Aplicação em Prolog para um Jogo de Tabuleiro Cequis - Sumo

Relatório Final

Universidade do Porto

Faculdade de Engenharia

FEUP

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo 45:
Duarte Duarte - ei11101
Hugo Freixo - ei11086

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

11 de Novembro de 2013

Resumo

No trabalho realizado pelo grupo foram encontrados vários problemas no percurso do seu desenvolvimento. Um dos problemas foi o facto de o tabuleiro do nosso jogo ser constituido por hexagonos e não quadrados como na generalidade dos casos. Este problema foi resolvido bastante cedo com a criação especifica da nossa lista de listas. Uma primeira linha com menos elementos resolve o problema do nosso tabuleiro, como neste exemplo:

```
[[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],
 [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, e, e, e]]
```

Outro problema que apareceu foi o facto do jogo proposto ser o Cequis e o grupo ter escolhido Cequis-Sumo. O grupo resolveu escolher continuar a implementação do Cequis-Sumo pois alguns predicados já estavam implementados e refazê-los tiraria tempo ao grupo de maneira que a finalização do Cequis seria impossível.

Conteúdo

1 Introdução

2 O Jogo Cequis - Sumo

Cequis - Sumo é um jogo de tabuleiro disputado entre dois adversários, que possuem três pedras cada. O seu objetivo é retirar a pedra objetivo (ou branca) da mesa, empurrando-a para fora do tabuleiro, antes que o jogador adversário o faça. O tabuleiro consiste num conjunto de hexágonos ligados entre si desta forma:

Os jogadores podem andar para cima, baixo, cima-direita, cima-esquerda, baixo-direita e baixo-esquerda e só podem efetuar um movimento por jogada. O jogador pode mover uma pedra ou um conjunto de pedras. Um conjunto de pedras são pedras da mesma cor que estão em espaços adjacentes. É possivel empurrar pedras adversárias, sem as retirar do tabuleiro, mas para isso é necessário um número igual ou superior de pedras que o número de pedras que vamos empurrar. Podemos empurrar a pedra objetivo juntamente com pedras adversárias, neste caso a pedra objetivo não tem "peso", ou seja não é necessário uma pedra extra para a empurrar, apenas precisamos de cobrir o número de pedras do nosso adversário que queremos empurrar.

3 Arquitetura do Sistema

O sistema está dividido em dois módulos: o módulo de Prolog, que trata de todo o processamento do jogo (desenrolar do jogo, verificação do cumprimento das regras do jogo, determinação do final do jogo) e o módulo de visualização, que ficará responsável por representar o jogo de uma forma intuitiva e apelativa, em 3D.

Para que este sistema funcione é necessário que os módulos comuniquem entre si, para isso existirá um sistema de comunicação entre os diferentes módulos. O sistema de comunicação será baseado em mensagens, por exemplo: move, init_board, reset_board, etc.

4 Lógica do Jogo

4.1 Representação do Estado do Jogo

O estado do tabuleiro será representado em Prolog recorrendo a uma lista de listas de peças válidas com a adição de uma peça fictícia para assinalar um espaço no tabuleiro sem peças.

- b - black - peça do jogador 1
- w - white - peça do jogador 2
- o - objective - peça objetivo
- e - empty - posição sem peça

Estado inicial (tabuleiro de tamanho 8):

```
[[e,    e,    e,    e],
 [e, e, e, e, e, e, e],
 [e, e, e, e, e, e, e],
 [e, e, w, w, w, e, e],
 [e, e, b, o, b, e, e],
 [e, e, e, b, e, e, e],
 [e, e, e, e, e, e, e],
 [e, e, e, e, e, e, e]]
```

Possível estado intermédio (tabuleiro de tamanho 8):

```
[[e,    e,    e,    e],
 [e, e, e, e, e, e, e],
 [e, e, e, e, e, e, o],
 [e, e, w, w, b, w, e],
 [e, e, b, e, b, b, e],
 [e, e, e, e, e, e, e],
 [e, e, e, e, e, e, e],
 [e, e, e, e, e, e, e]]
```

Possível estado final (tabuleiro de tamanho 8):

```
[[e,    e,    e,    e],
 [e, e, e, e, e, e, e],
 [e, e, e, e, e, e, w],
 [e, e, e, w, b, w, e],
 [e, e, b, e, b, e, e],
 [e, e, e, e, e, e, e],
 [e, e, e, e, e, e, e],
 [e, e, e, e, e, e, e]]
```

