao(paw
43          }
44      } while (!available && in_range);
45
46      // Step in the stair
47      if (!in_range) {
48          if (state.escada(pawn) == 0) {
49              state.setescada(pawn, max_in_escada(game, state)+1);
50              state.updatejogador(player, pawn);
51              state.updateatual();
52          }
53      }
54
55      // Update board: Discs under pawns are unavailable
56      for (int color = 0; color < game.num_cores; color++) {
```

```
57          // If pawn is in board
58          if (state.peao(color) != 0 and state.peao(color) <= game.num_di
59              // Removing disc under pawns' current position
60              game.board[game.color_index[color][state.peao(color)-1]] =
61          }
62      }
63
64      // If pawn is in board and was on the board before moving
65      if (state.peao(pawn)-1 > 0 and prev_pos > 0) {
66          // Replacing disc under pawn's previous position
67          game.board[game.color_index[pawn][prev_pos-1]] = '1' + pawn;
68      }
69
70      // Pick a disc if the pawn has moved within the range of the board
71      bool pick = false;
72      if (in_range) {
73          short pawn_pos = state.peao(pawn)-1;
74          short disc_pos = -1;
75
76          for (short i = 1;; i++) {
77              // Pick right
78              if (pick_right) {
79                  disc_pos = game.color_index[pawn][pawn_pos]+i;
80
81                  // Does not pick right (out of board)
82                  if (disc_pos >= (short) game.board.size()) {
83                      return game_res(false, ii(-1,-1));
84                  }
85              }
86              // Pick left
87              else {
88                  disc_pos = game.color_index[pawn][pawn_pos]-i;
89
90                  // Does not pick left (out of board)
91                  if (disc_pos < 0) {
92                      return game_res(false, ii(-1,-1));
93                  }
94              }
95
```

```
96              // Does not pick if disc is 0, try again
97              if (game.board[disc_pos] == '0') {
98                  continue;
99              }
100
101             // There is a disc to be picked
102             pick = true;
103             break;
104         }
105
106         // If There is a disc to be picked
107         if (pick) {
108             char pick_char = -1;
109
110             // Disc's char to pick
111             pick_char = game.board[disc_pos];
112
113             // Remove it from the board
114             state.settabuleiro(disc_pos, 0);
115
116             // Add it to the player's hand
117             state.updatejogador(player, pick_char-'1');
118
119             // Calculate next player
120             state.updateatual();
121         }
122     }
123
124     auto max_score = dp(dp_states, game, state);
125
126     return game_res(true, max_score);
127 }
128
129 ii dp(map<struct State,ii>& dp_states, struct Game game, struct State s
130 {
131     // If all pawns are in the stair
132     if (is_pawns_stair(game, state)) {
133         return calculate_score(game, state);
134     }
```

```
135
136     auto it = dp_states.find(state);
137     if (it != dp_states.end()) {
138         return dp_states[state];
139     }
140
141     vector<ii> results;
142     for (short pawn = 0; pawn < game.num_cores; pawn++) {
143         struct Turn right(state.atual(), pawn, true);
144         struct Turn left(state.atual(), pawn, false);
145
146         // DP após jogadas
147         game_res result = play(dp_states, game, state, left);
148         if (result.first) {
149             results.push_back(result.second);
150         }
151
152         result = play(dp_states, game, state, right);
153         if (result.first) {
154             results.push_back(result.second);
155         }
156     }
157
158     auto p1_order = [](const ii& a, const ii& b){
159         if (a.first > a.second) {
160             if (b.first > b.second) {
161                 return a.first > b.first ? true : false;
162             }
163             else {
164                 return true;
165             }
166         }
167         else if (a.first == a.second) {
168             if (b.first > b.second) {
169                 return false;
170             }
171             else if (b.first == b.second) {
172                 return a.first > b.first ? true : false;
173             }
```

