CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CURSO DE ENGENHARIA DE COMPUTAÇÃO

DAMASCODE: UM JOGO DE DAMAS COM INTELIGÊNCIA ARTIFICIAL PROGRAMÁVEL

THIAGO MENDES VIEIRA

Orientador: João Fernando Machry Sarubbi
Centro Federal de Educação Tecnológica de Minas Gerais – CEFET-MG

BELO HORIZONTE
DEZEMBRO DE 2014

THIAGO MENDES VIEIRA

DAMASCODE: UM JOGO DE DAMAS COM INTELIGÊNCIA ARTIFICIAL PROGRAMÁVEL

Dissertação de Trabalho de Conclusão de Curso apresentado ao Centro Federal de Educação Tecnológica de Minas Gerais como parte dos requisitos exigidos para obtenção do título de Engenheiro da Computação.

Área de concentração: Inteligência Artificial.

Orientador: João Fernando Machry Sarubbi
Centro Federal de Educação Tecnológica de Minas Gerais – CEFET-MG

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CURSO DE ENGENHARIA DE COMPUTAÇÃO
BELO HORIZONTE
DEZEMBRO DE 2014

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CURSO DE ENGENHARIA DE COMPUTAÇÃO
AVALIAÇÃO DO TRABALHO DE CONCLUSÃO DE CURSO

Aluno:	Thiago Mendes Vieira
Título do Trabalho:	DamasCode: Um jogo de damas com inteligência artificial programável
Data da defesa:	20/01/2015
Horário:	10:00 h às 12:00 h
Local da defesa:	Departamento de Computação - Prédio 17 do Campus II

O presente Trabalho de Conclusão de Curso foi avaliado pela seguinte banca:

João Fernando Machry Sarubbi - Orientador
Departamento de Computação
Centro Federal de Educação Tecnológica de Minas Gerais

Elizabeth Fialho Wanner - Membro da banca de avaliação
Departamento de Computação
Centro Federal de Educação Tecnológica de Minas Gerais

Paulo Eduardo Maciel de Almeida - Membro da banca de avaliação
Departamento de Computação
Centro Federal de Educação Tecnológica de Minas Gerais

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CURSO DE ENGENHARIA DE COMPUTAÇÃO
BELO HORIZONTE
DEZEMBRO DE 2014

Dedico este trabalho aos meus pais e amigos.

Agradecimentos

Agradeço ao professor Douglas cujas aulas de Inteligência Artificial me serviram de inspiração, e também ao meu professor orientador, João Sarubbi, que me ajudou bastante na escrita deste relatório. Agradeço também aos meus amigos que me ajudaram neste trabalho executando testes no jogo durante seu desenvolvimento.

"Tás a ver a linha do horizonte?
A levitar, a evitar que o céu se desmonte
Foi seguindo essa linha que notei que o mar
Na verdade é uma ponte" (Gabriel Pensador)

Resumo

O jogo de Damas tem sido alvo de estudos na área de Inteligência Artificial (IA) há décadas. Com o objetivo de facilitar tais estudos, este trabalho propõe a criação de uma arquitetura de código na linguagem Java, que codifica um jogo de Damas, incluindo a interface gráfica, implementação de regras de jogo, fluxos de jogo com interação com o usuário e uma interface de código para a criação de inteligências artificiais para o jogo. Para testar a interface de IA da arquitetura, foram implementadas algumas estratégias de IA para o jogo de Damas, baseadas nos algoritmos minimax e podas alfa-beta; e para testar o sistema foram realizados testes empíricos, com o foco no controle de fluxos pela interface gráfica, e testes unitários, com o foco na implementação das regras do jogo. O sistema também permite a comparação entre jogadores no que diz respeito ao número de vitórias, derrotas, empates; e tempo e memória gastos durante a escolha de cada jogada. Comparações entre alguns jogadores, realizadas neste trabalho, demonstraram coerência com os resultados esperados, aumentando a credibilidade e confiabilidade da arquitetura criada. Ao final, temos uma arquitetura que permite programar estratégias de IA para o jogo de damas, na linguagem Java, utilizando a interface de código disponibilizada pela arquitetura; além de permitir realizar comparações entre as estratégias criadas. Este trabalho contribuirá na realização de futuros estudos de inteligência artificial aplicada no jogo de Damas.

Palavras-chave: arquitetura de código. jogo de Damas. Java.

Abstract

The Checkers have been investigated in the area of Artificial Intelligence (AI) for decades. Aiming to facilitate such studies, this paper proposes the creation of an architecture code in the Java language, which encodes a game of checkers, including the graphical interface, implementing game rules, game flows with user interaction and a code interface for creating artificial intelligence strategies for the game. To test the architecture, some artificial intelligences have been implemented for the game of checkers, based on minimax and alpha-beta pruning algorithms; and to test the system have been conducted empirical tests, with a focus on control flows from the GUI, and unit testing, with the focus on the implementation of the rules of the game. The system also allows for comparison between players with regard to the number of wins, losses, draws; and time and memory costs for the choice of each move. Comparisons between some players, in this work, showed consistency with the expected results, increasing the credibility and reliability of the created architecture. Finally, we have an architecture that allows programming artificial intelligence strategies to the checkers, in the Java language using the code interface provided by the architecture; and also to enable comparisons between artificial intelligence strategies created. This work will help in future studies of artificial intelligence applied in Checkers.

Keywords: architecture code. Checkers. Java.

Lista de Figuras

Lista de Tabelas

Lista de Quadros

Lista de Algoritmos

Lista de Abreviaturas e Siglas

IA	Inteligência Artificial
IDE	*Integrated Development Environment*
MEF	Máquina de Estados Finitos
RNAs	Redes Neurais Artificiais

Lista de Símbolos

$\in$	Pertence
$\subseteq$	Está contido ou é igual a

Sumário

1 Introdução

O clássico jogo conhecido como Damas é um dos jogos mais antigos conhecido pela humanidade. Não há conhecimento exato das regras originais deste jogo, mas sabe-se que elas sofreram numerosas modificações com o passar do tempo. Segundo Flor (2010), em 1756 um matemático inglês escreveu um tratado sobre muitos livros escritos sobre o jogo, definindo assim as regras oficiais.

Jogos como Damas, Xadrez, Gamão, entre outros, têm sido alvos de estudos de Inteligência Artificial (IA) por muito tempo e já existem programas de computador extraordinariamente bons capazes de desafiar até os melhores jogadores mundiais. Um exemplo disso é "o Deep Blue da IBM"que "se tornou o primeiro programa de computador a derrotar o campeão mundial em uma partida de xadrez, ao vencer Garry Kasparov por um placar de 3,5 a 2,5 em uma partida de exibição"(GOODMAN; KEENE, 1997 apud RUSSELL; NORVIG, 2004).

Não muito distante, em reportagem do jornal Folha de S. Paulo, Garcia (2007) afirma que 'Em artigo na revista "Science", pesquisadores canadenses descrevem como conseguiram criar um programa de computador comprovadamente imbatível no jogo de damas'. O programa se chama Chinook e maiores informações sobre o projeto podem ser encontradas no site `http://webdocs.cs.ualberta.ca/~chinook/`. Algoritmos com tal capacidade, no entanto, requerem alto poder de processamento e/ou grande massa de dados, tornando inviável de ser aplicado em dispositivos que possuem pouca memória ou onde o baixo consumo de energia é de considerável importância, como dispositivos móveis, por exemplo. A razão disso é que o jogo possui quantidades enormes de jogadas possíveis, cerca de "500 bilhões de bilhões de possíveis movimentos"(COLATO, 2007), o que inviabiliza o cálculo de todas elas sem uma grande massa de dados e/ou poder de processamento elevado.

1.1 Motivação

A disciplina de IA possui uma carga teórica muito ampla e variada, que pode ser bem explorada em atividades práticas, como por exemplo, no desenvolvimento dos algoritmos abordados em sala de aula. No entanto, não é fácil cobrir toda a teoria com atividades práticas, talvez pelo tempo curto combinado à complexidade de se chegar aos objetivos.

A criação de um Jogo com inteligência artificial programável pode ajudar o aluno/professor a desenvolver atividades práticas para as teorias aprendidas na disciplina de IA. Consequentemente contribui como sendo um material didático que possa

despertar o interesse de estudantes pelo conhecimento em IA.

1.2 Objetivo

O objetivo principal deste projeto é codificar um jogo de damas que permita criar distintos algoritmos de IA para serem adicionados ao código, de forma simples e organizada. Espera-se desenvolver o jogo em Java, com arquitetura de código bem projetada e estruturada, assim como uma codificação de qualidade e bem documentada, para que possa no futuro ser usado como ferramenta de ensino, e que possa também ser evoluído com melhorias e colaborações de outros alunos e professores.

Todo o controle lógico do jogo, assim como a verificação das regras do jogo, interface gráfica e motor do jogo serão desenvolvidas. Desta forma, o aluno/professor precisará apenas programar estratégias de IA para o jogo e anexar o código ao programa.

Como exemplo de funcionamento e teste espera-se desenvolver três tipos de estratégias de IA, utilizando os seguintes algoritmos:

- Aleatório
- Minimax
- Podas Alfa-Beta

Espera-se também ser possível jogar no modo "algoritmo humano" no qual o controle do jogo é passado ao usuário (jogador humano), ao invés de usar uma IA programada.

Além de desenvolver o jogo e implementar diferentes IAs para ele, serão realizadas análises e comparações entre os diferentes algoritmos e heurísticas usados na implementação. Será utilizado como comparação o tempo, a quantidade de memória utilizada e o número de vitórias, derrotas e empates de cada abordagem.

Ao final teremos um jogo que permite utilizar diferentes algoritmos para sua IA e que avalia cada uma delas fazendo comparações entre si. O aluno inclusive poderá desenvolver novas estratégias de IA e anexá-las ao jogo de forma simples.

2 Referencial Teórico

2.1 Regras do jogo

O jogo de Damas possui muitas variantes ao redor do mundo. Por isso é útil apresentar as regras que foram usadas na criação deste jogo. Segundo Correia (2011) as regras oficiais são as seguintes:

1. O jogo de damas é praticado em um tabuleiro de 64 casas, claras e escuras. A grande diagonal (escura), deve ficar sempre à esquerda de cada jogador. O objetivo do jogo é imobilizar ou capturar todas as peças do adversário.
2. O jogo de damas é praticado entre dois oponentes, com 12 pedras brancas de um lado e com 12 pedras pretas de outro lado. O lance inicial cabe sempre a quem estiver com as peças brancas. Também joga-se damas em um tabuleiro de 100 casas, com 20 pedras para cada lado - Damas Internacional.
3. A pedra anda só para frente, em diagonal, uma casa de cada vez. Quando a pedra atinge a oitava linha do tabuleiro ela é promovida a dama.
4. A dama é uma peça de movimentos mais amplos. Ela anda para frente e para trás, quantas casas quiser. A dama não pode saltar uma peça da mesma cor.
5. A captura é obrigatória. **Não existe sopro**. Duas ou mais peças juntas, na mesma diagonal, não podem ser capturadas.
6. A pedra captura a dama e a dama captura a pedra. Pedra e dama têm o mesmo valor para capturarem ou serem capturadas.
7. A pedra e a dama podem capturar tanto para frente como para trás, uma ou mais peças.
8. Se no mesmo lance se apresentar mais de um modo de capturar, é obrigatório executar o lance que capture o maior número de peças (Lei da Maioria).
9. A pedra que durante o lance de captura de várias peças, apenas passe por qualquer casa de coroação, sem aí parar, não será promovida à dama.
10. Na execução do lance de captura, é permitido passar mais de uma vez pela mesma casa vazia, não é permitido capturar duas vezes a mesma peça.

11. Na execução do lance de captura, não é permitido capturar a mesma peça mais de uma vez e as peças capturadas não podem ser retiradas do tabuleiro antes de completar o lance de captura.

12. Após 20 lances sucessivos de damas, sem captura ou deslocamento de pedra, a partida é declarada empatada.

13. São declarados empatados, após 5 lances, finais de:

    - 2 damas contra 2 damas;
    - 2 damas contra uma;
    - 2 damas contra uma dama e uma pedra;
    - uma dama contra uma dama;
    - uma dama contra uma dama e uma pedra.

2.2 Técnicas de IA no universo dos jogos

O objetivo de aplicar IA ao desenvolvimento de um jogo é fazer com que o computador consiga jogar o jogo de forma inteligente, e que consiga, inclusive, desafiar jogadores humanos. Segundo Kurzweil (1990 apud RUSSELL; NORVIG, 2004), IA é: "A arte de criar máquinas que executam funções que exigem inteligência quando executadas por pessoas.".

A afirmação acima pode parecer exagero ou até mesmo ficção científica, mas é justamente isso que se espera da IA, ou pelo menos chegar o mais perto possível disso. Longe de ser uma tarefa fácil - devido à grande complexidade que é o modo humano de raciocinar - a IA teve muitos avanços nos últimos tempos; mas apesar de todo o avanço o cérebro humano ainda continua sendo o "dispositivo" de processamento de informações mais flexível e mais eficiente que se conhece.

Muitas técnicas foram desenvolvidas para se chegar a resultados satisfatórios de inteligência, cada qual é melhor ajustável ou adaptável a determinado tipo de problema. Olhando para o mundo dos jogos, não é novidade que os jogos utilizam técnicas de inteligência artificial em seu código, como por exemplo, **Máquinas de Estado Finito** (CASSANDRAS, 1993 apud TATAI, 2006), **Lógica Fuzzy** (PEDRYCZ; GOMIDE, 1998 apud TATAI, 2006), **Árvore de Decisão** (RUSSELL; NORVIG, 2004) e **Redes Neurais** (BRAGA et al., 2007). Por serem técnicas muito usadas, citaremo-las brevemente nos próximos tópicos.

2.2.1 Máquinas de Estado Finito

Uma máquina de estados finitos (MEF) é um modelo matemático que representa um conjunto de estados e as transições entre eles. Esta é uma técnica baseada em regras, de fácil gerenciamento e que possui baixa complexidade, e talvez seja por essas características que esta é uma das técnicas mais usadas na implementação de jogos. Formalmente, é definida como sendo a seguinte quintupla:

$$(E, X, f, x_0, F),$$

na qual:

- E é um conjunto finito de eventos;
- X é um conjunto finito de estados;
- f é uma função de transição de estado, $f : XxE \rightarrow X$;
- x_0 é um estado inicial, $x_0 \in X$;
- F é um conjunto de estados finais, $F \subseteq E$.

A ideia principal consiste em definir um conjunto de estados que o jogo pode assumir e definir também, para cada estado, uma transição para outro, de acordo com as condições e comportamentos do jogo. Um exemplo bem simples de uma MEF está representado pelo Quadro 1.

Ocasionalmente os jogos poderão ter um número muito maior de estados. O Algoritmo 1 representa uma possível codificação para o exemplo do Quadro 1.

Os seguintes jogos utilizam esta técnica e podem ser citados como exemplo:

- Age of Empires;
- Doom;
- Half Life;
- Quake.

2.2.2 Lógica Fuzzy

Sistemas fuzzy são sistemas que em sua constituição lidam formalmente com conceitos vagos e/ou imprecisos na tomada de decisão, de forma similar a como seres

Algoritmo 1: Algoritmo exemplo para tomadas de ações, baseado em MEF

```
Input: estado → estado atual da nave
switch estado do
    case NAVE_VOANDO
        if existeInimigoAVista() then
            aproximarDoInimigo()
            estado ← APROXIMANDO_DO_INIMIGO
        end
        break
    end
    case APROXIMANDO_DO_INIMIGO
        if !existeInimigoAVista() then
            estado ← NAVE_VOANDO
            break
        end
        aproximarMaisDoInimigo()
        if inimigoNoAlcanceDasArmas() then
            atacar()
            estado ← ATACANDO
        end
        break
    end
    case ATACANDO
        if !existeInimigoAVista() then
            pararAtaque()
            estado ← NAVE_VOANDO
            break
        end
        if inimigoMorto() then
            pararAtaque()
            estado ← NAVE_VOANDO
            break
        end
        if inimigoMuitoForte() then
            pararAtaque()
            estado ← FUGINDO
        end
        break
    end
    otherwise
        // Outros estados entrariam aqui ...
    end
endsw
```

Quadro 1 – Exemplo de uma MEF para um jogo fictício de nave no espaço.

Evento x Estado	**Nave voando, procurando inimigos**	**Nave se aproximando do inimigo**	**Nave atacando**	**Outros estados...**
Nenhum inimigo a vista	Continue voando	Siga voando	Pare de atacar e siga voando	...
Inimigo a vista	Se aproxime do inimigo	Aproximar mais do inimigo	...	...
Inimigo no alcance das armas	...	Atacar o inimigo	...	...
Inimigo vivo	...	...	Continue atacando	...
Inimigo morto	...	...	Pare de atacar e siga voando	...
Inimigo é muito forte	...	...	Pare de atacar e fuja!	...
Outras condições ...	...	...	...	...

humanos são capazes de tomar decisões levando em consideração somente conhecimentos vagos (TATAI, 2006). Em certos tipos de problemas, onde é difícil de definir um modelo matemático que descreva o problema, esta técnica pode se adaptar muito bem.

Com esta técnica é possível estabelecer e calcular um "nível de felicidade" que um personagem tem, o quanto um personagem pensa sobre "a água estar quente" ou "esta pessoa é jovem". Note que não há um limite numérico exato para definir uma barreira de transição entre estes conceitos.

Os seguintes jogos utilizam esta técnica e podem ser citados como exemplo:

- Swat 2;
- Call to Power;
- Close Combat;
- The Sims.

2.2.3 Árvore de decisão

Uma árvore de decisão toma como entrada um conjunto ou situação descritos por um conjunto de atributos e retorna uma "decisão" - o valor de saída previsto, de acordo com a entrada (RUSSELL; NORVIG, 2004).

O uso desta técnica está intimamente relacionado à teoria de jogos (BARRICHELO, 2014). Consiste em visualizar adiante as possíveis ações que poderão ser

tomadas pelos jogadores adversários ou as possibilidades de estados em que o jogo poderá transitar. Conhecendo essas informações é possível decidir com segurança qual a melhor escolha a ser tomada. Na maioria das vezes é inviável computar todos os possíveis caminhos e por isso as árvores de decisão frequentemente terão um limite de profundidade estabelecido.

Em outra abordagem as árvores de decisão também podem ser construídas dinamicamente de acordo com a experiência do agente (ou com treinamento). Isso significa que o agente pode aprender com suas próprias jogadas ou observando outros jogadores. Essas experiências são armazenadas no decorrer do jogo e organizadas em uma árvore.

