Universidade do Porto
Faculdade de Engenharia
FEUP

Jogo de Tabuleiro - Pentago

Relatório Final

Inteligência Artificial

3º ano do Mestrado Integrado em Engenharia Informática e Computação

Trabalho realizado por:

Ana Filipa Barroso Pinto – 201307852 – up201307852@fe.up.pt

Pedro Vieira Lamares Martins – 201005350 – up201005350@fe.up.pt

Pedro Miguel Vieira da Silva – 201306032 - up 201306032@fe.up.pt

29 de Maio de 2016

Resumo

Esta unidade curricular tem como objetivo dotar os seus estudantes de conhecimento relativo a sistemas inteligentes inteiramente criados, representados e manipulados por um computador. Nesse âmbito, propôs-se a implementação do jogo *Pentago*, utilizando o algoritmo de pesquisa adversarial *Minimax* com cortes alfa-beta, para a criação de um jogador inteligente. A implementação final coloca à disposição do utilizador uma interface gráfica intuitiva com três modos de jogo (Humano contra Humano, Humano contra Computador e Computador contra Computador) e um jogador automático com a possibilidade de escolha do nível de conhecimento (através da profundidade a que o algoritmo pode atingir).

Especificação

Descrição e regras do jogo

O *Pentago* é um jogo de estratégia do tipo soma-zero para dois jogadores concebido em 2005 por um *designer* sueco chamado *Tomas Flodén*. O jogo foi publicado no famoso *website* de jogos de tabuleiro *BoardGameGeek* e desde então, a partir de um *crowdfunding*, já foram vendidas mais de um milhão de cópias.

O jogo é realizado num conjunto de quatro tabuleiros quadrados de 3x3 cada, dispostos de forma a construir um tabuleiro maior de 6x6, totalizando 36 células onde cada um dos jogadores pode inserir peças (ou berlindes). Inicialmente, os tabuleiros encontram-se vazios.

Uma jogada consiste em dois movimentos sequenciais:

- O jogador em questão escolhe uma posição não ocupada num dos tabuleiros e coloca lá uma das suas peças.
- O jogador seleciona um dos tabuleiros para rodar 90 graus no sentido à sua escolha (no sentido dos ponteiros do relógio ou no sentido contrário).

O tabuleiro selecionado para rodar não tem de ser necessariamente o tabuleiro onde foi colocada a peça. O jogo desenrola-se ao longo de turnos consecutivos, trocando o jogador em cada turno. O jogo termina assim que, no final de cada turno, um dos jogadores conseguir pelo menos cinco peças em linha (no conjunto dos quatro tabuleiros).

Esta linha pode ser horizontal, vertical ou diagonal, e pode percorrer dois ou mesmo três tabuleiros. Se a última peça colocada pelo jogador resultou numa linha de cinco peças da sua cor, o jogo é finalizado sem a necessidade de se rodar um tabuleiro. O jogo termina num empate se não houver uma linha de pelo menos cinco peças no fim do jogo (quando todas as células do tabuleiro estiverem preenchidas com peças).

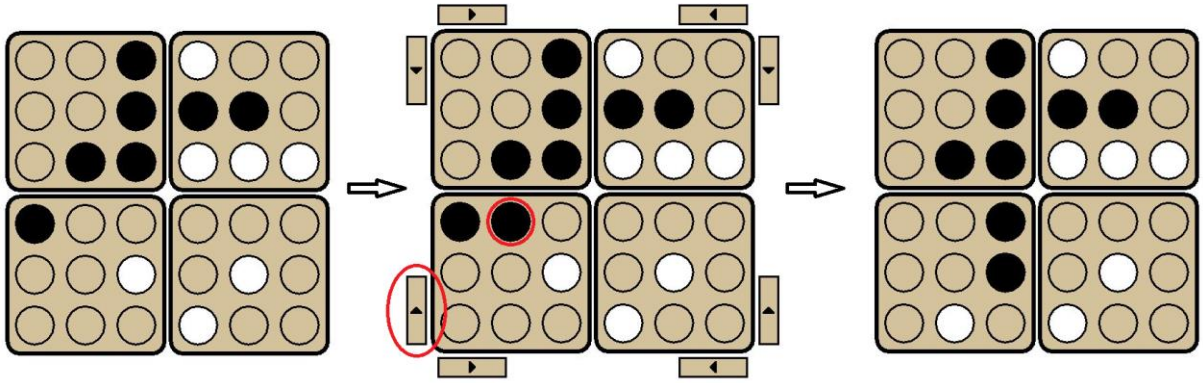

Figura 1 – Transição entre dois estados de jogo. É a vez do jogador representado pelas peças pretas. O jogador coloca a peça na posição destacada a vermelho e roda o tabuleiro no sentido dos ponteiros do relógio. Como existe uma sequência de pelo menos cinco peças da mesma cor, o jogador sagra-se vitorioso e é atingido um estado final.

Objetivo e tratamento do problema

O principal objetivo da realização deste projeto é a implementação de um jogador automático controlado pelo computador que possa jogar contra um utilizador humano. Este jogador inteligente deve utilizar o algoritmo de pesquisa adversarial *Minimax* com cortes alfa-beta e heurísticas apropriadas para adquirir conhecimento sobre o estado de jogo, decidindo, com base nisso, qual a sua melhor jogada possível para esse estado. Propõe-se também, para além disso, a implementação de uma interface gráfica de fácil interação com o utilizador, podendo este escolher entre três modos de jogo (humano contra humano, humano contra computador e computador contra computador) e o conhecimento que o jogador inteligente pode alcançar (através da profundidade máxima).

Para concretizar estes objetivos, foi feita a seguinte abordagem:

- Construção da mecânica de jogo e de todas as suas regras utilizando a linguagem de programação *Java*.
- O desenvolvimento de uma interface gráfica para interação com o utilizador utilizando o conjunto de ferramentas *Swing*, uma biblioteca de *Java*.
- A implementação do algoritmo *Minimax* e a sua aplicação ao jogo (através da criação da noção de “estado de jogo” e de heurísticas válidas).

