UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

JORDANO SARDI SCHÜTZ

Uma Aplicação de Inteligência Artificial Minimax para Jogos de Tabuleiro Abstratos e de Estratégia

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Renato Perez Ribas

Porto Alegre
Dezembro 2020

Ao meu amigo Renato, minha incansável fonte de inspiração e apoio.
À minha avó, inseparável companheira de tantas tardes de domingo.
Ao meu pai, único sorriso que nesse momento terei apenas pelas lembranças.
Ao meu irmão, meu melhor amigo e meu eterno herói.
À minha mãe, a fortaleza que sempre me protegeu e me fez quem eu sou hoje.
E à M, luz dos meus olhos, o grande amor que esperei por toda minha vida.

RESUMO

Este trabalho é uma proposta para o desenvolvimento de um sistema de jogos de tabuleiro abstratos e de estratégia, para os quais há uma Inteligência Artificial Minimax apta a personificar um dos jogadores. Com esse fito, é de particular interesse a problemática intrínseca à representação computacional da dita classe de jogos, em que se destacam as abstrações dos elementos fundamentais comuns a esses jogos, as estruturas de dados aplicáveis na transposição computacional e como diferentes categorias de jogos de tabuleiro compartilham conceitos quando inseridos no processo arquitetural de uma plataforma digital de jogos. Flexibilidade, portanto, é requisito essencial. Ao mesmo tempo, o empenho da digitalização dos jogos abstratos e de estratégia deve se manter coeso com o componente autônomo do sistema: a Inteligência Artificial, aqui conduzida pelo expoente máximo dos Algoritmos de Decisão Competitiva, o Minimax. A sinergia entre um intento e outro é fundamental, e por si só implica em inúmeros desafios pertinentes tanto a arquitetura do sistema quanto ao poder de decisão da Inteligência Artificial. A proposta do sistema que adere os jogos de tabuleiro ao Minimax é seguida de uma validação prática, a qual objetiva asseverar a factibilidade das resoluções tomadas no campo teórico a partir de um conjunto reduzido de jogos abstratos e de estratégia, criteriosamente selecionados a partir de aspectos que provoquem a evolução do sistema. Colhido o saldo da implementação, é conduzida uma análise crítica a respeito da escalabilidade da proposta, firmada em relevantes pormenores até então intocados, porém presentes em diferentes jogos de tabuleiro.

Palavras-chave: Jogos de tabuleiro. decisão competitiva. inteligência artificial. minimax.

Applying Minimax for Abstract and Strategic Board Games

ABSTRACT

This paper is a proposal for the development of an abstract and strategic board games system, for which there is an Artificial Intelligence able to personify one of the players. For this purpose, it is of particular interest the intrinsic problem relative to the computational representation of such class of games, in which are highlighted the abstractions of the fundamental elements that are common to these games, the data structures applicable in the computational transposition and how different categories of board games share concepts when inserted into the architectural process of a digital game's platform. Flexibility, therefore, is an essential requirement. At the same time, the effort in digitalizing the abstract and strategic board games must be cohesive with the autonomous component of the system: The Artificial Intelligence, here conducted by the Adversarial Search Algorithms' maximum representative, the Minimax. The synergy between one intent and the other is fundamental, and by itself implies countless challenges relevant both for the system's architecture and the Artificial Intelligence's decision power. The system's proposal which adheres the board games to the Minimax is followed by a practical validation, which objectives to assert the feasibility of the resolutions adopted in the theoretical field based on a reduced set of abstract and strategic board games, discerningly selected from aspects that provoke the evolution of the system. Once the implementation's settlement is collected, a critical analysis is conducted regarding the proposal's scalability, set on relevant details until then untouched, nonetheless present in different board games.

Keywords: Board games. Adversarial search. Artificial Intelligence. Minimax.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

OO Orientação a Objetos

IA Inteligência Artificial

SUMÁRIO

1 INTRODUÇÃO

1.1 Motivação

A computação nos ajuda e automatizar os mais diferentes processos segundo lógicas programadas. Nos últimos anos, a informática evoluiu para que essas lógicas programadas deixem de tomar impreteríveis decisões codificadas para que sejam trilhados caminhos inicialmente desconhecidos. Esse arroubo exploratório, dentre outras valorosas contribuições, instigou fugazmente o protagonismo da Inteligência Artificial.

Se a noção de que uma Inteligência Artificial pode seguir caminhos desconhecidos é talvez excessiva, ao menos pode-se conceder à IA o poder da resolução sem vacilos. Tal autonomia é especialmente relevante no contexto de disputas, no compasso observado em condições competitivas que demandam a predição cautelosa, porém diligente e eficaz. E alguns dos mais férteis terrenos para o desenvolvimento tangível de uma inteligência beligerante são os jogos.

O universo dos jogos, por seu turno, abrange uma pletora de tipos. Para uma Inteligência Artificial competitiva, contudo, os jogos de tabuleiro demonstram um inegável apelo. Destes, enxergam-se duas grandes categorias: jogos contextualizados e jogos abstratos e de estratégia. A primeira classe agrupa os ditos jogos modernos, característicos pela temática subjacente, como os jogos WAR, Detetive e Banco Imobiliário. O segundo grupo é abstrato por excelência, pois a estratégia não demanda uma roupagem motivadora, e tem como expoentes, por exemplo, o Jogo-da-Velha, Damas e o Xadrez.

Este trabalho se debruça justamente sobre os jogos de tabuleiro abstratos e de estratégia. Incrivelmente, afora o Xadrez e outros jogos bem conhecidos, inexiste uma plataforma computacional única onde outros notáveis jogos de menor fama podem ser jogados. A lacuna percebida pelo cenário multifacetado dos jogos de tabuleiro abstratos torna-se ainda mais substancial quando se requisita a possibilidade do confronto entre um usuário humano e uma Inteligência Artificial.

Logo, dois grandes desafios motivam o desenvolvimento deste trabalho: a representação computacional de jogos de tabuleiro e uma Inteligência Artificial voltada a tomada de decisão competitiva. A união entre esses dois elementos sintetiza o *leitmotiv* do projeto de desenvolvimento de um sistema no qual podem ser encontrados diferentes e menos conhecidos jogos de tabuleiro, tendo como adversário o computador.

1.2 Objetivo

Um sistema que apresente uma quantidade razoável de jogos de tabuleiro, acrescido de uma Inteligência Artificial que se eleve ao nível de um jogador, é claramente alvissareiro. No entanto, não se pode negligenciar os meandros pelos quais tal solução se enreda, em especial se estabelecidos critérios de qualidade de software mais rigorosos. Essa premissa, pois, exige que os meios qualifiquem os fins, em vez de apenas justificá-los.

Nesse sentido, o sistema de jogos abstratos e de estratégia e a IA que o acompanha não são, por si, o objetivo deste trabalho. O propósito-chave é o processo pelo qual a construção desse ambiente passa, as decisões de tecnologia, de algoritmos, as abstrações necessárias para que a transposição tabuleiro-computador seja não só intuitiva, mas escalável. E cada decisão, cada compensação serão sempre vigiadas pela onipresença da Inteligência Artificial.

O contexto apresentado coloca a IA em uma posição consideravelmente desafiadora: a de requisito ubíquo. Essa extensa difusão implica que todos os jogos podem ser disputados com uma Inteligência Artificial, a qual é, de fato, desejo primeiro de qualquer nova incursão de jogo abstrato e de estratégia. Dessa forma, tem-se como constante o dilema de que nenhuma decisão de projeto será totalmente favorável a um jogo específico, e nem tampouco priorizará unicamente o desenvolvimento da Inteligência Artificial.

Logo, nenhuma escolha, teórica ou prática, deve passar despercebida ou imerecida de análise. As estruturas de dados, o paradigma de linguagem de programação, as particularidades dos algoritmos que cercam a Inteligência Artificial: nenhum aspecto relevante pode ser preterido pela conveniência da implementação ou o caminho mais curto que tanto alija o êxito de um sistema computacional.

Assim, o objetivo deste trabalho não é necessariamente apresentar um sistema de jogos abstratos e de estratégia acompanhado de uma IA, ao menos não no estrito sentido de produto. Interessa aqui as estruturas fundamentais que podem ser implementadas para que se chegue a tal produto, em especial de que forma a arquitetura do sistema pode ser escalável tanto para a configuração de novos jogos de tabuleiro quanto para a capacidade de adaptação da Inteligência Artificial a esses jogos.

Tendo o como ponto central o núcleo de um sistema de jogos de tabuleiro enriquecido por uma Inteligência Artificial, parece desmedido adentrar na camada de visualização desse sistema. Não é intenção, portanto, o estudo de formas de visualização e interação

de usuários com a plataforma de jogos de tabuleiro. Essa responsabilidade já é suficientemente grande e abordada com muito mais competência pelas áreas de Computação Gráfica e Interação Humano-Computador.

1.3 Estrutura

Os capítulos foram organizados em uma sequência que expressa a evolução da proposta do projeto. Afora a contextualização técnica, necessária para a total compreensão computacional do trabalho, os demais capítulos avançam em sintonia com o que efetivamente foi observado no desenvolvimento prático do projeto. Dessa forma, a evolução da proposta vai se refletindo ao longo do texto, sem sobressaltos e com ampla clareza.

O Capítulo 2 apresenta os conceitos técnicos da computação essenciais para o entendimento pleno da proposta. Essa parte busca posicionar as noções elementares empregadas na idealização e validação do trabalho, visto que se farão presentes ao longo de toda a exposição da proposta. Esse capítulo oferece não apenas a explanação teórica, mas também o reforço por exemplos de autoria própria e das referências às quais este trabalho recorreu.

A proposta do sistema de jogos de tabuleiro, da Inteligência Artificial e de como ambos se entrelaçam é detalhada no Capítulo 3. O intento dessa parte é expressar gradativamente a evolução das ideias que surgiram ao longo do desenvolvimento do trabalho, tendo, por isso, um início fortemente abstrato. Ao fim desse capítulo, entretanto, há maturidade suficiente para que a proposta da plataforma computacional de jogos com IA seja colocada em prática.

A partir das sustentáveis bases fornecidas no Capítulo 3, o trabalho se volta para a validação da proposta. Dado o caráter inerentemente prático do sistema de jogos, o Capítulo 4 fica sendo o responsável por transpor o campo teórico e efetivamente materializar uma solução computacional que se estabeleça sobre as estruturas fundamentais destrinchadas até então. Assim, o Capítulo 4 parte para a implementação de alguns jogos abstratos e de estratégia e da Inteligência Artificial Minimax.

De posse dos resultados observados no desenvolvimento prático do projeto, o Capítulo 5 retoma o foco teórico e passa a apresentar, criticamente, as melhores formas de estender o sistema validado no Capítulo 4. Afinal, tão importante quanto a proposta e o desenvolvimento é a capacidade de percepção do quanto aquilo que foi construído pode se adequar às demandas de novos conceitos introduzidos por outros jogos de tabuleiro.

Por fim, o Capítulo 6 faz os reconhecimentos necessários e encerra o texto.

2 FUNDAMENTOS TÉCNICOS

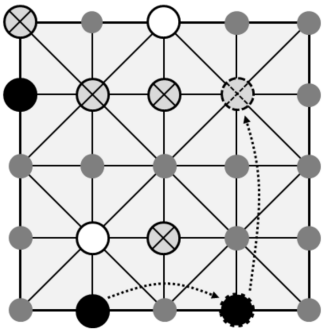
2.1 Jogos de tabuleiro abstratos e de estratégia

Os jogos de tabuleiro abstratos e de estratégia têm muitos elementos em comum. Em geral, as partidas por eles proporcionadas se desenrolam entre dois jogadores, que executam as jogadas alternadamente utilizando peças sobre um tabuleiro, estando limitados a um conjunto finito de movimentos que se submetem às chamadas regras de jogo. Cada jogador busca atingir um dos objetivos do jogo para alcançar a vitória.

Apesar de compartilharem elementos essenciais entre si, os jogos de tabuleiro podem ser classificados a partir dos alicerces que os distinguem uns dos outros: as regras de movimentação e os objetivos (critérios de fim de jogo). A mais tênue alteração nas regras ou no objetivo permite que um jogo se renove completamente, seja pela dinâmica da disputa, seja pela escalada gradativa do nível de dificuldade da partida em si, independente, portanto, da habilidade do oponente. Além disso, pequenas distinções nas regras e objetivos podem fazer jogos muito semelhantes pertencerem a categorias diferentes.

Uma classificação típica, muito bem detalhada por Ribas (2020), define seis grandes grupos para os jogos de tabuleiro abstratos e de estratégia. A composição dessas classes de jogos é feita com base em conceitos intuitivos: bloqueio, posicionamento, alinhamento, deslocamento, captura e caça. A chave para a inclusão de um jogo de tabuleiro em um desses grupos leva em consideração tanto as características que permeiam os movimentos dos jogadores quanto o objetivo de jogo, ou, em alguns casos, o conceito elementar que pode levar um jogador a vencer a partida.

Figura 2.1: Jogo Amazonas: deslocamento e lançamento de peça de bloqueio.



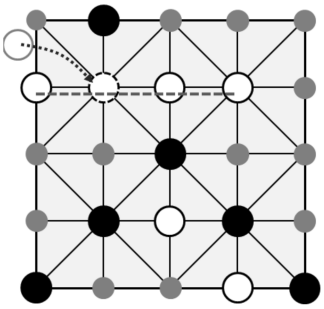
Fonte: Ribas (2020)

Em um jogo de bloqueio, cada jogador deve executar os movimentos com o intuito de imobilizar o adversário. As regras de jogo definem mecanismos que proporcionam o

anulamento de posições, restringindo cada vez mais as direções que uma peça pode tomar. Por exemplo, o jogo Amazonas determina que um jogador movimente a sua peça e, em seguida, lance uma peça de bloqueio sobre uma posição do tabuleiro, como mostra a Figura 2.1. Assim, se o jogador estiver sem movimentos possíveis, ele fica bloqueado e o jogo encerra com a vitória do adversário.

Os jogos de posicionamento se destacam não pelo objetivo do jogo, mas pela forma como os jogadores conduzem a partida. As peças, posicionados inicialmente fora do tabuleiro, são colocadas em posições livres. Entretanto, após o posicionamento de uma peça, esta se torna inamovível; assim, novas jogadas são conduzidas a partir de mais peças que são introduzidas no tabuleiro, em posições supostamente estratégicas. A primazia dos jogos de posicionamento reside, portanto, na inserção das peças, sendo que os objetivos de jogo podem ser os mais variados, como por exemplo o alinhamento de peças, ilustrado pelo Jogo-da-Velha 4 da Figura 2.2.

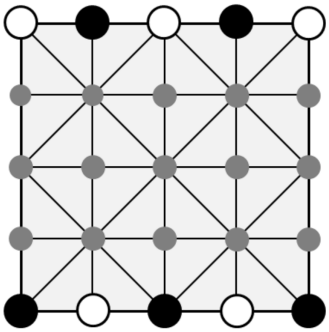
Figura 2.2: Jogo-da-Velha 4: posicionamento de peça com alinhamento.



Fonte: Ribas (2020)

Jogos de alinhamento, naturalmente, têm como critério de fim de jogo a disposição de peças sobre o tabuleiro de modo que estas estejam alinhadas. O Jogo-da-Velha 4, por exemplo, determina que um jogador chega a vitória somente se 4 peças estiverem alinhadas em linha, coluna ou em diagonal. Aqui cabe um adendo: embora seja um jogo de posicionamento, em que as peças são colocadas sobre o tabuleiro e não podem mais ser movimentadas, o Jogo-da-Velha 4 tem como objetivo o alinhamento. Entretanto, a dinâmica desse jogo se define muito mais pelo cuidado com que as peças são inseridas, com o fito de alinhá-las, do que pelo critério de vitória em si. O Tic Tackle é um representante típico de um jogo de alinhamento, pois as peças, já inseridas no tabuleiro, são movimentadas até que se decrete como vitorioso o jogador que alinhou 4 peças. Diferentemente do Jogo-da-Velha 4, o Tic Tackle requer uma estratégia de deslocamento de peças, em passos unitários, do início ao fim do jogo, tendo como posição inicial de partida um tabuleiro idêntico ao mostrado na Figura 2.3.

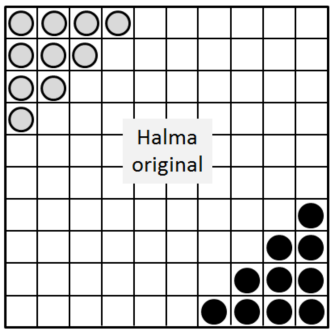
Figura 2.3: Tabuleiro inicial do Tic Tackle.



Fonte: Ribas (2020)

Outros jogos são cadenciados pelo propósito fundamental de se atingir uma determinada posição. Via de regra, cada jogador é o guardião de uma região do tabuleiro, e cada jogador deve executar movimentos que o façam atingir a posição de destino protegida pelo adversário. Alternativamente, ambos os jogadores podem ter como objetivo a mesma posição de destino; nesse caso, o jogo é uma corrida estratégica na qual o vencedor é o primeiro a chegar na posição alvo. Tais jogos são classificados como jogos de deslocamento, como, por exemplo, o Halma, visto em configuração para dois jogadores na Figura 2.4. Nesse jogo, cada participante deve transferir as peças a ele atribuídas de um canto a outro do tabuleiro.

Figura 2.4: Jogo de deslocamento Halma.

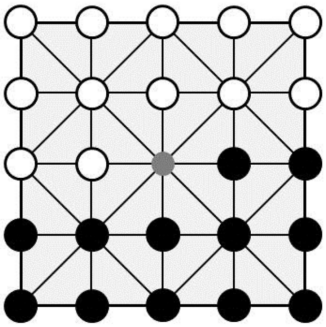


Fonte: Ribas (2020)

A captura é reservada aos jogos em que o movimento do jogador visa a eliminação das peças do adversário. O jogo Fetaix, por exemplo, dita que a captura de uma peça adversária é feita quando um jogador salta sobre a peça do adversário e recai sobre uma posição livre do tabuleiro. Nesse caso, a peça do oponente sobre a qual o salto foi realizado é custodiada pelo jogador que executou o movimento de salto. Jogos de captura se encerram quando todas as peças do adversário foram removidas do tabuleiro. A Figura 2.5 mostra o tabuleiro inicial do Fetaix, o qual apresenta inicialmente apenas o centro

livre para a movimentação de peça que inicia a partida.

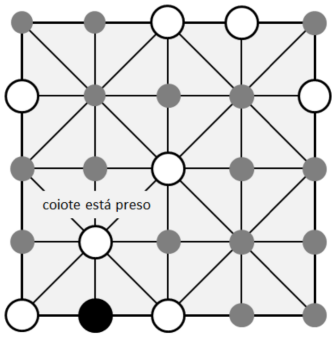
Figura 2.5: Tabuleiro inicial do Fetaix.



Fonte: Ribas (2020)

Por fim, os jogos de caça trazem o interessante desnível nas regras de jogo atribuídas a cada jogador. Similarmente, os objetivos de cada participante são também diferentes: um dos jogadores deve evitar e de alguma forma repelir o outro (a presa), ao passo que o adversário busca ativamente eliminar o adversário (o caçador). Jogos com regras e objetivos assimétricos são comumente nomeados por alegorias que representam a caça e a presa: Coiote e Galinhas (ilustrado na Figura 2.6), Raposa e Cachorros, Tigres e Vacas, dentre outros.

Figura 2.6: Jogo de Caça: Coiote e Galinhas.



Fonte: Ribas (2020)

2.2 Conceitos básicos de Orientação a Objetos

Em linguagens orientadas a objetos, as classes são as construções protagonistas do paradigma. Classes, assim chamadas no estrito sentido de classificação, são a estrutura fundamental da linguagem para a criação de entidades e distribuição de responsabilidades. Além disso, uma classe é uma interface bem definida para a execução de determinadas operações, em que as particularidades da implementação são, por assim dizer, omitidas

quando a classe é consumida. O termo instância é usado para designar a criação de uma variável com o tipo correspondente a uma classe.

Uma classe se potencializa com o conceito de encapsulamento. Essa noção introduz a possibilidade de restringir a visibilidade de membros da classe, como métodos (essencialmente funções) e variáveis definidas no escopo da classe. O encapsulamento se dá, de modo geral, pelos seguintes modificadores de acesso:

- **Privado:** método ou variável visível apenas dentro do escopo da classe. Isto é, instâncias não podem acessar membros definidos como privados.
- **Protegido:** membro da classe visível apenas nos escopos da própria classe e de classes herdeiras.
- **Público:** métodos e variáveis públicos são acessíveis pelo escopo da própria classe, de classes herdeiras e a partir de uma instância dessa classe.

Código 2.1: Exemplo de classe e modificadores de acesso usando a linguagem Kotlin.

```
1   class Calculator {
2       public fun sum(a: Int, b: Int): Double {
3           return castToDouble(a) + castToDouble(b);
4       }
5
6       private fun castToDouble(a: Int): Double {
7           return a.toDouble();
8       }
9   }
10
11  fun main(args: Array<String>) {
12      val c = Calculator()
13      println(c.sum(5, 2))
14      println(c.castToDouble(1))
15  }
```

O Código 2.1 mostra um breve exemplo sobre encapsulamento. A classe $Calculator$ define 2 métodos (funções): sum, método público que mapeia dois números inteiros para um número real, que representa a soma de ambos, e $castToDouble$, método privado que converte um inteiro em um número real. A função $main$, na linha 12, cria uma instância da classe $Calculator$, chamada de c. Em seguida, na linha 13, é impresso na saída padrão a soma entre os números inteiros 5 e 2 pelo uso do método público sum, definido na classe $Calculator$. Entretanto, a invocação (chamada) ao método $castToDouble$, na linha 14, implica em erro, pois esse método não é acessível pela instância por ser privado.

A herança de classes é outro conceito determinante nas linguagens orientadas a objetos. Pragmaticamente, se a classe D herda da classe C, então D absorve todos os

métodos e variáveis de classe públicas e protegidas definidas em C. Ademais, a nível de sistema de tipos, se o é uma instância de objeto do tipo D, então pode-se afirmar que a avaliação do tipo de o é verdadeira tanto quando testada para D quanto para o tipo C, i.e., o é do tipo D e, por herança, o é do tipo C também.

Código 2.2: Exemplo de herança na linguagem Kotlin.

```
1  class C {
2      public fun m1() { /* ... */ }
3
4      protected fun m2() { /* ... */ }
5
6      private fun m3() { /* ... */ }
7  }
8
9  class D : C() { /* ... */ }
```

No exemplo ilustrado pelo Código 2.2, a classe D herda da classe C. Nesse processo, D irá incorporar à sua própria estrutura os métodos $m1$ e $m2$, respectivamente público e protegido. Por definição, o método $m3$, privado, não será assimilado pela classe D.

Outra capacidade oportuna de linguagens orientadas a objetos é o conceito de classe abstrata. Em essência, uma classe abstrata A não pode ser instanciada: ela é definida apenas para que outras classes a estendam, isto é, herdem de A. Tal facilidade permite que conceitos por demais abstratos sejam modelados em classes não instanciáveis, mas, ao mesmo tempo, abre a possibilidade para a especialização desses mesmos conceitos por classes que os tornem palatáveis pela instanciação. Mais do que isso: métodos também podem ser abstratos, devendo obrigatoriamente ser implementados por quaisquer classes não abstratas que derivem, por exemplo, A.

Código 2.3: Classe abstrata escrita em Kotlin.

```
1  abstract class A {
2      public abstract fun m()
3
4      public fun f() { /* ... */ }
5  }
6
7  class B : A() {
8      public override fun m() { /* ... */ }
9  }
```

O Código 2.3 ilustra o conceito de classe abstrata. A classe B herda da classe A e é forçada, por não ser também abstrata, a descrever a implementação do método m, abstrato na definição dada por A. Além disso, pela herança anteriormente discutida, B

passa a contar com o método público f.

O último aspecto de interesse pertinente a linguagens OO é o polimorfismo paramétrico, o qual é comumente chamado em linguagens como Kotlin e Java de genéricos. Esse recurso foi introduzido como uma forma de manter o comportamento de uma classe sem depender necessariamente de um tipo, de modo que se mantenha a conveniência da tipagem estática, contribuindo para a consistência de um programa e reduzindo substancialmente a possiblidade de erros.

Código 2.4: Estrutura de dados Pilha sem o uso de polimorfismo paramétrico.

```
1   class AStack {
2       fun push(element: Any) { /* ... */ }
3
4       fun pop(): Any { /* ... */ }
5   }
6
7   fun main(args: Array<String>) {
8       val s = AStack()
9       s.push(1)
10      s.push(2)
11      val x: Int = s.pop() as Int
12      val y: String = s.pop() as String
13  }
```

O Código 2.4 ilustra o tipo abstrato de dados Pilha implementado em Kotlin sem o advento do polimorfismo paramétrico. As operações elementares de uma pilha, i.e., inserir (método $push$) e remover / recuperar (pop), são feitas sobre objetos em que os tipos não são garantidos. O tipo Any, de fato, aceita qualquer tipo de instância de objeto; é similar ao $java.lang.Object$ da linguagem Java. Logo, a operação de pop da pilha deve ser seguida de uma coerção (*cast*) de tipo, notada nas linhas 17 e 18 da $main$ definida no Código 2.4. Entretanto, como não há garantia de tipos, a coerção da linha 18 falha apenas em tempo de execução, pois a pilha contém apenas objetos Int (linhas 15-16).

Código 2.5: Estrutura de dados Pilha com polimorfimo paramétrico.

```
1   class AStack<T> {
2       fun push(element: T) { /* ... */ }
3       fun pop(): T { /* ... */ }
4   }
5   fun main(args: Array<String>) {
6       val s = AStack<Int>()
7       s.push(1)
8       s.push(2)
9       val x: Int = s.pop()
10      val y: String = s.pop()
11  }
```

Essa situação é trivialmente contornada pelo polimorfismo paramétrico. O Código 2.5 demonstra como a pilha pode ser implementada com polimorfismo paramétrico e, em especial, deixa claro o poder desse recurso. A notação $< T >$ adicionada ao lado do nome da classe indica que a declaração de uma instância de $AStack$ só será completa se o tipo for definido, como mostra a linha 13. Dessa forma, todas as operações de inserção e remoção de elementos da pilha são seguras a nível de tipo, logo, a tentativa de recuperar um elemento da pilha como uma $String$, conforme tenta a linha 17, incorre em erro em tempo de compilação, reforçando ainda mais o sistema de tipos e a confiabilidade da execução do programa ao detectar um erro em tempo de análise semântica.

2.3 Introdução aos Algoritmos de Decisão Competitiva

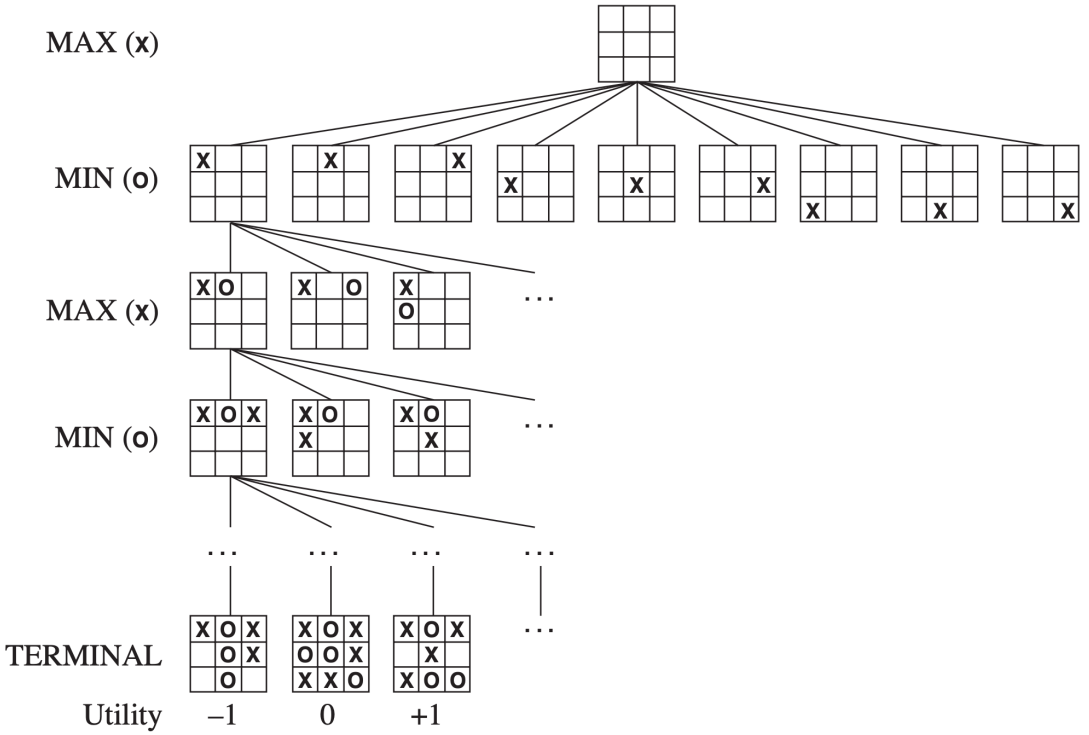
O Minimax é um dos mais proeminentes Algoritmos de Decisão Competitiva. Em especial, o Minimax personifica a intuição da tomada de decisão de jogos de tabuleiro, dadas as construções abstratas que definem o algoritmo. É, portanto, a escolha natural para a projeto de uma Inteligência Artificial apta a a representar um jogador de jogos abstratos e de estratégia.

