Choko

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo 23:
João Pedro Domingues da Rocha Marinheiro – 201101774
Luís Filipe Correia Cleto – 201104279

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

8 de Novembro de 2013

Resumo

O objetivo deste trabalho é criar uma implementação em Prolog do jogo de tabuleiro Choko, que permita três modos de jogo: humano contra humano, humano contra computador e computador contra computador. Deve ainda permitir três níveis de dificuldade para os jogadores controlados pelo computador (fácil, média e difícil).

Para isto foi necessário criar uma representação do estado do jogo, predicados que apliquem a sua mecânica, algoritmos de decisão para a inteligência artificial e especificar uma interface com o utilizador.

A representação do estado do jogo consiste numa lista de listas para o tabuleiro, dados agrupados para os jogadores e outras variáveis que indicam o turno e outros parâmetros do jogo e será explicada em detalhe na secção de Lógica do Jogo deste relatório, juntamente com os predicados que controlam a mecânica do jogo.

Os algoritmos de decisão para a inteligência artificial baseiam-se no estudado nas aulas teóricas (gerar todas as possibilidades e escolher a melhor). Na decisão optou-se ainda por incluir um fator aleatório de erro para evitar que duas AIs iguais fiquem em 'loop' uma contra a outra devido a usarem as mesmas estratégias. Foi incluído também o fator aleatório na escolha entre duas jogadas com o mesmo valor para evitar jogos repetitivos (duas Ais iguais fariam sempre as mesmas jogadas).

Para a interface, foi decidido apresentar sempre ao utilizador a última alteração ao estado de jogo, especificar o tipo de *input* esperado e pedir que o utilizador insira *input* conforme especificado.

Conteúdo

1. Introdução

Este trabalho tem como objetivo criar uma implementação, em linguagem Prolog, do jogo de tabuleiro Choko que permita os seguintes modos de jogo: humano contra humano, humano contra computador e computador contra computador. Permite também os modos de dificuldade fácil, média ou difícil para o jogador controlado pela inteligência artificial.

Os objetivos do trabalho foram subdivididos nas seguintes metas:

-representação do estado de jogo dentro do programa bem como criação de predicados para a sua visualização (tabuleiro, número de peças na mão de cada jogador, turno atual, entre outros);

-implementação de algoritmos e predicados lógicos que permitam a manipulação do estado do jogo com movimentos de peças, capturas, inserções de novas peças, avanço de turno e deteção do fim de jogo;

-criação de uma interface para os jogadores humanos escolherem as suas jogadas e interagirem indiretamente com os predicados anteriores;

-criação de uma inteligência artificial que, analisando o estado atual do jogo e os estados consequentes escolha a jogada apropriada (que pode ou não ser a melhor jogada dependendo da dificuldade escolhida previamente).

Neste relatório serão abordados o modo de funcionamento do jogo, a arquitetura do sistema desenvolvido para a sua implementação, a implementação dos aspetos da lógica do jogo em Prolog (estado de jogo, validação de input, execução de jogadas, cálculo de jogadas válidas, avaliação da qualidade de cada jogada para a inteligência artificial, entre outros), a especificação da interface com o Utilizador e um breve sumário sobre o projeto desenvolvido. No final do relatório será mencionada a bibliografia consultada e colocado um anexo com os predicados desenvolvidos em Prolog e um link para um ficheiro que permite o *parsing* de ficheiros ".pl" pelo Notepad++.

2. O Jogo de Choko

O Choko é um jogo de tabuleiro de estratégia originário da Africa ocidental, jogado principalmente pelas tribos Mandinka e Fula.

O objetivo do jogo é capturar todas as peças do oponente. O tabuleiro tem dimensões 5x5 e cada jogador tem, inicialmente, ao seu dispor 12 peças. Os jogadores podem tomar um de dois tipos de ações: colocar uma peça numa célula do tabuleiro (caso ainda tenham peças para colocar) ou fazer um movimento. Inicialmente o tabuleiro encontra-se vazio, mas após os jogadores decidirem a ordem, o primeiro jogador coloca uma das suas peças no tabuleiro. Este jogador tem a iniciativa: sempre que o jogador que tem a iniciativa colocar uma peça no tabuleiro, o seu oponente terá também que fazer o mesmo.

Figura 1- Tabuleiro de Choko

Quando o jogador que tem a iniciativa realizar um movimento (movimento normal ou de captura), este perde a iniciativa. Se nenhum dos jogadores tiver a iniciativa o próximo jogador que colocar uma das suas peças em campo ganha a iniciativa. Após a última peça ser colocada no tabuleiro (24ª inserção de peça) o segundo jogador joga a sua vez, mesmo que tenha sido ele a colocar a última peça.

O movimento das peças é ortogonal, uma casa em qualquer das quatro direções. Ao comer uma peça do adversário, as peças comportam-se similarmente ao jogo das Damas, saltando por cima da peça que vão comer, exceto que neste caso os saltos são ortogonais em vez de diagonais (tal como no movimento normal) e os jogadores apenas podem capturar uma peça por turno. Após capturar uma peça, o jogador pode ainda remover outra peça do oponente de qualquer célula. Assim, cada captura

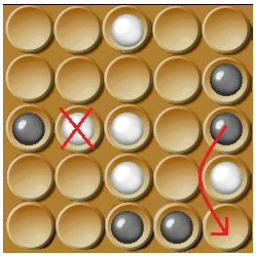

Figura 2- Exemplo de captura e remoção de peça adicional

corresponde a eliminar duas peças (desde que o jogador que perdeu a peça tenha outra para ser removida).

3. Arquitetura do Sistema

O sistema implementado está dividido em vários módulos (documentados também no código fonte do programa), com o intuito de separar as diferentes componentes do programa e tornar mais fácil a sua compreensão, testar com maior facilidade as componentes separadas e simplificar o processo de alteração do código já realizado.

Estas componentes são:

1. A criação de estruturas de dados para o estado do jogo e a sua inicialização. Optou-se por uma lista de listas para o tabuleiro e uma estrutura do tipo *player(tipo de peça, número de peças, …)* para os jogadores. Esta componente será explicada em detalhe na secção da representação do Estado do Jogo.
2. Componente de visualização dos diversos aspetos do estado de jogo (tabuleiro atual, turno atual, jogador com iniciativa, número de peças em mão, entre outros).
3. Predicados lógicos para estabelecer as regras da mecânica de jogo (movimentos, capturas, inserções de pelas) que analisam o estado do jogo e aplicam as regras conforme o estado atual o exigir (por exemplo, se o jogador não tem peças no tabuleiro, será obrigado a inserir uma peça, se não tiver peças a inserir, perde o jogo).
4. Predicados lógicos que decidem o comportamento da inteligência artificial. Este módulo faz uso do módulo *random* providenciado pelo SICStus Prolog 4.2.3 tanto para evitar jogos repetitivos (computador decide aleatoriamente entre jogadas de qualidade igual) como para inserir erros aleatórios com maior ou menor probabilidade consoante a dificuldade escolhida para o jogador controlado pelo computador. Deste modo evita-se que duas AIs, que usam a mesma estratégia, bloqueiem o jogo evitando os ataques uma da outra.
5. Predicados lógicos que permitem a interface com o utilizador, tanto para a escolha inicial do tipo de jogo como para a escolha de jogadas por parte dos jogadores humanos.

Inicialmente, é chamada uma interface que contem apenas os menus do jogo. Após o utilizador escolher o tipo de jogo, as estruturas do estado de jogo são inicializadas e o jogo passa para a interface principal que irá controlar o seu progresso. Os módulos de interface humano e de inteligência artificial são usados pela interface principal do jogo consoante o tipo de jogador que terá o seu turno. Esta interface trata também de chamar a visualização do estado de jogo a cada turno e interagir com os predicados lógicos principais que estabelecem a mecânica do Choko. Estas relações são visíveis no diagrama simplificado mostrado abaixo (Figura 3):

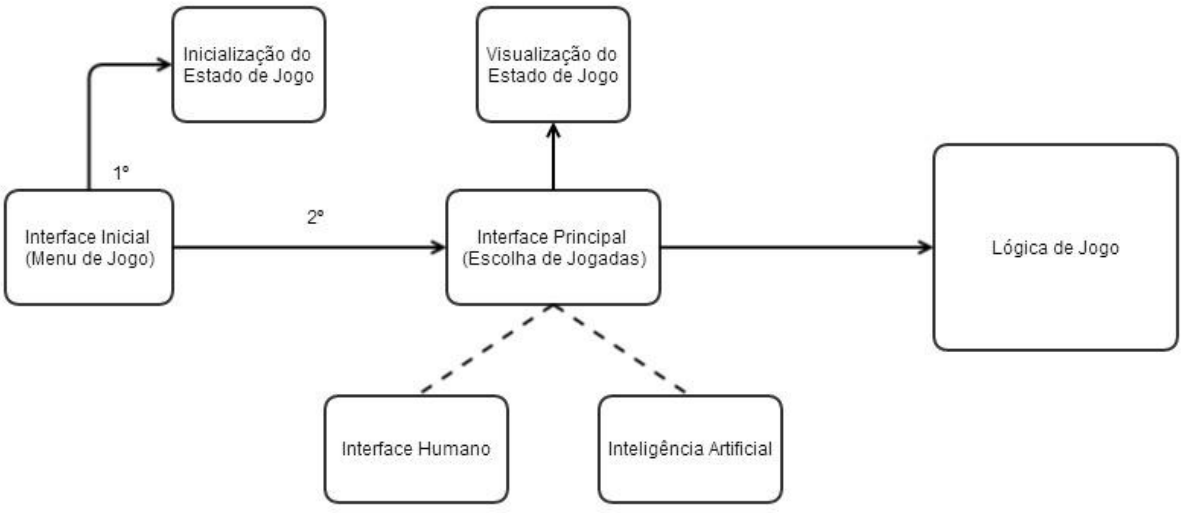

Figura 3- Diagrama da Arquitetura do Programa

Posteriormente será ainda desenvolvido um módulo para a comunicação com um visualizador 3D que será desenvolvido na cadeira de Laboratório de Aplicações com Interface Gráfica. Este módulo será usado no Interface Humano e nos menus iniciais nos pontos em que é pedido *input*, ficando à espera que a jogada seja escolhida através do visualizador em vez de utilizar *input* direto. A parte de visualização irá também usar este módulo para passar o estado de jogo ao visualizador.

Para simplificar a comunicação e impedir perdas/conflitos de informação, o visualizador tomará sempre a iniciativa da comunicação, pelo que o uso do módulo de comunicação com o visualizador implicará sempre a pausa do programa até ao visualizador estabelecer comunicação e esta ser concluída.

4. Lógica do Jogo

4.1. Representação do Estado do Jogo

Para representar o tabuleiro foi usada uma abordagem baseada numa lista de listas podendo cada elemento das sublistas ser um 'e', 'b' ou 'w' representando respetivamente uma célula vazia, uma célula com uma peça preta (*black*) e uma célula com uma peça branca (*white*). Inicialmente todas as células do tabuleiro encontram-se vazias (sendo representadas na nossa implementação pela letra 'e'):

initial_board([
[e , e , e , e , e],
[e , e , e , e , e],
[e , e , e , e , e],
[e , e , e , e , e],
[e , e , e , e , e]]).

Para representar os jogadores, optou-se por usar uma estrutura *player(tipo de peça, número de peças em mão, tipo de jogador (humano ou computador), predicado a ser usado quando este jogador tem o seu turno, dificuldade (apenas usada com jogadores do tipo computador)*).

