COMMUNICATIONS AND INNOVATIONS GAZETTE

ESTUDO DE IA BASEADO EM PROJETOS: IMPLEMENTANDO UMA *ENGINE* PARA JOGOS DE TABULEIRO

Thiago Lermen[1], **Eduardo Fantini**[1], **Érika Cota**[1]

[1]PET Computação – Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970
Porto Alegre – RS – Brazil

Abstract. *Two-player board games are widely studied in the history of artificial intelligence. Furthermore, learning logical board games always had a great impact on the intellectual development of children, youth and adults. Thus, with the development of technology, studies relating education and logical games have been developed. In this article, we generalize two different approaches to implementing a decision-making center in logical board games - the MiniMax and Q-learning algorithms -, aimed at the Tapatan game. Having both approaches, it was possible to compare their implementations and identify advantages and disadvantages.*

Resumo. *Jogos de tabuleiros de dois jogadores são amplamente estudados na história da inteligência artificial. Além disso, o aprendizado de jogos lógicos de tabuleiros sempre teve grande impacto no desenvolvimento intelectual de crianças, jovens e adultos. Desse modo, com o desenvolvimento da tecnologia, estudos que relacionam educação e jogos lógicos vêm sendo desenvolvidos. Neste artigo, generalizamos duas diferentes abordagens de implementação de centro de tomada de decisões em jogos lógicos de tabuleiro - os algoritmos MiniMax e Q-learning -, voltados para o jogo Tapatan. Tendo ambas as abordagens, foi possível comparar suas implementações e identificar vantagens e desvantagens.*

1. Introdução

O projeto de extensão LoBoGames[1], desenvolvido na Universidade Federal Do Rio Grande Do Sul (UFRGS), utiliza jogos lógicos de tabuleiro (Damas, Trilha, Xadrez, entre outros) como instrumento pedagógico a fim de potencializar o aprendizado e o exercício da lógica de alunos e professores de escolas de ensino fundamental, médio e em universidades. O objetivo do projeto LoBoGames é "promover a divulgação, a disseminação, o interesse pelos Jogos Lógicos de Tabuleiro (ou Jogos Abstratos de Estratégia) e seu aprendizado, trazendo como principal benefício o exercício do raciocínio lógico"[2].

Cerca de cem jogos diferentes são utilizados neste projeto, alguns retirados de livros e pesquisas, outros adaptados de acordo com o nível de complexidade exigido pela dinâmica. Os jogos utilizados no projeto LoBoGames são agrupados em diferentes módulos de acordo com o seu princípio de funcionamento, conforme apresentado na Tabela 1.

Módulos	Princípio Básico	Exemplos
Jogos de bloqueio	Bloquear as peças adversárias	Pong-hau K'i, Madelinette, Mu Torere, Amazonas, Rastro
Jogos de alinhamento	Movimentar as peças para alinhá-las	Picaria, Tapatan, Shisima, Three Men's Morris,Moinho, Tic Tackle, Tsoro, Yematatu, Dara, Tonkim
Jogos de deslocamento	Deslocar as peças para um lugar específico	Halma, Xadrez Chinês, Tábula, Gamão, Mancalas
Jogos de posicionamento	Posicionar estrategicamente as peças que não serão mais movimentadas	Jogo-da-velha, Reversi, Go, Gomoku, Quina, 4-em-linha
Jogos de captura	Capturar as peças adversárias	Alquerque, Felli, Luta-das-serpentes, Pretwa, Damas, Fanorona, Surakarka, Yoté
Jogos de caça	Um jogador persegue as peças do outro	Lebre e cachorros, Leopardo e caçadores, Tablut, Assaltom, Lobo e cabras, Jogo da onça

Tabela 1. Módulos de aprendizagem propostos pelo projeto LoBo Games

Considerando os resultados positivos do LoBoGames e buscando aumentar o seu alcance, a coordenação do mesmo iniciou um processo de expansão do projeto para o meio digital e, neste processo, estabeleceu uma parceria com o PET Computação para o desenvolvimento de um "jogador digital", ou seja, de uma *engine* que possa interagir com um jogador real tanto em modo aprendizado quanto em modo jogo. Iniciou-se assim o projeto Lobo Brain, que tem como objetivo implementar o centro de tomada de decisões dos jogos de tabuleiro do projeto LoBoGames. Através disso, um jogador poderá jogar sem a necessidade da disponibilidade de um outro jogador.

O projeto Lobo Brain iniciou em 2017 e sofreu diversas adaptações à medida em que evoluiu. A demanda de desenvolvimento inicial veio acompanhada de tentativas e aprendizados que já existiam no projeto LoBoGames. No entanto, ao longo do desenvolvimento da *engine*, novos desafios surgiram e foram moldando o escopo da solução.