O Minimax introduz o conceito dual de jogadores: o jogador MIN e o jogador MAX. Sob a ótica da Inteligência Artificial, o jogador MIN é o adversário e MAX é a própria IA; MAX, assim, busca sempre executar ações que resultem em maior ganho, ao passo que MIN, ao perseguir a sua própria vitória, objetiva reduzir ao máximo o ganho de MAX.

O mencionado ganho é, de fato, uma função que avalia o quanto a escolha de uma ação beneficia um jogador. A avaliação da qualidade de uma determinada ação se dá pela aplicação da função de valor sobre um estado; os termos função de valor, função custo e utilidade são intercambiáveis nesse contexto. Logo, o Minimax determina que MAX escolha, dentre as várias ações disponíveis, aquela que tem o maior valor obtido pela aplicação da função custo; e, na perspectiva de MAX, o jogador MIN irá optar pela ação que resulta no menor valor utilidade, visto que a função de valor é construída de tal forma que ações prejudiciais a MAX, i.e., que podem ou efetivamente o levam à derrota, tenham valores escalares com tendência a $-\infty$, e ações que beneficiem MAX, propulsionando-o em direção a vitória, resultem em valores tendendo a $+\infty$.

A aplicação do Minimax constrói uma árvore de aridade n. Cada nível dessa árvore contém nós que representam um turno do jogo, isto é, cada nível da árvore representa

Figura 2.7: Árvore parcial do Minimax para o Jogo-da-Velha.



Fonte: Russell and Norvig (2010)

os estados que podem ser atingidos ora pelo jogador MIN (nível l), ora pelo jogador MAX (nível $l + 1$.) Ou seja, os níveis da árvore Minimax são alternadamente interpretados. Então, quando MAX se encontra em vias de escolher uma ação, este optará pela ação que contém o maior valor utilidade, prevendo que MIN optará pelo caminho que dê o menor ganho possível a MAX. Naturalmente, na dinâmica de um jogo em que a Inteligência Artificial (MAX) confronta um usuário humano (MIN), MIN nem sempre irá recorrer às melhores ações. Entretanto, o algoritmo do Minimax assume que o adversário é sempre o melhor adversário, o que culmina em uma estratégia de jogo ao mesmo tempo gulosa e cautelosa. A Figura 2.7 ilustra uma árvore parcial do Minimax para o Jogo da Velha.

O Algoritmo 1 apresenta a construção clássica do Minimax, conforme demonstrado por Russell and Norvig (2010). Essa variante, como resta claro pela linha 23, inicia o jogo por MIN. Entretanto, o Minimax é indiferente ao jogador que começa a partida, sendo suficiente, nesse caso, que a linha 23 compute inicialmente $\arg\max_{a \in Acoes(s)} MaxValor(Resultado(estado, a))$ para que a disputa inicie com uma jogada de MAX.

Um dos problemas do Minimax, como bem apontdo por Russell and Norvig (2010), é que a expansão da árvore aumenta exponencialmente com a profundidade. A fim de

Algoritmo 1 Minimax (RUSSELL; NORVIG, 2010).

```
1:  function MaxValor(estado)
2:      if Terminal(estado) then
3:          return Utilidade(estado)
4:      end if
5:      v ← −∞
6:      for a ∈ Acoes(estado) do
7:          v ← max(a, MinValor(Resultado(s, a)))
8:      end for
9:      return v
10: end function
11:
12: function MinValor(estado)
13:     if Terminal(estado) then
14:         return Utilidade(estado)
15:     end if
16:     v ← +∞
17:     for a ∈ Acoes(estado) do
18:         v ← min(v, MaxValor(Resultado(s, a)))
19:     end for
20:     return v
21: end function
22:
23: function Minimax(estado)
24:     return arg max_{a∈Acoes(s)} MinValor(Resultado(estado, a))
25: end function
```

amortizar o custo da expansão, pode-se observar que nem todos os caminhos enumerados pelo Minimax são efetivamente atingíveis. Logo, algumas subárvores do Minimax, originadas por uma dessas decisões infactíveis segundo a dinâmica do algoritmo, podem ser totalmente eliminadas.

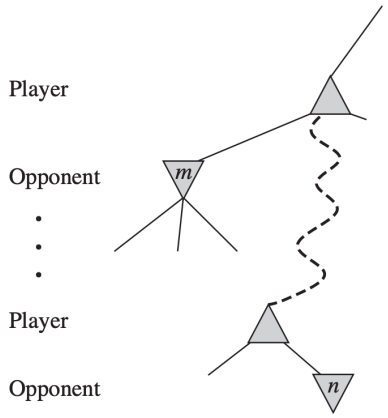
Por exemplo, quando as ações de MAX são enumeradas, não é necessário expandir ramificações da partida se uma dessas decisões garantidamente não será escolhida. Em especial, se MAX já anteviu, durante o percurso do algoritmo, que um estado s oferece uma utilidade menor que um estado s' anterior (no mesmo nível ou acima na árvore), então s nem sequer precisa ser analisado, pois escolhê-lo não é vantajoso em relação ao reconhecido s', como observado na Figura 2.8. Lógica similar se aplica no caso de MIN.

A noção da desnecessidade de algumas subárvores culmina em uma otimização do Minimax chamada Poda α-β. O termo *poda* endereça exatamente a diminuição do espaço de busca. O corte se dá pela introdução de duas variáveis no algoritmo Minimax:

- α**:** maior valor utilidade do nó observado para o jogador MAX durante a expansão dos estados.
- β**:** maior valor utilidade para o jogador MIN, analogamente a α.

O Algoritmo 2, aqui exposto conforme Russell and Norvig (2010), mostra exatamente a noção de eliminação de nós (e consequentemente subárvores). A introdução dos parâmetros α e β, consoantes com a semântica discutida, interrompem a a derivação da árvore ou porque já foi observado um valor mais vantajoso para MAX (linhas 8-10) ou porque MIN tinha também no percurso um valor utilidade maior (linhas 23-25).

Figura 2.8: Poda α-β: se a ação m é melhor que n para o jogador $Player$, então a jogada n nunca será executada e pode-se aplicar a poda na árvore.



Fonte: Russell and Norvig (2010)

A otimização ocasionada pela Poda α-β, portanto, é nada dispendiosa, requerendo

Algoritmo 2 Poda α-β (RUSSELL; NORVIG, 2010).

```
1:  function MaxValor(estado, α, β)
2:      if Terminal(estado) then
3:          return Utilidade(estado)
4:      end if
5:      v ← −∞
6:      for a ∈ Acoes(estado) do
7:          v ← max(a, MinValor(Resultado(s, a), α, β))
8:          if v ≥ β then
9:              return v                                   ▷ Poda.
10:         end if
11:         α ← max(α, v)
12:     end for
13:     return v
14: end function
15:
16: function MinValor(estado)
17:     if Terminal(estado) then
18:         return Utilidade(estado)
19:     end if
20:     v ← +∞
21:     for a ∈ Acoes(estado) do
22:         v ← min(v, MaxValor(Resultado(s, a)))
23:         if v ≤ α then
24:             return v                                   ▷ Poda.
25:         end if
26:         β ← min(β, v)
27:     end for
28:     return v
29: end function
30:
31: function BuscaAlfaBeta(estado) v ← MaxValor(estado, −∞, +∞)
32:     return arg max_{a∈Acoes(s)} MinValor(Resultado(estado, a))
33: end function
```

unicamente pontuais adições ao algoritmo do Minimax visto no Algoritmo 1. Ademais, a nível prático nada muda: a Poda α-β retorna as mesmas ações que seriam vistas no Minimax, visto que o corte de subárvores se dá para estados que, com efeito, jamais seriam selecionados pelos jogadores no Minimax puro.

3 PROPOSTA

3.1 Estrutura fundamental para a implementação de uma plataforma de jogos abstratos e de estratégia

A diversidade dos jogos abstratos e de estratégia é uma das mais proeminentes características que os tornam interessantes. A despeito da variabilidade, virtualmente todos os jogos de tabuleiro compartilham os mesmos elementos: as peças, as posições sobre as quais estas podem ser dispostas, a conectividade entre essas posições, além da própria geometria da tabuleiro. Todos esses elementos compõem, afinal de contas, o núcleo do que se considera um jogo de tabuleiro.

Ao explicarmos a um interlocutor como funciona um jogo de tabuleiro, os elementos fundamentais são apenas o preâmbulo. A dinâmica que efetivamente propulsiona um jogo de estratégia está nas regras que determinam quais movimentos cada jogador pode realizar – não raro acompanhados de momentos diferentes de jogo –, quais restrições se impõem conforme a partida avança e o porquê da imposição de tais limitações. Essa contextualização se torna particularmente difícil se a explicação de funcionamento de um jogo for descrita em passos bem definidos e livres de ambiguidade.

Descrever um jogo abstrato e de estratégia, em uma linguagem natural ou pseudocódigo, tem um interessante apelo prático. A enumeração da semântica das peças, das regras de movimento e os critérios de fim de jogo estariam de tal forma organizados que seria lícito admitir que essa descrição serviria como um guia para um jogador humano. Isto é, cada movimento desse jogador caracterizar-se-ia por uma passagem avaliativa sobre os passos ali detalhados.

Um problema interessante que nasce desse contexto é como tornar essa descrição informal computável. Uma primeira intuição sugeriria a construção de uma linguagem de programação de jogos abstratos e de estratégia: uma linguagem livre de contexto em que a gramática se caracteriza pela expressividade subjacente de um jogo de tabuleiro. Naturalmente, a semântica dessa linguagem deve materializar aquele que é o objetivo inicial: uma partida do jogo.

Quando um dos jogadores convocados para uma partida de jogo de tabuleiro passa a ser uma Inteligência Artificial, algumas abordagens se avultam. Os Algoritmos de Decisão Competitiva surgem como uma atraente alternativa: o Minimax, por exemplo, é facilmente aplicado na busca por bons movimentos em um jogo de tabuleiro. A intuição

do Minimax, i.e., enumerar os possíveis estados e valorar o resultado produzido por um movimento de um jogador, considerando as implicações futuras de uma escolha, são, ao fim e ao cabo, o que norteia a estratégia de um jogador em um jogo de tabuleiro.

No contexto da avaliação de um programa de uma linguagem de programação voltada a jogos de tabuleiro, a introdução de uma Inteligência Artificial é particularmente desafiadora. A complexidade da avaliação da árvore de sintaxa abstrata construída na análise semântica da linguagem seria simbioticamente absorvida pela implementação do motor da Inteligência Artificial, aumentando de forma dramática tanto o tempo de execução do algoritmo implementado pela IA quanto a escalabilidade do sistema, uma vez que quaisquer novas funcionalidades adicionadas à linguagem poderiam afetar diretamente o funcionamento da IA.

Se, por um lado, a implementação de uma linguagem de programação de jogos, ajustada a um sistema de jogos de tabuleiro que implementa uma IA como jogador, é tão interessante quanto desafiadora, por outro, essa ideia faz um convite à abstração. O processo de isolamento dos aspectos que permeiam um jogo abstrato de estratégia, indispensável na solução da linguagem de jogos de tabuleiro, mostra-se um importante ativo na construção de uma alternativa: um sistema menos flexível que uma linguagem de programação, mas ainda assim dotado de escalabilidade suficiente para que diferentes jogos sejam construídos e jogados por uma Inteligência Artificial. Nesse sentido, o paradigma de Orientação a Objetos oferece os dois mecanismos essenciais para essa empreitada: o alto nível de abstração e um considerável conjunto de linguagens de programação de propósito geral que o suportam.

Indubitavelmente, um sistema que conte com vários jogos abstratos de estratégia é consideravelmente menos expressivo e flexível que uma linguagem de programação de jogos. Entretanto, ainda assim é razoável admitir que a modelagem desse sistema pode prever a configuração e até mesmo a alteração pontual dos jogos, por um usuário leigo, de modo a originar diferentes jogos ou alterar aqueles já existentes. Além disso, a inclusão da IA nesse sistema seria largamente facilitada: teria menos vulnerabilidade a efeitos colaterais (em oposição aos efeitos potencialmente produzidos por alterações na linguagem de programação de jogos) e seria mais eficiente, visto que a etapa de interpretação/execução da linguagem estaria ausente.

Na busca pelo refinamento de um sistema OO com suporte a uma Inteligência Artificial, o primeiro exercício é a abstração dos elementos que formam o núcleo dos jogos abstratos e de estratégia. Com a facilidade da percepção alto nível, a compreensão

do universo dos jogos de tabuleiro é significativamente melhorada. A abstração, portanto, tende a viabilizar um modelo para a implementação do sistema, ao mesmo tempo em que aprofunda a cognição sobre os jogos.

3.2 Abstrações pertinentes aos jogos abstratos de estratégia

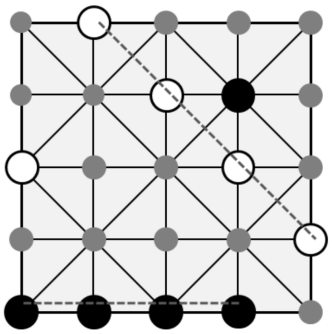
Fundamentalmente, um jogo abstrato de estratégia se caracteriza por um tabuleiro, peças, um conjunto de movimentos que cada jogador pode realizar sobre as posições do tabuleiro e um ou mais critérios de fim de jogo. As diferenças que constrastam os diferentes jogos entre si residem nos movimentos permitidos e no teste de fim de jogo, além da disposição inicial do tabuleiro - por exemplo, o tabuleiro inicia vazio ou com as peças de cada jogador já dispostas sobre ele. Tais simetrias, porém, não impedem que os jogos compartilhem inúmeras similaridades, o que permite a classificação deles em jogos de bloqueio e alinhamento, deslocamento, posicionamento, captura e caça.

O primeiro dos 21 jogos abstratos propostos por Ribas (2020) é o Jogo-da-Velha-4. Nele, cada jogador pode alternadamente inserir uma peça em uma posição livre do tabuleiro matricial de dimensões 5×5, o qual inicia o jogo vazio. O primeiro jogador a realizar um alinhamento vertical, horizontal ou diagonal com quatro peças, desde que feito sobre as arestas do tabuleiro, é o vencedor. A figura 2.2 mostra um tabuleiro do Jogo-da-Velha-4 em situação de fim de jogo.

Uma variação, por assim dizer, do Jogo-da-Velha-4 é o Tic Tackle. O mesmo tabuleiro matricial 5×5 inicia com as peças de cada jogador dispostas alternadamente nas extremidades superior e inferior do tabuleiro. Cada jogador tem, portanto, 5 peças, as quais podem ser deslocadas em um passo para qualquer posição livre, desde que o movimento seja feito sobre uma aresta do tabuleiro. O jogador que fechar um alinhamento com quatro peças é o vitorioso, exatamente como no Jogo-da-Velha-4; contudo, esse alinhamento não necessariamente precisa estar sobre uma aresta, como mostra a figura 3.1.

A primeira estrutura que ambos compartilham, sem qualquer alteração, é o tabuleiro matricial 5×5. O objetivo dos dois jogos também é o mesmo: alinhamento, inclusive com o mesmo número de peças. A primeira diferença que vemos é a configuração de começo de partida: o Jogo-da-Velha-4 inicia com o tabuleiro vazio, ao passo que o Tic Tackle prevê as peças de cada jogador devidamente posicionadas sobre o tabuleiro. Além disso, existe uma substancial diferença entre deslocar uma peça para uma

Figura 3.1: Tic Tackle em estado de fim de jogo.



Fonte: Ribas (2020)

posição adjacente e inserir uma peça em uma posição livre: são movimentos conceitualmente diferentes. Por fim, o critério de fim de jogo é ligeiramente relaxado no Tic Tackle: enquanto o Jogo-da-Velha-4 restringe que o 4-alinhamento ocorra sobre uma aresta, o Tic Tackle estende essa característica para suportar vitórias com alinhamentos em quaisquer linhas, mesmo aquelas que não representam arestas no tabuleiro.

É fácil ver que esses dois jogos de alinhamento, reservadas as pontuais divergências, se valem essencialmente dos mesmos preceitos. As peças não têm qualquer semântica, são apenas rótulos que identificam um jogador; somente posições livres podem ser conquistadas, seja por deslocamento (Tic Tackle), seja por inserção (Jogo-da-Velha-4); por fim, ambos objetivam um 4-alinhamento, oportunamente flexibilizado pelo Tic Tacke. Dissociar essas características da materialização física dos jogos e decompô-las como entidades classificáveis, desacopladas de uma implementação específica de um jogo, expande tanto a concepção que se pode ter de jogos abstratos de estrágias quanto as fronteiras para o início do desenvolvimento de um sistema capaz de suportar diferentes tipos de jogos de tabuleiro.

A peça é uma entidade, a priori, bastante simples. Há um jogador que a retém, numa visão de propriedade da peça: o dono pode tê-la no tabuleiro ou à mão, deixando-a pronta para o momento em que for preciso executar uma jogada de inserção. Estando colocada sobre o tabuleiro, a peça passa, então, a ser também caracterizada pela posição que ela ocupa, como, por exemplo, uma coordenada (i, j) de um tabuleiro matricial, onde i indexa a linha e j a coluna dessa matriz. Alguns jogos abstratos de estratégia, como o xadrez, atribuem semântica às peças: estas têm, além das propriedades aqui destacadas, uma alegoria (e.g., o Rei) e um conjunto específico de movimentos que podem ser executados.

No contexto dos jogos abstratos de estratégia, no entanto, uma peça é inócua sem

um tabuleiro. Apesar dessa relevância, o tabuleiro é uma entidade caracteristicamente passiva: ele apenas fornece uma estrutura física sobre a qual o jogo irá se desenrolar. Como grande representante do aspecto visual de um jogo de estratégia, o tabuleiro é dotado de uma geometria, dentro da qual as posições jogáveis estão fatalmente presentes, e pode ser definido como uma estrutura plana em que as posições são endereçadas por algum identificador e podem estar conectadas entre si, formando caminhos no tabuleiro.

A dinâmica de um jogo de tabuleiro começa efetivamente com os movimentos executados pelos jogadores. Tipicamente observado em jogos abstratos e de estratégia, um movimento pode ser especializado em dois grandes tipos: inserção e deslocamento. Um movimento de inserção acontece quando um jogador posiciona uma peça anteriormente ausente sobre o tabuleiro, enquanto o movimento de deslocamento prevê a movimentação de uma peça previamente disposta em uma posição do tabuleiro. O movimento de inserção é trivial, guardando nenhum potencial de especialização, diferentemente do movimento de deslocamento.

Independentemente das regras de jogo, um movimento de deslocamento tem algumas propriedades. Vejamos:

- **Quantidade:** quantas posições uma peça pode ser deslocada. Por exemplo, o Tic Tackle permite apenas movimentos unitários, também chamados de deslocamento em 1 casa.
- **Direção:** uma combinação livre de movimento horizontal, vertical ou diagonal.
- **Sentido:** o sentido de movimento é à esquerda, à direita ou ambos.
- **Alcance:** se o movimento pode ter como fim uma posição livre ou ocupada.
- **Salto:** saltar sobre uma ou mais peças, eventualmente limitadas às peças do jogador que iniciou o movimento ou às peças do adversário. As regras de jogo podem restringir tanto o dono das peças sobre as quais o salto é feito quanto o número de peças que podem ser cobertas pelo salto.

As regras de jogo, nas quais se pode incluir também os critérios de fim de jogo, são invariavelmente calcadas sobre as abstrações mais elementares. De fato, as regras de jogo estão presentes em todos as abstrações vistas até aqui: quais tipos de movimento cada jogador pode aplicar, como o tabuleiro pode ser explorado, de que forma uma peça pode carregar consigo um significado mais profundo que apenas um rótulo indicativo de jogador. Logo, as regras de jogo são a combinação dos elementos fundamentais de um jogo abstrato e de estratégia e dos diferentes papéis que esses elementos podem ter ao

longo de uma partida. Enquanto os elementos fundamentais são passíveis de generalização, as regras de fim de jogo estão intimamente associadas ao tipo mais específico do jogo de tabuleiro: definem o jogo em si, e devem ser esmiuçadas conforme surge a necessidade de se introduzir uma nova proposta de jogo.

Reunindo todas essas abstrações, temos um cenário bastante mais amplo que a descrição crua e objetiva de como funciona, por exemplo, o Jogo-da-Velha-4. Passamos, então, a enxergar o jogo abstrato e de estratégia como uma série de interações restringidas por regras (os movimentos) executadas sobre uma estrutura física, geometricamente fácil de compreender (o tabuleiro), a qual torna visual as estratégias adotadas por cada jogador em um determinado instante de tempo (as peças dispostas sobre o tabuleiro). Os dois jogadores têm, por fim, um só objetivo: vencer a partida. Toda essa dinâmica, inclusive a atribuição de vencedor, é cadenciada pelas regras de jogo, que acabam condensando a essência do que se considera o próprio jogo.

A introdução de uma Inteligência Artificial nesse contexto pode parecer desafiadora. Se existem várias abstrações comuns aos jogos, mas cada um é realmente tão particular que requer uma implementação específica, qual estratégia adotar para incluir o computador como um jogador? A primeira intuição pode sugerir que cada jogo contaria com a sua própria instância de Inteligência Artificial, afinal de contas, esta só pode operar sobre aquilo que é conhecido e, dada a especificidade de cada jogo, a IA seria igualmente especializada.

Tomemos o Minimax como exemplo: a árvore de decisão é derivada a partir de um estado s_i. No caso de um jogo de tabuleiro, o estado s_i representa exatamente a disposição do tabuleiro em um determinado instante de jogo. Um movimento m executado sobre o tabuleiro representado por s_i origina um novo estado s_i'. A valoração do quão boa é essa jogada pode, via de regra, ser inferida pelo resultado do movimento: se o jogo não terminou, esse valor pode ser 0, caso contrário, a jogada pode ter atribuído um valor não-nulo. Essa explicação didática, abstrata, pode ser mantida se a IA for externalizada da implementação do jogo de tabuleiro: nesse caso, o Minimax passaria a executar movimentos e a inferir o valor de cada jogada a partir de uma interface que abarcasse todo e qualquer jogo de tabuleiro.

Extrapolar o desígnio para um Minimax que recorre a interfaces bem-definidas para a expansão e valoração da árvore de decisão impõe uma situação singular. De fato, estaríamos diante de uma situação em que o Minimax desconhece absolutamente qual jogo está sendo jogado: a implementação do algoritmo apenas conhece meios de enu-

merar possíveis jogadas, aplicar essas jogadas sobre um tabuleiro e decidir, a partir da indicação de fim de jogo, qual valor atribuir a um nó da árvore. Em termos práticos, o sistema de jogos abstratos contaria com n implementações de jogos de tabuleiro e uma, e somente uma, implementação de Inteligência Artificial: um Minimax agnóstico ao jogo.

As abstrações discutidas iluminam consideravelmente o caminho para a modelagem de um sistema em que múltiplos jogos de tabuleiro podem ser implementados e jogados por uma Inteligência Artificial. Contudo, há a necessidade de colocar essas abstrações à prova, senão no todo, ao menos em parte. A consequência natural dessa decisão é a modelagem orientada a objetos de um sistema que comporte tanto quanto possível as ideias até aqui trabalhadas.

3.3 Modelagem orientada a objetos

A escolha do paradigma de linguagem de programação mais apropriado para o desenvolvimento de um sistema fortemente calcado na abstração é carregado de subjetivismo. No entanto, as considerações entre os três principais modelos - procedural, funcional e orientado a objetos - inevitavelmente devem ser feitas para que o processo de escolha considere diferente opções e, consequentemente, as vantagens e desvantagens de cada modelo. Todavia, a afinidade e a naturalidade com que a orientação a objetos é capaz de satisfazer a contento as abstrações e as estruturas imaginadas acabam por torná-la a predileta.

O modelo procedural mais conservador, em uma implementação similar a linguagem C, pouco oferece em termos de abstração. Apenas uma fração do que uma linguagem orientada a objetos oferece pode ser replicada, e ainda assim diante do sacrifício do encapsulamento e da herança. Ainda que seja possível alguma ginástica no uso de $structs$ e $unions$ do C, por exemplo, a expressividade se reduz de forma alarmante, ainda que o ganho de performance seja considerável. Logo, no que diz respeito à escalabilidade do sistema para o suporte a novos jogos, o modelo procedural seria factível, mas não adequado; e tal limitação implicaria em uma revisão de como incluir a IA como um ator desse sistema.

O paradigma funcional se sobressai em relação ao procedural justamente pela expressividade e abstração. No contexto do sistema de jogos abstratos e de estratégia e Inteligência Artificial, de fato, uma linguagem funcional ofereceria as mesmas facilidades que uma linguagem orientada a objetos, especialmente aquelas mais flexíveis, como

F#. Contudo, a visualização de um modelo pensado com o viés de OO ensejaria algumas pontuais adaptações no modelo funcional, fato que acarretaria algum retrabalho. Além disso, a linguagem escolhida - Kotlin -, ainda que não funcional por definição, traz uma proposta similar ao F# e oferece inúmeras construções herdadas do paradigma funcional.

O modelo de classes aqui detalhado se vale sobretudo da herança e do polimorfismo paramétrico para que o sistema seja escalável e passível de contar com uma Inteligência Artificial Minimax agnóstica ao jogo de tabuleiro. A herança viabiliza a definição de interfaces comuns para os métodos e propriedades das implementações dos jogos de tabuleiro. O polimorfismo paramétrico permite que propriedades e parâmetros de métodos de classes assumam diferentes tipos, estendendo ainda mais a flexibilidade tanto do modelo quanto da implementação, o que fica mais claro ao apresentarmos uma visão geral das classes modeladas para o desenvolvimento do sistema.

Um jogador é representado pela classe $Player$. Essa classe, vista no Código 3.1, encapsula as seguintes propriedades:

- **id:** número inteiro $i > 0$, identificador do jogador.
- **name:** $string$ que representa o nome do jogador.
- **type:** tipo do jogador conforme a visão do Minimax. É uma instância do enumerado $PlayerType$, o qual lista os valores MAX e MIN.
- **victories:** contador inteiro $v \geq 0$ do número de vitórias.
- **losses:** contador inteiro $l \geq 0$ do número de derrotas.
- **draws:** contador inteiro $d \geq 0$ do número de derrotas.

Código 3.1: Classe $Player$ e enumerado $PlayerType$.

```
data class Player(
    val id: Int,
    var name: String,
    val type: PlayerType,
    var victories: Int = 0,
    var losses: Int = 0,
    var draws: Int = 0
)

enum class PlayerType {
    MIN,
    MAX
}
```

O movimento de um jogador é retratado por uma sub-hierarquia de classes que tem como base a classe $PlayerMovement$ (Código 3.2). Esta é uma classe abstrata, i.e.,

não instanciável, e define as seguintes propriedades, herdadas por todas as subclasses:

- **player:** instância da classe $Player$. Referencia o jogador que executou o movimento.
- **targetCoordinate:** par (i, j) que representa a posição de destino da jogada.