Outros elementos do estado de jogo são armazenados em variáveis simples. Estes elementos são o turno atual (que corresponde a um tipo de peça) e o jogador que possui a iniciativa (corresponde também a um tipo de peça) (ver secção "2. O Jogo de Choko" do relatório para explicação da iniciativa).

Através dos predicados de visualização implementados (especificados na secção abaixo), o jogo é visualizado da seguinte forma:

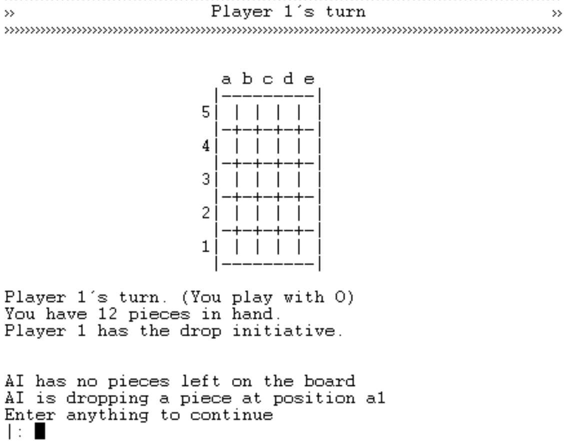

Figura 4- Estado inicial do jogo (jogador é uma AI)

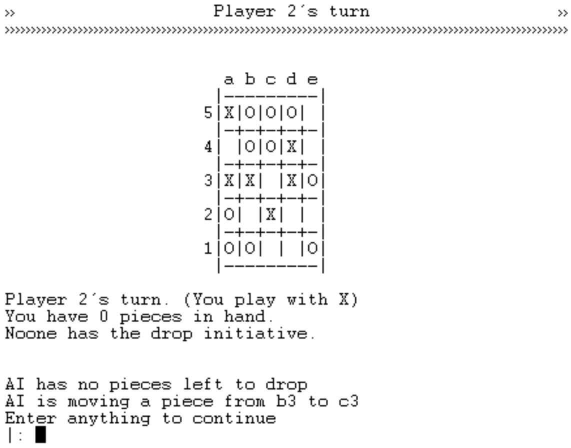

Figura 5- Estado intermédio do jogo (jogador é uma AI)

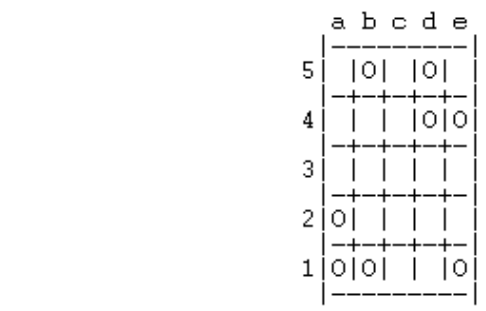

Figura 6- Estado final do jogo

4.2. Visualização do Estado do Jogo

Para visualizar o estado do jogo, foi implementado um predicado que percorre o tabuleiro (lista de listas) através do uso de predicados recursivos auxiliares, escrevendo no ecrã os símbolos correspondentes aos elementos do tabuleiro. Para a conversão dos elementos do tabuleiro nestes símbolos foram declarados os factos: *sign(b,'X')* para obter o símbolo correspondente a uma peça preta, *sign(w,'O')* para obter o símbolo correspondente a uma peça branca e *sign(e,' ')* para obter o símbolo correspondente a uma célula vazia.

Predicados de visualização do tabuleiro:

print_board(Board) – imprime o separador inicial, chama o predicado pb e imprime o separador final.

pb(Board, LineNumber) – imprime o número e a linha atual do tabuleiro, decrementa o número da linha atual e chama-se recursivamente para o resto de tabuleiro.

print_line(Line) – imprime a cabeça da linha e chama-se recursivamente para o resto da linha

Adicionalmente, implementaram-se os factos *piece_to_player/2*, que convertem um elemento do tabuleiro no descritor do jogador (w -> 'Player 1', b -> 'Player 2', e -> 'Noone'), os factos *letter_to_num/2* que permitem que o utilizador use a letra correspondente a uma coluna para a identificar, convertendo essa letra num número (a->1, b->2, … , e->5), e o predicado *show_player_status* que, recebendo uma estrutura *player/5*, especificada na representação do estado do jogo, escreve no ecrã as informações importantes relativas a esse jogador (descritor do jogador, peças que usa, peças que tem em mão).

4.3. Validação de Jogadas

A validação de jogadas escolhidas pelo utilizador é feita em duas etapas. Inicialmente, após ser pedido ao jogador que especifique uma posição no tabuleiro para inserir uma peça, ou remover uma peça do oponente após ter feito uma captura, ou duas posições para o movimento (origem-destino), as posições escolhidas são validadas pelo predicado *valid_position_parameters(ChosenC, ChosenL)* que apenas verifica se a posição escolhida pertence ao predicado. Em seguida é chamado um dos predicados de execução de jogadas (*drop_piece(Board, PosC, PosL, Piece, ResultingBoard)*, *remove_piece(Board, PosC, PosL, Piece, ResultingBoard)*, ou *move_piece(Board, InitialC, InitialL, DestC, DestL, Piece, ResultingBoard, PieceRemovalPredicate)*) consoante o tipo de jogada. Estes predicados irão falhar se a jogada não for válida (inserir uma peça numa célula não vazia, remover uma peça de uma célula onde não se encontra uma peça de um oponente ou movimentos inválidos) e o jogador receberá uma mensagem de erro e terá de escolher novamente a sua jogada.

4.4. Execução de Jogadas

A execução de jogadas para um jogador humano passa por uma etapa inicial no predicado *human_turn(Player, Board, DropInitiative, ResultingPlayer, ResultingBoard, ResultingDropInitiative)* em que será verificado se o jogador pode inserir uma peça, mover uma peça ou escolher entre estas duas opções. No último caso, será apresentado um menu para o jogador indicar se quer inserir ou mover uma peça.

Em seguida, serão executados os predicados *move_piece_human/6* ou *drop_piece_human/6* (mesmos parâmetros que o *human_turn*) que irão pedir ao jogador que insira a sua jogada, validá-la através dos predicados mencionados na secção anterior (4.3 Validação de Jogadas) e chamar os predicados de execução de jogadas *drop_piece(Board, PosC, PosL, Piece, ResultingBoard)* ou *move_piece(Board, InitialC, InitialL, DestC, DestL, Piece, ResultingBoard, PieceRemovalPredicate)*.

No caso do drop_piece, o seu funcionamento consiste apenas em inserir uma peça do tipo *Piece*, na posição (*PosC*,*PosL*) do tabuleiro armazenado em *ResultingBoard*, desde que nessa posição do tabuleiro original não se encontre nenhuma peça.

No caso do *move_piece*, o predicado tentará primeiro realizar um movimento normal através do predicado *standard_move(Board, InitialC, InitialL, DestC, DestL, Piece, ResultingBoard)* e, caso esse falhe por não se tratar de um movimento normal válido, tentará um movimento de captura usando o predicado *capture_move(Board, InitialC, InitialL, DestC, DestL, Piece, ResultingBoard, PieceRemovalPredicate)*. Caso o *capture_move* seja executado com sucesso, este executará ainda o predicado passado como parâmetro para remoção de uma peça extra do oponente, no caso de um jogador humano, este predicado será o predicado *remove_piece_after_capture_human(Board, OppPiece, ResultingBoard)* que funciona de forma semelhante ao *drop_piece_human/6* caso o adversário ainda tenha peças no tabuleiro, pedindo que o jogador escolha uma célula do tabuleiro com uma peça inimiga e chamando o predicado *remove_piece(Board, PosC, PosL, Piece, ResultingBoard)*, que irá substituir a célula especificada por uma célula vazia se a célula no tabuleiro original corresponder ao tipo de peça esperado (*Piece*).

A execução de jogadas para um jogador controlado pelo computador funciona através do uso do predicado *ai_turn(Player, Board, DropInitiative, ResultingPlayer, ResultingBoard, ResultingDropInitiative)* que para cada situação (pode inserir peças, pode mover peças, pode inserir ou mover peças) funciona de forma semelhante ao predicado *human_turn/6*, com a diferença que a escolha de jogadas não é feita pelo utilizador mas sim por predicados que definem o comportamento da *AI*. Estes predicados serão descritos posteriormente, na secção "4.8. Jogada do Computador".

Para qualquer tipo de jogada, é calculado um novo tabuleiro, novo conteúdo para estrutura do jogador e qual o jogador, se algum, que detém a iniciativa após concluída a jogada.

4.5. Lista de Jogadas Válidas

Para implementarmos uma inteligência artificial para o jogo, deparamo-nos com a necessidade de criar predicados que gerassem todas as jogadas possíveis de modo a que a AI possa escolher uma jogada adequada. Para isto, foram implementados os seguintes predicados:

- *get_drop_moves(Board,Moves)* procura recursivamente no tabuleiro todas as células vazias e guarda-as na lista *Moves* com o formato Coluna-Linha. Para percorrer o tabuleiro usa o predicado auxiliar *gdm_aux(Board, CurrentC, CurrentL, Moves)* com os valores iniciais (1,5) para (*CurrentC*,*CurrentL*) (primeira linha corresponde à linha 5 como é mostrado na secção "4.1 Representação do Estado do Jogo"). Este predicado vai atualizando os valores da coluna e linha atuais e chama-se recursivamente até o tabuleiro ter sido totalmente percorrido.

- *get_movements(Board,Turn,Movements)* percorre o tabuleiro com um predicado auxiliar (*gm_aux(Board, CurrentC, CurrentL, Moves, OriginalBoard)*) de forma semelhante ao *get_drop_moves/2* mas procura peças do tipo *Turn* em vez de células vazias. Ao encontrar uma peça, será executado o predicado *get_piece_movements(OriginalBoard, Turn, PosC, PosL, PieceMovements)* para obter todos os movimentos que essa peça pode executar.

- *get_piece_movements(OriginalBoard, Turn, PosC, PosL, PieceMovements)* tenta realizar todos os movimentos possíveis para essa peça (1 ou 2 células em todas as direções ortogonais) e guarda os resultados na lista *PieceMovements* com o formato ColunaOrigem/LinhaOrigem-ColunaDestino/LinhaDestino-ColunaRemoção/LinhaRemoção (caso não ocorra remoção, a última posição será 0/0). Recorde-se que ao ocorrer um movimento de captura, será chamado um predicado auxiliar para remover as peças. Neste caso será chamado o predicado *list_removes(Board, IniC, IniL, DestC , DestL, OppPiece, List)*, que irá colocar em *List* todas as remoções de peças possíveis, acompanhadas do movimento de captura correspondente no formato já especificado.

- *list_removes(Board, IniC, IniL, DestC , DestL, OppPiece, List)* percorre o tabuleiro à procura de peças do tipo *OppPiece* e guarda os movimentos de captura e remoção em *List* no formato já especificado. Usa o predicado recursivo auxiliar *lr_aux* para percorrer o tabuleiro.

4.6. Avaliação do Estado do Jogo

Para que a inteligência artificial possa escolher jogadas, é necessário que consiga avaliá-las de forma quantitativa de modo a poder compará-las. Para tal foi necessária a implementação de predicados que permitam a análise do estado de jogo atual e lhe atribuam um valor numérico.

