Identificação do trabalho (jogo) e do grupo

Jogo: Moxie
Grupo: 3
Trabalho realizado por:
José Pedro Teixeira Ramos – up202005460
José Leandro Rodrigues da Silva – up202008051
Contribuição: 50/50

Instalação e execução

Para executar o jogo:
PROLOG
consult('…/src/Game.pl')
play.

Descrição do jogo

Cada jogador começa com oito peças na reserva fora do tabuleiro. Uma jogada consiste em: 1) colocar uma peça da reserva em qualquer casa vazia; 2) mover uma peça que esteja em campo para qualquer quadrado adjacente, ortogonal ou diagonalmente (como um rei de xadrez), ou 3) capturar uma peça inimiga saltando sobre ela, como uma dama de 8 direções. Vários saltos são permitidos. Quando um salto é possível, o salto é obrigatório, embora o jogador possa escolher qualquer movimento de salto disponível.
O objetivo é formar uma linha de três com suas peças, ortogonal ou diagonalmente sem espaços intermediários, ou capturar seis peças inimigas (ou seja, reduzir as forças do inimigo a duas para que ele não possa vencer).
Fonte: https://boardgamegeek.com/boardgame/24546/moxie

Lógica do Jogo

Representação interna do estado do jogo: A variável GameState tem tudo que precisamos para fazer a representação interna do jogo, ela está estruturada da seguinte maneira [Arena, Peça_atual, Peças_restantes_X, Peças_restantes_O]. A arena inicial é uma lista de tamanho 16 que começa com espaços vazios ' ', e à medida que o jogo vai percorrendo vão se substituindo os espaços por peças, por ex 'x' ou 'o', de acordo com o input do jogador. A Peça_atual é a peça que está a jogar atualmente, por ex 'x' ou 'o', começa sempre a jogar a peça 'x'. As Peças_restantes_X e as Peças_restantes_O são o número de peças que restam em reserva e ainda podem ser colocadas em campo, sendo que Peças_restantes_X é o jogador X e o Peças_restantes_O é o jogador O, no início começam os dois com 8.
Exemplo de representação do jogo inicial: Arena == [' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '], Peça_atual == 'x', Peças_restantes_X == 8 e Peças_restantes_O == 8.
Exemplo de representação do jogo intermédio: Arena == [' ','x','x',' ',' ',' ',' ',' ','o','o',' ',' ',' ',' ','o',' '], Peça_atual == 'o', Peças_restantes_X == 2 e Peças_restantes_O == 5.
Exemplo de representação do jogo final: Arena == [' ','x',' ',' ','o',' ',' ',' ',' ','o','o',' ',' ',' ',' ','o',' '], Peça_atual == 'o', Peças_restantes_X == 2 e Peças_restantes_O == 4.

Visualização do estado de jogo: Quando fazemos play. O prolog mostra-nos um menu com 1 - play, 2 - rules e 3 - exit, para interagir com o menu escrevemos o número daquilo que queremos, por exemplo 1. vai para o play. O play mostra os diferentes modos de jogo, o rules mostra as regras e o exit sai do jogo. No play ele faz o mesmo procedimento, ou seja, mostra vários modos de jogo e espera por input do jogador. Quando estamos num jogo temos a função display_game(+GameState), no qual pega na Arena, Peças_restantes_X e Peças_restantes_O do GameState. No início ele chama a função remaining(Peça,+Peças_restantes) para as duas peças, que repete a Peça pelo nr de Peças_restantes, isto serve para representar o nrº de peças restantes de cada jogador. Depois disto ele chama a função arena_x(+Arena, +Row), esta função pega na arena e transforma a lista Arena numa representação gráfica, pegando nos elementos um a um recursivamente até não ter mais elementos de maneira a que fica na seguinte maneira 'x--', quando a Row é igual a 0 ele escreve 'x' ,nl, '| | | |',nl e depois repõe a Row para o seu valor original.

Execução de jogadas: Para executar a jogada selecionada pelo jogador desenvolveu-se o predicado move(+GameState, +Move, -NewGameState). Este predicado, precisa apenas de distinguir entre o tipo de jogadas ("eat", "move" ou "piece") e devolver um novo Game State que é o resultado da aplicação da jogada (+Move). Para distinguir entre os vários tipos de jogadas, primeiro verifica-se se +Move:
* É um inteiro, se for, então é uma jogada de "piece".
* Se é uma jogada do tipo "move" válida, então é uma jogada de "move".
* Se não é nenhuma das anteriores, então é uma jogada do tipo "eat" (ou jump).
A Jogada do tipo "eat" (ou jump) salta por cima de uma peça adversária, numa posição adjacente, e retira-a da arena. Esta jogada é representada por um par do tipo Posição_inicial-Posição_final e o seu efeito na arena é obtido através do predicado eat(+Arena, +Pos_inicial, +Pos_final, -NewArena).
A Jogada do tipo "move" move a peça para uma posição adjacente (ortogonal ou diagonal). Esta jogada é representada por um par do tipo Posição_inicial-Posição_final e o seu efeito na arena é obtido através do predicado move(+Arena, +Pos_inicial, +Pos_final, -NewArena).
A Jogada do tipo "piece" coloca a peça numa posição vazia qualquer. Esta jogada é representada por um inteiro do tipo Posição_final e o seu efeito na arena é obtido através do predicado piece(+Arena, +Piece, +Pos, -NewArena). Quando se realiza esta jogada deve-se decrementar o nº de peças disponíveis para a peça que realiza a jogada.