[1]LOBO GAMES, Jogos Lógicos de Tabuleiro. `http://www.inf.ufrgs.br/lobogames/index.php?option=com_content&view=article&id=83&Itemid=472` acesso em 05/05/2020

[2]LOBO GAMES, Projeto Lobogames. `https://www.inf.ufrgs.br/lobogames/index.php?option=com_content&view=article&id=83&Itemid=472#projeto`, acesso em 05/05/2020

Assim, percebemos que a proposta inicial de desenvolvimento de uma *engine* genérica que funcionasse para (pelo menos) um grupo de jogos que compartilhassem um tabuleiro não seria factível como primeira abordagem de desenvolvimento devido à sua complexidade. Da mesma forma, percebemos que o objetivo de desenvolver conjuntamente a *engine* e uma interface de jogo adequada exigiria mais recursos do que o que o grupo tinha disponível no momento. Assim, optamos por um desenvolvimento incremental. Decidimos priorizar o desenvolvimento de uma *engine* básica (porém desenvolvida de forma a permitir uma fácil evolução) para validação das tecnologias, aprendizado e validação da solução junto ao público-alvo. Assim, em seu formato atual, o Lobo Brain é uma *engine* que implementa um centro de tomada de decisões dos jogos lógicos Pong Hau-Ki e Tapatan.

Há, na literatura, diferentes técnicas voltadas para implementar a lógica de tomada de decisão em jogos. Neste artigo, descrevemos e comparamos duas dessas abordagens e suas respectivas implementações no contexto do projeto Lobo Brain.

São utilizadas diferentes formas de implementação, diferindo em suas estruturas de dados e algoritmos. Nas abordagens utilizadas o desenvolvimento dos métodos foram pensados a fim de possibilitar a expansibilidade da implementação das mesmas técnicas para diferentes tipos de tabuleiros com diferentes regras de jogo. Os algoritmos são implementados na linguagem de programação C++ puro (sem utilização de bibliotecas auxiliares), para que as técnicas de tomada de decisão possam ser implementadas na sua essência, permitindo, assim, que o grupo de alunos obtivessem o máximo aprendizado dos métodos utilizados.

Dessa forma, para o PET, o Lobo Brain se constitui em um meio de aprendizado baseado em projetos. O grupo desenvolveu competências em algoritmos de aprendizado de máquina através do desenvolvimento de uma solução para uma demanda específica.

Este artigo está organizado da seguinte forma: na Seção 2 definimos a estrutura computacional de um jogo de tabuleiro e as abstrações necessárias para a implementação das *engines*, na Seção 3 definimos os conceitos teóricos fundamentais de ambas abordagens, na Seção 4 descrevemos as estruturas básicas utilizadas voltadas para o universo dos jogos de tabuleiro, nas Seções 5 e 6 descrevemos como foram implementados os métodos utilizados e as Seções 7 e 8 detalhamos os resultados gerados baseando-se nas comparações realizadas das implementações e as considerações finais do projeto.

2. Jogos Lógicos: estrutura computacional de um jogo de tabuleiro

No Tapatan, o jogo inicia com seis peças (três para o jogador 1 e três para o jogador 2) posicionadas nas duas extremidades do tabuleiro, de forma que fiquem intercaladas, como mostrado na Figura 1. Com jogadas alternadas entre os jogadores, o objetivo é colocar três peças em linha (horizontal, vertical ou diagonal), sabendo que cada peça pode ser movimentada para a esquerda, direita, cima, baixo e diagonal quando há linha indicando tais possibilidades.

Um jogo de tabuleiro pode ser demorado, principalmente quando os jogadores são cuidadosos nas suas decisões. Nas oficinas do LoBo Games é definida uma condição de empate para manter a atividade interessante para os jogadores. Para o Tapatan, definiu-se que o jogo pode terminar empatado quando ocorrer de as peças voltarem para a posição inicial do tabuleiro três vezes consecutivas.

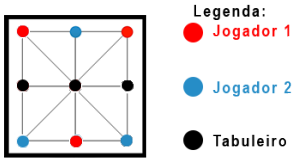


Figura 1. Representação da estrutura inicial do tabuleiro do jogo Tapatan.

2.1. Estrutura computacional

Para implementar uma versão digital do jogo de tabuleiro, é preciso modelar sua estrutura e comportamento para que sejam representadas computacionalmente. A estrutura de jogos como Xadrez, Go, Jogo-Da-Velha e Tapatan pode ser modelada a partir de dois elementos básicos:

1. O estado atual do tabuleiro: configuração das peças no momento;
2. Informação de quem irá se mover: jogador 1 ou jogador 2, peças azuis ou vermelhas, etc.

