UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
ENGENHARIA DE COMPUTAÇÃO

ARTHUR CARVALHO RAUTER

Plataforma Computacional de Apoio ao Projeto de Extensão Jogos Lógicos de Tabuleiro

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

Orientador: Prof. Dr. Renato Perez Ribas

Porto Alegre
2014

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Rauter, Arthur Carvalho

Plataforma Computacional de Apoio ao Projeto de Extensão Jogos Lógicos de Tabuleiro / Arthur Carvalho Rauter. – Porto Alegre: ECP da UFRGS, 2014.

47 f.: il.

Trabalho de conclusão (graduação) – Universidade Federal do Rio Grande do Sul. Engenharia de Computação, Porto Alegre, BR–RS, 2014. Orientador: Renato Perez Ribas.

I. Ribas, Renato Perez. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Prof. Sérgio Roberto Kieling Franco
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador da ECP: Prof. Marcelo Götz
Bibliotecário-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“A wizard is never late, nor is he early. He arrives precisely when he means to.”
— J.R.R. TOLKIEN

AGRADECIMENTOS

Apesar de se tratar de um trabalho de *conclusão de curso*, os agradecimentos a seguir são para as pessoas que foram valiosas em qualquer período do *curso* e não somente durante a sua *conclusão*. Uma distinção um tanto desnecessária, uma vez que todos que participaram da minha vida durante o *curso* inevitavelmente influenciaram na sua *conclusão*, mas importante para que eu possa mencionar livremente todos que eu gostaria.

Agradeço à minha mãe, Rachel Beatris Rauter, pelo seu apoio, sua compreensão e seu amor. Esse agradecimento estende-se aos demais membros da minha família materna: minha avó Beatriz, minhas dindas e meus dindos, os que são meio cariocas e minhas primas, que, assim como minha mãe, também sempre me apoiaram no festeiro estilo Carvalho.

Agradeço ao meu pai, André Rauter, pelo seu apoio, sua compreensão e seu amor. Esse agradecimento estende-se aos demais membros da minha família paterna: minha avó Ruth e meu avô Norton, minha dinda e meu dindo e meus primos, que, assim como meu pai, também sempre me apoiaram no inconfundível estilo Rauter.

Agradeço ao meu irmão, Diogo Carvalho Rauter, pela confiança mútua e a certeza de que eu sempre terei com quem contar.

Pela amizade e especialmente pelo companheirismo, agradeço aos meus colegas de faculdade: Alessandra Leonhardt, Jeckson Souza, Luís Henrique Reinicke e especialmente Lucas Schons, cujo apoio foi definitivo na realização deste trabalho de conclusão.

Dafür dass Berlin "mein zweites zuhause" war, bedanke ich mich bei Roswitha Paul-Walz, Josiane Aliste, Jean Santos, Michael Smart und inbesondere Amelie Thiel. Ich bedanke mich auch bei der Technische Universität Berlin, CAPES und DAAD für die Möglichkeit in Deutschland zu studieren.

Por tudo aquilo que entende-se por amizade; agradeço aos amigos das antigas, mas nem por isso menos atuais: Lucas Mangoni, Lucas Triches, Diego Santos, Henrique Grazziotin, Henrique Ullmann, Mauro Motta, Jorge Loureiro, Caio Miranda e Otto Grimm; agradeço aos "amigos do meu irmão" que são assim carinhosamente intitulados, mesmo que já façam parte do grupo chamado apenas de "meus amigos"; agradeço aos meus vizinhos, entre eles Anderson Turra e Luca Matzenbacher.

Agradeço à Sônia, por se preocupar e por manter a casa fisica- e espiritualmente em ordem.

Agradeço aos meus professores de música e também amigos, Allan César e Igor Dornelles, pelos ensinamentos, em especial a escala pentatônica, e por tornarem da música tocada e ouvida parte indissociável da minha vida.

Agradeço à Universidade Federal do Rio Grande do Sul e aos seus professores pela minha formação acadêmica e em especial ao meu orientador, o Prof. Dr. Renato Perez Ribas, pela serenidade com a qual ele lidou com a orientação deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

RESUMO

O Programa de Extensão Jogos Lógicos de Tabuleiro envolve atividades com Jogos Lógicos de Tabuleiro. Centenas de jogos foram organizados de forma didática em ordem crescente de complexidade. As atividades tem sido realizadas em escolas e centros comunitários com o intuito de desenvolver o raciocínio lógico dos participantes.

O Programa também aborda o desenvolvimento de jogos computacionais para serem uma das modalidades das atividades. Este trabalho, portanto, visa criar uma plataforma computacional de apoio para essas atividades. Mais especificamente, serão abordados os jogos que o programa classifica como jogos de captura. O desenvolvimento dessa plataforma consiste em estudar os jogos de captura e construir um ambiente que agregue esses jogos e permita que eles sejam apresentados em ordem didática. Para permitir que os jogadores joguem contra o computador, foram estudadas implementadas técnicas de inteligência artificial.

O resultado é uma plataforma para jogos de captura para o sistema Android e um aplicativo, criado com essa plataforma, que implementa uma sequência de jogos de captura.

Palavras-chave: Jogos Lógicos de Tabuleiro, Android.

Computational support platform for the Extension Program Logical Game Boards

ABSTRACT

The Extension Program Logical Board Games is based on activities with logical board games. Hundreds of games where organized in a didactic way and ordered by their complexity. The activities of the program took place in schools and community centers aiming to develop the logical thinking of the participants.

The Extension Program also works with the development of computer games to be used in its activities. This work, therefore, aims to create a computational platform to support such activities. More specifically, it will deal with the games classified by the Extension Program as capture games. The development of this platform consists of studying the capture games and constructing an environment that groups those games and allows them to be presented in a didactic fashion. So that the players may play against the computer, artificial intelligence techniques were studied and implemented.

The aftermath is a platform for capture games for the Android system and an application, created with this platform, that implements a didactic sequence of capture games.

Keywords: Logical Game Boards, Android.

1 INTRODUÇÃO

Jogos eletrônicos são programas de computador com o intuito de entreter e divertir o usuário. São cada vez mais comuns, especialmente por causa do crescente número de dispositivos móveis a preços acessíveis. Não é mais necessário um console para se tornar um jogador. Praticamente qualquer celular hoje em dia possui algum jogo embutido, ou possibilita que você baixe jogos.

Existe uma grande variedade de jogos que podem ser classificados de várias formas. São propostas classificações baseadas em características tais como a dimensionalidade dos gráficos, a quantidade de jogadores envolvidos, e desafio do jogo. A característica de dimensionalidade dos gráficos resume-se em 2D e 3D. A característica "quantidade de jogadores" envolvidos pode variar bastante: um único jogador, jogos de dois jogadores, jogos que são disputas entre dois times pequenos e até jogos conhecidos como MMO "Massive Multiplayer Online" onde centenas de jogadores compartilham o mesmo mundo virtual. A característica desafio de jogo é ainda mais variada: existem os famosos jogos plataforma, os de tiro em primeira pessoa, os de quebra-cabeças e os mais focados na narrativa, entre várias outras possíveis classificações. Neste trabalho foram discutidos jogos mais antigos que a computação, mas que também tem seu espaço no mundo digital, que são os clássicos jogos de tabuleiro.

Xadrez, Damas e Alquerque são apenas alguns exemplos de jogos clássicos de tabuleiro que de tão antigos, já fazem parte da nossa cultura e são mundialmente conhecidos. Esse jogos foram encontrados nos trabalhos de ALLUÉ (2002), BURNS (1998) e RIPOLL; CURTO (2011). Em tempos onde os jogos estão com gráficos progressivamente mais avançados e realistas e onde o número de jogadores simultâneos só aumenta, a simplicidade desses jogos de tabuleiro pode não ser muito atraente. Porém, uma das grandes vantagens desses jogos é que eles exigem um esforço de concentração e podem ser usados como forma de exercitar o raciocínio lógico, sem a necessidade de dispor de uma grande infraestrutura de laboratórios e salas de aula sofisticadas, dando a impressão que é mais uma atividade lúdica do que uma atividade cansativa no processo de ensino-aprendizagem.

1.1 Projeto de Extensão Lobogames

O projeto de extensão Lobogames é um projeto da PROREXT da UFRGS que utiliza jogos lógicos de tabuleiro para auxiliar na formação do raciocínio de alunos e professores. O diferencial deste projeto é justamente a forma como os jogos são apresentados. Uma sequência didática é sugerida, onde os jogos são jogados em ordem crescente de dificuldade. A motivação central do projeto é exercitar o pensamento e o raciocínio das pessoas. Os jogos são uma forma divertida de realizarmos esse exercício. Além disso, os

jogos nos ensinam que somos capazes de adquirir muita informação em pouco tempo se ela for apresentada de forma gradual e com emoção.

O objetivo principal do projeto é divulgar e reviver o interesse pelos jogos lógicos de tabuleiro, pois acredita-se nos benefícios do exercício do raciocínio lógico provido por essa atividade. Os objetivos específicos do projeto são: a inserção dos jogos nas atividades escolares regulares, a formação de professores e oficineiros, a criação de clubes de xadrez, promover olimpíadas escolares de jogos lógicos de tabuleiro em todas as suas modalidades, promover atividades recreativas para a maior idade, construir os jogos com material reciclável, resgatar o estudo desses jogos, promover a pesquisa na área da pedagogia em relação ao impacto deste tipo de atividade no ensino e promover o surgimento de empreendedores no desenvolvimento e na comercialização desses jogos em versão eletrônica e artesanal.

Dentro das escolas os jogos são construídos usando materiais reciclado e são jogados em três modalidades: tabuleiro, jogo "gigante" e jogo "vivo". Na modalidade tabuleiro, joga-se com o tabuleiro impresso e tampinhas de garrafas PET. No jogo "gigante" o tabuleiro é desenhado no chão e as peças são garrafas cheias de água com corante. Por fim, o jogo "vivo" é onde pessoas com coletes coloridos são as peças e precisam se posicionar sobre o tabuleiro. Cada modalidade apresenta o jogo de forma diferente, segundo o trabalho de GIORDANI; RIBAS (2014). Isso modifica a forma como os jogadores se relacionam com o jogo. Na modalidade tabuleiro, a atenção do jogador e do seu cérebro não é dispersada pelo movimento corporal e é fácil visualizar todo o tabuleiro de uma só vez. No jogo "gigante" a visão espacial é bem diferente e consequentemente é mais difícil prever as jogadas. Quando as pessoas passam a ser as peças no jogo "vivo", a visão espacial fica ainda mais complicada e as jogadas passam a ser uma decisão do grupo.

A sequência didática e as modalidades servem para evitar que estratégias sejam memorizadas. O objetivo é retirar os jogadores da sua zona de conforto e fazer com que eles lidem com situações inusitadas. Nesses momentos é que o raciocínio lógico é exercitado.

De acordo com suas características em comum, os jogos do projeto estão divididos em seis módulos: bloqueio e alinhamento, deslocamento, posicionamento, captura, caça e xadrez.

1.2 Motivação

Não existe nenhum software que agregue os jogos nos módulos e os apresente da forma didática esperada, nem nada semelhante. No máximo, encontra-se os jogos avulsos. É de interesse do projeto que os participantes das atividades do projeto possam continuar jogando depois que se encerram as atividades, sem depender de outras pessoas. Uma das motivações é desenvolver um software que permite que as pessoas joguem os jogos sozinhas e em qualquer lugar, contra o computador.

Outra motivação é o empreendedorismo. É fácil comercializar software, especialmente quando o software for um aplicativo para dispositivos móveis. Jogos digitais estão entre os aplicativos mais baixados e podem ser um bom começo para uma pequena empresa.

Também interessante é o estudo e a aplicação dos conceitos de áreas como: a engenharia de software, a inteligência artificial, a teoria de grafos, computação gráfica e sistemas embarcados.

1.3 Objetivo

O objetivo deste trabalho é desenvolver uma plataforma que agregue vários jogos lógicos de tabuleiro para Android, onde o usuário joga contra o computador. Os jogos são apresentados em ordem didática e de complexidade e dificuldade. Isso é feito separando os jogos em níveis. Cada nível é composto por um ou mais jogos do módulo de captura. A escolha dos jogos e os níveis busca apresentar uma complexidade crescente. O objetivo secundário, mas essencial para a realização deste trabalho, é o estudo de inteligência artificial e a experiência prática com programação para sistemas embarcados.

1.4 Estrutura do Texto

No próximo capítulo serão apresentados os jogos de captura e suas regras. Depois será apresentada uma revisão técnica onde os conceitos de inteligência artificial serão explicados. O capítulo de implementação documenta o código que foi escrito para a realização desse trabalho. No final, tem-se os resultados e as conclusões.

2 JOGOS DE CAPTURA

Jogos de captura são jogos onde o objetivo de cada jogador é capturar todas as peças do jogador adversário. Jogos desse tipo que estão no projeto Lobogames são: **Alquerque, Felli, Dash-Guti, Pretwa, Sixteen Soldiers, Awithlaknannai, Damas, Tablut, Hasami Shogi, Seega, Surakarta, Fanorona e Four Field Kono**. Esses jogos foram divididos em três famílias de acordo com suas semelhanças.

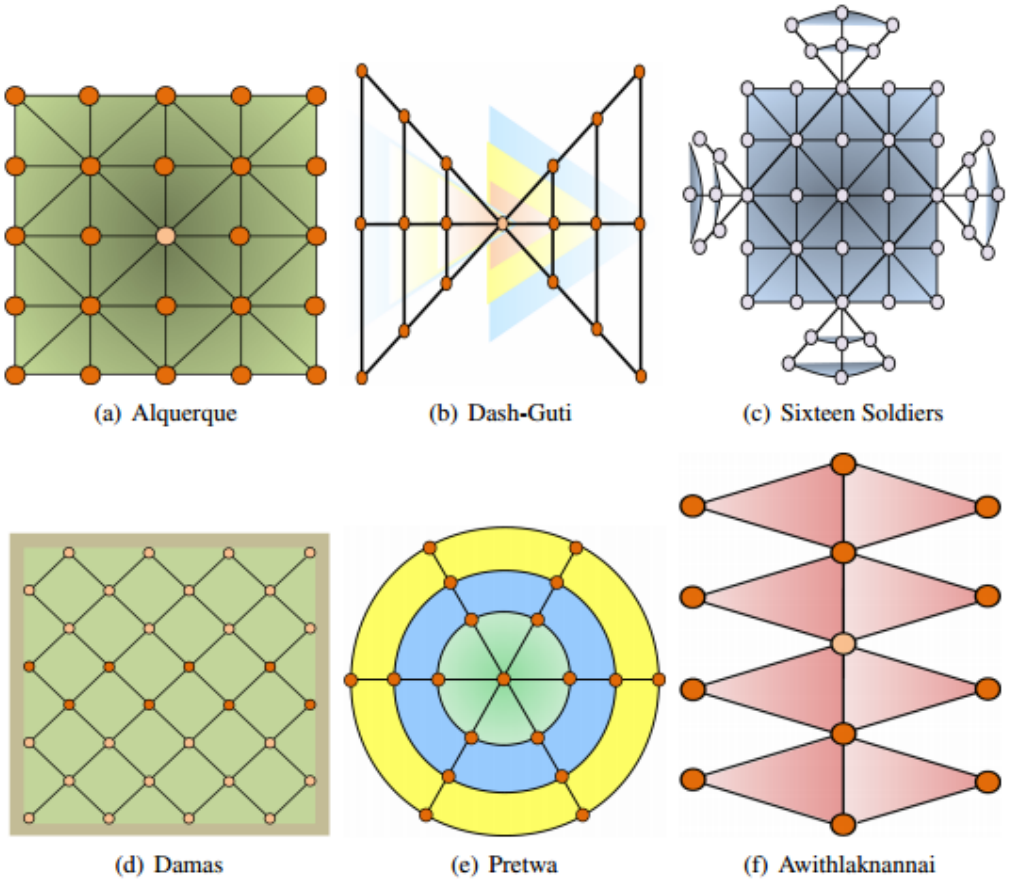


Figura 2.1: Jogos de captura da família do Alquerque
Fonte: inf.ufrgs.br/lobogames/

A primeira família é a do Alquerque, que contém os jogos: **Alquerque, Felli, Dash-Guti, Pretwa, Sixteen Soldiers, Awithlaknannai e Damas**. As peças se movimentam seguindo as linhas do tabuleiro, normalmente uma casa por vez. Uma captura ocorre quando uma peça salta sobre uma peça adversária. As capturas são compulsórias e mais de uma captura pela mesma peça no mesmo turno são possíveis.

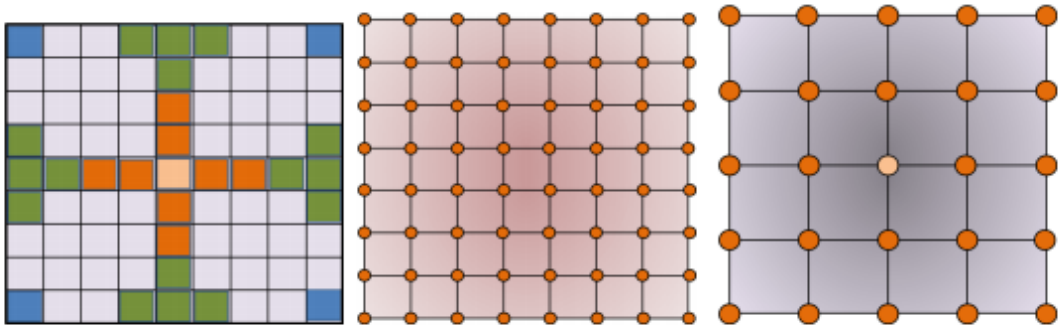


Figura 2.2: Jogos de captura da família Custodian Capture
Fonte: inf.ufrgs.br/lobogames/

A segunda família é a da *Custodian Capture* ou *Custodial Capture*. Nela estão: **Tablut, Hasami Shogi e Seega**. As peças se movimentam apenas ortogonalmente. No Hasami Shogi e no Tablut elas podem se mover mais de uma casa por vez (como a torre do xadrez). Para realizar uma captura, o jogador deve cercar uma peça do adversário horizontalmente ou verticalmente. A captura não é compulsória e só são possíveis capturas múltiplas no mesmo turno quando um único movimento cerca mais de uma peça.