Sua estrutura é fácil de ser construída e entendida. E o aprendizado a partir desta técnica se desenvolve de maneira eficiente.

O seguinte jogo utiliza esta técnica e pode ser citado como exemplo:

- Black & White.

2.2.4 Redes Neurais Artificiais

Redes Neurais Artificiais (RNAs) são sistemas paralelos distribuídos compostos por unidades de processamento simples que calculam determinadas funções matemáticas. Tais unidades são dispostas em uma ou mais camadas e interligadas por um grande número de conexões, geralmente unidirecionais. Na maioria dos modelos essas conexões estão associadas a pesos, os quais armazenam o conhecimento adquirido pelo modelo e servem para ponderar a entrada recebida por cada neurônio da rede (BRAGA et al., 2007).

Esta técnica é inspirada em sistemas biológicos e possui uma grande quantidade de aplicações. Há décadas, a informática tem usado RNAs para resolver muitos problemas do mundo real que envolvem tarefas como classificação, estimativa e controle.

Em jogos ela geralmente é aplicada durante o desenvolvimento do jogo, no qual é feito o ajuste dos parâmetros por treinamentos off-line. Também é possível fazer o ajuste dos parâmetros em tempo real, porém essa técnica não é muito utilizada em jogos, pois pode gerar comportamentos não aceitáveis. Jogos que utilizam esta técnica têm como objetivo fazer com que o agente aprenda a imitar o comportamento de outros agentes.

Esta técnica é difícil de ser desenvolvida e manuseada, principalmente por causa da grande quantidade de ajustes exigido pelos parâmetros.

Os seguintes jogos utilizam esta técnica e podem ser citados como exemplo:

- BC3K;
- Creatures;
- Heavy Gear.

2.3 Trabalhos Relacionados

Dentre as técnicas apresentadas anteriormente, decidiu-se usar a técnica de Árvores de decisão para a realização deste trabalho. Foi considerada esta técnica, a princípio, como a que mais se encaixa à natureza do jogo.

Analisando trabalhos anteriores de outros autores notou-se que esta técnica é muito usada e bem aceita para a criação de IA para o jogo de damas. Alguns desses trabalhos serão destacados nos tópicos seguintes. A maioria destes trabalhos demonstraram não implementar todas as regras dos jogos. De fato, a implementação das regras é custosa e de considerável complexidade. Também se torna dificil a verificação/testes dessas regras, devido a sua complexidade. Este trabalho irá contribuir para que futuros autores possam criar seus trabalhos a partir de uma arquitetura que já implementa todas as regras e o controle de fluxos do jogo de damas.

2.3.1 Jogo das Damas clássicas

Leite (1999), em um trabalho realizado em seu mestrado, criou o que chamou de **Jogo das Damas clássicas**. No desenvolvimento do programa ele usou o algoritmo **Minimax** com **podas Alfa-Beta** (RUSSELL; NORVIG, 2004) na árvore de procura (a-b Search), no entanto não desenvolveu nenhuma heurística para reordenação de jogadas de um determinado nível, que segundo ele "poder-se-ia desta forma cortar mais ramos da árvore e aumentar a velocidade de resposta". Ele fez uso deste algoritmo fixando um nível máximo de profundidade da árvore de procura, e desta forma definiu três níveis de dificuldade: nível 1 - profundidade 5; nível 2 - profundidade 6; e nível 3 - profundidade 7.

A **função de avaliação** que implementou é calculada segundo a fórmula $\frac{b-p}{b+p}$ e baseia-se em dois scores: score das brancas b e score das pretas p. Cada score é calculado através de caracteres defensivos (casas privilegiadas de defesa) e caracteres materiais (1 ponto para peão; 3 pontos para dama). Ele usou o conceito de *Quiescence* para evitar fazer avaliação estática do tabuleiro em posições muito sensíveis, isto é, que possibilitem uma grande variação da função de avaliação. Neste sentido, sempre que uma posição não fosse estável, o algoritmo descia mais um nível na árvore (para além da profundidade máxima) e assim sucessivamente.

Seu programa **não faz detecção de empates**, pois segundo autor, "viria complicar demasiado o Minimax (a função deveria ter globalmente lista de jogadas repetidas) tornando-o lento".

2.3.2 Jogo de Damas utilizando Realidade Aumentada

Luz et al. (2010) criaram um jogo de damas com realidade aumentada. O jogo é jogado por um jogador humano com o auxilio de um tabuleiro real (com apenas as peças do jogador humano) e um celular com câmera que desenha na tela do dispositivo as peças virtuais do jogador virtual, enquanto filma o tabuleiro real. Neste jogo eles utilizaram os algoritmos **Minimax** e poda **Alfa-Beta**.

O foco do trabalho parecia estar totalmente concentrado na realidade aumentada, deixando de lado algumas regras do jogo. Por exemplo, **não foi implementado o conceito de dama das regras originais**, ou seja, uma peça ao chegar ao outro lado do tabuleiro não tinha movimentos especiais de dama como sendo válidos (mover para trás quantas casas quiser). Também **não há verificação de empates**. Seu jogo termina quando não houver mais peças por parte de qualquer jogador, ou quando não houver mais movimentos válidos a serem feitos.

Além de desenvolver o jogo, eles também realizaram uma análise dos tempos de execução de cada etapa do processamento da imagem, no que diz repeito à realidade aumentada, ocultando os tempos de processamento para os algoritmos de inteligência artificial utilizados. A heurística utilizada para avaliação dos estados do tabuleiro no algoritmo Minimax e Alfa-Beta não é revelada.

2.3.3 D-VisionDraughts: Uma rede neural jogadora de damas

Em 2011, Barcelos (2011) propôs em sua dissertação de mestrado um sistema de aprendizagem de damas, o D-VisionDraughts: um agente distribuído jogador de damas baseado em **redes neurais** que aprende por reforço. Seu sistema corresponde a uma versão distribuída do eficiente jogador VisionDraughts, que utiliza como método de avaliação de um estado do tabuleiro uma rede neural MLP (Rede **perceptron com multicamadas**), que aprende pelo método das diferenças temporais. Em seu trabalho ele substituiu o algoritmo serial utilizado para a busca em árvore de jogos do Vision-Draughts, o **Minimax com podas alfa-beta**, pelo algoritmo distribuído Young Brothers Wait Concept.

Seu trabalho mostrou que as técnicas aplicadas no D-VisionDraughts reduziram expressivamente o tempo necessário para a etapa de busca.

2.4 Algoritmos para IA

Citaremos a seguir, com uma breve explicação, os algoritmos consultados para a criação das estratégias de IA neste trabalho. São eles: **Minimax** e **Poda Alfa-Beta**.

2.4.1 Algoritmo Minimax

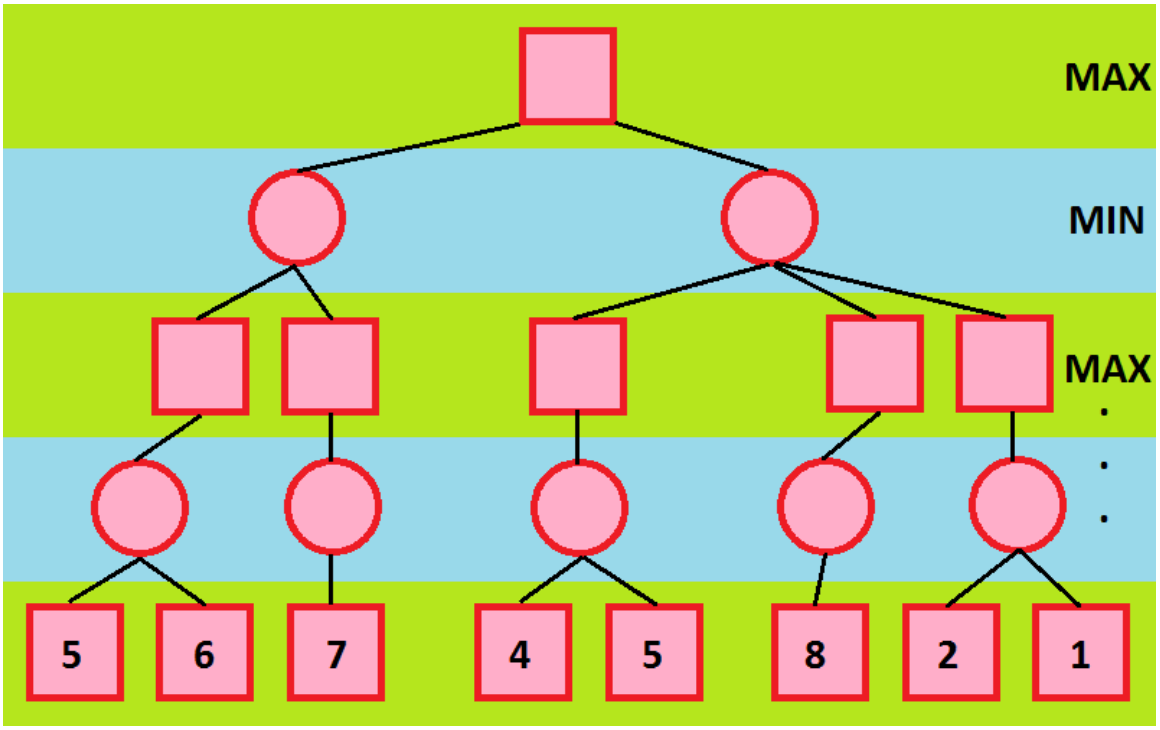
Vamos imaginar um jogo de damas em andamento, no qual é a vez do jogador A jogar. Neste momento o jogo se encontra em um **estado** E, e o jogador A tem algumas possibilidades de jogadas, como por exemplo, mover-se com diferentes peças do tabuleiro. Chamaremos essas possibilidades de jogadas de **transições**. Tais transições levam a outros estados do jogo, chamados **sucessores**. Dentre todas as transições possíveis devemos ser capazes de escolher a melhor entre elas que levará o jogador A à vitória, e consequentemente levará seu adversário B à derrota. O estado no qual o jogo acaba chama-se **estado final**.

A partir do estado em que o jogo se encontra é possível criar uma árvore que mapeia todas as transições para os estados seguintes, e depois todas as transições seguintes para novos estados, e assim sucessivamente, até que cheguemos a todos os estados finais possíveis a partir do estado inicial da busca. Esta árvore se chama **árvore de jogo**.

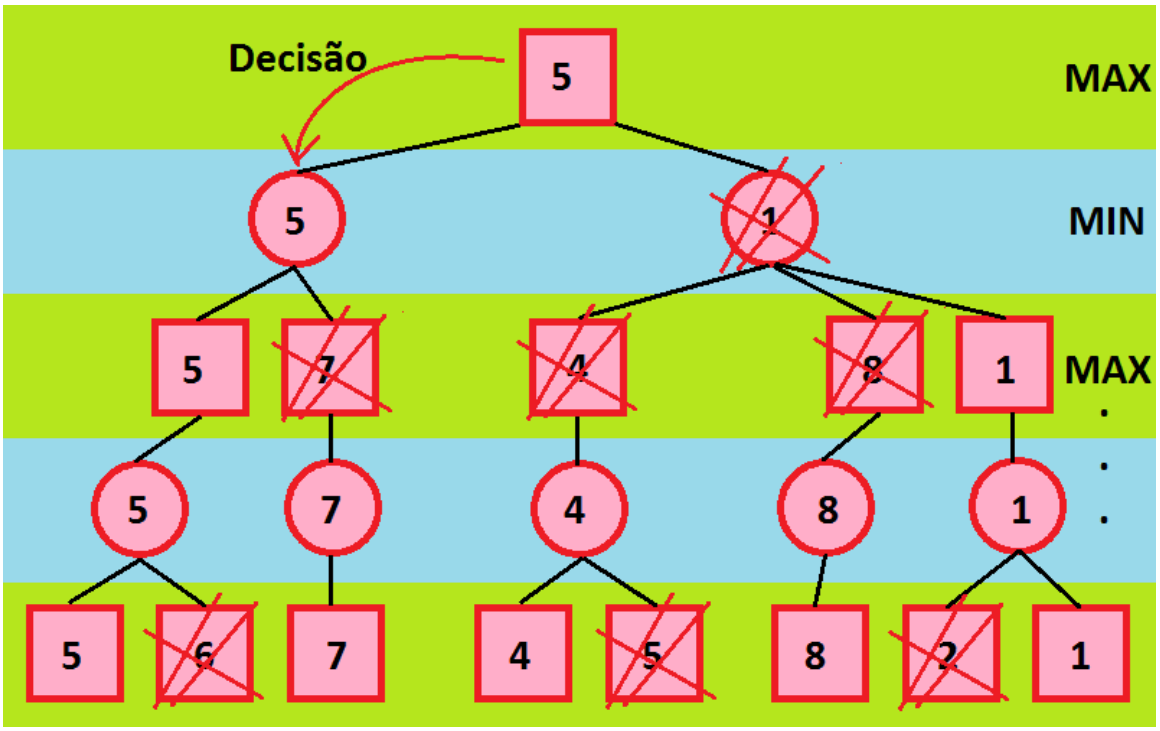
Analisando a árvore de jogo é possível verificar a melhor transição a ser tomada pelo jogador no estado de jogo atual. Para isso devemos verificar qual a melhor escolha a ser feita, a partir dos estados finais, e propagar pela árvore as melhores escolhas até que se chegue ao estado inicial da busca. Quando sabemos quais as melhores escolhas a serem feitas, temos então a nossa **solução ótima**. A Figura 1 ilustra uma árvore de um jogo fictício e a decisão de escolha da transições que o jogador deve escolher para ganhar o jogo. Este é um bom exemplo do uso da técnica de árvores de decisão.

Nem sempre é viável computar a árvore de jogo completa (devido a limitações de tempo), e, portanto, não conseguimos saber qual é a melhor transição a ser escolhida pelo jogador em determinado estado de jogo. Por isso devemos determinar um nível de profundidade da árvore que limita a busca de novas transições. Ao alcançar os estados deste limite de profundidade, não é possível determinar qual a melhor escolha a ser feita pelo jogador se estes estados não forem estados finais (dado que não seja possível dizer qual a melhor transição a ser escolhida a partir de um estado E, sem conhecer os estados sucessores de E). Neste caso, a estratégia a ser tomada é tentar avaliar, a partir de uma **heurística**, o quão favorável é o estado. Este valor é então propagado pela árvore até se chegar ao estado inicial, onde o jogador escolherá a transição que o leve ao estado de maior valor possível.

Vale ressaltar que nem sempre a transição escolhida levará o jogador ao estado



(a) Árvore de jogo antes de ser processada, com os valores estabelecidos para seus estados finais.



(b) Decisão tomada pelo jogador Max após execução do algoritmo minimax.

Figura 1 – Execução do algoritmo minimax.

com maior valor, pois devemos considerar que o adversário também faz escolhas que, pela natureza do jogo, são desfavoráveis ao outro jogador. Se pegarmos a mesma Figura 1 como exemplo, podemos ver que o caminho escolhido por Max dá acesso ao estado final de valor 7. No entanto, após a escolha de Max, Min escolhe o caminho que

impossibilita alcançar o estado final de valor 7. Imagine agora que Max queira alcançar o valor 8, pois é o maior valor possível entre os estados finais. Então, ingenuamente, ele escolhe o caminho oposto. No entanto, Min também toma decisões, e, no fim, o estado final alcançado é o de valor 1.

O algoritmo minimax é usado para descobrir a melhor decisão a ser tomada a partir de um estado de uma árvore de jogo. Este algoritmo realiza sua busca seguindo os mesmos princípios explicados anteriormente. Considerando um estado atual da busca, o que deve ser feito é:

- Calcular o quão bom são os estados "finais"(estados mais profundos possíveis de serem alcançados);
- Propagar os valores dos estados finais até o estado inicial, considerando as escolhas feitas por cada jogador;
- Escolher a transição que leva para o estado sucessor com o maior valor propagado.

Consideremos dois jogadores, que chamaremos de Max e Min (por razões que a seguir se tornarão óbvias). O algoritmo minimax utiliza uma função para avaliar o quão bom é determinado estado para o jogador Max. Desta forma o jogador Max fará escolhas de transições que o levam para estados sucessores com os **maiores valores**, e o jogador Min fará escolhas que o levam para estados sucessores com os **menores** valores (pois se o estado possuir um valor baixo, este não será uma boa escolha para o jogador Max, e consequentemente será uma boa escolha para o jogador Min, devido à natureza do jogo).

O bom funcionamento do algoritmo minimax está intimamente relacionado à qualidade da função de avaliação e ao limite de profundidade da busca. No Algoritmo 2 podemos verificar o pseudocódigo do minimax. A função SUCESSORES($estado$) retorna todos os nodos sucessores de um estado, a função UTILIDADE($estado$) calcula, a partir de uma heurística, qual o valor que representa a avaliação de utilidade daquele nodo, a favor do jogador Max, e a função TESTE_TERMINAL($estado$) indica se o estado passado por parâmetro é um estado terminal.

2.4.2 Poda Alfa-Beta

O algoritmo poda alfa-beta é uma otimização do algoritmo minimax. O que ele faz é evitar a verificação de alguns nós da árvore de jogo, quando sabe, de forma comprovada, que estes nós não precisam ser verificados. Este algoritmo não altera o resultado da busca e sua grande vantagem é que ao evitar a avalização de alguns nós, o tempo de busca e a quantidade de memória são reduzidas. Em outras palavras, este algoritmo é **mais rápido e menos custoso**.

Algoritmo 2: Algoritmo minimax

```
função DECISÃO_MINIMAX(estado)
Entrada: estado, estado corrente no jogo
Saída: uma ação
inicio
    v ← VALOR_MAX(estado);
    return a ação em SUCESSORES(estado) com valor v
fin
```

```
função VALOR_MAX(estado)
Entrada: estado, estado corrente no jogo
Saída: um valor de utilidade
inicio
    if TESTE_TERMINAL(estado) then
        return UTILIDADE(estado);
    v ← −∞;

    para cada s em SUCESSORES(estado) faça
            v ← MAX(v, VALOR_MIN(s));

    return v
fin
```

```
função VALOR_MIN(estado)
Entrada: estado, estado corrente no jogo
Saída: um valor de utilidade
inicio
    if TESTE_TERMINAL(estado) then
        return UTILIDADE(estado);
    v ← +∞;

    para cada s em SUCESSORES(estado) faça
            v ← MIN(v, VALOR_MAX(s));

    return v
fin
```

Fonte: Russell e Norvig (2004)

O princípio matemático presente no funcionamento do algoritmo alfa-beta é demonstrado pelo seguinte exemplo:

$$\begin{gathered} minimax = max\{min\{20, 8, 16\}, min\{4, x, y\}\} \\ minimax = max\{8, min\{4, x, y\}\} \\ minimax = max\{8, z\} \qquad com\, z \leq 4 \\ minimax = 8 \end{gathered}$$

Neste exemplo é possível notar que, independente dos valores de x e y, o resultado será sempre 8. Em analogia a uma árvore de jogo, se x e y representassem nodos da árvore e já soubéssemos os valores de seus nodos vizinhos, conforme representados pelo exemplo acima, não seria necessário calcular os valores dos nodos x e y para sabermos o resultado final da busca. Veja este mesmo exemplo, ilustrado na Figura 2.