O comportamento das peças é definido pelas regras do jogo em questão e consiste em:

1. conjunto de regras de movimentação: movimentos permitidos e proibidos;
2. conjunto de regras de posicionamento: determina se uma configuração de peças é válida;
3. condições de término do jogo: configurações dos tabuleiros que representam o fim de um jogo.

Com base na modelagem, podemos implementar algoritmos que simulem o comportamento de um jogador - que chamaremos daqui pra frente de **agente**.

3. Fundamentação teórica

Implementações de jogos lógicos descritos na literatura, tipicamente são baseadas em Inteligência Artificial (IA). Nesta seção, revisamos os conceitos essenciais de funcionamentos dos métodos de IA que são usados neste trabalho. Foram consideradas duas abordagens distintas: *MiniMax* - que explora todas as jogadas para encontrar a melhor -, e *Q-learning* - que busca aprender a melhor jogada para cada situação, analisando o jogo a partir de um algoritmo de treino baseado em *Reinforcement Learning* (RL). As principais diferenças entre os métodos considerados são mostradas na Tabela 2.

	MiniMax	***Q-learning***
Forma de operação	Executa internamente todas jogadas possíveis - atuais e seguintes - e seleciona a melhor opção encontrada	Escolhe qual jogada tomar seguindo uma tabela de avaliação aprendida nos jogos de treino
Processamento de dados	Gera e percorre uma árvore de estados	Gerencia uma tabela de dados interna
Consumo de memória	Exponencial (em função da profundidade máxima da árvore de estados)	Linear (tabela de tamanho fixo)

Tabela 2. Tabela de diferenças entre as duas abordagens consideradas.

3.1. MiniMax

Fruto de um dos primeiros teoremas da Teoria dos Jogos e idealizado por John von Neumman em 1928, trata-se de um avaliador e decisor para jogos de soma zero. A ideia fundamental do algoritmo MiniMax é minimizar o prejuízo máximo. Para isso, dada a situação atual do tabuleiro, o algoritmo analisa **todas** as jogadas possíveis e seus desdobramentos mais à frente no jogo, e então avalia qual lhe trará o **menor prejuízo** na partida [Escandon and Campion 2018]. Dessa forma, o algoritmo atua - internamente - como sendo seu próprio adversário, joga contra si mesmo e decide qual é a melhor jogada para o momento.

3.1.1. Estruturas de dados

O algoritmo MiniMax executa sobre uma estrutura de dados do tipo árvore n-ária (com até n filhos por nodo) e com profundidade máxima p_{max} [Escandon and Campion 2018]. A profundidade da árvore pode ser definida como o número de níveis da árvore até os nodos folha. A Figura 2 representa a implementaçào de um tabuleiro do Tapatan de profundidade $p_{max} = 2$. De acordo com a figura, percebe-se que temos três tipos de níveis que podem ser definidos:

- O nível p_0 está relacionado a posição do tabuleiro pai da árvore, ou seja, o tabuleiro no qual a busca na árvore se iniciou.
- Os níveis intermediários p_i, onde $0 < i < max$ são os níveis que não possuem definições de jogo. Ou seja, onde não se pode concluir que o jogo chegou ao fim, ou não foi atingida a profundidade máxima da árvore. Na Figura 2, o nível de profundidade p_1 é o nível intermediário.
- O nível p_{max} é um parâmetro de entrada do algoritmo e determina a profundidade máxima que a árvore pode ter. Em outras palavras, a profundidade máxima limita a quantidade de recursões que o algoritmo executa dentro da função de busca no algoritmo MiniMax.

No nível p_{max} da árvore MiniMax existem dois tipos de estados de tabuleiro: os estados terminais inconclusivos (os que não possuem nenhuma definição clara de jogo, mas atingiram a profundidade máxima da árvore - como se pode perceber na figura em p_2) e os estados terminais conclusivos (os que possuem algum tipo de definição clara de jogo, como por exemplo, vitória, derrota ou empate).

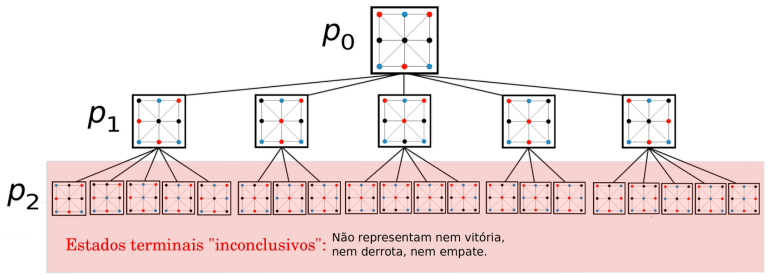


Figura 2. Representação de uma árvore de estados do algoritmo MiniMax aplicado ao jogo Tapatan, onde a profundidade máxima é alcançada sem encontrar estados terminais.

