Trabalho prático 2 - Prolog

4MATION

Descrição do jogo

O 4Mation é um jogo de tabuleiro cujo objetivo é colocar estrategicamente 4 cubos da mesma cor, de forma a criar uma linha - *vertical, horizontal ou diagonal* - de tamanho igual ou superior **4**.

Cada jogador inicia o jogo com todos os cubos de 1 cor[1] e vão jogando intercaladamente. **Cada cubo só poderá ser colocado num espaço adjacente - ortogonal ou diagonal - ao último cubo adicionado ao tabuleiro.**

O jogo termina quando um dos jogadores conseguir criar uma linha de tamanho igual ou superior a 4, **ganhando**, ou quando se esgotarem as jogadas, **empatando**.

Instalação e execução

Após a instalação do SICStus Prolog 4.7 é necessário:

1. dar `consult` do ficheiro `4mation.pl`
2. mudar a fonte em `settings` para a fonte `Consolas`
3. escrever na consola `play.`

Lógica de jogo

- Representação interna do estado do jogo

  O estado do nosso jogo é representado por:

  ```
  gameState(Board, turn(Player-Color), PreviousMove)
  ```

  ou seja, um `tuplo gameState` constituído pelo `board` atual, pelas informações necessárias do turno atual - quem é o jogador (humano ou computador) e qual a cor que este joga - e pela última jogada feita pelo adversário (`L-C`, sendo `L = -1` e `C = -1`) na primeira jogada.

  No caso do `board`, optamos por uma representação usando lista de listas com diferentes átomos para as peças, sendo estas:

  - `empty`: no caso de uma casa vazia
  - `black`: se essa casa possuir uma peça de cor preta
  - `white`: se essa casa possuir uma peça de cor branca

  Deste modo, no início do jogo, o estado de jogo irá possuir um `board` totalmente vazio (constituído apenas por átomos `empty`) e as informações do primeiro jogador a jogar (o jogador que inicia com as peças de cor `white`).

Uma possível representação de um `gameState` no início do jogo seria:

```
gameState([[empty, empty, empty, empty], [empty, empty, empty, empty],
[empty, empty, empty, empty], [empty, empty, empty, empty]], turn(human-
white), (-1)-(-1))
```

E conforme o jogo vai decorrendo, o `board` vai sendo preenchido com as jogadas:

```
  gameState([[empty, white, black, empty], [empty, white, black, empty],
[empty, empty, white, empty], [empty, empty, black, empty]], turn(human-
white), 3-2)
```

Até que algum dos jogadores consiga efetuar uma jogada vencedora:

```
gameState([empty, white, black, empty], [empty, white, black, empty],
[empty, white, white, empty], [black, white, black, empty]], turn(human-
white), 2-1)
```

- Visualização do estado de jogo

A nível de visualização, é o predicado `display_game` que trata da representação do `board`. Este `board` pode ter um tamanho variável, `>= 4x4`^{2}, dependendo do tamanho que o utilizador escolheu no início do jogo, e é representado por uma parte superior (constituída por um identificador da coluna e por uma borda superior), por um conjunto de linhas (também constituídas por um idenitificador da linha em questão, uma borda superior, uma borda inferior e uma divisória para separar as diferentes casas), e uma borda inferior praticamente semelhante à parte superior.

Assim, a representação do mesmo é praticamente desenvolvida pelo *display* da *top border* e *bottom border* e um ciclo para representar as diferentes linhas, visto que apenas difere nos conteúdos pelas quais são constituídas.

Quanto aos menus, dada a sua complexidade, estes já se encontram previamente elaborados, sendo apenas feita a chamada ao respetivo predicado que dá o *display* de um menu, levando a que este faça um conjunto de chamadas `write` com o conteúdo que lá se encontra *hard coded*.

Nestes menus e em qualquer outro local do programa onde sejam necessárias validações de entrada, é chamado o predicado `read_number`, que verifica se o *input* recebido é, realmente, um número tal como esperado. Este predicado vai verificar caracter a caracter se o caracter digitado pelo utilizador é um dígito, parando esta iteração quando encontrar um dígito `Final`, cujo *ascii code* é passado por *input*. Se tudo correr bem, então o utilizador forneceu um input de acordo com o formato esperado e é devolvido em `Res` o número correspondente.