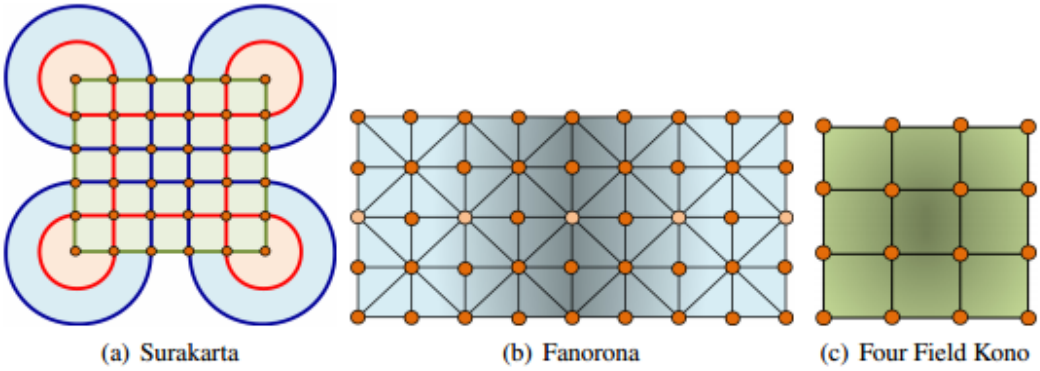


Figura 2.3: Jogos de captura especiais
Fonte: inf.ufrgs.br/lobogames/

A terceira família é a dos jogos especiais: **Surakarta, Fanorona e Four Field Kono**. No **Surakarta** as peças se movem ortogonal- e diagonalmente, uma casa por vez. Porém, para realizar uma captura, uma peça do jogador pode percorrer em linha reta qualquer número de casas livres até entrar em um dos loops do tabuleiro, fazer o loop e percorrer novamente qualquer número de casa livres até entrar em outro loop ou colidir com uma peça adversária, que será capturada. O **Fanorona** tem as mesmas regras do Alquerque, exceto por uma regra adicional de captura. Quando uma peça do jogador se afasta de uma peça adversária, essa peça adversária e as demais que estiverem atrás são capturadas. No **Four Field Kono** as peças se movimentam ortogonalmente uma casa por vez e capturam uma peça adversária ao saltar por cima de uma peça amiga e aterrizar sobre a peça adversária.

2.1 Jogos abordados

Os jogos abordados por este trabalho são os jogos da família do alquerque. Eles estão divididos didaticamente em três níveis. No primeiro nível temos jogos onde a movimentação é livre, ou seja, uma casa por vez para qualquer casa desocupada. No segundo nível as peças podem ser promovidas, podendo andar mais que uma casa por vez. No terceiro nível as peças não podem mais se mover para trás. Portanto, a cada nível adiciona-se uma regra e no final tem-se 3 níveis em ordem crescente de complexidade. As regras serão explicadas em mais detalhes adiante.

2.1.1 Primeiro nível

No primeiro nível estão os jogos: Alquerque, Felli, Serpente e Pretwa. Todos respeitam as regras de movimentação simples e captura.

Um movimento simples é um movimento realizado por uma peça que não foi promovida (quando houver promoção). A peça se desloca da sua casa para qualquer casa adjacente que estiver desocupada. A adjacência entre as casas é estabelecida pelas linhas que ligam as casas.

Uma captura é um movimento onde uma peça do jogador adversário é retirada do jogo. Para realizar uma captura é necessário que a peça percorra em linha reta (ou, no caso do Pretwa, também em semi-círculo) e no mesmo sentido duas casas. A primeira casa deve estar ocupada por uma peça adversário e a segunda deve estar vazia. É o típico movimento de captura do jogo de Damas.

As capturas são compulsórias, ou seja, se no seu turno um jogador puder capturar uma peça adversária, ele deve capturá-la. Se após a captura a peça do jogador aterrissar em uma casa onde ela pode realizar outra captura, essa outra captura deve ser realizada ainda nesse turno. Caso mais de uma peça puder realizar uma captura, o jogador pode optar qual das peças realizará a captura. Resumindo, apenas uma peça pode se mover por turno, mas essa peça pode realizar mais de uma captura.

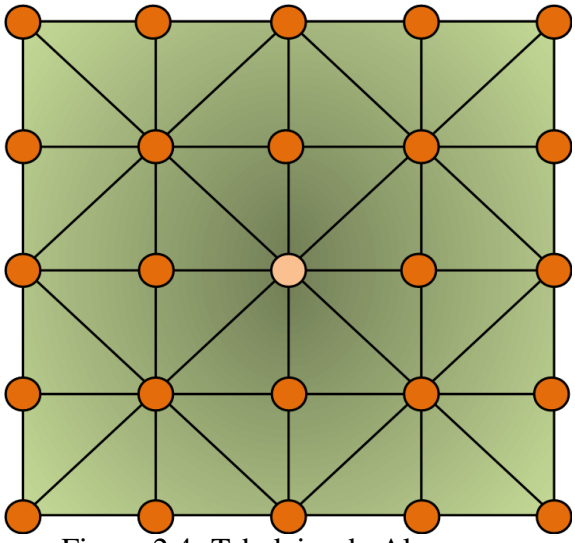


Figura 2.4: Tabuleiro do Alquerque
Fonte: inf.ufrgs.br/lobogames/

O Alquerque possui um tabuleiro cujas dimensões são 5 por 5. Cada jogador possui inicialmente 12 peças. As peças do jogador A preenchem as duas fileiras de baixo e duas as duas primeiras casas da linha central, da esquerda para direita. As peças do jogador

B espelham as peças do jogador A. O centro permanece livre. As regras válidas são a movimentação simples e a captura. Devido a disposição inicial, o jogador que inicia é obrigado a mover uma de suas peças para o centro e isso resultará em uma captura feita pelo jogador adversário no turno seguinte. A regra de promoção é opcional e para esse trabalho só será válida quando o Alquerque for apresentado no segundo nível.

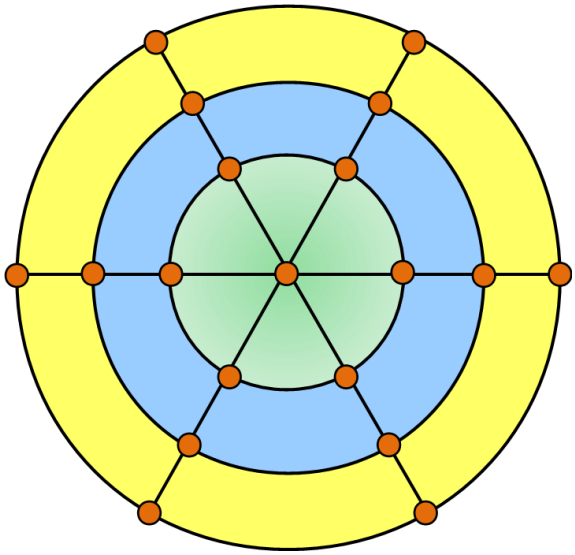


Figura 2.5: Tabuleiro do Pretwa
Fonte: inf.ufrgs.br/lobogames/

O tabuleiro do Pretwa é formado por anéis concêntricos, cada um com 6 casas. Cada jogador preenche uma metade de cada anel com suas peças e o centro fica livre. Tomando a figura como exemplo, temos 3 anéis. Portanto cada jogador terá 9 peças. Considerando uma linha vertical que divide o tabuleiro em dois, um jogador posiciona suas peças todas no lado direito e o outro posiciona todas suas peças no lado esquerdo, deixando o centro livre. Assim como no Alquerque as regras válidas são as de movimentação simples e captura. No Pretwa, as capturas circulares são possíveis: a casa de origem da peça captora, a casa da peça capturada e a casa de destino devem estar no mesmo anel e o movimento deve ser somente no sentido horário ou no sentido anti-horário.

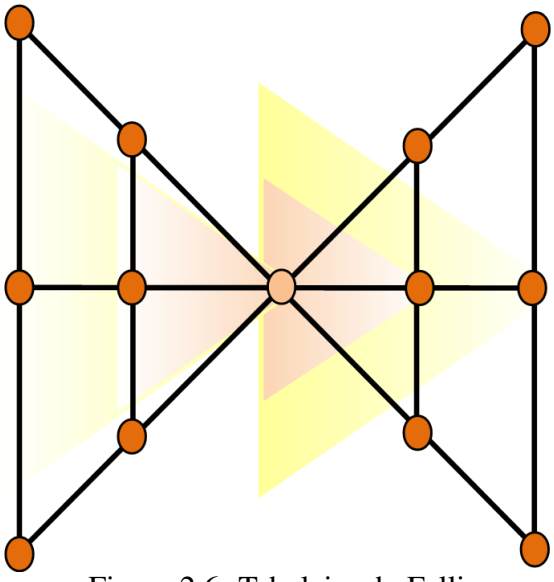


Figura 2.6: Tabuleiro do Felli
Fonte: inf.ufrgs.br/lobogames/

O tabuleiro do Felli é a união de dois triângulos como mostrado na figura. As regras válidas são a de movimentação simples e captura. Cada jogador possui inicialmente 6 peças e com elas ele preenche o triângulo esquerdo ou o direito, deixando o centro livre. A regra de promoção é opcional e para esse trabalho só será válida quando o Felli for apresentado no segundo nível.

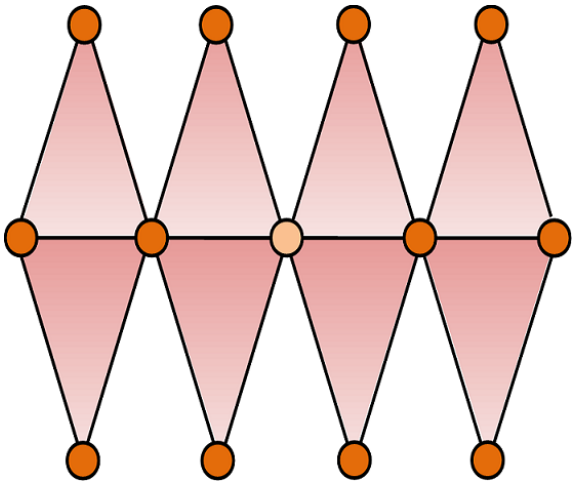


Figura 2.7: Tabuleiro do Awithlaknannai ou da Serpente
Fonte: inf.ufrgs.br/lobogames/

O tabuleiro da Serpente é composto por losangos, como visto na figura. Da mesma forma que o Pretwa e o Alquerque, as regras válidas são as de movimentação simples e captura. Um dos jogadores preenche a metade inferior do tabuleiro com as suas peças e o outro a metade superior, deixando o centro livre.

2.1.2 Segundo nível

No segundo nível estão os jogos: Alquerque e Felli, mas com a promoção de peças. As regras são as mesmas do nível um, porém adiciona-se a regra de promoção: Quando uma peça chegar no lado oposto do tabuleiro ela é promovida. Uma peça promovida pode

se deslocar mais de uma casa por vez em um único sentido, contanto que as casas pelas quais ela passar estejam livres e alinhadas.

2.1.3 Terceiro nível

No nível três, composto pelo jogo de Damas, adiciona-se a regra de restrição de movimento. Ela é apenas uma variação da movimentação simples que proíbe que as peças recuem. É típica do jogo de Damas.

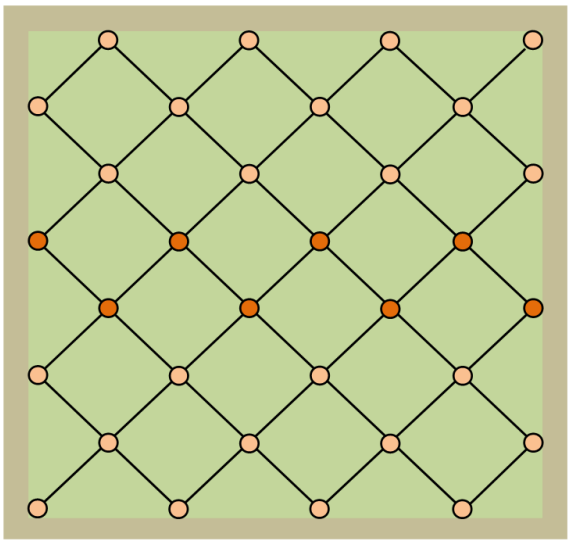


Figura 2.8: Tabuleiro das Damas
Fonte: inf.ufrgs.br/lobogames/

Provavelmente o jogo mais popular do módulo, o jogo de Damas possui várias versões, cada uma com seu conjunto de regras. Para esse trabalho, definiu-se o tabuleiro de damas com dimensões 8 por 8, com as regras de captura, movimentação simples com restrição de movimento e promoção. Cada jogador possui 12 peças. Um deles preenche as 3 linhas de baixo e o outro preenche as 3 linhas de cima, ficando a linha do centro desocupada.

2.2 Revisão Técnica

Segundo os objetivos deste trabalho, o jogador deve jogar contra o computador. Para o computador ser um adversário à altura, buscou-se algoritmos de inteligência artificial para decidir as jogadas do computador. O resultado dessa pesquisa foi o algoritmo Minimax.

Dentro da teoria de Inteligência Artificial, ambientes que forem multi agentes e competitivos são considerados jogos. Um agente é definido na literatura como qualquer coisa que realize ações. Um agente racional é um agente que age buscando o melhor resultado ou o melhor resultado esperado, quando o ambiente for incerto. A definição de ambiente multi agente vem da teoria dos jogos: qualquer ambiente onde as ações de um agente tem impacto significativo nos demais, estejam eles cooperando ou competindo, é classificado como multi agente.

Os jogos que serão implementados tem sempre apenas 2 agentes, cujas ações influenciam um ao outro. Portanto, eles são "jogos"dentro da definição de jogo dada pela teoria dos jogos. Um dos agentes será o usuário. O outro agente será o jogador computador, controlado pelo programa. Ele deve competir com o usuário, logo queremos um agente

racional.

Os jogos deste trabalho também são classificados como jogos de soma zero e com informação perfeita. Jogos de soma zero são jogos onde ou um jogador ganha e o outro perde ou ocorre um empate. Jogos com informação perfeita são jogos onde o ambiente é *completamente observável*: isso significa que os agentes podem extrair do ambiente todas as informações que são relevantes para decidir qual a sua próxima ação.

De acordo com RUSSELL (2010), o algoritmo Minimax escolhe as ações ótimas, para jogos com 2 jogadores de soma zero e com informação perfeita, com uma busca em profundidade na *árvore do jogo*. Assim, o Minimax é a escolha lógica para implementar a inteligência artificial do jogador computador.

2.2.1 Algoritmo Minimax

O algoritmo Minimax tem como entrada um estado de jogo e sua saída é uma ação ótima. O estado de jogo é definido como um conjunto das seguinte informações: quais as peças estão em jogo, onde elas estão posicionadas, a quem elas pertencem e quem é o dono do turno (ou seja, qual jogador deve jogar). Uma ação é definida como uma jogada, ou seja, um movimento ou uma captura. Uma ação ótima é definida como a melhor jogada possível para o dono do turno, o que significa aquela que o deixará mais próximo da vitória. Esse algoritmo considera que ambos os jogadores vão sempre optar por fazer a melhor jogada possível. O algoritmo é chamado de Minimax porque os jogadores serão identificados pelo algoritmo como MIN e MAX. Eles tem comportamentos opostos. Max busca maximizar a sua pontuação a cada jogada enquanto MIN vai responder às jogadas de MAX tentando minimizar a mesma pontuação. Apesar do termo "minimizar"ser empregado, MIN também está tentando vencer o jogo, pois uma pontuação negativa de MAX representa a vitória de MIN.

Para encontrar a ação ótima ou a melhor jogada, o Minimax simula todas as jogadas possíveis e as jogadas possíveis subsequentes. Na prática, ele está olhando várias jogadas a frente para determinar qual a melhor jogada deve ser realizada no turno atual. Esse processo cria a árvore do jogo. A árvore do jogo é uma estrutura que contém todos os estados possíveis decorrentes de todas as jogadas possíveis partindo do estado dado como entrada, até que se atinja um estado onde o jogo terminou.

Depois de criada a árvore do jogo, o jogador dono do turno será chamado de MAX e o jogador adversário será chamado de MIN. O algoritmo percorre a árvore até as folhas e analisa o estado final, atribuindo um valor de pontuação à esse estado, chamado de valor Minimax. Se nesse estado MAX for o vencedor, o valor Minimax nessa folha será positivo. Se MIN for o vencedor, o valor Minimax será negativo. Se houver empate, o valor será 0.

Estabelecidos os valores Minimax para todas as folhas, o algoritmo considera então os nodos um nível acima. Esses nodos representam um estado do jogo uma jogada anterior ao(s) estado(s) de fim de jogo. Olhando para esses nodos um a um, se nesse nodo for o turno de MAX, o valor Minimax desse nodo será igual ao **maior** valor Minimax de todos os seus filhos. Se nesse nodo for o turno de MIN, o valor Minimax desse nodo será igual ao **menor** valor Minimax de todos os seus filhos. O algoritmo realiza a mesma análise para o nodo um nível acima, que seria um estado duas jogadas anteriores aos estados de fim de jogo. Se no passo anterior era a vez MAX agora será a vez de MIN e vice-versa. O algoritmo chega ao fim quando for estabelecido o valor Minimax da raiz da árvore. Para encontrar a ação ótima, o Minimax procura qual dos filhos da raiz possui o mesmo valor Minimax que a raiz e quando encontrado, retorna essa ação (ou jogada). Essa jogada

representa a jogada que leva o jogo do estado atual para o estado do filho com o mesmo valor Minimax.