O algoritmo de podas alfa-beta está escrito, em pseudocódigo, no Algoritmo 3. Neste trabalho utilizaremos uma versão um pouco diferente desta apresentada, na qual trocaremos a linha *if v>=b then* por *if v>b then* e a linha *if v<=a then* por *if v<a then*. Esta modificação sutil faz com que o algoritmo possa encontrar ramificações diferentes na árvore que levam a estados de jogos diferentes, mas com o mesmo valor de avaliação. É importante que estas ramificações estejam presentes para que possamos escolher aleatoriamente entre as soluções encontradas nestes casos específicos, para que os jogadores não se comportem de forma totalmente determinística.

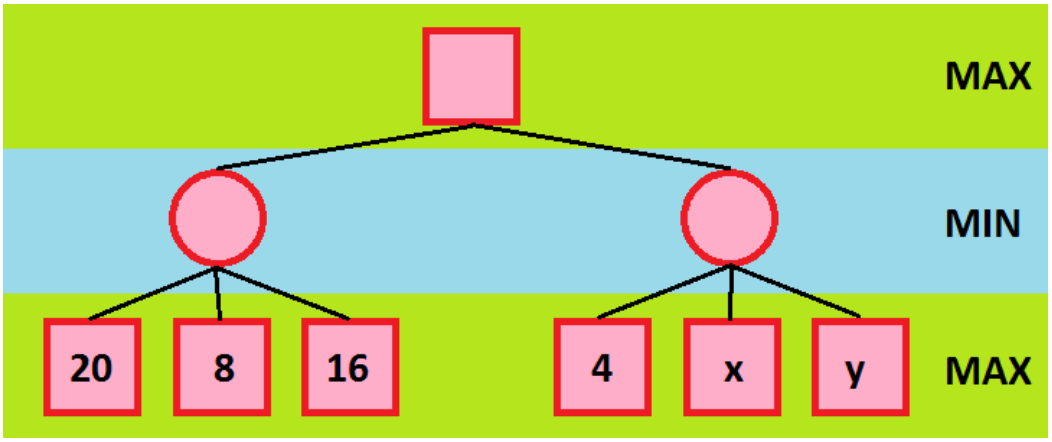
Algoritmo 3: Algoritmo de podas alfa-beta

```
função BUSCA_ALFA_BETA(estado)
Entrada: estado, estado corrente no jogo
Saída: uma ação
inicio
    v ← VALOR_MAX(estado, −∞, +∞);
    return a ação em SUCESSORES(estado) com valor v
fin
```

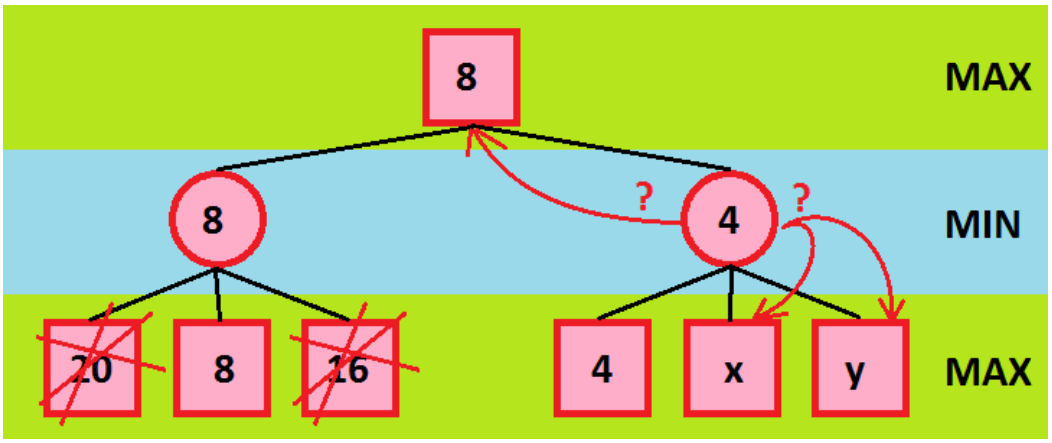
```
função VALOR_MAX(estado, α, β)
Entrada: estado, estado corrente no jogo
α, o valor da melhor alternativa para MAX ao longo do caminho até estado
β, o valor da melhor alternativa para MIN ao longo do caminho até estado
Saída: um valor de utilidade
inicio
    if TESTE_TERMINAL(estado) then
        return UTILIDADE(estado);
    v ← −∞;

    para cada s em SUCESSORES(estado) faça
        v ← MAX(v, VALOR_MIN(s, α, β));
        if v ≥ β then
            return v;
        α ← MAX(α, v);

    return v
fin
```

```
função VALOR_MIN(estado, α, β)
Entrada: estado, estado corrente no jogo
α, o valor da melhor alternativa para MAX ao longo do caminho até estado
β, o valor da melhor alternativa para MIN ao longo do caminho até estado
Saída: um valor de utilidade
inicio
    if TESTE_TERMINAL(estado) then
        return UTILIDADE(estado);
    v ← +∞;

    para cada s em SUCESSORES(estado) faça
        v ← MIN(v, VALOR_MAX(s, α, β));
        if v ≤ α then
            return v;
        β ← MIN(β, v);

    return v
fin
```

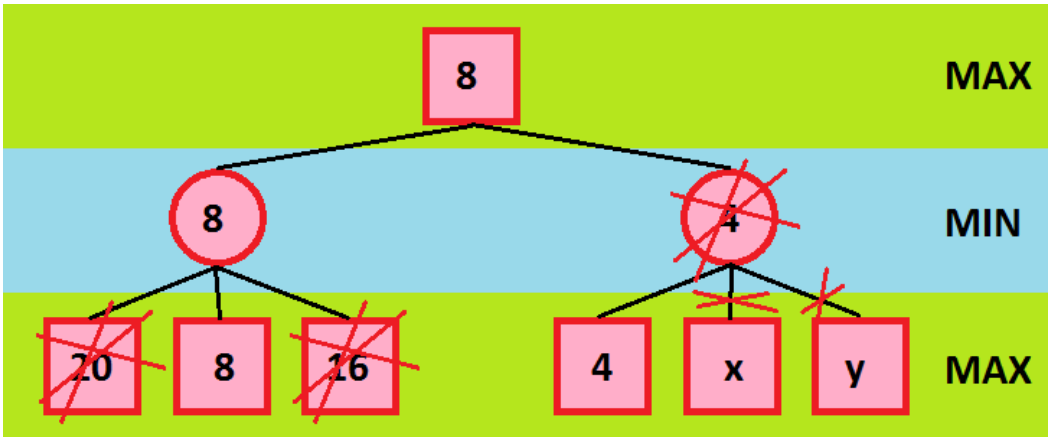
Fonte: Russell e Norvig (2004)



(a) Árvore de jogo antes de ser processada, com os valores estabelecidos para seus estados finais.



(b) Nodo Min avalia o primeiro filho. Ao perceber que nenhum filho fará o valor do resultado alterar o valor do seu nodo pai, os próximos filhos não precisam mais ser avaliados.



(c) Decisão tomada pelo jogador Max após execução do algoritmo alfa-beta.

Figura 2 – Exemplo de podas alfa-beta.

3 Metodologia

Desenvolver um jogo, por mais simples que ele pareça, sempre envolve grandes desafios para o programador. Com a finalidade de transmitir uma boa explicação, cada etapa deste trabalho será separada em tópicos.

Antes de começar o desenvolvimento do jogo foram consideradas as linguagens de programação Java e C++. Pela necessidade em utilizar um sistema multi thread e pela maior facilidade de se desenvolver interfaces gráficas em Java, esta foi a linguagem escolhida (Java, versão 7.1).

Para codificar o jogo de damas foi utilizado o software Eclipse Java EE IDE for Web Developers, que pode ser adquirido gratuitamente através do link http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/lunar. Esta IDE (*Integrated Development Environment*) foi escolhida por ser uma IDE bem reconhecida pela comunidade de desenvolvedores Java, por ser software livre e pelo conhecimento e experiência com a mesma.

O computador utilizado durante todo o projeto foi um ultrabook Acer Aspire S7 com as seguintes características:

- Processador Intel Core i7-3517U CPU 1.9GHz 2.40GHz;
- 4 GB de memória (RAM);
- Sistema operacional de 64 bits, Windows 8.1, processador x64.

3.1 Delineamento da pesquisa

A arquitetura do projeto se divide nos seguintes módulos:

- Interface Gráfica;
- Controller;
- Game Logic (Lógica de jogo);
- Regras;
- IA.

Na Figura 3 pode ser observada a arquitetura simplificada que organiza estes módulos.

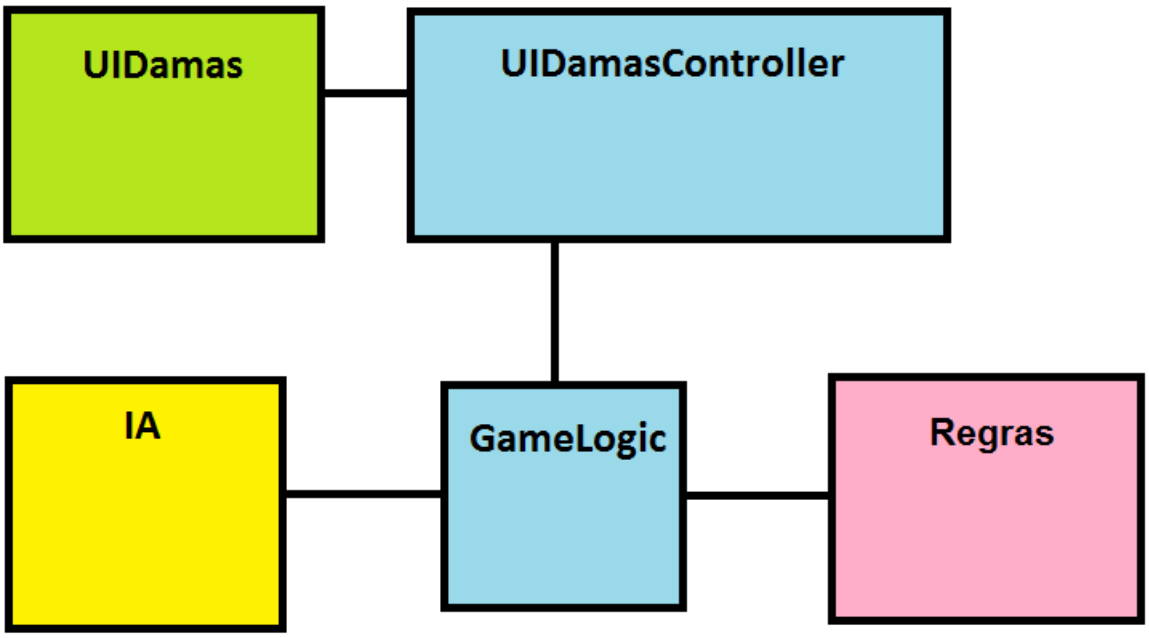


Figura 3 – Arquitetura simplificada entre módulos do sistema

O módulo Game Logic é o núcleo do sistema e serve como intermediário entre os outros módulos, se comunicando diretamente com todos eles (exceto com o módulo Interface Gráfica).

Inicialmente o usuário faz as configurações de cada jogador para depois iniciar o jogo. Ao iniciar o jogo, o módulo *Game Logic* é acionado, passando a controlar todos os fluxos do jogo na aplicação. O módulo *Controlador* é o intermediário entre a comunicação de *Game Logic* com o módulo *Interface Gráfica*. *Game Logic* consulta o módulo *Regras* para gerenciar as regras do jogo de forma correta, enviar as jogadas possíveis aos jogadores (Humano ou IA) e conferir as jogadas feitas por eles.

Começaremos com uma explicação aprofundada do módulo de interface gráfica, e, posteriormente, explicaremos os outros módulos da aplicação.

4 Desenvolvimento do Jogo

4.1 Interface Gráfica

Utilizamos a linguagem de programação Java junto com biblioteca "javax.swing" para implementação da interface gráfica. Não optamos por uma biblioteca gráfica robusta como OpenGL pois o foco deste trabalho não se resume a uma interface bonita e rica de detalhes, e sim ao desenvolvimento da arquitetura do jogo. Uma interface mais rica de detalhes demandaria um tempo maior do que o planejado para este trabalho. Além disso, a biblioteca javax.swing já atende perfeitamente a este projeto, dado que um jogo de damas pode ser implementado com gráficos 2D simples.

A partir da interface gráfica é possível configurar quais serão os jogadores (Humano ou Máquina) e quais serão os parâmetros de configuração das IAs para jogadores do tipo Máquina, se for o caso. As configurações são sempre ajustadas antes de o jogo começar e são preservadas até que o jogo termine. Desta forma, ao iniciar o jogo o painel de configurações fica desabilitado até que o jogo acabe. Para iniciar o jogo é preciso clicar no botão *Iniciar Jogo*.

Quando o jogo é iniciado as pedras são desenhadas no tabuleiro. Pedras brancas são desenhadas na parte de baixo e as pretas na parte de cima. O jogador 1, dono das pedras brancas, começa a partida e todas as peças possíveis de serem movimentadas são destacadas no tabuleiro, conforme a Figura 4.

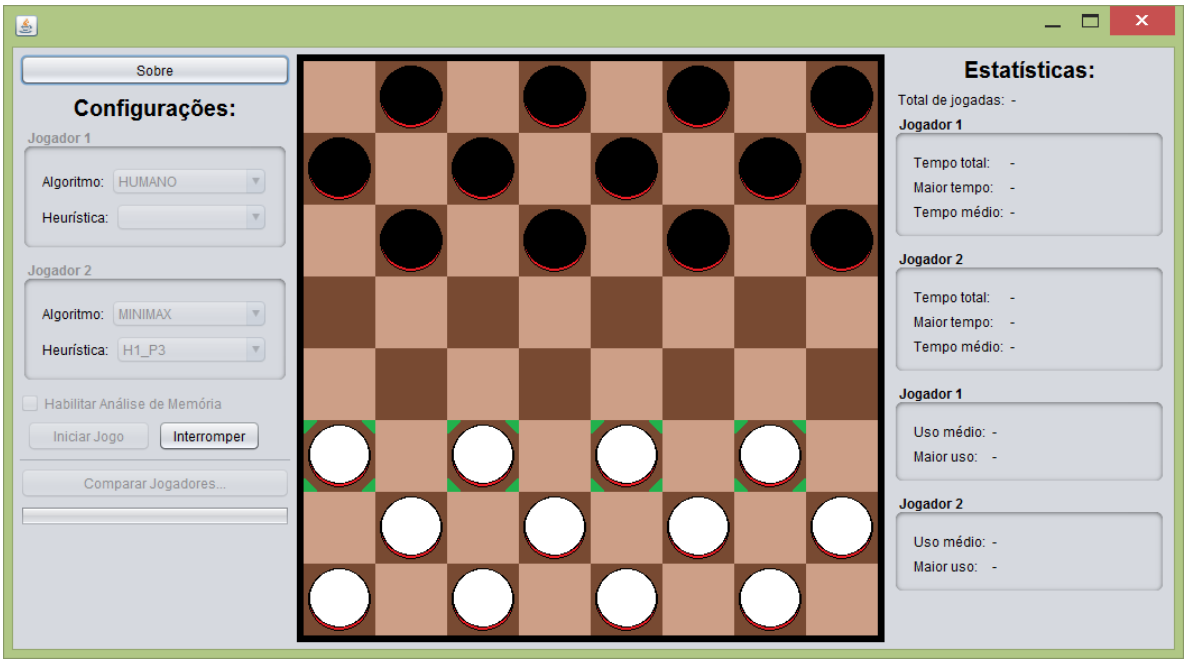


Figura 4 – Interface de usuário logo após o jogo ser iniciado

Se o jogador (humano) tentar efetuar uma jogada errada o sistema exibe uma mensagem informando que a jogada não é permitida, e o jogador então deve escolher uma jogada possível. As pedras possíveis de serem movimentadas são destacadas no tabuleiro.

Para facilitar o desenvolvimento das interfaces, ela foi dividida em vários agrupamentos de componentes. O agrupamento consiste em uma extensão do componente javax.swing.JPanel, na qual adicionamos componentes e/ou outros grupos (que também são componentes), e implementamos algumas funcionalidades específicas do grupo. A primeira divisão da interface foi em três agrupamentos principais de componentes: PainelMenu, PainelCasasTabuleiro e PanelEstatisticas. No grupo PainelMenu existe ainda subgrupos, que são: PainelJogador(1 e 2) e PainelAcoes. O grupo PainelCasasTabuleiro contém diversas instâncias de LabelCasa, que representam cada posição do tabuleiro. O grupo PanelEstatisticas contém dois subgrupos de PanelTempo e dois subgrupos de PanelMemoria, que representam estatísticas de tempo e memória para cada jogador. A Figura 5 ilustra a localização de todos os componentes na tela e a Figura 6 a hierarquia entre eles.