Código 3.2: Classe abstrata $PlayerMovement$.

```
abstract class PlayerMovement(
    val player: Player,
    val targetCoordinate: Pair<Int, Int>
)
```

As duas classes que especializam um movimento, i.e., herdam de *PlayerMovement*, são instanciáveis e classificam os conceitos de movimento de inserção e de deslocamento, vide o Código 3.3. A classe $InsertionMovement$ abstrai o movimento de inserção, e não define nenhuma propriedade ou método específico: apenas estende a classe $PlayerMovement$. O movimento de deslocamento é modelado pela classe *DisplacementMovement*, que, além das propriedades herdadas de $PlayerMovement$, define ainda:

- **sourceCoordinate:** par (i, j) que representa a posição de origem da jogada.

Código 3.3: Classes $InsertionMovement$ e $DisplacementMovement$.

```
class InsertionMovement(
    player: Player,
    targetCoordinate: Pair<Int, Int>
) : PlayerMovement(player, targetCoordinate)

class DisplacementMovement(
    player: Player,
    val sourceCoordinate: Pair<Int, Int>,
    targetCoordinate: Pair<Int, Int>
) : PlayerMovement(player, targetCoordinate)
```

Apresentada essa pequena sub-hierarquia, cabe aqui explicar o porquê de *PlayerMovement* definir a coordenada de destino do movimento. Ora, movimentos de inserção e deslocamento têm a mesma finalidade: atuar sobre uma posição final do tabuleiro, aqui entendida como uma coordenada alvo. Entretanto, apenas um movimento de deslocamento se caracteriza por sair de uma posição ($sourceCoordinate$) em direção a outra ($targetCoordinate$).

Uma peça é representada pela classe $Piece$ (Código 3.4). Essa classe concentra estas propriedades:

- **position:** coordenada (i, j) do tabuleiro sobre a instância está posicionada.
- **owner:** instância de $Player$, o dono da peça.

Nesse ponto, é preciso destacar o entendimento da modelagem de que um movimento é feito, de fato, sobre uma posição do tabuleiro. Esta pode ou não ter uma peça, instância de $Piece$, e o jogador que executa o movimento pode ou não ser o dono da peça - caso em que a jogada é invalidada. Esse entendimento, conquanto peculiar, é explicável: em um tabuleiro endereçado por coordenadas de linha e coluna (i, j), essas mesmas coordenadas são onipresentes no contexto: a peça está sobre uma posição de determinadas coordenadas, e o jogador especifica a jogada, em última análise, com base nessas mesmas coordenadas.

Código 3.4: Classe $Piece$.

```
class Piece(val position: Pair<Int, Int>) : Cloneable {
    var owner: Player? = null
        set(value) {
            field = value
            usable = field == null
        }

    var usable: Boolean = true
        private set

    fun hasOwner() = owner != null
}
```

Um tabuleiro é a síntese das regras de movimentação, critérios de fim de jogo e da geometria do tabuleiro. Assim, condensa responsabilidade suficiente para que um jogo abstrato de estratégia seja implementado. A fim de permitir uma interface bem definida para que a Inteligência Artificial seja agnóstica ao jogo em execução, todo e qualquer tabuleiro é derivado da classe $AbstractBoard$. Mais do que isso: a classe $AbstractBoard$ implementa um polimorfismo paramétrico para a tipagem do movimento executado sobre o tabuleiro. Numa notação similar aos genéricos da linguagem Java e Kotlin, temos $AbstractBoard < T >$, onde o tipo T obrigatoriamente deve estender a classe $PlayerMovement$. Dessa forma, quando um jogo é implementado o tipo do movimento que um jogador executa deve ser estaticamente definido. A classe $AbstractBoard$ define as seguintes propriedades:

- **player1:** instância de $Player$ que representa um dos jogadores.
- **player2:** análogo a $player1$.
- **height**: inteiro $h > 0$ que define a altura do tabuleiro.

- **width:** inteiro $w > 0$ que define a largura do tabuleiro.
- **boardMatrix**: matriz de dimensões $height \times width$ que retrata o tabuleiro efetivamente. Cada elemento é uma instância de $Piece$.

Código 3.5: Enumerado $PlayResult$.

```
enum class PlayResult {
    SUCCESS,
    UNREACHABLE_POSITION,
    OCCUPIED_POSITION,
    UNKNOWN_PLAYER
}
```

Os métodos da classe $AbstractBoard$ (Código 3.6) são todos abstratos, isto é, devem ser implementados pelas subclasses. Eles têm a seguinte semântica:

- **play:** $T \times bool \rightarrow PlayResult$**:** executa uma jogada a partir de um movimento do tipo paramétrico T (subclasse de $PlayerMovement$) e um booleano, o qual, se verdadeiro, apenas simula uma jogada. Retorna uma instância do enumerado $PlayResult$ (Código 3.5), assumindo o valor *SUCCESS* (jogada bem-sucedida), *UNREACHABLE_POSITION* (posição inatingível), *OCCUPIED_POSITION* (posição ocupada) ou *UNKNOWN_PLAYER* (jogador desconhecido).
- **enumeratePossibleMovements:** $Player \rightarrow [T]$**:** enumera todos os movimentos possíveis para uma instância de $Player$ no instante da invocação do método. Isto é, considera o estado atual da instância.
- **isMovementValid:** $T \rightarrow bool$**:** retorna verdadeiro se, e somente se, o movimento de tipo T é um movimento válido. A seguinte equivalência é garantida: $m.player = p \wedge isMovementValid(m) \iff m \in enumeratePossibleMovements(p)$.
- **getWinner:** $\perp \rightarrow \{Player, \perp\}$**:** retorna o jogador vencedor, instância de $Player$, caso aplicável. Se não há vencedor, então nulo ($\perp$) é retornado.

Código 3.6: Classe $AbstractBoard$.

```
abstract class AbstractBoard<T : PlayerMovement>(
    protected val player1: Player, protected val player2: Player,
    protected val height: Int, protected val width: Int
) {
    val boardMatrix by lazyOf(this.buildBoard())
    abstract fun play(movement: T, simulate: Boolean = false): PlayResult
    abstract fun enumeratePossibleMovements(player: Player): Iterable<T>
    abstract fun isMovementValid(movement: T): Boolean
    abstract fun getWinner(): Player?
}
```

A árvore Minimax tem como classe elementar um nó: $TreeNode$ (Código 3.7), o qual encapsula um estado do jogo de tabuleiro. Similarmente ao polimorfismo paramétrico aplicado a classe base dos tabuleiros, podemos reescrevê-la como $TreeNode < T >$, onde T é um tipo que seguramente herda da classe $PlayerMovement$. Isto é, o tipo do nó só é completamente resolvido quando $TreeNode$ é instanciada para um dos dois tipos de movimento possíveis na modelagem: $InsertionMovement$ ou $DisplacementMovement$. Isso fica claro ao analisarmos as propriedades da classe:

- **generatingMovement:** movimento gerador do nó, isto é, instância do tipo T. Se $generatingMovement$ for nulo, então a instância é a raiz da árvore Minimax.
- **movementsTrace:** coleção de movimentos do tipo T que retém o rastro de movimentos executados ao longo do jogo, isto é, todos os estados (nós) efetivamente visitados ao longo da partida. É uma lista escalonada por $FIFO$: o primeiro elemento é a primeira jogada do jogo, o segundo elemento a segunda e assim sucessivamente. Se $movementsTrace$ for vazio, então a instância é a raiz da árvore Minimax.
- **value:** valor $v \in \mathbb{R}$ do nó, resultado da aplicação da função custo do Minimax. Tem valor padrão 0.
- **children:** lista simplesmente encadeada de filhos do nó, i.e., cada elemento é uma instância de $TreeNode < T >$. Essa propriedade de encadeamento é a chave para a construção da árvore Minimax de aridade n.
- **level:** nível $l > 0$ da árvore Minimax.

Código 3.7: Classe $TreeNode$.

```
data class TreeNode<T : PlayerMovement>(
    val player: Player,
    val generatingMovement: T?,
    val movementsTrace: Collection<T> = mutableListOf(),
    var value: Double = 0.toDouble(),
    val children: MutableList<TreeNode<T>> = mutableListOf(),
    val level: Long
) {
    fun isLeaf() = children.isEmpty()
}
```

As classes que implementam o Minimax são duas: $Minimax$, que abstrai o algoritmo em sua forma pura, e $AlphaBetaSearch$, a qual projeta o Minimax com a otimização da Poda α-β. Ambas as classes, descritas no Código 3.9, estendem a classe abstrata $AdversarialSearch$ (Código 3.8). Esta prevê o duplo polimorfismo paramétrico:

é denotada por $AdversarialSearch < T, E >$, onde o tipo T garantidamente herda de $PlayerMovement$ e o tipo E é um $AbstractBoard < T >$. Ou seja, T é usado como um tipo paramétrico auxiliar para a restrição do tipo final de tabuleiro sobre o qual a IA irá executar as jogadas. As propriedades de $AdversarialSearch$, tacitamente absorvidas pelas subclasses $Minimax$ e $AlphaBetaSearch$, são:

- **maxPlayer:** instância de $Player$ do tipo MAX.
- **minPlayer:** instância de $Player$ do tipo MIN.
- **workingBoard**: instância do tabuleiro, de tipo $AbstractBoard < T >$. É utilizado para a manipulação dos Algoritmos de Decisão Competitiva durante o processo de construção da árvore Minimax.

Os métodos públicos e abstratos são:

- **computerPlays:** $\perp \rightarrow T$**:** a partir do estado atual retido pela instância, fornece uma jogada da Inteligência Artificial, i.e., retorna uma instância de movimento do tipo T. Executa, portanto, uma jogada de MAX ($maxPlayer$).
- **userPlays:** $T \rightarrow \perp$**:** computa um movimento do tipo T do jogador MIN ($minPlayer$).
- **restart:** $\perp \rightarrow \perp$**:** reinicia o jogo, redefinindo todas as estruturas internas da classe, em especial o estado atual. Deixa pronto, portanto, o reinício da aplicação do Minimax.

Código 3.8: Classe abstrata $AdversarialSearch$.

```
abstract class AdversarialSearch<T : PlayerMovement, E : AbstractBoard<T>>(
    protected val maxPlayer: Player,
    protected val minPlayer: Player,
    protected val workingBoard: E
) {
    abstract fun computerPlays(): T
    abstract fun userPlays(movement: T)
    abstract fun restart()
}
```

O duplo polimorfismo paramétrico de $AdversarialSearch$ materializa o desejo de uma IA agnóstica de jogo. A única informação de tipo sabida pela implementação da IA é que o jogo é uma implementação de $AbstractBoard$, sem mais informações adicionais. Portanto, todas as interações do Minimax e da Poda α-β são com os métodos e propriedades definidos na classe abstrata do tabuleiro: como esse métodos são implementados e o que efetivamente devem fazer é de inteira responsabilidade da classe que implementa $AbstractBoard$, ou seja, da implementação do jogo. Assim, a IA desconhece

qual jogo está sendo jogado: toda a interação dela com o tabuleiro é feito por métodos bem conhecidos, porém abstratos.

Código 3.9: Definições das classes $Minimax$ e $AlphaBetaSearch$.

```
abstract class Minimax<T : PlayerMovement, E : AbstractBoard<T>>(
    val maxPlayer: Player,
    val minPlayer: Player,
    val workingBoard: E
) : AdversarialSearch<T, E>(maxPlayer, minPlayer, workingBoard) {
    override fun computerPlays(): T { /* ... */ }
    override fun userPlays(movement: T) { /* ... */ }
    override fun restart() { /* ... */ }
}

abstract class AlphaBetaSearch<T : PlayerMovement, E : AbstractBoard<T>>(
    val maxPlayer: Player,
    val minPlayer: Player,
    val workingBoard: E
) : AdversarialSearch<T, E>(maxPlayer, minPlayer, workingBoard) {
    override fun computerPlays(): T { /* ... */ }
    override fun userPlays(movement: T) { /* ... */ }
    override fun restart() { /* ... */ }
}
```

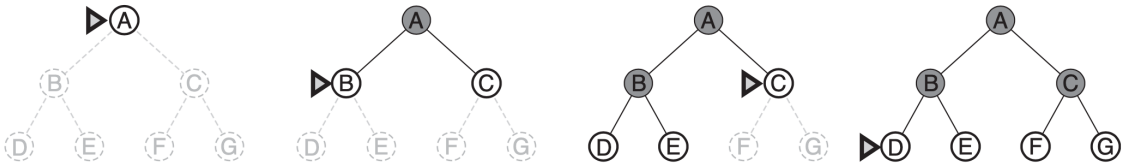
3.4 Algoritmos de Decisão Competitiva

O Minimax é um algoritmo de decisão competitiva bastante intuitivo. São poucos passos: expandir os nós da árvore, observando que cada nível é alternado por jogador (MAX ou MIN), aplicar um teste terminativo, que via de regra sinaliza se o estado leva a um nó que encerra o problema-objeto da busca, e atribuir a um nó não folha o valor máximo observado nos nós filhos (se o nível desse nó não folha for MAX) ou o valor mínimo dos filhos (se o nível corresponder a MIN). No entanto, existem algumas escolhas que podem ser feitas no momento de construir o algoritmo efetivamente.

A primeira delas é de que forma expandir os nós da árvore. Intuitivamente, parece interessante que a busca, ao visitar um estado s, verifique todas os possíveis estados a serem imediatamente atingidos a partir de s. Essa estratégia se reflete em uma busca em largura, isto é, na expansão completa nível a nível da árvore Minimax.

Uma alternativa a busca em largura é, naturalmente, a busca em profundidade. Nesse caso, ao visitar um estado s e descobrir que a partir deste pode-se chegar ao estado s', a busca se direciona para os estados atingíveis a partir de s', em vez de seguir para o

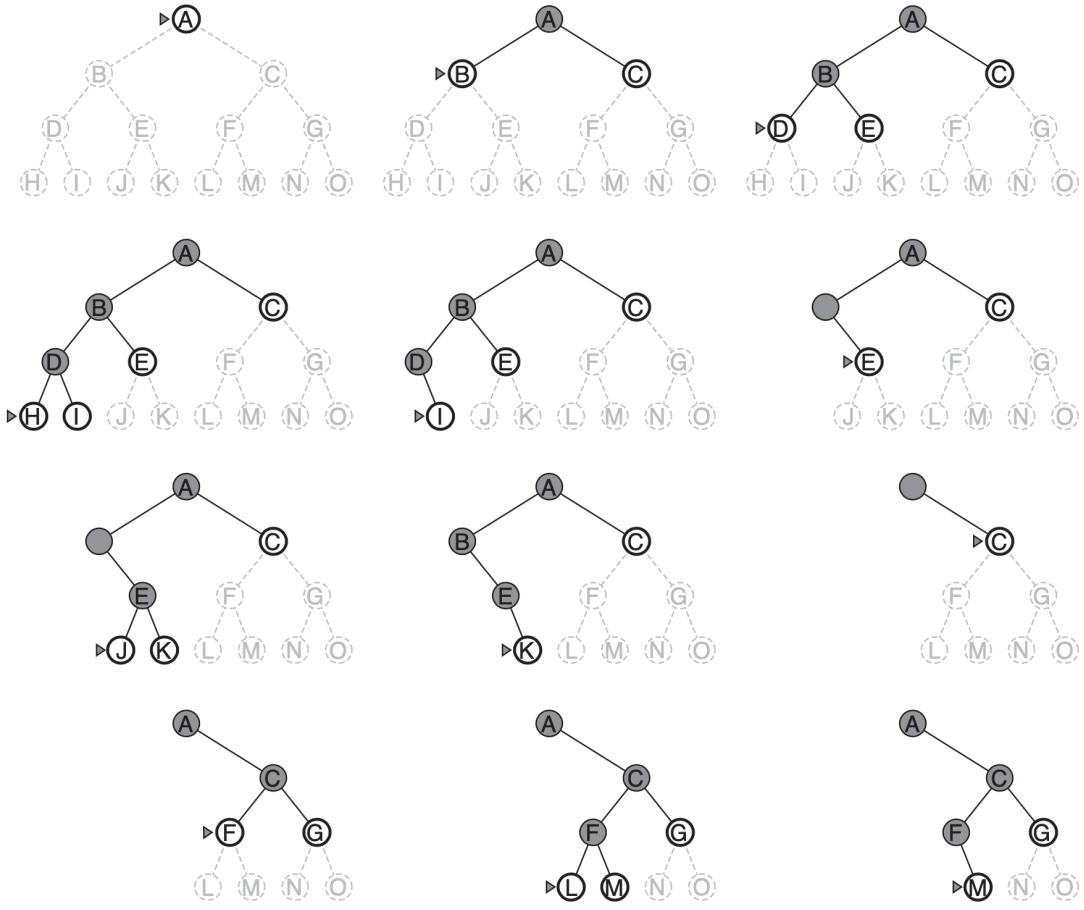
Figura 3.2: Exemplo de Busca em *Largura* em uma Árvore Binária.



Fonte: Russell and Norvig (2010)

próximo estado derivável a partir de s. Logo, se a busca em largura expande completamente um nível a cada iteração ou recursão, a busca em profundidade só completará um nível depois que todos os subníveis forem conhecidos. As Figuras 3.2 e 3.3 deixam clara essa diferença ao exemplificarem, respectivamente, buscas em largura e profundidade sobre uma árvore binária.

Figura 3.3: Exemplo de Busca em *Profundidade* em uma Árvore Binária.



Fonte: Russell and Norvig (2010)

O Minimax implementado no sistema de jogos abstratos e de estratégia aqui proposto se vale da busca em largura. A intuição para essa escolha é a seguinte: dada uma limitação de tempo ou profundidade, é mais vantajoso ter uma visão imediata de todas as possibilidades e riscos a um passo do estado atual do que ter o conhecimento do jogo k

níveis adiante (em profundidade). Essa abordagem cautelosa, divergente do Minimax em profundidade detalhado por Russell and Norvig (2010), antevê o problema que a busca em profundidade pode trazer em um regime de explosão de estados ou limitação de tempo: expandir apenas poucos nós por nível, deixando a decisão imediata do próximo passo consideravelmente deficitária. A implementação feita para o Minimax, expressa no Algoritmo 3, expande os nós da árvore em largura, e é auxiliado pelo Algoritmo 4, responsável pela aplicação da função de valor a partir de um percurso em profundidade sobre a árvore construída pelo Algoritmo 3.

O algoritmo do Minimax implementado pode ser expresso pelo Algoritmo 3, auxiliado pela função de aplicação do Minimax (esta, sim, em profundidade), demonstrada no Algoritmo 4.

Algoritmo 3 Minimax

```
1:  raiz ← tabuleiroInicial
2:  fila ← {raiz}
3:  while fila ≠ ∅ do
4:      tabuleiro ← proximo(fila)
5:      jogadorAtual ← estado.jogador
6:      if jogadorAtual = MAX then
7:          proximoJogador ← MIN
8:      else
9:          proximoJogador ← MAX
10:     end if
11:     proximosEstados ← tabuleiros possíveis por jogadas de jogadorAtual sobre
        tabuleiro
12:     for filho ∈ proximosEstados do
13:         tabuleiro retém nó filho
14:         if ¬terminal(filho) then
15:             fila ← fila ∪ {filho}
16:         else
17:             filho.valor ← utilidade(filho)        ▷ Função custo sobre nó terminal.
18:         end if
19:     end for
20: end while
21: aplicarMinimax(raiz)                  ▷ Aplica o Minimax sobre a árvore resultante.
```

Em essência, o Algoritmo 3 expande os nós da árvore Minimax - os tabuleiros $t_1, t_2, \ldots, t_n$ filhos do tabuleiro T - em largura, auxiliado por uma fila. A expansão prossegue se, e somente se, o nó não é terminativo; nesse caso, um valor utilidade é atribuído a esse nó. A aplicação do Minimax é feita depois da derivação da árvore, desta feita em profundidade, propagando os valores máximos dos nós filhos caso o nível do nó pai seja MAX, ou os valores mínimos para nó pai MIN. Essa é a implementação seguida pela

Algoritmo 4 Aplicação do Minimax sobre uma árvore derivada

```
1: function APLICARMINIMAX(nodo)
2:     filhos ← enumerarFilhos(nodo)
3:     if filhos = ∅ then
4:         return
5:     end if
6:     for c ∈ filhos do
7:         aplicarMinimax(c)
8:     end for
9:     if jogador(nodo) = MAX then
10:        nodo.valor = max_{∀f∈filhos} f.valor
11:    else
12:        nodo.valor = min_{∀f∈filhos} f.valor
13:    end if
14: end function
```

classe $Minimax$, abordada na seção 3.3.

Como explica Russell and Norvig (2010), o Minimax puro acaba por incorrer em um número exponencial de nós, conforme cresce a profundidade da árvore. Isso acontece porque o Minimax considera todos os tabuleiros possíveis para a tomada de decisão. Entretanto, nem todos os caminhos precisam ser analisados: suponhamos que o jogador, seja MIN, seja MAX, tem a possibilidade de escolher uma jogada j que resulta em um ganho de valor v. Se, em algum nível acima do nível em que a jogada j está localizada, o jogador tem a opção de selecionar uma jogada j' com valor v', tal que $v' > v$, então o nó (e eventualmente toda a subárvore da qual ele é raiz) pode ser podado, i.e., não precisa ser expandido.

A Poda α-β ataca exatamente o ponto da expansão desnecessária dos nós, reduzindo de forma dramática o espaço de busca do Minimax, sem afetar na escolha final da IA, ou seja, uma jogada resultante da aplicação da Poda α-β é a mesma do Minimax puro. O percurso de expansão em profundidade, aplicado pela classe $AlphaBetaSearch$ mostrada na seção 3.3, se caracteriza pela inclusão de duas informações-chave para a realização da poda:

- α**:** o maior valor utilidade de um nó encontrado até um determinado momento da busca quando o jogador é MAX.
- β**:** o menor valor utilidade de um nó, analogamente a α, que representa, com efeito, a melhor escolha para o jogador MIN.

Note-se, ainda, que a implementação de $AlphaBetaSearch$, diferentemente do exposto no Minimax concretizado pelos Algoritmos 3 e 4, é único. Isto é, a expansão da

árvore e a propagação efetiva dos valores $\min$ e $\max$ para os nós não folha da árvore são feitos de uma só vez. Essa é a abordagem mais convencional, próxima do que Russell and Norvig (2010) ensina, e está expressa no Algoritmo 5. Dado um tabuleiro inicial de jogo, representado pelo nó raiz r da árvore Minimax, e que o jogo inicia por MAX, o Algoritmo 5 pode ser chamado com o mais baixo valor possível para α e o mais alto para β: $alphaBetaSearch(r, MAX, -\infty, +\infty)$.

Algoritmo 5 Minimax com Poda α-β

```
1:  function ALPHABETASEARCH(nodo, jogadorAtual, α, β)
2:      if terminal(nodo) then
3:          return utilidade(nodo)
4:      end if
5:      nodo.valor ← −∞ Se jogadorAtual = MAX Senão +∞
6:      proximoJogador ← MAX Se jogadorAtual = MIN Senão MAX
7:      proximosEstados ← tabuleiros possíveis por jogadas de jogadorAtual
8:      for filho ∈ proximosEstados do
9:          nodo retém nó filho
10:         if jogadorAtual = MAX then
11:             nodo.valor ← max(nodo.valor, alphaBetaSearch(filho, proximoJogador, α, β))
12:             if nodo.valor ≥ β then
13:                 return nodo.valor                                  ▷ Poda.
14:             end if
15:             α ← max(α, nodo.valor)
16:         else                                                       ▷ JogadorMIN
17:             nodo.valor ← min(nodo.valor, proximoJogador, α, β)
18:             if nodo.valor ≤ α then
19:                 return nodo.valor                                  ▷ Poda.
20:             end if
21:             β ← min(β, nodo.valor)
22:         end if
23:     end for
24: end function
```

Os Algoritmos 3 e 5 referenciam a função $utilidade$, a qual implicitamente define um valor para o nó. Movimentos feitos em um jogo de tabuleiro, na dinâmica do Minimax, podem ter três resultados possíveis: empate, vitória (movimento de MAX que encerra o jogo) e derrota (movimento de MIN que encerra o jogo). Nesse sentido, a função de valor $utilidade$ pode ter uma saída ternária, em números inteiros que têm a seguinte semântica, exposta no algoritmo 6.

- 0: empate.
- 1: vitória de MAX.
- -1: vitória de MIN.

Algoritmo 6 Função custo (utilidade) de um nó da árvore Minimax

```
1: function UTILIDADE(nodo)
2:     if nodo representa tabuleiro com vitória de MAX then
3:         return 1
4:     else if nodo representa tabuleir com vitória de MIN then
5:         return −1
6:     else
7:         return 0                                          ▷ Empate.
8:     end if
9: end function
```

Esses valores são atribuídos de forma criteriosa a qualquer nó da árvore Minimax, independente da profundidade em que esse nó se encontra. De fato, para um jogo em que dois jogadores se confrontam alternadamente, se o jogo está posicionado sobre a profundidade p da árvore Minimax, então, assumindo que o nó representativo do tabuleiro inicial do jogo está no nível 1, $p - 1$ movimentos foram feitos por ambos jogadores. É fácil de ver, portanto, que p é diretamente proporcional ao tempo de jogo: quanto maior for o escalar p, mais jogadas precisaram ser feitas por cada jogador.

Essa observação, conquanto trivial, traz consigo a a noção de que, se um jogador busca vencer o mais rapidamente possível, então esse mesmo jogador irá buscar nós da árvore que têm função custo positiva a uma pequena profundidade. Logo, nós terminais não nulos mais próximos à raiz têm maior valor que nós mais distantes, pois, além de levarem o jogador a vitória, encerram a partida mais rapidamente. Dessa forma, a função custo ternária pode ser ligeiramente modificada para contemplar valores maiores para nós terminais não nulos, ponderando essa atribuição de acordo com o nível de profundidade, como mostra o Algoritmo 7.

Algoritmo 7 Função custo (utilidade) ponderada por nível de profundidade da árvore Minimax.

```
1: function UTILIDADEPONDERADA(nodo, profundidadeAtingida)
2:     valor ← utilidade(nodo)
3:     if valor = 0 then
4:         return valor
5:     end if
6:     valorPonderado ← profundidadeAtingida − nivel(nodo) + 1
7:     return valorPonderado × valor   ▷ Se MIN vence, negativo; se MAX vence, positivo.
8:
9: end function
```

A noção de função custo ternária ponderada por nível introduz um potencial re-

curso para a Inteligência Artificial. Na eventualidade da inclusão da funcionalidade de ajuste de nível de dificuldade, a busca da IA, seja por Minimax, seja por Poda α-β, pode automaticamente focar na magnitudade dos valores da função custo, objetivando ganhos grandes para um nível de dificuldade maior e ganhos menores para uma IA menos sequiosa da vitória. De fato, a ponderação por profundidade da árvore compensa a limitação de um conjunto $\{-1, 0, 1\}$ para a função utilidade, conferindo mais informações para a decisão da IA.

Idealmente, a árvore do Minimax poderia ser completamente expandida. Todavia, como frisado por Russell and Norvig (2010), o potencial de explosão combinatória é alarmente. Uma solução bastante simples, embora de enorme ganho, é limitar a profundidade da árvore: inicialmente, são expandidos nós até a profundidade p; se o jogo progredir ao ponto de a IA atingir um estado folha, assume-se que é necessário expandir mais p níveis da árvore para que o Minimax retorne um novo movimento. Isso não sacrifica a agilidade da Inteligência Artificial em vencer uma partida, visto que, dado p suficientemente grande, dificilmente o jogo irá ser encerrado tão prematuramente e eventuais cenários de derrota da IA podem ser contornados.

