FABRIK

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo Fabrik3:

André Cruz - 201503776
Edgar Carneiro - 201503748

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

12 de Novembro de 2017

Resumo

Este trabalho tem como objetivo a modelação e implementação do jogo de tabuleiro *Fabrik*, com uma representação na linha de comandos, e disponibilizando os modos de jogo *jogador vs jogador*, *jogador vs computador* e *computador vs computador*. Tratando-se de um jogo de tabuleiro, foi necessário decidir um modelo apropriado para representar o estado do jogo, desenvolver predicados para verificar se determinada jogada é válida, e predicados para determinar quando o jogo termina. Foi também necessário implementar resilientes mecanismos de obtenção de *input* do utilizador, e menus para interface com o utilizador.

Relativamente à escolha automática de uma jogada por parte do computador, esta tem dois níveis: um mais fácil, em que a jogada é escolhida aleatóriamente; e um mais difícil, em que é identificada a melhor jogada que é possível efetuar tendo em conta o ocntexto atual (algoritmo ganancioso). Neste sentido, desenvolvemos vários predicados para a avaliação quantitativa de uma jogada e do estado do jogo, tendo sido alcançados os nossos objetivos de dificuldade de jogo contra o computador.

Recorrendo à excelente bibliografia indicada pelo professores, [1] e [2], foi possível resolver todos os problemas que encontramos na implementação deste jogo, bem como uma aprendizagem contínua dos conceitos relacionados com este paradigma de programação.

Para finalizar, achamos que o trabalho apresentado representa uma eficiente, completa e compreensiva modelação do jogo *Fabrik* e dos seus conceitos, e temos orgulho no resultado final.

Conteúdo

1 Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular de Programação em Lógica, integrada no 3º ano do Mestrado Integrado em Engenharia Informática e Computação, tendo como objetivo aprofundar os conhecimentos adquiridos nas aulas teóricas e práticas desta unidade curricular, bem como a abordagem de problemas mais práticos com recurso à linguagem *PROLOG*. Deste modo, propusemo-nos a implementar e modelar do jogo de tabuleiro *Fabrik*, sendo possível os modos de jogo *jogador vs jogador*, *jogador vs computador* e *computador vs computador* (com dois níveis de dificuldade).

A escolha deste jogo, que foi selecionado dentro de um leque de opções disponibilizadas pelos docentes, prendeu-se no conjunto de conceitos intuitivos que incorpora (*e.g.* linhas de visão das peças), permitindo uma rápida aprendizagem das regras, assim como um desafio na implementação de restrições de jogadas e avaliação do tabuleiro. Esta aparente simplicidade é no entanto contrabalançada com o facto de cada jogada ter dois passos: a colocação do *worker*, e a colocação de uma peça *black*/*white* (estando este segundo passo fortemente constrangido pelo primeiro).

Este relatório está dividido em várias secções, sendo a presente a primeira:

1. Na segunda secção deste relatório, é feita uma pequena introdução à história do jogo *Fabrik*, aos componentes físicos necessários para jogar, e às regras oficiais do jogo;

2. Na secção seguinte, são apresentados vários detalhes da nossa implementação, desde a representação visual do tabuleiro ao algoritmo de cálculo da melhor jogada possível dado um determinado estado de jogo;

3. Na quarta secção, é descrita a interface com o utilizador, que consiste nos menus e input esperado em cada um destes;

4. Na última secção, são tecidas conclusões acerca do desenvolvimento deste trabalho e do seu enquadramento curricular.

2 O Jogo *Fabrik*

2.1 História

O jogo - *Fabrik* - foi recentemente desenvolvido por Dieter Stein, em Agosto de 2017, como parte de um estudo para o desenvolvimento de um novo jogo, Urbino.