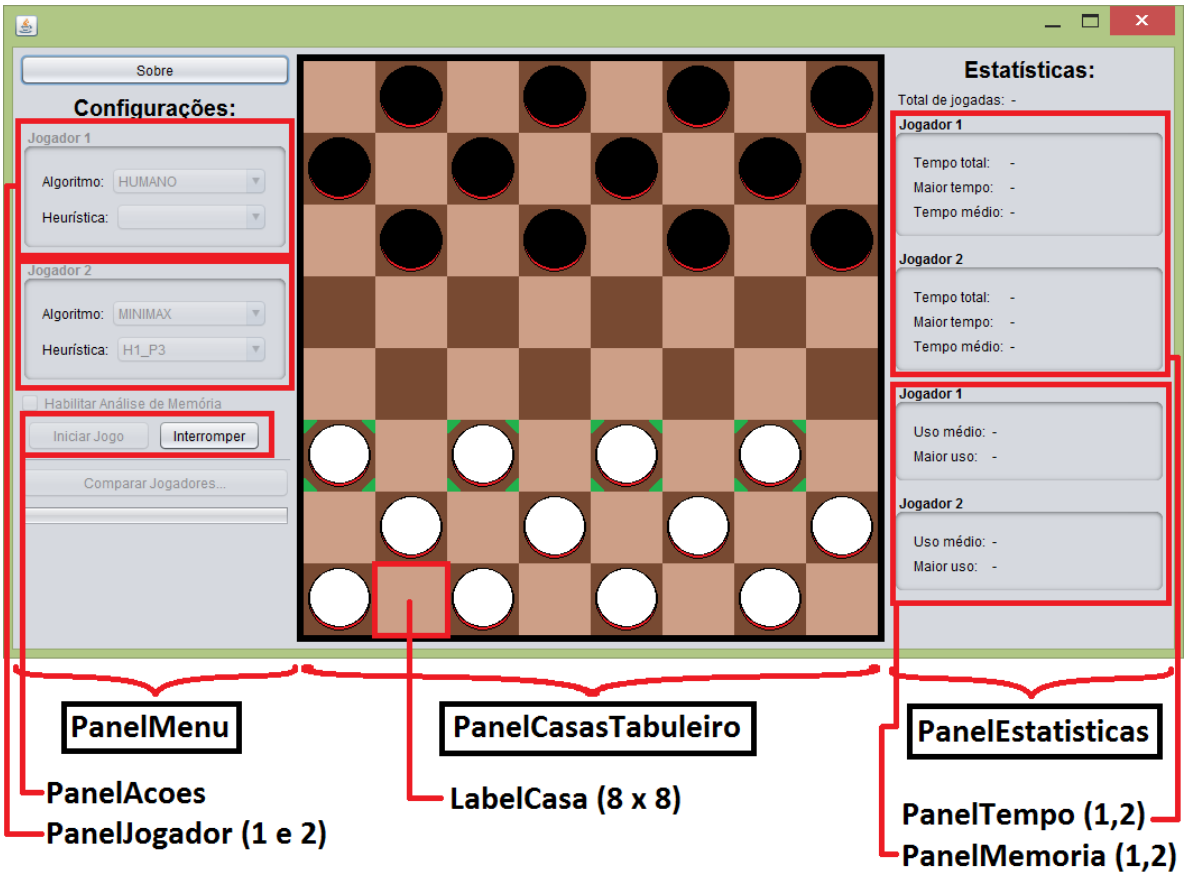


Figura 5 – Localização dos componentes

No subgrupo PainelJogador são definidos os componentes necessários para se fazer as configurações do jogador. Neste grupo existem dois conjuntos de listas de sele-

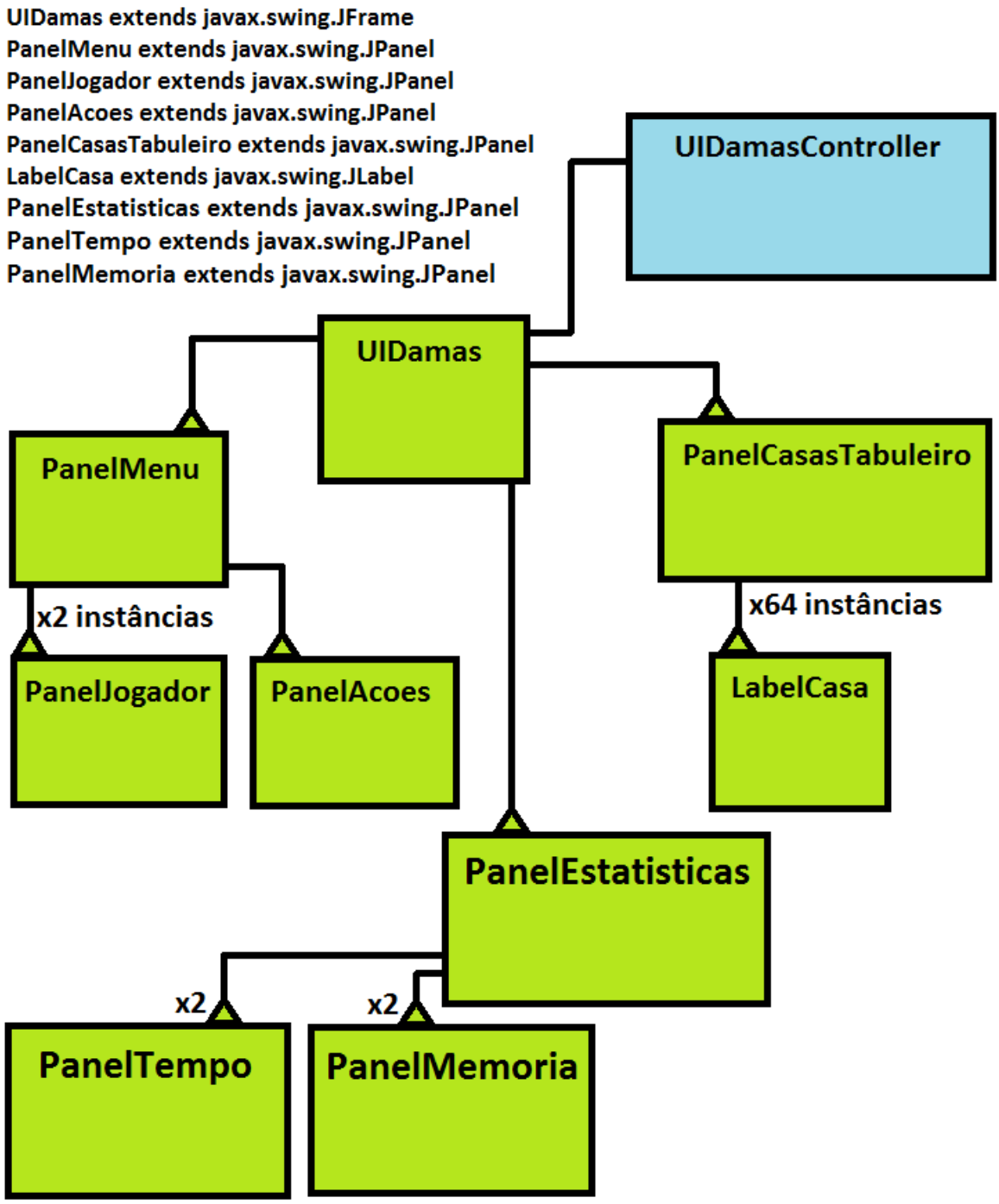


Figura 6 – Hierarquia entre componentes da interface gráfica

ção que gerenciam o Algoritmo e a Heurística. Dependendo do algoritmo selecionado, a lista de seleção de Heurísticas poderá ficar desabilitada. Todo esse controle é gerenciado e encapsulado na classe PanelJogador. Duas instâncias deste componente são utilizadas na tela: uma para configurar o jogador 1 e outra para configurar o jogador 2.

No subgrupo PainelAcoes são definidos dois botões. O botão *Iniciar Jogo* faz com que o jogo configure as pedras para suas posições iniciais e o fluxo de jogo é então iniciado. O botão *Interromper* é habilitado quando o botão *Iniciar Jogo* ou *Comparar*

Jogadores é acionado. Este botão interrompe o processamento do controlador e finaliza o jogo (ou o processamento dos jogos) imediatamente.

O botão *Comparar Jogadores* foi definido no próprio painel PanelMenu, junto com a barra de progresso de processamento de jogos. Ao acionar o botão *Comparar Jogadores* o programa pede ao usuário que informe um valor numérico que indica quantos jogos se deseja simular entre os jogadores configurados.

A classe PainelMenu reúne as duas instâncias do componente PainelJogador, o PainelAcoes e outros componentes mais simples, incluindo:

- Botão nomeado *Sobre*, que mostra ao usuário informações sobre a aplicação, ao ser acionado;
- Label para o texto *Configurações* exibido na tela;
- Botão de opção denominado *Habilitar Análise de Memória*, para que o usuário possa habilitar ou não esta opção;
- Botão *Comparar Jogadores*, para acionar o controlador que realiza comparação entre jogadores controlados por IAs;
- Barra de progresso para acompanhar o processamento dos jogos durante as comparações.

No *PainelCasasTabuleiro* são criadas 64 instâncias do componente *LabelCasa*, representando assim, cada casa do tabuleiro. O componente *LabelCasa* é uma extensão do componente *javax.swing.JLabel*, que adiciona os atributos *linha* e *coluna*, necessários para identificar qual a posição da casa no tabuleiro. A classe *PainelCasasTabuleiro* controla cada instância de *LabelCasa*. Isso inclui o controle gráfico do que deverá ser desenhado em cada casa.

O segredo por trás dos desenhos das casas é que existe uma imagem para cada estado possível de uma casa. Dependendo do estado em que esta casa se encontra, deve ser escolhida a imagem apropriada para representá-la na tela. Desta forma, se uma casa contém uma pedra branca do tipo dama, a classe *PainelCasasTabuleiro* deverá escolher a imagem correta que representa este estado e desenhá-lo no *LabelCasa*. Além deste controle gráfico esta classe gerencia os eventos de *mouseClicked* de cada *LabelCasa* para que os eventos de click com o mouse possam ser capturados e interpretados pela aplicação.

Todas as imagens usadas na representação dos estados possíveis das casas foram desenvolvidas neste projeto utilizando a software Microsoft Paint. Essas imagens podem ser visualizadas na Figura 7.

(a) Casa branca (b) Casa preta (c) Casa preta destacada (d) Pedra branca

(e) Pedra branca destacada (f) Pedra preta (g) Pedra preta destacada (h) Dama branca

(i) Dama branca destacada (j) Dama preta (k) Dama preta destacada

Figura 7 – Estados das casas do tabuleiro

No painel *PanelEstatisticas* são definidos dois subgrupos *PanelTempo* e dois subgrupos *PanelMemoria*, um para cada jogador. O subgrupo *PanelTempo* contém diversos labels que são atualizados ao final do jogo, e que informam algumas estatísticas de tempo das jogadas de determinado jogador, incluindo:

- Tempo total acumulado de cada jogada do jogador;
- Tempo de execução da jogada mais demorada do jogador;
- Média entre os tempos de execução de todas as jogadas do jogador.

O subgrupo PanelMemoria, de forma semelhante ao subgrupo PanelTempo, também contém diversos labels que são atualizados ao final do jogo. Estes, porém, informam algumas estatísticas de memória das jogadas de determinado jogador, incluindo:

- Uso médio de memória, pelo jogador, durante todo o jogo;
- Quantidade de memória usada pelo jogador durante a jogada que utilizou a maior quantidade de memória.

A classe principal do módulo de interface gráfica é a *UIDamas*. Esta classe define uma janela (*UIDamas* é uma extensão do componente *javax.swing.JFrame*) e organiza os três grupos de componentes encapsulados: *PainelMenu*, *PainelCasasTabuleiro* e *PanelEstatisticas*. Esta classe captura as configurações e os eventos gerados por esses componentes para tratá-los de forma apropriada. Os eventos capturados pelo grupo *PainelAcoes* e por cada *LabelCasa* são enviados ao modulo de controle da aplicação. O módulo de controle também pode enviar eventos ao módulo de interface gráfica solicitando atualizações gráficas das casas do tabuleiro; informando o início e o fim da partida; e solicitando que se habilite/desabilite o tabuleiro para controlar quando o jogador humano pode jogar. A classe *UIDamas* é a responsável por receber estes eventos e tratá-los. Quando é preciso fazer uma atualização nos componentes da tela, comandos de *UIDamas* são enviados aos componentes até chegar ao componente responsável por fazer a atualização. Veja o fluxo dos principais eventos na Figura 8.

Com essa visão geral da interface gráfica encerramos este tópico. No tópico seguinte será mostrada uma visão geral dos fluxos do programa, juntamente com uma explicação aprofundada das camadas Controlador e Game Logic.

4.2 Sistema de controle de fluxos de jogo

O módulo *Game Logic* é responsável por fazer todo o controle de fluxos do jogo, decidindo de quem é a vez de jogar e verificando se as regras do jogo estão sendo obedecidas. Para que essas verificações sejam possíveis, o módulo se comunica diretamente com o módulo Regras.

Ao iniciar a aplicação o sistema de controle fica em modo de espera aguardando o início do jogo. Nesta etapa podem ser feitas configurações dos jogadores na tela, através da interface gráfica. Ao acionar o botão "Iniciar Jogo", o *Controlador* aciona o controle de fluxos de jogo, controlado pelo módulo *Game Logic*. Neste momento o *Controlador* avisa a interface para se preparar para o início do jogo. A interface gráfica toma as ações necessárias, dentre as quais estão:

- Desabilitar os painéis de configurações dos jogadores;
- Atualizar os botões de ações;
- Desabilitar o tabuleiro, para impedir que eventos de click sejam gerados sobre as casas do tabuleiro, até que o controle de fluxo decida que o jogador já possa jogar.

Após as ações de início de jogo, a interface gráfica fica parcialmente bloqueada e o módulo *Game Logic* passa a controlar todos os fluxos de jogo. O código dentro do módulo *Game Logic* consiste em um laço de repetição que executa atualizações no jogo

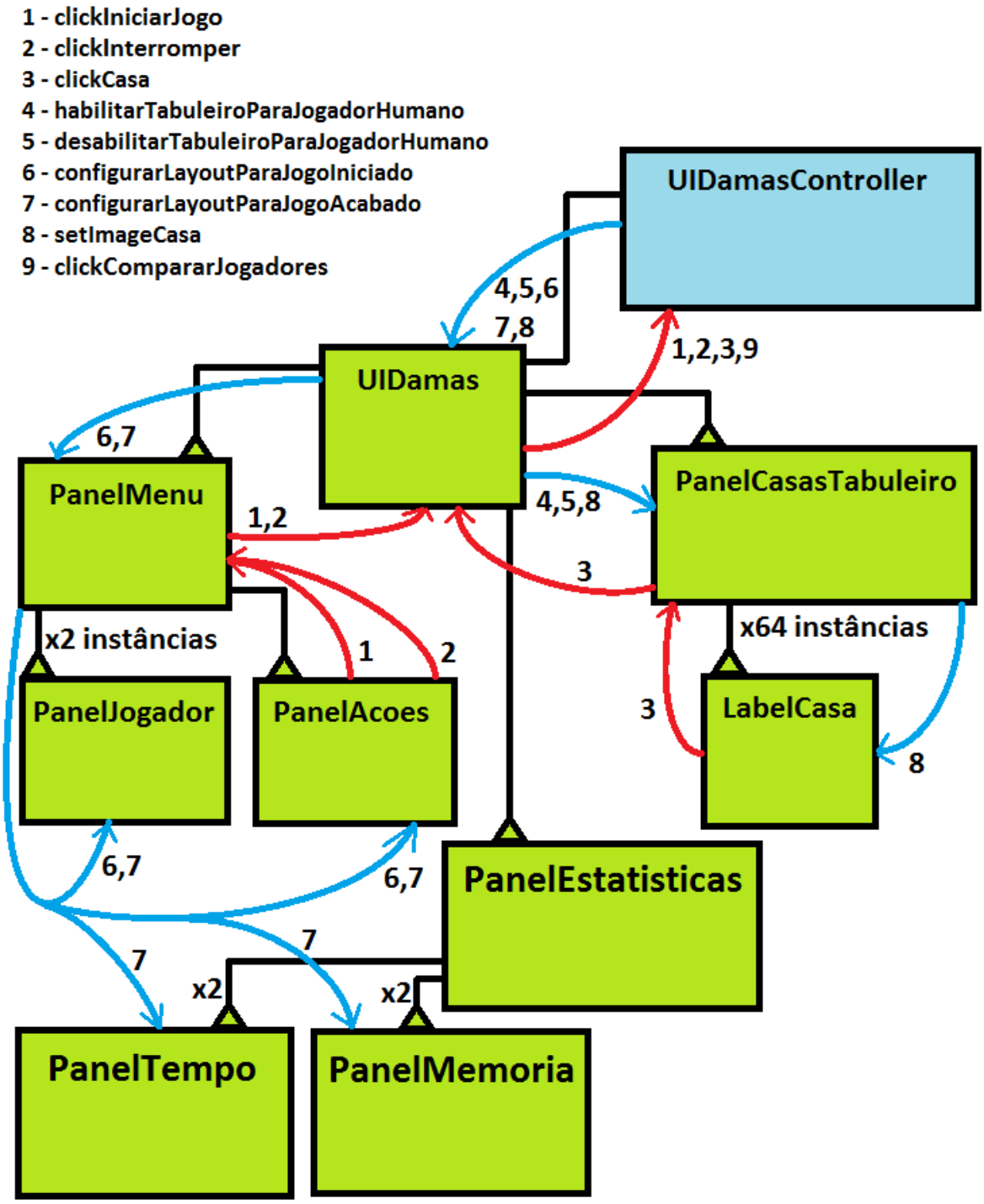


Figura 8 – Propagação dos principais eventos que chegam e saem do módulo de Interface Gráfica

até que o jogo acabe. Quando o jogo acaba, *Game Logic* retorna o controle de fluxos para a *Interface Gráfica* para que as configurações de um novo jogo possam ser feitas. *Game Logic* é acionado em uma *thread* independente para evitar que a *Interface Gráfica* fique totalmente bloqueada. Se a *Interface Gráfica* ficasse totalmente bloqueada não seria possível que o usuário fechasse a aplicação clicando no botão de fechar da janela, e nem

seria possível acessar outros comandos, como acionar o botão "Sobre", por exemplo. Por isso o bloqueio da *Interface Gráfica* é feito manualmente, habilitando e desabilitando alguns componentes da *Interface Gráfica* (*PanelJogador*(1 e 2), *PanelCasasTabuleiro*, botões dentro do componente *PanelAcoes*).

O laço de controle em *Game Logic* se resume na chama de dois métodos (*atualizarJogo* e *desenharJogo*) dentro de um laço de repetição, como pode ser visto no Algoritmo 4.

Algoritmo 4: Laço de controle principal, em GameLogic

```
try {
    do {
        atualizarJogo();
        desenharJogo();
    } while (!isJogoAcabado());
} catch {
    showGameLogicError();
    resultadoJogo = null;
}
this.listener.onGameLogic_JogoAcabado(resultadoJogo);
jogoIniciado = false;
```

O método atualizar jogo é dividido na chamada dos seguintes métodos:

- *void atualizarRegras()*;
- *void escolherJogada()*;
- *void verificarJogo()*;
- *void proximoJogador()*.