3.1.2. Algoritmo

Como mostrado no código abaixo, MiniMax é uma função recursiva que recebe como entradas um nodo *node* (na base da recursão - isto é, a primeira iteração - este nodo é a configuração do tabuleiro de entrada do algoritmo), a profundidade máxima p_{max} que a árvore pode atingir, um `booleano` ***maxJogador***, que define se a jogada é de maximização ou de minimização, e os valores α e β que serão utilizados a fim de otimizar o desempenho do algoritmo [Heineman and Pollice 2008].

```
1:  function MINIMAX(node, p_max, maxJogador, α, β)
2:      if p_atual = p_max ou node é terminal then
3:          return valoração de node
4:      end if
5:      if maxJogador then
6:          valor ← −∞
7:          for all filho de node do
8:              valor ← max(valor, MINIMAX(filho, p_atual + 1, FALSO, α, β))
9:              α ← max(α, valor)
10:             if α ≥ β then
11:                 break
12:             end if
13:             return valor
14:         end for
15:     else
16:         valor ← +∞
17:         for all filho de node do
18:             valor ← min(valor, MINIMAX(filho, p_atual + 1, VERDADEIRO, α, β))
19:             α ← min(β, valor)
20:             if β ≤ α then
21:                 break
22:             end if
23:             return valor
24:         end for
25:         return valor
26:
```

Ao longo da execução, o algoritmo alterna entre os papeis de agente (linhas 5-14) e adversário (linhas 16-24). Isso é feito alterando-se o valor do parâmetro $maxJogador$. Assim, na primeira iteração, $maxJogador$ é Verdadeiro, indicando que o algoritmo está agindo como um agente (jogador) e tem como objetivo minimizar a perda máxima. A iteração seguinte será iniciada por uma chamada recursiva com o parâmetro $maxJogador$ definido para Falso e o algoritmo estará simulando as possíveis jogadas do adversário, portanto privilegiará as configurações que maximizam as perdas. Na terceira iteração, o algoritmo avaliará as configurações novamente do ponto de vista do jogador ($maxJogador$ igual a verdadeiro) e assim por diante até atingir a condição de fim da recursão.

O fim da recursão é determinado por duas condições alternativas (linha 2): se o algoritmo atingiu a profundidade p_{max} definida ou se o nodo atual é terminal. Quando a condição de fim da recursão é atingida, o algoritmo (linha 3) avalia a configuração do tabuleiro informado naquela iteração (parâmetro node). Esta avaliação consiste em retornar a melhor heurística possível a fim de que o agente execute a melhor jogada.

Se a condição de parada não foi atingida, o algoritmo calcula a valoração de todas as configurações de tabuleiro derivadas da configuração atual (parâmetro node). Como explicado, quando em modo agente, o algoritmo identifica o ganho máximo possível a partir da configuração atual (linhas 6-8) e retorna este valor (linha 13). Quando em modo adversário, o algoritmo identifica o ganho mínimo possível a partir da configuração atual (linhas 16-18) e retorna este valor (linha 23).

3.2. Q-Learning

Q-Learning é um método de RL no qual o agente aprende na metodologia de ”tentativa e erro” por meio de interações com o ambiente (neste caso, o tabuleiro). Ao passo que o agente ”conhece” o ambiente, suas interações geram um retorno que chamamos de recompensas, que podem ser tanto positivas (que encorajam a repetição da ação) quanto negativas (que desencorajam a ação).

O fundo teórico desse algoritmo utiliza-se do Processo de Decisão de Markov (*Markov Decision Process* - MDP) [Sutton and Barto 2018], cujo nome remete ao cientista russo Prof. Andrei A. Markov (1856–1922). Tal processo será explicado adiante, mas, primeiramente, para o melhor entendimento desse, devemos abordar dois conceitos fundamentais relacionados: a Propriedade de Markov e a Cadeia de Markov.

- **Cadeia de Markov**: É um modelo estocástico, uma caminhada aleatória e sem memória sobre uma sequência de estados finitos, onde a probabilidade de transição entre estados segue a Propriedade de Markov, que será abordada a seguir. Vemos uma representação de uma Cadeia de Markov na Figura 3 .

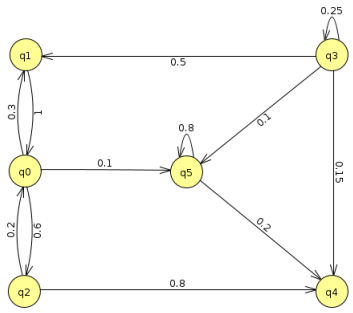


Figura 3. Exemplo de uma Cadeia de Markov

- **Propriedade de Markov**: Se refere à ausência de memória de uma cadeia de Markov, ou seja, a probabilidade de transição para um próximo estado do processo depende apenas do estado atual e não do histórico de estados. Em outras palavras, ela nos diz que o futuro da cadeia não depende do passado, mas sim do presente.