A limitação da profundidade é uma forma natural de garantir bons resultados com considerável desempenho, i.e., tomada rápida de decisão por parte da IA. A performance do Minimax, especialmente inserido em um contexto maior de sistema de múltiplos jogos abstratos e de estratégia, é uma questão essencial. É indesejado que o sistema demore um tempo excessivamente longo para a tomada de decisão, logo, não apenas otimizações pontuais como a limitação de profundidade e a introdução da Poda α-β devem ser consideradas: as estruturas de dados usadas na construção dos algoritmos devem ser criteriosamente escolhidas.

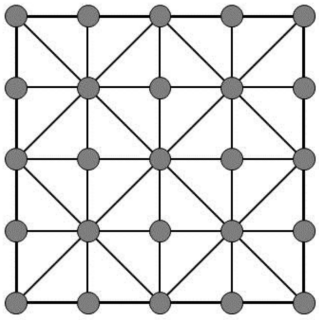
3.5 Estruturas de dados aplicadas na representação dos jogos

De todas as estruturas pertinentes aos jogos abstratos de estratégia, o tabuleiro é, certamente, aquele que primeiro suscita as discussões a respeito de quais estruturas de dados utilizar. Afinal de contas, a escolha de qual estrutura visa a apontar não apenas as vantagens e desvantagens na implementação de novos jogos ou a camada de visualização para interação com o usuário (por princípio, ausente neste trabalho). Em função da potencial explosão combinatória inerente ao Minimax, discutida na Seção 3.4, existe uma preocupação natural com a performance que o sistema vai ter a cada rodada do jogo, seja

na execução de um movimento, seja na expansão da árvore do Minimax.

A proposta de tabuleiro feita por Ribas (2020) consiste em uma estrutura matricial quadrada, de dimensões 5×5, na qual existe o potencial de exploração, portanto, de 25 posições para inserção e deslocamento de peças. Ora, essa sucinta descrição traz consigo uma resposta para a pergunta de qual estrutura de dados escolher para respresentar o tabuleiro: uma matriz bidimensional, claramente suscitada pela Figura 3.4

Figura 3.4: Tabuleiro matricial 5×5 (25 posições jogáveis).



Fonte: Ribas (2020)

A intuitividade com que a estrutura de dados da matriz aparece é evidente: posições de jogo são trivialmente indexadas a partir da linha i e da coluna j. A performance é considerável, haja visto que as linguagens de programação comumente relacionam vetores e matrizes como tipos de dados primitivos e os compiladores e interpretadores são especialmente otimizados para contemplar a indexação vetorial e matricial (AHO et al., 2006). Isso torna, em particular, o sistema bastante simples no tangível a execução de movimentos e identificação de características do tabuleiro, como, por exemplo, alinhamentos e contagem de quantos passos foram dados em cada jogada. Ademais, a matriz apresenta uma geometria rígida, fácil de representar na camada de visualização.

No entanto, a implementação do tabuleiro como matriz pode trazer alguns inconvenientes. Em especial, a dificuldade em flexibilizar a geometria se, por exemplo, for sugerida a implementação de um jogo cujo tabuleiro é circular. Além disso, as relações entre as posições, de fato, as arestas do tabuleiro, são implicitamente definidas em função das coordenadas (i, j) do tabuleiro, sempre respeitando os limites da matriz. Logo, relacionar uma posição (i, j) com outra posição (i', j'), tal que $|i - i'| \geq 1 \vee |j - j'| \geq 1$, é contraintuitivo e deve ser prevista na implementação particular do jogo, considerando a modelagem proposta.

Tais contratempos observadas na aplicação da matriz como estrutura de dados do tabuleiro podem ser corrigidas pelo grafo. Um grafo $G = (V, E)$, enquanto tabuleiro de jogos abstratos de estratégia, tem no conjunto de vértices V as posições sobre as quais

os jogadores posicionam as peças, e no conjunto de arestas E a conectividade entre essas posições. Enquanto meio para um fim, um grafo é tão bem-sucedido quanto a matriz e resolve adequadamente o problema da representação do tabuleiro.

A principal vantagem do grafo é na relação entre os vértices, i.e., as posições de jogo. Enquanto um ciclo no tabuleiro exigiria alguma carga algorítmica na matriz, ainda que simples, no grafo seria absolutamente trivial. Além disso, a forma geométrica do tabuleiro pode ser flexibilizada ao arbítrio do jogo que se almeja, desde que a modelagem OO contemple tal facilidade.

Sob a visão de perda, o grafo, especialmente se estruturado em listas encadeadas de adjacências entre os nós para a construção das arestas, é bastante mais ineficiente que a matriz. Mais elegante, sim, mas menos performático. Ademais, a necessidade de a modelagem prever propriedades e classes que consigam abstrair qual forma geométrica adotar para o tabuleiro acaba por reforçar a maior complexidade do grafo, não apenas de tempo, mas também de simplicidade na implementação e descrição dos jogos dentro do sistema.

Idealmente, no entanto, o grafo em lista ou mesmo matriz de adjacência pode ser levemente alterado para conter os benefícios da matriz. Os vértices, ou posições do tabuleiro, podem ser classes na qual há propriedades que identifiquem as coordenadas de linha e coluna de uma posição no espaço do tabuleiro. A indexação dar-se-ia, portanto, por meio de métodos que iteram sobre os vértices do grafo até que a posição seja encontrada; perde na eficiência da estrutura matricial primitiva, já bastante otimizada, contudo, mesmo essa abordagem tem boa margem de ganho por algoritmos de busca eficientes, como uma busca binária. Embora, em realidade, um tabuleiro seja pequeno, o que faz o impacto da busca linear marginal.

No entanto, os benefícios e simplicidade de implementação da matriz acabaram por torná-la a preferida para a implementação deste sistema. Dado o baixo acoplamento do sistema, substituir a estrutura de dados elementar do tabuleiro não incorreria em grades investidas de refatoração, logo, a matriz pode ser preterida pelo grafo caso a escalabilidade se mostre um problema. Dentro da proposta dos 21 Jogos abordados por Ribas (2020), no entanto, a matriz é mais do que satisfatória e apresenta bons resultados tanto em implementação quanto em tempo de execução.

Outra estrutura fundamental do sistema que requer alguma atenção na escolha envolve a árvore do Minimax. Esta, por definição, é uma árvore de aridade n, isto é, cada nó pode ter um número $n \geq 0$ de nós filhos, recursivamente. Nesse sentido, não

há muito que se possa argumentar no escopo deste trabalho: um nó considerado como raiz é o ponto de partida do percurso, e os descendentes desse nó são varridos de forma recursiva ou com o auxílio de filas (para expansão em largura) ou pilhas (para expansão em profundidade).

Entretanto, a própria definição do nó demanda a escolha de como enumerar os nós filhos. Ignorando qualquer heurística que se sugira para a organização dos descendentes imediatos de um nó, a questão passa a ser qual a estrutura mais adequada para a retenção dos nós filhos. Considerando, em especial, que a árvore pode ter um tamanho considerável em memória e que, via de regra, os jogos abstratos implementados não têm tantos estados possíveis como o xadrez, por exemplo.

Trivialmente, uma lista é usada para armazenar os nós filhos. A questão que se apresenta, então, é: qual abordagem de lista utilizar? Encadeamento ou vetor são os mais proeminentes candidatos. A lista encadeada é facilmente estendida, podendo receber novos elementos a um custo desprezível; em contrapartida, a lista vetorial requer um redimensionamento (variável de acordo com a estratégia de implementação, por exemplo, se com uma capacidade inicial fixa).

O redimensionamento da lista vetorial não é um problema particularmente difícil. É, todavia, bastante custoso: para uma lista vetorial de k posições, a inserção do elemento $k + 1$ requer a alocação de um novo vetor de dimensão $k' \geq k + 1$, a cópia de todos os elementos nas posições $1, \ldots, k$ e a inserção do elemento em $k + 1$. Ainda que uma capacidade inicial seja definida, a variabilidade do número de estados possíveis pode oscilar grandemente de jogo para jogo, o que pode incorrer em dois cenários indesejados: a subutilização da memória alocada (capacidade inicial demasiado grande) ou o constante redimensionamento vetorial (uso maior de memória e processamento).

Com os elementos filhos são iterados de acordo com alguns atributos, como o movimento do jogador que origina o nó filho (jogada de MIN) ou o nó de maior valor utilidade (jogada de MAX), pouco benefício traz a indexação eficiente da lista vetorial. Logo, os filhos de um nó da árvore Minimax podem ser implementados por lista encadeada, em que a inserção é bastante mais eficiente que a lista vetorial, apesar da penalidade no percurso sobre os elementos. Logo, o encadeamento é preferido para a implementação Minimax pela facilidade de inserção, dada a desnecessidade de mecanismos de redimensionamento, como visto na lista vetorial.

Toda a discussão que envolve as escolhas tomadas no projeto de um sistema de jogos abstratos de estratégia com Inteligência Artificial se torna mais plausível na análise

da implementação de alguns jogos. É imprescindível, neste momento, que a fronteira teórica seja cruzada, a fim de que todo o modelo e as escolhas aqui discutidos sejam postos à prova.

4 IMPLEMENTAÇÕES

4.1 A abordagem da implementação

Um sistema de jogos de tabuleiro com suporte a Inteligência Artificial tem um grande apelo prático. A própria ideia é por si um convite à programação, pela considerável dinâmica e escolhas de projeto. Quando aliados ao potencial da orientação a objetos e aos desafios que envolvem os Algoritmos de Decisão Competitiva, a implementação passa a ser uma escolha natural.

Apesar do anseio prático despertado pela proposta do trabalho, é importante notar que a prática não pode sacrificar as características que norteiam o projeto. Isto é, a escalabilidade, ponto-chave para que o sistema seja versátil na construção de novos e diferentes jogos de tabuleiro, não pode ser posta de lado. Nem tampouco deve-se incorrer em grandes simplificações da Inteligência Artificial, sob pena de combalir um dos pilares do sistema.

Contudo, podem surgir suspeitas nada imerecidas a respeito do quão evasiva a escalabilidade surge nesse contexto. Sim, entende-se bem que o desejo é uma significativa abrangência dos jogos de tabuleiros e que a Inteligência Artificial deve se fazer presente e absoluta em todo e qualquer jogo que venha a ser suportado pelo sistema. A questão que se impõe passa a ser, então, qual o destino da escalada? Quão ambiciosos e quais são os jogos visados para a concretização do sistema?

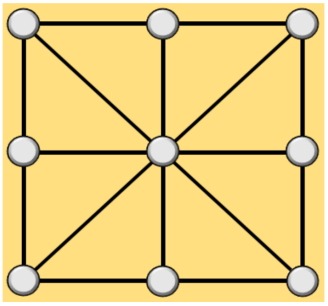
Os 21 jogos abstratos de estratégia apresentados por Ribas (2020) vêm ao auxílio para responder a esses questionamentos. Nesse trabalho estão descritos jogos de variadas categorias - posicionamento, caça, alinhamento, dentro outros -, restritos a um mesmo tabuleiro. Cobre diferentes estratégias e formas de pensar, ao mesmo tempo em que representa um excelente guia para um sistema computacional de jogos de tabuleiro.

A despeito da base fornecida por Ribas (2020), uma abordagem preliminar de grande utilidade é o protótipo. No contexto deste trabalho, o protótipo tem a função mestra de abrir caminho para uma solução maior e mais abrangente e, ao mesmo tempo, expor os principais conceitos de jogos abstratos e de estratégia a serem observados na implementação. E, acima de tudo, não deixar de manter a implementação flexível e receptiva ao advento de novos jogos.

4.2 Protótipo: o Tapatan

O Tapatan, conquanto ausente em sua forma mais pura dos 21 jogos presentes em Ribas (2020), é um excelente ponto de partida. Por ser um jogo simples e direto, poucos problemas especificamente relacionados ao jogo podem ser antevistos. O Tapatan é, portanto, o pano de fundo ideal para o desenho inicial de uma plataforma digital de jogos de tabuleiro com suporte a Inteligência Artificial.

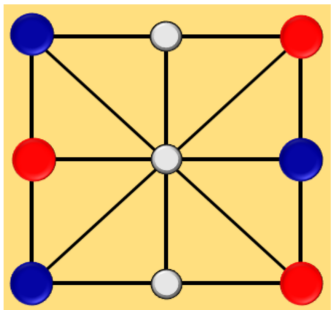
Figura 4.1: Tabuleiro do Tapatan.



Fonte: Giordani and Ribas (2014)

Jogado sobre um tabuleiro matricial 3×3, visto na Figura 4.1, o Tapatan determina que cada um dos dois jogadores tenha 3 peças. O tabuleiro inicial tem todas as peças dos dois jogadores dispostas alternadamente nas extremidades laterais, como mostra a Figura 4.2. Cada jogador pode deslocar uma peça por jogada com movimentos horizontais, verticais e diagonais, desde que feitos sobre as arestas do tabuleiro; além disso, os movimentos devem ser unitários, i.e., as peças se deslocam apenas uma casa. O vencedor é aquele que conseguir alinhar 3 peças suas na horizontal, na vertical ou nas diagonais.

Figura 4.2: Posição inicial do Tapatan.



Fonte: Giordani and Ribas (2014)

O tabuleiro do Tapatan é representado por uma matriz 3×3. De fato, a estrutura de dados utilizada é um vetor – ou *array* – que possui três elementos, em que cada elemento é também um vetor com três componentes; cada elemento desses vetores internos é uma

instância de um objeto $Piece$. Como discutido na Seção 3.3, a classe $Piece$ representa uma peça do tabuleiro e é informalmente identificada pela posição que ocupa; a peça possui associado uma instância de um objeto $Player$, o jogador, o qual é visto como o dono da peça em uma determinada fotografia do tabuleiro. Essa representação tem uma particularidade: ao instanciar um tabuleiro do Tapatan, as peças serão instanciadas em posição inalterável; a movimentação, assim, é feita não pela alteração da coordenada $(linha, coluna)$ da peça no tabuleiro, mas pela transferência de propriedade da peça de um jogador para outro, com possibilidade de nulidade para o dono da peça – nesse caso, considera-se a posição do tabuleiro vazia.

Importante ressaltar que a abstração de tabuleiros detalhada na seção 3.3, em que a classe $AbstractBoard$ é o ancestral de todo e qualquer tabuleiro da sub-hierarquia, ainda não tinha sido prevista em tempo de protótipo. Logo, o tabuleiro do Tapatan foi encapsulado numa classe final, não extensível, chamada $TapatanBoard$ (Código 4.1), que antecipa praticamente todos os métodos e propriedades públicas de $AbstractBoard$: a $boardMatrix$ para o tabuleiro, o método $play$, um método para enumeração de possíveis jogadas para um determinado jogador e a validação de um movimento candidato.

Seja T uma instância do tabuleiro do Tapatan. T é um vetor de vetores de instância de $Piece$ possivelmente nulas:

$$T = \begin{bmatrix} \begin{bmatrix} p_{0,0} : Piece & p_{0,1} : Piece & p_{0,2} : Piece \end{bmatrix} \\ \begin{bmatrix} p_{1,0} : Piece & p_{1,1} : Piece & p_{1,2} : Piece \end{bmatrix} \\ \begin{bmatrix} p_{2,0} : Piece & p_{2,1} : Piece & p_{2,2} : Piece \end{bmatrix} \end{bmatrix}$$

É oportuno observar que as posições de cada instância de $Piece$ (propriedade $position$, um par ordenado de inteiros) são atribuídas como indexação matricial em intervalo $[0, n)$: a primeira peça tem posição $(0, 0)$, i.e., primeira linha e primeira coluna, e a última peça tem posição $(2, 2)$, última linha e última coluna. A estrutura vetorial do tabuleiro é encapsulada pela classe $TapatanBoard$ na propriedade $boardMatrix$. A instanciação de um objeto $TapatanBoard$ é feita impreterivelmente com duas instâncias de objetos $Player$, ou seja, o tabuleiro está limitado exatamente por dois jogadores. O tabuleiro é, assim, dinamicamente construído obedecendo a disposição inicial clássica do Tapatan: peças alternadas por jogador com a coluna central livre. O aspecto dinâmico é essencial, pois, apesar da disposição inicial ser fixa nessa implementação, é permitida uma inicialização arbitrária do tabuleiro pela invocação do método $buildBoardRow$, para cada vetor v contido em T.

Código 4.1: Classe $TapatanBoard$.

```
class TapatanBoard(
    private val player1: Player,
    private val player2: Player,
    _boardHeight: Int = 3
) : Cloneable {
    val boardMatrix = arrayOf(
        buildBoardRow(0, _boardHeight, player1),
        buildBoardRow(1, _boardHeight, player2),
        buildBoardRow(2, _boardHeight, player1)
    )

    fun play(player: Player, movement: PlayerMovement, simulate: Boolean = false) {
        /* ... */
    }
    fun calculateReachingMovements(origin: Pair<Int, Int>): Iterable<Pair<Int, Int>> {
        /* ... */
    }
    fun isMovementValid(movement: PlayerMovement): Boolean { /* ... */ }
    fun searchAlignment(): BoardAlignment? { /* ... */ }

    override fun clone(): TapatanBoard { /* ... */ }

    private fun buildBoardRow(
        rowIndex: Int,
        size: Int,
        firstPlayer: Player
    ): Array<Piece> { /* ... */ }
}
```

A concretização das regras de jogo inicia pelo método $play$, responsável por executar uma movimentação de um jogador no tabuleiro. Auxiliado pelo método *isMovementValid*, $play$ faz valer as premissas do Tapatan: jogadas de passos unitários em linha, coluna e diagonais principal e secundária. Logo, se o movimento satisfizer todas as restrições do Tapatan, $play$ transfere a propriedade da peça de destino para o jogador (instância de $Piece$ anteriormente nula), ao mesmo tempo em que deixa disponível a propriedade da peça de origem, i.e., ocupa uma posição do tabuleiro liberando outra. Se, entretanto, for definido que o movimento é apenas uma simulação, então a transferência de propriedade da peça de destino e a desapropriação da peça de origem não são feitas – tem-se, assim, apenas um teste de qual o resultado de uma jogada sem que se comprometa o estado atual do tabuleiro, caso o movimento seja factível.

Para uma determinada posição do tabuleiro, o método *calculateReachingMovements* é responsável por enumerar quais as possíveis coordenadas nas quais se pode chegar. O cálculo das posições atingíveis é feito de modo descompromissado, isto é, ig-

norando o fato de haver alguma peça em uma posição aceitável de destino. Em suma: $calculateReachingMovements$ define as regras de movimentação do Tapatan.

Por fim, de uma forma um tanto implícita, o método $searchAlignment$ informa se houve fim de jogo. Essa informação, todavia, deve ser interpretada pelo objeto consumidor de uma instância de $TapatanBoard$ (um controlador de jogo, por exemplo); ativa ou explicitamente, uma instância de $TapatanBoard$ não declara fim de jogo: se $searchAlignment$ detectar um alinhamento sobre o tabuleiro matricial, então há um vencedor a ser designado. Com efeito, ainda que conveniente, essa abstração é pouco objetiva e nada diz sobre como um tabuleiro pode ter um vencedor declarado; um objeto consumidor de $TapatanBoard$ deve interpretar o resultado de $searchAlignment$ e considerar que um alinhamento liquidou uma partida. Inexiste, de fato, uma regra de fim de jogo materializada; para tal, poder-se-ia convencionar um método $isOver$ ou até mesmo uma $callbackfunction$ que notificasse um ouvinte sobre o encerramento da partida. Ademais, a classe $TapatanBoard$ permite movimentação de peças mesmo que haja um alinhamento no tabuleiro.

O Minimax puro representa a Inteligência Artifical no protótipo do Tapatan. A árvore do Minimax tem como nodo estrutural um objeto que retém as informações de qual o jogador do nível (MIN ou MAX), o movimento que originou o nodo, o estado do tabuleiro para o qual o nodo foi definido, o valor efetivo do nodo (conforme a convenção da função de valor discutida no Seção 3.4) e o conjunto de filhos do nó como referência para a recursão. É, pois, uma versão preliminar de $TreeNode$ que pode ser vista no Código 4.2.

A classe $Minimax$ (Código 4.3) encapsula todas as operações do algoritmo Minimax em sua forma mais típica. Oportunamente, é possível tanto construir uma árvore nova a partir de um tabuleiro inicial – método $buildTree$ – quanto expandir mais nós de uma árvore já existente – método $demandMoreTreeLevels$. Esse recurso é particularmente importante, pois expandir toda a árvore Minimax do Tapatan é inviável, visto que o jogo pode durar eternamente a depender das jogadas. Logo, a limitação da profundidade como forma de ponto de parada da expansão da árvore naturalmente levanta a necessidade de um método que retome a expansão a partir do posicionamento do jogo em um estado (nó) da árvore. Para que um objeto Minimax seja construído são requeridos dois jogadores, fundamentais para a derivação da árvore, seja na definição dos níveis (se tabuleiros de escolha de MIN ou MAX), seja na geração do nó raiz da árvore, dado que MIN e MAX têm diferentes possibilidades de jogada. Assim, a classe Minimax não restringe o

jogador que dá início a partida.

Código 4.2: Classe $TreeNode$ do protótipo Tapatan.

```
data class TreeNode(
    val player: Player,
    val generatingMovement: PlayerMovement?,
    val board: TapatanBoard,
    var value: Double = 0.toDouble(),
    val children: MutableList<TreeNode> = mutableListOf(),
    val level: Long
) {
    fun isLeaf() = children.isEmpty()
}
```

O algoritmo de construção da árvore descrito para a IA do Tapatan corresponde ao Algoritmo 3, desconsiderando as acomodações pertinentes à implementação na linguagem Kotlin. Os nodos são, portanto, expandidos utilizando a abordagem BFS (*Breadth-First Search*) e efetivamente instanciados, com especial importância, a partir da instância de $TapatanBoard$, a qual encapsula tanto o estado do jogo no ponto exato em que o nodo se encontra quanto o movimento gerador deste tabuleiro. O movimento gerador permite identificar qual peça teve transferência de propriedade e qual a nova peça vaga a receber um novo dono (origem e destino). Essa abordagem é útil não apenas sob o ponto de vista da compreensão do algoritmo, mas também porque facilita enormemente a busca por uma jogada originária do usuário: quando este fornece as coordenadas de movimento de origem e destino, é suficiente, dado um nodo sobre o qual o jogo está correntemente posicionado, encontrar um filho deste que contenha a propriedade de movimento gerador do nó, a $generatingMovement$, exatamente igual às coordenadas escolhidas pelo usuário. Considerando que o algoritmo proposto exaure todos os tabuleiros para os quais o jogador pode empurrar a partida, assumindo a entrada do usuário como válida, deve existir um nó com $generatingMovement$ correspondente às coordenadas da jogada. Isso é uma premissa inarredável do algoritmo: se a jogada é válida e o jogador corresponde ao nível da raiz corrente da árvore Minimax, então existe um filho da raiz que possui um $generatingMovement$ correspondente à jogada.

A função custo se caracteriza pelos resultados pertencentes ao conjunto ternário de valores abordado na seção 3.4. No caso do Tapatan, portanto, a utilidade de um nó é -1 se há um alinhamento de 3 peças do jogador MIN, 0 em caso de empate e 1 para tabuleiros com alinhamento de 3 peças do jogador MAX. Como se nota, a função de valor não é elástica para permitir a flexibilização do nível de dificuldade da IA. Como esta procura sempre maximizar sua jogada mantendo uma perspectiva pessimista em relação

ao passo a ser tomado pelo adversário, é virtualmente impossível o usuário vencer uma partida; empiricamente, no melhor caso, o jogo pode se estender indefinidamente – o que indicaria um empate.

Código 4.3: Definição da classe $Minimax$ para o protótipo Tapatan.

```
class Minimax(
    private val startPlayer: Player,
    private val otherPlayer: Player
) {
    fun buildTree(kickoffBoard: TapatanBoard, depth: Long = Long.MAX_VALUE): TreeNode {
        /* ... */
    }

    fun demandMoreTreeLevels(root: TreeNode, depth: Long) { /* ... */ }

    fun selectMostValuableNode(root: TreeNode): TreeNode? { /* ... */ }

    fun selectNodeWithMatchingMovement(
        root: TreeNode,
        movement: PlayerMovement
    ): TreeNode? { /* ... */ }

    private fun applyMinimax(node: TreeNode) { /* ... */ }
}
```

Um ponto interessante de ênfase é a retenção dos estados do tabuleiro do Tapatan nos nós da árvore Minimax. Por construção, Kotlin, analogamente a Java, faz uso irrestrito de atribuição por referência. Na expansão dos nodos da árvore Minimax é fundamental que se execute uma jogada para alterar o estado do tabuleiro que originou o nodo. Ao fazê-lo, se o nodo que estiver sendo expandido tiver como tabuleiro uma instância que referencie a instância do tabuleiro do nó pai, por atribuição, a execução do método $play$ sobre o tabuleiro retido pelo nó filho irá também alterar a matriz que representa o tabuleiro do nodo pai. Nesse caso, a árvore do Minimax terá tabuleiros degenerados e será, portanto, inútil.

Como alternativa, as classes que circundam o tabuleiro ($Piece$ para a peça e a própria $TapatanBoard$) implementam um método para a criação de uma cópia profunda, ou, no jargão Kotlin-Java, um clone. A cópia profunda não faz referência a qualquer outra instância: em vez disso, um novo objeto é alocado em memória e todos os campos e propriedades são inicializados com valores igualmente desprovidos de referência. Desse modo, tanto as peças (que podem ser degeneradas na transferência de propriedade, que abstrai a movimentação) quanto o tabuleiro são únicos, a nível de instância de objeto, para cada nodo, a fim de assegurar a não interferência. Conquanto efetiva, essa solução

incorre e um substancial impacto no consumo de memória e consequentemente no tempo de execução da derivação da árvore Minimax.

A interação entre a Inteligência Artificial e um usuário pode ser feita pela codificação de uma controladora de jogo. Essa controladora deve, alternadamente, requisitar jogadas do usuário (pela entrada do teclado, por exemplo), realizá-las, e em seguida invocar a Inteligência Artificial Minimax para que esta forneça uma jogada. O tabuleiro é visível para o usuário e o jogo termina quando um 3-alinhamento for detectado no tabuleiro. O Algoritmo 8 sintetiza o algoritmo implementado para uma controladora IA vs. Usuário.

Algoritmo 8 Tapatan: implementação de uma controladora de jogo para IA vs. Usuário.

1: $jogadorMax \leftarrow$ IA
2: $jogadorMin \leftarrow$ Usuário
3: $jogadorInicio \leftarrow$ jogador inicial por entrada do usuário
4: $jogadorAtual \leftarrow jogadorInicio$
5: $tapatan \leftarrow$ tabuleiro do Tapatan com jogadores $jogadorMax$ e $jogadorMin$
6: $raizMinimax \leftarrow$ árvore Minimax com profundidade k
7: **while** $tapatan$ não apresenta um 3-alinhamento **do**
8: Mostra o tabuleiro para o usuário
9: **if** $raizMinimax$ é folha **then**
10: $raizMinimax \leftarrow raizMinimax$ expandida em mais k níveis
11: **end if**
12: **if** $jogadorAtual = jogadorMax$ **then**
13: $movimento \leftarrow$ melhor jogada filha de $raizMinimax$
14: $jogadorAtual = jogadorMin$
15: **else**
16: $movimento \leftarrow$ entrada válida do usuário para jogada
17: $raizMinimax \leftarrow$ filho de $raizMinimax$ com movimento gerador $movimento$
18: $jogadorAtual = jogadorMax$
19: **end if**
20: $tabuleiro \leftarrow tabuleiro$ alterado por jogada $movimento$
21: **end while**
22: $vencedor \leftarrow$ jogador que realizou o 3-alinhamento
23: Sinaliza fim de jogo com vencedor $vencedor$

Ainda que o protótipo apresente algumas cruezas, especialmente nas abstrações, a implementação do Tapatan demonstrou que uma plataforma digital de jogos abstratos e de estratégia, com suporte a Inteligência Artificial, é viável. Muitas estruturas idealizadas para o Tapatan serviram de base para a modelagem orientada a objetos detalhada na Seção 3.3, ainda que refinadas e adaptadas a um contexto mais generalista. Além disso, a experiência adquirida ao longo do desenvolvimento, em especial aquelas adquiridas pela

construção do algoritmo Minimax, foram fundamentais para a projeção de estruturas e conceitos mais receptivos às ideias introduzidas por novos jogos que seriam programados futuramente.