Para conseguir isto implementou-se o predicado *evaluate_board(Board, Turn, Value, DropInitiative)* que para um dado tabuleiro e iniciativa (ver "2. O Jogo de Choko") calcula o valor do tabuleiro para o jogador que usa as peças do tipo *Turn*. Este predicado usa o predicado auxiliar *ev(Board, Turn, CurrentValue, PosC, PosL, OriginalBoard, FinalValue, DropInitiative)* com valores iniciais 0, 1 e 5 para os parâmetros *CurrentValue*, *PosC* e *PosL* respetivamente. O predicado *ev* irá percorrer o tabuleiro *Board* (através de chamadas recursivas), decrementando o *CurrentValue* em 5 sempre que encontra uma peça inimiga. Ao encontrar uma peça do tipo *Turn*, irá incrementar o *CurrentValue* em 5 e ainda acrescentar o valor dessa peça através do predicado *calculate_piece_value(OriginalBoard, Turn, PosC, PosL, PieceValue, DropInitiative)* que recebe o tabuleiro original (completo), uma peça, a sua posição e informação sobre quem tem a iniciativa atualmente (*DropInitiative*) e calculará o valor da peça (*PieceValue*) com base nestas informações.

O valor de uma peça é -18*N em que N é o número de peças que a podem capturar se o jogador atual não tiver a iniciativa (adversário poderá mover uma peça no próximo turno), -2*N caso tenha a iniciativa. A este valor soma-se 6*M em que M é o número de peças que esta peça pode capturar caso o inimigo não tenha iniciativa, 2*M caso tenha. (Deste modo é sempre dada prioridade a proteger as próprias peças: a AI não colocará uma peça em posição de captura se isso implicar que o inimigo pode capturar a sua peça no turno seguinte).

4.7. Final do Jogo

A verificação do final do jogo é feita através do caso base do predicado principal de jogo *play(P1,P2,Board,Turn,DropInitiative,NumPiecesDropped,Winner)*. Para realizar esta verificação, o predicado é declarado em primeiro lugar com as cláusulas *play(player(Piece,0,_,_,_), P2, Board ,_ ,_, _, P2)* e *play(P1,player(Piece,0,_,_,_),Board,_,_,_,P1)*. Estas cláusulas apenas ocorrerão se um dos jogadores já não tiver peças na mão (2º parâmetro de *player/5)* e terão se sucesso se não se verificar o predicado *has_piece(Board,Piece)*, que procura a existência de peças do tipo *Piece* no tabuleiro *Board*. Caso o predicado falhe (*\+ has_piece(Board,Piece)*), o vencedor fica o jogador 2 para o primeiro predicado, jogador 1 para o segundo.

O predicado *has_piece(Board,Piece)* funciona de forma semelhante a outros já abordados, percorrendo o tabuleiro de forma recursiva até encontrar uma peça do tipo *Piece*, parando ao encontrar essa peça.

4.8. Jogada do Computador

Ao implementar a AI, tinha-se inicialmente optado por usar um algoritmo MiniMax com níveis de profundidade. Esta escolha rapidamente revelou vários problemas, nomeadamente o aumento exponencial de computações devido a, neste jogo, cada captura ser acompanhada de todas as remoções possíveis, ou seja, estando todas as peças em jogo e sendo possível efetuar 4 capturas diferentes (valor máximo visto haver apenas 1 célula livre nesta situação) haverão posteriormente 11 remoções possíveis para cada captura, originando portanto 44 jogadas possíveis. Juntado a isto cálculos adicionais para níveis de profundidade superiores a 1, chega-se a milhares de possibilidades com nível máximo 2, sendo que para níveis superiores a 3, o tempo de computação começa a ser notavelmente longo. Diferenciando portanto as AIs em níveis de profundidade, como se tinha originalmente feito, ou a diferença de dificuldade seria praticamente negligenciável, devido a pequenas diferenças entre níveis de profundidade, ou o tempo de computação seria impraticavelmente longo. Outro problema era que duas AIs iguais ficariam presas em '*loop*' uma contra outra visto usarem as mesmas estratégias.

Tendo em conta estas dificuldades, foi decidido abandonar esta estratégia e usar apenas nível de profundidade 1 para todas as AIs e introduzir a possibilidade de escolha errada com base na geração de números aleatórios. A probabilidade de tal acontecer é o que diferencia os níveis de dificuldade. Deste modo, mesmo colocando uma AI contra outra, eventualmente uma irá perder. As diferenças de dificuldade entre as AIs são também bastante mais notáveis, sendo que o modo difícil quase nunca cometerá um erro, o modo médio irá ocasionalmente fazer uma má jogada, e o modo fácil comete erros com alguma frequência.

Para o controlo de jogadores por parte do computador foram implementados os predicados *choose_drop_AI(+Board,+Turn,-ChosenC,-ChosenL,+Difficulty,+DropInitiative)*, *choose_movement_AI(+Board,+Turn,-IniC,-IniL,-DestC,-DestL,-RemC,-RemL,+Difficulty,+DropInitiative)* e *choose_move_AI(+Board,+Turn,-ChosenMove,+Difficulty,+DropInitiative)* para quando a AI é obrigada a inserir uma peça, mover uma peça ou pode realizar tanto uma inserção como um movimento, respetivamente. O modo de funcionamento destes 3 predicados é semelhante sendo que geram todas as jogadas possíveis, a diferença está em que apenas são geradas inserções, movimentos ou ambos consoante o predicado usado (por exemplo, se a AI não tiver peças no tabuleiro, será chamado o *choose_drop_AI*, se não tiver peças em mão será chamado o *choose_movement_AI*) de modo a minimizar as computações necessárias. Todos estes predicados recorrem posteriormente ao predicado *evaluate_and_choose(+Board,+Turn,+DropMoves, +Movements, +RandomLimit, -ChosenMove, +DropInitiative)* em que *RandomLimit* é o valor limite máximo para geração de um número aleatório que, sendo igual a 1, originará uma escolha errada por parte da AI e é determinado consoante a *Difficulty* da AI (6 para *easy*, 51 para *medium* e 201 para *hard*).

O predicado *evaluate_and_choose* começa por definir a primeira *DropMove* como a melhor jogada (ou o primeiro *Movement* caso *DropMove* seja uma lista vazia) e calcula o valor do tabuleiro resultante dessa jogada. Em seguida chama o predicado auxiliar *eac_aux(Board, Turn, RemainingDrops, RemainingMovements, RandomLimit, CurrentBestMove, CurrentValue, ChosenMove, DropInitiative)* que percorre recursivamente as listas *RemainingDrops* e *RemainingMovements*, efetuando as jogadas e comparando o valor do estado de jogo resultante com o da jogada atual e

substituindo caso seja superior. Caso seja igual, seleciona aleatoriamente uma das duas jogadas usando um número gerado aleatoriamente para esse efeito.

Aos cálculos do valor do estado de jogo são acrescentados os valores inerentes de cada posição do tabuleiro caso seja efetuada uma inserção de peça (dá uma pequena prioridade aos cantos e margens) e 20 pontos caso seja efetuada um movimento de captura. Os valores inerentes das posições do tabuleiro são obtidos com o predicado *board_position_values(-Values)* que origina uma lista de listas de dimensões iguais ao tabuleiro do jogo em que cada elemento é um inteiro. Para obter o valor de uma posição é usado o predicado *get_piece(Board,PosC,PosL,Piece)* visto que as listas são percorridas da mesma forma.

5. Interface com o Utilizador

A módulo de interface com o utilizador, implementado no ficheiro *user_interface.pl* contém os vários menus chamados no início do jogo. O menu principal *menu(-Player1, -Player2)* pede ao utilizador que escolha entre 3 modos de jogo: Humano contra Humano, Humano contra Computador ou Computador contra Computador.

Ao ser escolhida a primeira opção, são inicializados dois jogadores do tipo *human* e nenhum menu adicional é chamado.

Escolhendo a opção Humano contra Computador, é chamado o menu auxiliar *menu_human_vs_ai(-Player1, -Player2)*. Neste menu o utilizador deve escolher quem joga primeiro, o jogador humano ou o computador. Após esta escolha é chamado mais um menu auxiliar, o predicado *choose_ai_difficulty(-Player, +PieceType)* em que *PieceType* tem o valor b se for o jogador 2, w se for o jogador 1 (peças pretas e brancas respetivamente).

Para a última opção, é chamado o menu *menu_ai_vs_ai(Player1, Player2)* em que apenas é pedido que o jogador escolha a dificuldade da AI que joga em primeiro lugar, e da AI que joga depois, através do predicado *choose_ai_difficulty(-Player, +PieceType)*.

No predicado *choose_ai_difficulty*, o utilizador deve escolher entre dificuldade fácil, média ou difícil.

Em todos os casos em que é apresentada uma lista de hipóteses, a lista é apresentada com as opções numeradas e o jogador deve escolher o índice da opção desejada. Uma escolha inválida leva a que ocorra uma mensagem de erro e o menu é chamado novamente.

Uma vez dentro do jogo a interface com o Utilizador processa-se da seguinte forma:

- Turno de um jogador controlado por computador: O estado de jogo é apresentado conforme especificado na secção “4.1. Representação do Estado do Jogo” deste relatório, a jogada escolhida pela AI é descrita no ecrã e é pedido que o jogador insira qualquer *input* para prosseguir (ver figura 5).

- Turno de um jogador humano: O estado do jogo é apresentado da forma anteriormente descrita e o jogador deve escolher uma jogada. Se apenas puder inserir uma peça, é pedido que especifique a posição da célula onde pretende colocar a sua peça no formato Coluna/Linha (coluna varia de ‘a’ a ‘e’ e linha de 1 a 5). Se apenas puder mover uma peça, deve especificar o movimento com o formato ColunaOrigem/LinhaOrigem-ColunaDestino/LinhaDestino, podendo no caso de se realizar uma captura ser pedido que especifique a posição de uma peça inimiga adicional a remover do tabuleiro com formato idêntico a quando o jogador insere uma peça.
  Caso o jogador possa tanto inserir como mover uma peça, é apresentado um menu que o permitirá escolher o tipo de jogada a fazer e posteriormente é pedida a especificação da jogada como mencionado acima.
  No caso de a jogada ser inválida, o jogador verá uma mensagem de erro e voltará a ser pedido que faça a escolha da jogada.

6. Conclusões e Perspetivas de Desenvolvimento

Este trabalho serviu para aprofundar os conhecimentos adquiridos ao longo das aulas teóricas e teórico-práticas através da sua aplicação num projeto que consiste na implementação em Prolog de um jogo de tabuleiro (no nosso caso, o jogo de Choko).

Durante o desenvolvimento do projeto concluímos que o aspeto mais complexo da implementação do jogo de tabuleiro Choko é a criação de uma inteligência artificial capaz de jogar este jogo de uma forma inteligente. Apesar de haverem vários algoritmos conhecidos para definir uma AI (como *MiniMax* e as suas variantes), cada jogo tem aspetos únicos que precisam de ser considerados quando se tenta atribuir um valor quantitativo a uma jogada. Havendo também o problema do custo computacional que aumenta exponencialmente com o número de jogadas a analisar e o facto de que duas AIs *MiniMax* iguais, jogando uma contra a outra, nenhuma conseguirá ganhar.

Neste caso optou-se por analisar apenas uma jogada e o estado de jogo consequente (inclui as opções de captura deixadas ao inimigo) e usar a geração de números aleatórios fornecida pelo SICStus Prolog 4.2.3 para introduzir escolhas "erradas" por parte do jogador controlado pelo computador. Sendo esta probabilidade menor ou maior consoante a dificuldade escolhida para a AI.

A implementação da mecânica do jogo decorreu rapidamente e sem que tenham surgido grandes dificuldades.