2.2 Material

- Tabuleiro quadrangular
- Quantidade suficiente de peças pretas e brancas
- Duas peças vermelhas chamadas trabalhadores

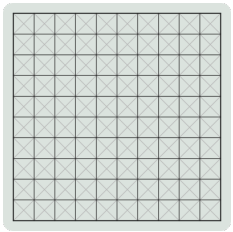

Figura 1: Tabuleiro vazio de 11 x 11 espaços

2.3 Regras

A implementação deste jogo foi baseada no manual de regras oficiais [3].

As pretas (jogador que joga com peças de cor preta) começam por colocar um dos trabalhadores num espaço à sua escolha. De seguida, as brancas (jogador que joga com peças de cor branca) colocam o outro trabalhador num espaço livre. De seguida, as pretas decidem quem começa por jogar.

O jogo procede por turnos, sendo que em cada turno um jogador pode, se assim optar, mover um dos trabalhadores para um espaço vazio. De seguida, o jogador deve jogar colocar uma das suas peças num ponto de interseção entre as "linhas de visão dos dois trabalhadores". As linhas de visão dos trabalhadores são as linhas na diagonal, horizontal e vertical sobre as quais os trabalhadores se encontram posicionados.

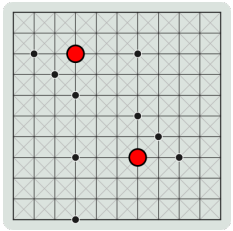

Figura 2: Pontos de interseção entre os dois trabalhadores

No caso especial em que os dois trabalhadores se encontram sobre uma mesma linha ortogonal ou diagonal, apenas os espaços entre eles são considerados pontos de interseção (se estiverem vazios), ao invés da totalidade dessa linha.

Ganha o jogo o jogador que consiga criar uma linha de pelo menos 5 pedras da sua cor, ortogonalmente ou diagonalmente. Um jogador ganha também o jogo se o seu adversário não conseguir posicionar nenhum dos trabalhadores de forma a poder colocar uma pedra sua no tabuleiro.

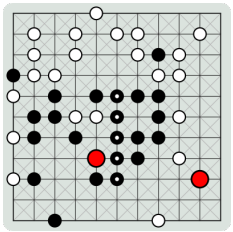

Figura 3: Final de uma partida de Fabrik, com vitórias das pretas

Referências:
https://spielstein.com/games/fabrik
https://spielstein.com/games/fabrik/rules

3 Lógica do Jogo

Descrever o projeto e implementação da lógica do jogo em Prolog, incluindo a forma de representação do estado do tabuleiro e sua visualização, execução de movimentos, verificação do cumprimento das regras do jogo, determinação do final do jogo e cálculo das jogadas a realizar pelo computador utilizando diversos níveis de jogo. Sugere-se a estruturação desta secção da seguinte forma:

3.1 Representação do Estado do Jogo

A representação dos estados de jogo é feita com recurso a uma lista de listas, de forma a simular o uso de uma Matriz. Seguem de seguida, a representaçao de diferentes estados de jogo:

Representação do **estado inicial**:

```
[[none, none, none, none, none, none, none, none, none, none, none],
 [none, none, none, none, none, none, none, none, none, none, none],
 [none, none, none, none, none, none, none, none, none, none, none],
 [none, none, none, none, none, none, none, none, none, none, none],
 [none, none, none, none, none, none, none, none, none, none, none],
 [none, none, none, none, none, none, none, none, none, none, none],
 [none, none, none, none, none, none, none, none, none, none, none],
 [none, none, none, none, none, none, none, none, none, none, none],
 [none, none, none, none, none, none, none, none, none, none, none],
 [none, none, none, none, none, none, none, none, none, none, none],
 [none, none, none, none, none, none, none, none, none, none, none]]
```