4.2.1 Método *atualizarRegras*

O método *atualizarRegras* é responsável por solicitar ao módulo Regras que se atualize, passando para ele um objeto que define o estado atual do jogo. Com essa informação, o módulo Regras consegue encontrar todas as jogadas possíveis de serem efetuados pelo jogador atual. As informações dessas jogadas possíveis de serem efetuadas pelos jogadores são usadas para informar aos jogadores que jogadas podem ser realizadas e também para verificar, posteriormente, se as jogadas retornadas pelos jogadores são realmente uma das jogadas válidas. Os detalhes de implementação que definem o estado atual do jogo, as jogadas e outras estruturas são explicados no tópico Implementação de Estruturas do Jogo.

4.2.2 Método *escolherJogada*

O método *escolherJogada* é responsável por efetuar a jogada do jogador atual, e por verificar se existem movimentos válidos que possam ser executados por ele. Caso não existam movimentos válidos as próximas etapas no método atualizarJogo não são executadas e a vitória do jogo é dada ao jogador adversário. Isso respeita a regra número 1 deste jogo:

- O jogo de damas é praticado em um tabuleiro de 64 casas, claras e escuras. A grande diagonal (escura) deve ficar sempre à esquerda de cada jogador. O objetivo do jogo é imobilizar ou capturar todas as peças do adversário.

Apesar de existir um módulo específico para gerenciar as regras do jogo, esta e outras regras foram implementadas no módulo Game Logic, por serem mais convenientes de serem assim implementadas, devido à arquitetura da própria codificação. Deixamos no módulo Regras a implementação das regras associadas aos movimentos de pedras do jogo, assim como as regras de capturas e de promoções de damas.

Quando é a vez de algum jogador jogar é verificado se ele é do tipo Humano ou se é um jogador controlado por alguma IA.

Caso seja humano, o módulo *Controlador* é acionado para que faça todo o controle de fluxos necessários para interagir com o usuário e atualizar os gráficos no módulo *Interface Gráfica*. Quando o jogador humano finaliza sua jogada, este controle é finalizado e o módulo *Controlador* retorna a jogada efetuada pelo jogador para o método *escolherJogada*. Para que o módulo *Controlador* consiga fazer a interação com o usuário e retornar a jogada para o método *escolherJogada*, no mesmo fluxo de execução, é feito o bloqueio do fluxo dentro do módulo *Controlador*, através de um lock. Logo antes de bloquear o fluxo, os eventos de click no tabuleiro são habilitados. Desta forma, o fluxo de *Game Logic* fica temporariamente travado, e o usuário fica possibilitado de realizar sua jogada pela interface gráfica. Quando a interação com o usuário acaba, e a jogada do mesmo já está finalizada e definida, o tabuleiro é desabilitado e o fluxo é liberado. Com isso a jogada é retornada de volta ao método *escolherJogada*.

Caso o jogador seja controlado por alguma IA, o módulo *Inteligência Artificial* é acionado para que execute os algoritmos implementados pela IA do jogador. Após a execução, o módulo de Inteligência Artificial retorna a jogada escolhida para o método escolherJogada.

Após a jogada de cada jogador, o módulo *Controlador* é acionado para que se comunique com a *Interface Gráfica* e seja feita a atualização gráfica das casas do tabuleiro. Qualquer outro tipo de comunicação que impacte em atualizações gráficas (como por exemplo o fim do jogo) é feita através do módulo *Controlador*.

Um resumo abstrato dos principais fluxos no método escolherJogada são ilustrados na Figura 9.

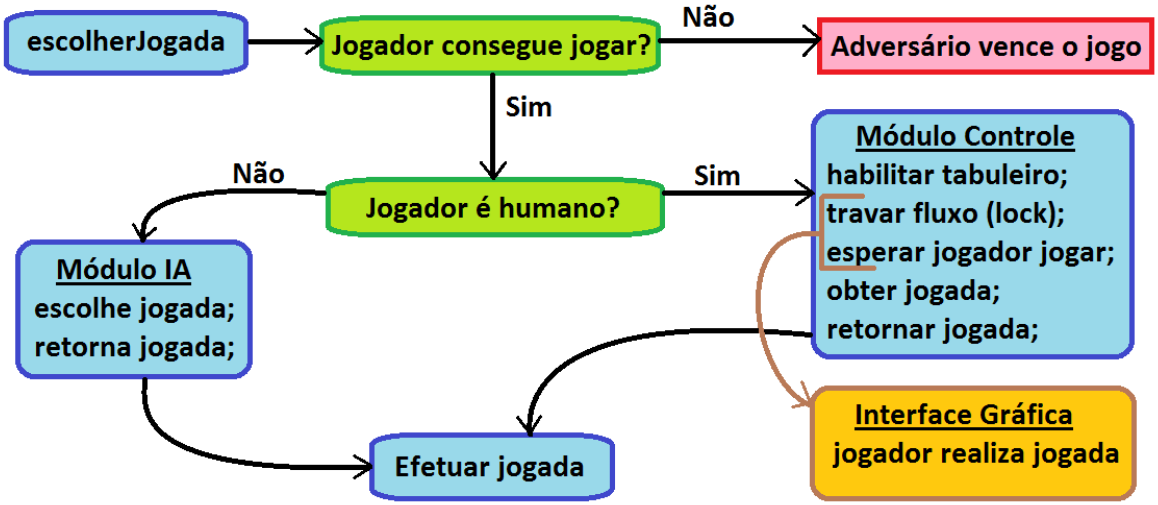


Figura 9 – Principais fluxos no método escolherJogada

4.2.3 Método *verificarJogo*

Este método é responsável por detectar finais e empates no jogo. Uma das possibilidades de finalizações de jogo já é detectada no método *escolherJogada*, que é a vitória do adversário quando o jogador não possui mais movimentos válidos. Quando for este o caso, nenhuma verificação é feita neste método, pois já sabemos o resultado do jogo.

As verificações feitas neste método são feitas na seguinte ordem:

- Primeiro é verificada a quantidade de peças dos jogadores. Se o jogador atual fez uma captura em sua jogada pode ser que tenha capturado a última peça do adversário. Neste caso a vitória do jogo é dada a ele e o fluxo no método é finalizado;

- A segunda verificação é feita para detectar empates no jogo relacionados à seguinte regra: "Após 20 lances sucessivos de damas de cada jogador, sem captura ou deslocamento de pedra, a partida é declarada empatada". Para fazer esta verificação foi criado uma variável que conta o número de jogadas consecutivas de damas. Toda vez que a jogada é realizada com uma pedra do tipo dama, esta variável é incrementada. Caso a pedra da jogada não seja do tipo dama, o contador é zerado. Quando o contador chegar a este método com o valor igual a 40 (20 jogadas para cada jogador) a partida é declarada empatada e o fluxo deste método é finalizado;

- A terceira verificação é feita para detectar o restante dos empates possíveis de acordo com a regra: "Finais de 2 damas contra 2 damas; 2 damas contra uma; 2 damas contra uma dama e uma pedra; uma dama contra uma dama e uma dama contra uma dama e uma pedra, são declarados empatados após 5 lances de cada jogador."Para fazer esta verificação foram mantidas duas variáveis: "provavelEmpateFinal"e "numLancesFinais". A primeira variável identifica qual dos tipos de finais, verificados nesta regra, o estado atual do jogo se encontra. São eles: VENCE_JOGADOR1, VENCE_JOGADOR2, EMPATE_MOVIMENTO_DAMAS, EMPATE_2D_2D, EMPATE_2D_1D, EMPATE_2D_1D_1N, EMPATE_1D_1D, EMPATE_1D_1D_1N. Caso o estado atual do jogo não se enquadre em nenhum destes estados a variável recebe um valor null. A segunda variável conta quantas vezes consecutivas o jogo permaneceu em estados que apontassem para o mesmo tipo de final. Toda vez que o tipo de final apontado pelo estado muda ou é estabelecido como *null*, este contador é zerado. No momento que este contador chega ao valor 10 (5 jogadas de cada jogador), o jogo é declarado empatado.

4.2.4 Método *proximoJogador*

O método *proximoJogador* é responsável por fazer a troca de jogadores. Para isso, o objeto que define o estado do jogo é atualizado, informando para ele o novo jogador atual (o jogador atual é o jogador que possui a vez de jogar).

4.3 Implementação das regras

Como foi visto nos tópicos anteriores algumas regras do jogo são implementadas no módulo *Game Logic*, por serem mais convenientes de serem implementadas desta forma. A regra número 2, por outro lado, está controlada por uma classe chamada *Global* que define o tamanho do tabuleiro de forma estática, que pode ser consultado por toda a aplicação.

O controle do tamanho do tabuleiro foi feito para funcionar de forma genérica. Assim sendo, o tabuleiro é criado de forma dinâmica, tendo como base o tamanho estabelecido na classe *Global*. As regras de movimento de pedras no jogo também foram feitas para funcionar independente do tamanho do tabuleiro. De fato as regras não dependem do tamanho do tabuleiro, e sim da posição das pedras no tabuleiro. O tamanho do tabuleiro pode limitar o movimento das pedras, dado que as pedras só podem se mover dentro do tabuleiro, mas as regras são as mesmas para o tabuleiro no formato 8x8 ou 10x10. A diferença mais impactante entre os dois modos de jogo é a quantidade de pedras iniciais.

Apesar da classe *Global* definir o tamanho do tabuleiro como sendo 8x8, é fácil

modificar o modo de jogo para 10x10. Basta mudar os valores na classe *Global* para configurar um tabuleiro no formato 10x10. Os valores da classe *Global* que devem ser modificados são:

- *private static final int NUM_LINHAS = 10*;
- *private static final int NUM_COLUNAS = 10*.

O restante das regras, que são as regras de número 3 a 11, é implementado neste módulo. Essas regras definem os movimentos das pedras no tabuleiro, e a promoção de pedras comuns a damas.

A interface de uso definida por este módulo consiste em apenas três funções:

- *void atualizarRegras(EstadoJogo)*;
- *List<Jogada> getJogadasPermitidas()*;
- *boolean isJogadaPermitida(Jogada)*.

A função *atualizarRegras* é a mais complexa de todas elas. Esta função recebe como parâmetro o estado atual do jogo e, a partir das informações que este parâmetro proporciona, são calculadas todas as jogadas possíveis de serem realizadas pelo jogador, segundo as regras estabelecidas. O método se torna complicado devido à variedade de regras. Por exemplo, suponha uma situação em que o jogador possa mover uma pedra e que também seja possível ele capturar uma pedra inimiga. Segundo as regras a captura é obrigatória. Neste caso a primeira jogada não pode ser realizada e portanto não entra na lista de jogadas válidas. Se o jogador, por sua vez, puder capturar pedras por vários caminhos diferentes, somente as jogadas com a maior quantidade de capturas são contabilizadas como jogadas válidas. Movimentos de damas também são complicados de tratar, pois suas regras fogem um pouco do padrão. Damas podem capturar pedras executando longos saltos, mas nunca saltando por mais de uma pedra de uma só vez. Todas essas regras são tratadas dentro deste método. Para tornar a codificação mais fácil e legível, esta função foi dividida em métodos menores, mais simples, que verificam cada tipo de movimento válido no jogo. Após descobrir todas as jogadas válidas, é criada uma lista contendo todas essas jogadas e a lista é mantida no módulo *Regras* até que seja solicitada outra atualização na lista.

O metodo *getJogadasPermitidas* retorna a lista de jogadas válidas, calculada no método *atualizarRegras*. Essa lista de jogadas permitidas é repassada para o módulo IA e para o módulo de Interface Gráfica. Essa lista guia o jogador, que deverá escolher somente uma jogada dentre as jogadas permitidas pelas regras.

O método *isJogadaPermitida* é usado para conferir se uma jogada escolhida por um jogador é válida. A lista de jogadas válidas é passada para o módulo de IA, que deve escolher uma dentre as jogadas possíveis. Mas existe a possibilidade de que o modulo retorne uma jogada que não esteja nesta lista. Portanto as jogadas retornadas pelos jogadores devem ser sempre verificadas. Na implementação deste jogo não é permitido realizar uma jogada inválida, portanto, quando um jogador retorna uma jogada que não seja válida, a vitória é dada a seu adversário.

4.4 Implementação de Estruturas do Jogo

Os métodos e fluxos explicados até agora propagam informações através de um conjunto de estruturas de dados que serão abordadas em seguida. A primeira estrutura se chama *PosicaoTabuleiro*, que é formada basicamente por dois atributos: linha e coluna, conforme Algoritmo 5. Esta estrutura representa uma posição no tabuleiro. Esta é a estrutura mais essencial da aplicação.

Algoritmo 5: Estrutura PosicaoTabuleiro

```
public class PosicaoTabuleiro {
    private final int linha;
    private final int coluna;
    .
    .
    .
}
```

A estrutura *MovimentoSimples* representa um movimento simples no tabuleiro. O *MovimentoSimples* pode ser um movimento com ou sem captura. A estrutura *Jogada* representa uma jogada no tabuleiro, que é constituída de no mínimo um *MovimentoSimples*. O *MovimentoSimples* é representado por três atributos: *posicaoOrigem*, *posicaoCaptura* e *posicaoDestino*. Estes 3 atributos são do tipo *PosicaoTabuleiro*, explicado anteriormente. Se o movimento não tiver captura, o atributo *posicaoCaptura* é configurado para *null*. A estrutura *Jogada* é representada por duas listas. A primeira é uma lista de *MovimentoSimple*(s), na ordem que devem ser executadas para a jogada. A outra lista contém todas as posições (*PosicaoTabuleiro*) que possuem pedras adversárias que serão capturadas pela jogada, ordenados sequencialmente na ordem de captura. Os algoritmos Algoritmo 6 e Algoritmo 7 representam essas duas estruturas.

Para representar o tabuleiro foi criada a estrutura *Tabuleiro* que contém três atributos: *numLinhas*, *numColunas* e uma matriz *pedrasPosicoes*. Os atributos *numLinhas* e *numColunas* representam as dimensões do tabuleiro, informando o número de linhas e

Algoritmo 6: Estrutura MovimentoSimples

```
public class MovimentoSimples {
    private final PosicaoTabuleiro posicaoOrigem;
    private final PosicaoTabuleiro posicaoCaptura;
    private final PosicaoTabuleiro posicaoDestino;
    .
    .
    .
}
```

Algoritmo 7: Estrutura Jogada

```
public class Jogada {
    private final List<MovimentoSimples> movimentos =
        new ArrayList<MovimentoSimples>();
    private final List<PosicaoTabuleiro> pecasCapturadas =
        new ArrayList<PosicaoTabuleiro>();
    .
    .
    .
}
```

colunas que o tabuleiro possui, respectivamente. A matriz *pedrasPosicoes* representa o estado de cada posição do tabuleiro. Para esta matriz, usamos o valor *null* para representar posições vazias. Caso a posição esteja ocupada, a pedra e o jogador dono da pedra são identificados pela enumeração *IndicadorPedraJogador*. Enumerações na linguagem Java são muito poderosas. Elas definem constantes que podem conter atributos, como objetos. Esse potencial foi explorado nessa e em outras enumerações. A enumeração IndicadorPedraJogador é formada pelas seguintes constantes:

- NORMAL_J1;
- DAMA_J1;
- NORMAL_J2;
- DAMA_J2.

Cada constante possui dois atributos: jogador e tipo. O atributo jogador representa o jogador dono da pedra e o atributo tipo identifica o tipo da pedra. Cada pedra possui uma constante que representa seu tipo, definidas na enumeração *IndicadorTipoPedra*, que possui as seguinte constantes:

- NORMAL;
- DAMA.

O atributo jogador, por sua vez, é representado por uma das constantes definidas na enumeração *IndicadorJogador*, que são:

- JOGADOR1;
- JOGADOR2.

Assim como IndicadorPedraJogador, a enumeração IndicadorJogador também possui constantes como atributos, que são: **algoritmo** e **heuristica**. O atributo **algoritmo** representa o algoritmo usado pelo jogador. Este algoritmo é identificado pela enumeração IndicadorAlgoritmo. Cada constante representa um tipo de algoritmo que pode ser escolhido para o jogador. O algoritmo HUMANO indica que o jogador é humano e neste caso suas jogadas serão efetuadas através da interface gráfica. Outros tipos de algoritmos são usados para distinguir diferentes processamentos no módulo IA. Para este projeto foi definido as constantes ALEATORIO, MINIMAX e ALPHA_BETA. Cada uma dessas três constantes ativam um tipo diferente de algoritmo usado no módulo IA no momento em que se executa a jogada do jogador controlado por inteligência artificial. Para adicionar novos tipos de algoritmos, é necessário adicionar nesta enumeração uma nova constante. A interface gráfica foi construída de forma genérica, ao passo que todo o controle de configuração de algoritmo é gerado dinamicamente a partir das constantes aqui definidas. A enumeração IndicadorJogador também possui os métodos *setAlgoritmo* e *setHeuristica* que são usados para estabelecer a cada jogador o algoritmo e a heurística pertencentes a cada um deles, configurados antes do início do jogo.

O atributo **heuristica** se comporta de forma semelhante ao atributo algoritmo. Este atributo é representado por uma das constantes definidas na enumeração IndicadorHeuristica. Essas constantes funcionam como parâmetros para o algoritmo escolhido. Quando o jogador não é do tipo HUMANO, o módulo IA é acionado para que se execute algum algoritmo de inteligência artificial implementado pelo programa. O atributo algoritmo indica qual algoritmo usar, e o atributo heuristica funciona como uma parametrização para o algoritmo escolhido. Para os exemplos implementados neste projeto, cada heuristica define uma profundidade para a árvore de jogo utilizada nos algoritmos Minimax e Alfa-Beta. Caso o algoritmo seja do tipo ALEATORIO, este parâmetro é simplesmente desconsiderado. O Algoritmo 8 ilustra essas estruturas criadas para representar o Tabuleiro.

Para finalizar, o estado do jogo é representado pela estrutura EstadoJogo. Esta estrutura contém apenas dois atributos: tabuleiro e jogadorAtual. O atributo tabuleiro

Algoritmo 8: Estruturas usadas para representar o tabuleiro do jogo

```
public class Tabuleiro {
    private final int numLinhas;
    private final int numColunas;
    private final IndicadorPedraJogador[][] pedrasPosicoes;
    ...
}

public enum IndicadorPedraJogador {
    NORMAL_J1(IndicadorJogador.JOGADOR1, IndicadorTipoPedra.NORMAL),
    DAMA_J1(IndicadorJogador.JOGADOR1, IndicadorTipoPedra.DAMA),
    NORMAL_J2(IndicadorJogador.JOGADOR2, IndicadorTipoPedra.NORMAL),
    DAMA_J2(IndicadorJogador.JOGADOR2, IndicadorTipoPedra.DAMA);

    private final IndicadorJogador jogador;
    private final IndicadorTipoPedra tipo;

    private IndicadorPedraJogador(IndicadorJogador jogador,
        IndicadorTipoPedra tipo) {
        this.jogador = jogador;
        this.tipo = tipo;
    }
    ...
}

public enum IndicadorTipoPedra {
    NORMAL, DAMA
}

public enum IndicadorJogador {
    JOGADOR1, JOGADOR2;
    private IndicadorAlgoritmo algoritmo;
    private IndicadorHeuristica heuristica;
    ...
}

public enum IndicadorAlgoritmo {
    HUMANO, ALEATORIO, MINIMAX, ALPHA_BETA
}

public enum IndicadorHeuristica {
    H1_P3, H1_P4, H1_P5, H1_P6
}
```

representa o estado atual do tabuleiro, e o atributo jogadorAtual representa o jogador que possui a vez de jogar. Esta estrutura é utilizada no módulo GameLogic, no qual é mantido e modificado com a dinâmica do jogo.