Implementação da estrutura do jogo

Decidiu-se implementar a totalidade do projeto na linguagem de programação *Java*. As duas primeiras fases descritas anteriormente foram feitas em conjunto de modo a facilitar a sua criação.

A estrutura dos vários tabuleiros foi pensada como um *array* de *arrays* de inteiros (ou seja, uma matriz de inteiros). Cada espaço nessa matriz pode ser ocupado com três números diferentes, sendo o 0 representativo de uma célula sem peça, o 1 ocupado por uma peça de cor preta e o 2 por uma peça de cor branca.

A junção desses tabuleiros e as várias operações relacionadas (tais como, por exemplo, colocar uma peça, rodar um tabuleiro no sentido dos ponteiros do relógio ou verificar se foi encontrado um estado final para terminar o jogo) são feitas como funções simples nas classes correspondentes. O diagrama destas classes é apresentado mais à frente neste relatório.

A interface gráfica, como ponte de ligação entre o utilizador e o jogo, deve ser simples e intuitiva. Nesse sentido, a partir do conjunto de classes da biblioteca *Swing* do *Java*, bem como da utilização de interfaces que implementam o uso do rato e o funcionamento de *threads* (as interfaces *MouseListener*, *MouseMotionListener* e *Runnable*), foi possível criar uma interface que permite ao utilizador selecionar e decidir entre os vários parâmetros disponíveis na aplicação, tais como:

- Escolher o tipo de jogo (um humano contra outro humano, um humano contra o computador ou o computador contra o computador).
- Escolher o nível de inteligência para o computador (inteligência inexistente ou conhecimento dos estados de jogo para determinar a melhor jogada possível).
- Escolher a profundidade do algoritmo de pesquisa, caso se aplique (*minimax* com cortes alfa-beta, até uma profundidade máxima de 4).

Durante o jogo, o utilizador interage com a aplicação unicamente pelo uso do rato (clicando na célula e no botão para onde deseja colocar a peça e rodar o tabuleiro, respetivamente). Uma zona branca, por baixo do tabuleiro de jogo, clarifica ao utilizador humano aquilo que deve fazer ou explica a forma como o jogador automático procura/encontra a sua jogada. No caso do modo “Computador contra Computador”, o utilizador prossegue ao longo das jogadas clicando num botão. O jogador pode voltar ao menu inicial a qualquer momento.

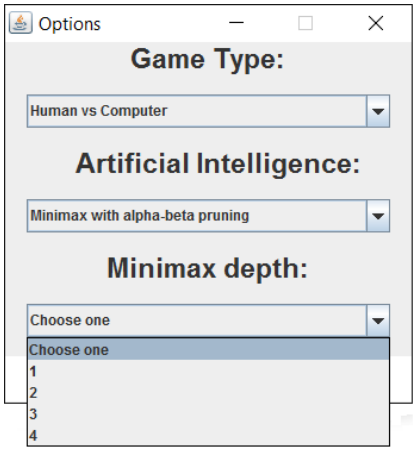

Figura 2 – Um utilizador, antes do início de cada partida, pode escolher o tipo de jogo e, caso se aplique, o nível de inteligência artificial (inexistente ou minimax com cortes alfa-beta) bem como a profundidade que o minimax pode atingir.

Abordagem algorítmica

Para a implementação de um jogador inteligente controlado pelo computador, utilizou-se o algoritmo de pesquisa adversarial *Minimax* lecionado na unidade curricular. Este algoritmo foi implementado inicialmente sem cortes alfa-beta, tendo sido os cortes implementados numa fase posterior. O *Minimax* é um algoritmo que lida com o problema de tentar minimizar a maior perda possível. Na teoria da decisão, é empregue, de entre outros, a jogos de soma zero (dos quais o *Pentago* faz parte). O algoritmo funciona da seguinte forma: descrevendo ambos os jogadores como Jogador A e Jogador B, e sendo o Jogador A o jogador artificial, se o mesmo tiver a possibilidade de vencer com uma próxima jogada, então o melhor movimento possível é essa mesma jogada. A cada passo assume-se que o Jogador A está a tentar maximizar as suas hipóteses de ganhar, enquanto que no turno seguinte o jogador B (o adversário) está a tentar minimizar as hipóteses de isso acontecer (ao maximizar a sua própria hipótese de ganhar).

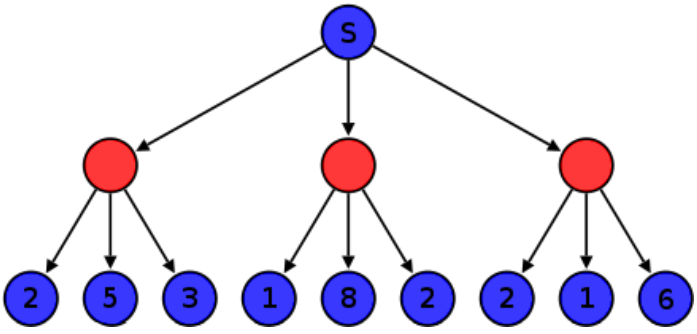

Figura 3 – Grafo ilustrativo de uma pesquisa em profundidade do algoritmo Minimax. Se no nível azul se estiver a maximizar e no nível vermelho a minimizar, serão escolhidos para o segundo nível os estados 2, 1 e 1 (nível a minimizar), e finalmente, de entre esses, é escolhido o estado 2 (nível a maximizar).

O funcionamento é garantido com base numa pesquisa em profundidade (com uma profundidade máxima estabelecida, escolhida pelo utilizador): a partir do estado atual de jogo, irá percorrer os estados seguintes possíveis, alternando cada nível como um nível maximizador (o seu) ou minimizador (o do adversário), até atingir um estado final ou a profundidade máxima. Esse estado final é então avaliado como muito ou pouco promissor, e esse valor é devolvido e utilizado pelos estados seguintes a percorrer.