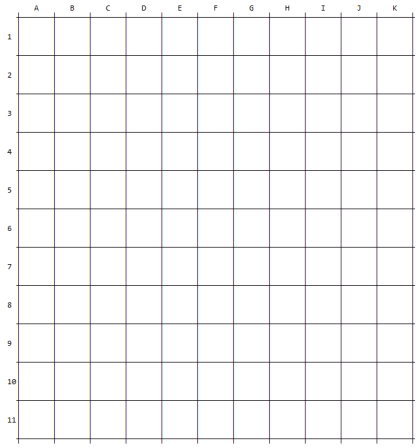

Figura 4: Representação do estado inicial na consola

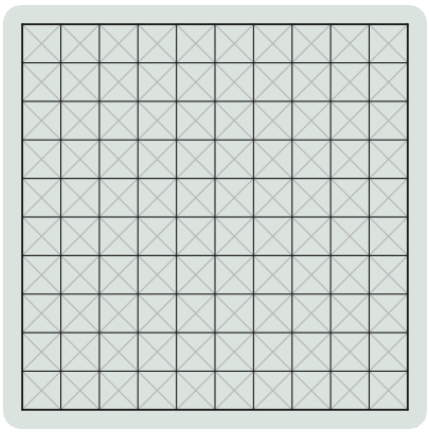

Figura 5: Tabuleiro original Vazio

Representação de um possível **estado intermédio**:

```
[[ none, none, none, none, none, none, none, none, none, none, none],
 [ none, none, none, none, none, white, worker, none, none, none, none],
 [ none, none, none, none, none, none, white, black, white, none, none],
 [ none, white, white, none, none, none, none, white, white, none, none],
 [white, none, none, none, black, black, none, black, none, none, none],
 [ none, none, black, white, none, none, none, none, white, none, none],
 [ none, black, none, none, none, black, black, black, none, none, none],
 [ none, none, none, none, none, none, black, none, white, none, none],
 [ none, none, none, none, none, black, none, none, none, none, none],
 [ none, none, none, none, worker, none, none, none, none, none, none],
 [ none, none, none, none, none, none, none, none, none, none, none]]
```

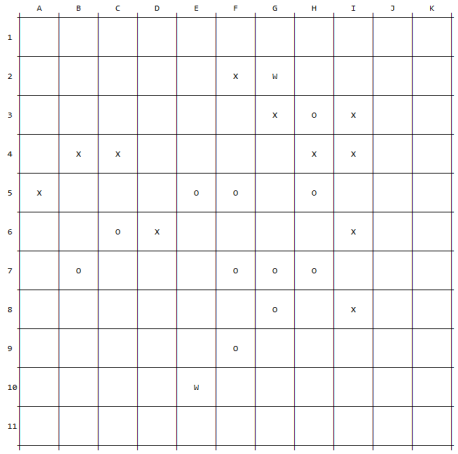

Figura 6: Representação na consola, de um possível estado intermédio

Representação de um possível **estado final**:

```
[[ none, none, none, none, white, none, none, none, none, none, none],
 [ none, white, none, white, none, white, white, none, none,white, none],
 [ none, white, none, white, none, none,white, black, white, none, none],
 [black, white, white, none, none, none, none, white, white, none, none],
 [white, none, black, none,black, black, black, black, none, none, none],
 [none, black, black, white,white, black, none, black,white, none, none],
 [white, black, none,black, none, black, black, black, none, none, none],
 [ none, none, none, none, worker, black, black, none,white, none, none],
 [white, black, none, none,black, black, none, none, none, worker, none],
 [ none, none, none, none, none, none, none, none, none, none, none],
 [ none, none, black, none, none, none, none, white, none, none, none]]
```