Modelando estes conceitos para o domínio de nosso problema, uma Cadeia de Markov pode ser vista como a transição de diversos estados do jogo. Assim, os nodos representam as configurações do tabuleiro, as arestas determinam a transição entre dois tabuleiros e seu valor é a probabilidade dessa transição ocorrer.

Através dessa modelagem, construímos um método de aprendizado com o objetivo de treinar o agente com base nos últimos movimentos realizados. No início de seu treinamento, o agente toma movimentos puramente aleatórios, que na maioria dos casos o guiará a derrotas. Por esse motivo, a tendência é de obter derrotas em quase 100% de seus jogos iniciais. Com a aplicação de sequências de treinos, o agente aprende quais movimentações geram maior retorno positivo e tende a reproduzi-las, aproximando-se da solução ideal.

A inteligência desse método está, principalmente, em uma matriz chamada de *Q-Table*, que servirá como a memória do agente. Essa matriz relaciona dois estados (representados pela letra q e seus respectivos índices na matriz) a um valor numérico (chamado

de *q-value*), que é usado para representar a recompensa esperada ao se transicionar entre estes dois estados. Quanto maior for esse valor, melhor é aquela jogada na visão do agente.

Contudo, se fosse guiado apenas por conhecimentos prévios armazenados na *Q-Table*, teríamos um caso onde o agente segue sempre apenas um caminho que, por mais que dê resultados positivos, pode estar longe de ser o ideal (similar ao conceito de sobreajuste (*overfitting*)). Assim, introduz-se a ideia de tipos de ação, conceito essencial para um bom funcionamento do algoritmo. Os dois tipos de ações definidos são os seguintes:

- *Exploration*: Ação tomada de forma puramente aleatória, com o intuito de encontrar novas e melhores ações para o estado atual da máquina. Isto é importante por possibilitar ao agente descobrir novas estratégias, podendo assim expandir seu leque de opções.
- *Exploitation*: Ação que utiliza a tabela aprendida como base de tomada de decisão. Busca o maior valor armazenado na *Q-Table* que parte do estado onde a máquina se encontra.

Para a construção do algoritmo, utiliza-se uma equação que atualiza o *q-value* de acordo com as recompensas recebidas pelo agente. A chamada equação de Bellman [Watkins and Dayan 1992] é uma iteração sobre os valores que desejam ser atualizados de acordo com uma transição de estados. Através dessa equação, podemos determinar se o agente tomará decisões de *exploratrion* ou *exploitation*, verificando, a cada passo, se a decisão tomada é boa ou ruim.

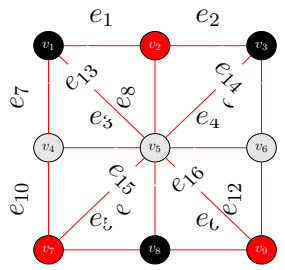
Dado um número suficientemente grande de iterações, o agente chega em uma *Q-Table* que aproxima-se da tabela ótima, o que significa que sempre que uma ação do tipo *exploitation* é tomada, a jogada escolhida tende a ser a melhor possível. Após um certo número de iterações, deixamos que o agente passe a tomar menos ações aleatórias e mais ações baseadas na matriz, a fim de melhorar a performance do mesmo. Entretanto, tal decisão precisa ser tomada com cuidado, pois quanto menos liberdade para explorar o ambiente, menor a chance de novas estratégias e aprendizados serem descobertos pelo agente. Aplicando esse método diversas vezes (treinamento), eventualmente chegaremos a uma solução ótima para o problema, onde os valores de *q-value* estão devidamente ajustados.

4. Abordagem proposta

Nesta seção apresentamos a modelagem estrutural junto com o conjunto de regras propostas neste projeto para o jogo Tapatan. Para este jogo, os dois jogadores serão representados pelas cores de suas peças, ou seja, jogador 1 será representado pelas peças de cor preta e o jogador 2 pelas peças de cor vermelha. As regras do jogo são as descritas na Seção 2.

Antes de pensar na implementação da lógica dos algoritmos, precisamos definir o ambiente onde o jogo ocorrerá, isto é, o tabuleiro. Tendo em vista que estes são formados por "casas" e "caminhos", podemos representá-los de diversas formas. As implementadas nesse projeto foram duas: grafo (que pode ser generalizado para tabuleiros $N \cdot N$) e uma matriz de `char` (de tamanho $3 \cdot 3$ no caso do Tapatan), onde os possíveis caminhos são definidos a partir das regras de cada jogo. Contudo, a implementação base utilizada foi através de matriz de `char`, logo as ideias serão todas modeladas a partir dessa implementação.