O algoritmo *MiniMax* implementado neste projeto é uma variante denominada "MiniMax com cortes *alfa-beta*", que pode ser escolhida pelo utilizador na janela de opções no início do programa. Estes cortes *alfa-beta* correspondem, de forma bastante sucinta, a podas da árvore de pesquisa que tornam o algoritmo bastante mais eficiente tanto em tempo como em recursos utilizados (ver Figura 4).

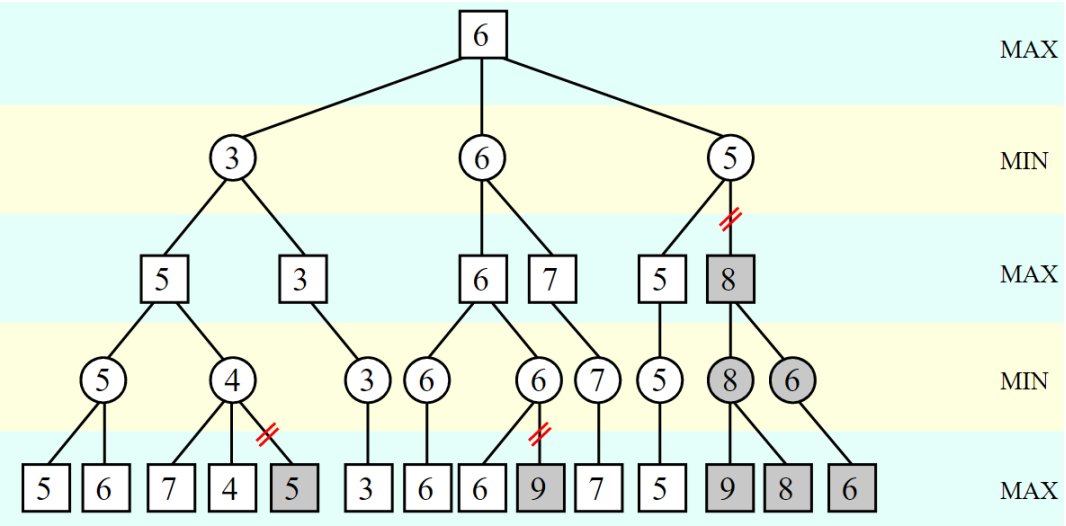

Figura 4 – Outra imagem de um algoritmo Minimax a funcionar, desta vez com cortes alfa-beta. O algoritmo deixa de pesquisar na árvore, cortando os nós filhos ainda por pesquisar, sempre que a função heurística provar que o movimento a examinar é pior que o movimento anterior já examinado.

Heurísticas para avaliação de estados

Tal como referido, de cada vez que é alcançada a profundidade máxima estabelecida (ou quando se atinge um estado final, não havendo a necessidade de continuar a pesquisar mais estados nessa ramificação), o algoritmo tem que trabalhar com um valor que será calculado com uma função de avaliação do estado de jogo atingido nesse momento. Para atingir este objetivo, foi determinada uma hierarquia de interesses estudada sobre o jogo. Esta vai determinar se um certo conjunto de condições estão reunidas no estado a ser avaliado, e irá atribuir pontuações a cada uma dessas condições, somando-as no final, e devolvendo esse valor. As heurísticas, por ordem crescente de importância, são:

- Padrões de vitória (linhas na vertical, horizontal ou diagonal com cinco peças da mesma cor) em cada linha de jogo.
- Padrões com 4 peças da mesma cor e com espaço para uma quinta peça, de forma a atingir um estado final.
- Padrões com 3 peças da mesma cor e com espaço para mais duas peças, de forma a atingir um estado final.
- Padrões com 2 peças da mesma cor e com espaço para mais três peças, de forma a atingir um estado final.
- Movimento aleatório.

Ao mesmo tempo, todas estas heurísticas são aplicadas ao jogador adversário, devolvendo valores negados dos estipulados. Um resultado positivo significa que o estado, em princípio, é promissor para o jogador em questão (e vice-versa, para o jogador adversário). Devido à natureza do jogo, é expectável que o algoritmo devolva bastantes vezes um resultado positivo.

O movimento aleatório resulta das heurísticas devolverem um valor nulo, não conseguindo prever o quão promissor é o estado observado (algo que nunca deverá acontecer, pelo menos para o *Pentago*, sempre que for usada uma profundidade que permita avaliar pelo menos duas jogadas do computador. Isto acontece porque, neste caso, a função heurística irá avaliar positivamente estados que juntem 2 peças da mesma cor num potencial padrão de vitória, e irá escolher esses estados como sucessores).

Desenvolvimento

A totalidade do projeto foi implementada no *Eclipse IDE* utilizando a linguagem de programação *Java*. A especificação foi bastante seguida à regra, priorizando-se a concretização do jogo propriamente dito, para de seguida lidar com as questões relacionadas com a inteligência artificial. Nesta secção irão ser descritos todos os detalhes importantes que levaram este projeto desde a concepção ao produto final.

Diagrama de Classes

O código do trabalho ficou dividido em dois pacotes (*packages*) diferentes, cada um agrupando um conjunto de classes com propósitos relacionados.