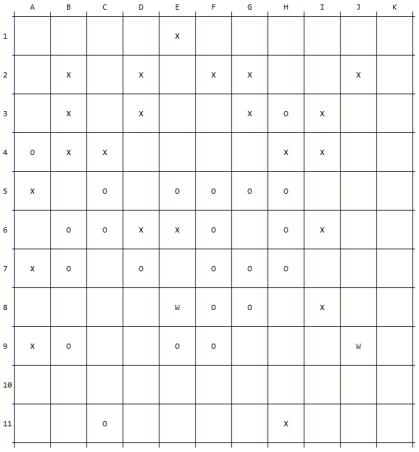

Figura 7: Representação do estado final na consola

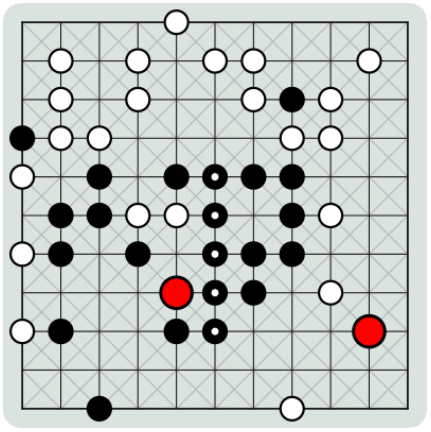

Figura 8: Representção do mesmo estado final, no tabuleiro original

3.2 Visualização do Tabuleiro

Para a representação do tabuleiro em modo de texto, foi criado o seguinte código em prolog:

```
% Dictionary for Board Elements
translate(none, 32). %Empty Cell
translate(black, 79). %Dark Pieces
translate(white, 88). %White Pieces
translate(worker, 9608). %Red Workers

(...)

% General PrintBoard
printBoard(Board):-
        boardSize(N),
        printBoard(Board, N), !.

% Board Printing - arguments: Board and Board size
printBoard(Board, N):-
        clearConsole,
        write(' '), printHorizontalLabel(N, N),
        printBoard(Board, N, 1), !.

printBoard([], N, _):-
        printRowDivider(N), nl.

printBoard([Line | Board], N, CurrentL):-
        printRowDivider(N),
        printDesignRow(N),
        printVerticalLabel(CurrentL),
        put_code(9474),
        printLine(Line),
        printDesignRow(N),
        NewL is (CurrentL + 1),
        printBoard(Board, N, NewL).

printLine([]):- nl.
printLine([Head | Tail]) :-
        translate(Head, Code),
        write('  '),
        put_code(Code),
        write('  '), put_code(9474),
        printLine(Tail).

%  AESTHETICS

printRowDivider(N):-
        write(' '),
        put_code(9532),
        printRowDividerRec(N).

printRowDividerRec(0) :- nl.
printRowDividerRec(N) :-
        put_code(9472), put_code(9472), put_code(9472), put_code(9472),
        put_code(9472), put_code(9472), put_code(9472), put_code(9532),
        N1 is (N-1),
```

```
        printRowDividerRec(N1).

printDesignRow(N):-
        write(' '),
        put_code(9474),
        printDesignRowRec(N).

printDesignRowRec(0) :- nl.
printDesignRowRec(N) :-
        write('      '), put_code(9474),
        N1 is (N-1),
        printDesignRowRec(N1).

%Dictionary for Labels
getLabel( 0, 'A').
getLabel( 1, 'B').
getLabel( 2, 'C').
getLabel( 3, 'D').
getLabel( 4, 'E').
getLabel( 5, 'F').
getLabel( 6, 'G').
getLabel( 7, 'H').
getLabel( 8, 'I').
getLabel( 9, 'J').
getLabel(10, 'K').
getLabel(11, 'L').
getLabel(_,_):-
        write('Error: Unrecognized Label.'), nl,
        fail.

printHorizontalLabel(0, _):- nl.
printHorizontalLabel(N, Total):-
        Pos is (Total-N),
        getLabel(Pos, L),
        write('   '), write(L), write(' '),
        N1 is (N-1),
        printHorizontalLabel(N1, Total).

printVerticalLabel(CurrentL):-
        CurrentL < 10,
        write(CurrentL),
        write(' ').

printVerticalLabel(CurrentL):-
        write(CurrentL).
```

Representação de um tabuleiro, usando o código mencionado:

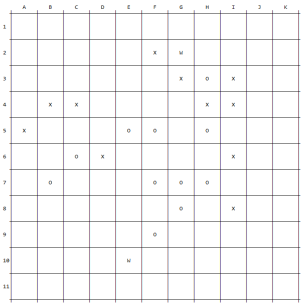

Figura 9: Representação de um tabuleiro na consola

3.3 Lista de Jogadas Válidas

Obtenção de uma lista de jogadas possíveis. Exemplo: *valid_moves(+Board, -ListOfMoves).*

3.4 Execução de Jogadas

Validação e execução de uma jogada num tabuleiro, obtendo o novo estado do jogo. Exemplo: *move(+Move, +Board, -NewBoard)*.