```
174             else {
175                 return true;
176             }
177         }
178         else {
179             if (b.first >= b.second) {
180                 return false;
181             }
182             else {
183                 return a.second < b.second ? true : false;
184             }
185         }
186     };
187 
188     auto p2_order = [](const ii& a, const ii& b){
189         if (a.second > a.first) {
190             if (b.second > b.first) {
191                 return a.second > b.second ? true : false;
192             }
193             else {
194                 return true;
195             }
196         }
197         else if (a.second == a.first) {
198             if (b.second > b.first) {
199                 return false;
200             }
201             else if (b.second == b.first) {
202                 return a.second > b.second ? true : false;
203             }
204             else {
205                 return true;
206             }
207         }
208         else {
209             if (b.second >= b.first) {
210                 return false;
211             }
212             else {
```

```
213                 return a.first < b.first ? true : false;
214             }
215         }
216     };
217
218
219     if (state.atual() == 0) {
220         sort(results.begin(), results.end(), p1_order);
221     }
222     else {
223         sort(results.begin(), results.end(), p2_order);
224     }
225
226     dp_states[state] = results.size() == 0 ? ii(-1, -1) : results.front
227
228     return dp_states[state];
229 }
230
231 bool is_pawns_stair(struct Game& game, struct State& state)
232 {
233     for (int i = 0; i < game.num_cores; i++) {
234         if (state.escada(i) == 0) {
235             return false;
236         }
237     }
238
239     return true;
240 }
241
242 ii calculate_score(struct Game& game, struct State& state)
243 {
244     vector<short> score(game.num_jogadores, 0);
245     for (int j = 0; j < game.num_jogadores; j++) {
246         for (int disc = 0; disc < game.num_cores; disc++) {
247             if (state.escada(disc) != 0) {
248                 score[j] += state.jogador(j,disc)*(game.num_cores - sta
249             }
250         }
251     }
```

```
252
253     return ii(score[0],score[1]);
254 }
255
256 short max_in_escada(struct Game& game, struct State& state)
257 {
258     short highest = 0;
259
260     for (size_t i = 0; i < game.num_cores; i++) {
261         highest = max(highest, (short) state.escada(i));
262     }
263
264     return highest;
265 }
266
267 //void print_game(ostream& out, struct Game game, struct State& state)
268 //{
269 //  out << state.jogador_atual+1 << " - ";
270 //  if (state.peao[0] && state.peao[0] <= game.num_discos) {
271 //      game.board[game.color_index[0][state.peao[0]-1]] = 'R';
272 //  }
273 //  if (state.peao[1] && state.peao[1] <= game.num_discos) {
274 //      game.board[game.color_index[1][state.peao[1]-1]] = 'G';
275 //  }
276 //  out << game.board << "  (";
277 //  for (short c = 0; c < game.num_cores; c++) {
278 //      if (c) out << ",";
279 //      out << state.escada[c];
280 //  }
281 //  out << ")  ";
282 //  for (short j = 0; j < game.num_jogadores; j++) {
283 //      if (j) out << "  ";
284 //      out << j+1 << ": ";
285 //      for (short c = 0; c < game.num_cores; c++) {
286 //          if (c) out << ",";
287 //          out << state.jogadores[j][c];
288 //      }
289 //  }
290 //
```

```
291 //   return;
292 //}
```

3.4.2 Função play

```
 1 #include <bitset>
 2 #include <iostream>
 3 #include <vector>
 4
 5 #include "dp.h"
 6 #include "game.h"
 7 #include "turn.h"
 8
 9 using namespace std;
10 using ii = pair<int, int>;
11 using game_res = pair<bool, ii>;
12
13 game_res play(map<struct State,ii>& dp_states, struct Game game, struct
14 {
15     short player = turn.current_player;
16     short pawn = turn.pawn_to_move;
17     bool pick_right = turn.pick_right;
18     short prev_pos = state.peao(pawn);
19
20     // Cannot move pawn, it's already on the stair
21     if (state.escada(pawn) != 0) {
22         // cout << "Can't move. Pawn already on the stair" << endl;
23         return game_res(false, ii(-1,-1));
24     }
25
26     // Remove discs from the board according to tabuleiro
27     for (size_t i = 0; i < game.board.size(); i++) {
28         if (state.tabuleiro(i) == 0) {
29             game.board[i] = '0';
30         }
31     }
32
33     // Moving Pawn
34     bool available = false;
```