4.5 Módulo IA

É no módulo IA que se implementa toda a lógica de inteligência artificial dos jogadores. Este módulo se comunica com o modulo GameLogic através de uma interface simples e eficiente. Durante o fluxo de jogo, o modulo GameLogic solicitará aos jogares que façam suas jogadas. Neste momento o módulo GameLogic invocará a seguinte função no módulo IA:

Jogada escolherJogada(List<Jogada> jogadasPermitidas, EstadoJogo ej);

Esta função recebe como parâmetro uma lista contendo todas as jogadas validas de serem realizadas e um objeto que representa o estado atual do jogo. Estas informações são representadas pelas estruturas *Jogada* e *EstadoJogo*, explicadas nas seções anteriores.

Para realizar a jogada esta função deve decidir qual jogada escolher, dentre as jogadas permitidas (que são informadas na lista passada por parâmetro), e então retornar, nesta função, a jogada escolhida. Pela estrutura *EstadoJogo* é possível descobrir o jogador atual e os atributos desse jogador (algoritmo e heurística). Esses atributos são usados para decidir quais algoritmos serão executados para representar a inteligência artificial deste jogador. Podemos implementar quantas inteligências artificiais quisermos. Basta controlar qual implementação será chamada, a partir desses atributos. Para adicionar novas implementações de algoritmos e heurísticas, devemos configurar as enumerações *IndicadorAlgoritmo* e *IndicadorHeuristica*. As constantes que estão nessas enumerações são visíveis no painel de configuração de cada jogador, na interface gráfica, e podem ser escolhidas antes de iniciar o jogo. Estes parâmetros são estabelecidos aos jogadores para que possamos tratar, neste módulo *IA*, qual implementação de inteligência artificial escolher para o jogador.

Se o algoritmo falhar, lançando uma exceção, ou retornar uma jogada inválida, o módulo *GameLogic* identificará estes eventos e dará a vitoria para o jogador adversário.

Para garantir a integridade das informações no controle de GameLogic, os parâmetros enviados ao módulo IA são apenas cópias dos objetos originais, evitando que o módulo IA altere as informações a seu favor, ou até mesmo causando falhas ao jogo. O módulo *Regras* disponibiliza um serviço através do método estático *List<Jogada> calcularJogadasPermitidas(EstadoJogo)*. Este método retorna todas as jogadas possíveis de serem realizadas, segundo as regras do jogo, a partir de um estado de jogo definido pelo parâmetro passado à função. Este serviço é muito útil para a implementação das

estratégias, para que elas possam saber que jogadas podem ser executadas em momentos futuros do jogo, tornando desnecessário o conhecimento, por parte da estratégia de IA, das regras que regem as jogadas no jogo.

4.6 Controle de tempo e memória

Foram projetados dois sistemas para avaliação dos jogadores: um sistema para medir o tempo e outro para medir a quantidade de memória utilizada pelos jogadores durante a escolha de suas jogadas. Estes sistemas são utilizados pelo módulo *GameLogic* e estão implementados nas classes *TimeController* e *MemoriaController*, respectivamente.

4.6.1 Sistema de controle de tempo

O sistema de controle de tempo é responsável por coletar medidas de tempos gastos pelos jogadores no momento em que escolhem suas jogadas, e por processar os dados coletados para que se possa fazer análises e comparações entre os jogadores. O sistema define 3 métodos importantes para seu funcionamento:

- public void clear();
- public void logTime(IndicadorJogador jogador, long elapsedTime);
- public void processar().

O método *clear* é utilizado para configurações iniciais do controlador de tempo. Este método é chamado no início de uma partida.

O método *logTime* é utilizado para informar ao controle de tempo quanto tempo um jogador gastou para efetuar sua jogada. Este método é chamado dentro do método *escolherJogada* do módulo GameLogic. Para calcular o tempo gasto durante a jogada de um jogador é preciso capturar o tempo em dois momentos: antes de solicitar a jogada ao jogador, e após o jogador retornar a jogada escolhida. A diferença de tempos entre as duas medições definem o intervalo de tempo gasto pelo jogador. A cada jogada efetuada por cada jogador essas medições são coletadas e enviadas ao controle de tempo através do método *logTime*.

O método *processar* é executado ao final da partida. Este método processa as medições realizadas durante o jogo para que possamos mais tarde obter informações úteis, como o tempo da jogada mais demorada, a média de tempo de jogada de cada jogador e o desvio padrão entre as medições.

Os seguintes métodos foram implementados para se obter essas informações após o processamento:

- public long getSomaTotal(IndicadorJogador jogador);
- public long getMaiorTempo(IndicadorJogador jogador);
- public long getMediaTempo(IndicadorJogador jogador);
- public long getDesvioPadrao(IndicadorJogador jogador).

Estes métodos possuem nomes bem intuitivos que por si só já descrevem bem o que eles fazem. Cada método recebe por parâmetro um objeto que identifica o jogador ao qual queremos associar as informações buscadas. Então se queremos saber, por exemplo, a soma total dos tempos de cada jogada realizada pelo jogador 1, basta invocarmos o método *getSomaTotal* e passarmos a constante que representa o jogador 1 (IndicadorJogador.JOGADOR1).

O sistema de controle de tempo não precisa de implementações extras no módulo de IA para o seu funcionamento, diferente do sistema de controle de memória, como será explicado a seguir.

4.6.2 Sistema de controle de memória

O sistema de controle de memória utiliza os mesmos princípios do sistema de controle de tempo, coletando dados durante as jogadas dos jogadores e processando-os posteriormente. No entanto, não é fácil obter a quantidade de memória gasta durante o processamento.

O Java possui um "coletor de lixo" que faz o gerenciamento automático de memória utilizada em um software. Esse gerenciador é executado periodicamente, liberando espaços de memória que não são mais utilizados pela aplicação para que possam ser reutilizados. A desvantagem desse gerenciamento é que o programador não sabe exatamente quando ele será executado. A situação crítica que se tem é que a cada momento novos objetos são criados e perdidos durante a execução dos algoritmos, e em algum momento o coletor de lixo vai ser executado para fazer uma limpeza na memória, tornando impraticável a medição correta de quanta memória foi gasta no processamento de uma jogada. O coletor de lixo irá liberar o lixo de memória criado não só pelo processamento da jogada, mas também qualquer lixo de memória criado por qualquer outra parte da aplicação.

Para contornar a situação e melhorar a precisão, a medição de memória foi feita da seguinte forma:

Primeiro foi criado um método que informa a quantidade de memória utilizada pela aplicação no exato momento. Essa informação pode ser obtida pela classe Runtime. Esta classe possui os métodos *totalMemory* e *freeMemory* que informam a quantidade

de memória atualmente disponível para aplicação e a quantidade de memória livre que a aplicação pode usar. Para saber a quantidade de memória usada pela aplicação basta subtrair os valores informados por *totalMemory* e *freeMemory*. Vale ressaltar que o resultado desse cálculo é aproximado.

Em seguida foi criado um método que força a execução do coletor de lixo. Este método é executado antes de uma nova jogada, para que se possa medir com maior precisão a quantidade de memória gasta durante o processamento da jogada. A classe *Runtime* possui os métodos *runFinalization* e *gc* que nos auxiliam a forçar a execução do coletor de lixo. O método *runFinalization* executa os métodos de finalização de quaisquer objetos pendentes de finalização, e o método *gc* executa o coletor de lixo. No entanto, nem sempre o coletor de lixo libera toda a memória, e nem sempre o coletor é de fato executado quando solicitamos. Por esse motivo estes dois métodos são chamados exaustivamente 4000 vezes para que seja liberada a maior quantidade de memória possível, tornando a medição mais precisa. A cada ciclo de chamadas é calculado a quantidade de memória atual e comparada à quantidade de memória medida anteriormente. O laço de repetição de chamadas continua até que as medidas não sejam mais diferentes ou até que sejam executadas as chamadas 4000 vezes. Este valor (4000) foi encontrado por tentativas através de experimentos e se mostrou bem estável.

Para finalizar, foi criado um método, chamado *usedMemoryByte*, que realiza a chamada desses dois métodos criados, fazendo com que todo o lixo de memória seja recolhido antes de obter a quantidade de memória utilizada atualmente pela aplicação. Este método é chamado antes e depois da realização de cada jogada.

Para medir a quantidade de memória gasta pela aplicação durante o processamento de decisão da jogada do jogador não basta apurar a quantidade de memória usada antes e depois do processamento e calcular a diferença entre elas. Após o processamento, toda a memória que utilizamos pode ser descartada e provavelmente será "recolhida" pelo coletor de lixo. Por esse motivo a diferença seria sempre zero, e não estaríamos de fato medindo a memória utilizada pelo sistema. Diante deste fato a estratégia tomada foi realizar diversas apurações de memória durante o processamento de uma jogada. O ideal é que essas apurações sejam realizadas de forma uniforme e em momentos estratégicos do processamento da jogada, quando a quantidade de memória utilizada tende a ser maior. A cada apuração de memória realizada durante a jogada obtemos um valor de memória utilizada naquele momento e o tempo em que a apuração foi feita. Após a jogada ter sido finalizada temos um conjunto de apurações feitas em cada intervalo de tempo. Para calcularmos a quantidade média de memória utilizada pela jogada basta calcularmos a área formada pela interpolação linear entre as medições de memória ao longo do tempo e dividirmos pelo tempo total da jogada. Ao

calcularmos a área de memória devemos desconsiderar o valor de memória apurado antes de iniciar a jogada, pois esta quantidade de memória está sendo utilizada por outros módulos da aplicação.

A Figura 10 ilustra um exemplo de cálculo de memória utilizada durante uma suposta jogada realizada por um jogador. Neste exemplo a memória utilizada pelo sistema antes e depois da realização da jogada foi de 200 KB. Durante a jogada foram apuradas duas medições de memória com valores de 230 KB e 520 Kb, nos intervalos de tempo 10 ms e 20 ms. A jogada teve duração de 50 ms. Ao calcular a memória utilizada, o resultado foi de 6700 KB utilizados durante 50 ms, o que nos dá uma média de 134 KB de memória. Temos também a quantidade de memória máxima desta jogada, que foi de 520 KB - 200 KB, que é igual a 320 KB.

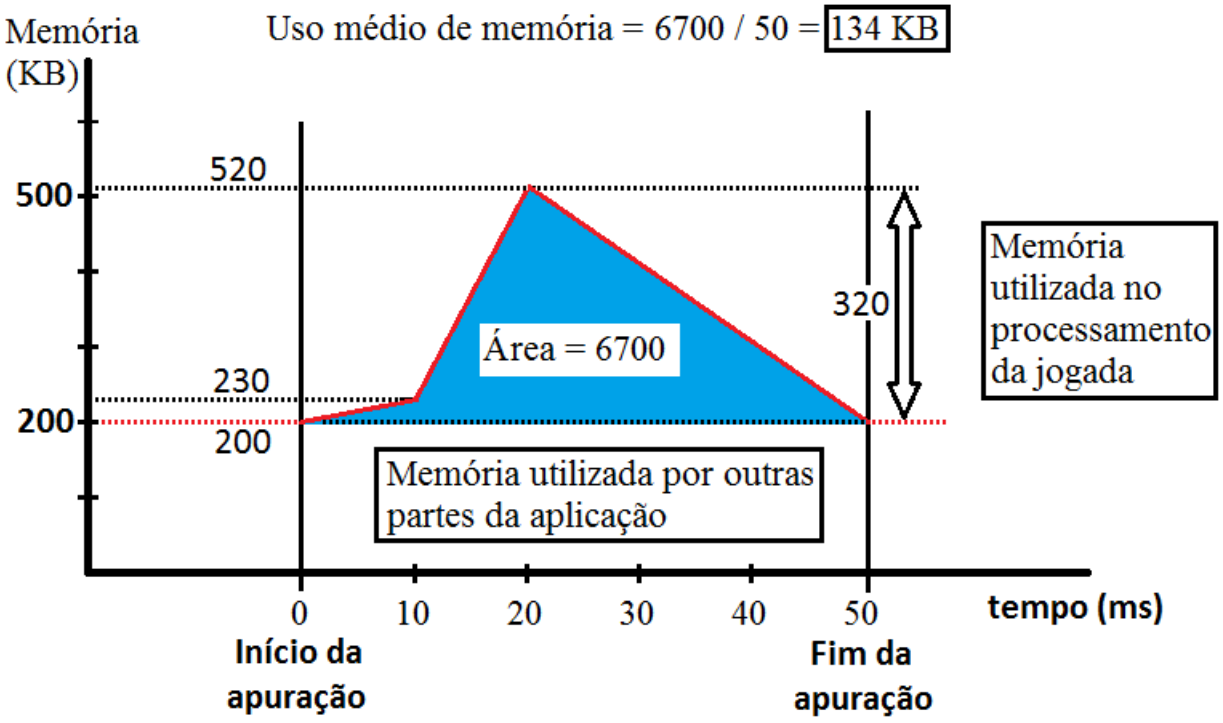


Figura 10 – Cálculo do uso de memória durante uma jogada.

Este sistema de controle de memória realiza todos os cálculos de memória das jogadas de forma automática. A única coisa que deve ser feito manualmente na hora de criar uma IA é solicitar novas apurações ao sistema. Essas apurações devem ser feitas em momentos estratégicos durante o processamento da escolha da jogada, no qual a quantidade de memória utilizada tende a ser maior, pois assim teremos uma aproximação mais realista da quantidade de memória que a IA está usando para realizar uma jogada. Para solicitar ao sistema uma apuração de memória basta chamar o método *MemoriaController.contabilizarMemoria()*. No sistema de controle de memória está implementado um mecanismo que já se encarrega de realizar automaticamente

os cálculos de memória média, memória máxima e desvio padrão durante a jogada e associá-los ao jogador que está jogando atualmente. As IAs Minimax e Alfa-Beta criadas neste trabalho realizam uma solicitação ao sistema em cada nodo alcançado pela árvore de decisão, fazendo com que as apurações fiquem bem distribuídas ao longo do processamento da jogada.

4.6.3 Configuração dos controles de tempo e memória

O sistema de controle de memória é muito custoso, pois realiza aproximadamente 4000 chamadas exaustivas ao coletor de lixo java. Por esse motivo foi criado um mecanismo para habilitar/desabilitar a análise de memória. Existe um componente no painel de configurações do jogo denominado *Habilitar Análise de Memória*, no qual o usuário pode selecionar para que a análise de memória seja habilitado pelo sistema.

Toda vez que um novo jogo é iniciado a aplicação configura o sistema de controle de memória, habilitando-o ou desabilitando-o conforme configurações do usuário. Caso o sistema seja desabilitado, toda vez que a aplicação solicitar uma apuração de memória, assim como a execução exaustiva do coletor de lixo do Java, o sistema irá apenas ignorar a solicitação e prosseguir com o processamento da jogada, sem causar nenhum impacto no processamento. Os resultados dos jogos também decidem quando deve ser mostradas estatísticas de memória de acordo com essa mesma configuração.

A análise de tempo sempre é realizada para todos os jogos. Vale ressaltar que a análise de memória não causa nenhum impacto na análise de tempo, pois as apurações de memória são feitas quase que instantaneamente, com custos baixíssimos de processamento. Apenas as solicitações de execução do coletor de lixo são custosas, que por sua vez são realizadas antes da apuração de tempo inicial e após a apuração de tempo final, evitando assim que as apurações de tempo contabilizem os tempos de limpeza de memória pelo coletor de lixo do Java.

Com a implementação destes controles de tempo e memória o sistema fica apto a realizar comparações entre as IAs implementadas, a nível de recursos computacionais.

4.7 Sistema de Comparação entre IAs

Na aplicação há um botão denominado *Comparar Jogadores*. Este botão é habilitado somente quando os dois jogadores são configurados para serem controlados por uma IA. Quando o usuário clica neste botão o sistema exibe uma caixa de texto pedindo que o usuário informe o número de comparações que devem ser feitas, e logo em seguida, o sistema aciona um controlador que realiza internamente a quantidade de jogos solicitada pelo usuário. Este controlador é programado na classe *CompararJogadoresController*. Uma barra de progresso é exibida abaixo do botão para que o usuário

possa acompanhar o progresso do processamento desta funcionalidade. Após todos os jogos finalizarem, o sistema exibe um relatório contendo as seguintes informações:

- Informações sobre o Algoritmo e Heurística associados a cada um dos jogadores.
- Quantidade de jogos realizados;
- Número de vitorias de cada jogador e número de empates, assim como seus respectivos percentuais;
- Maior tempo de processamento de uma jogada, para cada jogador (considerando todos os jogos);
- Maior quantidade de memória utilizada por uma jogada, para cada jogador (considerando todos os jogos);
- Média de número de jogadas por jogo;
- Média entre as jogadas mais demoradas de cada jogador, considerando a jogada mais demorada de cada jogo;
- Média de tempo de jogada de cada jogador, considerando todas as jogadas de todos os jogos;
- Desvio padrão da média de tempo de jogada de cada jogador;
- Média entre as maiores quantidades de memória utilizadas durante as jogadas de cada jogador, considerando a jogada mais custosa de cada jogo;
- Média de quantidade de memória utilizada durante as jogadas de cada jogador, considerando todas as jogadas de todos os jogos;
- Desvio padrão da média de memória utilizada durante as jogadas de cada jogador.

A implementação deste controlador utiliza os mesmos princípios para o controle de jogo comum. A diferença é que internamente este controlador aciona o módulo GameLogic várias vezes para que seja realizada a quantidade de jogos solicitadas pelo usuário. A cada iteração realizada internamente a este controlador, uma atualização é realizada na tela informando o percentual de conclusão do processamento, para que a barra de progresso seja sempre atualizada conforme o processamento.