No estágio do protótipo do Tapatan, o jogo, como presumido, não foi uma fonte de desafios por si só. Em vez disso, modelar as estruturas que envolvem o jogo, objetivando a flexibilidade, foi muito mais desafiador. Embora se note alguma penalidade em, por exemplo, referenciar por todo o código-fonte uma classe chamada $TapatanBoard$ e, portanto, tornar explícito o jogo sobre o qual o motor da IA e o usuário estão atuando, tal refatoração não representa por si um esforço hercúleo. O pequeno número de classes e o escopo por ora específico do projeto certamente suavizam o impacto da revisão da modelagem e, consequentemente, da reescrita de código.

A grande incitação à criatividade veio, de fato, da necessidade de construção da Inteligência Artificial. Algumas particularidades, como a retenção de estados (tabuleiros) nos nós da árvore, passam a ter uma dimensão maior e mais importante a nível de implementação do que quando analisamos o Minimax didaticamente. As decisões que devem ser tomadas - no caso do protótipo do Tapatan, retenção de estado por cópia profunda das instâncias de objetos - procuram, num primeiro momento, solucionar a contento o problema. No entanto, a antecipação de que existe espaço para melhorias é constante, e se a princípio se aceita esta ou aquela solução, não quer dizer que a estrutura adotada nos estágios iniciais do projeto serão imutáveis.

4.3 Implementação: Jogo-da-Velha 4

O primeiro dos 21 jogos da série apresentada por Ribas (2020) foi o selecionado para a primeira implementação após o protótipo do Tapatan. O apelo do Jogo-da-Velha 4 é dado, qual o Tapatan, pela simplicidade do jogo e, ao mesmo tempo, pela sensível mudança nas regras de jogo. O Jogo-da-Velha-4 demanda, por exemplo, um tabuleiro maior que o 3×3 matricial do Tapatan; além disso, tem a missão de colocar à prova a estrutura projetada em tempo de protótipo.

Jogado sobre o tabuleiro matricial 5×5 onipresente nos jogos abordados por Ribas (2020), o Jogo-da-Velha 4 apresenta um tabuleiro inicialmente vazio. Cada jogador deve, alternadamente, inserir peças em posições livres, sendo esta a única restrição dada durante as jogadas. O fim de jogo é sinalizado quando um dos jogadores forma um alinhamento de 4 peças na diagonal, vertical ou horizontal, desde que feito sobre uma das arestas (linhas)

do tabuleiro.

O primeiro desejo de alteração surge no aumento das dimensões do tabuleiro. Contudo, expandir as dimensões de uma matriz, além de trivial, é apenas parte do problema: o Jogo-da-Velha-4 deve ser incorporado ao sistema mantendo o suporte ao que já existe, nesse caso, o Tapatan. Aqui fica claro que é insustentável manter uma referência direta entre as classes $Minimax$, que representa a Inteligência Artificial, e a $TapatanBoard$, a qual implementa o jogo do Tapatan. Em suma, o mesmo código que é executado pela IA deve ser capaz de jogar tanto o Tapatan quanto o Jogo-da-Velha-4.

O problema que se apresenta é: dados dois jogos, de que forma encapsular e abstrair ambos para que o Minimax possa ser aplicado indistintamente sobre eles? Sob a ótica de execução de um jogo de tabuleiro, podemos perceber que os dois jogos compartilham algumas operações e propriedades em comum, independentes de como o jogo funciona:

- **Jogadores:** os dois jogadores que compõem uma partida.
- **Tabuleiro:** a estrutura física sobre a qual as jogadas são feitas e as regras de movimentação e fim de jogo são aplicadas.
- **Jogar:** permitir a movimentação de peças sobre o tabuleiro, dado um movimento de um jogador.
- **Enumerar movimentos permitidos:** em um determinado estado do tabuleiro, um jogador é capaz de executar uma quantidade finita e discreta de movimentos. Enumerar esses movimentos equivale a oferecer as opções de jogadas já com as restrições impostas pelas regras do jogo.
- **Validar:** indicar se uma intenção de movimento é válida.
- **Fim de jogo:** decretar fim de jogo e qual jogador é o vitorioso.

A sugestão de que esse pequeno conjunto de responsabilidades e propriedades é capaz de sintetizar um jogo tem grande apelo. As operações são simplificadas, deixando larga margem de especialização pelas implementações de outros jogos, ao mesmo tempo em que fornecem as bases para que um jogador seja mecanizado. Isto é, as operações e propriedades enumeradas fornecem aquilo que se precisa para que se construa um Minimax genérico.

Uma vez convencionado que o tabuleiro é uma estrutura matricial, pode-se permitir a parametrização dele pelas implementações de jogos. No caso do Tapatan e do Jogo-da-Velha-4, com efeito, a primeira distinção é o tamanho: o Tapatan deve definir um tabuleiro matricial de dimensões 3×3 e o Jogo-da-Velha-4 uma matriz 5×5. Como o

tabuleiro inicia, se vazio ou com peças já dispostas, é uma característica inerente a cada jogo e deve ser tratado na definição da classe que descreve esse jogo.

As jogadas, entretanto, requerem alguma atenção. O Tapatan inicia com um tabuleiro em que todas as peças dos jogadores já estão colocadas; o Jogo-da-Velha-4 começa com um tabuleiro vazio. Logo, enquanto o Tapatan prevê deslocamentos de peças, o Jogo-da-Velha-4 requer a inserção destas. Ambas operações têm o mesmo resultado líquido: ocupar uma posição. Ambas operações, porém, são conceitualmente diferentes.

A abstração de movimentos de um jogador exaurida na Seção 3.2 endereça justamente essa distinção conceitual. Um movimento pode ser especializado em dois tipos: inserção e deslocamento. Portanto, uma classe de tabuleiro deve tipificar qual movimentação os jogadores podem realizar; classificar o tipo de movimento que um jogador executa é também classificar o tipo de movimento ao qual o jogo está associado. Contextualizando, o Tapatan é um jogo com movimentos de deslocamento e o Jogo-da-Velha-4 é um jogo com movimentos de inserção; tal classificação deve igualmente se refletir nas classes que implementam os dois jogos.

As abstrações de movimento e tabuleiro exercitadas no desenvolvimento do Jogo-da-Velha-4 levam ao que foi apresentado na Seção 3.3. Isto é, admitimos a existência de um movimento - classe abstrata $PlayerMovement$ - que só é efetivamente concreto ao ser tipificado em um movimento de inserção - classe $InsertionMovement$ -, ou em um movimento de deslocamento - classe $DisplacementMovement$.

Ademais, o tabuleiro deve conter as operações e propriedades básicas enumeradas anteriormente. Logo, a classe abstrata $AbstractBoard$ assume a responsabilidade de modelar essas características. Entretanto, $AbstractBoard$ é, por si só, incompleta: faz-se necessário que $AbstractBoard$ seja genérica; usando polimorfismo paramétrico, temos $AbstractBoard < T >$, onde T é um tipo qualquer que necessariamente estende a classe $PlayerMovement$. Tem-se, como consequência, duas classes abstratas filhas de $AbstractBoard$ que definem estaticamente qual o tipo do movimento destinado ao tabuleiro: $AbstractInsertionBoard$ e $AbstractDisplacementBoard$, classes-base, respectivamente, para tabuleiros de movimentos de inserção e de deslocamento. Uma forma equivalente de representá-las é pelo polimorfismo paramétrico: $AbstractInsertionBoard$ equivale a $AbstractBoard < InsertionMovement >$ e $AbstractDisplacementBoard$ equivale a $AbstractBoard < DisplacementMovement >$ (Código 4.4).

Código 4.4: Classe $AbstractInsertionBoard$ e $AbstractDisplacementBoard$.

```
abstract class AbstractInsertionBoard(
    player1: Player,
    player2: Player,
    height: Int,
    width: Int
) : AbstractBoard<InsertionMovement>(player1, player2, height, width)

abstract class AbstractDisplacementBoard(
    player1: Player,
    player2: Player,
    height: Int,
    width: Int
) : AbstractBoard<DisplacementMovement>(player1, player2, height, width)
```

Tais hierarquizações permitem que encaixemos o Tapatan e o Jogo-da-Velha-4 por classes de implementação. Nesse sentido, a classe de construção do Tapatan, *TapatanBoard*, passa a estender $AbstractDisplacementBoard$ (Código 4.5), e a classe do Jogo-da-Velha-4, chamada de $TicTacToeIVBoard$ (Código 4.6), passa a estender $AbstractInsertionBoard$. É essencial notar que essa hierarquia de classes garante, por um lado, a tipagem estática dos movimentos executados sobre um tabuleiro (verificável em tempo de compilação), fato introduzido pela quebra de $AbstractBoard$ em duas classes abstratas filhas. Ao mesmo tempo, um objeto consumidor de um tabuleiro - especificamente, a Inteligência Artificial com o Minimax - precisa conhecer somente os métodos e propriedades fornecidos pela interface exposta por $AbstractBoard$.

Código 4.5: Classe *TapatanBoard* como herdeira de *AbstractDisplacementBoard*.

```
class TapatanBoard(
    player1: Player,
    player2: Player,
    height: Int,
    width: Int
) : AbstractDisplacementBoard(player1, player2, height, width) {
    override fun play(movement: DisplacementMovement, simulate: Boolean): PlayResult {
        // ...
    }
    override fun enumeratePossibleMovements(
        player: Player
    ): Iterable<DisplacementMovement> { /* ... */ }
    override fun isMovementValid(movement: DisplacementMovement): Boolean { /* ... */ }
    override fun getWinner(): Player? { /* ... */ }
}
```

Assim, o grande salto feito durante o Jogo-da-Velha-4 foi o fortalecimento da

modelagem do sistema, a qual culmina no que está exposto na Seção 3.3. Mais ainda: a introdução das classificações e hierarquias de movimento e tabuleiro viabiliza a ideia de uma IA Minimax capaz de jogar um jogo sem de fato saber qual é esse jogo. De forma livre, pode-se afirmar que a IA apenas conhece os métodos e propriedades definidos pelo contrato da classe $AbstractBoard$, uma vez que as operações por esta definidas são mais que suficientes para a construção do algoritmo Minimax. Na prática, o mesmo código que implementa o Minimax, sem quaisquer alterações, será executado tanto para o Tapatan quanto para o Jogo-da-Velha-4.

Código 4.6: Classe *TicTacToeIVBoard* como herdeira de *AbstractInsertionBoard*.

```
class TicTacToeIVBoard(
    player1: Player,
    player2: Player,
    height: Int,
    width: Int
) : AbstractInsertionBoard(player1, player2, height, width) {
    override fun play(movement: InsertionMovement, simulate: Boolean): PlayResult {
        // ...
    }
    override fun enumeratePossibleMovements(
        player: Player
    ): Iterable<InsertionMovement> { /* ... */ }
    override fun isMovementValid(movement: InsertionMovement): Boolean { /* ... */ }
    override fun getWinner(): Player? { /* ... */ }
}
```

Uma vez contornada a necessidade de codificação de um Minimax para o Tapatan e outro para o Jogo-da-Velha-4, a atenção pode ser direcionada para as regras de movimentação e de fim de jogo do Jogo-da-Velha-4. Considerando os métodos e propriedades elementares de $AbstractBoard$, a classe $TicTacToeIVBoard$ materializa as regras do Jogo-da-Velha-4 da seguinte forma:

- **Tabuleiro:** matriz de dimensão 5×5.
- **Início de jogo:** tabuleiro vazio.
- **Jogar:** movimento de inserção de peça sobre uma posição (i, j) livre do tabuleiro matricial.
- **Enumerar movimentos permitidos:** para um jogador j, um movimento de inserção m sobre uma posição (i, j) do tabuleiro T é permitido se, e somente se, (i, j) não está ocupada por nenhuma peça.
- **Validar:** seja M o conjunto de movimentos permitidos para o jogador j em um determinado instante de jogo. Um movimento candidato m é válido se, e somente

se, $m \in M$.

- **Fim de jogo:** se for observado, sobre uma das arestas (linhas) do tabuleiro, um alinhamento vertical, horizontal ou diagonal de 4 peças tal que essas 4 peças pertençam ao jogador j, então j é o vencedor e a partida é encerrada.

Como visto na Seção 4.2, o Minimax implementado para o Tapatan foi satisfatoriamente aplicado. Uma vez estipulado que o Minimax irá atuar sobre um jogo abstrato, desconhecendo a implementação particular deste jogo, a intuição passa a sugerir que o Jogo-da-Velha-4 não terá dificuldades de ter viabilizada uma Inteligência Artificial. No entanto, é crucial notar uma alteração até então subestimada no sistema: a expansão das dimensões do tabuleiro, que tinha 9 posições no Tapatan (com apenas 3 livres), e passou a contar com 25 posições no Jogo-da-Velha-4, sendo todas livres no início do jogo.

De forma descompromissada, podemos afirmar que o núcleo do Minimax consiste no Algoritmo 9. Pelas regras do Jogo-da-Velha-4, desconsiderando totalmente o jogador que irá executar o movimento, a jogada 1 oferece 25 possibilidades de jogada, a jogada 2 permite 24 movimentos, a jogada 3 reduz para 23 movimentos e assim sucessivamente, até que todo o tabuleiro esteja ocupado. Entretanto, se aplicarmos a lógica do Algoritmo 9, notaremos que cada uma das 25 possibilidades de jogada da rodada 1 permite 24 movimentos na rodada 2 e que cada um desses 24 movimentos viabilizará 23 inserções na jogada 3, até que novamente o tabuleiro esteja totalmente preenchido ou o jogo tenha se encerrado.

Algoritmo 9 Síntese da visita de estados (tabuleiros) do algoritmo Minimax.

```
1: function VISITARESTADOS(estadoInicial, arvoreMinimax)
2:     vizinhos ← estados atingíveis por estadoInicial
3:     for v ∈ vizinhos do
4:         arvoreMinimax é atualizada com nodo v
5:         visitarEstados(v, arvoreMinimax)      ▷ Recursão para visita aos estados atingíveis por v.
6:     end for
7: end function
```

Apesar da informalidade desse pensamento, é fácil de ver a tendência que seguida pela expansão dos nós do Minimax para o Jogo-da-Velha-4. Para cada uma das 25 possibilidades, tem-se 24, logo a árvore terá 25×24 nós; indo adiante, para a expansão do terceiro nível, 23 posições disponíveis: $25 \times 24 \times 23$ nós da árvore. Ao exaurir essa série, fica claro que a quantidade total de nós da árvore Minimax para o Jogo-da-Velha-4, em um tabuleiro matricial com 25 inicialmente livres, é $25 \times 24 \times 23 \times \cdots \times 1 = 25!$. Essa

complexidade é claramente intolerável e, se não contornada, inviabiliza integralmente uma partida em que a Inteligência Artificial é um dos jogadores.

Embora não resolva particularmente a explosão combinatória observada na dedução da árvore Minimax do Jogo-da-Velha-4, a introdução da Poda α-β é muito bem vinda. A redução do espaço de busca proporcionada pela eliminação de nós-folha e subárvores inteiras é, pois, uma tendência natural a se seguir quando se projeta um sistema com IA Minimax. Implementada com a mesma premissa genérica proporcionada pelo polimorfismo paramétrico que caracterizou a refatoração da IA para o Jogo-da-Velha-4, a Poda α-β foi trivialmente incorporada ao sistema, como discutido na seção 3.4; importante ressaltar, ainda, que o Minimax puro e a Poda α-β são classes apartadas e pode-se escolher, numa controladora de jogo, qual dos dois algoritmos usar para o jogador IA.

Um adendo acerca da Poda α-β quando aplicada ao Jogo-da-Velha-4: o jogo caminha lentamente para uma situação de tabuleiro em fim de jogo. Ainda que a poda de caminhos desinteressantes para os jogadores compacte o número de estados a serem expandidos, o número de possíveis casos é ainda assim considerável. O impacto é reduzido, mas o tempo de decisão da IA em decorrência da alta complexidade de tempo observada no Minimax / Poda α-β do Jogo-da-Velha-4 segue inaceitável.

Como o Minimax puro com expansão em largura, visto na Seção 3.4, não é o mais performático para o Jogo-da-Velha-4, a análise aqui conduzida considera a execução sobre do jogo conduzido por uma IA com Poda α-β. Mesmo sendo executado com essa variante do Minimax, as jogadas da IA demoraram longos períodos de tempo, geralmente tendo a execução do programa interrompida pela JVM em função do excesso de alocação de memória dinâmica.

Como, então, reduzir o tempo de decisão da Inteligência Artificial sem comprometer a qualidade de uma jogada? Uma abordagem tipicamente empregada para forçar o retorno de uma jogada é o limite de tempo, ou *timeout*. O Algoritmo 10 mostra, em linhas muito generalistas, como é a implementação da Poda α-β com limitação de tempo.

Algoritmo 10 Aplicação de *timeout* como forma de conter a explosão combinatória do Minimax / Poda α-β.

```
1: function PODAALFABETACOMTIMEOUT(timeout)
2:     tempoInicio ← agora()
3:     while agora() − tempoInicio ≤ timeout do
4:         Executar Minimax com Poda α-β
5:     end while
6: end function
```

Essa abordagem, conquanto simples, é bem-sucedida em reduzir o espaço de busca. A decisão da IA é garantidamente retornada, contudo, sem a menor garantia de qualidade. Dois fatores competem para que a jogada selecionada para o Minimax não seja a melhor jogada: o número de possíveis estados observado para o Jogo-da-Velha-4 e a expansão em profundidade do Minimax com Poda α-β.

Isso fica claro ao analisarmos novamente o Algoritmo 9. Dado um limite de tempo normalmente aceitável para uma partida com um ser humano, digamos, pouco menos de 1 minuto, a expansão em profundidade irá garantir uma visão distante do jogo (muitas jogadas adiante), porém pouca visão geral das possibilidades mais próximas (em oposição à expansão em largura). Mesmo a previsão de jogadas em maior profundidade é precária, visto que o *timeout* impõe um fator não determinístico de quantos nós serão expandidos.

A contenção da explosão combinatória por limite de tempo é, assim, inadequada. Existe a garantia de uma decisão a ser tomada em tempo hábil e há alguma visão em profundidade por parte da IA, contudo, a árvore fica com largura muito menor e as jogadas de curto prazo tendem a apresentar qualidade duvidosa.

De fato, a solução ideal para o Jogo-da-Velha-4 deve consistir na deteção de que uma explosão combinatória é iminente. De antemão, deve-se assumir que essa alternativa, posto que factível para o Jogo-da-Velha-4, não deve interferir no funcionamento satisfatório da Inteligência Artifical para o Tapatan. Deve, ademais, evitar ao máximo que falsos positivos se manifestem, i.e., sinalizar que há uma tendência a explosão de estados quando esta com efeito inexiste.

Algoritmo 11 Esboço do algoritmo de detecção de tendência a explosão combinatória.

```
1: TamanhoTabuleiroDeteccao ← K
2: function DETECTAEXPLOSAOCOMBINATORIA(tamanho, historico, threshold)
3:     if tamanho < TamanhoTabuleiroDeteccao ∨ #historico < threshold then
4:         return false
5:     end if
6:     for indice ∈ [0, #historico) do
7:         if |historico[indice] − historico[indice + 1]| > 1 then
8:             return false
9:         end if
10:    end for
11:    return true
12: end function
```

O Algoritmo 11 mostra uma forma de detectar a explosão combinatória para o Jogo-da-Velha-4. Os parâmetros dessa função têm a seguinte semântica:

- $tamanho$**:** número total de posições que o tabuleiro oferece. Por exemplo, o Tapatan tem 9 posições (matriz 3×3).
- $historico$**:** fila que armazena o número de estados filhos que foram expandidos pelo Minimax / Poda α-β. Por exemplo, se o estado inicial s tem 3 estados atingíveis, então $historico = \{3\}$. Ao expandir o primeiro estado s_1, filho de s, se verifica que s_1 possui 2 estados filhos; logo, $historico = \{3, 2\}$.
- $threshold$**:** dado que a cardinalidade da fila $historico$ representa a profundidade da árvore Minimax, $threshold$ é apenas uma variável para controlar se a profundidade atingida (equivalente a $\#historico$) é grande o suficiente para que se tente detectar uma explosão de estados.

Uma análise mais detida sobre o algoritmo mostra o seguinte comportamento:

- **Linha 1:** constante que determina quantas posições o tabuleiro deve ter para que a análise de explosão de estados seja aplicada. Aqui o Tapatan pode ser sumariamente ignorado se definirmos $K = 25$, considerando o tabuleiro matricial 5×5 do Jogo-da-Velha-4.
- **Linhas 3-5:** verificação básica para a aplicação da função: se a área do tabuleiro for menor que o limite estipulado na linha 1 ou se a profundidade da árvore Minimax com Poda α-β for menor que o limiar fornecido como parâmetro, i.e., ainda é razoável avançar na expansão dos possíveis tabuleiros (mesmo sob o risco de explosão combinatória), então a função de detecção não é aplicada.
- **Linhas 6-7:** iteração em pares sobre os elementos pertecentens ao rastro de quantidades de filhos de nós da árvore representado por $historico$: $historico[indice]$ é o elemento atual, $historico[indice + 1]$ é o próximo elemento (por isso o intervalo da iteração é aberto sobre $\#historico$). Caso não se apresente uma tendência de valores em que se observe a função fatorial, i.e., $v, v - 1, v - 2, \ldots, 1$, então a detecção não sinaliza tendência a explosão de estados. Caso contrário, todos os elementos contidos em $historico$ compõem, juntos, a função fatorial: há, portanto, tendência a explosão de estados.

Durante a expansão da árvore Minimax para o Jogo-da-Velha-4, a cada recursão em profundidade feita pela Poda α-β, o $historico$ satisfaz organizadamente o almejado pelo Algoritmo 11. Como discutido anteriormente, a raiz terá 25 possibilidades de jogo, todos os 25 nós filhos terão 24 ofertas de movimentação, todos os 24 destes últimos terão 23 estados atingíveis, etc. Logo, para cada subárvore recursivamente descendente da raiz

da árvore Minimax, o Algoritmo 11 irá operar sobre $historico = \{25, 24, 23, 22, \dots\}$. Nesse caso, a explosão combinatória será descoberta e a expansão do Minimax com Poda α-β será interrompida, fazendo a IA retornar a melhor jogada possível.

O Algoritmo 11 efetivamente resolve o excesso de tempo para a tomada de decisão da IA. Entretanto, diferentemente do *timeout*, a largura da árvore Minimax não é comprometida. A detecção de explosão combinatória apresentada, portanto, atua como um limitador criterioso de profundidade: em vez de um escalar constante, a expansão da árvore de jogadas é submetida a uma análise mais sofisticada de limite de profundidade. Naturalmente, ainda que a introdução do Algoritmo 11 faça as vezes de corte de profundidade, o Minimax e a Poda α-β seguem contando com um limite fixo de profundidade, até mesmo como alternativa ao eventual retorno $false$ do Algoritmo 11. É Importante ressaltar também que o Tapatan não é afetado pela detecção de explosão combinatória, o que forçosamente implica na necessidade de se manter o limite de profundidade percebido nas Seções 3.4 e 4.2.

Claramente, no entanto, o Algoritmo 11 é bastante específico. Desconsidera, por exemplo, uma distribuição *quase* fatorial sobre o histórico de estados expandidos ao longo do Minimax / Poda α-β. Isto é, um $historico = \{25, 23, 21, \dots\}$ passará incólume pelo Algoritmo 11, mesmo que represente claramente a tendência a explosão de estados. Entretanto, como solução pontual para o Jogo-da-Velha-4, essa detecção é satisfatória; ademais, revisitar o algoritmo para que a análise sobre o número de estados expandidos ao longo da execução da Inteligência Artificial é trivial, e o Algoritmo 11 tem plenas condições de ser acrescido de heurísticas menos pontuais. O Algoritmo 12, nos moldes de simplificação do Algoritmo 10, mostra como é a Poda α-β auxiliada pela detecção de explosão de combinatória do Algoritmo 11.

Como visto, o salto aparentemente simples do Tapatan para o Jogo-da-Velha-4 não foi tão trivial quanto inicialmente suposto. A implementação do Jogo-da-Velha-4 não foi particularmente complicada, tendo esbarrado apenas na modelagem um tanto engessada do Tapatan, mostrada na Seção 4.2. Essa revisão na modelagem, fundamental para a viabilização de uma IA agnóstica ao jogo de estratégia em execução, mostra-se de grande valor, pois é o fator mais significante na evolução escalável do projeto. A controladora demonstrada para o Tapatan no Algoritmo 8 é tacitamente adaptável para o Jogo-da-Velha-4.

O segundo ganho na implementação do Jogo-da-Velha-4 foi a efetiva implementação do Minimax com Poda α-β. Naturalmente, esse é o caminho natural de soluções de jogos abstratos e de estratégia calcados no Minimax. O ganho, porém, é muito maior

Algoritmo 12 Aplicação do Algoritmo 11 como forma de conter a explosão combinatória do Minimax / Poda α-β.

```
1: L ∈ ℕ+        ▷ Limite de profundidade para aplicar detecção de explosão de estados.
2: function PODAALFABETADETECCAOEXPCOMB(nodo, historico)
3:     t ← tamanho do tabuleiro referenciado por nodo
4:     explode ← detectaExplosaoCombinatoria(t, historico, L)
5:     if explode then
6:         return                                  ▷ Interrompe expansão da árvore.
7:     end if
8:     colecaoNumeroEstados ← ∅
9:     if nodo não é raiz then
10:        colecaoNumeroEstados ← historico
11:    end if
12:    tabuleirosPossiveis ← tabuleiros atingíveis a partir de nodo
13:    for filho ∈ tabuleirosPossiveis do
14:        nodo ← nodo atualizado com o filho filho
15:        if filho não é nó terminal then
16:            histFilho ← historico ∪ {#tabuleirosPossiveis}
17:            podaAlfaBetaDeteccaoExpComb(filho, histFilho)
18:        end if
19:    end for
20: end function
```

que o esforço, visto que todos os jogos atualmente construídos (Tapatan e Jogo-da-Velha-4), além dos futuros, irão se beneficiar dessa considerável otimização. Otimização, aliás, com zero impacto, porque a jogada fornecida por uma Inteligência Artificial com Minimax puro é a mesma de uma IA Minimax com Poda α-β.

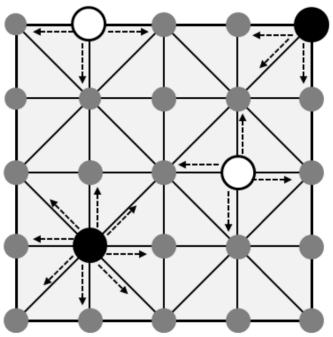
A percepção da possibilidade de explosão de estados e a necessidade de detectá-la e evitá-la é mais um benefício ocasionado pela proposta do Jogo-da-Velha-4. Ainda que limitada, a função de detecção de explosão combinatória já é um ganho para o sistema, e pode facilmente ser expandida - ou até mesmo acrescentada de mais funções auxiliares, adaptáveis à diferentes situações que aumentam brutalmente a complexidade de tempo. Uma possível forma de tornar a IA mais robusta é, por exemplo, combinar o *timeout* com a função de prevenção de explosão de estados. Isso, entretanto, não fere a solução construída até aqui: tem-se o Tapatan e o Jogo-da-Velha-4 como dois casos implementados com sucesso.

4.4 Implementação: Tic Tackle

Seguindo a linha de exploração de jogos abstratos e de estratégia distintos entre si, o Tic Tackle surge como o terceiro - e de fato último - caso de implementação. O apelo do Tic Tackle vem da composição de conceitos observados tanto no Tapatan quanto no Jogo-da-Velha-4, o que faz supor, mais uma vez, que poucos impedimentos se avizinham. Afinal, como visto na Seção 4.3, o sistema foi fortalecido em diversos aspectos importantes.