- Execução de jogadas

Tal como especificado no enunciado, a validação e execução de uma jogada, obtendo um novo estado de jogo, são efetuadas no predicado `move(+GameState, +Move, - NewGameState)`.

A validação da jogada é realizada na chamada ao predicado `validator`, que por sua vez verifica se o `Move` recebido é válido. Para uma jogada ser válida tem que cumprir os seguintes requisitos:

- Estar dentro do board

```
validator(L-C, Board):-
  length(Board,Max),
  between(0, Max, L),
  between(0, Max, C).
```

- O local a jogar estar vazio

```
is_empty(Pos, Board):-
  verify_position(Pos, gameState(Board, turn(_-empty), _)).
```

O predicado `verify_position`, tal como o nome indica, verifica se o `Board` contido no `gameState` contém o valor `Value` passado como parâmetro.

- Ser adjacente à jogada anterior

  Esta regra de jogo obriga a que a próxima jogada seja numa posição adjacente à jogada anterior. A verificação é realizada de maneira verbosa para cada uma das 9 opções possíveis, no predicado `valid_move`.

- Final do jogo

O final do jogo é avaliado através do predicado `game_over`. Com base na jogada efetuada nessa ronda, o predicado avalia se esta gerou 4 ou mais peças seguidas da mesma cor. Caso isto se verifique, o jogo acaba. Este predicado chama `end_game` que verifica para as 4 possíveis direções a existência de 4 peças iguais seguidas e com base na resposta chama, ou não, o predicado `score_menu` que recebe a cor do estado de jogo atual. A cor atual será a do vencedor.

- Lista de jogadas válidas

Neste jogo, para uma jogada ser válida tem de se cumprir um dos seguintes requisitos:

- Ser a primeira jogada: para a primeira jogada, todas as jogadas dentro do `board` são consideradas válidas
- Ser uma casa vazia (representada com `emtpy` no `board`) e ser adjacente à última jogada realizada (partilha algum vértice com a casa que foi preenchida pela última jogada do adversário)

Deste modo, para obtermos todas as jogadas válidas, apenas teremos de encontrar todas as casas que obedeçam a estes requisitos. Como já tinhamos desenvolvido a função que valida uma jogada em função do

estado atual do `board` e da última jogada do adversário, para obtermos todas as jogadas possíveis, apenas precisamos de utilizar o seguinte predicado:

```
findall(Move, validator(Move, PM, Board), Moves).
```

- Jogada do computador

  Relativamente à jogada do computador - bot -, esta pode ser realizada contra o próprio utilizador ou contra um outro bot.

  Na 1ª situação mencionada, o utilizador pode escolher o nível de dificuldade do bot contra o qual vai jogar no menu `singlePlayer`.

  Tem as opções ***easy peasy***, em que o bot joga aleatoriamente, ***hardcore***, em que o bot tem em conta 2 algoritmos - a explicar a seguir -, e por último e menos importante, a opção ***impossible***, que gera um `board` 3x3 num jogo em que é necessário ter 4 peças seguidas para se ganhar.

  Na 2ª situação, também é permitida a escolha do nível de dificuldade dos bots, estando dísponiveis as opções ***dummy vs foo*** - bot nível 1 vs bot nível 1 -, ***dummy vs Roussel*** - bot nível 1 vs bot nível 2 - e ***Colmeraurer ^{3} vs Roussel ^{3}*** - bot nível 2 vs bot nível 2.

  A estratégia do bot nível 2 está subdividida nas 3 seguintes partes, por ordem de execução:

  - Bloquear o oponente

    A primeira avaliação que o bot executa nas suas possíveis jogadas é verificar se alguma delas impede o oponente de ganhar. Se sim, essa jogada é escolhida.

  - Minimax

    A 2ª estratégia e idealmente a última, é o algoritmo minimax que avalia todas as possibilidades de jogo e escolhe a melhor a longo prazo. Neste caso específico, o algoritmo percorre em largura todas as possíveis jogadas e quando encontra uma jogada que leva à vitória do jogador em questão, essa é a jogada escolhida.