Outra diferença sutil está associada ao tempo de espera entre jogadas. No controlador de jogo comum, a aplicação sofre pequenas pausas com duração de 500 ms, para que o usuário possa acompanhar melhor as jogadas entre dois jogadores controlados por IA. Esta configuração é feita na classe ArchitetureSettings, através da constante:

- private static long DEFAULT_UPDATE_TIME = 500;

Este valor deve ser alterado caso queira reconfigurar o tempo de espera entre jogadas. No controlador que realiza as comparações, o tempo de espera entre jogadas é ignorado para acelerar ao máximo a processamento interno de todos os jogos. Caso algum erro ocorra durante o processamento dos jogos, devido a alguma falha na implementação de uma IA, o sistema interrompe o processamento dos jogos e exibe uma mensagem de erro ao usuário, com informações sobre o erro ocorrido.

4.8 Testes

Durante e após o desenvolvimento do projeto, foram planejadas duas abordagens de testes: testes empíricos e testes unitários.

A primeira abordagem consiste em testes empíricos para testar as funcionalidades e os fluxos durante o uso da interface gráfica. Neste tipo de teste são analisados:

- Posicionamento correto dos componentes na tela;
- Comportamento da interface ao redimensionar a tela;
- Controle de ativação/inativação de componentes da tela;
- Início de jogo, com posicionamento correto das pedras no tabuleiro;
- Detecção de jogadas inválidas durante o jogo;
- Fluxo correto de troca de jogadores durante o jogo;
- Detecção de vitória;
- Detecção de empate;
- Comportamento correto do controle de desenho das casas do tabuleiro. Verifica-se se a casa fica vazia quando se captura uma pedra, e quando uma pedra sai de seu local de origem durante um movimento.
- Promoção de damas ao atingir o outro lado do tabuleiro.
- Movimentos livres de damas.
- Detecção correta de movimentos válidos.
- Avaliação geral das regras do jogo.
- Inicialização de novo jogo, após o término de um jogo.

- Fechamento do programa.

A segunda abordagem consiste em testes unitários, criados a partir de códigos Java, utilizando o framework *JUnit*. Os testes unitários cobrem as regras implementadas pelo módulo Regras, abordando diversas situações de jogo. Apesar de serem efetuados muitos testes empíricos no sistema, é interessante criar testes unitários que verificam regras pontuais no sistema. Sendo assim, os testes unitários se comportam da seguinte forma:

- São simulados diversos estados de jogo, configurados na estrutura EstadoJogo;
- Estes estados são passados ao método *atualizarRegras*, do módulo Regras;
- É verificado se o método *getJogadasPermitidas* retorna o conjunto de jogadas válidas esperado, segundo as regras.

O teste entre dois jogadores controlados por IA também é utilizado. Para este caso, o objetivo é testar o fluxo de início e fim de partida, retornando um vencedor ou um empate entre os jogadores.

O controle de fluxo de jogo também é testado em uma simulação de jogo completa, na qual o jogador 1 ganha do oponente após imobilizar todas as suas pedras, impossibilitando-o de jogar.

Na Figura 11 podemos observar os testes unitários criados neste projeto, divididos em três arquivos. O arquivo *GameLogicTest* testa o controle de fluxo do jogo; o arquivo *IATest* testa fluxos de jogos entre as estratégias de IA criadas; o arquivo *RegrasTest* testa as regras implementadas no módulo *Regras*.

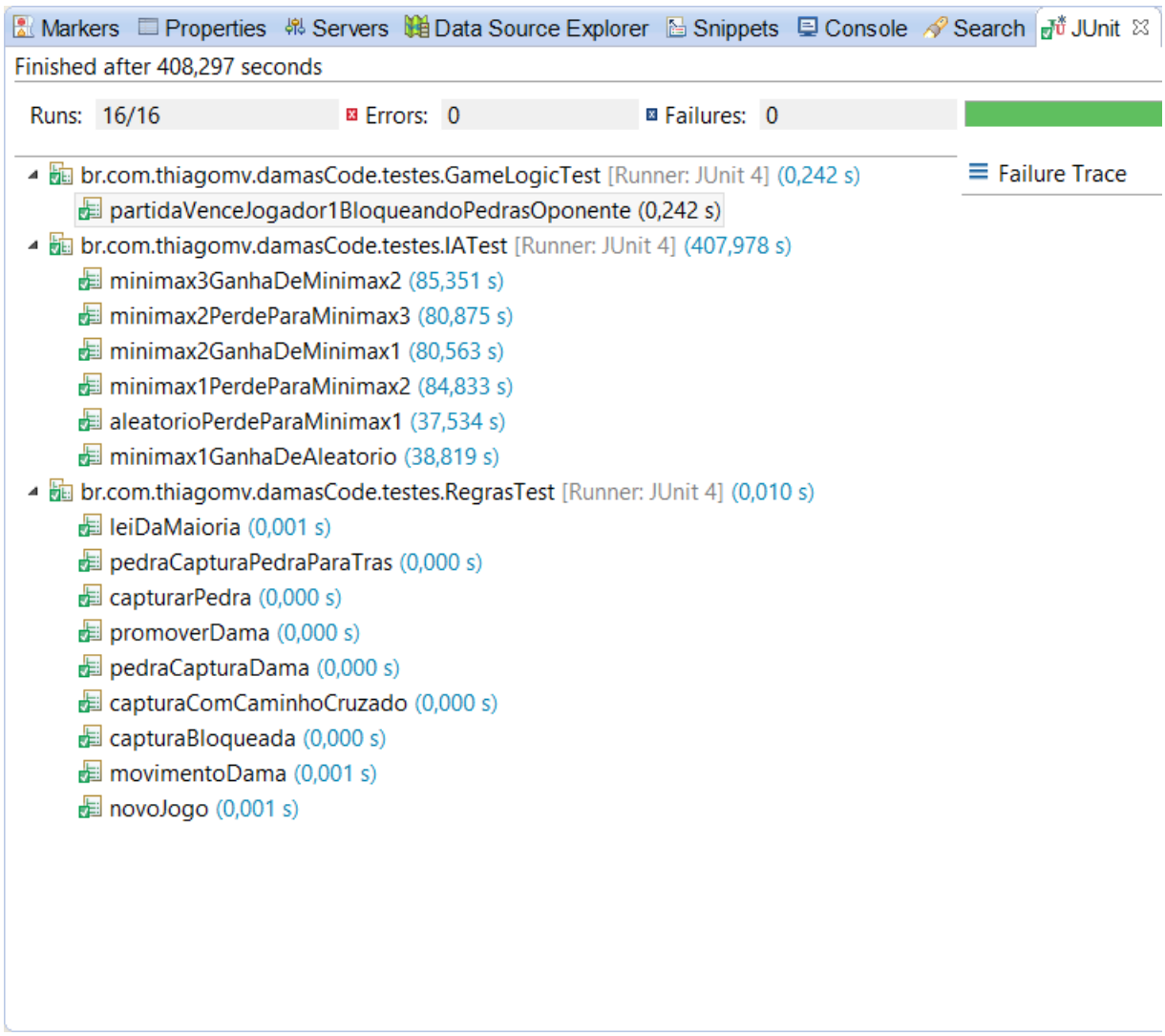


Figura 11 – Pós-execução de testes unitários pela IDE.

5 Resultados Computacionais

Durante a geração da árvore de decisão, nos algoritmos Minimax e Alfa-Beta, podem ser encontrados vários caminhos diferentes com o mesmo custo ótimo. A solução ótima neste caso será qualquer um destes caminhos com o melhor custo para o jogador. Neste caso a decisão é tomada de forma aleatória, o que garante resultados de jogos diferentes a cada novo jogo. Esta possibilidade de jogos com resultados diferentes torna viável a realização de comparação entre os jogadores, caso contrário o resultado entre os jogos de mesmos jogadores seria sempre o mesmo.

Após a conclusão da implementação de todo o sistema, assim como as estratégias de IA, foi possível realizarmos comparações entre as estratégias implementadas, contabilizando os resultados dos jogos, tempo médio de execução por jogada e memória média utilizada.

As estratégias de IA comparadas foram:

- Aleatório;
- Minimax;
- Alfa-Beta.

Para as estratégias Minimax e Alfa-Beta devemos indicar uma heurística. Cada heurística implementa a mesma função de avaliação, utilizada para avaliar o estado do tabuleiro, mas com profundidades diferentes para a geração da árvore de jogo. As Heurísticas implementadas são:

- H1_P3;
- H1_P4;
- H1_P5.

As Heurísticas H1_P3, H1_P4 e H1_P5 indicam que as árvores de jogo podem avançar até a profundidade 3, 4 e 5, respectivamente.

A função de avaliação implementada para as heurísticas foi

$$(B - P)/(B + P),$$

onde $B = Bn + 3 * Bd$, e $P = Pn + 3 * Pd$, tal que :

Bn = número de pedras brancas do tipo Normal

Bd = número de pedras brancas do tipo Dama

Pn = número de pedras pretas do tipo Normal

Pd = número de pedras pretas do tipo Dama

5.1 Apuração dos dados

Para apuração dos dados foram simulados 100 jogos entre cada uma das IAs do conjunto A contra cada uma das IAs do conjunto B:

O conjunto A inclui as IAs:

- Aleatório;
- Minimax com heurística H1_P3;
- Alfa-Beta com heurística H1_P4.

O conjunto B inclui as IAs:

- Aleatório;
- Minimax com heurística H1_P3;
- Minimax com heurística H1_P4;
- Minimax com heurística H1_P5;
- Alfa-Beta com heurística H1_P3;
- Alfa-Beta com heurística H1_P4;
- Alfa-Beta com heurística H1_P5.

A estratégia escolhida foi a de comparar jogos fixando um jogador do conjunto A, para que este realize jogos contra todos os jogadores do grupo B.

Para fazer a simulação, os seguintes passos foram seguidos:

- Fechar qualquer programa que esteja sendo executado no computador que possa interferir nos resultados da simulação;
- Executar a aplicação;

- Configurar o menu de jogadores para que a IA do conjunto A seja o primeiro jogador;
- Configurar o menu de jogadores para que a IA do conjunto B seja o segundo jogador;
- Marcar a opção de "Habilitar Análise de Memória";
- Clicar no botão "Comparar Jogadores";
- Indicar a quantidade de 100 simulações;
- Aguardar processamento das simulações e anotar os resultados finais da simulação.

Os resultados coletados das simulações são detalhados nas tabelas Tabela 1, Tabela 2 e Tabela 3. Tais resultados contém informações de:

- Número de vitórias/derrotas/empates do jogador B ao jogar contra o jogador A;
- Tempo médio de execução de cada jogada do jogador B;
- Média da quantidade de memória utilizada em cada jogada do jogador B;
- Média da quantidade máxima de memória utilizada em cada partida, pelo jogador B.

5.2 Análise dos dados

A primeira análise tem como foco a quantidade de vitórias entre os jogadores. Os gráficos Figura 12, Figura 13 e Figura 14 mostram os percentuais de vitórias, derrotas e empates de cada jogador do grupo B ao disputarem jogos contra cada um dos jogadores do grupo A. Podemos observar que esses percentuais são bem parecidos entre jogadores Minimax e Alfa-Beta que possuem a mesma Heurística. Podemos observar também que o jogador Aleatório está sempre com os percentuais mais desfavoráveis e que os jogadores que utilizam heurística H1_P5 possuem resultados melhores que as heurísticas H1_P4 e H1_P3, nessa ordem.

De fato, estes resultados são esperados por diversos motivos:

- Como a IA Alfa-Beta é uma otimização da Minimax, espera-se que ambas as implementações retornem os mesmos resultados.

Tabela 1 – Estatísticas de cada jogador do grupo B ao disputar 100 jogos contra o Jogador Aleatório.

	Vitórias	Derrotas	Empates	Média de tempos por jogada (ms)	Média de mem. por jogada (kB)	Média de mem. máx. por jogo (kB)
Aleatório	48	50	2	0,020	1735,406	7507,392
Minimax H1_P3	99	0	1	0,636	1906,673	7997,440
Minimax H1_P4	100	0	0	2,436	2081,187	8768,113
Minimax H1_P5	100	0	0	12,619	6592,807	39791,182
Alfa-Beta H1_P3	100	0	0	0,551	1816,660	7956,394
Alfa-Beta H1_P4	100	0	0	1,920	2055,278	8510,055
Alfa-Beta H1_P5	99	0	1	8,026	4418,359	28838,604

Tabela 2 – Estatísticas de cada jogador do grupo B ao disputar 100 jogos contra o Jogador Minimax - H1_P3.

	Vitórias	Derrotas	Empates	Média de tempos por jogada (ms)	Média de mem. por jogada (kB)	Média de mem. máx. por jogo (kB)
Aleatório	0	100	0	0,021	1706,246	6756,998
Minimax H1_P3	45	42	13	0,507	1800,334	8034,141
Minimax H1_P4	82	9	9	2,405	2105,381	9401,937
Minimax H1_P5	85	4	11	13,130	7154,737	46705,761
Alfa-Beta H1_P3	38	41	21	0,488	1796,425	8026,807
Alfa-Beta H1_P4	76	14	10	2,196	2090,714	9373,182
Alfa-Beta H1_P5	83	7	10	8,419	4832,910	33396,525

- A implementação das heurísticas seguem os mesmos métodos de avaliação de estados de jogo. A única diferença entre elas está no nível de profundidade da

Tabela 3 – Estatísticas de cada jogador do grupo B ao disputar 100 jogos contra o Jogador Alfa-Beta - H1_P4.

	Vitórias	Derrotas	Empates	Média de tempos por jogada (ms)	Média de mem. por jogada (kB)	Média de mem. máx. por jogo (kB)
Aleatório	0	100	0	0,017	1735,312	6043,567
Minimax H1_P3	8	74	18	0,500	1812,745	7557,103
Minimax H1_P4	27	45	28	3,487	2543,290	12230,384
Minimax H1_P5	66	18	16	15,663	8777,897	57624,083
Alfa-Beta H1_P3	14	71	15	0,464	1735,116	7369,893
Alfa-Beta H1_P4	34	31	35	3,348	2481,268	12229,127
Alfa-Beta H1_P5	66	14	20	11,344	7131,156	47823,490

árvore de jogo que cada algoritmo consegue atingir. Desta forma, espera-se que quanto maior for o nível de profundidade, maior será sua capacidade de atingir melhores resultados. Isso é refletido nos resultados encontrados durante os experimentos.

- O jogador Aleatório toma decisões sem nenhum critério lógico. Apenas escolhe, aleatoriamente, uma jogada possível de ser realizada. Como esperado, este jogador possui os piores resultados. Vale a pena observar que apesar deste jogador realizar jogadas aleatórias, com sorte ele ainda pode obter algumas poucas vitórias e/ou empates.

As médias de tempos de jogadas dos jogadores Minimax e Alfa-Beta foram coletadas durante 100 simulações de jogos realizadas contra cada um dos jogadores do grupo A. O gráfico da Figura 15 ilustra os resultados obtidos contra o jogador *Aleatório* (Os resultados obtidos contra outros jogadores foram similares). Como pode ser observado, o tempo gasto pelos algoritmos Minimax e Alfa-Beta aumenta na medida com que é aumentada a profundidade da árvore percorrida por cada heurística. Também nota-se que o algoritmo Alfa-Beta tem melhor desempenho quando comparado ao Minimax.

Esta mesma análise pode ser feita para o gráfico da Figura 16, no qual se compara a média de memória gasta em cada jogada, e a média entre as jogadas mais custosas

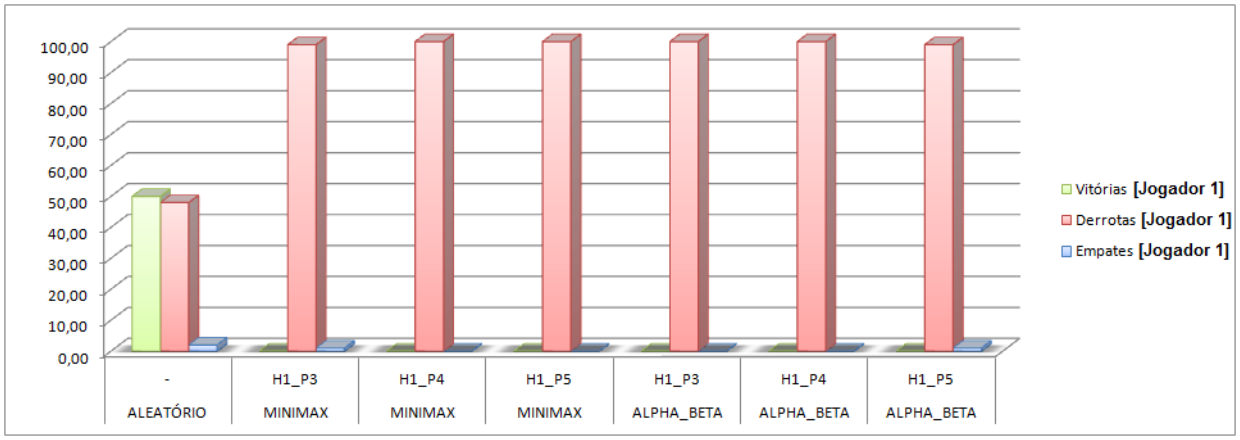


Figura 12 – Percentual de vitórias, empates e derrotas do jogador Aleatório ao realizar 100 jogos contra cada jogador do grupo B.

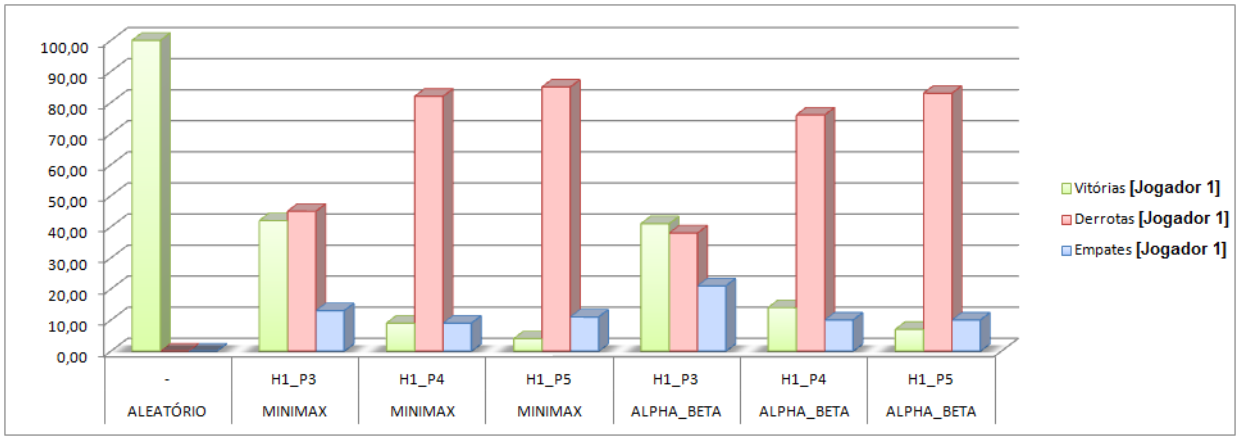


Figura 13 – Percentual de vitórias, empates e derrotas do jogador Minimax de heurística H1_P3 ao realizar 100 jogos contra cada jogador do grupo B.