Um dos 21 jogos abstratos e de estratégia descritos por Ribas (2020), o Tic Tackle é jogado sobre o ubíquo tabuleiro matricial 5×5. Como no Tapatan, as peças de cada jogador iniciam colocadas alternadamente sobre as extremidades superior e inferior do tabuleiro (Figura 2.3); como no Tapatan, cada jogador pode deslocar uma peça por jogada para qualquer posição livre adjacente, desde que em apenas um passo, podendo fazê-lo na vertical, horizontal e diagonal, como mostra a Figura 4.3. Como no Jogo-da-Velha-4, o objetivo do jogo é formar um alinhamento de 4 peças na vertical, na horizontal ou na diagonal; diferentemente do Jogo-da-Velha-4, todavia, o alinhamento em diagonal não precisa ser feito sobre uma das arestas (linhas) do tabuleiro, conforme ilustrado pela Figura 4.4.

Figura 4.3: Regras de movimentação do Tic Tackle.



Fonte: Ribas (2020)

Estruturalmente, o tabuleiro do Tic Tackle não requer nenhuma alteração em relação a matriz 5×5 vista na Seção 4.3. A nível de classe, a priori, podemos classificar o Tic Tackle como um $AbstractDisplacementBoard$, isto é, um tabuleiro em que os jogadores executam movimentos de deslocamento. No entanto, não podemos ignorar as semelhanças do Tic Tackle com o Tapatan.

As similaridades entre ambos são tão notórias que permitem uma sutil readequação na modelagem orientada a objetos. Os dois jogos preveem movimentos de um passo

apenas, tanto em linha e coluna quanto em diagonal, restritos apenas à posições adjacentes livres. Por *movimento de um passo apenas* pode-se entender, num jargão mais próximo da abstração vista na Seção 3.2, movimento de deslocamento unitário.

O intuito da reclassificação do Tapatan e do Tic Tackle esconde mais do que apenas coesão na hierarquia de classes. A classe $TapatanBoard$ já apresenta a codificação de validação e enumeração dos movimentos de deslocamento unitários em linhas, coluna e diagonal. Ainda, o método que computa uma jogada - $play$ - também contém todo o código necessário para a efetivação da jogada sobre o tabuleiro. Em síntese, o código escrito para a classe $TapatanBoard$ será essencialmente o mesmo que deveria ser escrito para o Tic Tackle.

O objetivo, portanto, é criar uma classe abstrata da qual tanto o Tapatan quanto o Tic Tackle irão herdar. Sabe-se que essa classe se caracteriza por movimentos de deslocamento, logo, terá como classe base a $AbstractDisplacementBoard$. Surge, então, a classe abstrata $SingleStepBoard$ (Código 4.7): ***single step*** pela compreensão de movimentos unitários (naturalmente, de deslocamento, logo instâncias de ***DisplacementMovement***), responsável por enumerar e validar esses movimentos unitários restritos às regras que são comuns ao Tic Tackle e ao Tapatan.

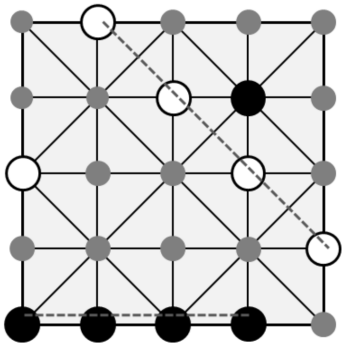
Código 4.7: Classe abstrata $SingleStepBoard$.

```
abstract class SingleStepBoard(
    player1: Player,
    player2: Player,
    height: Int,
    width: Int
) : AbstractDisplacementBoard(player1, player2, height, width) {
    override fun play(movement: DisplacementMovement, simulate: Boolean): PlayResult {
        /* ... */
    }
    override fun enumeratePossibleMovements(
        player: Player
    ): Iterable<DisplacementMovement> { /* ... */ }
    override fun isMovementValid(movement: DisplacementMovement): Boolean { /* ... */ }
}
```

Assim, a classe de tabuleiro e regras de movimentação e fim de jogo do Tic Tackle, batizada $TicTakcleBoard$, tacitamente estende $SingleStepBoard$. A classe $TapatanBoard$, antes filha de $AbstractDisplacementBoard$, passa uma geração adiante ao adotar esta como avó, passando a estender, qual o Tic Tackle, a $SingleStepBoard$. Além da organizada reestruturação da sub-hierarquia de classes enraizada por $AbstractBoard$, as classes $TapatanBoard$ e $TicTackleBoard$ ficaram com um número extremamente

pequeno de linhas de código, visto que as regras de jogo das quais ambas se valem estão corretamente implementadas na $SingleStepBoard$. Essa reestruturação pode ser vista no Código 4.8.

Figura 4.4: Fim de jogo no Tic Tackle com 4-alinhamento fora de uma linha.



Fonte: Ribas (2020)

De fato, apenas três diferenças se observam entre as classes $TapatanBoard$ e $TicTackleBoard$. A primeira é o tamanho do tabuleiro, solucionado pela parametrização de largura e altura da $AbstractBoard$, como vista na Seção 3.3. A segunda é o tamanho do alinhamento necessário para a vitória: 3 peças para o Tapatan, 4 para o Tic Tackle. Por fim, o Tic Tackle aceita como fim de jogo todas as disposições de alinhamento que caracterizam fim de jogo no Tapatan (horizontal, vertical e diagonal), mais um 4-alinhamento diagonal fora das linhas do tabuleiro. Essas são, portanto, as diferenças na implementação de cada classe que $SingleStepBoard$ é compreensivelmente incapaz de englobar.

Código 4.8: Classes *TapatanBoard* e *TicTackleBoard* como herdeiras de *SingleStepBoard*.

```
class TapatanBoard(
    player1: Player,
    player2: Player,
    height: Int,
    width: Int
) : SingleStepBoard(player1, player2, height, width) {
    override fun getWinner(): Player? { /* ... */ }
    override fun buildBoard(): Array<Array<Piece>> { /* ... */ }
}
class TicTackleBoard(
    player1: Player,
    player2: Player,
    height: Int,
    width: Int
) : SingleStepBoard(player1, player2, height, width) {
    override fun getWinner(): Player? { /* ... */ }
    override fun buildBoard(): Array<Array<Piece>> { /* ... */ }
}
```

A impressão inicial de que o Tic Tackle não incorreria em problemas, como visto na Seção 4.3, se verificou falsa. Ao executar uma rodada do jogo a partir de uma controladora análoga à demonstrada no Algoritmo 8, a qual se valeu, como no Jogo-da-Velha-4, do Minimax com Poda α-β, mostrou um comportamento errático. Ora os movimentos de deslocamento lançados pela Inteligência Artificial se repetiam *ad aeternum*, ora a execução do programa travava em uma expansão interminável do árvore Minimax. Claramente se fazia necessário a análise de quais fatores estariam influenciando no mau comportamento, por assim dizer, da IA.

A primeira anomalia que se verificou na execução de jogos em que a Inteligência Artificial participou foi o alarmante consumo de memória. Tal situação tinha sido observada durante o desenvolvimento do Jogo-da-Velha-4, no entanto, se agravou nos testes conduzidos com o Tic Tackle. Uma análise mais detida mostrou que a expansão da árvore do Minimax, fosse na variante pura, fosse na Poda α-β, mostrava uma retenção insustentável de memória dinâmica (porção da memória conhecida por monte ou *heap*), segundo estatísticas da JVM.

A retomada das decisões admitidas na concepção do protótipo do Tapatan, detalhadas na Seção 4.2, denuncia um culpado de certa forma antevisto. Ao reter o estado do tabuleiro para um nó da árvore Minimax, isto é, uma instância de objeto filha de $AbstractBoard$ (e.g., $TapatanBoard$ e $TicTacToeIVBoard$), a implementação do Minimax / Poda α-β efetua uma cópia profunda da instância do tabuleiro, a fim de garantir que cada nó poderá ter o tabuleiro por ele representado avaliado sem o risco de efeitos colaterais introduzidos pela manipulação por outras partes do código, haja visto que Kotlin, qual Java, emprega a referência de memória quando o operador de atribuição $=$ é usado em comandos como $objetoA = objetoB$. O Algoritmo 13 exemplifica esse comportamento.

Algoritmo 13 Exemplo de alteração indesejada em um tabuleiro ocasionada pelo uso de referência de memória.

1: $tabuleiroA \leftarrow$ novo tabuleiro do jogo J
2: $tabuleiroB \leftarrow tabuleiroA$
3: $movimento \leftarrow$ movimento permitido para o $tabuleiroA$
4: $jogar(tabuleiroA, movimento)$ ▷ A jogada do movimento $movimento$ sobre o $tabuleiroA$ irá alterar o estado deste e do $tabuleiroB$.

Durante a expansão dos nós, dado que cada nodo da árvore representa uma possível jogada e tem, em sua estrutura interna, um tabuleiro do jogo, esse comportamento resultaria em tabuleiros degenerados para todos os nós da árvore Minimax. Seja n um nó

qualquer da árvore Minimax, tal que $n = (m, t, c)$, onde m é o movimento do jogador que leva o jogo ao nó n do Minimax, t é a instância do tabuleiro que n representa (onde t garantidamente apresenta o movimento m computado) e c é o conjunto potencialmente vazio de nós filhos de n. Seja T o tabuleiro inicial de um jogo abstrato de estratégia qualquer. O Algoritmo 14, na chamada $minimax(T, null, MAX)$, exemplifica o porquê de a atribuição sem cópia profunda degenerar os nós do Minimax (essa noção se aplica tanto ao Minimax quanto a Poda α-β). O parâmetro T representa o tabuleiro inicial do jogo, $nodo$ é o nodo a ser expandido e j é o jogador da vez.

Algoritmo 14 Degeneração dos tabuleiros retidos pelos nós durante o Minimax sem cópia profunda.

```
1:  function EXPANDIR(T, nodo, j)
2:      if nodo = null then
3:          nodo ← (null, T, ∅)                    ▷ Raiz não possui movimento gerador.
4:      end if
5:      movimentos ← conjunto de movimentos permitidos ao jogador j sobre nodo.t
6:      proximoJogador ← MAX se j = MIN senão MIN
7:      for m ∈ movimentos do
8:          tabuleiroFilho ← nodo.t
9:          jogar(tabuleiroFilho, m)    ▷ Atribuição fará a execução da próxima linha
    alterar o tabuleiro dos nós pai (nodo.t) e filho (tabuleiroFilho).
10:         filho ← (m, tabuleiroFilho, ∅)
11:         nodo.c ← nodo.c ∪ {filho}
12:         expandir(T, filho, proximoJogador)
13:     end for
14: end function
```

Considerando o efeito produzido pela referência de memória, brevemente demonstrado no Algoritmo 13, pode-se notar a dimensão do impacto causado no expansão mostrada no Algoritmo 14. Na linha 8, a instância do tabuleiro do nó filho, $tabuleiroFilho$, é criada como um objeto que referencia a mesma região de memória dinâmica que o tabuleiro do nó pai, $nodo.t$. Logo, a execução da jogada m (geradora do nó $filho$) na linha 9 altera tanto o tabuleiro filho $tabuleiroFilho$, aquele que se deseja modificar efetivamente, quanto o tabuleiro $nodo.t$ do nó pai, o qual seguramente não se almeja alterar, visto que o tabuleiro do nó pai apresentará tanto a jogada que eventualmente o originou quanto a jogada que originou o nodo filho. Eis, portanto, o porquê de a atribuição da linha 8 ter sido implementada não como $tabuleiroFilho \leftarrow nodo.t$, mas sim como $tabuleiroFilho \leftarrow$ **clone** $nodo.t$, onde o *clone* executa uma cópia profunda - sem referência a nenhum objeto que compõe o tabuleiro $nodo.t$. Assim, as jogadas podem ser executadas sobre os tabuleiros que auxiliam na representação dos nós da árvore Minimax sem que o estado destes

na tabuleiro, na forma final da árvore, tenham sido degenerados.

Por um lado, a cópia profunda das instâncias de tabuleiro garante a integridade dos estados retidos pelos nós da árvore Minimax. Por outro, aumenta significativamente o consumo de memória nos momentos de derivação da árvore do Minimax / Poda α-β. Para o Tic Tackle, o benefício da correção cobra o preço de comprometer a execução do jogo com a Inteligência Artificial como jogador, diferentemente do que foi visto no Tapatan e no Jogo-da-Velha-4. É imprescindível, portanto, encontrar uma nova solução para a retenção dos estados dos nós da árvore Minimax que não se degenere e ao mesmo tempo não consuma ferozmente a memória durante o Minimax / Poda α-β.

A solução para isso deve satisfazer duas restrições: amortizar o consumo de memória característica da cópia profunda dos tabuleiros e evitar que os tabuleiros se degenerem, como mostrado no Algoritmo 14. Surge, então, uma ideia interessante: forçar a classe responsável pela Inteligência Artificial a utilizar uma, e somente uma, instância de tabuleiro para a expansão dos nós da árvore Minimax. A cada nodo expandido, a jogada é feita, o tabuleiro é avaliado e, antes de seguir para o próximo nó, a jogada que originou o nodo expandido é desfeita, deixando o tabuleiro no mesmo estado de antes da expansão desse nó.

A primeira consequência dessa alteração é que a classe que representa um nó da árvore Minimax, i.e., $TreeNode$ (como visto na Seção 3.3), não irá mais conter uma propriedade que referencia uma instância de $AbstractBoard$ (como mostrado no Tapatan da Seção 4.2). Em vez disso, $TreeNode$ deve assumir uma propriedade discutida na Seção 3.3: o rastro de movimentos executados até o nodo. Se, portanto, um nó n da árvore contém em sua estrutura o movimento g que o originou, então, a partir dessa modificação, n conterá também a coleção de movimentos M que foram feitos até o ponto do jogo em que n é uma imediata possível jogada, sendo M ordenado pelo instante em que os movimentos foram executados, ou seja, se $M = \{m_1, m_2, \dots\}$, então m_1 precede m_2 no nível da árvore Minimax (o movimento m_1 foi executado antes de m_2). Logo, o rastro efetivo do jogo, sob a ótica de n, é dado por $M \cup \{g\}$.

O conjunto ordenado dos rastros de movimentos e o movimento gerador permitem que a partir de um nó da árvore Minimax seja possível reconstruir o jogo executado. Isso equivale, ainda que de forma um tanto mais trabalhosa, a ter uma instância do tabuleiro em que todas as jogadas dos ancestrais de um nó foram executadas, mais a jogada que origina o próprio nó. Reter, pois, a instância do tabuleiro por cópia profunda dentro de uma instância de $TreeNode$ é, além de tudo, um preço caro a pagar, porquanto o tabuleiro

será inteiramente copiado apenas para ser mantido como atributo para avaliação do nó, imutável; com efeito, é uma abordagem intuitiva, porém dispendiosa.

A alteração no núcleo da Inteligência Artificial Minimax e Poda α-β consiste em:

1. Remover a propriedade que referencia uma instância não nula e única ao tabuleiro do jogo.
2. Adicionar uma nova propriedade, uma lista ordenada de instâncias de *PlayerMovement*, que representam os rastros dos movimentos executados por todos os nós ancestrais. Por definição, a raiz da árvore Minimax terá essa lista vazia. Nesse caso específico, não há efeitos colaterais introduzidos pela referência a instâncias de $PlayerMovement$, i.e., essa lista não requer que os objetos sejam a ela adicionados por cópia profunda.
3. Alterar as classes $Minimax$ e $AlphaBetaSearch$ para que estas retenham uma instância de $AbstractBoard$ (de fato, uma instância de uma classe que a estende, como o Tic Tackle com a $TicTackleBoard$). Essa instância terá inicialmente a configuração do tabuleiro inicial, como definido pela implementação do jogo; ao longo da expansão do Minimax, o tabuleiro será constantemente alterado para que os nós da árvore possam ser avaliados pela função custo. Entretanto, ao fim de uma rodada de execução do Minimax / Poda α-β, o tabuleiro deverá voltar a esse exato estado inicial.
4. Alterar os algoritmos do Minimax puro e da Poda α-β para que pontualmente executem a operação de desfazer uma jogada sobre o tabuleiro. Em especial, na demanda de execução de mais níveis da árvore, por exemplo, se a árvore foi expandida até o nível k e é requerida a expansão até o nível $k + l$, dado requisito anterior, os rastros de movimento do nó sobre o qual a Inteligência Artificial está posicionada no instante k do jogo devem ser executados sobre a instância do tabuleiro que retém o estado inicial do jogo. Assim, a demanda de novos níveis da árvore Minimax executa todas as jogadas para que a busca inicie a partir do estado em que o jogo efetivamente se encontra.

Dando seguimento ao que foi apresentado no Algoritmo 14, o Algoritmo 15 mostra como a inclusão dos rastros de movimentos de jogo é aplicada no Minimax / Poda α-β. O Algoritmo 14 define o nó da árvore como $n = (m, t, c)$, sendo m o movimento originador do nodo, t a instância do tabuleiro representado por n (depreciada nesta nova implementação) e c o conjunto de nós filhos de n, tendo T como o tabuleiro inicial da

partida. Mantendo a definição de T, n é redefinido para $n = (m, c, r)$, onde m e c mantém a semântica do Algoritmo 14 e r representa o conjunto dos rastros de movimentos executados pelo nós ancestrais de n. Novamente, a hipotética execução completa do jogo até o nodo n é dado por $r \cup \{m\}$.

Algoritmo 15 Expansão dos nós da árvore Minimax (aplicável também a Poda α-β) com rastros de movimento e operação de desfazer jogada sobre um tabuleiro.

```
1:  function EXPANDIR(T, nodo, j)
2:      if nodo = null then
3:          nodo ← (null, ∅, ∅)                    ▷ Raiz não possui movimento gerador.
4:      end if
5:      movimentos ← conjunto de movimentos permitidos ao jogador j sobre nodo.t
6:      proximoJogador ← MAX se j = MIN senão MIN
7:      for m ∈ movimentos do
8:          filho ← (m, ∅, nodo.r ∪ {m})    ▷ Cria nó com movimento m, sem filhos,
    com rastros de movimento herdado do nó pai.
9:          nodo.c ← nodo.c ∪ {filho}                ▷ Nó filho é acumulado no nó pai.
10:         jogar(T, m)                              ▷ Executa a jogada m sobre o tabuleiro.
11:         expandir(T, filho, proximoJogador)
12:         desfazerJogada(T, m)    ▷ Desfaz a jogada sobre o tabuleiro, preparando a
    expansão para o nó irmão de filho, que não deve conter a jogada deste.
13:     end for
14: end function
```

Como mostra a linha 12 do Algoritmo 15, a cada novo nó avaliado e acumulado em um mesmo nível, o tabuleiro tem desfeita a jogada que originou esse nó. Isto é, todos os nodos que se originam da enumeração de movimentos da linha 5 são nodos irmãos (são iterados sobre o for em que $nodo$ é o pai), logo, a jogada deve ser desfeita, pois nodos irmãos têm como último movimento do rastro de jogadas o movimento que originou o nó pai, mas nunca jogadas de nós no mesmo nível. Com efeito, essa é a definição do nível: inúmeros caminhos a partir de uma só escolha. É fácil de ver que, ao fim da expansão dos nós da árvore Minimax, as sucessivas chamadas a função que desfaz uma jogada, $desfazerJogada(T, m)$, restaura o tabuleiro T ao estado inicial que este tinha imediatamente antes do início da expansão, deixando-o pronto para mais expansões Minimax / Poda α-β sem a necessidade de retenção de cópia profunda do tabuleiro.

Durante a expansão dos nós da árvore Minimax, o *footprint* de memória diminui consideravelmente a partir substituição da cópia profunda dos tabuleiros pelos rastros de jogada acompanhados da desfeita de movimento. Isso, claro, acelera substancialmente o tempo de expansão da árvore do Minimax, ainda mais com a Poda α-β, e por consequência faz a Inteligência Artificial decidir mais rapidamente qual movimento executar. No

entanto, essa considerável melhoria não foi suficiente para que a IA fornecesse retornos satisfatórios para o Tic Tackle, chegando mesmo a ser vencida em alguns cenários de jogo.

A esse ponto, notar que a Inteligência Artificial pode ser vencida é, no mínimo, intrigante. Afinal, mesmo um Minimax / Poda α-β parcamente construído garante ao menos um empate. Há que se considerar, entretanto, que o número de possibilidades de jogo introduzido por um tabuleiro maior que o 3×3 do Tapatan, visto na Seção 4.2, é fonte certa de imprevistos; o tabuleiro matricial 5×5 proposto por Ribas (2020) torna mais complicado o cenário de ataque da IA.

O porquê de o Minimax / Poda α-β optar por um movimento que leva à derrota guarda relação com o aumento de possibilidades de jogo que um tabuleiro maior oferece. Em geral, um jogo restrito a um tabuleiro menor, como o Tapatan clássico, é encerrado mais cedo do que um jogo disputado sobre um tabuleiro maior, ou que um jogo mais elástico, com mais movimentos possíveis por jogador (o que guarda relação intrínseca com o espaço disponibilizado pelo tabuleiro). Logo, os jogos disputados sobre o tabuleiro matricial 5×5 dos jogos cobertos por Ribas (2020) caminham muito mais lentamente para um cenário de fim de jogo.

Tal constatação se reflete na distribuição dos valores dos nós pela árvore do Minimax / Poda α-β. Nós terminais não limitados pela profundidade, isto é, nodos que apresentam valor de aplicação da função custo diferente de 0 (sendo 0 valor de convenção para empate) estão situados a uma profundidade substancialmente maior; requerem, portanto, mais tempo de execução do motor da Inteligência Artificial para que sejam atingidos. Tempo este que, como vimos até aqui, não está disponível, a menos que se presuma um usuário humano de paciência inexorável.

O resultado líquido dessa observação é que a tomada de decisão da Inteligência Artificial pode ser afetado pelo limite de profundidade introduzido para a amortização do tempo de execução do Minimax / Poda α-β. Ou seja, é possível que a próxima jogada selecionada ofereça uma situação não terminal de jogo, i.e., empate, mas deixe o jogo, no movimento adversário seguinte, em vias de ser vencido pelo oponente. Em síntese: um movimento pode parecer razoável num determinado instante de jogo, contudo, no instante imediatamente seguinte, desconhecido até então pela IA, se revela como o arauto da derrota.

Para deixar essa intuição mais clara, suponhamos que a árvore Minimax foi expandida até o nível de profundidade k, em que k é suficientemente grande, e.g., $k = 10$.

Nessa árvore de profundidade 10, todos os nós folha são não terminais, isto é, foram valorados como 0 pela função custo e são interpretados como estados de empate de jogo. Nada pode ser afirmado sobre os nós descendentes desses nodos-folha de profundidade 10, pois o Minimax / Poda α-β ainda não foi aplicado para que a IA possa enxergar além de 10 movimentos. Contudo, é factível assumir que um ou mais desses nodos de profundidade 10 têm um ou mais nós filhos, situados na profundidade 11 da árvore, que são efetivamente terminais (têm valor diferente de 0).

Ora, esse cenário pode facilmente permitir que a Inteligência Artificial opte por um movimento de ganho 0 (empate) no nível k, sendo que este mesmo movimento leva, no nível $k + 1$ (ainda não calculado pelo Minimax / Poda α-β), à derrota da IA por um movimento do oponente. Em suma, a derrota se esconde pela miopia da Inteligência Artificial, induzida pelo limite de profundidade.

De que forma, então, evitar que a derrota - ou mesmo a vitória, desinteressante nesse cenário fatalista - se esconda a um passo de distância? Se observarmos com cuidado, o problema de decisão têm relação direta com qual jogador é o último a executar um movimento na profundidade limitada: se MIN ou MAX. De fato, essa relação se estende a uma propriedade do valor de limite da profundidade: a paridade, sempre condicionada a qual jogador executou o primeiro movimento do jogo.

Se o jogador MAX inicia o jogo, isto é, a Inteligência Artificial, então o limite l de profundidade deve ser par. Dessa forma, as jogadas se alternarão entre $MAX, MIN, \ldots$ MAX, MIN, de modo que a última jogada executada, no instante l de jogo, é do oponente, garantindo que a Inteligência Artificial irá antever um movimento que pode levá-la à derrota. Não importa quão boa é a próxima jogada da IA, visto que o Mininmax / Poda α-β leva à decisões razoáveis em que MAX não será, por assim dizer, encurralado por MIN.

Similarmente, se MIN faz o primeiro movimento do jogo, ou seja, se o oponente da Inteligência Artificial começa a partida, então a profundidade l deve ser ímpar, de modo que a alternância se dê pela distribuição de jogadas $MIN, MAX, \ldots, MAX, MIN$. O objetivo é o mesmo que a limitação de profundidade por um escalar par, quando MAX principia a partida: garantir que a última jogada do oponente seja conhecida, com margem de correção por parte da IA caso o jogo apresente tendência de derrota para MAX.

Não é trivial perceber essa particularidade da limitação do nível de profundidade. O exercício de ampliação do sistema a partir da implementação de novos jogos é, portanto, fundamental, pois revela as fragilidades do sistema ao expô-lo às mais variadas

condições. Além disso, é fundamental ressaltar que as correções sobre as deficiências do sistema, em especial no tocante ao motor da Inteligência Artificial, beneficiam todos os jogos já implementados, com considerável probabilidade de fazer o mesmo para futuras construções de jogos abstratos e de estratégia que venham a compor o sistema.

Conquanto jogável, o Tic Tackle não foi um caso de implementação bem-sucedido. As tomadas de decisão da Inteligência Artificial foram melhoradas pela otimização de tempo. Os indesejados cenários em que a IA se mostrou surpreendentemente falível ao ser derrotada foram descartados. Contudo, a qualidade dos movimentos e a aguda tendência à vitória não se verificaram como no Tapatan e no Jogo-da-Velha-4.

A monotonia nos movimentos retornados pelo Minimax / Poda α-β é fator decisivio para o insucesso do Tic Tackle. A Inteligência Artificial toma decisões em tempo tolerável e não incorre em derrota, porém demonstra notória estagnação nas jogadas selecionadas. Por ser um jogo que oferece possibilidades muito similares de movimentação, muitas das quais indistintas entre si, o Tic Tackle pode ser estendido por longas e intermináveis sessões de jogo.

De fato, a indistinção entre os movimentos só ocorre se não houver uma análise mais minuciosa do quão bom é um movimento. Até aqui, o que se implementou como função de valor (amplamente discutida nas Seções 3.4 e 4.2) foi com base em critérios estritamente terminais: vitória, derrota e empate. Todavia, um movimento de jogo não deve ser positivamente qualificado somente quando o jogo é por ele encerrado.

Aqui se faz latente a carência de mais heurísticas no sistema, em especial no Minimax / Poda α-β, representante da Inteligência Artificial. Ora, não seria razoável supor que um movimento que efetiva um alinhamento de 3 peças no Tic Tackle é melhor do que um movimento que não o faz? Se os movimentos são qualificados apenas como um fim, ignorando o meio, então todos os movimentos não terminais são iguais entre si, o que claramente não é verdade.

A observância de movimentos que *conduzam* a uma situação de fim de jogo deveria fazer parte do sistema. Por exemplo, se o jogador MAX apresenta um alinhamento de 2 peças no Tic Tackle, e tem duas alternativas de jogadas: o movimento m_1, o qual forma um 3-alinhamento para MAX, e o movimento m_2, que simplesmente desloca uma peça isoladamente. Nesse caso, a função de valor aplicada a m_1 deve ter o mesmo resultado quando aplicada a m_2? A resposta é obviamente não. Embora tanto m_1 quanto m_2 não encerrem o jogo, m_1 aproxima MAX da vitória com o 4-alinhamento determinado para encerrar o Tic Tackle com muito mais certeza que o desconexo m_2. Portanto, seria não só

razoável como fundamental atribuir, por exemplo, um valor v_1 para m_1, com $0 < v_1 < 1$, e um valor v_2, com $0 < v_2 < v_1$, para m_2.

Com efeito, o Tic Tackle sofre pela tricotomia do sistema: derrota, empate, vitória. Inexiste um termo contínuo que caminhe entre esses três conceitos, fortes e determinantes. Essa constatação, portanto, termina por concluir que o sistema foi incapaz de obter o sucesso do Tapatan e do Jogo-da-Velha-4 com o Tic Tackle. A carência de heurísticas para a função de valor, amparadas pela implementação específica dos jogos, acaba por limitar o alcance que o sistema de múltiplos jogos de tabuleiro com Inteligência Artificial poderia ter.