Lista de Jogadas Válidas: Para obter a lista de válidas possíveis para uma determinada peça numa determinada situação, desenvolveu-se o predicado valid_moves(+GameState,, -ListOfMoves). Não foi necessário incluir o parâmetro +Player, pois qualquer peça (x ou o) pode realizar os mesmos tipos de movimentos. Este predicado, no fundo, analisa o GameState e devolve uma lista com todos os possíveis movimentos válidos, tendo em conta a peça a jogar, utilizando 3 outros predicados eat_option(+Arena, +Piece, -Eat_moves), piece_option(+Arena, -Piece_Moves) e move_option(+Arena, +Piece, -Move_moves).
eat option devolve todos os possíveis movimentos de jump ou "comer" que são uma lista de pares constituídos pela posição inicial e final da peça.
move_option devolve todos os possíveis movimentos de "mover" numa uma lista de pares constituídos pela posição inicial e final da peça.

piece_option devolve todos os possíveis movimentos de “colocar” numa uma lista de inteiros representando a posição final da peça.
No caso de ser possível comer, valid_moves devolve apenas os movimentos válidos de “comer”, senão devolve todos os movimentos válidos de “colocar” e “mover”.
Avaliação do Tabuleiro: Para o AI (do tipo 2), saber quais as melhores jogadas possíveis num determinado momento/estado do jogo, desenvolveu-se o predicado value(+GameState, +Player, -Value). Este predicado analisa o GameState e atribui-lhe um determinado valor seguindo os seguinte parâmetros:
* Value = 0 se Piece pode ser comida e não pode comer
* Value = 1 se Piece não pode fazer nada / outro
* Value = 2 se Piece pode comer uma peça e não pode ser comida
* Value = 3 se Piece ganhou o jogo
Como este predicado só é utilizado pelo AI (do tipo 2), o parâmetro +Player é sempre 2.

Jogada do Computador: Para um determinado jogador, seja ele humano ou AI, escolher uma jogada, desenvolveu-se o predicado choose_move(+GameState, +Player, -Move), onde +Player pode ser “human” ou “1” (AI aleatório), ou “2” (AI do tipo 2, usa a avaliação do GameState).
Este predicado quando o parâmetro +Player é:
* “human”, pede o input ao user e verifica se o input é uma jogada válida (pertence à lista dada pelo predicado valid_moves).
* “1”, escolha uma jogada válida aleatória.
* “2”, obtêm-se as jogadas válidas, analisa-se o valor do GameState resultante após a realização de cada jogada e associa-se esse valor à jogada através de uma lista de pares onde o 1º elemento é o valor e o 2º é o move. É escolhida aleatoriamente uma das jogadas com maior valor associado.

Final do Jogo: A função game_over(+GameState, -Winner) é verificada antes de um jogador começar a sua jogada e se falhar nenhum jogador ganhou. Esta função pode ser dividida em duas partes, que correspondem aos dois objetivos do jogo, um deles é verificar se um jogador ganhou por número de peças e o outro é verificar se ganhou por 3 em linha. Começando pelo primeiro objectivo. A função end_piece(+Arena, +Peças_restantes_x, +Peças_restatantes_o, -Winner) calcula se alguém ganhou por peças, portanto o que ele faz é uma conta entre as peças restantes na reserva e as peças que estão em campo, se o total entre essas duas for menor que três o Winner é dado, sendo que as peças que estão em campo são calculadas diretamente da Arena a partir da função amount(+Arena,-Campo,+Peça), que percorre a Arena toda até acabar e sempre que encontra uma Peça ele aumenta 1 no Campo. A função end_3row(+Arena, +Coluna, -Winner), esta função calcula se algum jogador ganhou por 3 em linha, isto é possível com a função auxiliar what_piece(+Arena,+Posição,+Peça) que verifica se uma Peça encontra-se na Posição da Arena, com esta função verificamos se as peças que queremos encontram-se nas posições que queremos, e assim verificamos se as posições fazem 3 em linha, a coluna serve de restrição para que não sejam calculados 3 em linha impossíveis, se não houver 3 em linha passa-se ao próximo elemento da Arena, e quando a Coluna chega a 3 ele faz reset para 0.

Conclusões

A realização deste trabalho prático foi bastante enriquecedora no que toca a vários conhecimentos associados à linguagem prolog e ao paradigma de programação lógica, que apesar de exigir bastante raciocínio lógico, permite a criação de scripts relativamente complexos em poucas linhas de código.
Este trabalho em específico, apresenta a limitação de que o tamanho da arena não é flexível, mas a futura implementação desta feature foi facilitada no código, pois seria apenas preciso trocar os valores específicos do tamanho default por variáveis que poderiam ser inseridas na representação do GameState (ou representar a arena usando uma lista de listas) e implementar um menu que permita o user escolher um tamanho para a arena.
Uma das possíveis melhorias que se poderiam realizar era uma tonificação da técnica de avaliação do estado de um jogo, isto tornaria o AI do tipo 2 mais perspicaz na escolha de jogadas.