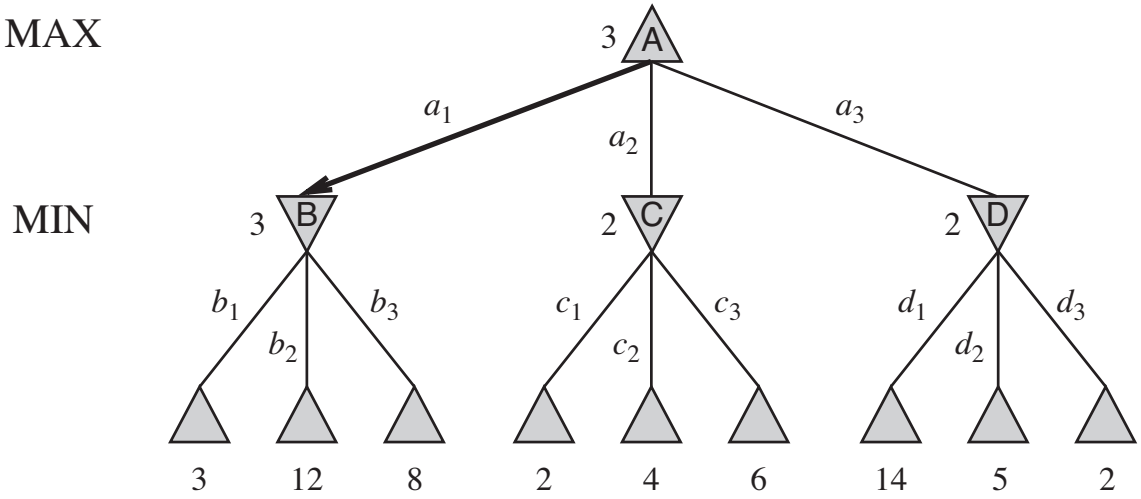


Figura 2.9: Árvore de um jogo simples

A figura 2.9 será usada para exemplificar o algoritmo Minimax. O nodo A é a raiz. Os nodos B, C e D são os seus primeiros filhos. Os demais nodos são as folhas. Os números abaixo do filhos representam os valores Minimax deles. As arestas representam as jogadas. MAX possui três possibilidades de jogadas: a_1, a_2 e a_3. O Minimax simula o resultado dessas jogadas, o que gera os nodos B, C e D. Como esses novos nodos não são nodos finais, ele vai uma jogada adiante e simula o resultado de todas as jogada possíveis de MIN nesses 3 estados. Os filhos de B, C e D são estados finais, e portanto recebem uma pontuação, que será o seu valor Minimax. Percorrendo a árvore de baixo para cima, começamos pelo nodo B. Em B, é a vez de MIN. MIN possui 3 alternativas: b_1, b_2 e b_3. MIN quer minimizar o valor Minimax e o algoritmo supõe que MIN fará a melhor jogada disponível, logo, caso MAX fizesse a jogada a_1, MIN faria a jogada b_1. Assim, atribui-se ao nodo B o valor Minimax 3. Isso significa que se o jogo chegar no estado do nodo B e MIN realizar a melhor jogada, a pontuação de MAX será 3. Analisando o nodo C, novamente é a vez de MIN e novamente ele vai minimizar o valor Minimax. Caso MAX faça a jogada a_2, MIN fará a jogada c_1 e MAX terminará o jogo com 2 pontos. Logo, o valor Minimax de C é 2. O mesmo procedimento ocorreu no nodo D, e o seu valor Minimax é 2. Assim, entre as três opções de MAX, a que maximiza o valor Minimax é a ação a_1, que será respondida pela ação b_1 de MIN. As jogadas a_1 e a_2 seriam respondidas por MIN respectivamente pelas jogadas c_1 e d_3 o que em ambos os casos resultaria numa pontuação 2.

Como base para uma explicação mais formal, é necessário definir os seguinte tipos de dados e funções:

- *S*: estado inicial, especifica o início de jogo e a posição das peças.
- *s*: estado qualquer durante o jogo.
- *p*: um jogador.
- PLAYER(*s*): retorna qual do jogadores tem o direito de se mover.
- ACTIONS(*s*): retorna quais as jogadas possíveis no estado *s*.
- RESULT(*s,a*): retorna o estado seguinte ao estado *s*, considerando a ação *a*.
- TERMINAL-TEST(s): retorna verdadeiro se o jogo tiver chegado ao fim no estado

s.

- UTILITY(*s,p*): A função de utilidade. Retorna um valor numérico para um jogo que termine no estado *s* para o jogador *p*. Por exemplo, 1 se *p* tiver vencido, -1 se ele tiver perdido ou 0 se ocorreu um empate.

A criação da árvore é realizada pelo seguinte trecho de código recursivo:

```
function MINIMAX-DECISION(state) returns an action
        return  max_MinimaxValue_action(
            MIN-VALUE(RESULT(state, action)        )
```

Figura 2.10: Pseudocódgio para a função de decisão do Minimax

```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v = -infinity
    for each action in ACTIONS(state) do
        v = MAX(v, MIN-VALUE(RESULT(state,action)))
    return v
```

Figura 2.11: Pseudocódgio para a função Max-value

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v = +infinity
    for each action in ACTIONS(state) do
        v = MIN(v, MAX-VALUE(RESULT(state,action)))
    return v
```

Figura 2.12: Pseudocódgio para a função Min-value

O Minimax realiza uma busca em profundidade por toda a árvore do jogo. Se a profundidade máxima da árvore é m e existem aproximadamente b jogadas possíveis em cada nodo, a complexidade do Minimax é $O(m^b)$. Em RUSSELL (2010), SHANNON (1950) e WINSTON (2010) é demonstrado que para o jogo de xadrez essa complexidade torna o uso do Minimax impraticável. Os jogos desse trabalho não são tão complexos quanto o xadrez, mas são complexos o suficiente para tornar a árvore completa incalculável.

2.2.2 *Alpha-Beta Prunning*

A árvore do jogo cresce exponencialmente e a busca em profundidade é extremamente custosa. Não é possível eliminar o expoente da complexidade, mas no melhor caso é possível reduzí-lo pela metade com a técnica de *alpha-beta prunning*, ou poda alpha-beta. Essa técnica reduz a quantidade de subárvores pesquisadas, pois elimina subárvores que não influenciam no resultado do Minimax. A poda alpha-beta pode ser aplicada a

árvores de qualquer tamanho e a resposta do Minimax com essa técnica é exatamente a mesma que a resposta do Minimax. A poda alpha-beta é portanto uma otimização do Minimax.

O Minimax assume que tanto MIN quanto MAX sempre fazem as jogadas ótimas. Sabendo o comportamento dos jogadores, a poda alpha-beta pode desconsiderar ramos que não seriam escolhidos por nenhum deles. Por exemplo, considere o nodo *n* em algum lugar da árvore, tal que o jogador *P* tenha a possibilidade de realizar uma jogada e chegar em *n*. Se *P* tiver uma jogada melhor *m* ou em um nodo pai de *n* ou em qualquer ponto acima na árvore, *P* jamais escolherá a ação que leva a *n*. Logo, toda a subárvore que começa em *n* pode ser desconsiderada, ou *podada*. O nome alpha-beta vem dos parâmetros α e β que são utilizados durante a busca. α contém o valor da melhor escolha para MAX (ou seja, o maior valor) encontrada até chegarmos em *n* e β contém o valor da melhor escolha para MIN (ou seja, o menor valor). Esse valores são atualizado conforme a busca prossegue e tão logo o valor do nodo atual for sabido e avaliado pior do que α ou β para MAX e MIN, respectivamente, esse nodo será podado.

O desempenho da poda alpha-beta depende da ordem em que os ramos são avaliados. Se, por exemplo, a busca for iniciada da esquerda para a direita e as jogadas ótimas estiverem todas nos ramos da extrema direita, nenhuma subárvore será podada. Se fosse possível definir uma função que retorna a ordem dos ramos que devem ser percorridos, a complexidade poderia ser efetivamente reduzida para $O(m^{\frac{b}{2}}))$. Obviamente, tal função não existe, pois ela poderia ser usada para jogar um jogo perfeito. Segundo RUSSELL (2010), para o xadrez, uma função de ordenação simples que escolhe os ramos a serem avaliados de acordo com a ação, dando prioridade para captura, ameaça, movimentos para frente e movimentos para trás, é possível chegar próximo à uma fração do caso ótimo de $O(m^{\frac{b}{2}}))$. RUSSELL (2010) cita outras técnicas para selecionar os ramos que devem ser considerados, como a heurística dos *killer moves* e a tabela de transposição.

2.2.3 Função de Custo

A poda alpha-beta reduz a quantidade de nodos que o Minimax deve pesquisar por no máximo 2. Entretanto, o Minimax ainda precisa gerar toda a árvore do jogo e pesquisar da raiz até as folhas o que ainda é impraticável. A escolha da jogada não deve demorar, no pior dos casos, mais que alguns segundos. Mesmo que fosse possível aplicar o Minimax, não é desejado que o jogador computador demore muito tempo para realizar sua jogada.

Claude Shannon propôs em SHANNON (1950) que os programas deveriam parar a busca antes de atingir as folhas e aplicar a heurística da *evaluation function*, ou função de custo, para tornar nodos não terminais em folhas. Em relação ao algoritmo do Minimax apresentado na revisão técnica, a proposta de Shannon é substituir a função TERMINAL-TEST por uma função de corte CUTOFF e a função UTILITY por uma função de custo EVALUATION. A função de CUTOFF determina quando o algoritmo deve parar de percorrer a árvore com base na profundidade ou no tempo. A função EVALUATION deve examinar o estado atual do jogo e retornar uma estimativa do valor da função UTILITY naquele estado.

A função de custo tem uma enorme influência na performance do jogador computador. Portanto, é preciso projetar uma boa função de custo, que se aproxime ao máximo dos resultados de UTILITY. Tal função deve: avaliar os estados terminais da mesma forma que UTILITY avaliaria (vitória melhor que empate, empate melhor que derrota), a computação não pode ser demorada (o propósito é justamente tornar o algoritmo mais rápido) e por fim, para os estados não terminais, o resultado deve ser fortemente correlacionado

com as verdadeiras chances de vitória RUSSELL (2010). A heurística da função de custo por ser imperfeita, introduz incerteza para reduzir o tempo de processamento. As imperfeições ou problemas da função de custo como a quiescência de um estado e o efeito horizonte são discutidos em RUSSELL (2010).

Em RUSSELL (2010) e SHANNON (1950) funções de custo para o xadrez são discutidas. Conclui-se que a vantagem material é o fator mais importante de qualquer função de custo e combinado com uma árvore de jogo profunda o bastante, o algoritmo já é capaz de jogar um jogo de alto nível WINSTON (2010). Vantagem material é diferença entre o número de peças de cada jogador.

Não é necessário nem desejado que o jogador computador seja invencível. Agora que o funcionamento do Minimax e suas melhorias foi apresentado, formas de torná-lo progressivamente menos eficiente podem ser analisadas. Entre as formas de controle de dificuldade, serão consideradas: o limite da profundidade de busca e a escolha não ótima de uma ação.

A profundidade da busca é controlada pela função CUTOFF e pode ser passada como um dos parâmetros da função. Assim, a eficiência do Minimax pode ser controlada dinamicamente durante a partida, se necessário. A limitação de profundidade na prática limita quantas jogadas a frente o jogador computador está considerando portanto é claro que alterando a profundidade de busca estamos alterando as capacidades do computador de escolher a melhor jogada.

Evitar escolher a solução ótima é uma outra alternativa. O algoritmo irá encontrar a ação ótima (dentro das limitações impostas pela profundidade de busca, pelo tempo de processamento e pela qualidade da função de custo) mas não irá escolher esta ação. Dessa forma, em todas as suas jogadas o computador comete um pequeno erro do qual o jogador humano pode tirar proveito.

Uma combinação das técnicas acima mencionadas também pode ser aplicada. Limitar somente a profundidade de busca pode ser exageradamente prejudicial para a performance do Minimax.

3 IMPLEMENTAÇÃO

3.1 Decisões de projeto

Existem várias plataformas para os quais um programa pode ser desenvolvido. As particularidades de cada plataforma determinam como os programas devem ser implementados, especialmente na parte de interação com o usuário. Portanto, é necessário definir qual a plataforma alvo de desenvolvimento previamente. Opções consideradas como plataforma alvo foram: Web, ferramentas de desenvolvimento multi-plataforma, como por exemplo Unity3D e Game Salad, Windows, Linux, Anroid e iOs.

Escolheu-se trabalhar com dispositivos móveis devido a sua portabilidade e popularidade. O jogo pode ser jogado em qualquer lugar por qualquer um que tenha um smartphone ou um tablet. A tela sensível a toque desses dispositivos também foi outro fator importante nessa escolha, pois é uma forma intuitiva de interação do usuário com o programa. Outra vantagem do desenvolvimento de software para esses dispositivos é a existência de lojas de aplicativos. Essas lojas facilitam a distribuição dos aplicativos e dão segurança para os usuários de que os produtos não são maliciosos.

Os sistemas operacionais Windows e Linux apesar de populares, não possuem as vantagens que os dispositivos móveis oferecem de mobilidade e facilidade de interação com o usuário. A distribuição do software tem menos credibilidade do que pela loja de aplicativos e os jogos ficariam limitados a desktops e notebooks. Valorizou-se bastante a mobilidade dos smartphones e tablets e por isso esses sistemas não foram escolhidos.

O desenvolvimento para Web foi descartado por adicionar uma camada adicional de complexidade e por dificultar um futuro empreendimento. A complexidade adicional seria a necessidade um servidor para hospedar o serviço. No caso de um futuro empreendimento, para obter um retorno financeiro seria necessário colocar anúncios no site onde os jogos estivessem hospedados. Conclui-se que isso seria inapropriado e que os anúncios além de poluírem visualmente podem distrair os jogadores.

Foram caracterizadas como ferramentas de desenvolvimento multi-plataforma frameworks e bibliotecas que podem gerar software para várias plataforma diferentes. Existem dezenas de opções, o que torna extremamente difícil escolher uma delas. Algumas mais populares foram levadas em consideração como a Unity3D e o Game Salad. De modo geral, a grande vantagem dessas ferramentas, além da distribuição multi-plataforma, é a facilidade para lidar com animações. As desvantagens são a adição de uma camada de complexidade, que é a própria ferramenta, e a imposição de uma metodologia de desenvolvimento. Como a parte gráfica do projeto é simples, conclui-se que as desvantagens são maiores que a vantagem de distribuição multi-plataforma.

Entre os sistemas para os dispositivos móveis, o sistema Android é certamente o mais popular, e esse foi um dos critérios de desempate quando comparado com o iOs. Além da

sua popularidade, desenvolver para Android é inicialmente mais simples. As ferramentas são gratuitas, assim como a publicação dos aplicativos no mercado.

O Android é um sistema operacional móvel baseado no Linux e desenvolvido pela empresa de tecnologia Google. Ele foi projetado tendo em vista dispositivos móveis com tela sensível ao toque, sendo essa a principal forma de interação entre o usuário e o sistema. Ele pode ser encontrado predominantemente em Smartphones e Tablets. Outros aparelhos eletrônicos possuem versões customizadas. Por exemplo, existe o Android Wear, feito para relógios, o Android Auto, feito para carros e o Android TV, feito para televisões. O sistema também pode ser encontrado em câmeras, videogames e até alguns computadores pessoais. O código fonte desse sistema operacional é disponibilizado pela Google sob licenças de código aberto. Como o foco é smartphones e tablets, o software será desenvolvido para o Android convencional. Entretanto, é relativamente fácil adaptar o programa para as outras versões do sistema. Muitas vezes, basta recompilar o código para a versão do Android desejada. Dessa forma, em um futuro próximo e sem grande esforço será possível que o programa desenvolvido rode também nos relógios Android, por exemplo.

3.2 Ambiente de Desenvolvimento e Linguagem de Programação

Escolheu-se a linguagem de programação Java. Primeiro porque Java é uma das linguagens mais comuns para se trabalhar com o Android e as ferramentas de desenvolvimento dão suporte escrita de código nessa linguagem. O segundo motivo é a portabilidade. Os programas escritos em Java rodam em qualquer plataforma que tenha uma Máquina Virtual Java. Além disso, a linguagem Java é bastante popular e esta bem difundida no academia e na indústria. A linguagem Java é uma linguagem orientada a objetos. Orientação a objetos é uma forma de modelar os objeto da vida real em estruturas de dados e este programa foi desenvolvido com esse paradigma de modelagem.

O jogo foi desenvolvido na ferramenta Android Studio, feita pela própria Google. A ferramenta ainda está na fase de testes, mas já foi anunciado que ela irá substituir a IDE (Integrated Development Environment) Eclipse, que até então era a forma padrão de se desenvolver para Android. A IDE Eclipse ainda recebe suporte e é consideravelmente mais popular que o Android Studio. Mesmo assim, escolheu-se trabalhar com o Android Studio porque valorizou-se mais a adaptação à nova ferramenta do que as facilidades da ferramenta antiga.

3.3 Codificando o Jogo

O foco da implementação foi aproveitar ao máximo as semelhanças entre os jogos para escrever código reutilizável. As peças e os tabuleiros foram modelados de forma que pudessem ser usados por qualquer um dos jogos de captura que este trabalho se propôs a implementar. Da mesma forma, projetou-se uma única implementação do algoritmo Minimax que fosse capaz de rodar para todos os jogos. Buscou-se inspiração no modelo de desenvolvimento Model-View-Controller para manter tanto quanto possível separadas as partes de controle, de visualização e lógica. Assim o aplicativo foi divido em 3 partes: a lógica, a gráfica e a de controle. A parte lógica providencia as funções e as estruturas de dados necessárias para o jogo funcionar. A parte gráfica está encarregada de exibir as peças e o tabuleiro na tela. A parte de controle processa os toques do usuário e gerencia o laço do jogo.

As subseções a seguir apresentarão para cada uma das 3 partes a filosofia de desenvolvimento, as classes utilizadas e no final discutirá as alternativas de implementação para justificar a escolha feita.

3.3.1 Parte Lógica

A filosofia de desenvolvimento foi escrever um código genérico para jogos de captura. Assim ele poderia ser utilizado para implementar todos os jogos propostos por este trabalho. Por ser a função mais custosa utilizada, o Minimax influenciou fortemente na modelagem dos dados pois elas tem impacto direto no desempenho desse algoritmo.

Observou-se que as regras são as mesmas para todos os jogos, alguns tendo apenas regras adicionais. No caso específico da captura, notou-se que em uma captura uma peça deve sempre se movimentar pelo mesmo **tipo de aresta**. Portanto, a maior diferença entre os jogos está no tabuleiro. Assim, o primeiro passo foi encontrar uma forma unificada de representar os tabuleiros. O segundo passo foi definir como seria a representação das peças. Com essas estruturas de dados definidas, pode-se implementar as funções que estabelecem os movimentos e as capturas dos jogos, de forma que a mesma função que movimenta as peças as peças no Alquerque, movimente as peças no Pretwa e nos demais jogos. Para o algoritmo Minimax é necessário que haja uma forma de representar o estado atual do jogo e que existam funções que retornem as jogadas possíveis em um dado turno. Isso também foi considerado e o Minimax implementado é válido para todos os jogos.

Outra observação importante é que a forma encontrada para padronizar os tabuleiros foi representá-los como grafos. Um grafo é uma estrutura composta por vértices e arestas. A teoria dos grafos é um ramo da matemática que usa grafos para estudar as relações dos objetos de um certo conjunto. Os objetos são representados pelos vértices a as relações são representadas pelas arestas. Assim, cada casa do tabuleiro passa a ser um vértice e a adjacência entre as casas, que é a linha que liga as casas, passa a ser uma aresta.

A modelagem dos elementos em classes será explicada e acompanhada de uma descrição e do trecho de código correspondente. A descrição das classes tem o seguinte formato: a relação entre a estrutura de dados e o objeto que ela representa, a relação entre os campos da estrutura e as características do objeto na ordem em que os campos foram declarados na função e algumas observações em relação as funções quando necessário.

A classe Vertex modela as casas do tabuleiro. Os campos informam: identificador e as ligações entre essa casa e suas casas vizinhas, que são as arestas entre os vértices do grafo, chamadas de edges. Para facilitar a geração dos tabuleiros, tem-se a função addEdges, que permite que as arestas sejam adicionadas a qualquer momento e não apenas na instanciação do objeto. Um tabuleiro é uma lista de objetos da classe Vertex.