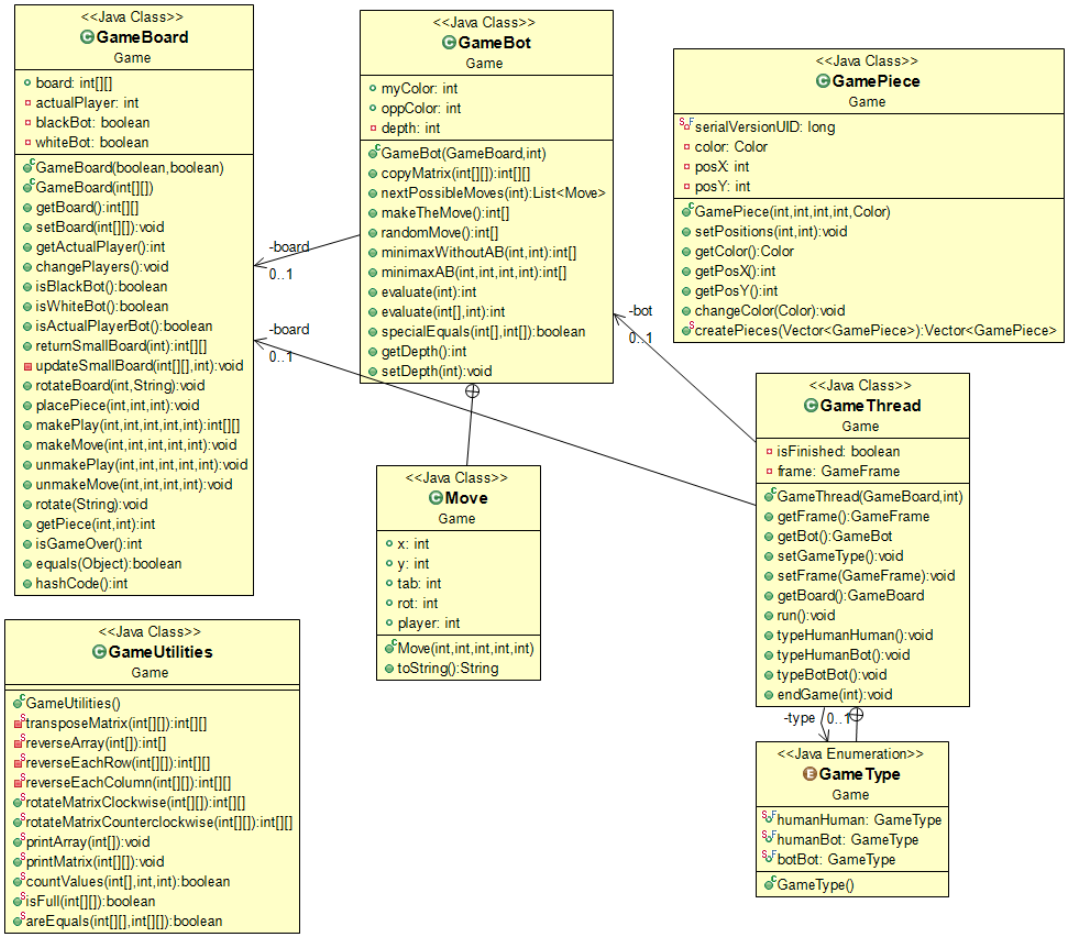

Figura 5 – Diagrama de classes do package “Game”.

O pacote *Game* reúne as classes responsáveis pela mecânica do jogo *Pentago*, desde a construção do esqueleto do jogo, passando pela definição das suas regras até à implementação de toda a inteligência artificial do jogador automático.

- *GamePiece* – é a classe mais básica cuja função é definir uma peça no jogo. Cada peça é identificada pelo sua cor e pelas suas coordenadas (na matriz de jogo).
- *GameBoard* – responsável pela implementação de todas as regras de jogo. Verifica a validade de cada jogada, opera sobre o tabuleiro para colocar peças e rodar os tabuleiros mais pequenos e averigua se já se atingiu, ou não, um estado final. Um *GameBoard* é constituído por uma matriz de 6 por 6 (correspondente ao tabuleiro de jogo) e informação sobre cada jogador (se são, ou não, controlados pelo computador, bem como se podem jogar no turno atual).
- *GameUtilities* – classe que faz várias operações no contexto do jogo, tal como transpôr e inverter uma matriz, ou verificar se duas matrizes são iguais.

- *GameThread* - agrupa todas as restantes classes e coloca as mesmas em funcionamento constante e flexível numa *thread.* Contém um Game*Board* que é o tabuleiro de jogo onde se desenrolará a partida, um *GameFrame* que é criado a partir desse tabuleiro e que interage com o utilizador (mostrando, graficamente, o estado atual do jogo), um *GameBot* (caso exista) e um *GameType*, que identifica o tipo de jogo em questão.
- *GameBot* - é responsável pela implementação do MiniMax necessário ao funcionamento do jogador inteligente controlado pelo computador, bem como as heurísticas de avaliação de um estado de jogo utilizadas por esse algoritmo. Funciona no tabuleiro (*Board*) que lhe é atribuído aquando da sua criação. Tem ainda uma profundidade que é escolhida pelo utilizador no início da aplicação.