Cada “casa” representa uma posição que pode ser ocupada pelo jogador 1 (representado em preto), 2 (representado em vermelho), ou que pode estar vazia (representada em cinza). Tais estruturas são denotadas nas Figuras 4a e 4b, ambas descrevendo o jogo Tapatan em seu estado inicial:



(a) Tabuleiro representado por um grafo

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ 2 & 1 & 2 \end{bmatrix}$$

(b) Tabuleiro representado por uma matriz

Percebe-se, através da Figura 4b que o tabuleiro representado em uma matriz do tipo `char` possui três linhas e três colunas. Na linha 0, tem-se três caracteres, cada um em uma coluna distinta os quais representam os jogadores acima descritos. Na linha 1, tem-se três caracteres dispostos da mesma forma os quais representam posições vazias no tabuleiro. E na linha 2, da mesma forma que a linha 0, tem-se três caracteres que representam os jogadores.

A Figura 4a refere-se a uma representação de tabuleiro em um grafo de 9 vértices e 16 arestas. Essa estrutura é uma possibilidade de abordagem genérica, mas não será detalhada pois foge do escopo deste artigo. As diferenças da representação em grafo e matriz ocorrem exclusivamente em funções intermediárias que avaliam o tabuleiro, como a que encontra um valor heurístico em determinado estado e a que verifica possíveis movimentações.

Assim, caso queiramos realizar alguma movimentação ou verificação da regra do jogo, basta verificar se a movimentação que o jogador deseja realizar é válida para determinado conjunto de regras de jogo. Nas próximas seções serão vistas as formas de implementação e modelagem do nosso universo utilizando a representação de tabuleiro vista na Figura 4b.

5. Implementação da *engine* através de MiniMax

Um dos principais problemas enfrentados pelo grupo ao implementar o algoritmo MiniMax foi o crescente consumo de memória ao construir a árvore de estados e o tempo de processamento da busca. Neste quesito, enfrenta-se uma questão de **eficiência do agente *vs* uso de recursos**. Isso se dá pelo fato de que o número de nodos cresce de maneira exponencial - devido ao caráter n-*ário* da árvore - enquanto a profundidade cresce linearmente. Por isso, precisa-se encontrar uma profundidade que entregue um bom balanço entre os dois pontos citados.

Outra dificuldade encontrada é que quando um nodo da árvore é folha por ter profundidade máxima, podemos considerar que grande parte do processamento daquele caminho da árvore foi inútil, pois retira-se pouca informação deste. Isso ocorre por a avaliação feita sobre os tabuleiros ser heurística. Isso acaba por não nos permitir avaliar tabuleiros intermediários eficientemente, pois olhamos para cada estado da árvore por

uma perspectiva extremamente simplista onde não levamos em conta a organização das peças em si, apenas a qualidade de fim de jogo.

5.0.1. Valoração de tabuleiros

O algoritmo funciona percorrendo a árvore a partir de chamadas recursivas da função *minimax*, vistas na Seção 3.1, e fazendo uma valoração de todos os nodos percorridos, para descobrir a qual subgrupo eles pertencem. Tal valoração é representada consiste em analisar se o tabuleiro representa o final de um jogo. Por exemplo, um tabuleiro onde há 3 peças do jogador 1 alinhadas representa uma vitória do jogador 1. Este tipo de estado recebe o nome de **estado terminal** e é visto como nodo folha da árvore. Assim, quando encontramos um terminal damos fim a recursão naquele caminho da árvore de estados.

Para representar esses casos, usamos valores heurísticos, cujo objetivo é dizer de forma simples o quão boa aquela configuração do tabuleiro é para o agente. Os valores escolhidos por esta implementação foram os seguintes: 100 que representa uma vitória do agente, 0 que representa um empate, -1 que representa um estado final inconclusivo e -100 que representa uma derrota do agente.

A fim de controlar a dificuldade do jogo, podemos subtrair a profundidade em que a árvore se encontra no momento da busca pelos valores definidos de representação de derrota ou empate. Com isso, quanto maiores os valores retornados, mais rápido a máquina irá vencer, enquanto para valores menores - maior profundidade de busca -, mais demorada será a vitória. Essa abordagem possibilita a escolha da jogada ótima a fim de otimizar o tempo de jogo. A Figura 5 exemplifica a estrutura da árvore e um caso do processo de valoração heurística:

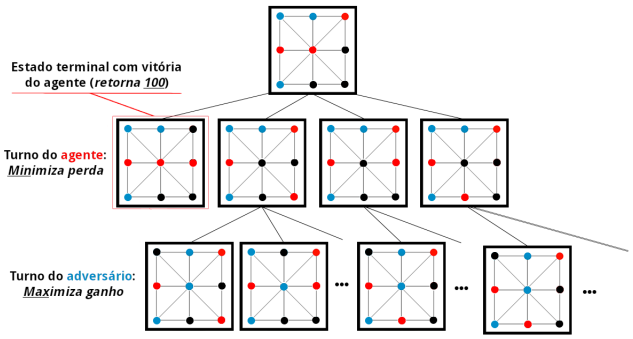


Figura 5. Representação do algoritmo MiniMax aplicado na árvore de estados do jogo Tapatan.

