AGERE_4

Trabalho realizado por

- Haochang Fu (up202108730)
- João Miguel Vieira Cardoso (up202108732)

Pré-requesitos, instalação e execução

- Ter Sicstus instalado;
- Descomprimir todos os ficheiros da pasta zip para uma pasta nova desde que consiga correr Sicstus nela;
- Executar Sicstus, consultar o ficheiro "proj.pl" e começar a jogar.

Descrição do jogo

O nome do jogo é Agere e trata-se de um jogo de tabuleiro que pode ser jogado por 2 jogadores, que por sua vez podem ser pessoas ou o computador.

O objetivo do jogo é ligar os 3 lados do tabuleiro com as peças da sua cor.

No início do jogo, o primeiro jogador usa as peças de cor amarela (representadas por Y) e o segundo jogador utiliza as peças de cor azul (representadas por B). Na eventualidade de haver uma peça por cima de outra, nós também representamos o número de peças que estão numa determinada posição do tabuleiro com um número. Por exemplo se houver uma peça amarela numa posição com uma azul por baixo dela, então representamo-la desta forma (Y2). O número máximo de altura é 9.

O tabuleiro apresenta forma triangular, tendo cada posição ou "casa" um aspeto hexagonal. O tamanho do tabuleiro pode variar consoante o que o jogador escolhe no menu: small, medium, large.

No menu também será possível escolher o tipo de adversário que pode ser human vs human, human vs computer ou computer vs computer.

As regras de jogo resume-se a duas decisões que os jogadores podem tomar em cada turno:

- Adicionar uma peça da sua cor ao tabuleiro: Os jogadores selecionam a posição onde pretendem colocar a sua peça, sendo que a posição onde a colocam não pode ter uma peça já nela colocada.
- Mover uma peça da sua cor para cima de uma peça adversária: Pegar numa peça da sua cor e movê-la para cima de uma peça adversária vizinha dela. No entanto, isto só é possível se a peça adversária tiver a mesma altura que a peça que pretendes mover. Na maneira como o

nosso jogo está feito, só se o número ao lado das peças forem iguais é que será possível fazer isto.

O jogo acaba quando houver uma ligação contínua de peças da mesma cor a ligar os 3 lados do tabuleiro. Mal que este estado se verifique, o jogo imediatamente termina e o jogador dono das peças que ligaram os lados do tabuleiro pronuncia-se como vencedor do jogo.

Link para regras do jogo em pdf

Lógica do jogo

O código do programa foi separado em diferentes ficheiros de acordo com as suas funções no programa:

- AI.pl: Responsável pela inteligencia arteficial que é responsável por trás do computador como jogador;
- board.pl: Responsável pelos predicados que fazem a pesquisa e alteram o estado do tabuleiro (GameState);
- display.pl: Responsável pelo output de vários elementos como o tabuleiro, o menu e o jogo;
- game.pl: Responsável pela lógica durante a execução de uma partida em Agere;
- game_over.pl: Responsável por verificar o fim do jogo.
- menu.pl: Responsável pelas interações feitas ao menu e tudo relacionado;
- proj.pl: Ficheiro a executar para começar o programa e inclui as livrarias que o programa usa para correr.

1. Representação do estado do jogo

Para representar o GameState, o programa usa uma lista de listas (matrix) em que cada fila corresponde a uma fila no triangulo e cada elemento dentro de uma sublista corresponde a uma peça que está na forma **Color-Height**, neste caso o GameState corresponde ao estado do tabuleiro. O comprimento que o jogador introduzir no inicio, no menu, determina o tamanho do tabuleiro sendo este tamanho limitado entre 3 e 9 para um jogo ideal. O triângulo é uma porção da matrix de comprimento lado x lado. Este é inicializado com predicado **initial_state(+Size,-GameState)** em que leva tamanho do lado do triangulo como input e unifica o GameState com posições vazias no inicio da forma o-0.

Para representar o jogador temos 2 cores que os corresponde e uma cor que representa espaço vazio: y(yellow) , b(blue) e o(empty).