5 ANÁLISE

5.1 Melhorias futuras

Invariavelmente, todo o trabalho conduzido até aqui força uma reflexão sobre o que poderia ser revisto para a evolução do sistema de jogos de tabuleiro com suporte a Inteligência Artificial. Essa reflexão, conquanto necessária, não condiciona o reinício do projeto: em vez de imaginá-lo como um quadro novamente em branco, é possível visualizar os pontos de falha e de melhoria a partir daquilo que foi proposto e implementado. A maturidade para essa análise, de fato, só se manifesta após o copioso processo de planejamento e desenvolvimento demonstrado ao longo deste trabalho.

Um dos primeiros aperfeiçoamentos que se avultam é a divisão de responsabilidades nas classes de implementação de jogos. As classes baseadas na $AbstractBoard$, afinal de contas, foram imbuídas de três grandes obrigações: definir o tabuleiro, descrever as regras de jogo e determinar quando um jogo é encerrado. A simples enumeração desses três objetivos deixa clara a margem que existe para a decomposição do modelo em, ao menos, mais duas abstrações: uma para que de descrevam as regras de jogo e outra para que as condições de fim de jogo sejam explicitadas.

A nova classe responsável por abstrair o tabuleiro do jogo poderia ser remodelada para que a estrutura física que define essa entidade dos jogos abstratos e de estratégia seja encapsulada ao máximo. Isto é, poder-se-ia introduzir uma sub-hierarquia de classificações de tabuleiros que particularizam desde a geometria até a estrutura de dados computacional. Isso aumentaria a flexibilidade de implementação de novos jogos e ao mesmo tempo eliminaria a tomada de decisões similares àquelas discutidas na Seção 3.5.

Naturalmente, como o projeto de jogos de tabuleiro com Inteligência Artificial se restringiu ao tabuleiro matricial 5×5 sugerido por Ribas (2020), essa abordagem pode parecer inócua. Entretanto, esse mesmo tabuleiro poderia ser implementado de diferentes formas, cada uma oferecendo meios para a melhor adequação de determinados jogos. Por exemplo, alguns jogos poderiam se beneficiar da objetividade de uma matriz, ao passo que outros poderiam ser codificados com menos complexidade se a estrutura de dados do tabuleiro fosse um grafo. Além disso, o isolamento do tabuleiro em uma classe que não cobre as regras de movimentação e fim de jogo permite que o sistema extrapole sem dificuldades o escopo delimitado pelos 21 jogos de tabuleiro de Ribas (2020).

A concepção de movimentos prevista pelo sistema também poderia ser revisitada.

Da forma que se apresenta, o sistema prevê uma certa crueza para os movimentos executados sobre o tabuleiro: inserção e deslocamento. Esse, contudo, não é o principal ponto de limitação, visto que as abstrações da Seção 3.2, em grande medida, cobrem a contento a variedade de jogos abstratos e de estratégia. A inflexão consiste em deixar de enxergar cada rodada do jogo como um único movimento e passar a perceber que cada jogador pode, a seu tempo, dispor de uma série de movimentos, como um *pipeline* de jogadas, algo infactível na modelagem atual, porquanto esta atrela, por polimorfismo paramétrico, um tabuleiro a um tipo de movimento (inserção ou deslocamento).

O *pipeline* de k jogadas poderia iniciar com um movimento m_1 e encerrar com um movimento m_k. Ou seja, em vez de o jogador j lançar mão de um único movimento m', sendo este limitado ao tipo inserção ($InsertionMovement$) ou deslocamento ($DisplacementMovement$), j iria a cada rodada lançar os movimentos $\{m_1, m_2, \ldots, m_k\}$, onde $m_{i \in [k]}$ é do tipo inserção ou deslocamento. Mesmo que k seja limitado a 1, o que com efeito elimina a ideia de *pipeline*, o sistema se flexibiliza para ser mais receptivo aos jogos de tabuleiro que preveem dois movimentos por jogador. Entretanto, o *pipeline* poderia admitir k variável conforme as regras de jogo: a depender do estado do tabuleiro, j poderia executar mais ou menos movimentos, reduzindo ou ampliando o tamanho do *pipeline* de jogadas.

A penalidade do *pipeline* dar-se-ia sobre o motor da Inteligência Artificial, qual seja, o Minimax / Poda α-β. Com k movimentos por jogador, cada expansão do nó da árvore teria um número exponencial de possibilidades. Em vez de expandir os estados para cada possível movimento, a estratégia de visita aos tabuleiros passaria a fazer uma análise combinatória do *pipeline* a fim de antever os próximos níveis da árvore e aplicar a função de valor aos nós. Contudo, a maioria dos jogos de tabuleiro conhecidos não prevê um conjunto excessivamente numeroso de movimentos por rodada, o que implica em um *pipeline* relativamente pequeno e administrável por parte da IA.

De qualquer forma, com ou sem *pipeline* de jogadas, a abstração das regras de jogo em classes apartadas é bastante interessante. Diferentes classes emergiriam para atender diferentes tipos de movimento, que se traduzem em regras, logo, cada jogo poderia contar com uma coleção de instâncias de classes de regras de jogo que lhe são pertinentes. Aqui se atingiria a granularidade de uma classe por regra de jogo, por exemplo, um movimento de deslocamento unitário em linha, coluna ou diagonal poderia ser multifacetado em três classes, uma para cada sentido / direção de movimento, em que as limitações e particularidades de cada uma (se aplicáveis) poderiam ser individualmente validadas e

particularizadas. Dessa forma, um jogo como o Tic Tackle iria, em sua implementação, se valer de ao menos três instâncias de objetos de regras de jogo para descrever como os jogadores se comportariam durante o jogo.

Similarmente, as regras de fim de jogo seriam separadamente classificadas. Uma classe de fim de jogo por alinhamento, por exemplo, seria encapsulada pela implementação do Jogo-da-Velha-4. Ao mesmo tempo, essa separação permite que um jogo possa facilmente se caracterizar por n situações de fim de jogo, em que cada classe de encerramento de partida conduziria a análise do tabuleiro e declararia (ou não) o término da partida. Mais do que isso, as implementações dessas classes de fim de jogo poderiam fazer estimativas do quanto cada jogador está próximo de encerrar o jogo, efetivamente qualificando o quão bom um determinado tabuleiro, ou estado, pode ser visto para cada jogador.

A computação da distância de um jogador até a vitória endereça uma carência do sistema. Como visto na Seção 4.4, a incapacidade de qualificar quão bom um movimento é limita a decisão da Inteligência Artificial a um mundo limitado pelo empate, pela derrota e pela vitória. Ora, se a cada perturbação no tabuleiro for computado o quão próximo um dos dois jogadores se encontra de vencer a partida, esse valor aproximado deve integrar a função custo aplicada aos nós da árvore Minimax. Dessa forma, o algoritmo da Inteligência Artificial irá gradativamente direcionar as jogadas para nós que aproximem a IA da vitória, com margem mínima para a indecisão e movimentos de baixa qualidade e pouca agressividade.

Portanto, uma forma de evoluir o sistema e viabilizar a implementação do não tão bem-sucedido Tic Tackle é introduzir análises intermediárias do jogo. Isso independe da decomposição das classes de tabuleiro em três grandes grupos, como discutido até aqui, e pode ser incorporado na implementação atual. Basta que cada classe herdeira de $AbstractBoard$ defina um método para sinalizar o valor de um tabuleiro para um determinado jogador. Por exemplo, a função de valor se tornaria contínua: o 0 empregado para o empate serviria apenas para situações muito específicas (e.g., início de jogo ou a inexistência de alinhamentos de 2 ou mais peças para o Tic Tackle ou Jogo-da-Velha-4), ao passo que os valores inteiros positivos seguiriam representando vitória de MAX e os inteiros negativos a vitória de MIN. Contudo, o intervalo real $(0, 1)$ poderia ser a imagem de situações intermediárias que aproximam os jogadores da vitória. Um exemplo para o Tic Tackle é: se MIN tem um alinhamento de 3 peças, então o nó da árvore Minimax terá valor $-0,75$; se MAX tem um alinhamento de 2 peças, então o nó da árvore Minimax

irá ser avaliado pela função de valor com $0,5$.

De fato, as sugestões de melhorias podem ser agrupadas por algumas características dos jogos de tabuleiro que não foram cobertas pela implementação. Dessa forma, a argumentação em favor do amadurecimento do sistema pode ser conduzida sobre aspectos observáveis em jogos de tabuleiro consolidados, especialmente aqueles presentes em Ribas (2020), os quais não foram oportunamente implementados qual os demais, presentes no Capítulo 4.

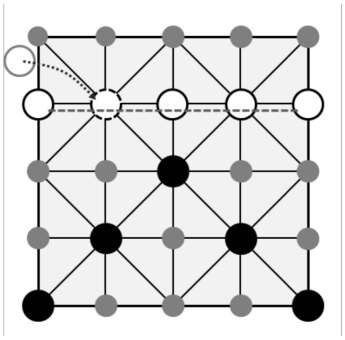
5.2 Jogos com diferentes momentos durante a partida

Muitos jogos abstratos e de estratégia preveem uma mudança na forma como os jogadores lançam mão das peças. Essa mudança pode ser vista como uma guinada na forma como o jogo é conduzido, isto é, o jogo inicia com um conjunto de regras, mas, em função do estado do tabuleiro ou da disponibilidade de peças, as regras se modificam. O chaveamento entre os conjuntos de regras pode ser visto como um novo momento de jogos.

Tomemos como exemplo o Tonkin, aqui ilustrado com base em Ribas (2020). Os dois jogadores se enfrentam sobre o tabuleiro matricial 5×5 e têm, cada um, 5 peças à disposição. Inicialmente, as peças dos jogadores estão fora do tabuleiro, ou seja, o tabuleiro começa a partida vazio. Alternadamente, cada jogador insere uma peça no tabuleiro, como mostra a Figura 5.1, tendo como objetivo o preenchimento de uma aresta inteira: se qualquer linha for totalmente preenchida, independentemente do comprimento em peças, o jogo encerra tendo como vencedor o jogador que ocupou integralmente essa aresta qualquer, conforme ilustra a Figura 5.2. No entanto, assim que ambos jogadores entrarem com todas as peças, efetivamente impossibilitando mais movimentos de inserção, então o jogo passa a admitir que cada jogador faça a movimentação das suas peças pelo tabuleiro, em passos unitários e sobre qualquer linha que conecte as posições de origem e destino do deslocamento. O objetivo do jogo, apesar da mudança nas regras de movimentação, permanece o mesmo: preencher completamente qualquer uma das 16 linhas do tabuleiro.

Claramente, o Tonkin se divide em dois momentos de jogo: primeiro a entrada de peças e depois, terminadas as 5 peças de cada jogador, a movimentação destas. Em termos práticos, o primeiro momento corresponde *ipsis litteris* ao Jogo-da-Velha-4 (Seção 4.3), ao passo que o segundo é equivalente ao Tic Tackle (Seção 4.4). A diferença, nos dois momentos, é o objetivo do jogo, pois tanto o Jogo-da-Velha-4 quanto o Tic Tac-

Figura 5.1: Movimento de posicionamento no Tonkin.

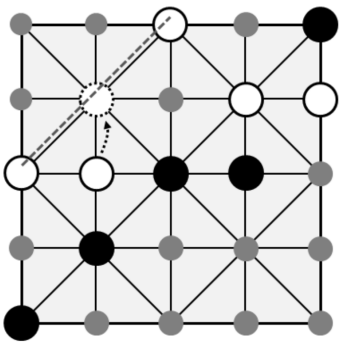


Fonte: Ribas (2020)

kle encerram com um alinhamento de 4 peças, enquanto o Tonkin requer unicamente o preenchimento total de uma linha do tabuleiro.

Aqui a proposta e a implementação apresentados esbarram na impossibilidade de suporte ao Tonkin. A vinculação estática do tipo do movimento, inserção (classe $InsertionMovement$) ou deslocamento (classe $DisplacementMovement$), à classe que estende $AbstractBoard$ inviabiliza que o sistema admita a união de dois jogos em um só. A modelagem, inclusive, deixa clara essa distinção: a implementação do Jogo-da-Velha-4 é abstraída na classe $TicTacToeIVBoard$, que estende $AbstractInsertionMovement$, enquanto o Tic Tackle é representado pela $TicTackleBoard$, herdeira de *AbstractDisplacementMovement*. Essas classes restringem, de modo inflexível, os movimentos que podem ser executados sobre o tabuleiro: o Jogo-da-Velha-4 admite unicamente a entrada de peças e o Tic Tackle permite somente a movimentação de peças. A nível de execução do sistema, qualquer tentativa de instância de movimento diferente de $InsertionMovement$ para o Jogo-da-Velha-4 ou instância que não seja $DisplacementMovement$ para o Tic Tackle irá incorrer em uma exceção por incompatibilidade de tipos.

Figura 5.2: Tonkin: fim de jogo por preenchimento de linha do tabuleiro.



Fonte: Ribas (2020)

A amarração estática de tipos ocasionada pelo polimorfismo paramétrico aplicado

à classe $AbstractBoard$ é a responsável por limitar os tipos de movimentos possíveis para um determinado jogo. Logo, para suportar alternados movimentos de jogo sobre o tabuleiro, seria necessário que $AbstractBoard < T >$ deixasse de lado o tipo paramétrico T, o qual seguramente estende $PlayerMovement$. Dessa forma, as definições de métodos que aplicam T como o tipo genérico do movimento, como mostrado no Código 5.1, passariam a consumir e retornar instâncias da classe $PlayerMovement$, como mostra o Código 5.2.

Código 5.1: Classe abstrata de tabuleiro com polimorfismo paramétrico para tipo do movimento.

```
abstract class AbstractBoard<T : PlayerMovement> {
    abstract fun play(movement: T, simulate: Boolean = false): PlayResult
    abstract fun enumeratePossibleMovements(player: Player): Iterable<T>
    abstract fun isMovementValid(movement: T): Boolean
}
```

Código 5.2: Classe abstrata de tabuleiro agnóstica ao tipo do movimento.

```
abstract class AbstractBoard {
    abstract fun play(movement: PlayerMovement, simulate: Boolean = false): PlayResult
    abstract fun enumeratePossibleMovements(player: Player): Iterable<PlayerMovement>
    abstract fun isMovementValid(movement: PlayerMovement): Boolean
}
```

Da forma posta no Código 5.2, os movimentos admitidos deveriam apenas herdar de $PlayerMovement$. Ou seja, os métodos $play$ (para executar jogadas), *enumeratePossibleMovements* (para enumerar os possíveis movimentos de um jogador) e *isMovementValid* (validador de movimentos) consumiriam e retornariam instâncias de objeto que tanto poderiam corresponder à classe $InsertionMovement$, para movimento de inserção de peça, quanto à classe $DisplacementMovement$, para deslocamento de peças. O único requerimento é que, para a computação efetiva dos movimentos, far-se-ia necessário o teste de tipo sobre o movimento para que a alteração no tabuleiro correspondesse à rotina de movimentação ou de entrada de peças.

O chaveamento entre os momentos de jogo seria de responsabilidade da classe responsável por codificar o Tonkin. Inicialmente, o jogo retornaria instâncias de *InsertionMovement* para que os jogadores pudessem entrar livremente com as peças no tabuleiro; na prática, $enumeratePossibleMovements$ retornaria instâncias de objetos do tipo *InsertionMovement*. Detectada a existência de 5 peças de cada jogador sobre o tabuleiro, a classe de implementação do Tonkin passaria a fornecer instâncias de *DisplacementMovement* como movimentos factíveis, validando a unicidade do deslocamento e verificando

se houve o preenchimento de linha, em todos os momentos, para sinalizar fim de jogo.

Fica claro, portanto, que o inconveniente da modelagem para o suporte a jogos com diferentes momentos de jogo residia na amarração de tipos entre a classe do jogo e os movimentos possíveis. Removida essa restrição, a modelagem deixa a implementação livre para controlar em que momento o chaveamento deve ocorrer, dando grande flexibilidade, inclusive, para que um jogo apresente até mais do que dois momentos diferentes.

Do ponto de vista da Inteligência Artificial, essa alteração não é impactante. Às classes que implementam o Minimax e a Poda α-β interessa apenas que sejam fornecidos movimentos possíveis de jogo para que os nós da árvore possam ser construídos, nível a nível. Logo, não haveria substancial alteração no motor da IA na eventualidade de o sistema passar a suportar diferentes momentos de jogo.

5.3 Possibilidade de aplicar mais de uma vez as regras de jogo em uma única jogada

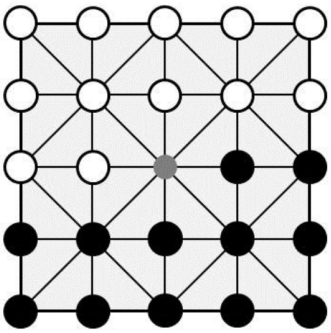
É comum nos jogos abstratos e de estratégia a repetição do mesmo movimento em uma mesma jogada. Por exemplo, no jogo de Damas é possível capturar mais de uma peça do oponente se a posição de destino do salto que consumiu a peça adversária oferecer condições para uma nova captura. Pode-se encarar esse cenário como múltiplos movimentos em uma mesma jogada ou, de forma similar mas não exatamente equivalente, que uma das regras de jogo foi aplicada mais de uma vez em uma mesma jogada.

A implementação do sistema apresentada opta, por assim dizer, pela primeira opção. Ou seja, se o jogo de Damas fosse codificado dentro da arquitetura esmiuçada nos Capítulos 3 e 4, então os múltiplos saltos seriam considerados como múltiplos movimentos executados sobre o tabuleiro. Como a modelagem prevê a alternância movimento a movimento, isto é, ora um, e somente um, movimento de MAX, ora um, e somente um, movimento de MIN, a flexibilidade de aceitar a repetição de movimentos em uma mesma jogada forçaria uma refatoração do sistema.

Tomando como pano de fundo mais um dos 21 jogos de Ribas (2020), podemos analisar a refatoração olhando para o jogo Alquerque. Conduzido sobre o ubíquo tabuleiro matricial 5×5, o Alquerque concede 12 peças para cada jogador, as quais iniciam sobre as extremidades superior e inferior do tabuleiro, deixando apenas a posição central livre, como mostra a Figura 5.3. Cada jogador pode movimentar suas peças em passos unitários e para qualquer posição vizinha, desde que conectada por uma aresta com posição de origem do movimento, exatamente como ilustrado na Figura 5.4. Similarmente ao jogo

de Damas, a captura é feita pelo salto sobre uma peça adversária, contanto que a posição de destino do salto esteja livre. Ainda, se uma potencial oportunidade de captura de peça se apresentar logo após um salto, então o jogador pode continuar executando saltos até que não seja mais possível prosseguir. Nesse caso, n saltos capturam n peças, vide a Figura 5.5. O jogo termina quando um dos jogadores elimina todas as peças adversárias do tabuleiro.

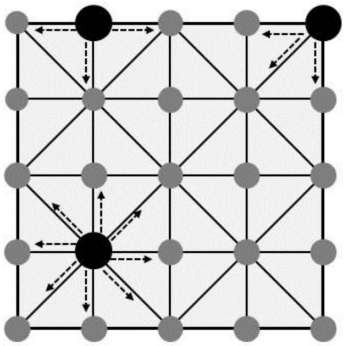
Figura 5.3: Posição inicial do Alquerque.



Fonte: Ribas (2020)

No sistema idealizado e desenvolvido, a dinâmica de alternância de movimentos únicos se faz especialmente presente na Inteligência Artificial. É na expansão dos nós da árvore que a correspondência 1 movimento para 1 jogada é verificada, como mostram os Algoritmos 3 e 5. Contudo, ao analisarmos o Código 5.1, e mesmo a hipotética refatoração proposta pelo Código 5.2, notamos que a enumeração dos possíveis movimentos de um jogador, definida pelo método $enumeratePossibleMovements$, retorna um objeto do tipo $Iterable$, i.e., uma coleção iterável, em que cada elemento é uma instância de $PlayerMovement$ - um movimento de um jogador.

Figura 5.4: Movimentação no Alquerque.



Fonte: Ribas (2020)

Ora, se cada elemento do conjunto de jogadas factíveis é uma instância de objeto que representa um movimento, certamente não é possível, numa mesma jogada,

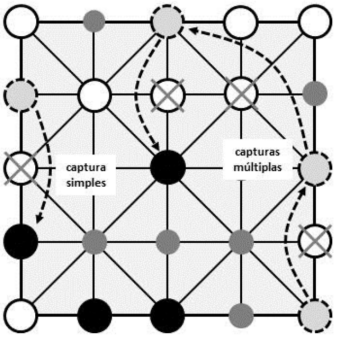
aplicar mais de um movimento. Aqui se pode notar uma relação implícita e, no cenário aqui posto, indesejada: a restrição do número de movimentos possíveis a cada jogada está condicionada ao fato de cada elemento contido na coleção retornada por $enumeratePossibleMovements$ ser *uma* instância de um movimento. Portanto, se o tipo de cada elemento da coleção projetada por $enumeratePossibleMovements$ passar a ser também um conjunto de instâncias de movimento, a leitura que a Inteligência Artificial fará é que se faz necessário iterar sobre esse conjunto de movimentos. Ao chegar no último elemento, a jogada foi completada, permitindo, assim, que mais de um movimento seja executado por jogada.

Código 5.3: Refatoração para suporte a n movimentos por jogada.

```
1  abstract class AbstractBoard {
2      // ...
3      abstract fun enumeratePossibleMovements(
4          player: Player
5      ): Iterable<Iterable<PlayerMovement>>
6      // ...
7  }
```

O Código 5.3 mostra exatamente essa alteração. Necessária a ressalva de que o método que computa uma jogada ($play$) e o que valida um movimento ($isMovementValid$) não requerem alteração, porquanto seguem a operar sobre sobre uma única instância de movimento. Todavia, apenas a alteração mostrada no Código 5.3 é insuficiente: tanto o Minimax quanto a Poda α-β devem ser alterados para que, na expansão dos tabuleiros possíveis, seja previsto um conjunto de movimentos por iteração, em vez de apenas um. Para exemplificar essa alteração, podemos tomar como base o Algoritmo 15 e alterá-lo para que se obtenha uma solução igual a apontada pelo Algoritmo 16.

Figura 5.5: Capturas no Alquerque.



Fonte: Ribas (2020)

Naturalmente, a estrutura de um nó da árvore - classe $TreeNode$ - seria forçosamente alterada. Em vez de possuir um movimento gerador, um nó da árvore teria um

conjunto de movimentos geradores. De fato, a terminologia teria de ser também readequada: um nó da árvore teria uma jogada geradora. Essa jogada consistiria, portanto, em um conjunto de movimentos responsável por dar origem ao estado resultante da aplicação de cada movimento pertencente a essa coleção.

Algoritmo 16 Minimax (aplicável também a Poda α-β) com suporte a mútiplos movimentos por jogada.

```
1:  function EXPANDIR(T, nodo, j)
2:      if nodo = null then
3:          nodo ← novo nó da árvore Minimax
4:      end if
5:      movimentos ← conjunto de movimentos permitidos ao jogador j sobre o tabuleiro T
6:      proximoJogador ← MAX se j = MIN senão MIN
7:      for m ∈ movimentos do                    ▷ m é um conjunto de movimentos.
8:          filho ← novo nó da árvore Minimax
9:          Retém filho como nó filho de nodo
10:         for m' ∈ m do
11:             jogar(T, m')
12:         end for
13:         expandir(T, filho, proximoJogador)
14:         for m' ∈ m do
15:             desfazerJogada(T, m')
16:         end for
17:     end for
18: end function
```

Com efeito, as alterações no Minimax / Poda α-β seriam pontuais. As linhas 10-12 garantem a iteração sobre o conjunto de movimentos retornado, aplicando-os um a um sobre o tabuleiro. Dessa forma, na expansão de um nó que eventualmente se caracterize por mais de um movimento, o tabuleiro T será passado com todos os movimentos devidamente computados. Nas linhas 14-16, mantendo a proposta de desfazer uma jogada sobre o tabuleiro, aventada pela maior eficiência observada no Tic Tackle, deixa-se o tabuleiro no estado desejado para a expansão do próximo nó.

As alterações necessárias para que o sistema suporte múltiplos movimentos por jogada, portanto, perturbam tanto as abstrações responsáveis pela representação computacional dos jogos de tabuleiro quanto os algoritmos de decisão competitiva da Inteligência Artificial. Essa refatoração não é assombrosa, contudo, cuidados seriam necessários quanto ao desempenho na derivação da árvore do Minimax, visto que a visita aos estados passa a envolver *loops* adicionais para contemplar múltiplos movimentos.

5.4 Movimento em 2 momentos de jogo

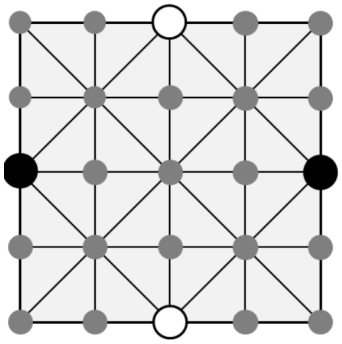
Interessantemente, alguns jogos de tabuleiro combinam, de certa forma, diferentes momentos de jogo com mais de um movimento por jogada. Assim, essas variantes de jogos abstratos e de estratégia aplicam alguns conceitos externados nas Seções 5.2 e 5.3, mas com uma diferença fundamental em relação à possibilidade de mais de um movimento por jogada: tais movimentos não são apenas possíveis e sim obrigatórios. Via de regra, um diferente momento de jogo é representado por um segundo movimento dentro da mesma jogada, sendo admissível até mesmo dois ou mais movimentos obrigatórios.

Essa noção é sutilmente distinta do que foi discutido na Seção 5.3. Afinal, os ditos múltiplos movimentos por jogada não necessariamente ocorreriam a cada turno do jogador. São, pois, movimentos condicionados ao estado do tabuleiro, como por exemplo quando um salto para captura de peça se torna uma possibilidade no jogo Alquerque. Constatar essa peculiaridade é fundamental, porque, como veremos adiante, a solução para os jogos que têm movimento em 2 momentos de jogo é substancialmente diferente do que foi sugerido na Seção 5.3.

O jogo Amazonas, parte dos 21 jogos cobertos por Ribas (2020), traz uma proposta que se encaixa perfeitamente no tópico em discussão. Tendo como pano de fundo o tabuleiro matricial 5×5, o Amazonas objetiva impossibilitar que o adversário execute movimentos (jogo de bloqueio). Cada jogador pode ter entre 1 e 2 peças, as quais podem iniciar o jogo dentro ou fora do tabuleiro. A Figura 5.6, por exemplo, propõe uma variante do Amazonas com 2 peças por jogador, inicialmente já posicionadas sobre o tabuleiro. Uma vez presente no tabuleiro, uma peça pode ser movida por n casas sobre uma aresta do tabuleiro, desde que as posições estejam livres (saltos não são permitidos). Esse deslocamento, porém, é apenas o primeiro movimento da jogada; a este se segue imediata e indispensavelmente o segundo movimento, que é o lançamento de uma peça de bloqueio, a qual deve ter como ponto de origem a posição de destino da peça deslocada no primeiro movimento. O lançamento é também feito sobre uma aresta do tabuleiro e pode se estender por n casas livres (novamente, saltos não são permitidos). A posição sobre a qual a peça de bloqueio torna-se imediatamente indisponível, não sendo possível usá-la para mais nenhuma jogada até o fim da partida. A Figura 5.7 transmite com clareza a sequência de movimentos do Amazonas. Quando um jogador não tiver mais movimentos para executar, o jogo termina com a vitória do adversário que o bloqueou.

A implementação detalhada no Capítulo 4 é incapaz de absorver um jogo como

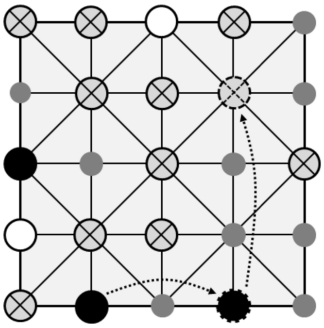
Figura 5.6: Posição inicial do Amazonas com 2 peças por jogador.