```
35      bool in_range = false;
36      do {
37          if (state.peao(pawn) <= game.num_discos) {
38              state.movepeao(pawn);
39          }
40          in_range = state.peao(pawn) <= game.num_discos;
41          if (in_range) {
42              available = game.board[game.color_index[pawn][state.peao(paw
43          }
44      } while (!available && in_range);
45
46      // Step in the stair
47      if (!in_range) {
48          if (state.escada(pawn) == 0) {
49              state.setescada(pawn, max_in_escada(game, state)+1);
50              state.updatejogador(player, pawn);
51              state.updateatual();
52          }
53      }
54
55      // Update board: Discs under pawns are unavailable
56      for (int color = 0; color < game.num_cores; color++) {
57          // If pawn is in board
58          if (state.peao(color) != 0 and state.peao(color) <= game.num_di
59              // Removing disc under pawns' current position
60              game.board[game.color_index[color][state.peao(color)-1]] =
61          }
62      }
63
64      // If pawn is in board and was on the board before moving
65      if (state.peao(pawn)-1 > 0 and prev_pos > 0) {
66          // Replacing disc under pawn's previous position
67          game.board[game.color_index[pawn][prev_pos-1]] = '1' + pawn;
68      }
69
70      // Pick a disc if the pawn has moved within the range of the board
71      bool pick = false;
72      if (in_range) {
73          short pawn_pos = state.peao(pawn)-1;
```

```
74          short disc_pos = -1;
75
76          for (short i = 1;; i++) {
77              // Pick right
78              if (pick_right) {
79                  disc_pos = game.color_index[pawn][pawn_pos]+i;
80
81                  // Does not pick right (out of board)
82                  if (disc_pos >= (short) game.board.size()) {
83                      return game_res(false, ii(-1,-1));
84                  }
85              }
86              // Pick left
87              else {
88                  disc_pos = game.color_index[pawn][pawn_pos]-i;
89
90                  // Does not pick left (out of board)
91                  if (disc_pos < 0) {
92                      return game_res(false, ii(-1,-1));
93                  }
94              }
95
96              // Does not pick if disc is 0, try again
97              if (game.board[disc_pos] == '0') {
98                  continue;
99              }
100
101             // There is a disc to be picked
102             pick = true;
103             break;
104         }
105
106         // If There is a disc to be picked
107         if (pick) {
108             char pick_char = -1;
109
110             // Disc's char to pick
111             pick_char = game.board[disc_pos];
112
```

```
113             // Remove it from the board
114             state.settabuleiro(disc_pos, 0);
115 
116             // Add it to the player's hand
117             state.updatejogador(player, pick_char-'1');
118 
119             // Calculate next player
120             state.updateatual();
121         }
122     }
123 
124     auto max_score = dp(dp_states, game, state);
125 
126     return game_res(true, max_score);
127 }
128 
129 ii dp(map<struct State,ii>& dp_states, struct Game game, struct State s
130 {
131     // If all pawns are in the stair
132     if (is_pawns_stair(game, state)) {
133         return calculate_score(game, state);
134     }
135 
136     auto it = dp_states.find(state);
137     if (it != dp_states.end()) {
138         return dp_states[state];
139     }
140 
141     vector<ii> results;
142     for (short pawn = 0; pawn < game.num_cores; pawn++) {
143         struct Turn right(state.atual(), pawn, true);
144         struct Turn left(state.atual(), pawn, false);
145 
146         // DP após jogadas
147         game_res result = play(dp_states, game, state, left);
148         if (result.first) {
149             results.push_back(result.second);
150         }
151 
```