Para aglomerar o estado do tabuleiro e o estado do jogador, temos predicado **game(+CurrPlayerType, +NextPlayerType, +MoveCount, +GameState, -Winner)** que representa o estado do jogo e vai alternando os turnos do jogador fazendo chamadas recursivas. Para distinguir os jogadores , usamos o MoveCount que é um atomo incremental e fazemos *MoveCount mod 2* para distinguir se par ou ímpar,atribuindo a jogada de acordo com o predicado

turn(+GameState,+MoveCount,+PlayerType,-NewGameState) que gere o turno de jogador e a sua jogada.

Durante a execução de uma partida, a lógica do jogo junta as peças do tabuleiro com as suas respetivas posições no tabuleiro em forma de par [lista de todas as peças]-[lista das suas respetivas posições no tabuleiro], enquanto calcula as posições válidas para o jogador com o predicado **valid_moves(+GameState, +Player, -ListOfMoves)**. A tradução das posições na matrix para posições de um tabuleiro triangular foi um dos desafios, o tabuleiro está representado com matrix de filas desde 1 elemento até ao número de lado que jogador introduziu. Desta forma, para calcular a coluna da posição de uma peça correspondente ao tabuleiro, foi usado a fórmuma *(Número de filas do tabuleiro) - (Número de elementos de uma fila) + 1 - 2 x (numero da coluna da peça no matrix)*.

2. Visualização do jogo

Para uma melhor experiência, usar fonte de tamanho 12 no terminal de Sicstus.

Para visualizar o jogo, temo várias momentos: durante a interação com o menu e durante o jogo.

Durante a interação com o menu, temos o predicado **menu/0** que chama funções auxiliares que avaliam o estado do menu,**menu_state(+Event)**, e que mostram o menu de diferentes tipos de draw:
draw_initial_menu/0,**draw_choose_player_type_menu/0**,**draw_choose_board_size_menu/0**,**draw_choose_difficulty_menu(-Text)**. Os predicado **menu_input(?Input,?Event)**, **menu_player_input(?Input,?Event)** e **menu_endgame_input(?Input,+PlayerType1,+PlayerType2,?Event)** escolhem estado de acordo com input do jogador,com **read_number_input(+Min,+Max,-Result)**, com verificação de validade. A interação com menu pode acontecer antes e depois de um jogo.

Durante a execução de uma partida, o **initial_state/2** é usado para inicializar o tabuleiro vazio e o **display_game(+GameState)** dentro do **game/5** é usado para mostrar o tabuleiro.

3. Validação de jogadas e execução

A validação de jogada em Agere depende da cor do jogador e das 2 operações possíveis: adicionar peça ou mover peça. O predicado **move/3** avalia o input escolhido pelo jogador ou gerado pelo computador e verifica se é uma posição vazia ou uma posição com peça da cor dele e se é valida, executa **execute_move(+GameState, +Letter-Number, -Color-Height, +Player, -NewGameState)** para ver se é adicionar peça ou mover peça. Se for para adicionar peça,o que significa que a posição escolhida é vazia, este unifica com o -NewGameState que é um dos argumentos do move/3 e que contém GameState atualizado com peça adicionada em cima do vazio, falha e não executa o **make_another_move/7** que serve para mover a peça para outra posição. Caso o input seja uma posição que tem peça da cor do jogador, este é considerado uma ação de mover e então execute/5 não falhará e passará a executar **make_another_move(+GameState,+Size,+Piece,+Position,?NewGameStateAcc,-NewGameState,+PlayerType)**, o que pede input do jogador para posição

pretendida dos vizinhos válidos, que é avaliada e no final executa a ação de mover atualizando o GameState.

4. Lista de jogadas válidas e verificação do input do jogador

O jogo utiliza o predicado valid_moves(+GameState, +Player, -ListOfMoves) em vários outros predicados e o seu uso principal é validar o input do jogador quer seja para adicionar quer mover, e obter jogadas possíveis caso seja para mover a peça, em conjunto com outros predicados **get_neighbour_pieces/3** e **get_pieces_same_height/3**.