3.5 Avaliação do Tabuleiro

Avaliação do estado do jogo, que permitirá comparar a aplicação das diversas jogadas disponíveis. Exemplo: *value(+Board, +Player, -Value).*

A avalliação do jogo pode ser conceptualmente divida em duas partes: a análise de sequências de peças, quer da própria cor, quer de peças adversárias; e o bloqueio de sequências de peças adversárias. As peças adversárias têm cotação negativa, enquanto as peças da mesma cor tem cotação positiva.

Na primeira fase, de forma a valorizar a realização de sequências, cada peça terá valor associado igual ao cubo da posição na sequência que ocupa. Exemplificando, uma sequência de três peças na horizontal tira respetiva valorização de: (1^3) + (2^3) + (3^3), ou seja, 1 + 8 + 27; enquanto uma sequência de três peças opostas teria o valor de (-1) + (-8) + (-27). Resumindo, a análise de cada peça é feita através da formula:

- **NumSequência^3** - para peças da mesma cor;
- **- (NumSequência^3)** - para peças adversárias;

Esta análise é feita nos quatro sentidos possíveis: horizontalmente, da esquerda para a direita; verticalmente, de cima para baixo; diagonalmente, com orientação ascendente; diagonalmente, com orientação descente.

A segunda fase, surgiu como nessecidade de a peça não permitir a realização de sequências por parte do adversário. Se apenas se usasse a primeira fase de avaliação as peças nunca tentarima quebrar sequências inimigas mas sim realizar as suas sequências. Assim, para evitar que isso aconteça, um peça que quebre uma sequência inimiga, vale tanto como a sequência que quebra. Esta avaliação é feita analisando as peças adjacentes a cada peça da mesma cor, e, se a peça adjacente adjacente for inimiga, é analisa-se a peça seguinta nessa linha, sendo que no final é adicionado o valor da sequência adversária. Exemplificando, uma sequência de três peças inimigas , a adição de uma peça que quebra esse sequência teria o valor de 1 + 4 + 27. Resumindo, a análise do bloqueio de sequências adversárias é feita através da fórmula (para cada sequência quebrada):

- **(NumSequênciaAdversária^3) * factorDefesa**

De destacar, que após vários testes o factor de Defesa foi definido para 1 sendo este o valor que maior equilíbrio dá entre 'ataque' e 'defesa'.

A avaliação do tabuleiro é feita através da chamada do predicado:

- **evaluateBoard(Side, Board, BoardValue)**

Que por sua vez usa os predicados:

- **horizontalEvaluation(Side, Board, HorizontalValue)**
- **verticalEvaluation(Side, Board, VerticalValue)**
- **diagonalEvaluation(Side, Board, DiagonalValue)**
- **defensiveEvaluation(Side, Board, DefensiveValue)**

3.6 Final do Jogo

Verificação do fim do jogo, com identificação do vencedor. Exemplo: *game_over(+Board, -Winner).*

Existem três maneiras possíveis de terminação do jogo: a inexistência de posições que sejam a interseção da linha de visão dos Trabalhadores; o tabuleiro estar cheio e não ser possível mover um Trabalhador; a existência de cinco em linha de um tipos de peças. Assim, internamente, a verificação destas condições é feita de três formas distintas:

- No caso de uma jogada feita pelo utlizador, após a colocação dos Trabalhadores no tabuleiro é verificado se é possível a execução de uma jogada, recorrendo ao predicado **isPiecePlayPossible(Board)**, que verifica se a interseção das linhas de visão dos dois trabalhadores não é uma lista vazia. Caso a lista seja vazia, a primeira declaração do predicado **gameLoop**, que controla o ciclo de jogo, falhará, realizando assim a segunda declaração, que declara vitorioso o adversário e termina o ciclo de jogo. No caso de uma jogada controlada automaticamente, caso a lista dos tabuleiros possíveis, correspondentes a uma jogada, seja vazia, é declarada a vitória do adversária, num processo semelhante ao executado para o jogador.