6. Implementação da *engine* através de Q-Learning

A implementação desse método tem seu centro de tomada de decisões baseado em uma tabela chamada *Q-Table*, cujas linhas e colunas representam configurações de tabuleiros (posições das peças). E os espaços da tabela contêm a valoração da transição de um para o outro. É construída a partir de um algoritmo de treino baseado em *exploration* e *exploitation*, como foi descrito anteriormente na Seção 3.2.

Para a construção da tabela, foram utilizadas uma codificação das posições das peças no tabuleiro e um modelo de probabilidade MDP para atualização do *rating* baseado em recompensas negativas ou positivas, como já dito. A codificação de tabuleiros, basicamente, transforma a transição de um tabuleiro $t_1 = a$ para um tabuleiro $t_2 = b$ em uma palavra w pertencente ao alfabeto $\Sigma = (0, 1, 2)$, cujo comprimento $|w| = 18$. Sendo assim, temos que '0' representa uma posição do tabuleiro não ocupada, '1' representa uma posição ocupada pelo jogador 1 e '2' uma posição ocupada pelo jogador 2. Por exemplo, se tivermos o seguinte tabuleiro apresentado na Figura 6:

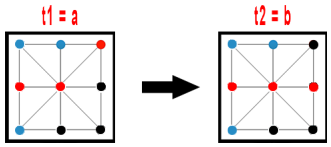


Figura 6. Representação de uma transição de estado do tabuleiro Tapatan

Podemos mapear este tabuleiro para uma *string* de conteúdo "221110200220111200" que representa a transição do estado $t_1 = a$ para o estado $t_2 = b$. Com isso, através do modelo probabilístico MDP podemos atribuir valores para cada um desses estados baseados na efetividade da jogada, ou seja, a probabilidade do jogador vencer, perder ou empatar o jogo a partir de um estado t_1.

Dessa forma, durante o algoritmo de treino, realizamos diversas consultas nessa tabela a fim de realizarmos uma valoração para cada transição de estado baseando-se em métodos de *exploration* e *exploitation*, nas quais jogadas do tipo *exploration* podem ou não descobrir um novo estado de transição - o qual será incluído na tabela -, enquanto jogadas do tipo *exploitation* serão sempre baseadas na tabela, a fim de realizar a melhor jogada corrente.

Esse tipo de abordagem possibilita o maior controle sobre os estados passados, validando as transições de estado de acordo com a experiência do agente. Além disso, após treinado, esse modelo realiza as consultas de forma direta na *Q-table* sem a necessidade de realizar nenhum tipo de busca, o que torna este método escalável para jogos de maior complexidade.

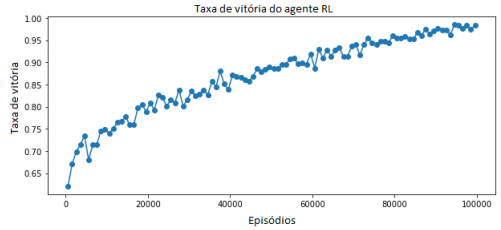
7. Resultados experimentais

O projeto LoBo Brain possibilita examinar a aplicabilidade dos dois métodos de IA utilizados - *Minimax* e *RL* - para implementação do centro de tomada de decisões dos jogos de tabuleiros anteriormente citados. Primeiramente, através dos algoritmos implementados na linguagem de programação C++ podemos realizar comparações envolvendo aplicabilidade e eficiência para jogos lógicos de tabuleiro de baixa complexidade, como Tapatan. Os resultados de taxa de vitória e os parâmetros utilizados são puramente experimentais.

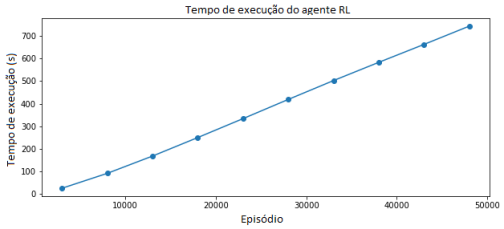
Em primeira análise, podemos comparar a aplicabilidade das duas implementações através da taxa de vitória do agente quando ele joga contra um outro agente que toma decisões puramente aleatórias. Na Figura 7a estão representadas as taxas de vitória de cada um dos agentes de acordo com os parâmetros dados como entrada. No caso do agente RL, calculamos a média a cada dois episódios de treino de um número total de 100.000 episódios, de acordo com os seguintes parâmetros: $\alpha = 0.2$,