```
public class Vertex {}
   private int id;
   private Edge[] edges;

   public Vertex(Edge[] edges, int id) {...}

   public int getId() {...}

   public Edge[] getEdges() {...}

   public void addEdges(Edge[] e) {...}
```

```
    private void createEdges(int howMany) {...}

}
```

A classe Edge modela as ligações entre as casas, que na representação dos tabuleiros por grafos são as arestas entre os vértices. Os campos informam: o tipo de aresta e qual o vértice destino.

```
public class Edge {
    private EdgeType type;
    private Vertex v;

    public Edge(EdgeType type, Vertex v) {...}

    public EdgeType getType() {...}

    public Vertex getV() {...}
}
```

A enumeração EdgeType representa os diferentes **tipos de arestas**.

```
public enum EdgeType {
    VERTICAL, HORIZONTAL, DIAGONAL_R, DIAGONAL_L, CIRCULAR,
        RADIAL
}
```

A classe Stone modela as peças. Os campos informam: identificador, dono, se está promovida e a posição no tabuleiro. As funções são apenas Getters e Setters. Esta classe possui dois construtores. O primeiro é utilizado na criação da peça e determina seus atributos. O segundo e usado para criar uma cópia de uma peça. Durante uma partida, todas as peças do jogo estão armazenadas em uma lista.

```
public class Stone {
    private int id;
    private Owner owner;
    private boolean promoted;
    private Vertex v;

    public Stone(int id, Owner owner, Vertex v) {...}

    public Stone(Stone s) {...}

    public int getId() {...}

    public Owner getOwner() {...}

    public boolean isPromoted() {...}

    public void setPromoted(boolean promoted) {...}

    public Vertex getV() {...}
```

```
    public void setV(Vertex v) {...}
}
```

A enumeração Owner representa os jogadores. Mais especificamente é usada para determinar quem é o dono de uma peça ou do turno.

```
public enum Owner {
    PLAYER1, PLAYER2, CPU, NONE
}
```

A classe State modela um estado de jogo. Os campos informam: todas as peças existentes nesse estado e o turno desse estado. A função nextTurn() incrementa o campo turn e foi criada para legibilidade do código.

```
public class State {
    public List<Stone> stones;
    public int turn;

    public State(List<Stone> stones, int turn) {...}

    public State(State original) {...}

    public void nextTurn() {...}
}
```

A classe MinimaxNode representa um nodo da árvore do jogo e é criado e utilizado apenas pelo algoritmo Minimax. Os campos informam: o estado, o movimento que deu origem a esse estado, o nodo pai, os nodos filhos, o valor de Minimax, o parâmetro alpha e o parâmetro beta.

```
public class MinimaxNode
    public State state;
    public List<Integer> move;
    public MinimaxNode father;
    public List<MinimaxNode> children;
    public int evaluationValue = 0;
    public int alpha;
    public int beta;

    public MinimaxNode(State state,
                   List<Integer> move, MinimaxNode father,
                      List<MinimaxNode> children,
                   int evaluationValue, int alpha, int beta) {...}
}
```

Uma função necessária para rodar o algoritmo Minimax é o cálculo de todas as jogadas possíveis. Sabe-se que o Minimax irá gerar uma árvore do jogo, composta por objetos da classe MinimaxNode. Cada nodo contém um estado do jogo. Sabe-se também que essa árvore cresce exponencialmente. Portanto, buscou-se uma forma de representação para os grafos que facilitasse o cálculo das jogadas disponíveis, e uma forma de modelar os estados que economizasse espaço.

Foram analisadas três formas de se representar um grafo: listas de adjacências, matriz

de adjacências e matriz de incidência. A matriz de incidência é uma variação da matriz de adjacência, que contempla os casos onde os grafos são orientados. Esta opção foi descartada pois os grafos dos tabuleiros não precisam ser grafos orientados. Assim, restam apenas a representação por matriz de adjacência e por lista de adjacências. Considerou-se as operações mais frequentes realizadas: retornar todos os vértices que são adjacentes a um vértice e verificar se dois vértices são adjacentes. Uma verificação de adjacência entre dois vértices tem complexidade $O(1)$ na matriz de adjacência e $O(|V|)$ na lista. Verificar quais são todos os vértices adjacentes a um vértice é uma simples leitura na lista de adjacências, enquanto que na matriz de adjacências essa operação tem complexidade $O(|V|)$. Entre as duas operações, a mais importante é a de verificar todos os vértices adjacentes a um vértice, pois ela é usada no cálculo das jogadas possíveis e esse cálculo é realizado para cada nodo da árvore. Logo, por questões de desempenho, optou-se pela lista de adjacências.

Os estados foram modelados como uma lista de peças e um contador de turnos. A única forma dessa modelagem conter as informações de posicionamento das peças é se as peças indicarem qual o vértice que elas ocupam. Em outra alternativa de modelagem, os vértices do tabuleiro teriam um campo indicando qual peça está posicionada ali. Consequentemente, o estado seria composto pelo tabuleiro apontando para as peças e pelo contador de turnos. Essa outra alternativa, porém, vai utilizar muito mais memória. A lista de vértices que compõe o tabuleiro é maior que a lista de peças e ss vértices são maiores que as peças, pois além das informações individuais, eles tem referências para todos os vértices adjacentes. Não há nenhum benefício em repetir essas referências em todos os estados e por isso essa alternativa foi descartada.

Para agrupar todos os elementos da parte lógica e fazer o jogo fluir, existe a classe Capture. Abaixo, estão apenas os campos dessa classe. A variável realState contém o estado atual do jogo, que por usa vez contém a lista de peças e o contador de turno. A variável board é um vetor de vértices que representa o tabuleiro. Todos os campos são inicializados no momento de instanciação conforme o jogo escolhido. O campo movesHistory guarda uma lista dos movimentos realizados para servir de registro do jogo. Os demais campos serão abordados conforme sua necessidade na descrição das funções da classe Capture.

```
public class Capture {

   //game mechanics:
   public State realState;
   private Vertex[] board;
   private List<Integer> promotionVertexesP1,
      promotionVertexesP2;
   private int treeDepth;
   private Owner currentMAX;
   private Owner winner;
   //other
   public List<String> movesHistory;
   //game definitions:
   private String gameName;
   private int size;
   private int capturesPerPlayerLimit;
   private int turnLimit;
   private boolean compulsoryCapture;
```

```
    private boolean multipleCaptures;
    private boolean movementRestriction;
    private boolean promotionOn;
}
```

Três funções são fundamentais para a execução das jogadas. São elas: a função que retorna todos os movimentos válidos de uma peça, a função que retorna todas as capturas válidas para uma peça e a função que realiza os movimentos.

```
private List<List> capturesCheck(State state, Stone stone) {...}
private List<List> simpleMovesCheck(State state, Stone stone)
    {...}
public String makeTheMove(State state, List<Integer> move) {...}
```

A função que retorna todos os movimentos possíveis foi batizada de simpleMovesCheck. Ela tem como entrada um estado do jogo e uma peça. Sua saída é uma lista de todos os movimentos possíveis para aquela peça naquele estado. Seu funcionamento é relativamente simples. Primeiro ela busca todos os vértices adjacentes do vértice ocupado pela peça e chama-os de vizinhos. Todos os vizinhos que não estiverem ocupados, são considerados movimentos válidos e adicionados a lista de movimentos possíveis para aquela peça.

A função que retorna todas as capturas possíveis é chamada de capturesCheck. Assim como a simpleMovesCheck, ela tem como entrada um estado do jogo e uma peça e tem como saída uma lista com todas as possíveis capturas para aquela peça naquele estado. Primeiro ela busca todos os vértices adjacentes do vértice ocupado pela peça e considera apenas aqueles vizinhos ocupados por peças do adversário. Para cada vértice vizinho ocupado por uma peça adversária ocorre o seguinte procedimento: a função armazena qual o tipo de aresta que conecta àquele vértice. A partir do vértice vizinho ocupado, a função verifica se existe alguma aresta do mesmo tipo. Se essa aresta existir e levar a um vértice desocupado, é um movimento de captura válido e ele é adicionado a lista de capturas. Além disso é preciso verificar se mais capturas por essa peça são possíveis nesse mesmo turno. Para tal, a função simula a captura que foi recém encontrada e chama a si mesma novamente. Como dito anteriormente, esse processo é feito para cada vértice vizinho ocupado pelo adversário.

A função que realiza os movimentos de fato é a função makeTheMove. Ela recebe como entrada uma jogada e um estado. O resultado de sua aplicação é a realização da jogada no estado. A sua saída é uma String, contendo um registro da jogada. As duas funções anteriores garantem que a jogada da entrada é um movimento válido, portanto a única tarefa da função makeTheMove é redefinir as posições das peças, remover as peças capturadas e gerar o registro da jogada.

Para calcular o movimento das peças promovidas, utiliza-se a função promotedSimpleMove que é pequena modificação na função simpleMovesCheck. Para cada vértice vizinho desocupado, ela entra em um laço e calcula o quanto mais a peça promovida poderia se deslocar seguindo o mesmo tipo de aresta.

A restrição de movimentos é ativada apenas para o jogo de damas. Devido a forma como o tabuleiro foi construído, quando a função simpleMovesCheck for chamada, compara-se o identificador do vértice destino e do vértice de origem. Se for o turno do jogador 1, o destino deve ter identificador maior que a origem. Se for o turno do jogador 2, o destino deve ter identificador menor que a origem.

3.3.1.1 Implementação do algoritmo Minimax

O algoritmo Minimax foi implementado seguindo o modelo mostrado na revisão técnica.

Para gerar a árvore do jogo, o Minimax precisa de uma função que retorne todos as jogadas possíveis em um turno. Para tal, foi escrita a função actions. Essa função encapsula as funções descritas anteriormente: simpleMovesCheck e capturesCheck. Ela recebe como entrada um estado e roda a função capturesCheck para todas as peças desse estado. Caso nenhuma captura seja possível, ela executa a função simpleMovesCheck para todas as peças do estado.

O algoritmo Minimax foi implementado com a otimizaçã alpha-beta e por isso a função que o implementa é chamada de minimaxDecisionAB. Ela tem como entrada um estado e retorna uma jogada. Essa função inicializa a raiz da árvore e define o jogador dono do turno como Max. A construção da árvore começa com a chamada da função maxValue.

```
public List<Integer> minimaxDecisionAB(State state) {
      List<Integer> move;
      MinimaxNode root = new MinimaxNode(state,
            null, null, new ArrayList<MinimaxNode>(), -1, -1,
               -1);

      int cutOff = 1;
      this.currentMAX = turnOf(state);
      int v = maxValue(root, cutOff, -9999, 9999);
      move = findTheMove(root, v);
      return move;
   }
```

A função maxValue primeiro verifica se o estado atual é um estado final ou se a árvore atingiu uma altura limite. Se alguma dessas condições for verdadeira, ela chama a evaluationFunction, armazena seu valor e retorna esse mesmo valor, finalizando a recursão. Caso não seja um estado final, ela chama a função actions e recebe todas as jogadas possíveis de Max naquele turno. Para cada jogada, ela copia o estado atual, aplica a jogada nesse estado através da função makeTheMove, cria um nodo na árvore e chama a função minValue. O campo evalutationValue representa o valor Minimax daquele nodo. Quando esta função receber o resultado da avaliação dos seus filhos ela atualizará o valor Minimax somente se o valor recebido for maior que o valor armazenado. Ele é inicializado em -9999 porque a função de custo implementada jamais irá retornar um valor tão baixo. O parâmetro beta é analisado e se for constatado que a poda pode ser efetuada, finaliza-se a recursão para esse ramo. Caso contrário, atualiza-se o parâmetro alpha.

```
private int maxValue(MinimaxNode node, int cutOff, int alpha,
   int beta) {
      if (isTheGameOver(node.state) || cutOff >= treeDepth) {
         node.evaluationValue = evaluationFunction(node.state,
            currentMAX);
         return node.evaluationValue;
      }
      node.evaluationValue = -9999;
```

```
      List<List> possibleMoves = actions(node.state);

      for (List<Integer> m : possibleMoves) {

         MinimaxNode child = new MinimaxNode(node.state,
               m, node, null, -1, -1, -1);
         makeTheMove(child.state, m);
         child.state.nextTurn();
         if (node.children == null) node.children = new
            ArrayList<MinimaxNode>();
         node.children.add(child);
         int resultMinValue = minValue(child, cutOff + 1,
            alpha, beta);
         if (node.evaluationValue < resultMinValue)
            node.evaluationValue = resultMinValue;
         if (node.evaluationValue >= beta) return
            node.evaluationValue;
         else if (alpha < node.evaluationValue) alpha =
            node.evaluationValue;
      }
      return node.evaluationValue;
   }
```

A função minValue é análoga a função maxValue. Algumas linhas de código foram omitidas no trecho abaixo, pois são idênticas a função maxValue. As diferenças são: o evaluationValue do nodo é inicializado com +9999, a função chama maxValue ao invés de minValue, tenta manter o maior valor Minimax ao invés do menor, o parâmetro analisado é o alpha e o atualizado é o beta.

```
private int minValue(MinimaxNode node, int cutOff, int alpha,
   int beta) {

      if (isTheGameOver(node.state) || cutOff >= treeDepth)
         {...}

      node.evaluationValue = 9999;

      List<List> possibleMoves = actions(node.state);

      for (List<Integer> m : possibleMoves) {
         MinimaxNode child = new MinimaxNode(...);
         makeTheMove(child.state, m);
         if (node.children == null) node.children = new
            ArrayList<MinimaxNode>();
         node.children.add(child);
         int resultMaxValue = maxValue(child, cutOff + 1,
            alpha, beta);
         if (node.evaluationValue > resultMaxValue)
            node.evaluationValue = resultMaxValue;
         if (node.evaluationValue <= alpha) return
            node.evaluationValue;
```

```
            else if (beta < node.evaluationValue) beta =
               node.evaluationValue;
         }
         return node.evaluationValue;
      }
```

No final de cada turno é chamada a função isTheGameOver, que recebe um estado e tem como saída um valor verdadeiro ou falso. Essa função avalia o estado do jogo e determina se ele é final ou não. Existem 3 maneiras de se atingir um estado final: o limite de capturas for atingido, o limite de turnos for atingido ou se algum dos jogadores não possuir mais nenhum movimento válido. Essa função também determina o vencedor. O vencedor é o jogador que tiver mais peças em um estado final, se esse estado final for atingido pelo limite de capturas ou de turnos. Caso o estado final seja atingido devido a falta de movimentos de um dos jogadores, o jogador sem movimentos é o perdedor e seu adversário é o vencedor.

A função de custo é chamada de evaluationFunction. Durante o desenvolvimento, 3 diferentes implementações foram testadas. A primeira era apenas uma contagem de peças para o jogador considerado Max. A segunda era a diferença entre as peças de Max e as peças de Min elevado ao quadrado. A terceira leva em consideração a diferença entre a quantidade de peças, a proximidade do final do jogo em relação ao limite de capturas e a contagem total de peças.