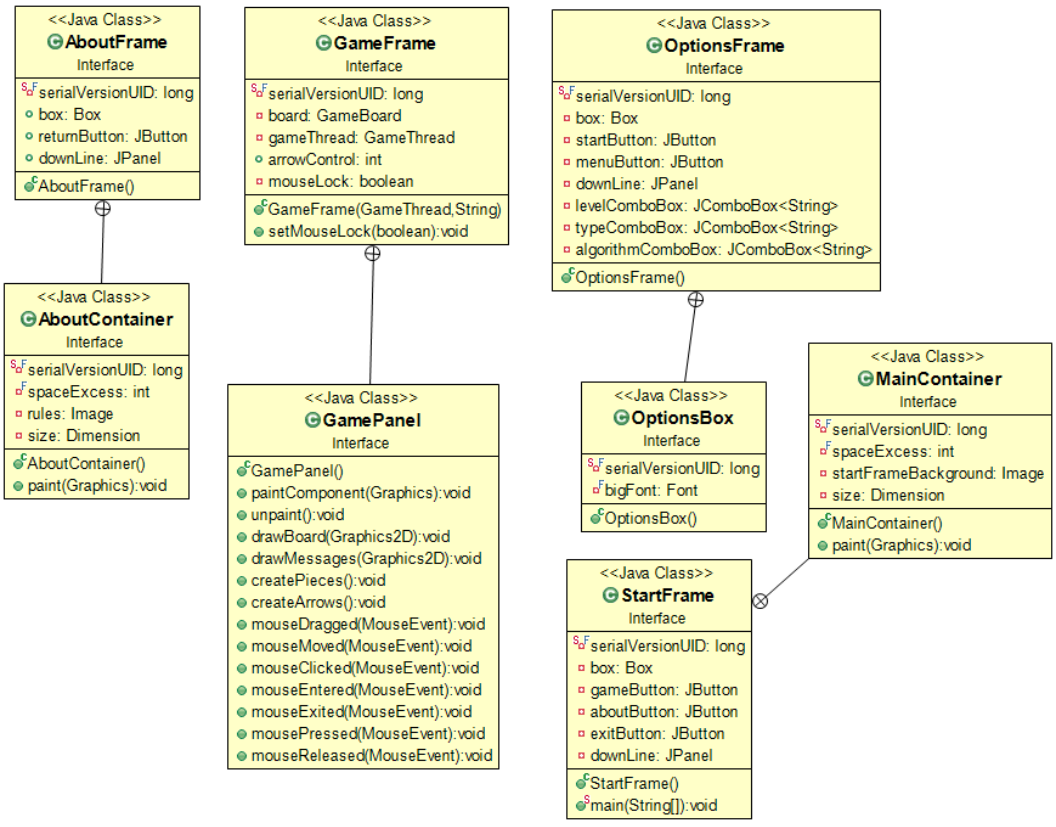

Figura 6 – Diagrama de classes do package "Interface".

O pacote *Interface* diz respeito, tal como o nome indica, à componente gráfica de interação com o utilizador exigida pelo trabalho.

Toda esta parte do projeto foi desenvolvida recorrendo à biblioteca Swing, um conjunto de ferramentas de *Graphical User Interface* disponibilizados pela linguagem.

- *StartFrame* – janela inicial de interação com o utilizador. Introduz o jogo e o âmbito do mesmo. Permite a escolha de iniciar uma partida ou fechar o programa.
- *OptionsFrame* – janela que permite ao utilizador a escolha do modo de jogo (de entre três disponíveis), o nível de inteligência artificial e a profundidade de conhecimento para o algoritmo *MiniMax* (caso se aplique).
- *AboutFrame* - janela onde é possivel visualizar uma pequena explicação de como funciona o jogo e quais são as suas regras. A partir desta janela é também possível visualizar informação sobre os criadores do jogo.
- *GameFrame* – é a janela principal onde se desenrola a partida. Dependendo do formato do jogo, o utilizador pode escolher a sua jogada seguinte. A interação é exclusivamente feita através do uso do rato (implementando as interfaces necessárias).

Implementação da inteligência artificial

Ainda que o desenvolvimento do jogo com os vários modos de escolha e uma interface gráfica fossem requiridos no fim do projeto, o grande objetivo sempre foi avaliar o estudante no âmbito da unidade curricular, fazendo com que este se debruce num determinado tema da área. Surgiu, assim, a necessidade de implementar um jogador inteligente que recorresse ao algoritmo de pesquisa adversarial *MiniMax*, lecionado na cadeira.

O produto final coloca à disposição do utilizador a possibilidade de escolher entre um jogador sem conhecimento do jogo (chamado de "*Valid random play*", que consiste numa função que devolve uma jogada seguinte válida porém completamente aleatória) ou outro que, a partir de heurísticas de avaliação do estado de jogo, decide o que fazer a seguir.

Para conseguir este jogador foi criada uma função chamada *MinimaxAB* que aceita a profundidade máxima estipulada pelo utilizador e a cor do jogador a maximizar, e devolve um *array* com o *score* e o movimento selecionado.

```
public int[] minimaxAB(int depth, int player, int alpha, int beta) {
        List<Move> possibleMoves = nextPossibleMoves(player);

        int p1;

        if (player == 1)
                p1 = 2;
        (...)
```

```
(...)
else
        p1 = 1;

int score;

int cx = -1, cy = -1, ct = -1, cd = -1;
                        (...)
```

A função começa por determinar, a partir do estado em que o algoritmo é chamado, todos os movimentos que resultem em estados sucessores possíveis. Isto é feito com a função *nextPossibleMoves*, que aceita o jogador do turno do momento e devolve, numa lista, todos os *Moves* possíveis (da classe *Move*, que constituem um movimento para outro estado dadas as coordenadas da posição da peça a colocar bem como o número e a direção do tabuleiro a rodar). De seguida é declarada a variável que irá armazenar os vários scores lidos ao longo do algoritmo, sendo ainda também iniciados os futuros valores do melhor movimento encontrado.

```
                                        (...)
if (possibleMoves.isEmpty() || depth == 0)
                return new int[] { evaluate(p1), cx, cy, ct, cd };
        else
                for (Move move : possibleMoves) {
                        board.makeMove(move.x, move.y, move.tab, move.rot, move.player);

                        if (player == myColor) {
                                score = minimaxAB(depth - 1, oppColor, alpha, beta)[0];

                                if (score > alpha) {
                                        alpha = score;

                                        cx = move.x;
                                        cy = move.y;
                                        ct = move.tab;
                                        cd = move.rot;

                                }
                        } else {
                                score = minimaxAB(depth - 1, myColor, alpha, beta)[0];

                                if (score < beta) {
                                        beta = score;

                                        cx = move.x;
                                        cy = move.y;
                                        ct = move.tab;
                                        cd = move.rot;
                                }
                        }

                        board.unmakeMove(move.x, move.y, move.tab, move.rot);

                        // corte
                        if (alpha >= beta)
                                break;
                }

    if (cx != -1)
                return new int[] { bestScore, cx, cy, ct, cd };
        else
                return new int[] { bestScore, -1, -1, -1, -1 };
}
```

De seguida é inicializada a pesquisa pelo algoritmo. Esta pesquisa é feita em profundidade e de forma recursiva. O caso de terminação acontece quando a profundidade máxima é atingida ou assim que deixam de haver movimentos possíveis (quando se atinge um estado de fim de jogo). Para cada movimento possível (obtidos a partir de uma função auxiliar que analisa o estado de jogo atual), começa-se por aplicar de forma efetiva esse movimento no jogo (colocando-o num estado válido seguinte, a partir da função *makeMove*). A função é então chamada novamente (agora funcionando sobre o novo estado de jogo), até que se atinja um dos casos de terminação já mencionados.