Se pudéssemos começar novamente o projeto ou tendo mais tempo para o melhorar, iríamos despender algum tempo a simplificar predicados que possam ter ficado demasiado complexos, criando mais predicados auxiliares que possam ser reutilizados ou removendo complexidade desnecessária. A maioria do tempo no entanto seria despendida a aperfeiçoar a inteligência artificial do jogo, tentando criar uma implementação baseada em *MiniMax* com níveis de profundidade elevados, necessitando portanto de otimização (por exemplo, não explorando caminhos cujos estados de jogo já atingiram valores demasiado baixos comparativamente à escolha atual) e, preferencialmente, que evitasse '*loops*' no jogo mesmo que sejam duas AIs iguais a jogar uma contra outra (por exemplo mantendo um registo das últimas N jogadas e não permitindo repetir sequências de tamanho N, sendo que N teria de ser escolhido de forma a impedir que a AI se tornasse "fraca" por desistir demasiado rapidamente).

Bibliografia

[1] Sterling, Lehon; Shapiro, Ehud. *The Art of Prolog: Advanced Programming Techniques*, The MIT Press, 1994.

[2] Russel, Stuart; Norvig, Peter. *Artificial Intelligence: A Modern Approach (Third Edition)*, Prentice Hall, 2009.

[3] NIAD&R – Distributed Artificial Intelligence and Robotics Group. Materiais de Apoio da cadeira de Programação Lógica da FEUP disponíveis no moodle da cadeira. https://moodle.up.pt/course/view.php?id=511, 2013. Online em Novembro de 2013.

[4] M. Winther. Página web sobre jogos de tabuleiro. http://hem.passagen.se/melki9/choko.htm, 2006. Online em Novembro 2013.

A. Sintaxe Prolog em *Notepad++*

No seguinte link encontra-se um ficheiro .xml para configurar uma sintaxe de Prolog para o programa *Notepad++*, usado para o desenvolvimento deste projeto: http://gprolog.univ-paris1.fr/userDefineLang-GNU_Prolog.xml (online a 8 de Novembro de 2013). Para a sua instalação basta correr a versão mais recente de *Notepad++* e no menu "Language" escolher "Define your own language" e em seguida carregar em "Import" e escolher o ficheiro .xml descarregado. Após carregar o ficheiro, deve-se fechar e voltar a abrir o *Notepad++* e os ficheiros com extensão ".pl" e ".pro" devem ser reconhecidos automaticamente como ficheiros Prolog (caso contrário, selecione "Prolog (GNU)" no menu "Languages").

Instruções disponíveis também em http://gprolog.univ-paris1.fr/ (online a 8 de Novembro de 2013).

B. Predicados Lógicos Desenvolvidos

Segue em anexo o conteúdo dos ficheiros ".pl" criados para a implementação desenvolvida do jogo de Choko. Recomenda-se que sejam lidos os ficheiros diretamente em *Notepad++* com a linguagem "Prolog (GNU)", mencionada no anexo anterior, por questões de legibilidade.

main.pl:

```
%this file contains predicates used both for printing the board and converting board elements into the symbols used to represent each of the elements
%it also contains predicates to print other aspects of the game's status such as current turn, player info, etc.
?- ensure_loaded('status_printing.pl').
%this file contains predicates used to interact with the user and human players
?- ensure_loaded('user_interface.pl').
%this file contains predicates used to generate the AI's decisions
?- ensure_loaded('choko_ai.pl').


%----------------------------------------------------SECTION: PREDICATES TO INITIALIZE GAME VARIABLES-----------------------------------------------------
%this sections contains several predicates used to initialize game variables such as the board and players

%used to create a board with only empty cells (initial situation)
initial_board([
	[e,e,e,e,e],
	[e,e,e,e,e],
	[e,e,e,e,e],
	[e,e,e,e,e],
	[e,e,e,e,e]]).

%initializes a human player of type player(Piece,NumPiecesInHand,TypeOfPlayer,TurnPredicate,Difficulty) with pieces of type Piece, 12 pieces in hand and the specified type (human, computer)
init_player(Type,Piece,P):-
	Type = human,
	P = player(Piece,12,Type,human_turn,_).
init_player(Type,Piece,Difficulty,P):-
	Type = comp,
	P = player(Piece,12,Type,ai_turn,Difficulty).
```

```
%initializes the players for a game of human vs human.
init_humans(P1, P2):-
	init_player(human,w,P1),
	init_player(human,b,P2).

%initializes an AI player
init_ai_player(Difficulty,Piece,P):-
	init_player(comp,Piece,Difficulty,P).
%-----------------------------------------------------------------END OF SECTION-----------------------------------
-------------------------------


%-----------------------------------------------------------SECTION: GAME INTERFACE------------------------
------------------------------------
%this section contains several predicates used as the game interface including the 'main' predicate
play/0.

%initializes the board, calls the initial menu to set up the players (human or AI (and AI's difficulty)
and starts the game using play/7
%when the game is over it also declares the winner
play:-
	initial_board(B),
	menu(P1,P2),
	!,
	%init_humans(P1,P2),
	player(DropInitiative,_,_,_,_) = P1,
	play(P1,P2,B,DropInitiative,DropInitiative,0,Winner),
	player(WinPiece,_,_,_,_) = Winner,
	piece_to_player(WinPiece,Win),
	write(Win),write(' is Victorious!'),nl.

%checks condition for game over
	%P1 has no pieces in hand or on the board, P2 wins
play(player(Piece,0,_,_,_),P2,Board,_,_,_,P2):-
	\+ has_piece(Board,Piece),!,
	print_board(Board).
	%P2 has no pieces in hand or on the board, P1 wins
play(P1,player(Piece,0,_,_,_),Board,_,_,_,P1):-
	\+ has_piece(Board,Piece),!,
	print_board(Board).
%the last piece (24th piece) has just been dropped, P2 has his turn
play(P1,P2,Board,_,DropInitiative,24,Winner):-
	NewInitiative = e,
	player(Turn,_,_,TurnPred,_) = P2, Pred =..
[TurnPred,P2,Board,NewInitiative,NewP2,NewBoard,NewInitiative], NewP1 = P1,
	write('»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»'),nl,
	write('»All pieces have been dropped, player 2 has his turn.»'),nl,
	write('»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»'),nl,
	print_board(Board),
	show_player_status(P2,DropInitiative),
	Pred,
	NewNumPiecesDropped is 25, %prevents this predicate from ever happening again
	get_opponent_piece(Turn,NextTurn),
	play(NewP1,NewP2,NewBoard,NextTurn,NewInitiative,NewNumPiecesDropped,Winner),
	!.
%normal turn occurring, active player is chosen and the turn predicate is created
%player information is printed on the screen along with the board and the turn predicate is
executed
```

```
%another play predicate is called at the end with the resulting board, initiative, players, active turn
and number of pieces dropped
play(P1,P2,Board,Turn,DropInitiative,NumPiecesDropped,Winner):-
	(player(Turn,_,_,TurnPred,_) = P1, Pred =..
[TurnPred,P1,Board,DropInitiative,NewP1,NewBoard,NewInitiative], NewP2 = P2, Player = P1;
	player(Turn,_,_,TurnPred,_) = P2, Pred =..
[TurnPred,P2,Board,DropInitiative,NewP2,NewBoard,NewInitiative], NewP1 = P1, Player = P2), !,
	piece_to_player(Turn,PText),
	write('»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»'),nl,
	write('»                ’),write(PText),write('´s turn               »'),nl,
	write('»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»'),nl,
	print_board(Board),
	show_player_status(Player,DropInitiative),
	Pred,
	%----calculating total number of pieces dropped----
	player(_,IniPieces1,_,_,_) = P1, player(_,IniPieces2,_,_,_) = P2,
	player(_,FinalPieces1,_,_,_) = NewP1, player(_,FinalPieces2,_,_,_) = NewP2,
	PiecesDropped is IniPieces1-FinalPieces1+IniPieces2-FinalPieces2,
	NewNumPiecesDropped is NumPiecesDropped+PiecesDropped,
	%--------------------calculated--------------------
	get_opponent_piece(Turn,NextTurn),
	play(NewP1,NewP2,NewBoard,NextTurn,NewInitiative,NewNumPiecesDropped,Winner).
%-----------------------------------------------------------------END OF SECTION-----------------------------------
-------------------------------

%----------------------------------------------------SECTION: CHOKO'S CORE LOGIC PREDICATES------------
--------------------------------------------
%this section contains the predicates that handle the game's core logic functions

%gets the piece type of the opponent's piece
get_opponent_piece(w,b).
get_opponent_piece(b,w).

%drops a piece of type Piece on the Board, in the specified position (if and only if that position is
vacant)
%the result is stored in ResultingBoard
%(Piece type cannot be an empty cell (e))
drop_piece(Board, PosC, PosL, Piece, ResultingBoard):-
	Piece \= e,
	dp(Board, PosC, PosL, Piece, ResultingBoard).
%searches the board for the specified position and checks that it is empty, replacing the
corresponding position on the resulting board by Piece
%end case is (1,5) and not (1,1) since the board's lines are numbered from 5 to 1 (decreasing for
every line checked)
dp([[e|Rs]|Bs],1,5,Piece,[[Piece|Rs]|Bs]).
dp([[R|Rs]|Bs], PosC, 5, Piece, [[R|RBs]|Bs]) :-
	PosC > 1,
	C1 is PosC-1,
	dp([Rs|Bs],C1,5,Piece,[RBs|Bs]).
dp([B|Bs], PosC, PosL, Piece, [B|RBs]):-
	PosL<5,
	L1 is PosL+1,
	dp(Bs,PosC,L1,Piece,RBs).
```

```
%removes a piece from the board if the piece in the specified position corresponds to the type of
piece to be removed
%(Piece type cannot be an empty cell (e) )
remove_piece(Board, PosC, PosL, Piece, ResultingBoard):-
        Piece \= e,
        letter_to_num(PosC,CNumber),
        rp(Board, CNumber, PosL, Piece, ResultingBoard).
%functions int the same way as the predicate dp/5 except this one verifies that the specified
position in the board corresponds to the expected Piece
%and replaces it by an empty cell on the resulting board
rp([[Piece|Rs]|Bs],1,5,Piece,[[e|Rs]|Bs]).
rp([[R|Rs]|Bs], PosC, 5, Piece, [[R|RBs]|Bs]) :-
        PosC > 1,
        C1 is PosC-1,
        rp([Rs|Bs],C1,5,Piece,[RBs|Bs]).
rp([B|Bs], PosC, PosL, Piece, [B|RBs]):-
        PosL<5,
        L1 is PosL+1,
        rp(Bs,PosC,L1,Piece,RBs).
%----------------------------------------------------------------------END OF SECTION----------
----------------------------------


%-------------------------------------------------------------SECTION: MOVEMENT PREDICATES----------
-----------------------------------------
%contains predicates that allow piece movements to occurr (both standard and capture
movements)
%its essentially a specialized extension to the core logic section

%attempts to move a piece from (InitialC,InitialL) to (DestC,DestL) by checking if it is a valid
standard move or a valid capture move
%it stores the type of move that occured in TypeOfMove
move_piece(Board, InitialC, InitialL, DestC, DestL, Piece, ResultingBoard, PieceRemovalPredicate) :-
        InitialC > 0, InitialC < 6, InitialL > 0, InitialL < 6,
        DestC > 0, DestC < 6, DestL > 0, DestL < 6,
        (standard_move(Board, InitialC, InitialL, DestC, DestL, Piece, ResultingBoard);
        capture_move(Board, InitialC, InitialL, DestC, DestL, Piece,
ResultingBoard,PieceRemovalPredicate)).

%checks if a given move is a valid standard move (Piece is in the initial position, moves only one
space and final position is an empty cell)
%resulting board is storing in the variable ResultingBoard
standard_move(Board,InitialC, InitialL, DestC, DestL, Piece, ResultingBoard):-
        DiffC is DestC-InitialC,
        DiffL is DestL-InitialL,
        (1 =:= abs(DiffC), 0 =:= DiffL;
        1 =:= abs(DiffL), 0 =:= DiffC),
        rp(Board,InitialC,InitialL,Piece,NB2),
        dp(NB2,DestC,DestL,Piece,ResultingBoard).
```

```
%checks if a given move is a valid capture move
%(Piece is in the initial position, moves exactly two spaces (in a straight line), final position is an empty cell and the cell inbetween contains a piece of the opponent's type)
%resulting board is storing in the variable ResultingBoard
%the PieceRemovalPredicate is called after the capture move occurs successfully
capture_move(Board, InitialC, InitialL, DestC, DestL, Piece, ResultingBoard,PieceRemovalPredicate):-
        valid_capture_move_positions(InitialC, InitialL, DestC, DestL, MedC, MedL),
        rp(Board, InitialC, InitialL, Piece, TempB),
        get_opponent_piece(Piece, OppPiece),
        rp(TempB, MedC, MedL, OppPiece, TempB2),
        dp(TempB2,DestC, DestL,Piece,ResultingBoard),
        PieceRemovalPredicate.


%checks if the given positions are valid for a capture movement (moving either 2 lines or 2 columns)
%calculates the position that the moving piece will jump over (position (MedC,MedL))
valid_capture_move_positions(InitialC, InitialL, DestC, DestL, MedC, MedL):- %case where the piece moves two lines (up or down)
        InitialC > 0, InitialC < 6, InitialL > 0, InitialL < 6,
        DestC > 0, DestC < 6, DestL > 0, DestL < 6,
        InitialC =:= DestC,
        MedC is InitialC,
        (DestL =:= InitialL+2, MedL is InitialL+1;
        DestL =:= InitialL-2, MedL is InitialL-1).
valid_capture_move_positions(InitialC, InitialL, DestC, DestL, MedC, MedL):- %case where the piece moves two columns (left or right)
        InitialC > 0, InitialC < 6, InitialL > 0, InitialL < 6,
        DestC > 0, DestC < 6, DestL > 0, DestL < 6,
        InitialL =:= DestL,
        MedL is InitialL,
        (DestC =:= InitialC+2, MedC is InitialC+1;
        DestC =:= InitialC-2, MedC is InitialC-1).
%----------------------------------------------------------------------END OF SECTION----------------------------------------------------------------------


%----------------------------------------------------SECTION: BOARD HANDLING AUXILIARY FUNCTIONS----------------------------------------------------
%this section contains predicates useful for checking the board status (which kind of pieces remain, which piece is in which cell, etc)

%retrieves a piece from the specified board in the position indicated py PosC (column number) and PosL (line number)
%can also be used to check if a given piece is in the specified board in the indicated position
get_piece(Board,PosC,PosL,Piece):-
        get_line(Board,PosL,Line),
        get_line_element(Line,PosC,Piece).
%retrieves a line from the board (remmember: lines are numbered from 5 to 1, therefore line 5 would correspond to the first line)
%uses a counter to check when the desired line has been reached (increments the counter until it reaches 5, moving on to the following line with each increment)
get_line([Line|_],5,Line):- !.
get_line([_|Bs],L,Line):-
        L < 5,
        L1 is L+1,
        get_line(Bs,L1,Line).
```

```
%retrieves an element from a line
%uses a counter to check when the desired line has been reached (decreases the counter until it reaches 1, moving on to the following element with each decrement)
get_line_element([Piece|_], 1, Piece):- !.
get_line_element([_|Ls],N,Piece):-
        N > 1,
        N1 is N-1,
        get_line_element(Ls,N1,Piece).

%checks if a given board still has pieces of the type piece
%it starts by checking if a line has that piece and if it does not, it moves on to the next line of the board
has_piece([B|Bs],Piece):-
        (line_has_piece(B,Piece);
        has_piece(Bs,Piece)).
%checks if a given line contains a piece of the the type Piece
%stops upon finding that type of piece or after having checked all elements of the line
line_has_piece([Piece|_],Piece):- !.
line_has_piece([_|Rs],Piece):- line_has_piece(Rs,Piece).

%checks if a player has any piece on the board which he can move
player_can_move(Board, Piece):-
        pcm_aux(Board, 1, 5, Piece, Board), !.

%iterates through the board checking for every piece of type Piece if that can move. If it can, the predicate is successfull
pcm_aux([[]|Rb],_, PosL, Piece, CompleteBoard):-
        NewL is PosL-1,
        NewC is 1,
        pcm_aux(Rb, NewC, NewL, Piece, CompleteBoard).
pcm_aux([[Piece|Ls]|Rb],PosC, PosL, Piece, CompleteBoard):-
        (piece_can_move(CompleteBoard, PosC, PosL, Piece);
        NewC is PosC+1, pcm_aux([Ls|Rb],NewC, PosL, Piece, CompleteBoard)).
pcm_aux([[_|Ls]|Rb], PosC, PosL, Piece, CompleteBoard):-
        NewC is PosC+1, pcm_aux([Ls|Rb],NewC, PosL, Piece, CompleteBoard).

%stub for removing pieces from the board. to be used only when you dont actually want to let the capture_move remove a piece from the board
fake_remove.
%checks if the expected piece is in the specified position of the board and if it can move
piece_can_move(Board, PosC, PosL, Piece):-
        (DestC is PosC+1, DestL is PosL; DestC is PosC-1, DestL is PosL; DestC is PosC+2, DestL is PosL; DestC is PosC-2, DestL is PosL;
        DestL is PosL+1, DestC is PosC; DestL is PosL-1, DestC is PosC; DestL is PosL+2, DestC is PosC; DestL is PosL-2, DestC is PosC),
        move_piece(Board, PosC, PosL, DestC, DestL, Piece, _, fake_remove),
        !.

%----------------------------------------------------------------END OF SECTION----------------------------------------------------------------
```