- No início de cada jogada é verificado a se o tabuleiro se encontra totalmente preenchido, não senod assim possível mexer Trbaalhadores ou posicionar peças. Esta verificação é realizada através do predicado **boardIsNotFull(Board)**, que itera pelo tabuleiro até encontrar uma posição vazia. Caso não seja possível a priemira declaração do ciclo de jogo falha e, semelhante ao referido no ponto anterior, o jogador adversário é declarado vitorioso.

- No final de cada jogada, é verificada a vitória desse jogador. Para tal, o predicado **decideNextStep** chama o predicado **gameIsWon(Side, Board)**, que, fazendo uso dos predicados **checkHorizontalWin(PieceSide, Board)**, **checkVerticalWin(PieceSide, Board)** e **checkDiagonalWin (PieceSide, Board)**, verifica se existem 5 peças em linha, significando assim a vitória do jogador atual. Caso nenhum destes predicados se verifique, o predicado **decideNextStep** volta a chamar o ciclo do jogo, par ao próximo jogador.

3.7 Jogada do Computador

Escolha da jogada a efetuar pelo computador, dependendo do nível de dificuldade. Por exemplo: *choose_move(+Level, +Board, -Move)*.

4 Interface com o Utilizador

Descrever o módulo de interface com o utilizador em modo de texto.

5 Conclusões

O presente trabalho exigiu um grande empenho por parte de ambos os elementos do grupo, tendo, no entanto, sido uma experiência recompensadora.

Consideramos que o nosso conhecimento de programação em lógica foi amplamente aumentado, tendo sido alcançado tudo o que nos propusemos a fazer no tempo requerido, e muitas vezes ultrapassado por termos encontrado soluções mais interessantes.

As maiores dificuldades encontradas no decorrer do desenvolvimento foram relacionadas com a verificação das jogadas válidas (pois o jogo impõe múltiplas restrições a esta ação), assim como na elaboração da função que avalia as jogadas possíveis. No entanto, todos os problemas foram eventualmente ultrapassados, e estamos contentes com as soluções encontradas.

Achamos importante indicar que próximo do fim do desenvolvimento tentamos aplicar o algoritmo *Minimax* para a tomada de decisão sobre a melhor jogada a escolher, no entanto tornou-se eventualmente claro que, devido às custosas verificações de validade de jogadas e de fim de jogo, esta alteração implicaria um grande tempo de espera para cada jogada do computador, tendo no fim sido escolhido um algoritmo ganancioso para a referida tomada de decisão.

Referir também que em alguns casos decidimos beneficiar legibilidade em detrimento de performance, privilegiando predicados curtos e com uso de funções *standard*, mantendo a elegância de código permitida em linguagens de alto nível como o *PROLOG*.

Em suma, apesar de *PROLOG* se pautar por um paradigma diferente do que estamos habituados, rápidamente nos habituamos e aprendemos a apreciar as facilidades e dificuldades que este apresenta, tendo culminado num trabalho no qual nos orgulhamos.

Bibliografia

[1] L. Sterling, E. Y. Shapiro, and D. H. D. Warren, *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 2010.

[2] M. Carlsson and T. Frühwirth, *SICStus Prolog User's Manual*. SICS Swedish ICT AB, 4.3.5 ed., 2016.

[3] D. Stein, "Fabrik a 'worker placement' abstract."

A Nome do Anexo

Código Prolog implementado devidamente comentado e outros elementos úteis que não sejam essenciais ao relatório.