Assim que isso acontecer, o estado de jogo nesse momento é então avaliado pela função heurística (*evaluate*), que aceita a cor do jogador em questão e devolve um valor que irá ser utilizado pelos estados anteriores: caso o nível seja maximizador, se esse valor for maior que o valor guardado em alfa (que é chamado como parâmetro da função, inicialmente, como *Integer.MIN_VALUE*), então esse valor é guardado em alfa e o movimento é tomado como aquele com mais potencial. O raciocínio é idêntico para o nível minimizador: caso o valor do *score* atual seja inferior ao valor de beta (que é instanciado como *Integer.MAX_VALUE*), esse valor é guardado em beta e o movimento é tomado com aquele com mais potencial. O grande poder dos cortes alfa-beta reside na poda da árvore que é feita sempre que o valor de alfa for superior ao valor de beta, fazendo um *break* no ciclo *for* e evitando que os restantes estados sejam percorridos e avaliados. De realçar ainda que, ao voltar a estados anteriores, é essencial desfazer o movimento feito durante essa chamada à função (assegurado por *unmakeMove*).

Esta implementação procurou seguir a conceptualização inicial do nosso algoritmo *MiniMax* (cujo pseudo-código, retirado do relatório intermédio, é apresentado de seguida):

```
minimax(profundidade, jogador) // jogador definido pela sua cor
se (fim do jogo || profundidade == 0)
  devolve avaliacaoDoEstado
descendentes = movimentos possíveis a partir deste estado
se (jogador == PECAS_PRETAS) // caso para o jogador a maximizar
  // encontra o máximo
  melhorValor = -INFINITO
  para cada descendente: {
    valor = minimax(profundidade - 1, PECAS_BRANCAS)
    se  (valor > melhorValor)
        melhorValor = valor
  }
  devolve melhorValor
caso contrário (jogador == PECAS_BRANCAS) // caso minimizar
  // encontra o mínimo
  melhorValor = +INFINITO
  para cada descendente: {
    valor = minimax(profundidade - 1, PECAS_PRETAS)
    se (valor < melhorValor)
        melhorValor = valor
  }
  devolve melhorValor

// chamada inicial para profundidade 3
minimax(3, PECAS_PRETAS)
```

Heurísticas para avaliação de estados

Um estado de jogo tem um determinado valor consoante as peças que o preenchem nesse momento. Um estado final existe quando cinco peças da mesma cor existem na diagonal, na vertical ou na horizontal. Isto significa que, caso a função de avaliação atinja um estado deste tipo, outro estado não final não pode ter melhor avaliação que esse estado.

A função *evaluate*, que aceita o identificador do jogador do turno onde a função é chamada, irá chamar outra função *evaluate* que aceita a linha de jogo e devolve uma avaliação dessa linha consoante padrões conhecidos. Esta função é executada seis vezes (para cada linha do tabuleiro) e o valor é progressivamente calculado. Por exemplo, imaginemos que o seguinte estado entra na função de avaliação (1 representa o jogador preto, 2 o jogador branco e 0 uma célula sem peça):

```
int[][] example = {
                {1, 1, 0, 1, 0, 0},
                {2, 0, 0, 0, 0, 0},
                {2, 0, 0, 2, 0, 0},
                {0, 0, 0, 0, 0, 2},
                {0, 0, 0, 0, 0, 0},
                {0, 0, 0, 0, 0, 0}
};
```

Como é a vez do jogador representado pelo número 1 fazer a sua jogada, o algoritmo percorre a primeira linha do tabuleiro e identifica-a como uma linha com potencial. De facto, é possível colocar peças nessa linha para eventualmente atingir um estado final. O algoritmo vai comparar com um dos padrões conhecidos (neste caso, com o padrão identificado como "`int[] p32 = { w, w, 0, w, 0, s }`") e devolver o valor associado a esse padrão (para este exemplo, o valor 3000, devolvido sempre que três peças estão juntas numa linha onde existe a possibilidade de se juntarem cinco peças). O ciclo irá percorrer, depois, as restantes linhas, somando o novo valor lido ao valor das anteriores linhas já calculadas. Esta leitura é feita três vezes: primeiro para as linhas, depois para as colunas (que são entendidas como linhas da matriz rodada no sentido dos ponteiros do relógio) e por fim aos casos específicos em que pode haver uma linha de peças na diagonal. O valor final é depois devolvido e colocado em *score* na função do *MiniMax*.

Os padrões existentes relacionam as diversas possibilidades de existência de peças numa linha de jogo com os valores que esses padrões têm para o utilizador (quanto mais perto de um estado final, maior valor terá). As heurísticas devem ainda procurar ser admissíveis. No caso do nosso jogo, esse facto é salvaguardado garantindo que um único padrão com mais peças devolve sempre um valor superior a vários outros com menos peças (por exemplo, um padrão com 4 peças é sempre melhor do que 5 padrões com 3 peças).

Foram criados padrões para qualquer combinação de peças numa linha que garante a chegada a um estado final. O valor é tanto maior quanto maior for o número de peças já existente nessa linha. Todos os padrões possuem uma variável do tipo 'w' (correspondente a um valor com peça, que deve ser igual no padrão todo), uma variável do tipo 's' (um valor qualquer que não faz diferença na avaliação do padrão, que pode ser 0, 1 ou 2) e vários zeros, correspondentes aos espaços em branco. Assim, por exemplo, o já mostrado padrão p32, representado por "`int[] p32 = { w, w, 0, w, 0, s }`", iguala com linhas cujo primeiro, segundo e quarto elemento têm o mesmo número (1 ou 2), o terceiro e o quinto são espaços em branco (0) e o sexto elemento é um outro número qualquer (0, 1 ou 2).