status_printing.pl:

```
%----------------------------------------------------SECTION: PREDICATES FOR PRINTING THE BOARD----------------------------------------------------
%this section of the code contains predicates used both for printing the board and converting board elements into the symbols used to represent each
%of the elements

%converts the piece type into a corresponding symbol to be displayed on screen
sign( b, 'X'). %b indicates a black piece type and is represented on screen by an X
sign( w, 'O'). %w indicates a white piece type and is represented on screen by an O
sign( e, ' '). %e indicates an empty cell (absence of piece). When used as a value to represent the drop initiative, it indicates no player currently has the initiative.

%prints the column indexes and the top and bottom borders, calling an auxiliary recursive function in-between to print the contents of the board
print_board(B):-
        nl,nl,
        print('     a b c d e  '),nl,
        print('    |---------|'),nl,
        length(B,N),
        pb(B,N),
        print('    |---------|'),nl, nl.


%recursively prints each line of the board followed by a line separator until there is only one element left (in this case no separator is printed).
pb([B], N) :-
        print('  '),
        print(N),
        print_line(B), !.
pb([B|Bs], N) :-
        print('  '),
        print(N),
        print_line(B),
        print('    |-+-+-+-+-|'),nl,
        N1 is N-1,
        pb(Bs,N1).
%recursively prints a vertical separator and the on-screen symbol corresponding to the piece of the current line element.
%When the line has been fully printed, an additional vertical separator is printed (right border of the board).
print_line([]) :-
        print('|'),nl.
print_line([L|Ls]):-
        sign(L,Symbol),
        print('|'),
        print(Symbol),
        print_line(Ls).
%----------------------------------------------------------------END OF SECTION----------------------------------------------------------------


%-----------------------------------------------SECTION: AUXILIARY PREDICATES FOR VISUALIZING GAME STATUS-----------------------------------------------
%this section contains predicates used to validate user input
%it also contains predicates used to convert several game elements into 'human friendly' descriptions that will be printed on screen
```

```
%these predicates do not interact directly with the user (they don't read user input)

%gets the player 'name' corresponding to a piece type (used to represent turns)
piece_to_player(w,'Player 1').
piece_to_player(b,'Player 2').
piece_to_player(e,'Noone').

%converts letters to the corresponding column numbers
letter_to_num(a,1).
letter_to_num(b,2).
letter_to_num(c,3).
letter_to_num(d,4).
letter_to_num(e,5).


%prints the status of the specified player on screen (used when a player has to take an action)
show_player_status(Player):-
	player(Turn,NumPieces,_,_,_) = Player,
	piece_to_player(Turn,PText),
	sign(Turn,Symbol),
	write(PText), write('´s turn.'), write(' (You play with '), write(Symbol), write(')'),
	nl,
	write('You have '),write(NumPieces), write(' pieces in hand.'),nl.
%prints the status of the specified player on screen (used at the beginning of his turn) as well as
who has the drop initiative
show_player_status(Player, DropInitiative):-
	player(Turn,NumPieces,_,_,_) = Player,
	piece_to_player(Turn,PText),
	sign(Turn,Symbol),
	piece_to_player(DropInitiative,PInitiative),
	write(PText), write('´s turn.'), write(' (You play with '), write(Symbol), write(')'),
	nl,
	write('You have '),write(NumPieces), write(' pieces in hand.'),nl,
	write(PInitiative),write(' has the drop initiative.'),nl, nl.
%----------------------------------------------------------------------END OF SECTION----------------------------------
---------------------------------
```