```
//segunda funcao de custo implementada.
private int evaluationFunction(State state, Owner player) {
      int dif = 0;
      int stonesP1 = countStones(state, Owner.PLAYER1);
      int stonesP2 = countStones(state, Owner.PLAYER2);
      if (player == Owner.PLAYER1) {
         dif = stonesP1 - stonesP2;
      } else {
         dif = stonesP2 - stonesP1;
      }

      if (dif > 0) dif = dif * dif;
      else dif = -(dif * dif);

      return dif;
   }
```

Existem 3 níveis de dificuldade: fácil, médio e difícil. No nível fácil, o computador faz uma jogada aleatória dentro das jogadas possíveis. No nível médio, a cada duas jogadas o computador escolhe uma jogada não ótima. No nível difícil, o jogador computador joga utiliza a função de custo do trecho de código acima, que é o quadrado da diferença entre o número de peças do jogadores.

A função translateHumanMove recebe dados gerados pela parte de controle que representam um movimento e com esses dados seleciona a jogada adequada dentro das jogadas possíveis naquele estado.

```
public List<Integer> translateHumanMove(List<Integer>
   humanMove) {
      List<List> possibleMoves = actions(realState);
```

```
        for (List<Integer> m : possibleMoves) {
            if (m.get(0) == humanMove.get(0)) {
                if (m.get(m.size() - 1) ==
                    humanMove.get(humanMove.size() - 1)) {
                    return m;
                }
            }
        }
        return null;
    }
```

Para cada jogo implementado existe uma função que gera o tabuleiro e a configuração inicial das peças. Essa função é chamada no momento de instanciação do jogo. Os vértices dos tabuleiros foram numerados e foi possível estabelecer relações entre os números e as arestas. Essas relações são usadas por essas funções para gerar os tabuleiros. A vantagem desse método é que tabuleiros de diferentes tamanhos podem ser criados passando apenas um parâmetro diferente para a função.

3.3.2 Parte Gráfica

A ideia central ao desenvolver a parte gráfica foi mantê-la completamente transparente para a parte lógica. Para desenhar os tabuleiros e as peças na tela foram utilizadas classes dos pacotes android.graphics, android.view e android.bitmap. As mais relevantes delas são: SurfaceView e Canvas.

Para manter a separação entre a parte gráfica e a parte lógica criou-se mais duas classes: DrawableVertex e DrawableStone. Essas classes possuem campos para referenciar quais elementos da parte lógica elas representam e campos com as coordenadas que elas ocupam na tela. Além dos Getters e Setters, elas possuem a função handleActionDown que detecta se as coordenadas do toque na tela batem com as coordenadas do objeto.

A classe DrawableVertex:

```
public class DrawableVertex {}
    private Vertex v;
    private int x, y;
    private int touchRadius = 50;
    private boolean touched;

    public DrawableVertex(Vertex v, int x, int y) {...}
    public DrawableVertex(Vertex v) {...}
    public Vertex getV() {...}
    public void setV(Vertex v) {...}
    public int getX() {...}
    public void setX(int x) {...}
    public int getY() {...}
    public void setY(int y) {...}
    public void handleActionDown(int eventX, int eventY) {...}
    public void setTouched(boolean touched) {...}
    public boolean isTouched() {...}
}
```

A classe DrawableStone:

```
public class DrawableStone {
   private Stone s;
   private int x, y;
   private Bitmap bitmap;
   private boolean touched;

   public DrawableStone(Stone s, int x, int y, Bitmap bitmap)
      {...}
   public DrawableStone(Stone s) {...}
   public Stone getS() {...}
   public void setS(Stone s) {...}
   public int getX() {...}
   public void setX(int x) {...}
   public int getY() {...}
   public void setY(int y) {...}
   public Bitmap getBitmap() {...}
   public void setBitmap(Bitmap bitmap) {...}
   public void draw(Canvas canvas) {...}
   public void handleActionDown(int eventX, int eventY) {...}
   public boolean isTouched() {...}
   public void setTouched(boolean touched) {...}
}
```

A diferença entre essas classes é que DrawableStone possui um campo para armazenar a imagem da peça e uma função para desenhar essa imagem na tela. DrawableVertex é apenas um mapeamento entre os vértices da parte lógica e as coordenadas na tela.

A classe DrawingSurface é uma extensão da classe SurfaceView onde foram implementadas as funções que criam, atualizam e desenham todos os elementos visuais do jogo e que também gerenciam os objetos DrawableStone e DrawableVertexes. Alguns campos foram omitidos para facilitar a leitura. O campo background será carregado com a imagem do tabuleiro que será desenhado na tela.

```
public class DrawingSurface extends SurfaceView implements
   SurfaceHolder.Callback {

   int h;
   int w;
   int xPadding;
   int yPadding;

   private EngineThread coreThread;

   private boolean stoneSelected = false;
   private DrawableStone selectedDrawableStone = null;

   private Bitmap background;
   private List<DrawableStone> drawableStones;
   private List<DrawableVertex> drawableVertexes;

}
```

A função makeBoards acessa a parte lógica para criar as DrawableStones e os DrawableVertexes correspondentes às Stones e aos Vertexes. Depois, ela acessa o novamente a parte lógica para descobrir qual o jogo que foi selecionado para mapear corretamente os pontos da tela para os DrawableVertexes.

```
public void makeBoards() {
      List<Stone> coreStones =
         coreThread.getCapture().realState.stones;
      List<Vertex> coreVertexes =
         Arrays.asList(coreThread.getCapture().getBoard());
      drawableStones = new ArrayList<DrawableStone>();
      drawableVertexes = new ArrayList<DrawableVertex>();

      for (Vertex coreVertex : coreVertexes) {
         DrawableVertex dv = new DrawableVertex(coreVertex);
         drawableVertexes.add(dv);
      }

      for (Stone coreStone : coreStones) {
         DrawableStone ds = new DrawableStone(coreStone);
         if (coreStone.getOwner() == Owner.PLAYER1)
            ds.setBitmap(BitmapFactory.decodeResource(getResources(),
               R.drawable.green_stone));
         if (coreStone.getOwner() == Owner.PLAYER2)
            ds.setBitmap(BitmapFactory.decodeResource(getResources(),
               R.drawable.yellow_stone));
         Bitmap bmp = ds.getBitmap();

         drawableStones.add(ds);
      }

      //atribuicao das coordenadas aos vertices de acordo com o
         jogo.

      (...)

      }
```

A função updateDrawablesPosition atualiza a posição das peças. A posição dos vértices é estática e definida pela função makeBoard. updateDrawablesPosition acessa cada DrawableStone, descobre qual a peça que ela referencia, e dessa referência descobre qual o vértice aquela peça está ocupando. Ela então procura qual dos DrawableVertexes referencia o vértice ocupado e atribui àquela DrawableStone as coordenadas do x e y do DrawableVertex encontrado.

```
  public void updateDrawablesPosition() {

      List<Stone> coreStones =
         coreThread.getCapture().realState.stones;

      Iterator<DrawableStone> dsIterator =
```

```
            drawableStones.iterator();
        while (dsIterator.hasNext()) {
            DrawableStone ds = dsIterator.next();
            if (coreStones.contains(ds.getS())) {
                Vertex v = ds.getS().getV();
                DrawableVertex dv =
                    findDrawableVertex(drawableVertexes, v);
                ds.setX(dv.getX());
                ds.setY(dv.getY());
            } else {
                dsIterator.remove();
            }
        }
    }
```

A função onDraw é responsável por desenhar na tela. Ela limpa toda a tela, desenha o tabuleiro e chama a função draw de todas as DrawableStones.

```
    @Override
    protected void onDraw(Canvas canvas) {
        canvas.drawColor(Color.WHITE);
        canvas.drawBitmap(background, xPadding, yPadding, null);

        for (DrawableStone ds : drawableStones) {
            ds.draw(canvas);
        }
    }
```

A função onTouchEvent também foi escrita na classe DrawingSurface. Ela recebe as coordenadas dos toques do usuário e envia-as para as DrawableStones, que iram executar sua função handleAnctionDown. Caso nenhuma peça tenha sido tocada, ou caso alguma peça tivesse sido tocada anteriormente, as coordenadas são enviadas para os DrawableVertexes. O comportamento observado pelo usuário é que seu primeiro toque seleciona uma peça e o seu segundo toque seleciona a casa para qual a peça deve ser mover. Caso ele clique em qualquer lugar que não especifique um movimento válido, a peça deixa de estar selecionada. Um toque na tela gera uma interrupção e essa função é chamada pelo sistema operacional automaticamente assim que a interrupção for processada.

```
    @Override
    public boolean onTouchEvent(MotionEvent event) {

        if (event.getAction() == MotionEvent.ACTION_DOWN) {
            Log.d(TAG, "Taunty");
            if (stoneSelected) {
                for (DrawableVertex v : drawableVertexes) {
                    v.handleActionDown((int) event.getX(), (int)
                        event.getY());
                    if (v.isTouched()) {

                        move.add(selectedDrawableStone.getS().getId());
                        move.add(v.getV().getId());
```

```
                waitingForInput = false;
                stoneSelected = false;
                selectedDrawableStone.setTouched(false);
                selectedDrawableStone = null;
                break;
            }
        }
        if (selectedDrawableStone != null)
            selectedDrawableStone.setTouched(false);
        stoneSelected = false;
        selectedDrawableStone = null;

    } else {
        for (DrawableStone s : drawableStones) {
            s.handleActionDown((int) event.getX(), (int)
                event.getY());
            if (s.isTouched()) {
                if (s.getS().getOwner() == Owner.PLAYER1) {
                    selectedDrawableStone = s;
                    stoneSelected = true;
                }
                break;
            }
        }
    }
}

return true;
```

Os tabuleiros são imagens que são carregadas pelo programa. Utilizou-se as imagens que o Projeto Lobogames fornece para impressão no site. Isso não apenas economizou tempo, pois não foi necessário desenhar os tabuleiros, como também deixa claro para o usuário que participou de alguma das atividades do projeto que esses jogos fazem parte do mesmo. As peças foram desenhado no estilo pixel-art especialmente para este aplicativo e também podem ser escaladas para telas maiores.

Inicialmente a parte gráfica foi desenvolvida utilizando a API OpenGL ES. Ao longo do desenvolvimento ela foi abandonada por ser excessivamente complicada. Os gráficos dos jogos são bastante simples e constatou-se que o resultado desejado poderia ser obtido com menos esforço usando a classe do Android SurfaceView. Assim, a parte gráfica em OpenGL foi descartada e desenvolvida novamente com a classe SurfaceView.

As coordenadas na tela dos vértices e das peças poderiam ter sido colocadas dentro da parte gráfica. As classes DrawableStones e DrawableVertexes geram redundância dentro do código e algumas operações a mais são necessárias devido a sua existência. Porém somente dessa forma seria possível manter a parte gráfica completamente transparente para a parte lógica.

3.3.3 Parte de Controle

O objetivo da parte de controle é unir a parte lógica e a parte gráfica e principalmente gerenciar o laço do jogo. A classe que representa o controle é a EngineThread, uma classe que estende a classe Thread. Ela possui uma DrawingSurface, que representa a

parte gráfica e uma Capture que representa a parte lógica.

```
public class EngineThread extends Thread {
   private boolean running;
   private SurfaceHolder surfaceHolder;
   private DrawingSurface drawingSurface;
   private Capture capture;
}
```

Essa classe possui a função run e dentro dela existe o laço que controla as chamadas da parte lógica e da parte gráfica. Resumidamente, o laço do jogo alterna entre uma chamada da parte lógica, que deve fazer o jogo prosseguir e uma chamada da parte gráfica que desenha os elementos na tela.

O loop de jogo implementado funciona da seguinte maneira: Se for o turno do jogador dois, que foi arbitrariamente escolhido como o jogador computador, o algoritmo Minimax é executado, e o movimento é realizado. Se for o turno do jogador um, o laço continua a ser executado sem realizar operações na parte lógica até que um toque na tela seja detectado. A detecção do toque é feita pela classe DrawingSurface e é comunicada pela variável waitingForInput. Quando esta variável for falsa, significa que o usuário fez um movimento. A função translateHumanMove se encarrega de interpretar esse movimento. Se for um movimento válido, ele será executado. Depois de efetuada a jogada, é verificado se o jogo chegou ao fim. Caso isso ocorra a tela de fim de jogo é desenhada e o próximo jogo é criado e executado. Caso o jogo não tenha terminado, atualiza-se a posição das peças na parte gráfica e elas são desenhadas pela função onDraw.

Há vários detalhes de controle para exceções que podem ocorrer na tentativa de desenho e sincronização da Thread que poluem visualmente o código. Por tanto, será apresentada uma versão simplificada que omite algumas linhas não relevantes para a lógica do funcionamento dessa função.

```
public void run() {
     drawingSurface.updateDrawablesPosition();
      drawingSurface.waitingForInput = true;

      while (running) {

       if (capture.turnOf(capture.realState) == Owner.PLAYER2) {
           m = capture.minimaxDecisionAB(capture.realState);
           s = capture.makeTheMove(capture.realState, m);
           capture.updateMovesHistory(s);
           capture.realState.nextTurn();
         }
          else {
            if (drawingSurface.waitingForInput == false) {
              m = capture.translateHumanMove(drawingSurface.move);
              s = capture.makeTheMove(capture.realState, m);
              capture.updateMovesHistory(s);
              capture.realState.nextTurn();
            }
            drawingSurface.move.clear();
            drawingSurface.waitingForInput = true;
          }
```

```
            if(capture.isTheGameOver(capture.realState)){
              Owner winner =
                 capture.whoIsTheWinner(capture.realState);
               drawEndGameScreen();
               if(moreGames) {
                 capture = new Capture(...);
                  setRunning(true);
                  drawingSurface.makeBoards();
                  drawingSurface.updateDrawablesPosition();

               }

            }

            drawingSurface.updateDrawablesPosition();
            canvas = this.surfaceHolder.lockCanvas();
            this.drawingSurface.onDraw(canvas);
            surfaceHolder.unlockCanvasAndPost(canvas);
        }
}
```

Para finalizar o capítulo, temos a classe MyActivity que é instanciada pelo Android quando o programa é inicializado. Um arquivo XML de layout define o que será exibido na tela. No caso o arquivo XML descreve um botão que quando clicado chama a função runGame. Essa função simplesmente instancia a classe DrawingSurface, que cuida do resto na sua inicialização.

```
public class MyActivity extends ActionBarActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
            WindowManager.LayoutParams.FLAG_FULLSCREEN);
        setContentView(R.layout.activity_my);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if
            it is present.
        getMenuInflater().inflate(R.menu.my, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
```

```
        // automatically handle clicks on the Home/Up button, so
           long
        // as you specify a parent activity in
           AndroidManifest.xml.
        int id = item.getItemId();
        if (id == R.id.action_settings) {
           return true;
        }
        return super.onOptionsItemSelected(item);
    }

    private void runGame(View view) {
        runWithGraphics(view);
    }

    public void runWithGraphics(View view) {
       setContentView(new DrawingSurface(this));
    }