```
152         result = play(dp_states, game, state, right);
153         if (result.first) {
154             results.push_back(result.second);
155         }
156     }
157 
158     auto p1_order = [](const ii& a, const ii& b){
159         if (a.first > a.second) {
160             if (b.first > b.second) {
161                 return a.first > b.first ? true : false;
162             }
163             else {
164                 return true;
165             }
166         }
167         else if (a.first == a.second) {
168             if (b.first > b.second) {
169                 return false;
170             }
171             else if (b.first == b.second) {
172                 return a.first > b.first ? true : false;
173             }
174             else {
175                 return true;
176             }
177         }
178         else {
179             if (b.first >= b.second) {
180                 return false;
181             }
182             else {
183                 return a.second < b.second ? true : false;
184             }
185         }
186     };
187 
188     auto p2_order = [](const ii& a, const ii& b){
189         if (a.second > a.first) {
190             if (b.second > b.first) {
```

```
191                 return a.second > b.second ? true : false;
192             }
193             else {
194                 return true;
195             }
196         }
197         else if (a.second == a.first) {
198             if (b.second > b.first) {
199                 return false;
200             }
201             else if (b.second == b.first) {
202                 return a.second > b.second ? true : false;
203             }
204             else {
205                 return true;
206             }
207         }
208         else {
209             if (b.second >= b.first) {
210                 return false;
211             }
212             else {
213                 return a.first < b.first ? true : false;
214             }
215         }
216     };
217 
218 
219     if (state.atual() == 0) {
220         sort(results.begin(), results.end(), p1_order);
221     }
222     else {
223         sort(results.begin(), results.end(), p2_order);
224     }
225 
226     dp_states[state] = results.size() == 0 ? ii(-1, -1) : results.front
227 
228     return dp_states[state];
229 }
```

```
230
231 bool is_pawns_stair(struct Game& game, struct State& state)
232 {
233     for (int i = 0; i < game.num_cores; i++) {
234         if (state.escada(i) == 0) {
235             return false;
236         }
237     }
238
239     return true;
240 }
241
242 ii calculate_score(struct Game& game, struct State& state)
243 {
244     vector<short> score(game.num_jogadores, 0);
245     for (int j = 0; j < game.num_jogadores; j++) {
246         for (int disc = 0; disc < game.num_cores; disc++) {
247             if (state.escada(disc) != 0) {
248                 score[j] += state.jogador(j,disc)*(game.num_cores - sta
249             }
250         }
251     }
252
253     return ii(score[0],score[1]);
254 }
255
256 short max_in_escada(struct Game& game, struct State& state)
257 {
258     short highest = 0;
259
260     for (size_t i = 0; i < game.num_cores; i++) {
261         highest = max(highest, (short) state.escada(i));
262     }
263
264     return highest;
265 }
266
267 //void print_game(ostream& out, struct Game game, struct State& state)
268 //{
```

```
269 //   out << state.jogador_atual+1 << " - ";
270 //   if (state.peao[0] && state.peao[0] <= game.num_discos) {
271 //       game.board[game.color_index[0][state.peao[0]-1]] = 'R';
272 //   }
273 //   if (state.peao[1] && state.peao[1] <= game.num_discos) {
274 //       game.board[game.color_index[1][state.peao[1]-1]] = 'G';
275 //   }
276 //   out << game.board << "  (";
277 //   for (short c = 0; c < game.num_cores; c++) {
278 //       if (c) out << ",";
279 //       out << state.escada[c];
280 //   }
281 //   out << ")  ";
282 //   for (short j = 0; j < game.num_jogadores; j++) {
283 //       if (j) out << "  ";
284 //       out << j+1 << ": ";
285 //       for (short c = 0; c < game.num_cores; c++) {
286 //           if (c) out << ",";
287 //           out << state.jogadores[j][c];
288 //       }
289 //   }
290 //
291 //   return;
292 //}
```

- 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.

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

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⟩.

SARTINI, B. A. et al. *Uma Introdução a Teoria dos Jogos*. 2004.

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

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