user_interface.pl:

```
%-----------------------------------------------------------SECTION: HUMAN INTERFACE - MENUS----------------
-------------------------------------------
%this section contains the game menus used when the game is initialized

%allows the user to choose the game type (human vs human, human vs ai or ai vs ai) and calls the
apropriate menu for their choice
menu(P1,P2):-
	write('Choose game type:'),nl,nl,
	write('1- Human vs Human'),nl,
	write('2- Human vs AI'),nl,
	write('3- AI vs AI'),nl,
	(get_user_choice(1,3,Choice),
		(Choice =:= 1, init_humans(P1,P2); %no menu needed since there are no
difficulties or player orders to choose
		Choice =:= 2, menu_human_vs_ai(P1,P2);
		Choice =:= 3, menu_ai_vs_ai(P1,P2));
	nl,write('Invalid choice!'),nl,nl,menu(P1,P2)).
```

```
%allows the user to choose who goes first (human or AI) and choose the AI's difficulty mode
menu_human_vs_ai(P1, P2):-
	nl,write('Choose player order:'),nl,nl,
	write('1- Human goes first'),nl,
	write('2- AI goes first'),nl,
	(get_user_choice(1,2,Choice),
		(Choice =:= 1, init_player(human,w,P1), choose_ai_difficulty(P2,b);
		Choice =:= 2, init_player(human,b,P2), choose_ai_difficulty(P1,w));
	nl,write('Invalid choice!'),nl,nl,menu_human_vs_ai(P1,P2)).

%asks the user to choose the difficulty for the AI players
menu_ai_vs_ai(P1,P2):-
	nl,write('Player 1:'),nl,
	choose_ai_difficulty(P1,w),
	nl,write('Player 2:'),nl,
	choose_ai_difficulty(P2,b).

%asks the user to choose a difficulty mode for the AI (easy, medium or hard) and initializes an AI
player with that difficulty and the specified piece type
choose_ai_difficulty(AIplayer,PieceType):-
	write('AI difficulty level:'),nl,nl,
	write('1- easy'),nl,
	write('2- medium'),nl,
	write('3- hard'),nl,
	(get_user_choice(1,3,Choice),
		(Choice =:= 1, init_ai_player(easy,PieceType, AIplayer);
		Choice =:= 2, init_ai_player(medium,PieceType, AIplayer);
		Choice =:= 3, init_ai_player(hard,PieceType, AIplayer));
	nl,write('Invalid choice!'),nl,nl,choose_ai_difficulty(AIplayer,PieceType)).

%----------------------------------------------------------------------END OF SECTION----------------------------------
---------------------------------

%-------------------------------------------------------------SECTION: HUMAN INTERFACE-------------------------
---------------------------------------
%this section contains interface predicates specific to human players

%checks if a given position parameters are valid and belong to the board
valid_position_parameters(ChosenC,ChosenL):-
	integer(ChosenL),
	nonvar(ChosenC),
	letter_to_num(ChosenC,_),
	ChosenL > 0, ChosenL < 6.

%waits for user input before continuing
wait_for_input:-
	write('Enter anything to continue'),nl,
	read(_),
	!.
%reads user input corresponding to a single cell position and validates it
read_and_validate_position(ChosenC/ChosenL):-
	read(ChosenC/ChosenL),
	valid_position_parameters(ChosenC,ChosenL).
%reads user input corresponding to two cell positions and validates them
read_and_validate_position(IniC/IniL,DestC/DestL):-
	read(IniC/IniL-DestC/DestL),
	valid_position_parameters(IniC,IniL),
	valid_position_parameters(DestC,DestL).
```

```
%reads and validates user input corresponding to a choice between two integers
get_user_choice(ChoiceMin,ChoiceMax, Input):-
	read(Input),
	integer(Input),
	Input >= ChoiceMin,
	Input =< ChoiceMax.

%in this case, the player is forced to pass his turn due to having no moves available and no pieces to drop
human_turn(Player,Board,DropInitiative,Player,Board,DropInitiative):-
	player(Turn,0,_,_,_) = Player,
	\+ player_can_move(Board, Turn),
	print_board(Board),nl,write('No moves available!'),nl,
	wait_for_input,
	!.
%in this case, the player is forced to move a piece due to having none left to drop (if there were also none on the board the game would be over already)
human_turn(Player,Board,DropInitiative,ResultingPlayer,ResultingBoard,ResultingDropInitiative):-
	player(Turn,0,_,_,_) = Player,
	nl,write('No pieces left to drop'),nl,
	(move_piece_human(Player,Board,Turn,ResultingBoard,ResultingPlayer, DropInitiative, ResultingDropInitiative);
	print_board(Board),write('Invalid move!'),nl,
	human_turn(Player,Board,DropInitiative,ResultingPlayer,ResultingBoard,ResultingDropInitiative)),
	!.
%in this case, the player is forced to drop a piece either due to lack of pieces on the board, opponent having initiative or all pieces being blocked
human_turn(Player,Board,DropInitiative,ResultingPlayer,ResultingBoard,ResultingDropInitiative):-
	player(Turn,_,_,_,_) = Player,
	(get_opponent_piece(Turn,DropInitiative), nl, write('Opponent has the drop initiative'), nl;
	\+ has_piece(Board,Turn), nl, write('You have no pieces left on the board'), nl;
	\+ player_can_move(Board, Turn),nl,write('All your pieces are blocked!'),nl),
	!,
	(drop_piece_human(Player,Board,Turn,ResultingBoard,ResultingPlayer, DropInitiative, ResultingDropInitiative);
	print_board(Board),write('You can not drop your piece there!'),nl,
	human_turn(Player,Board,DropInitiative,ResultingPlayer,ResultingBoard,ResultingDropInitiative)),
	!.
```

```
%in this case, the player can choose to drop a piece or perform a move
human_turn(Player,Board,DropInitiative,ResultingPlayer,ResultingBoard,ResultingDropInitiative):-
	player(Turn,_,_,_,_) = Player,
	write('1- Drop piece'),nl,write('2- Move piece'),nl,
	(get_user_choice(1,2,Input),
		(Input = 1,!,
		(print_board(Board),drop_piece_human(Player,Board,Turn,ResultingBoard,Resulti
		ngPlayer, DropInitiative, ResultingDropInitiative);
		print_board(Board),write('You can not drop your piece there!'),nl,
		human_turn(Player,Board,DropInitiative,ResultingPlayer,ResultingBoard,
		ResultingDropInitiative));
		Input = 2,!,
		(print_board(Board),move_piece_human(Player,Board,Turn,ResultingBoard,Result
		ingPlayer, DropInitiative, ResultingDropInitiative);
		print_board(Board),write('Invalid move!'),nl,
		human_turn(Player,Board,DropInitiative,ResultingPlayer,ResultingBoard,Resulting
		DropInitiative)));
	write('Invalid choice!'),
	nl, !,human_turn(Player,Board,DropInitiative,ResultingPlayer,ResultingBoard,ResultingDro
	pInitiative)),
	!.

%asks the player to choose where to drop the piece and attempts to execute the drop
move_piece_human(Player,Board,Turn,ResultingBoard,Player, DropInitiative,
ResultingDropInitiative):-
	player(Turn,_,_,_,_) = Player,
	get_opponent_piece(Turn,OppPiece),
	write('Choose movement (IniColumn/IniRow-DestColumn/DestRow)'),nl,
	read_and_validate_position(IniC/IniL,DestC/DestL),letter_to_num(IniC,COrig),letter_to_nu
m(DestC,CFinal),
	move_piece(Board,COrig,IniL,CFinal,DestL,Turn,TempBoard,remove_piece_after_capture_h
uman(TempBoard,OppPiece,TempBoard2)),
	!,
	(var(TempBoard2),ResultingBoard = TempBoard;
	ResultingBoard = TempBoard2),
	(DropInitiative = Turn, ResultingDropInitiative = e;
	ResultingDropInitiative = DropInitiative).

%asks the player to choose where to drop the piece and attempts to execute the drop
drop_piece_human(Player,Board,Turn,ResultingBoard,ResultingPlayer, DropInitiative,
ResultingDropInitiative):-
	player(Turn,NumPieces,_,Pred,_) = Player,
	write('Choose location to drop piece in (Column/Row)'),nl,
	read_and_validate_position(ChosenC/ChosenL),letter_to_num(ChosenC,CPos),drop_piece(B
oard,CPos,ChosenL,Turn,ResultingBoard),
	!,
	NewNumPieces is NumPieces-1,
	ResultingPlayer = player(Turn,NewNumPieces,human,Pred,_),
	(DropInitiative = e, ResultingDropInitiative = Turn;
	ResultingDropInitiative = DropInitiative).

%asks the player which piece to remove after the capture in case there are any available to remove
	%opponent has no pieces on the board. nothing happens
remove_piece_after_capture_human(Board,OppPiece,Board):-
	\+ has_piece(Board,OppPiece), !.
```

```
	%user is asked to choose a piece to remove (an invalid choice originates an error message
and the predicate is called again)
remove_piece_after_capture_human(Board,OppPiece,ResultingBoard):-
	print_board(Board),
	get_opponent_piece(OppPiece,Turn),
	piece_to_player(Turn, PText), sign(Turn, Symbol),
	write(PText), write(' (You play with '), write(Symbol), write(')'),nl,
	write('Choose an opponent´s piece to remove (Column/Row) '),nl,
	(read_and_validate_position(ChosenC/ChosenL),letter_to_num(ChosenC,CNum),rp(Board,C
Num,ChosenL,OppPiece,ResultingBoard);
	print_board(Board),write('You can´t remove anything from
there!'),nl,remove_piece_after_capture_human(Board,OppPiece,ResultingBoard)).
%------------------------------------------------------------------END OF SECTION----------------------------------
--------------------------------
```