}
```

4 RESULTADOS E ANÁLISE

O resultado final deste trabalho é uma plataforma de desenvolvimento, praticamente um framework, de jogos de captura para o sistema Android. Essa plataforma de desenvolvimento é composta pelas partes lógica, gráfica e de controle. Ela foi utilizado para implementar os seguintes jogos em uma sequência didática: Felli, Pretwa, Serpente, Alquerque, Felli com promoção, Alquerque com promoção, Damas.

Os resultados da implementação da parte lógica foram excelentes. As funções e as classes apresentadas no capítulo da implementação funcionaram como previsto. Qualquer jogo de captura da família do Alquerque pode ser adicionado e irá funcionar perfeitamente, inclusive com o algoritmo Minimax. Adicionar um jogo novo significa descrever o tabuleiro utilizando as classes Vertex, Edge e Stone. Por exemplo, a classe Capture foi toda escrita usando o Alquerque como teste e os demais jogos foram adicionados depois e funcionaram corretamente desde a primeira execução sem nenhuma modificação no código. Para adicionar jogos das outra famílias, as funções simpleMovesCheck e capturesCheck precisam ser sobrescritas para se adequaram às regras de movimento e de captura dessas outras famílias. A classes que modelam o tabuleiro, as peças e os estado podem ser utilizadas sem nenhuma modificação e dessa forma, até o Minimax poderá ser executado para esses jogos.

Ainda na parte lógica tem-se os resultados do Minimax que também são considerados muito bons. Trabalhando com 6 níveis de profundidade na árvore, a execução do algoritmo é rápida a ponto de ser imperceptível pelo usuário. Para os jogos com tabuleiros pequenos, a profundidade 6 é um excelente resultado, pois o jogo não costuma ter mais que 10 rodadas. Para profundidades maiores, as dificuldades são encontradas no Alquerque, onde o tabuleiro é grande e há muitas possibilidades de movimento. Todos os jogos com promoção de peças também vão retardar a execução do Minimax, pois as peças promovidas tem muitas possibilidades de movimento e todas devem ser consideradas. Para os jogos menores, a precisão da função de custo é quase irrelevante, contanto que ela pontue corretamente a vitória e a derrota. Isso acontece porque a árvore do jogo é pequena e rapidamente atinge-se os estados finais. Um resultado interessante, entretanto, é o da função de custo que aplica o quadrado da diferença. Quando essa função for usada há uma mudança no comportamento do Minimax. Ele busca a vitória, mas com a maior vantagem numérica de peças sobre o adversário. O computador faz jogadas gananciosas e ignora jogadas que poderiam levá-lo a vitória mais rapidamente. Foram implementados três níveis de dificuldade de formas diferentes para analisar os resultados. No nível fácil o computador faz uma jogada aleatória. As capturas continuam sendo obrigatórias e portanto em alguns casos ele se sai surpreendentemente bem. Porém, no final do jogo quando restam poucas peças, é fácil para o usuário ficar movendo as peças para frente e para trás até que o computador coloque uma de suas peças em risco. No nível médio

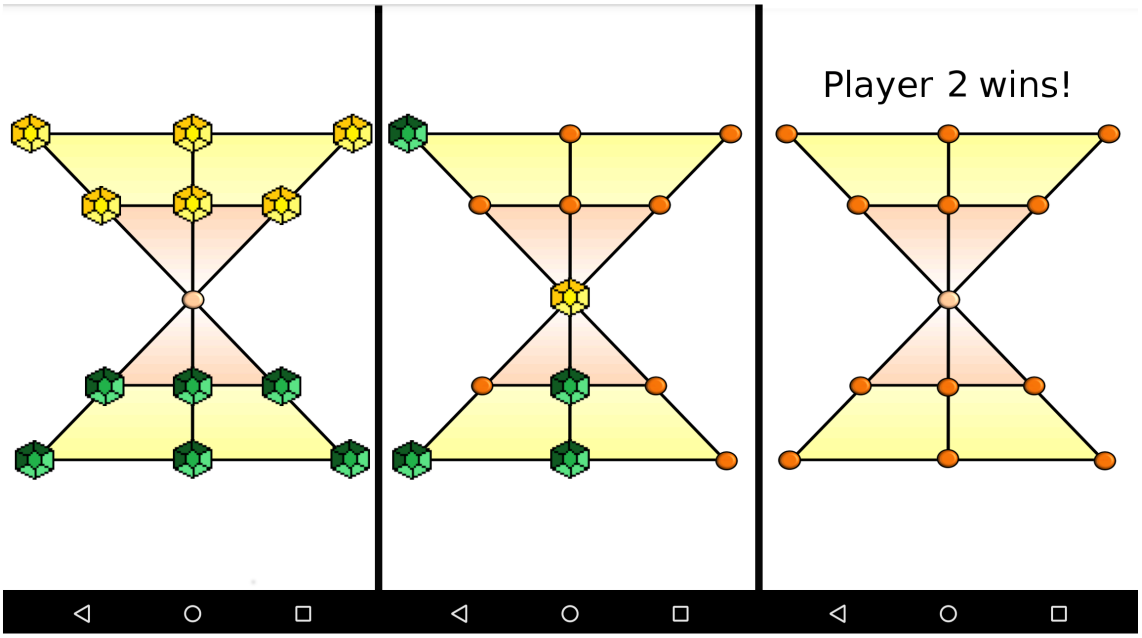


Figura 4.1: Capturas de tela do Felli

o jogador computador escolhe aleatoriamente uma jogada não ótima a cada três jogadas. Os resultados para o nível médio são pouco conclusivos. Em alguns jogos foi desafiante, em outros foi impossível vencê-lo. No nível difícil usa-se a função de custo do quadrado da diferença entre as peças. O seu comportamento ganancioso pode ser explorado pelo usuário para chegar à vitória. Caso ele não o faça, a sua derrota será devastadora. A vulnerabilidade do nível difícil foi exposta porque em alguns casos de teste em que o computador jogou contra outro computador, o que jogava aleatoriamente (ou seja, no nível fácil) venceu do que usava essa função de custo. Considerando as três implementações de controle de dificuldade, prefere-se a segunda, onde a cada n jogadas o Minimax erra de propósito. Apesar do esquema atual de controle de dificuldade ser suficiente, na segunda implementação tem-se um controle muito mais preciso sobre o comportamento do Minimax. É apenas uma questão de realizar mais testes e achar os parâmetros ideais para criar os níveis de dificuldade. Para finalizar, a parte lógica também gera um registro das jogadas realizadas. Esse registro pode ser facilmente utilizado para salvar um jogo, para implementar um sistema de replay ou para desfazer jogadas.

Os resultados obtidos na parte gráfica são satisfatórios. A implementação com as classes DrawableVertexes e DrawableStones se mostrou muito vantajosa. Primeiro porque garantiu que a parte gráfica ficasse totalmente transparante para a parte lógica. Segundo porque dá um bom suporte para adicionar animações para o movimento das peças. Terceiro porque elas podem se adaptar a qualquer tabuleiro. A dificuldade encontrada na parte gráfica é justamente a imagem dos tabuleiros. Para a parte lógica, criar um tabuleiro maior do Pretwa é apenas modificar um parâmetro. Para a parte gráfica é necessário que seja feito um desenho específico para aquele tamanho. Além disso, é necessário mapear as coordenadas x e y da tela para os vértices. Se o tabuleiro não for simétrico, essa tarefa, no momento, é praticamente manual. A vantagem de ter os tabuleiro como imagens é que qualquer um pode desenhar um tabuleiro e mudar o visual do jogo, contanto que sejam mapeados os vértices do desenho para os DrawableVertexes. As figuras que ilustram esse capítulo são capturas de telas do aplicativo rodando no smartphone Nexus 4.

Devido a natureza de funcionamento do Android, algumas funções que deveriam per-

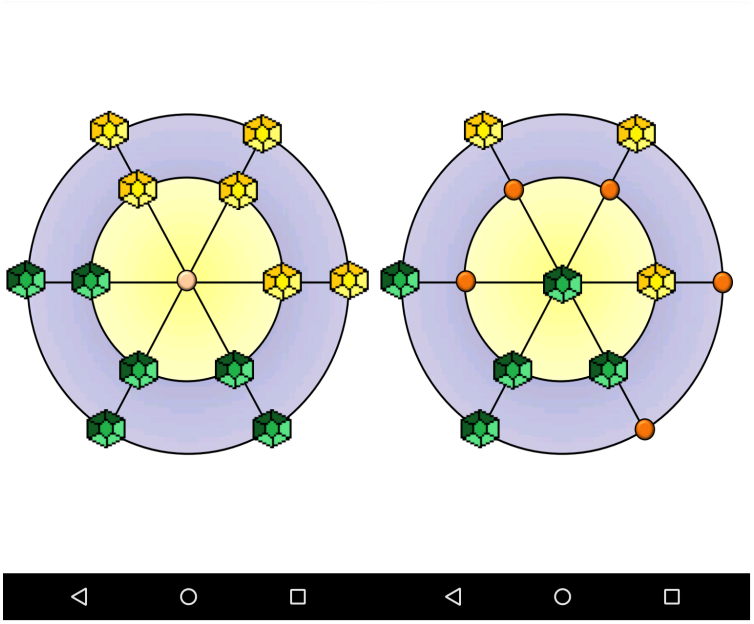


Figura 4.2: Capturas de tela do Pretwa

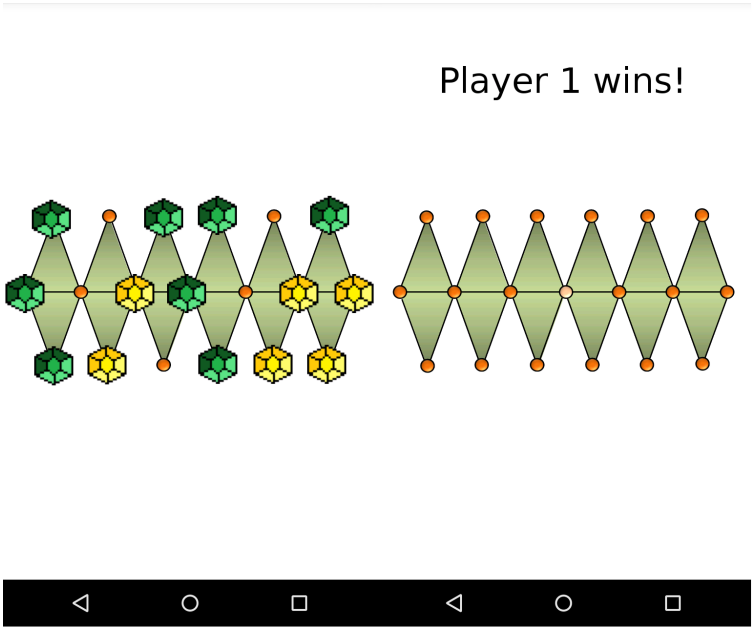


Figura 4.3: Capturas de tela do Serpente

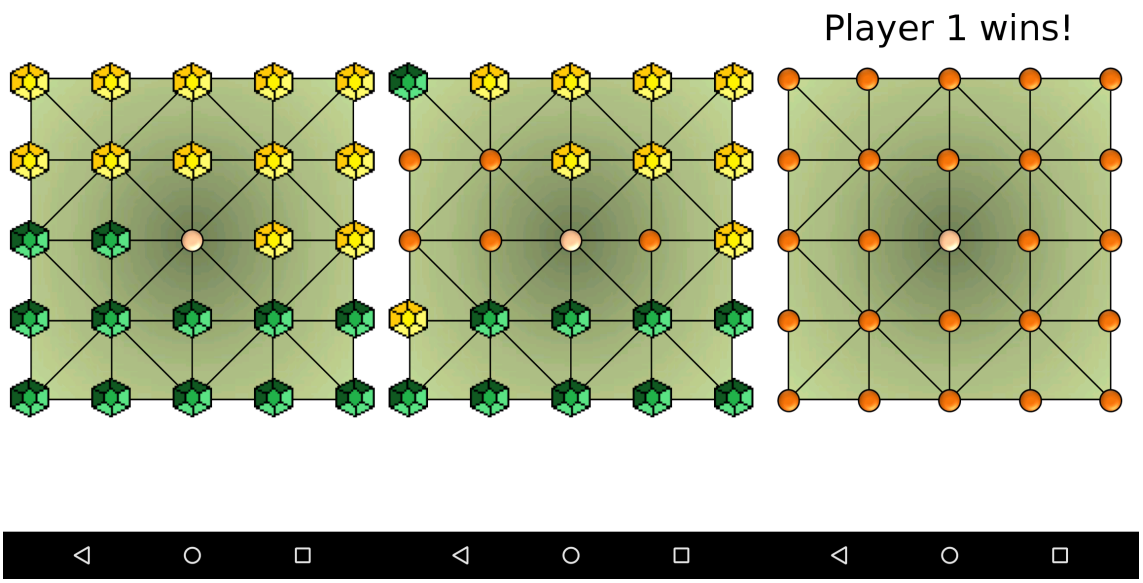


Figura 4.4: Capturas de tela do Alquerque

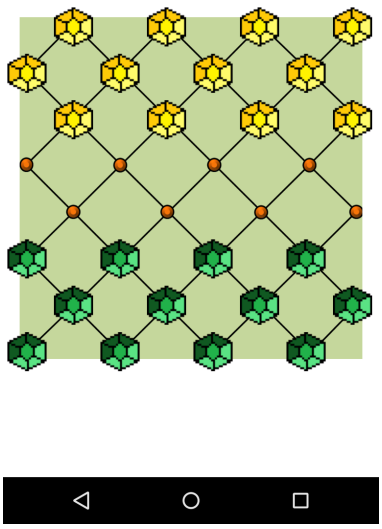


Figura 4.5: Captura de tela das Damas

tencer à parte de controle ficaram na parte gráfica, como por exemplo o processamento do toque do usuário. Apesar disso, considera-se que a parte de controle teve resultados bons.

Conclui-se dessa análise que de fato criou-se uma plataforma para o desenvolvimento de jogos de captura. A parte lógica oferece todas as ferramentas necessárias para processar o jogo e adicionar-se novos jogos com pouco esforço enquanto que a parte gráfica é bastante adaptável, mesmo que isso exija trabalho braçal. Essa plataforma foi então utilizada para implementar os jogos abordados em sua sequência didática.

5 CONCLUSÕES E TRABALHOS FUTUROS

O objetivo deste trabalho era construir uma plataforma computacional de apoio para o projeto de extensão Lobogames. O esperado é que essa plataforma implementasse os jogos do módulo de captura.

A versatilidade da plataforma permite que outros jogos de captura sejam adicionados com relativa facilidade e vários parâmetros podem ser ajustados para criar novos níveis de dificuldade e novas sequências didáticas de jogos.

Considerando os resultados apresentados no capítulo anterior, pode-se dizer que a plataforma foi implementada com sucesso e foi utilizada para construir um pequeno jogo seguindo uma sequência didática.

5.1 Trabalhos Futuros

Sugestões de trabalhos futuros são: implementar jogos de captura das outras famílias, aprimorar a interface gráfica e aproveitar os registros das jogadas.

Expandindo o uso dos registros das jogadas, algumas das funcionalidades interessantes são: salvar jogos, reproduzir partidas turno a turno e aplicações didáticas. As duas primeiras funcionalidades se complementam. A ideia é que os jogos possam ser salvos e analisados. No ambiente escolar, um professor poderia analisar os jogos do aluno e discutir com ele sobre as suas jogadas. Por aplicações didáticas, entende-se que o registro pode ser usado para o ensino de interpretação de coordenadas, como se faz com a Batalha Naval. O Pretwa, por exemplo, pode ser usado para ensinar coordenadas polares ou noções de geografia como a rosa dos ventos.

REFERÊNCIAS

ALLUÉ, J. M. **Jogos para o Todo o Ano - Primavera, Verão, Outono e Inverno**. [S.l.]: Editora Ciranda Cultural, 2002.

BURNS, B. **The Encyclopedia of Games**. [S.l.]: Barns & Noble Books, 1998.

FIEDLER, G. **Fix Your Timestep!** Acesso em novembro de 2014, `http://gafferongames.com/game-physics/fix-your-timestep/`.

GIORDANI, L. F.; RIBAS, R. P. Jogos lógicos de Tabuleiro. **XXXII Seminário de Extensão Universitária da Região Sul**, [S.l.], 2014.

GOOGLE. **Android Developers**. Acesso em novembro de 2014, `http://developer.android.com/index.html`.

KOUSH. **Universal Windows ADB Driver**. Acesso em novembro de 2014, `http://www.koushikdutta.com/post/universal-adb-driver`.

LOBOGAMES. **Lobogames, Jogos Lógicos de Tabuleiro**. Acesso em outubro de 2014, `http://www.inf.ufrgs.br/lobogames/`.

MEYER, B. **Object-Oriented Software Construction**. 1st.ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

MMXIII, O. M. **against the grain, game development explained**. Acesso em novembro de 2014, `http://http://obviam.net/`.

ORACLE. **Java Docs**. Acesso em novembro de 2014, `http://docs.oracle.com/javase/7/docs/api/`.

RIPOLL, O.; CURTO, R. M. **Jogos de Todo o Mundo**. [S.l.]: Editora Ciranda Cultural, 2011.

RUSSELL, S. **Artificial intelligence** : a modern approach. Upper Saddle River, NJ: Prentice Hall, 2010.

SHANNON, C. E. XXII. Programming a computer for playing chess. **Philosophical magazine**, [S.l.], v.41, n.314, p.256–275, 1950.

WINSTON, P. H. **Lecture 6**: search: games, minimax, and alpha-beta. Acesso em junho de 2014, `http://ocw.mit.edu/`.

Desenvolvimento de Jogos Lógicos de Tabuleiro

Arthur Carvalho Rauter[1]

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

acrauter@inf.ufrgs.br

Abstract. *This work is the development project of logical board games for the Android plataform. The games are associated with the UFRGS's extension project called Lobogames. This work describes the logical structure of the system and studies the artificial intelligence that will be implemented.*

Resumo. *Este Trabalho de Graduação é um projeto de desenvolvimento de jogos lógicos de tabuleiro para a plataforma Android. Os jogos estão associados ao projeto de extensão da UFRGS chamado Lobogames. O trabalho descreve a estrutura lógica do sistema e estuda a inteligência artificial que serão implementadas.*

1. Introdução

Jogos de Tabuleiro são um dos passatempos mais antigos da humanidade. Várias civilizações em diferentes épocas criaram seus próprios jogos ou adaptaram outros. Muitos jogos sobreviveram e tornaram-se parte da cultura contemporânea. Esses jogos permanecem interessantes porque são uma competição entre o raciocínio lógico dos jogadores envolvidos na disputa.

O objetivo geral do Projeto de Extensão da UFRGS Lobogames é promover e disseminar os jogos lógicos de tabuleiro, de acordo com [Lobogames 2010]. A motivação para tal é ensinar às pessoas a pensar logicamente. Os jogos de tabuleiro são a ferramenta escolhida porque são uma maneira divertida de exercitar o raciocínio lógico. Esses jogos já foram utilizados com fins semelhantes em [Giordani and Ribas 2013] . Uma das ações do projeto é o desenvolvimento de jogos de computador que implementem esses jogos de tabuleiro de forma didática. Os jogos, encontrados em [Allué 2002], [Burns 1998] e [Ripoll and Curto 2011], foram separados em 5 módulos diferentes e organizados por ordem de complexidade. Este trabalho trata do desenvolvimento dos jogos de um dos módulos, conhecidos como jogos de captura.

Vários jogos de tabuleiro de captura estão disponíveis online. Os mais populares como Damas, Alquerque e Hasami Shogi, por exemplo, podem ser encontrados também na loja de aplicativos Android da Google. Entretanto, as implementações estão descentralizadas e não existe nenhum programa que apresenta os jogos juntos e na ordem de complexidade que o projeto deseja.

A proposta deste trabalho de graduação é desenvolver os jogos de um dos módulos, o módulo dos jogos de captura, para o projeto Lobogames. A plataforma alvo é Android. Os jogos devem estar organizados como fases de um único jogo onde o jogador

humano, o usuário, joga contra o computador. Cada fase é um jogo de captura diferente ou uma variação dele. Os jogos que serão implementados são jogos clássicos de tabuleiro e estão expostos na seção 2. O trabalho consiste em projetar e codificar a estrutura lógica dos jogos, a interface gráfica e a inteligência artificial do jogador computador.

Na seção 2, serão apresentados os jogos do módulo de captura, as suas regras gerais, as semelhanças e as diferenças entre eles. Na seção 3 serão analisados os jogos que já existem e será explicada a proposta do trabalho. A seção 4 vai tratar sobre o desenvolvimento dos jogos: as decisões de projeto, a forma de representação dos tabuleiros e a codificação das regras dos jogos. O jogador computador será discutido na seção 5. O algoritmo Minimax será apresentado, pois ele é a base da inteligência artificial do jogador computador. Uma breve discussão sobre as ferramentas que serão utilizadas estão na seção 6. A última seção contém o cronograma de implementação do Trabalho de Graduação 2.

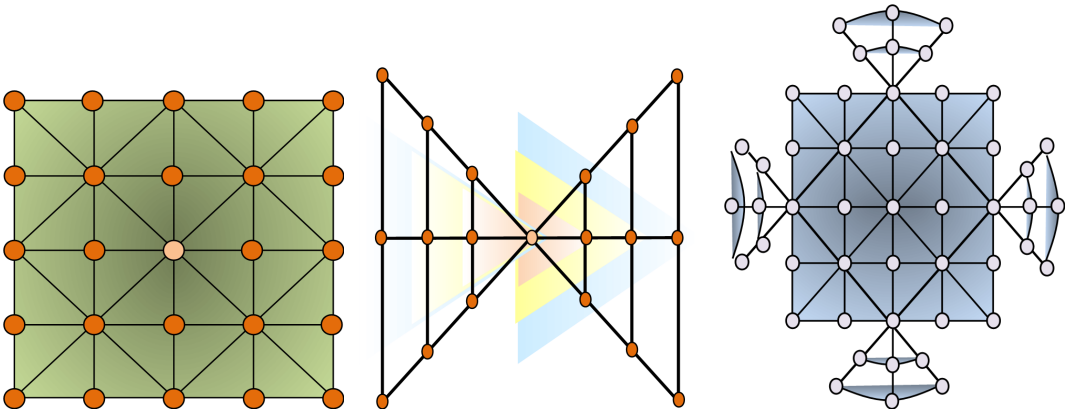
2. Jogos de Captura

Jogos de captura são jogos onde o objetivo de cada jogador é capturar todas as peças do jogador adversário. Jogos desse tipo que estão no projeto Lobogames são: **Alquerque, Felli, Dash-Guti, Pretwa, Sixteen Soldiers, Awithlaknannai, Damas, Tablut, Hasami Shogi, Seega, Surakarta, Fanorona e Four Field Kono**. Esses jogos foram divididos em três famílias de acordo com suas semelhanças.

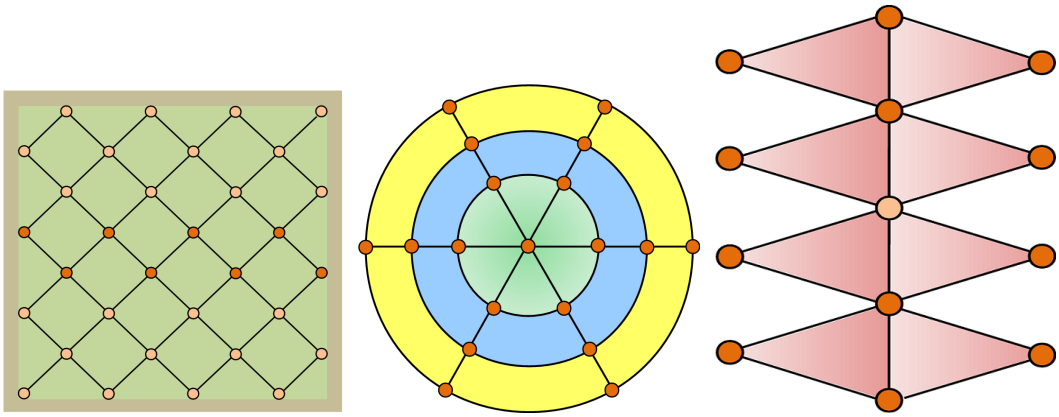
A primeira família é a do Alquerque, que contém os jogos: **Alquerque, Felli, Dash-Guti, Pretwa, Sixteen Soldiers, Awithlaknannai e Damas**. As peças se movimentam seguindo as linhas do tabuleiro, normalmente uma casa por vez. Uma captura ocorre quando uma peça salta sobre uma peça adversária. A casa de destinho do salto deve estar desocupada e alinhada com a casa de origem e a casa da peça capturada. De forma geral, as capturas são compulsórias e mais de uma captura pela mesma peça no mesmo turno são possíveis.