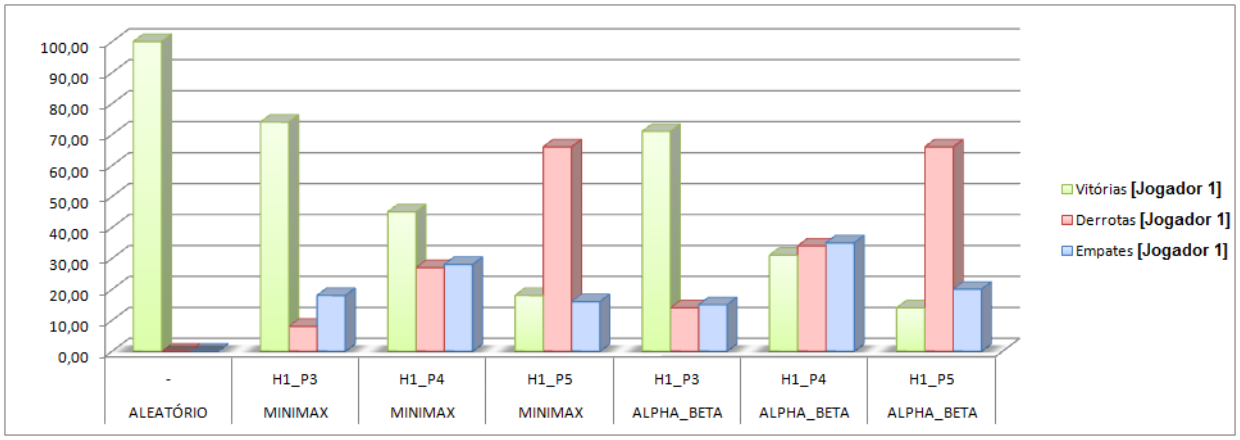


Figura 14 – Percentual de vitórias, empates e derrotas do jogador Alfa-Beta de heurística H1_P4 ao realizar 100 jogos contra cada jogador do grupo B.

de cada partida. De forma semelhante à análise de tempo, a quantidade de memória exigida por cada algoritmo aumenta na medida com que se aumenta a profundidade de busca na árvore de jogo, configurada em cada heurística.

Estes resultados podem ser explicados pelas podas feitas durante o processamento do algoritmo Alfa-Beta. O algoritmo Minimax percorre todos os estados possíveis, até o nível de profundidade definido. Já o Alfa-Beta é diferente: ele realiza podas, quando possível, evitando o processamento de alguns nodos da árvore. Isso diminui o tempo de processamento, e também faz com que a quantidade média de memória seja reduzida, já que a memória que seria utilizada durante o processamento dos nodos que foram podados não será mais necessária.

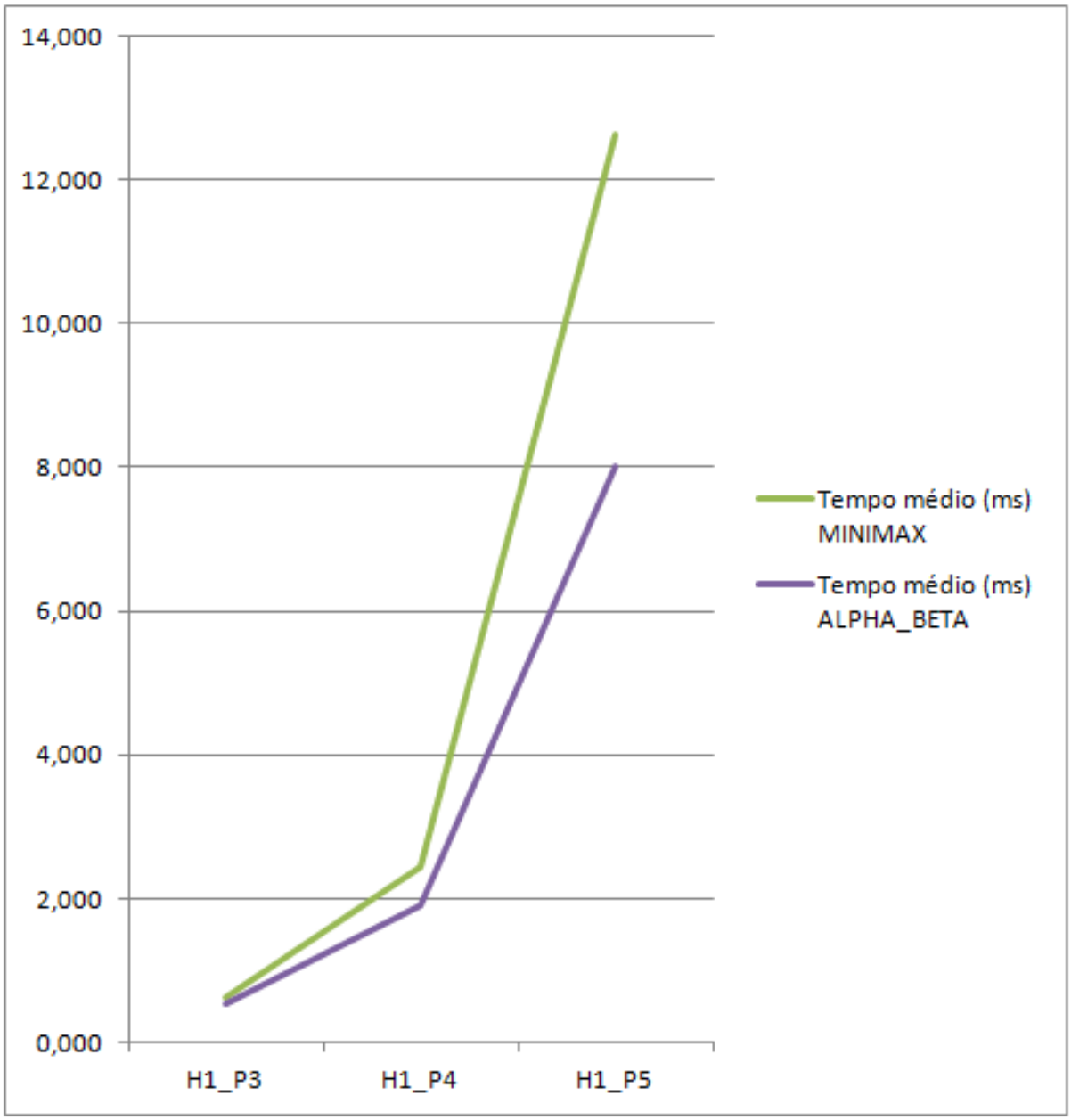


Figura 15 – Médias de tempos gastos pelos jogadores Minimax e Alfa-Beta, em cada heurística, ao disputarem 100 jogos contra o jogador Aleatório.

Pela tabela de apuração de dados (Tabela 1) também pode ser observado que o algoritmo Aleatório é o mais rápido, mais leve (em termos de memória), e o que menos vence. Jogar um jogo de damas de forma aleatória realmente não é uma boa estratégia!

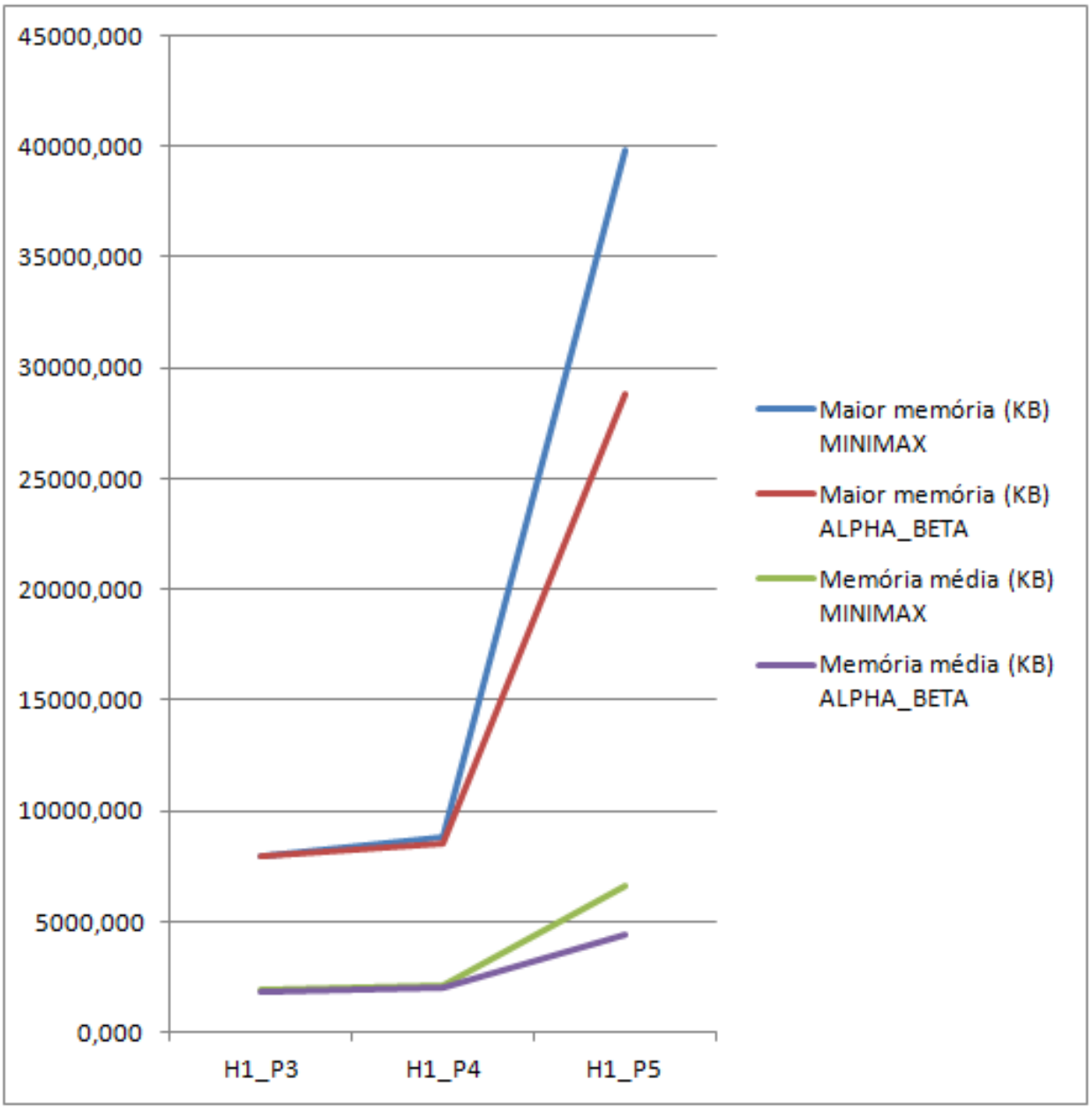


Figura 16 – Médias de memória utilizada pelos jogadores Minimax e Alfa-Beta, em cada heurística, ao disputarem 100 jogos contra o jogador Aleatório.

6 Conclusão

Com a realização deste trabalho foi possível investigar as principais técnicas de inteligência artificial no universo dos games. Baseado na técnica de árvores de decisão, relacionada à Teoria dos Jogos, foram planejados alguns dos algoritmos de inteligência artificial que os jogadores de damas terão. Algumas destas IAs implementam os algoritmos Minimax e pode Alfa-beta.

A interface gráfica criada demonstrou ser adequada para o jogo entre jogadores humanos, permitindo, também, a configuração das estratégias de IA dos jogadores. Os testes realizados até o momento indicam que o sistema está funcionando corretamente, segundo as regras do jogo, viabilizando, assim, a continuação deste trabalho. O uso da arquitetura projetada foi exemplificado com a implementação de diversas estratégias de IA e a comparações e análises entre elas.

Os resultados encontrados na análise estão de acordo com o esperado. Foram encontrados, nos resultados, que o algoritmo Minimax possui resultados similares aos do jogador Alfa-Beta, em termos de vitórias. Afinal, Alfa-Beta é uma otimização de Minimax. Também por este motivo, Alfa-Beta gasta menos memória e menos tempo de execução que Minimax, pois ao podar ramos da árvore de decisão, o processo de busca é simplificado, exigindo assim menos tempo de processamento e memória.

Os resultados também revelam que quanto mais profunda é a busca na árvore de decisão, mais tempo e memória são necessários. Conforme os gráficos apresentados, este crescimento tem uma forma que se assemelha a uma curva exponencial.

A conclusão mais importante, diante dos resultados, é que o sistema se mostrou confiável e eficaz na comparação entre as estratégias, conforme planejado em nossos objetivos. O sistema permitiu a programação de diversas estratégias e a comparação entre elas, a partir de uma interface de programação simples e objetiva. A interface gráfica é suficiente para a interação com o usuário e ainda permite que jogadores humanos joguem contra as próprias estratégias implementadas.

6.1 Trabalhos futuros

Uma boa proposta para trabalhos futuros é a de reduzir o tempo de processamento durante a análise de memória das estratégias de IA.

Outra proposta é a criação de novas estratégias. Este trabalho teve como foco estratégias que utilizam árvores de decisão. Uma boa alternativa seria a implementação de estratégias que façam uso de redes neurais.

A interface gráfica também pode ser melhorada, mostrando, por exemplo, a pedra com a qual o adversário realizou a última jogada. Outra funcionalidade que pode ser implementada para melhorar a jogabilidade é a melhoria dos feedbacks da jogada do adversário quando o mesmo realiza várias capturas de uma só vez. Atualmente o sistema apenas mostra o estado final da jogada, após o adversário realizar todas as capturas da jogada. Isso pode deixar o usuário meio confuso, pois nem sempre é fácil perceber a jogada realizada quando o adversário captura três pedras de uma só vez. O sistema pode melhorar, fazendo com que pausas sejam realizadas entre cada captura, além de atualizar a interface gráfica para que o usuário possa ver cada captura acontecendo, passo a passo.

Referências

BARCELOS, A. R. A. **D-VisionDraughts: Uma rede neural jogadora de damas que aprende por reforço em um ambiente de computação distribuída**. Dissertação (Mestrado) — Universidade Federal de Uberlâdia, 2011. Disponível em: <http://repositorio.ufu.br/bitstream/123456789/622/1/D-VisionDraughtsRede.pdf>. Citado na página 10.

BARRICHELO, F. **A Ciência da Estratégia**: Insights da teoria dos jogos para competir e colaborar. 2014. Disponível em: <http://www.cienciadaestrategia.com.br/index.asp>. Acesso em: 16 de agosto de 2014. Citado na página 7.

BRAGA, A. de P.; CARVALHO, A. P. de Leon F. de; LUDERMIR, T. B. **Redes Neurais Artificiais**. 2. ed. Rio de Janeiro: LTC, 2007. 226 p. ISBN 978-85-216-1564-4. Citado 2 vezes nas páginas 4 e 8.

CASSANDRAS, C. G. **Discrete Events Systems**: Modeling and performance analysis. [S.l.]: Aksen Associates Incorporated Publishers, 1993. Citado na página 4.

COLATO. **Jogos: Damas e as jogadas perfeitas**. 2007. Disponível em: <http://emassa.blogspot.com.br/2007/08/jogos-damas-e-as-jogadas-perfeitas.html>. Acesso em: 23 de abril de 2014. Citado na página 1.

CORREIA, M. **Jogo de Damas: Regras Oficiais**. 2011. Http://www.xadrezregional.com.br/regrasdm.html. Disponível em: <http://www.xadrezregional.com.br/regrasdm.html>. Citado na página 3.

FLOR, P. **História e as regras do jogo de damas**. 2010. Disponível em: <http://www.livresportes.com.br/reportagem/historia-e-as-regras-do-jogo-de-damas>. Acesso em: 23 de abril de 2014. Citado na página 1.

GARCIA, R. Cientistas criam robô imbatível nas damas. **Folha de S. Paulo**, Julho 2007. Disponível em: <http://www1.folha.uol.com.br/fsp/ciencia/fe2007200702.htm>. Acesso em: 23 de abril de 2014. Citado na página 1.

GOODMAN, D.; KEENE, R. **Man versus Machine**: Kasparov versus deep blue. Cambridge, Massachusetts: H3 Publications, 1997. Citado na página 1.

KURZWEIL, R. **The Age of Intelligent Machines**. Cambridge, Massachusetts: MIT Press, 1990. Citado na página 4.

LEITE, R. M. S. R. **Jogo das "Damas"** ***clássicas***. Porto, Portugal, 1999. Trabalho realizado para a Disciplina Metologias da Inteligência Artificial do Mestrado em Inteligência Artificial e Computação. Disponível em: <http://paginas.fe.up.pt/~eol/IA/DAMAS/RELATORIO/relatorio.html>. Citado na página 9.

LUZ, L.; LUZ, M.; TEóFILO, M. Jogo de damas utilizando realidade aumentada e inteligência artificial para telefones celulares. In: **IX SBGames - Computing Track - Short Papers**. Florianópolis, SC: SBC, 2010. p. 300–303. Disponível em: <http://www.sbgames.org/papers/sbgames10/computing/short/Computing_short21.pdf>. Citado na página 10.

PEDRYCZ, W.; GOMIDE, F. **An Introduction to Fuzzy Sets**: Analysis and design. [S.l.]: MIT Press, 1998. Citado na página 4.

RUSSELL, S.; NORVIG, P. **Inteligência Artificial**. Rio de Janeiro: Elsevier, 2004. 1021 p. ISBN 5-352-1177-2. Citado 6 vezes nas páginas 1, 4, 7, 9, 14 e 16.

TATAI, V. K. **Técnicas de Sistemas Inteligentes Aplicadas ao Desenvolvimento de Jogos de Computador**. Dissertação (Mestrado) — Unicamp, 2006. Disponível em: <ftp://ftp.dca.fee.unicamp.br/pub/docs/gudwin/publications/TeseTatai.pdf>. Citado 2 vezes nas páginas 4 e 7.

Apêndices

APÊNDICE A – Código do Projeto

O projeto contendo o código completo deste trabalho está disponível na internet através do link `https://github.com/Kuem/DamasCode/`.