choko_ai.pl:

```
%the random module is used to insert random errors into the AIs computations (to prevent them
from getting stuck in an infinite loop against eachother
%it is also used when the AI has to choose between two moves with the same worth (to prevent
repetitive games)
?- use_module(library(random)).

%------------------------------------------------------------------SECTION: AI INTERFACE-----------------------
---------------------------------
%this section contains predicates to handle an AI player's moves.

%in this case, the AI is forced to pass his turn due to having no moves available and no pieces to
drop
ai_turn(Player,Board,DropInitiative,Player,Board,DropInitiative):-
	player(Turn,0,_,_,_) = Player,
	\+ player_can_move(Board, Turn),
	print_board(Board),nl,write('AI has no moves available!'),nl,
	wait_for_input,
	!.
%in this case, the AI is forced to move a piece due to having none left to drop (if there were also
none on the board the game would be over already)
ai_turn(Player,Board,DropInitiative,ResultingPlayer,ResultingBoard,ResultingDropInitiative):-
	player(Turn,0,_,_,_) = Player,
	nl,write('AI has no pieces left to drop'),nl,
	move_piece_AI(Player,Board,Turn,ResultingBoard,ResultingPlayer, DropInitiative,
ResultingDropInitiative),
	!.
%in this case, the AI is forced to drop a piece either due to lack of pieces on the board, opponent
having initiative or all pieces being blocked
ai_turn(Player,Board,DropInitiative,ResultingPlayer,ResultingBoard,ResultingDropInitiative):-
	player(Turn,_,_,_,_) = Player,
	(get_opponent_piece(Turn,DropInitiative), nl, write('Opponent has the drop initiative'), nl;
	\+ has_piece(Board,Turn), nl, write('AI has no pieces left on the board'), nl;
	\+ player_can_move(Board, Turn),nl,write('All of the AI´s pieces are blocked!'),nl),
	drop_piece_AI(Player,Board,Turn,ResultingBoard,ResultingPlayer, DropInitiative,
ResultingDropInitiative),
	!.
%in this case, the AI can drop a piece or perform a move
ai_turn(Player,Board,DropInitiative,ResultingPlayer,ResultingBoard,ResultingDropInitiative):-
	player(Turn,_,_,_,_) = Player,
	perform_move_AI(Player,Board,Turn,ResultingBoard,ResultingPlayer, DropInitiative,
ResultingDropInitiative),
	!.
```

```
%the AI calculates all possible drops and chooses the best option (or not deppending on difficulty
and 'luck')
drop_piece_AI(Player,Board,Turn,ResultingBoard,ResultingPlayer, DropInitiative,
ResultingDropInitiative):-
        player(Turn,NumPieces,_,Pred,Difficulty) = Player,
        choose_drop_AI(Board,Turn,ChosenC,ChosenL,Difficulty,DropInitiative),
        letter_to_num(Col, ChosenC),
        write('AI is dropping a piece at position '), write(Col), write(ChosenL), nl,
        wait_for_input,
        drop_piece(Board,ChosenC,ChosenL,Turn,ResultingBoard),
        !,
        NewNumPieces is NumPieces-1,
        ResultingPlayer = player(Turn,NewNumPieces,comp,Pred,Difficulty),
        (DropInitiative = e, ResultingDropInitiative = Turn;
        ResultingDropInitiative = DropInitiative).

%the AI calculates all possible piece movements and chooses the best option (or not deppending on
difficulty and 'luck')
move_piece_AI(Player,Board,Turn,ResultingBoard,Player, DropInitiative, ResultingDropInitiative):-
        player(Turn,_,_,_,Difficulty) = Player,
        get_opponent_piece(Turn,OppPiece),
        choose_movement_AI(Board,Turn,IniC,IniL,DestC,DestL,RemC,RemL,Difficulty,DropInitiati
ve),
        letter_to_num(ColI, IniC),letter_to_num(ColD, DestC),
        write('AI is moving a piece from '), write(ColI),write(IniL),write(' to '),
write(ColD),write(DestL),
        (letter_to_num(ColR, RemC), write(' and removing opponent´s piece from '), write(ColR),
write(RemL), nl;
        nl),
        wait_for_input,
        move_piece(Board,IniC,IniL,DestC,DestL,Turn,TempBoard,ai_remove(TempBoard,OppPiec
e,RemC,RemL,TempBoard2)),
        !,
        (var(TempBoard2),ResultingBoard = TempBoard;
        ResultingBoard = TempBoard2),
        (DropInitiative = Turn, ResultingDropInitiative = e;
        ResultingDropInitiative = DropInitiative).

%the AI calculates all drops and possible piece movements and chooses the best option (or not
deppending on difficulty and 'luck')
perform_move_AI(Player,Board,Turn,ResultingBoard,ResultingPlayer, DropInitiative,
ResultingDropInitiative):-
        player(Turn,NumPieces,_,Pred,Difficulty) = Player,
        get_opponent_piece(Turn,OppPiece),
        choose_move_AI(Board,Turn,ChosenMove,Difficulty,DropInitiative),
        (ChosenMove = IniC/IniL-DestC/DestL-RemC/RemL,
                %movement action occurred
                letter_to_num(ColI, IniC),letter_to_num(ColD, DestC),
                write('AI is moving a piece from '), write(ColI),write(IniL),write(' to '),
write(ColD),write(DestL),
                (letter_to_num(ColR, RemC), write(' and removing opponent´s piece from '),
write(ColR), write(RemL), nl;
                nl),
                wait_for_input,

        move_piece(Board,IniC,IniL,DestC,DestL,Turn,TempBoard,ai_remove(TempBoard,OppPiec
e,RemC,RemL,TempBoard2)),
                !,
```

```
                ResultingPlayer = Player,
                (var(TempBoard2),ResultingBoard = TempBoard;
                ResultingBoard = TempBoard2),
                (DropInitiative = Turn, ResultingDropInitiative = e;
                ResultingDropInitiative = DropInitiative);
        ChosenMove = DropC/DropL,
                %drop action occurred
                letter_to_num(Col, DropC),
                write('AI is dropping a piece at position '), write(Col), write(DropL), nl,
                wait_for_input,
                drop_piece(Board,DropC,DropL,Turn,ResultingBoard),
                !,
                NewNumPieces is NumPieces-1,
                ResultingPlayer = player(Turn,NewNumPieces,comp,Pred,Difficulty),
                (DropInitiative = e, ResultingDropInitiative = Turn;
                ResultingDropInitiative = DropInitiative)
        ).


%piece removal predicate for the AI. It simply removes the piece from the Board
        %no piece is to be removed
ai_remove(Board,_,0,0,Board):-!.
        %piece at RemC/RemL will be removed
ai_remove(Board,OppPiece,RemC,RemL,ResultingBoard):-
        rp(Board,RemC,RemL,OppPiece,ResultingBoard).

%----------------------------------------------------------------------END OF SECTION----------------------------------
---------------------------------


%-------------------------------------------------------------SECTION: AI AUXILIARY PREDICATES----------------
-----------------------------------------
%this section contains auxiliary predicates for the AI's computations

%initial values of each board position for when the AI is calculating the board's total value
board_position_values([
        [5,2,2,2,5],
        [2,0,0,0,2],
        [2,0,0,0,2],
        [2,0,0,0,2],
        [5,2,2,2,5]]).


%AI generates all possible drops for the board and calls evaluate_and_choose to choose the
appropriate board
%the appropriate board is usually the best one but random 'miscalculations' are generated with
frequency deppending on the AI's difficulty
choose_drop_AI(Board,Turn,ChosenC,ChosenL,Difficulty,DropInitiative):-
        get_drop_moves(Board,DropMoves),
        (Difficulty = easy, RandomLimit is 6;
        Difficulty = medium, RandomLimit is 51;
        Difficulty = hard, RandomLimit is 201),
        evaluate_and_choose(Board,Turn,DropMoves,[],RandomLimit,ChosenMove,DropInitiative),
        ChosenMove = ChosenC/ChosenL.
```

```
%AI generates all possible movements (and piece removals after capture movements) for the board and calls evaluate_and_choose to choose the appropriate board
%the appropriate board is usually the best one but random 'miscalculations' are generated with frequency deppending on the AI's difficulty
choose_movement_AI(Board,Turn,IniC,IniL,DestC,DestL,RemC,RemL,Difficulty,DropInitiative):-
	get_movements(Board,Turn,Movements),
	(Difficulty = easy, RandomLimit is 6;
	Difficulty = medium, RandomLimit is 51;
	Difficulty = hard, RandomLimit is 201),
	evaluate_and_choose(Board,Turn,[],Movements,RandomLimit,ChosenMove,DropInitiative),
	ChosenMove = IniC/IniL-DestC/DestL-RemC/RemL.

%AI generates all possible drops and movements for the board and calls evaluate_and_choose to choose the appropriate board
%the appropriate board is usually the best one but random 'miscalculations' are generated with frequency deppending on the AI's difficulty
choose_move_AI(Board,Turn,ChosenMove,Difficulty,DropInitiative):-
	get_drop_moves(Board,DropMoves),
	get_movements(Board,Turn,Movements),
	(Difficulty = easy, RandomLimit is 6;
	Difficulty = medium, RandomLimit is 51;
	Difficulty = hard, RandomLimit is 201),
	evaluate_and_choose(Board,Turn,DropMoves,Movements,RandomLimit,ChosenMove,DropInitiative).

%selects the first drop move as the best one (or first movement if there are no drops) and uses an auxiliary predicates to check all the remaining moves
	%and compare them to the one currently selected as the best move
evaluate_and_choose(Board,Turn,DropMoves,Movements,RandomLimit,ChosenMove,DropInitiative):-
	get_opponent_piece(Turn,OppPiece),
	board_position_values(PosValues),
	%drop list is not empty, first drop is chosen as best move
	(DropMoves = [DropC/DropL|RemainingDrops],
drop_piece(Board,DropC,DropL,Turn,CurrentBestBoard),RemainingMovements = Movements,
		CurrentBestMove = DropC/DropL,
		(get_opponent_piece(Turn,DropInitiative), NewDropInitiative = DropInitiative;
		NewDropInitiative = Turn),
		get_piece(PosValues,DropC,DropL,PosValue), %value of having a piece in that position
		EatValue is 0;
	%drop list is empty, first movement is chosen as best move
	Movements = [IniC/IniL-DestC/DestL-RemC/RemL | RemainingMovements],
		move_piece(Board, IniC, IniL, DestC, DestL, Turn, TempBoard,
ai_remove(TempBoard,OppPiece,RemC,RemL,TempBoard2)),
		(var(TempBoard2),CurrentBestBoard = TempBoard, EatValue is 0;
		CurrentBestBoard = TempBoard2,EatValue is 20), %capture occurred, move value is increased
		CurrentBestMove = IniC/IniL-DestC/DestL-RemC/RemL,
		RemainingDrops = DropMoves,
		(DropInitiative = Turn, NewDropInitiative = e;
		NewDropInitiative = DropInitiative),
		PosValue is 0),
	evaluate_board(CurrentBestBoard,Turn,EValue,NewDropInitiative),
	CurrentValue is EValue+PosValue,
	%remaining moves are evaluated
	eac_aux(Board,Turn,RemainingDrops,RemainingMovements,RandomLimit,CurrentBestMove,CurrentValue,ChosenMove,DropInitiative).
```

```
%base case for the evaluation auxiliary predicate, both lists of moves are empty, final result is the current best move
eac_aux(_,_,[],[],_,CurrentBestMove,_,CurrentBestMove,_):-!.
%drop moves are being evaluated and compared to the current best move
eac_aux(Board,Turn,[DropC/DropL | RemainingDrops],RemainingMovements,RandomLimit,CurrentBestMove,CurrentValue,ChosenMove,DropInitiative):-
	drop_piece(Board,DropC,DropL,Turn,ResultingBoard),
	board_position_values(PosValues),
	get_piece(PosValues,DropC,DropL,PosValue),
	(get_opponent_piece(Turn,DropInitiative), NewDropInitiative = DropInitiative;
	NewDropInitiative = Turn),
	evaluate_board(ResultingBoard,Turn,EValue,NewDropInitiative),
	RBValue is EValue+PosValue,
	random(1,RandomLimit,DoSomethingStupid), %randomly inserts errors into the AI computation deppending on the 'RandomLimit' (calculated from the AI difficulty)
	(DoSomethingStupid =\= 1, %in this case, the AI is choosing wisely
		(RBValue > CurrentValue, NewBestMove = DropC/DropL, NewValue is RBValue;
			%if the move has the same value as the current best move, the AI randomly chooses between the two (to prevent repetitive games)