Padrões	Valor devolvido
p5x	500000
p4x	40000
p3x	3000
p2x	200

Outros detalhes relevantes

Além do que já foi mencionado, destaca-se ainda a implementação da função *nextPossibleMoves* já referida, responsável por gerar todos os estados sucessores possíveis a partir do estado em que é chamada. Esta função tem dois comportamentos que são relevantes para a eficiência e para os resultados do algoritmo *MiniMax*:

- **Diminuição dos estados sucessores possíveis:** sendo os estados seguintes determinados a partir de todos os movimentos válidos, dois movimentos, ainda que diferentes, podem gerar o mesmo estado sucessor. Por exemplo, considerando-se uma jogada inicial com o tabuleiro ainda vazio, o estado resultante de colocar a peça na posição (0,0) e rodar o tabuleiro 1 no sentido dos ponteiros do relógio é igual ao estado resultante de colocar a peça na mesma posição mas rodar o tabuleiro 1 no sentido contrário. No entanto, são dois movimentos diferentes. A função é responsável por evitar que isto aconteça, diminuindo significativamente o número de estados seguintes a percorrer e aumentando a eficácia do algoritmo (por exemplo, para um estado com uma peça, o número de estados sucessores vai de 280 para 105) .

- **Baralhamento da lista de resultados**: é importante garantir que o algoritmo de pesquisa não percorre o conjunto de estados sucessores sempre pela mesma ordem (para um dado estado). A função assegura que isso acontece utilizando uma chamada a *Collection.shuffle*, que vai baralhar aleatoriamente a lista resultante ainda antes desta ser percorrida.

À medida que o trabalho ia sendo desenvolvido, houve sempre a necessidade de testar os vários casos possíveis e impedir os impossíveis, revestindo toda a interação com o utilizador de modo a evitar potenciais erros ou *bugs* que pudessem estragar a integridade de uma partida. Por exemplo, garantir que um utilizador não pode interagir com a janela de jogo enquanto o algoritmo escolhe e realiza a jogada seguinte, ou garantir que não é instanciado um jogador automático para o modo de jogo "Humano contra Humano".

Um dos objetivos em paralelo com a concretização deste projeto foi sempre tentar seguir as boas práticas de programação. Nesse sentido, procurou-se sempre deixar o código o mais limpo e explícito possível, indentando-o e comentando-o sempre que havia a necessidade, corrigindo todos os avisos ("*warnings*") lançados pelo compilador, evitando os chamados "números mágicos" colocando constantes para o mesmo efeito e dando sempre importância à eficácia (ainda que pouco ou nada relevante para um projeto desta dimensão), tal como evitar ciclos desnecessários e importar apenas as bibliotecas necessárias para o progresso do trabalho.

Experiências

Para avaliar o resultado final alcançado, foram feitas três experiências com três objetivos diferentes:

- Estudo da complexidade do algoritmo com cortes face à versão sem cortes.
- Comparação de conhecimento para profundidades diferentes.
- Eficiência das heurísticas para estados conhecidos.

Estudo da complexidade

Esta experiência foi necessária para determinar se todos os estados possíveis estavam a ser percorridos e analisados pela função de avaliação (por forma a garantir que o algoritmo estava corretamente implementado). Além disso, permitiu verificar se efetivamente o algoritmo tinha complexidade temporal exponencial (como esperado pelo *MiniMax*) e compara-la com a versão sem cortes (implementada numa fase inicial).

Para atingir este objetivo, foi cronometrado o tempo da sua execução, desde o estado inicial ao estado sucessor encontrado. Os resultados apresentam-se na seguinte tabela:

Profundidade (*depth*)	***MiniMax* sem cortes**	***MiniMax* com cortes**
1	0 segundos	0 segundos
2	12 segundos	2 segundos
3	4 minutos, 32 segundos	8 segundos.
4	-	3 minutos, 53 segundos.

Foram apenas testados valores de profundidade admitidos para escolha pelo utilizador. Uma vez que a pesquisa de *MiniMax* sem cortes para uma profundidade de 4 demorou mais de 10 minutos a encontrar um resultado, houve uma desistência dessa tentativa.

Comparação de conhecimento

Esta experiência resultou da necessidade de avaliar que, quanto mais profundo o algoritmo percorrer (ou seja, quantos mais estados forem analisados), melhor será o conhecimento que este terá do jogo e maior probabilidade ele tem de ganhar. Para isso foram corridos 100 jogos no modo Computador contra Computador, um deles correndo o *MiniMax* com uma profundidade de 3 e o outro com profundidade de 1. O resultado, um *print* na consola, foi o seguinte:

```
After 100 games...
MiniMax with depth 3 won 76 times.
MiniMax with depth 1 won 24 times.
```

A experiência demorou cerca de três horas a concluir.

Eficiência das heurísticas

O objetivo desta experiência foi avaliar o resultado da função de avaliação para vários estados e garantir que este valor resultava no estado sucessor possível esperado. Para isso, além dos vários testes que foram feitos ao longo do tempo de implementação no jogo propriamente dito, foram criados três estados possíveis e chamada a função de avaliação para esses mesmos estados para avaliar o resultado alcançado.

Matrix utilizada	Valor horizontal	Valor vertical	Valor esperado

`int[][] testMatrixOne = {`
`{1, 1, 0, 1, 0, 0},`
`{2, 0, 0, 0, 0, 0},`
`{2, 0, 0, 2, 0, 0},`
`{0, 0, 0, 2, 0, 0},`
`{0, 0, 0, 2, 0, 0},`
`{0, 0, 0, 2, 0, 0}`
`};`
	1ª linha – 3000
	2 ª linha – 0
	3ª linha – 200
	4ª linha – 0
	5 ª linha – 0
	6ª linha – 0
	TOTAL = 3200
		1ª linha – 200
		2 ª linha – 0
		3ª linha – 0
		4ª linha – 40000
		5 ª linha – 0
		6ª linha – 0
		TOTAL = 40200
			43400
			Valor obtido
			43400