A segunda família é a da *Custodian Capture* ou *Custodial Capture*. Nela estão: **Tablut, Hasami Shogi e Seega**. As peças se movimentam apenas ortogonalmente. No Hasami Shogi e no Tablut elas podem se mover mais de uma casa por vez (como a torre do xadrez). Para realizar uma captura, o jogador deve cercar uma peça do adversário horizontalmente ou verticalmente. Mais especificamente, no seu turno, o jogador deve mover uma de suas peças para uma casa adjacente a peça do adversário que já esteja ao lado de uma peça do jogador. Se as 3 peças em questão estiverem em linha vertical ou ortogonal, a peça que foi prensada será capturada. A captura não é compulsória e só são possíveis capturas múltiplas no mesmo turno quando um único movimento cerca mais de uma peça.

A terceira família é a dos jogos especiais: **Surakarta, Fanorona e Four Field Kono**. No **Surakarta** as peças se movem ortogonal- e diagonalmente, uma casa por vez. Porém, para realizar uma captura, uma peça do jogador pode percorrer em linha reta qualquer número de casas livres até entrar em um dos loops do tabuleiro, fazer o loop e percorrer novamente qualquer número de casa livres até entrar em outro loop ou colidir com uma peça adversária, que será capturada. O **Fanorona** tem as mesmas regras do Alquerque, exceto por uma regra adicional de captura. Quando uma peça do jogador se afasta de uma peça adversária, essa peça adversária e as demais que estiverem atrás são

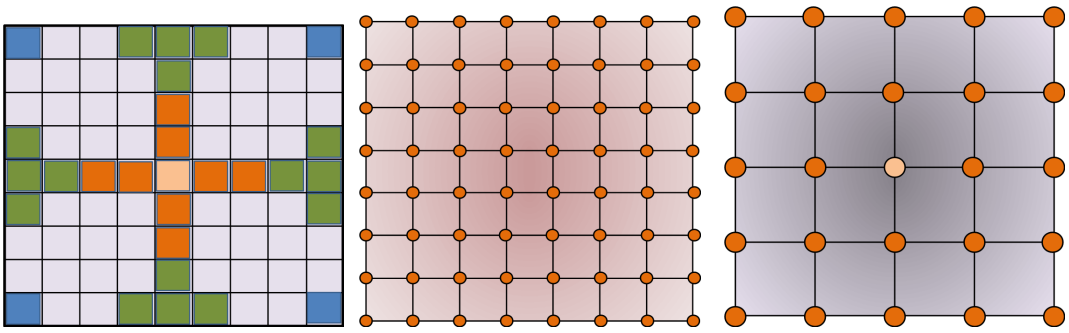


(a) Alquerque (b) Dash-Guti (c) Sixteen Soldiers



(d) Damas (e) Pretwa (f) Awithlaknannai

Figura 1. Jogos de Captura da Família do Alquerque



(a) Tablut (b) Hasami Shogi (c) Seega

Figura 2. Jogos de Captura da Família *Custodian Capture*

capturadas. No **Four Field Kono** as peças se movimentam ortogonalmente uma casa por vez e capturam uma peça adversária ao saltar por cima de uma peça amiga e aterrizar sobre a peça adversária.

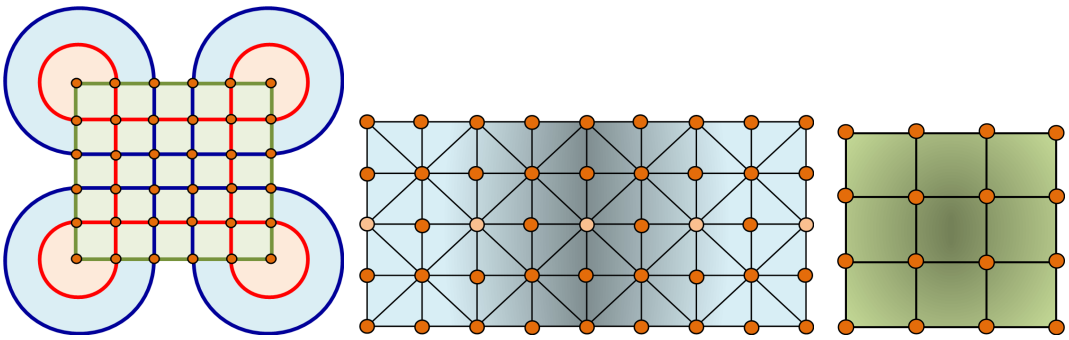
É comum que esses jogos possuam versões alternativas com modificações nas regras ou no tabuleiro. Algumas versões populares de damas, por exemplo, permitem que uma peça capture outra andando para trás. Para a maioria dos jogos, existe um tamanho

tradicional para o tabuleiro (número de casas e peças iniciais). Esse tamanho pode variar, sendo maior para versões mais longas e mais complicadas ou menor para versões mais simples.

3. Proposta

Os jogos de captura são jogos clássicos e estão disponíveis na Web em várias implementações diferentes. Os jogos que foram encontrados para Android na *Play Store* (`https://play.google.com/store`) estão listados abaixo, com links para os aplicativos.

1. Damas
   Como um dos mais populares jogos de captura, existem vários Apps. Entre os mais baixados estão:
   (a) `https://play.google.com/store/apps/details?id=net.elvista.checkers`
   (b) `https://play.google.com/store/apps/details?id=com.optimesoftware.checkers.free`
   (c) `https://play.google.com/store/apps/details?id=com.magmamobile.game.Checkers2`
2. Fanorona
   `https://play.google.com/store/apps/details?id=jp.qerozon.fanorona`
3. Surakarta
   `https://play.google.com/store/apps/details?id=com.masepio.surakarta`
4. Alquerque
   `https://play.google.com/store/apps/details?id=com.banjen.app.alquerque`
5. Hasami Shogi
   `https://play.google.com/store/apps/details?id=jp.co.ultimaarchitect.android.hasamishogi.free`
6. Tablut
   `https://play.google.com/store/apps/details?id=com.jocly.android.app10_vc_as`



(a) Surakarta (b) Fanorona (c) Four Field Kono

Figura 3. Jogos de Captura *especiais*

Damas, Hasami Shogi e Alquerque possuem implementações de boa qualidade com uma interface gráfica amigável e bem desenhada. O Tablut não possui nenhuma opção de jogar contra o computador, portanto só é possível jogar consigo mesmo. Os aplicativos do Fanorona e do Surakarta são rudimentares. A conclusão da análise dos jogos disponíveis para Android é que com eles, não seria possível construir o jogo que o projeto Lobogames deseja.

A proposta deste trabalho de graduação é projetar e desenvolver em Java e OpenGL um aplicativo com jogos de captura. Os jogos serão organizados em fases por ordem crescente de complexidade como proposto pelo projeto Lobogames. Portanto, será necessário projetar e codificar a estrutura lógica dos jogos, a interface gráfica e um jogador computador com inteligência artificial. Esse itens serão discutidos com mais detalhes nas próximas seções.

O valor do trabalho está na sua associação com o projeto de extensão Lobogames e seu caráter didático. O trabalho é inovador pela função didática do projeto de extensão e também por que irá implementar versões eletrônicas de jogos clássicos que ainda não existem. Entre as vantagens de possuir esse jogos em formato eletrônico estão a possibilidade de jogar contra um computador com vários níveis de dificuldade e a grande facilidade de modificar e configurar os jogos.

4. Projeto de desenvolvimento

O objetivo é aproveitar as semelhanças entre os jogos para tornar o código simples e evitar que cada jogo tenha que ser desenvolvido em separado. Por isso os jogos foram analisados e classificados em famílias, como apresentados na seção 2. Foi buscada uma representação interna de dados que fosse reutilizável para todos os jogos de captura e que facilitasse as operações do algoritmo de inteligência artificial descrito na seção 5. Esta seção, portanto, descreve o *core* dos jogos.

4.1. Estrutura de dados

Foram levadas em consideração três alternativas para a representação interna de dados do tabuleiro:

1. Matriz modelando o tabuleiro
2. Grafo representado por uma matriz de adjacência
3. Grafo representado por uma lista de adjacências

1 seria a escolha mais simples se todos os jogos tivessem o tabuleiro e as regras de movimentação semelhante ao jogo de damas: movimentos sempre para as casas adjacentes e o tabuleiro em formato retangular. Entretanto, o módulo inclui jogos onde o tabuleiro é circular e nesses casos a matriz não modelaria completamente a adjacência de todas as casas. Para tais jogos, seria necessário uma função especial para verificar todas as casas que são adjacentes a uma casa X, assim como funções especiais para verificar a validade dos movimentos. Tabuleiros triangulares também seria mais difíceis de serem modelados dessa forma.

A outra alternativa é modelar o tabuleiro como um grafo. A desvantagem dessa representação é que ela é menos visual que a matriz que modela o tabuleiro. A vantagem é que o grafo descreve perfeitamente o tabuleiro e a adjacência das casas de qualquer um

dos jogos de captura da seção 2, sem necessidade de regras especiais nos casos circulares, triangulares ou especiais. A questão seguinte é como representar o grafo.

Os tabuleiros foram modelados como grafos e a complexidade das operações que serão realizadas foram comparadas para ambas as formas de representação.

Grafos densos e esparsos são melhor representados, respectivamente, por matrizes de adjacência e por listas de adjacência. A densidade dos grafos pode ser calculada pela seguinte fórmula $D = \frac{2|E|}{|V|\,(|V|-1)}$ cujo resultado varia entre 1 a 0, sendo 1 a densidade máxima e 0 a densidade mínima. Dos jogos em questão, o que possui mais arestas por vértice quando modelado por um grafo é o Surakarta. Considere V como o conjunto de vértices e E como o conjunto de arestas. O Surakarta possui $|V| = 36$, $|E| = 236$ e, usando a fórmula da densidade, $D = 0,3746$. Tal densidade *sugere* (pois não existe um valor definido de densidade que separe os grafos esparsos dos grafos densos) que o grafo seja esparso. Como o grafo do Surakarta é o grafo mais denso que será utilizado e ele pode ser considerado esparso, pode-se afirmar que os demais grafos também podem ser considerados esparsos.

O espaço necessário para armazenar a matriz de adjacência é $O(|V|^2)$ e para a lista de adjacências é $O(|V| + |E|)$. Usando novamente o Surakarta como exemplo, a razão entre o tamanho da matriz e da lista é $4,76$. Portanto, no pior dos casos que deve ser considerado para a lista de adjacência, ela ocupa significativamente menos espaço que a matriz.

Uma verificação de adjacência entre dois vértices tem complexidade $O(1)$ na matriz e $O(|V|)$ na lista. Verificar quais são todos os vértices adjacentes a um vértice V é uma simples leitura na lista de adjacências, enquanto que na matriz de adjacências essa operação tem complexidade $O(|V|)$. Listar todos os vértices adjacentes é uma operação mais frequente que a consulta de adjacência, devido ao algoritmo da inteligência artificial que será utilizado e explicado na seção 5.

As operações de adição e remoção de vértice e de aresta são irrelevantes, pois os tabuleiro e os grafos são imutáveis ao longo do jogo.

Os jogos possuem grafos esparsos, cuja representação é mais econômica em listas de adjacência e sobre eles as operações mais realizadas são mais simples também com as listas. Assim, a conclusão é que a melhor forma de representar os tabuleiros é com grafos descritos por listas de adjacências.

Como mencionado na seção 2, o tabuleiro dos jogos pode variar em tamanho. A estrutura de dados permite que um tabuleiro maior ou menor seja facilmente gerado, sem influenciar as demais funções do programa. Essa é uma das vantagens de ter o jogo em formato eletrônico.

4.2. Movimento e Captura

As regras de movimentação variam de jogo para jogo, mas de forma geral, é necessário verificar se o vértice destino é adjacente ao vértice de origem e se ele está desocupado. A estrutura de dados que representa os tabuleiros possui essas informações. Se o movimento for válido, a estrutura de dados será atualizada e o movimento será realizado. No caso da captura, é necessário fazer verificações adicionais: Se a casa adjacente está ocupada de fato por uma peça adversária, se a casa destino está vaga e se a casa destino, a casa

ocupada e a casa origem estão todas alinhadas. Assim como o tamanho do tabuleiro, funções para diferentes tipos de captura podem ser facilmente implementadas e deixadas como opcionais para tornar o jogo mais configurável.

4.3. Final do Jogo

Os jogos terminam quando um jogador capturar todas as peças do jogador adversário. Assim, uma contagem de peças determinará o fim de jogo. Nos jogos em que há um limite de jogadas, um contador de turnos determina o fim de jogo e o vencedor será aquele com mais peças em jogo. Nos casos em que um jogador não puder realizar nenhum movimento ou captura no seu turno o jogo termina e o jogador adversário é declarado o vencedor.

Para os fins didáticos do projeto Lobogames, diferentes condições de final de jogo podem ser exploradas. Por exemplo, o jogo pode terminar quando algum dos jogadores perder apenas 2 peças. Essa condição força os jogadores a serem cautelosos. Novamente, isso é bastante simples de ser implementado e pode ser mais das opções de configuração do aplicativo.

4.4. Movimento e Captura do Jogador Computador

As jogadas do computador são determinadas pelo algoritmo Minimax que será discutido na seção 5. O Minimax age baseado apenas nas jogadas possíveis e portanto qualquer que seja a jogada escolhida, ela será valida. Uma vez escolhida a ação do jogador computador, as funções que atualizam a estrutura de dados que são usadadas para os movimentos do jogador podem ser reutilizadas.

4.5. Interface Gráfica

A interface gráfica estará separada do núcleo do jogo e será programada em OpenGL, conforme discutido na seção 4. Os gráficos dos jogos devem representar claramente o tabuleiro e as peças.

5. Jogador Computador

Dentro da teoria de Inteligência Artificial, ambientes que forem multi agentes e competitivos são considerados jogos. Um agente é definido na literatura como qualquer coisa que realize ações. Um agente racional é um agente que age buscando o melhor resultado ou o melhor resultado esperado, quando o ambiente for incerto. A definição de ambiente multi agente vem da teoria dos jogos: qualquer ambiente onde as ações de um agente tem impacto significativo nos demais, estejam eles cooperando ou competindo, é classificado como multi agente.

Os jogos que serão implementados tem sempre apenas 2 agentes, cujas ações influenciam um ao outro. Portanto, eles são jogos dentro da definição de jogo dada pela teoria dos jogos. Um dos agentes será o usuário. O outro agente será o jogador computador, controlado pelo programa. Ele deve competir com o usuário, logo queremos um agente racional.

Os jogos deste trabalho também são classificados como jogos de soma zero e com informação perfeita. Jogos de soma zero são jogos onde ou um jogador ganha e o outro perde ou ocorre um empate. Jogos com informação perfeita são jogos onde o ambiente é

completamente observável: isso significa que os agentes podem extrair do ambiente todas as informações que são relevantes para decidir qual a sua próxima ação.

De acordo com [Russell 2010], o algoritmo Minimax escolhe as ações ótimas com uma busca em profundidade na *árvore do jogo*, para jogos com 2 jogadores de soma zero e com informação perfeita. Assim, o Minimax é a escolha lógica para implementar a inteligência artificial do jogador computador em todos os jogos.

O jogador computador terá diferentes níveis de dificuldade. O objetivo dele não é vencer todos os jogos, mas sim jogar de forma competitiva com o usuário. Como controlar a dificuldade do jogador computador será discutido após a apresentação do algoritmo de inteligencia artificial que o controla.

5.1. Minimax

Considere um jogo de dois jogadores, MAX e MIN. MAX começa jogando. MAX e MIN alternam turnos realizando jogadas até que a condição de fim de jogo seja satisfeita. No final do jogo, uma função determina a pontuação de cada jogador. Tal jogo pode ser formalmente definido como um problema de busca com os seguintes elementos:

- *S*: estado inicial, especifica o início de jogo e a posição das peças.
- *s*: estado qualquer durante o jogo.