$\gamma = 0.7$, $\epsilon = 0.3$ com decaimento. No caso do agente Minimax, calculamos a taxa de vitória de 1.000 episódios de treino para uma dada profundidade máxima de árvore p_{max}

Além disso, podemos comparar o tempo de execução das duas implementações, executando os procedimentos em uma máquina Intel(R) Core(TM) i5-4440 CPU @ 3.10GHz, 8 GB RAM e Windows 10 Home. Na Figura 7b estão representados os tempos de execução de cada um dos agentes. No caso do agente RL, ao calcular o tempo de execução para treinar o agente por 50.000 rounds, percebe-se que o crescimento do tempo é linear no período de treino, porém é constante no período de busca, devido ao fato de realizar a busca pela política ótima em uma tabela *hash*. No agente MiniMax, ao calcular o tempo de execução para realizar a busca pela melhor jogada para cada nível de profundidade máxima, percebe-se que o tempo cresce de forma exponencial, devido à complexidade do algoritmo MiniMax com poda alfa-beta ser da ordem $O(b^{d/2})$, onde b é o fator de ramificação da árvore e d ser a profundidade máxima definida.



(a) Comparação das taxas de vitória ao longo do treino do agente RL e do agente Minimax.



(b) Comparação dos tempos de execução do treino do agente RL e da execução da Minimax

Quando executamos o algoritmo MiniMax, é perceptível que o tempo de resposta para jogos de baixa complexidade é maior. Todavia, o consumo de memória é extremamente alto e está diretamente relacionado com a profundidade máxima atribuída . Esse algoritmo roda em tempo real, sem necessidade de treinamento prévio da máquina, o que possibilita com que a eficiência para determinados tipos de jogos simples seja alta. De outra forma, a aplicabilidade deste algoritmo é baixa, pois pode ser utilizado de forma eficiente somente em jogos de baixa complexidade, porque, caso contrário, a utilização de memória para a execução do algoritmo em tempo real é muito alta.

Por outro lado, quando executamos o processamento do método de aprendizado de RL, precisamos treinar o agente de uma forma que ele possa satisfazer a dificuldade desejada. Em outras palavras, quanto maior a quantidade de treino recebido pelo agente, melhor serão as suas decisões e, portando, maior será a dificuldade do jogo. Sendo assim, uma vez treinado, o algoritmo pode ser aplicado em situações reais de jogo, apenas baseando-se nas valorações armazenadas na *Q-Table*. Consequentemente, percebe-se que o método de RL implementado possui uma maior aplicabilidade devido ao fato de podermos armazenar valorações de transições de estados de jogo em uma tabela simples apenas usando funções *hash*, o que potencializa a eficiência e o uso de memória e torna o método

de aplicação mais genérico, variando, de acordo com os jogos, somente suas regras base e de movimentação.

Dessa forma, podemos concluir que, para jogos de maior complexidade a utilização do método de RL é mais eficaz pelo fato da utilização de uma tabela que guarda as transições de estados de jogo, enquanto que o algoritmo MiniMax utiliza uma grande parte da memória para armazenamento da árvore de estados, além do tempo de execução crescer de forma exponencial de acordo com o fator de ramificação e profundidade máxima de busca. Sendo assim, uma vez treinado, o algoritmo de *Q-Learning* possibilita um maior controle sobre a tomada de decisões do agente, além de ocupar uma quantidade significativamente menor de memória.

8. Considerações finais

Com o desenvolvimento do projeto Lobo Brain, podemos destacar diversos aprendizados relacionados aos métodos de IA abordados, como por exemplo eficiência e aplicabilidade dos algoritmos. Além disso, o conhecimento adquirido a partir do estudo da fundamentação teórica de cada um dos métodos serve como base para a expansão do escopo de aplicabilidade, podendo, assim, serem implementados em outros ramos relacionados à Teoria Dos Jogos.

Dessa forma, tendo a inteligência dos jogos implementada, o grupo tem como objetivo iniciar a construção de uma interface gráfica de usuário para que os jogos de tabuleiro do projeto LoBoGames possam ser utilizados em aparelhos móveis, como *spartphones*, ou *desktops*.

Referências

Escandon, E. R. and Campion, J. (2018). Minimax checkers playing gui: A foundation for ai applications. In *2018 IEEE XXV International Conference on Electronics, Electrical Engineering and Computing (INTERCON)*, pages 1–4.

Heineman, G. T. and Pollice, G. (2008). *Algorithms in a Nutshell*. Shroff Publishers & Distributors Pvt Ltd.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press, second edition.

Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3-4):279–292.

Russell, Stuart J.; Norvig, Peter (2010). Artificial Intelligence: A Modern Approach (3rd ed.).