			RBValue =:= CurrentValue, random(1,3,I),(I =:=1, NewBestMove = DropC/DropL, NewValue is RBValue;

	NewBestMove = CurrentBestMove, NewValue is CurrentValue);
			NewBestMove = CurrentBestMove, NewValue is CurrentValue);
		%AI makes a 'mistake'. swaps moves but not values to prevent further iterations from 'correcting' the mistake
		(RBValue >= CurrentValue, NewBestMove = CurrentBestMove, NewValue is RBValue;
			NewBestMove = DropC/DropL, NewValue is CurrentValue)
	),
	!,
	%remaining moves are evaluated
	eac_aux(Board,Turn,RemainingDrops,RemainingMovements,RandomLimit,NewBestMove,NewValue,ChosenMove,DropInitiative).
```

```
%movements (and possible piece removals) are being evaluated and compared to the current best
move (no drops left to evaluate)
%functions similarly to the evaluation of drop moves
eac_aux(Board,Turn,[],[IniC/IniL-DestC/DestL-RemC/RemL |
RemainingMovements],RandomLimit,CurrentBestMove,CurrentValue,ChosenMove,DropInitiative):
-
	get_opponent_piece(Turn,OppPiece),
	move_piece(Board, IniC, IniL, DestC, DestL, Turn, TempBoard,
ai_remove(TempBoard,OppPiece,RemC,RemL,TempBoard2)),
	(var(TempBoard2),ResultingBoard = TempBoard,AcrescValue is 0; %checks if
TempBoard2 (after removal) was initialized or not (whether a capture move or standard move
occurred)
	ResultingBoard = TempBoard2,AcrescValue is 20), %assigns the appropriate current board

	(DropInitiative = Turn, NewDropInitiative = e;
	NewDropInitiative = DropInitiative),
	evaluate_board(ResultingBoard,Turn,EValue,NewDropInitiative),
	RBValue is EValue + AcrescValue,
	random(1,RandomLimit,DoSomethingStupid),
	(DoSomethingStupid =\= 1,
		(RBValue > CurrentValue, NewBestMove = IniC/IniL-DestC/DestL-RemC/RemL ,
NewValue is RBValue;
			RBValue =:= CurrentValue, random(1,3,I), (I =:=1, NewBestMove =
IniC/IniL-DestC/DestL-RemC/RemL , NewValue is RBValue;

	NewBestMove = CurrentBestMove, NewValue is CurrentValue);
			NewBestMove = CurrentBestMove, NewValue is CurrentValue);
			%swap moves but not values to prevent further iterations from
'correcting' the mistake
		(RBValue >= CurrentValue, NewBestMove = CurrentBestMove, NewValue is
RBValue;
			NewBestMove = IniC/IniL-DestC/DestL-RemC/RemL, NewValue is
CurrentValue)
	),
	!,
	eac_aux(Board,Turn,[],RemainingMovements,RandomLimit,NewBestMove,NewValue,Chos
enMove,DropInitiative).

%calculates all possible movements for a given player on the specified board
get_movements(Board,Turn,Movements):-
	gm_aux(Board,Turn,1,5,Movements,Board).

%auxiliary predicate for calculating all possible movements for a player
	%checks the board for every piece belonging to that player and calculates all possible
moves for that piece
gm_aux([],_,_,_,[],_):-!.
gm_aux([[]|Rb],Turn,_,PosL,Movements,OB):-
	NewC is 1,
	NewL is PosL-1,
	gm_aux(Rb,Turn,NewC,NewL,Movements,OB).
gm_aux([[Turn|Rl]|Rb],Turn,PosC,PosL,Movements,OriginalBoard):-
	!,
	get_piece_movements(OriginalBoard,Turn,PosC,PosL,PieceMovements),
	NewC is PosC+1,
	gm_aux([Rl|Rb],Turn,NewC,PosL,RestOfMovements,OriginalBoard),
	append(PieceMovements,RestOfMovements,Movements).
gm_aux([[_|Rl]|Rb],Turn,PosC,PosL,Movements,OriginalBoard):-
	NewC is PosC+1,
	gm_aux([Rl|Rb],Turn,NewC,PosL,Movements,OriginalBoard).
```

```
%calculates all possible movements for a piece of the board at the specified position
get_piece_movements(Board,Piece,PosC,PosL,PieceMovements):-
	get_opponent_piece(Piece,OppPiece),
	%checks for possible normal movements
	(DestC is PosC+1, DestL is PosL, move_piece(Board, PosC, PosL, DestC, DestL, Piece, _,
fake_remove), NormalMoveRight=[PosC/PosL-DestC/DestL-0/0]; NormalMoveRight=[]),
	(DestC2 is PosC-1, DestL2 is PosL, move_piece(Board, PosC, PosL, DestC2, DestL2, Piece, _,
fake_remove), NormalMoveLeft=[PosC/PosL-DestC2/DestL2-0/0]; NormalMoveLeft=[]),
	(DestL3 is PosL+1, DestC3 is PosC, move_piece(Board, PosC, PosL, DestC3, DestL3, Piece, _,
fake_remove), NormalMoveDown=[PosC/PosL-DestC3/DestL3-0/0]; NormalMoveDown=[]),
	(DestL4 is PosL-1, DestC4 is PosC, move_piece(Board, PosC, PosL, DestC4, DestL4, Piece, _,
fake_remove), NormalMoveUp=[PosC/PosL-DestC4/DestL4-0/0]; NormalMoveUp=[]),
	%checks for possible capture moves (uses piece removal predicate list_removes which
stores all possible moves combined with all possible removes in a list)
	(DestC5 is PosC+2, DestL5 is PosL, move_piece(Board, PosC, PosL, DestC5, DestL5, Piece,
RB, list_removes(RB,PosC,PosL,DestC5,DestL5,OppPiece,CaptureMoveRight));
CaptureMoveRight=[]),
	(DestC6 is PosC-2, DestL6 is PosL, move_piece(Board, PosC, PosL, DestC6, DestL6, Piece,
RB2,list_removes(RB2,PosC,PosL,DestC6,DestL6,OppPiece,CaptureMoveLeft));
CaptureMoveLeft=[]),
	(DestL7 is PosL+2, DestC7 is PosC, move_piece(Board, PosC, PosL, DestC7, DestL7, Piece,
RB3,list_removes(RB3,PosC,PosL,DestC7,DestL7,OppPiece,CaptureMoveDown));
CaptureMoveDown=[]),
	(DestL8 is PosL-2, DestC8 is PosC, move_piece(Board, PosC, PosL, DestC8, DestL8, Piece,
RB4,list_removes(RB4,PosC,PosL,DestC8,DestL8,OppPiece,CaptureMoveUp)); CaptureMoveUp=[]),
	%combines all the produced results
	append(NormalMoveRight,NormalMoveLeft,Temp1),
	append(NormalMoveDown,NormalMoveUp,Temp2),
	append(CaptureMoveUp,CaptureMoveDown,Temp3),
	append(CaptureMoveRight,CaptureMoveLeft,Temp4),
	append(Temp1,Temp2,TP),
	append(Temp3,Temp4,TP2),
	append(TP,TP2,PieceMovements),!.

%when called, a capture move has occurred for a piece moving from PosC/PosL to DestC/DestL
%it checks all possible piece removals and creates a list with elements containing all of the moves'
information
	%in this case there are no pieces to be removed
list_removes(Board,PosC,PosL,DestC,DestL,OppPiece,[PosC/PosL-DestC/DestL-0/0]):-
	\+ has_piece(Board,OppPiece),!.
	%in this case an auxiliary predicate is used to check the board for all of the opponent's
pieces
list_removes(Board,PosC,PosL,DestC,DestL,OppPiece,List):-
	lr_aux(Board,PosC,PosL,DestC,DestL,1,5,OppPiece,List).

%goes through the entire board, listing positions where OppPieces are in the format PosC/PosL-
DestC/DestL-CurC/CurL where
	%CurC/CurL is the current position of the board being checked and PosC/PosL-
DestC/DestL are provided before-hand (capture move format)
lr_aux([],_,_,_,_,_,_,[]):- !.
lr_aux([[]|Rb],PosC,PosL,DestC,DestL,_,CurL,OppPiece,RList):-
	NewC is 1,
	NewL is CurL-1,
	lr_aux(Rb,PosC,PosL,DestC,DestL,NewC,NewL,OppPiece,RList).
```

```
lr_aux([[OppPiece|Rl]|Rb],PosC,PosL,DestC,DestL,CurC,CurL,OppPiece,[PosC/PosL-DestC/DestL-
CurC/CurL | RList]):-
	!,
	NewC is CurC+1,
	lr_aux([Rl|Rb],PosC,PosL,DestC,DestL,NewC,CurL,OppPiece,RList).
lr_aux([[_|Rl]|Rb],PosC,PosL,DestC,DestL,CurC,CurL,OppPiece,RList):-
	NewC is CurC+1,
	lr_aux([Rl|Rb],PosC,PosL,DestC,DestL,NewC,CurL,OppPiece,RList).

%gets all possible drop moves on the board (does not deppend on piece type)
get_drop_moves(Board,Moves):-
	gdm_aux(Board,1,5,Moves).

%goes through the board saving the positions of empty cells in a list in the format PosC/PosL
gdm_aux([],_,_,[]):-!.
gdm_aux([[]|Rb],_,PosL,Moves):-
	NewC is 1,
	NewL is PosL-1,
	gdm_aux(Rb,NewC,NewL,Moves).
gdm_aux([[e|Rl]|Rb],PosC,PosL,[PosC/PosL | RMoves]):-
	NewC is PosC+1,
	gdm_aux([Rl|Rb],NewC,PosL,RMoves).
gdm_aux([[_|Rl]|Rb],PosC,PosL,Moves):-
	NewC is PosC+1,
	gdm_aux([Rl|Rb],NewC,PosL,Moves).

%receives a board and a piece type and calculates the board's 'worth' for that piece type
evaluate_board(Board,Turn,Value,DropInitiative):-
	IniValue is 0,
	ev(Board,Turn,IniValue,1,5,Board,Value,DropInitiative).

%this predicates checks the whole board, decreasing the final value by 5 whenever an opponent's
piece is found or increasing by 5 when a piece of type 'Turn' is found
%when a piece of type turn is found it calculates the piece's value (using calculate_piece_value/5)
%the piece's value depends on what pieces it can eat or by which pieces it can be eaten (danger and
offensive values)
ev([],_,FValue,_,_,_,FValue,_):- !.
ev([[]|Rb],Turn,Value,_,PosL,OriginalBoard,FValue,DropInitiative):-
	NewC is 1,
	NewL is PosL-1,
	ev(Rb,Turn,Value,NewC,NewL,OriginalBoard,FValue,DropInitiative).
ev([[Turn|Rl]|Rb],Turn,Value,PosC,PosL,OriginalBoard,FValue,DropInitiative):-
	!,
	NewValue is Value+5,
	calculate_piece_value(OriginalBoard,Turn,PosC,PosL,PValue,DropInitiative),
	NewValue2 is NewValue+PValue,
	NewC is PosC+1,
	ev([Rl|Rb],Turn,NewValue2,NewC,PosL,OriginalBoard,FValue,DropInitiative).
ev([[Piece|Rl]|Rb],Turn,Value,PosC,PosL,OriginalBoard,FValue,DropInitiative):-
	NewC is PosC+1,
	(get_opponent_piece(Piece,Turn), NewValue is Value - 5;
	Piece = e, NewValue is Value),
	ev([Rl|Rb],Turn,NewValue,NewC,PosL,OriginalBoard,FValue,DropInitiative).
```

```
%this predicate calculates the value of a given piece of PieceType on the specified position of the
board
%this value consists of danger value plus offensive value (danger value might be negative)
calculate_piece_value(Board,PieceType,PosC,PosL,PValue,DropInitiative):-
        calculate_danger_value(Board,PieceType,PosC,PosL,DValue,DropInitiative),
        calculate_offensive_value(Board,PieceType,PosC,PosL,OValue,DropInitiative),
        PValue is DValue+OValue.

%this predicate calculates a danger value for a piece on the board. The value is -18*(number of
pieces that can eat the specified piece) if opponent can
%capture on the next turn (depends on drop initiative), -2*(number of pieces that can eat the
specified piece) otherwise.
%(since after this turn, the opponent will have his turn, being in danger of losing the piece nullifies
any 'eat moves' that piece can do
%       since any eat moves are worth only 6 points and if the piece can be eaten it can only eat 3
pieces).
calculate_danger_value(Board,PieceType,PosC,PosL,Value,DropInitiative):-
        (DropInitiative = PieceType, Penalizer is -2;
        Penalizer is -18),
        CLeft is PosC-1, CRight is PosC+1, LUp is PosL+1, LDown is PosL-1,
        get_opponent_piece(PieceType,Piece),
        (capture_move(Board, PosC, LUp, PosC, LDown, Piece, _,fake_remove), UpValue is
Penalizer;UpValue is 0),
        (capture_move(Board, PosC, LDown, PosC, LUp, Piece, _,fake_remove), DownValue is
Penalizer;DownValue is 0),
        (capture_move(Board, CLeft, PosL, CRight, PosL, Piece, _,fake_remove),LeftValue is
Penalizer;LeftValue is 0),
        (capture_move(Board, CRight, PosL, CLeft, PosL, Piece, _,fake_remove), RightValue is
Penalizer;RightValue is 0),
        Value is UpValue+DownValue+LeftValue+RightValue.




%this predicate calculates an offensive value for a piece on the board. The value is 6*(number of
pieces that the specified piece can eat) if the opponent
% does not have the drop initiative, 2*(number of pieces that the specified piece can eat) otherwise.
calculate_offensive_value(Board,PieceType,PosC,PosL,Value,DropInitiative):-
        (get_opponent_piece(PieceType,DropInitiative), Valorizer is 2;
        Valorizer is 6),
        CLeft is PosC-2, CRight is PosC+2, LUp is PosL+2, LDown is PosL-2,
        (capture_move(Board, PosC, PosL, PosC, LUp, PieceType, _,fake_remove), UpValue is
Valorizer; UpValue is 0),
        (capture_move(Board, PosC, PosL, PosC, LDown, PieceType, _,fake_remove), DownValue is
Valorizer; DownValue is 0),
        (capture_move(Board, PosC, PosL, CLeft, PosL, PieceType, _,fake_remove), LeftValue is
Valorizer; LeftValue is 0),
        (capture_move(Board, PosC, PosL, CRight, PosL, PieceType, _,fake_remove), RightValue is
Valorizer; RightValue is 0),
        Value is UpValue+DownValue+LeftValue+RightValue.

%----------------------------------------------------------------END OF SECTION----------------------------------
---------------------------------
```