    Infelizmente em _ boards_ de grandes dimensões (> 9x9), era muito dispendioso em termos de tempo de computação e retirava usabilidade ao jogo. Assim sendo, o algoritmo foi limitado a uma profundidade de 1000 iterações (escolhido de maneira a não exceder os 2 segundos ideais de tempo de espera).

    Caso o minimax exceda a profundidade limite, o bot recorre à 3ª estratégia.

  - Algoritmo guloso

    Neste, o bot avalia apenas as jogadas imediatamente possíveis - no máximo 8. A escolha é feita com base no número de peças da sua cor que existem na ortogonal e diagonal da posição a jogar.

    Para avaliar, o bot calcula o número de peças da sua cor até encontrar uma peça da cor do oponente. De maneira a distinguir situações em que, por exemplo, uma linha está vazia, e outra

tem 3 peças pretas, é utilizado um sistema de pesos.

empty	black	white	black
empty	black	X	black
empty	empty	empty	black
empty	white	empty	Y

No exemplo anterior, a jogada `X` tem peso máximo de 5 na horizontal e a jogada `Y` tem peso máximo de 6.

Os pesos foram necessários para conseguir fazer a distinção entre linhas vazias (potencial jogada), de linhas já com peças da cor em questão (jogada com ainda mais potencial).

Nota: Após a decisão da jogada, é exposta uma mensagem personalizada para cada um dos algoritmos, de modo a ser percetível qual foi o algoritmo utilizado.

Block Opponent - `Too easy m8.`

Minimax - `Careful. Ive got everything under control.`

Guloso - `Im tired of thinking. Ill just go with the most obvious one.`

- Avaliação do estado do jogo

A avaliação do estado do jogo é realizada de 3 maneiras diferentes para os 3 casos supra-mencionados.

A 1ª avaliação consiste na verificação de todas as posições passíveis de jogada imediata (nível de profundidade 1) e avaliar se essa é uma das posições do `board` que permite que o oponente ganhe. Para tal, simulamos uma jogada do oponente nessa posição e verificamos se foi `game_over`. É o predicado `check_opponent_win` que realiza esta avaliação.

A segunda avaliação consiste em percorrer recursivamente todas as jogadas possíveis até encontrar o primeiro conjunto de jogadas que gere vitória.

Por último, como já mencionado, é também realizada uma avaliação por pesos. No algoritmo guloso é necessário saber qual a melhor jogada imediata e, para tal, são calculados o número de peças da cor do jogador, com peso de 2, e o número de peças empty, com peso de 1, nas duas direções possíveis de jogar - ortogonal e diagonal - até encontrar uma peça do oponente. A jogada com maior peso é a escolhida. Este cálculo é efetuado no predicado `value`.

[1] Dadas as limitações gráficas, a distinção entre os cubos dos jogadores será a preto e branco - ao invés de cores.

[2] À exceção do modo de jogo `impossible`

[3] Criador da línguagem de programação PROLOG.

Conclusões

O projeto teve como objetivo aplicar o conhecimento adquirido nas aulas teóricas e práticas, da 2ª parte da unidade curricular de Programação Lógica.

Apesar das limitações gráficas, foi um projeto bastante interessante de desenvolver. Embora apenas um jogo - seria interessante ser outro tipo de projeto que nos permitisse perceber as potencialidades de Prolog -, o projeto em si permitiu que desenvolvessemos bastante o pensamento recursivo e reconhecemos ser essencial na aprendizagem da linguagem.

Gostavamos também de demonstrar o apreço pela organização e estrutura do projeto. A divisão em subpartes com prazos, não obrigatórios, ajudou bastante na organização e planeamento do projeto, o que, por exemplo, permitiu manter-nos, ao longo de todo o projeto, interessados.

No geral, foi um bom projeto que achamos ter cumprido o seu objetivo de aplicar e assentar o conteúdo lecionado nas aulas teóricas e práticas da unidade curricular.

Bibliografia

Link fonte

Trabalho realizado por por:

Ana Beatriz Melo Aguiar @up201906230 (50%)

Tiago Caldas da Silva @up201906045 (50%)