5. End Game

O predicado **game_over(+GameState,Winner)** verifica-se quando existir algum jogador que ganhe por unir os 3 lados com as peças da sua cor. Essa verificação é feita pelo predicado **check_end_condition(+GameState,-Winner)** , o que faz um bfs pelas peças já colocadas pelo jogador e verifica se alguma dessas peças pertence a lista das peças que estão nos lados do triângulo, caso não haja pelo menos uma peça em cada lado o predicado falha e se o no caminho percorrido não encontrar as 3 peças nos lados no caso de ganhar, tamém falha. Nesse caso não se executa o game_over/2.

6. Verificação de estado do jogo

value(+GameState,+Player,-Values) avalia cada posição valida para o jogador e a tribui-lhes um valor de prioridade, quanto maior for prioridade mais provável será o AI escolher a posição para as suas jogadas.Retorna uma lista de valores para as posições válidas.

7. AI

Temos predicados no ficheiro AI.pl em que tem em conta o input recebido pelo menu e que divide o computador em 2 dificuldades: fácil e difícil.

O predicado **choose_move(+GameState, +Level, +Player, -Move)** distingue AI facil do difícil e utiliza diferentes implementaões de acordo para obter a move escolhida e distingue se é um input inicial para escolher posição ou se é o segundo input para escolher a posição para mover, unificando -Move em forma de Peça-Posição.

Para o AI fácil, este escolhe dentro da lista de jogadas possíveis um aleatório, e de acordo se existe peça nessa posição da sua cor ou não este distingue de adicionar uma peça ou mover uma peça. Caso seja adicionar o tabuleiro atualiza como quando o jogador joga e unifica com o novo GameState. Caso for mover peça este vê através de valid_moves/3, get_neightbour_pieces/3 e get_pieces_same_height/3 a lista de posições possíveis para mover a peça escolhida e de seguida escolhe uma aleatória dentro desta e executa a jogada.

Para o AI difícil, para verificar a melhor posição a escolher tem-se em conta várias condições que são calculadas através de um algoritmo que depende do estado do jogo atual:

- Se houver uma jogada que ganha o jogo, então essa jogada tem prioridade máxima
- Se houver uma jogada que impede o adversário de ganhar então essa jogada tem prioridade muito alta, perto da máxima
- Jogar uma peça próxima das peças da sua cor é uma ação greedy que a longo prazo pode levar a uma vitória, logo será atribuída a essa jogada maior pontuação
- Se houver um espaço que já esteja a ser vigiado por uma peça inimiga, então descontar uma certa pontuação a essa jogada
- Se o adversário tiver um peça de 2 ou mais de altura em jogo, atribuír maior pontuação a jogar perto dela
- Se o adversário tiver uma peça de altura 1, prioritorizar jogar perto dela
- Se houver uma peça inimiga disponível para ser saltada para cima, então atribuír uma pontuação extra a essa jogada também Se a peça onde estiver a tentar saltar for de altura maior do que 1 então não atribuír muita pontuação por essa jogada

  O AI difícil irá avaliar cada uma das condições referidas anteriormente, e irá calcular uma jogada apropriada para o estado do jogo.

8. Conclusão

Em suma, o trabalho ajudou-nos a perceber melhor como os predicados funcionam e as suas interações para criar um programa, tabem ajudou-nos a melhor entender bfs.

Limitações atuais do programa consiste na inteligência limitada do AI, este joga seguindo um conjunto específico de regras que não conseguem prever todos os casos, mas uma boa quantidade delas. Em termos de performance alguns sitios também podem ser melhorados.

Dado a oportunidade, o AI vai ser mudado no futuro depois de conhecermos melhor as jogadas deste jogo.

9. Bibliografia

Este trabalho foi feito com apoio nos slides das aulas teóricas e com base no pdf das regras de jogo que nos foram fornecidos. Como apoio auxiliar este site.