Fonte: Ribas (2020)

o Amazonas. Ao mesmo tempo, as sugestões evolutivas da Seção 5.3 são insuficientes para que jogos com movimentos em 2 ou mais momentos de jogo sejam codificados. Tal fato se apresenta em decorrência da sutil distinção destacada nesta seção: a possibilidade de mais de um movimento por jogada é inerentemente diferente da garantida ocorrência de um segundo movimento por jogada, o qual, de fato, define com muito mais expressividade as regras de jogo. Mais ainda: jogos com movimentos em 2 momentos, como o Amazonas, requerem que o Minimax / Poda α-β preveja as possibilidades em dois níveis recursivos: primeiro para todas as possibilidades do movimento 1 e depois, para cada um desses movimentos atingíveis, as possibilidades do movimento 2. Afinal, visto que 2 movimentos por jogada são impreteríveis, o estado do tabuleiro após o turno de um jogador só será efetivo se esses 2 movimentos forem executados.

Figura 5.7: Os 2 momentos de jogos do Amazonas.



Fonte: Ribas (2020)

Antes, no entanto, a análise da refatoração deve se voltar para as estruturas de classes definidas no sistema. Desde a Seção 5.3 restou claro que a abstração de movimento, i.e., $PlayerMovement$, parece insuficiente para representar uma jogada. Faz-se necessária a introdução de uma classe que represente uma jogada, onde esta é caracterizada por um conjunto de movimentos M, tal que $\#M > 0$, ou seja, uma jogada tem ao me-

nos um movimento que é uma instância de $PlayerMovement$. Aproveitando, portanto, a sugestão da Seção 5.2, não há amarração estática de tipos por polimorfismo paramétrico, portanto cada elemento pertencente a M pode ser um $InsertionMovement$ ou um $DisplacementMovement$. Além do conjunto de movimentos, uma jogada tem também um jogador responsável por executá-la, logo, retém-se na classe uma instância de objeto $Player$. O Código 5.4 mostra como seria essa classe, chamada de $Move$.

Código 5.4: Classe que representa uma jogada.

```
1 class Move {
2     val movements: Iterable<PlayerMovement> = mutableListOf()
3     var player: Player
4 }
```

A propriedade $movements$ representa o conjunto de movimentos que compõem uma jogada, ao passo que a propriedade $player$ referencia o jogador responsável por lançar mão dessa jogada. A introdução dessa classe induz uma alteração sobre a classe base para a implementação de tabuleiros, a $AbstractBoard$, que deve, então, passar a enumerar não mais movimentos possíveis por jogador, mas sim jogadas, i.e., instâncias de $Move$. Essa modificação está contemplada no Código 5.5. Fundamental destacar que a tipificação do método que enumera as possíveis jogadas para um participante do jogo, i.e., $enumeratePossibleMovements$, corresponde a um conjunto em que cada elemento é um conjunto de movimentos, ou seja, as possíveis jogadas podem ser formalizadas como o conjunto de k jogadas $J = \{M_1, M_2, \ldots, M_k\}$, onde cada conjunto tem p movimentos: $M_{i \in [k]} = \{m_1, m_2, \ldots, m_p\}$, tal que $m_{j \in [p]}$ é uma instância de $PlayerMovement$ - um movimento de inserção ou de deslocamento. Aplicando essa formalização para o jogo Amazonas, temos $k = 2$, logo $J = \{M_1, M_2\}$. M_1 representa o primeiro movimento de jogo, isto é, um único movimento de deslocamento da peça sobre o tabuleiro. Tem-se $M_1 = \{a\}$, onde a é um movimento do tipo deslocamento ($DisplacementMovement$). M_2 é responsável por materializar o segundo momento de jogo, portanto $M_2 = \{b\}$, em que b é um movimento de inserção ($InsertionMovement$), o qual representa o lançamento da peça de bloqueio. Assim, essa proposta de solução ataca tanto os múltiplos movimentos por jogada, mantendo a sugestão da Seção 5.3, quanto os várias movimentos em diferentes momentos de jogo.

Código 5.5: Refatoração da classe base de tabuleiros para suporte a jogadas.

```
abstract class AbstractBoard {
    abstract fun play(move: Move, simulate: Boolean = false): PlayResult
    abstract fun enumeratePossibleMovements(player: Player): Iterable<Iterable<Move>>
    abstract fun isMovementValid(move: Move): Boolean
}
```

Essa refatoração, no entanto, deve se estender também à classe que representa o nó da árvore Minimax. Desta feita, um $TreeNode$ não será mais identificado por um movimento gerador ou um conjunto de movimentos geradores, como visto na Seção 5.3: o nó passará a ser mapeado pelas jogadas geradoras, isto é, por uma coleção de jogadas (conjunto de instâncias de $Move$). Como a abstração de $Move$ ataca a limitação discutida na Seção 5.3, a classe representante do nó da árvore Minimax não deixa de lado a capacidade de a Inteligência Artificial tomar decisões sobre jogos com múltiplos movimentos em uma mesma jogada.

O grande entrave que se pode antever para esses casos reside justamente na implementação da Inteligência Artificial. Diferentemente dos múltiplos movimentos por jogada, que são ordinariamente executados sobre o tabuleiro, os 2 momentos de jogo proposto pelo Amazonas tem como consequência mais um nível de avaliação das possibilidades de jogo, como mostra o Algoritmo 17, o qual teria de ser absorvido pela etapa de expansão de nós da árvore Minimax, sendo também aplicável a Poda α-β.

Algoritmo 17 Laço para avaliação do Minimax / Poda α-β com 2 momentos de jogo por jogada.

```
1: movimentosMomento_1 ← movimentos possíveis do primeiro momento
2: for m_1 ∈ movimentosMomento_1 do
3:     Computar m_1 no tabuleiro
4:     movimentosMomento_2 ← movimentos possíveis pela computação de m_1 sobre o tabuleiro
5:     for m_2 ∈ movimentosMomento_2 do
6:         Computar m_2 no tabuleiro
7:     end for
8: end for
```

Sob a ótica do jogo Amazonas, o impacto dessa alteração é significativo, pois o aumento da largura da árvore Minimax é brutal. A cada nível expandido, devem ser computados todos os movimentos possíveis do primeiro momento de jogo e, para cada um destes, devem ser avaliados os movimentos factíveis para a aplicação do segundo momento de jogo. Só depois dessa avaliação consecutiva um nó poderia ser considerado apto a integrar a árvore, visto que a avaliação do tabuleiro dar-se-ia unicamente mediante

a integralização da jogada composta por 2 momentos de jogo distintos, os quais devem ser previstos pela Inteligência Artificial. Logo, se para um determinado tabuleiro há m movimentos do primeiro momentos aplicáveis, e para cada um destes há n movimentos que satisfazem o segundo momento, um nó terá $m \times n$ nodos filhos. Deve-se considerar, ainda, a perturbação na performance do sistema, pois $m \times n$ movimentos devem ser executados sobre o tabuleiro para que o nó possa ser avaliado pela função de valor.

Naturalmente, de modo similar ao que se viu na Seção 4.3, o Amazonas caminha para a compactação da árvore, pois as possíveis jogadas vão sendo restringidas pela ocupação do tabuleiro no lançamento das peças de bloqueio. Contudo, a nível de performance do sistema, é inviável expandir a contento a árvore Minimax, mesmo com a Poda α-β. Nesse caso, o controle de explosão combinatória defendido na Seção 4.3 seria provavelmente insuficiente, precisando ser socorrido por abruptas interrupções na expansão, por exemplo, por *timeout*. Além disso, visto que o jogo vai aos poucos afunilando a liberdade de movimentação dos jogadores, as primeiras decisões da IA poderiam ser aleatórias, na suposição de que ao longo da partida eventuais más escolhas fossem sendo corrigidas pelo aumento da capacidade de expansão da árvore, consequência da diminuição paulatina de estados atingíveis.

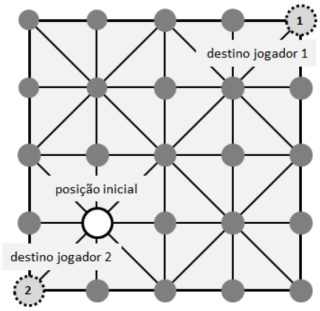
5.5 Jogos com múltiplos objetivos

Muitos jogos de tabuleiro definem mais de um objetivo de jogo, isto é, existe mais de uma forma de um jogador se consagrar como vencedor. Embora muitos jogos permitam que um jogo seja encerrado por mais de uma regra, nenhuma das implementações do Capítulo 4, por exemplo, apresentou essa característica. Tal fato motiva que se reflita a respeito da facilidade com que jogos com múltiplos objetivos se encaixariam na proposta deste trabalho.

Novamente, os 21 jogos de Ribas (2020) fornece um rico exemplo para introduzir o conceito. O jogo Rastros, conduzido sobre o bem conhecido tabuleiro matricial 5×5, concede apenas 1 peça, que deve ser compartilhada pelos dois jogadores. Cada jogador é responsável por guardar uma posição do tabuleiro, a qual representa o destino do adversário. As posições de destino de cada um dos jogadores estão localizadas em extremos do tabuleiro. O jogador que inicia a partida tem, então, a peça posicionada o mais longe possível do ponto de destino que a ele se impõe, como mostrado na Figura 5.8. Cada jogador, alternadamente, faz um movimento unitário para uma posição vizi-

nha (conectada por uma aresta do tabuleiro), objetivando chegar à posição protegida pelo oponente. Ao realizar esse movimento, a posição na qual a peça estava posicionada (a origem do movimento) é imediatamente bloqueada, não sendo mais permitido, em jogadas futuras, posicionar a peça compartilhada sobre ela. A Figura 5.9 mostra a anulação de posições após uma pequena sequência de jogadas. O jogador que não puder mais executar movimentos em decorrência dos bloqueios perde o jogo. Portanto, o Rastros admite dois objetivos: a chegada à posição de destino (protegida pelo jogador adversário) e o bloqueio de movimentos.

Figura 5.8: Posição inicial do Rastros.



Fonte: Ribas (2020)

De acordo com as implementações descritas no Capítulo 4, o Tapatan, o Jogo-da-Velha 4 e o Tic Tackle tinham um único (e similar) objetivo: alinhamento de peças. No caso do Tapatan, um 3-alinhamento, e para o Jogo-da-Velha 4 e o Tic Tackle, um 4-alinhamento (o Tic Tackle admite, ainda, um alinhamento fora das linhas do tabuleiro). O objetivo desses e de eventuais futuros jogos é feito pela definição do corpo do método $getWinner$, da classe base de jogos $AbstractBoard$, como mostra o Código 5.6.

Código 5.6: Interface do método abstrato para sinalizar fim de jogo.

```
1 abstract class AbstractBoard<T : PlayerMovement>() {
2     // ...
3     abstract fun getWinner(): Player?
4     // ...
5 }
```

O retorno do método $getWinner$ é uma instância de $Player$. A ocorrência do operador $?$ indica que essa instância pode ser nula, isto é, uma referência $null$. Logo, se $getWinner$ retornar $null$, então não há jogador vencedor. Ao retornar uma instância de $Player$ não nula, o resultado do método deve implicar no fim do jogo, e a instância de $Player$ fornecida como resultado indica o jogador que venceu a partida, i.e., MIN ou MAX.

Algoritmo 18 Exemplo de fim de jogo com $getWinner$ para o Jogo-da-Velha 4.

```
1: function GETWINNER
2:     if tabuleiro apresenta um 4-alinhamento sobre uma aresta then
3:         return jogador que realizou o 4-alinhamento
4:     end if
5:     return null                                          ▷ Não há vencedor.
6: end function
```

Para jogos com apenas 1 objetivo, o Algoritmo 18 traz um exemplo de implementação. O jogo analisado é o Jogo-da-Velha 4, entretanto, é fácil de estender esse algoritmo para os demais jogos analisados no Capítulo 4. Para jogos com mais de um objetivo de jogo, tomando como exemplo o Rastros, o Algoritmo 19 mostra uma implementação possível.

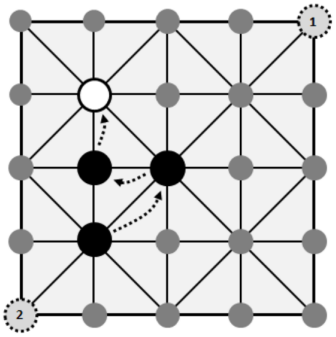
Algoritmo 19 Exemplo de fim de jogo com $getWinner$ com múltiplos objetivos: Rastros.

```
1: function GETWINNER
2:     if MIN chegou à posição de destino destino_MIN then
3:         return MIN
4:     end if
5:     if MAX chegou à posição de destino destino_MAX then
6:         return MAX
7:     end if
8:     if MAX não tem mais movimentos possíveis then
9:         return MIN
10:    end if
11:    if MIN não tem mais movimentos possível then
12:        return MAX
13:    end if
14:    return null                                          ▷ Não há vencedor.
15: end function
```

Para o Rastros, os múltiplos objetivos estão distribuídos no Algoritmo 19 pelos testes (didaticamente colocados). As linhas 2-7 externam o objetivo de um dos jogadores chegar à posição de destino. As linhas 8-13 representam a aplicação do objetivo de bloqueio de movimentos do adversário. Quando um desses testes se verificar verdadeiro, então o jogo encerra, independentemente do objetivo que o terminou.

O sistema proposto neste trabalho, portanto, não encontraria dificuldades em suportar jogos com vários objetivos de jogo. Como os objetivos e regras são de responsabilidade da classe de implementação do jogo de tabuleiro, o sistema se mantém genérico e agnóstico aos gatilhos que determinam o encerramento da partida.

Figura 5.9: Bloqueio de jogador no Rastros.



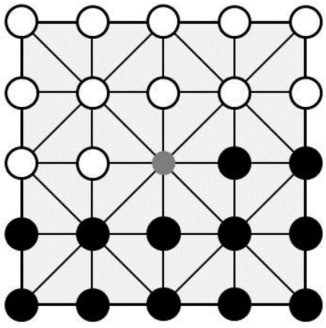
Fonte: Ribas (2020)

5.6 Promoção de peças

Os jogos vistos até aqui, independentemente de terem sido ou não alvo da implementação do Capítulo 4, tratam todas as peças como iguais. Nenhuma peça tem um papel em particular, o que contrasta, por exemplo, com o Xadrez. Apesar disso, alguns jogos podem iniciar com peças, por assim dizer, anônimas, mas que a partir de determinado estado do jogo passam a ter uma função especial frente às demais.

Um dos 21 jogos apresentados por Ribas (2020), o Fetaix pertence à classe de jogos que preveem a chamada promoção de peças, isto é, peças inicialmente ordinárias, indistinguíveis das demais, que são promovidas se atingido algum critério específico de jogo. O Fetaix, um jogo de estratégia em alguns aspectos semelhante ao Alquerque e ao jogo de Damas, sendo similar a este, por exemplo, na posição inicial do tabuleiro (Figura 5.10), propõe a peça Mullah. No contexto do Fetaix, se uma peça atinge o extremo oposto do tabuleiro em relação às posições iniciais das peças do jogador, então ela é promovida a Mullah, e, diferentemente de peças *não* Mullah, pode ser livremente movida pelo tabuleiro.

Figura 5.10: Posição inicial do Fetaix.

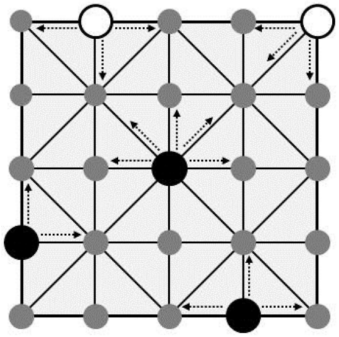


Fonte: Ribas (2020)

Esse diferencial é desconsiderado no sistema proposto neste trabalho. Natural-

mente, a rotulação da peça é irrelevante para essa análise e pode ser facilmente implementada, por exemplo, ao adicionar uma propriedade do tipo $string$ que representa a distinção aplicada a peça (e.g., Mullah para uma peça do Fetaix). A relevância na promoção de peças se dá pela concessão particular de regras de movimentação para uma peça específica. De fato, uma peça promovida a Mullah durante uma partida de Fetaix é agraciada com movimentos possíveis unicamente a ela; logo, existem 2 conjuntos de movimentos em sinergia durante a partida: um para as peças ordinárias e outro, mais poderoso, para a peça Mullah. Esse contraste é nítido ao observarmos as Figuras 5.11 e 5.12.

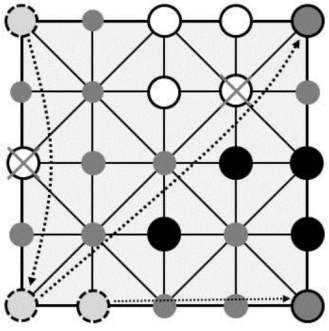
Figura 5.11: Movimentação no Fetaix (sem Mullah).



Fonte: Ribas (2020)

A limitação notada para o sistema proposto reside na inegável limitação de uma peça. Da forma como esta foi abstraída, ela é meramente um artefato que ocupa uma posição no tabuleiro; equivalentemente, uma peça representa apenas uma forma de clamar uma posição do tabuleiro para um determinado jogador. Rotular, pois, essa posição. Toda e qualquer interação de um jogador com o tabuleiro é feita não por qual peça será movida, mas qual posição será conquistada. A peça é, em si, apenas um meio de ocupar uma casa do tabuleiro.

Figura 5.12: Movimentação de uma peça Mullah no Fetaix.



Fonte: Ribas (2020)

O sistema proposto, além de despir as peças de qualquer semântica, usa um sistema global de regras de jogo. Isto é, as regras definidas para uma partida se aplicam a toda e qualquer jogada, e os movimentos de um jogador são realizados pelas coordenadas do tabuleiro. A incidência de uma peça sobre essas coordenadas é, sim, uma consequência, mas o ato de realizar um movimento nem sequer referencia a entidade peça.

Algoritmo 20 Sem promoção de peças: computação de uma jogada de movimentação.

```
1:  function COMPUTARJOGADA(jogador, tabuleiro)
2:      movimentos ← movimentos possíveis de jogador sobre tabuleiro
3:      m_input ← solicitar movimento de jogador
4:      if m_input ∉ movimentos then
5:          erro
6:      end if
7:      (linha_origem, coluna_origem) ← origem(m_input)
8:      (linha_destino, coluna_destino) ← destino(m_input)
9:      peça ← tabuleiro[linha_origem][coluna_origem]
10:     tabuleiro[linha_destino][coluna_destino] ← peça
11:     liberarPosicao(tabuleiro[linha_origem][coluna_origem])
12: end function
```

O Algoritmo 20 mostra como uma jogada é computada conforme a implementação detalhada no Capítulo 4. A jogada ilustrada, um movimento de deslocamento, consiste essencialmente em fazer uma troca entre elementos da matriz que representa o tabuleiro. As linhas 9-10 deixam claro a irrelevância da peça, pois toda a jogada é computada em função das coordenadas do tabuleiro. A peça, como deixa claro a linha 9, é apenas uma forma de sinalizar que uma posição (i, j) do tabuleiro está ocupada. Alem disso, a linha 2 enfatiza que os movimentos possíveis são enumerados para o tabuleiro de uma forma global: as regras de jogo são igualitariamente aplicadas a todas as peças, restando evidente a impossibilidade de promoção de qualquer uma dessas peças.

Algoritmo 21 Promoção de peças: computação de uma jogada de movimentação.

```
1:  function COMPUTARJOGADA(jogador, tabuleiro)
2:      p_input ← solicitar posição sobre a qual jogador quer executar movimento
3:      peça ← peça nas coordenadas de p_input sobre tabuleiro
4:      movimentos ← movimentos possíveis para a peça peça
5:      if movimentos = ∅ then
6:          erro
7:      end if
8:      m ← solicitar movimento m' a jogador tal que m' ∈ movimentos
9:      jogar(m, tabuleiro)
10: end function
```

Em contraste, o Algoritmo 21 mostra uma alternativa de execução de jogada que permite a granularização de movimento por peça. A dinâmica é diametralmente diferente daquela vista no Algoritmo 20: o jogador interage 2 vezes durante a computação da jogada: a primeira para decidir sobre qual posição deseja atuar e a segunda para finalmente decidir, a partir do conjunto de movimentos disponíveis para a peça que ocupa a posição do primeiro passo, qual movimento será efetivamente feito sobre o tabuleiro. Tal granularização, de fato, permite que as peças possam, a seu tempo, receber mais de uma promoção; colocando de outra forma, cada peça pode ter seu próprio conjunto de regras. Logo, com a alteração proposta no Algoritmo 21, o Fetaix, ao ser implementado no contexto do sistema proposto, não encontraria impedimentos para a promoção de uma peça a Mullah.

O Minimax e a Poda α-β, naturalmente, teriam de ser refatorados para que a expansão de nós da árvore considerasse essa consulta adicional ao tabuleiro: a primeira para recuperar as posições sobre os quais o jogador do nível (MIN ou MAX) têm peças, e a segunda para enumerar os conjuntos de movimentos realizáveis por essas mesmas peças. A nível de predição, nada é afetado: o algoritmo escolheria a mesma jogada tanto no caso da computação de movimento mostrada no Algoritmo 20 quanto na proposta do Algoritmo 21. A diferença, aqui, residiria unicamente na incidência de operação adicional para a consulta de movimentos por peça, em vez de a enumeração geral consolidada na implementação.

6 CONSIDERAÇÕES FINAIS

A representação computacional de jogos abstratos e de estratégia é uma desafiadora tarefa em um contexto generalista. A especificidade da implementação de um único jogo de tabuleiro, conquanto igualmente dotada de suas próprias idiossincrasias, requer menos precauções durante o processo de desenvolvimento. A abrangência de diversos jogos por uma mesma plataforma é, de fato, um processo caracterizado por constantes adaptações e novas visitas à arquitetura do sistema, expandindo, a cada introdução de jogo, não apenas o leque de possibilidades do sistema, mas também a percepção de como diferentes jogos de tabuleiro se relacionam por particularidades nem sempre triviais de se perceber.

No tempo em que novas propostas de jogos abstratos são lançadas, há que se preocupar em como uma Inteligência Artificial Minimax será capaz de se adaptar tanto à dinâmica proporcionada pelos novos jogos quanto de manter a hegemonia que dela se espera. O extremo cuidado requerido por uma pontual otimização no motor da IA se faz absolutamente indispensável, pois é fundamental que uma perturbação na arquitetura da Inteligência Artificial não comprometa aquilo que se tem estabelecido como funcional, isto é, o sucesso consolidado na implementação de jogos pregressos. Não foi ocorrência única ou isolada, durante o desenvolvimento deste trabalho, a imposição de escolhas que mostraram a relação muitas vezes inversa entre otimizar o núcleo dos jogos no sistema e potencializar ainda mais as tomadas de decisão da Inteligência Artificial Minimax.

A adoção de um modelo inicialmente abstrato foi fundamental para que se adquirisse o conhecimento de relações não triviais no contexto dos jogos de tabuleiro. A libertação do vício causado pela visão de produto paga dividendos quando passamos a reconhecer a importância do questionamento ininterrupto, da experimentação de ideias e da consciência de que mais importante do que obter um resultado tangível ao final é colher o aprendizado e o conhecimento adquiridos ao longo do processo de desenvolvimento. Nesse sentido, a etapa de validação foi essencial para o nascimento posterior da percepção de quais pontos foram satisfeitos e quais ficaram para trás, as causas dos sucessos e insucessos.

A estratégia proposta, discutida e validada serve como um interessante caso de estudo para a implementação de uma plataforma digital de jogos de tabuleiro. Ao se inserir a Inteligência Artificial Minimax nesse cenário, a despeito das já bem estabelecidas bases dos Algoritmos de Decisão Competitiva, tem-se uma visão singular de como ainda

é relevante estressar ideias amplamente difundidas e estudadas. Fica claro, no entanto, que a proposta definida neste trabalho, ainda que consistente e bem alicerçada, não deve ser encarada sem as considerações levantadas na análise da receptividade a outras características presentes em diversos jogos de tabuleiro.

Se, por um lado, os pontos de melhoria se mostram imprescindíveis na condução de futuros trabalhos relacionados, por outro, esses mesmos pontos não são brutalmente críticos. Isto é, se considerados em uma nova proposta ou mesmo implementação de jogos de tabuleiro em que a IA é um competidor, não apresentam mudanças disruptivas no modelo estabelecido. A abstração, a preocupação com a escalabilidade do sistema e a intenção de que a Inteligência Artificial Minimax se mantenha agnóstica ao jogo estão muito bem colocadas na proposta, fato refletido pela etapa de implementação. As posteriores readequações sobre a qual este trabalho se debruçou, ainda que representem significativa refatoração tanto no modelo quanto na codificação, em nada ferem o paradigma adotado desde o início do trabalho.

Assim, a contribuição deste trabalho não se limita apenas à proposta de um sistema de jogos abstratos e de estratégia, jogáveis por uma Inteligência Artificial Minimax, característico pela flexibilidade de incorporar novas implementações de jogos. Resta claro, por tudo que foi discutido até aqui, o quão fecundo é o universo dos jogos de tabuleiro dentro da computação, seja pelo rico catálogo de jogos conhecido, seja pelo potencial transformador que os jogos têm mesmo quando minimamente alterados. Aliado aos desafios de decisão competitiva pelos quais Inteligência Artificial Minimax se distingue, esse universo, portanto, passa a ser ao mesmo tempo fascinante e de extrema relevância na computação.

REFERÊNCIAS

AHO, A. V. et al. **Compilers - Principles, Techniques, and Tools**. 2. ed. Massachusetts, EUA: Pearson/Addison Wesley, 2006. 381–384 p. ISBN 0321486811.

CAMPBELL, M. S.; MARSLAND, T. A comparison of minimax tree search algorithms. **Artificial Intelligence**, v. 20, n. 4, p. 347 – 367, 1983. ISSN 0004-3702.

CASTRO, R. M.; NOWAK, R. D. Minimax bounds for active learning. **IEEE Transactions on Information Theory**, v. 54, n. 5, p. 2339–2353, 2008.

GIORDANI, L. F.; RIBAS, R. P. **Jogos Lógicos de Tabuleiro: Jogos de Bloqueio e Alinhamento**. 2014. Disponível na Internet: <https://www.inf.ufrgs.br/lobogames/wp-content/uploads/2015/09/jogos_modulo1_texto.pdf>.

MEYER, B. **Object-Oriented Software Construction**. 1. ed. New Jersey, EUA: Prentice-Hall, 1988. ISBN 9780136290315.

RAUTER, A. C. **Plataforma computacional de apoio ao projeto de extensão jogos lógicos de tabuleiro**. 2014. Monografia (Bacharelado em Engenharia da Computação), UFRGS (Universidade Federal do Rio Grande do Sul), Porto Alegre, Brasil.

RIBAS, N. L. **Jogos de tabuleiros movimentando a escola**. 2019. Monografia (Licenciatura em Pedagogia), UFRGS (Universidade Federal do Rio Grande do Sul), Porto Alegre, Brasil.

RIBAS, R. P. **21 Jogos Abstratos e de Estratégia no Mesmo Tabuleiro**. Porto Alegre, Brasil: Editora Metamorfose, 2020. ISBN 9786586623178.

RUSSELL, S.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. New Jersey, EUA: Pearson, 2010. 161–195 p. (Prentice Hall Series in Artificial Intelligence). ISBN 9780136042594.

SEBESTA, R. W. **Concepts of Programming Languages**. 10. ed. New Jersey, EUA: Addison-Wesley, 2012. 473–574 p. ISBN 9780131395312.

STOCKMAN, G. A minimax algorithm better than alpha-beta? **Artificial Intelligence**, v. 12, n. 2, p. 179 – 196, 1979. ISSN 0004-3702.

SUBRAMANIAM, V. **Programming Kotlin**. 1. ed. Massachusetts, EUA: O'Reilly, 2019. ISBN 9781680506358.