- PLAYER(*s*): retorna qual do jogadores tem o direito de se mover.
- ACTIONS(*s*): retorna quais as jogadas possíveis no estado *s*.
- RESULT(*s,a*): retorna o estado seguinte ao estado *s*, considerando a ação *a*.
- TERMINAL-TEST(s): retorna verdadeiro se o jogo tiver chegado ao fim no estado s.
- UTILITY(*s,p*): A função de utilidade. Retorna um valor numérico para um jogo que termine no estado *s* para o jogador *p*. Por exemplo, 1 se o jogador tiver vencido, -1 se ele tiver perdido ou 0 se ocorreu um empate.

Com S, ACTIONS e RESULT pode-se construir a árvore de jogadas (ou árvore do jogo) para qualquer jogo. Aplicando ACTIONS sobre *S*, temos os primeiros ramos e aplicando RESULT sobre *S* e o resultado de ACTIONS, temos os primeiros nodos. Assim, os ramos representam as jogadas (movimento ou captura) e os nodos a posição das peças no tabuleiro. Para construir a árvore completa, basta aplicar ACTIONS e RESULT repetidamente até que a aplicação de TERMINAL-TEST retorne verdadeiro para todas as folhas. Por fim, aplicamos nas folhas a função UTILITY para determinar a pontuação de MAX. No final desse processo, temos a árvore do jogo.

Dada a árvore do jogo, uma estratégia ótima pode ser determinada a partir do valor do Minimax em cada nodo. Tal valor é definido pelo seguinte algoritmo:

$$Val\_Minimax(s) = \begin{cases} UTILITY(s) & \text{if TERMINALTEST(s)} \\ max_a Minimax(RESULT(s,a)) & \text{if PLAYER(s) = MAX} \\ min_a Minimax(RESULT(s,a)) & \text{if PLAYER(s) = MIN} \end{cases}$$

Considerando que ambos jogadores escolherão as decisões ótimas, o valor de Minimax em *s* é o valor de UTILITY (do ponto de vista de MAX) para esse estado. No turno de MAX ele busca a ação que maximiza a função UTILITY. MIN faz o contrário,

ele busca a função que minimiza UTILITY (pois estamos tratando do ponto de vista de MAX).

Com a árvore do jogo e o valor de Minimax para cada nodo, basta percorrer a árvore a partir da raiz, buscando alternadamente entre o maior valor de Minimax (jogada de MAX) e menor valor (jogada de MIN). Os ramos percorridos são as jogadas feitas. Um pseudo algoritmo que implementa a escolha das jogadas pode ser encontrado em [Russell 2010].

Um exemplo do Minimax pode ser visto na figura 4: Um jogo simples, de apenas 2 turnos, começado por MAX. Nas folhas, temos o valor da função UTILITY. Percorrendo a árvore de baixo para cima obtém-se os valores de Minimax. Para o nodo B, o valor de Minimax é 3, pois este é o valor mínimo que a função UTILITY pode retornar para os filhos de B. MIN faz essa escolha para minimizar a a função UTILITY de MAX. Os valores de Minimax para os nodos C e D são encontrados de forma análoga e estão indicados a esquerda dos seus respectivos nodos. O valor de Minimax para o nodo A deve maximar a função UTILITY para MAX, portanto é o valor de Minimax do nodo B. A etapa seguinte é escolher as jogadas. MAX e MIN escolhem as jogadas que levam ao maior e ao menor valor de Minimax respectivamente. Portanto, MAX fará a jogada a_1 e MIN fará a jogada b_1. Esse é o caso de um jogo perfeito, onde MAX e MIN fizeram as jogadas ótimas.

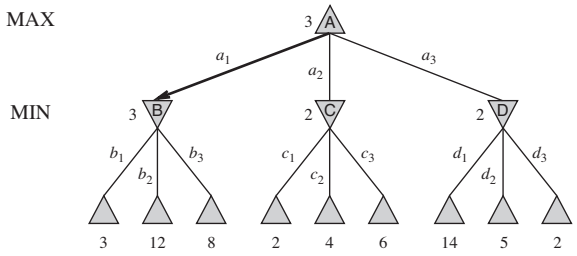


Figura 4. Árvore de um jogo simples

O Minimax realiza uma busca em profundidade por toda a árvore do jogo. Se a profundidade máxima da árvore é m e existem b jogadas possíveis em cada nodo a complexidade do Minimax é $O(m^b)$. Em [Russell 2010], [Shannon 1950] e [Winston 2010] é demonstrado que para o jogo de xadrez essa complexidade torna o uso do Minimax impraticável. Os jogos desse trabalho não são tão complexos quanto o xadrez, mas é fácil demonstrar que eles se aproximam mais do xadrez do que do jogo da figura 4. Assim, fica evidente que o Minimax não pode ser utilizado sem modificações. Entretanto, ele será a base do algoritmo que será implementado para a inteligência artificial do jogador computador. Aplicando a técnica de *Alpha-Beta Prunning* e a heurística da função de custo, que serão discutidas nas subseções 5.2 e 5.3, é possível tomar decisões em tempo real que se aproximam das decisões ótimas.

5.2. *Alpha-Beta Prunning*

Como visto na subseção anterior, a árvore do jogo cresce exponencialmente e a busca em profundidade é extremamente custosa. Não é possível eliminar o expoente da complexidade, mas no melhor caso é possível reduzí-lo pela metade com a técnica de *alpha-beta*

prunning, ou poda alpha-beta. Ela reduz a quantidade de subárvores pesquisadas, pois elimina subárvores que não influenciam no resultado do Minimax. A poda alpha-beta pode ser aplicada a árvores de qualquer tamanho e a resposta do Minimax é exatamente a mesma que a resposta do Minimax com a poda alpha-beta. A poda alpha-beta é uma otimização do Minimax.

O Minimax assume que tanto MIN quanto MAX sempre fazem as jogadas ótimas. Sabendo o comportamento dos jogadores, a poda alpha-beta pode desconsiderar ramos que não seriam escolhidos por nenhum deles. Por exemplo, considere o nodo *n* em algum lugar da árvore, tal que o jogador *P* tenha a possibilidade de realizar uma jogada e chegar em *n*. Se *P* tiver uma jogada melhor *m* ou em um nodo pai de *n* ou em qualquer ponto acima na árvore, *P* jamais escolherá a ação que leva a *n*. Logo, toda a subárvore que começa em *n* pode ser desconsiderada, ou *podada*. O nome alpha-beta vem dos parâmetros α e β que são utilizados durante a busca. α contém o valor da melhor escolha para MAX (ou seja, o maior valor) encontrada até chegarmos em *n* e β contém o valor da melhor escolha para MIN (ou seja, o menor valor). Esse valores são atualizado conforme a busca prossegue e tão logo o valor do nodo atual for sabido e avaliado pior do que α ou β para MAX e MIN, respectivamente, esse nodo será podado.

O desempenho da poda alpha-beta é altamente dependente da ordem em que os ramos são avaliados. Se, por exemplo, a busca for iniciada da esquerda para a direita e as jogadas ótimas estiverem todas nos ramos da extrema direita, nenhuma subárvore será podada. Se fosse possível definir uma função que retorna a ordem dos ramos que devem ser percorridos, a complexidade poderia ser efetivamente reduzida para $O(m^{\frac{b}{2}}))$. Obviamente, tal função não existe, pois ela escolheria as jogadas ótimas para ambos os jogadores e eles poderiam jogar um jogo perfeito. Segundo [Russell 2010], para o xadrez, uma função de ordenação simples que escolhe os ramos a serem avaliados de acordo com a ação, dando prioridade para: captura, ameaça, movimentos para frente e movimentos para trás, é possível chegar próximo à uma fração do caso ótimo de $O(m^{\frac{b}{2}}))$. [Russell 2010] cita outras técnicas para selecionar os ramos que devem ser considerados, como a heurística dos *killer moves* e a tabela de transposição.

5.3. Função de Custo

A poda alpha-beta discutida na subseção 5.2 reduz a quantidade de nodos que o Minimax deve pesquisar por no máximo 2. Entretanto, o Minimax ainda precisa gerar toda a árvore do jogo e pesquisar da raiz até as folhas o que ainda é impraticável para árvores tão profundas quanto as árvores geradas pelos jogos em questão. A escolha da jogada não deve demorar, no pior dos casos, mais que alguns segundos. Mesmo que fosse possível aplicar o Minimax, não é desejado que o jogador computador demore muito tempo para realizar sua jogada.

Claude Shannon propôs em [Shannon 1950] que os programas deveriam parar a busca antes de atingir as folhas e aplicar a heurística da *evaluation function*, ou função de custo, para tornar nodos não terminais em folhas. Em relação ao algoritmo do Minimax apresentado na subseção 5.1, a proposta de Shannon é substituir a função TERMINAL-TEST por uma função de corte CUTOFF e a função UTILITY por uma função de custo EVALUATION. A função de CUTOFF determina quando o algoritmo deve parar de percorrer a árvore com base na profundidade ou no tempo. A função EVALUATION deve

examinar o estado atual do jogo e retornar uma estimativa do valor da função UTILITY naquele estado.

A função de custo tem uma enorme influência na performance do jogador computador. Portanto, é preciso projetar uma boa função de custo, que se aproxime ao máximo dos resultados de UTILITY. Tal função deve: avaliar os estados terminais da mesma forma que UTILITY avaliaria (vitória melhor que empate, empate melhor que derrota), a computação não pode ser demorada (o propósito é justamente tornar o algoritmo mais rápido) e por fim, para os estados não terminais, o resultado deve ser fortemente correlacionado com as verdadeiras chances de vitória [Russell 2010]. A heurística da função de custo por ser imperfeita, introduz incerteza para reduzir o tempo de processamento. As imperfeições ou problemas da função de custo como a quiescência de um estado e o efeito horizonte são discutidos em [Russell 2010].

Cada jogo implementado terá a sua função de custo. [Russell 2010] e [Shannon 1950] discutem sobre possíveis funções de custo para o xadrez e estas serão usadas como base para as funções de custo que serão projetadas. A vantagem material é o fator mais importante de qualquer função de custo e combinado com uma busca de profundidade profunda o bastante, o algoritmo já é capaz de jogar um jogo de alto nível [Winston 2010]. Além da vantagem material outros movimentos e situações serão considerados, uma vez que nem sempre será possível pesquisar fundo o suficiente.

5.4. Níveis de Dificuldade

Como mencionado no início dessa seção, não é necessário nem desejado que o jogador computador seja invencível. Agora que o funcionamento do Minimax e suas melhorias foi apresentado, formas de torná-lo progressivamente menos eficiente podem ser analisadas. Entre as formas de controle de dificuldade, serão consideradas: o limite da profundidade de busca e a escolha não ótima de uma ação.

A profundidade da busca é controlada pela função CUTOFF e pode ser passada como um dos parâmetros da função. Assim, a eficiência do Minimax pode ser controlada dinamicamente durante a partida, se necessário. A limitação de profundidade na prática limita quantas jogadas a frente o jogador computador está considerando portanto é claro que alterando a profundidade de busca estamos alterando as capacidades do computador de escolher a melhor jogada.

Evitar escolher a solução ótima é uma outra alternativa. O algoritmo irá encontrar a ação ótima (dentro das limitações impostas pela profundidade de busca, pelo tempo de processamento e pela qualidade da função de custo) mas não irá escolher esta ação. Dessa forma, em todas as suas jogadas o computador comete um pequeno erro do qual o jogador humano pode tirar proveito.

Uma combinação das técnicas acima mencionadas também pode ser aplicada. Limitar somente a profundidade de busca pode ser exageradamente prejudicial para a performance do Minimax por causa do efeito horizonte. Nunca escolher a melhor ação pode criar um jogador computador que está visivelmente jogando mal de propósito o que pode ser frustante para o usuário.

6. Ferramentas de Desenvolvimento

A plataforma alvo é Android devido à sua popularidade. Ferramentas completas de desenvolvimento de jogos (como Unity 3D e GameSalad) e bibliotecas/APIs (como OpenGL) foram analisadas.

As ferramentas completas de desenvolvimento de jogos não exigem conhecimento de programação. Os jogos são feitos no estilo *drag and drop*, arrastando e configurando os elementos que o designer deseja inserir. No caso de jogos 3D, elas oferecem suporte bastante amplo a movimentação e animação de modelos 3D. As vantagens dessas ferramentas são a simplicidade para utilizar modelos 3D e desenhar a interface, a distribuição multi-plataforma e a facilidade de uso. A desvatagem é que a forma como os jogos são "programados"entrelaça a interface gráfica e a estrutura lógica do programa. Seria fácil fazer um jogo com uma bela interface gráfica, mas será difícil reaproveitar a estrutura lógica dele para os demais jogos. Assim, seria necessário desenvolver os jogos quase que individualmente. As facilidades das ferramentas não compensam esse trabalho extra. As ferramentas analisadas foram GameSalad e Unity3D.

O aplicativo será desenvolvido com a API OpenGL. Os gráficos dos jogos são simples e por isso as principais vantagens que as ferramentas oferecem não seriam aproveitadas. Há também uma motivação pessoal do autor para desenvolver diretamente para Android e com OpenGL. A linguagem de programação utilizada será Java pois é a linguagem típica de desenvolvimento para esta plataforma.

7. Cronograma

O projeto será desenvolvido ao longo de 17 semanas, iniciando em 04/08/2014 e terminando em 1/12/2014.

Semana	Atividade
1	Configurar o ambiente de programação
2	Explorar o OpenGL
3	Codificar as estruturas de dados para os jogos da família do Alquerque
4	Protótipo de um jogo de Damas, sem AI
5	Adaptar a estrutura de dados para os demais jogos
6	Protótipo de um jogo de Hasami Shogi, sem AI
7	Implementar o Minimax
8	Implementar a poda alpha-beta
9	Definir as funções de custo para os jogos
10	Adicionar a inteligência artificial aos protótipos
11	Finalizar os jogos da família do Alquerque.
12	Finalizar os jogos da família da Custodian Capture
13	Finalizar os demais jogos
14	Interface Gráfica do Menu de jogo
15	Testes
16	Ajustes finais
17	Ajustes finais

Referências

Allué, J. M. (2002). *Jogos para o Todo o Ano - Primavera, Verão, Outono e Inverno*. Editora Ciranda Cultural.

Burns, B. (1998). *The Encyclopedia of Games*. Barns & Noble Books.

Giordani, L. F. and Ribas, R. P. (2013). Jogos de raciocínio lógico na escolarização de surdos: promovendo movimentos no currículo. In *VIII Congresso Internacional de Educação e III Congresso Internacional de Avaliação*. UNISINOS.

Lobogames (2010). Lobogames, jogos lógicos de tabuleiro.

Ripoll, O. and Curto, R. M. (2011). *Jogos de Todo o Mundo*. Editora Ciranda Cultural.

Russell, S. (2010). *Artificial intelligence : a modern approach*. Prentice Hall, Upper Saddle River, NJ.

Shannon, C. E. (1950). Xxii. programming a computer for playing chess. *Philosophical magazine*, 41(314):256–275.

Winston, P. H. (2010). Lecture 6: Search: Games, minimax, and alpha-beta.