Matrix utilizada	**Valor horizontal**	**Valor vertical**	**Valor esperado**

`int[][] testMatrixTwo = {`
`{2, 2, 2, 2, 2, 0},`
`{1, 1, 1, 1, 0, 0},`
`{1, 1, 1, 1, 0, 0},`
`{0, 0, 0, 0, 0, 0}`
`{0, 0, 0, 0, 0, 0}`
`{0, 0, 0, 0, 0, 0}`
`};`
	1ª linha – 500000
	2 ª linha – 40000
	3ª linha – 40000
	4ª linha – 0
	5 ª linha – 0
	6ª linha – 0
	TOTAL = 580000
		1ª linha – 200
		2 ª linha – 200
		3ª linha – 200
		4ª linha – 200
		5 ª linha – 0
		6ª linha – 0
		TOTAL = 800
			580800
			Valor obtido
			580800

Matrix utilizada	**Valor horizontal**	**Valor vertical**	**Valor esperado**

`int[][] testMatrixThree = {`
`{1, 0, 0, 0, 0, 0},`
`{0, 0, 0, 0, 0, 0},`
`{0, 0, 0, 0, 0, 0},`
`{0, 0, 0, 0, 0, 0},`
`{0, 0, 0, 0, 0, 0},`
`{0, 0, 0, 0, 0, 0}`
`};`
	1ª linha – 0
	2 ª linha – 0
	3ª linha – 0
	4ª linha – 0
	5 ª linha – 0
	6ª linha – 0
	TOTAL = 0
		1ª linha – 0
		2 ª linha – 0
		3ª linha – 0
		4ª linha – 0
		5 ª linha – 0
		6ª linha – 0
		TOTAL = 0
			0
			Valor obtido
			0

Conclusões e considerações finais

Através das experiências efetuadas, conseguimos retirar as seguintes conclusões:

- O algoritmo MiniMax com *cortes alfa-beta* é bastante mais eficiente que a mesma versão do algoritmo sem cortes. Isto permite que o algoritmo percorra mais estados e ganhe mais conhecimento sobre o jogo. Não obstante, o algoritmo continua com crescimento exponencial (ainda que o factor de exponencialidade seja mais pequeno que o do *MiniMax* sem cortes), e demora ainda bastante tempo a terminar para profundidades acima de 4.

- O facto de se conhecer estados sucessores mais longínquos (através da profundidade que o algoritmo percorre) permite que, em boa parte das vezes, esse conhecimento resulte numa vitória.
- Uma boa heurística devolve valores admissíveis para os mais variados estados, sendo esse valor posteriormente utilizado no contexto do algoritmo *MiniMax* para obter a melhor jogada possível no momento.

Este projeto permitiu ainda consolidar noções de Java (nomeadamente ao nível da biblioteca *Swing*) e de estruturas de dados (utilização de vectores, listas, *arrays* e *sets*).

Melhoramentos futuros

Em relação ao trabalho já efetuado, propõe-se como melhoramentos futuros as seguintes adições (dentro, obviamente, do campo de matéria em que o âmbito deste projeto se insere):

- Implementação de algoritmos alternativos de pesquisa adversarial (tal como o *NegaMax* ou o *ExpectiMiniMax*).
- Aprofundamento das heurísticas para permitir uma maior habilidade de dar uma avaliação mais precisa aos estados de jogo (de forma a poder uniformizar o jogador vitorioso na segunda experiência da comparação do conhecimento).
- Possibilidade de criação de um registo de jogadas para o utilizador poder analisar o raciocínio do jogador artificial.
- Outro tipo de interações com o utilizador, tais como retrocesso de jogadas, pausa na partida ou a realização de testes em *run-time* para avaliar a eficiência das soluções inteligentes disponíveis.

ANEXO A – Recursos utilizados:

Bibliografia:

1. Caso de estudo para a implementação do algoritmo *MiniMax*, "Tic-tac-toe AI", http://www3.ntu.edu.sg/home/ehchua/programming/java/javagame_tictactoe_ai.html
2. Regras e estratégia de jogo, "Pentago Game Rules", https://webdav.info.ucl.ac.be/webdav/ingi2261/ProblemSet3/PentagoRulesStrategy.pdf
3. Tutoriais de Swing, The Java ™ Tutorials, "Trail: Creating a GUI With JFC/Swing", http://docs.oracle.com/javase/tutorial/uiswing/ (vários links).
4. *"Using Heuristics to Evaluate the Playability of Games",* 2004, Heather Desurvire, Martin Caplan, Jozsef A. Toth, algumas visões de como testar a eficiência de heurísticas em jogos, http://userbehavioristics.com/downloads/usingheuristics.pdf.
5. GitHub – repositório com o código desenvolvido para o projeto (Branch Game), https://github.com/arcanjo45/IART

Software utilizado na realização do trabalho:

1. Eclipse Mars SDK (versão 4.5.1) – para a implementação do código.
2. Adobe Photoshop (versão CC 1.2) – para a edição das várias imagens utilizados no projeto.
3. ObjectAid UML Explorer (versão 1.1.8) – para a geração do diagrama de classes.
4. Microsoft Word (Office 2015) – para a escrita do Relatório Intermédio e do Relatório Final.

Percentagem aproximada de trabalho efetivo:

O trabalho foi equitativamente dividido por todos os elementos do grupo:

Ana Filipa Barroso Pinto – 33%

Pedro Vieira Lamares Martins – 33%

Pedro Miguel Vieira da Silva – 33%

ANEXO B – Manual do utilizador:

- Fazer o download do ficheiro *IART1516_FINAL_GC5_4.zip* e extrair o código em "Código" para uma pasta qualquer.
- Fazer *Import* e *Build* dos vários ficheiros .java utilizando um qualquer IDE (por exemplo, Eclipse).
- Correr o programa.

- Clicar em "Play" para jogar, em "About" para ler as regras do jogo ou em "Exit" para sair do programa.
- Clicando em "Play", escolher as devidas opções de jogo (ver Figura 2) e iniciar um jogo. Voltar para o menu anterior sempre que quiser clicando no 'X'.