4.2 Visualização do Estado do Jogo

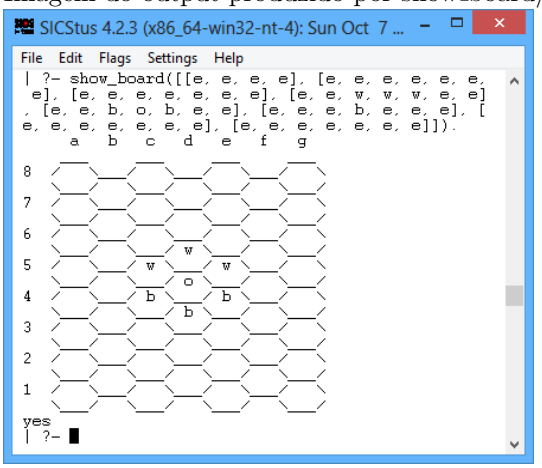
A visualização do tabuleiro em modo de texto é feita através da composição de caracteres ASCII. Para tal foi desenvolvido um conjunto de predicados, sendo show_board/1 (que aceita uma lista de listas de peças por argumento) o predicado principal. Os predicados auxiliares são: draw_piece/1, draw_letters/1,

draw_letters_aux/1, draw_line_numbers/1, draw_slashes/1, draw_lower_cell/1, show_line/2 e show_piece/2.

O predicado show_board/1 é génerico, ou seja, funciona com qualquer tamanho de tabuleiro (desde que a lista de listas de peças fornecidas seja válida).

O código destes predicados encontra-se em anexo.

Imagem do output produzido por show_board/1:



4.3 Validação de Jogadas

valid_move(+Game, +Player, +LSrc, +CSrc, +LDest, +CDest, -LiSrc, -CiSrc, -LiDest, -CiDest)

- *+Game*: lista de listas de peças
- *+Player*: jogador que executou o movimento (*player1, player2, computer1, computer2*)
- *+LSrc,+CSrc*: posição da peça a mover (linha e coluna) escolhida pelo utilizador
- *+LDest,+CDest*: posição de destino da peça (linha e coluna) escolhida pelo utilizador
- *-LiSrc,-CiSrc*: posição efectiva da peça a mover (linha e coluna)
- *-LiDest,-CiDest*: posição efectiva de destino da peça (linha e coluna)

O predicado começa por converter a posição das peças escolhidas pelo utilizador (a-z, X-0) em índices (1-X) usados internamente, verificando se as posições estão dentro dos limites do tabuleiro (*column_to_index/4* e *line_to_index/4*). Depois, é verificado se a posição a mover e de destino estão em células adjacentes (*adjacent/4*) e por fim é verificado se a peça a mover pertence ao jogador que efetuou a jogada (*object/3* e *piece/2*).

(Nota: predicado incompleto, não é verificado o caso em que é possível mover várias peças de uma só vez.)

4.4 Execução de Jogadas

move(+Game, +MoveSrc, +MoveDest, -Game1

- *+Game*: lista de listas de peças
- *+MoveSrc*: posição da peça a mover (lista de dois elementos (linha e coluna))
- *+MoveDest*: posição de destino da peça (lista de dois elementos (linha e coluna))
- *-Game1*: lista de listas de peças alterada, com a peça a mover na nova posição

Como todas as posições das peças, mesmo as vazias, são representadas por um átomo na lista de listas, implementou-se um método que troca a posição de duas peças: *swap_piece/4*, o que facilita a implementação do predicado *move/4*.

(Nota: apenas foi implementado o movimento em que a posição de destino está vazia)

4.5 Lista de Jogadas Válidas

(Nota: não implementado.)

4.6 Avaliação do Tabuleiro

(Nota: não implementado.)

4.7 Final do Jogo

game_over(+Game, +Player, -Result)

- *+Game*: lista de listas de peças
- *+Player*: jogador que efectuou a última jogada
- *-Result*: resultado do jogo, com indicação do vecendor

Este predicado retorna *no* quando o jogo ainda não terminou e *yes* quando o jogo terminou, sendo que *Result* é unificado com o nome do jogador vencedor. O predicado precorre recursivamente a lista de listas de peças procurando a peça objetivo com o predicado *member/2*, falhando se a encontrar.

4.8 Jogada do Computador

(Nota: não implementado.)

5 Interface com o Utilizador

A representação do tabuleiro foi feita como indicada no ponto 4.2 Visualização do Tabuleiro.

O input do utilizador é tratado pelo predicado *choose_move_player/4* e *choose_move_player_aux/8*, que apenas termina quando o utilizador introduz posições válidas. Após este passo, o turno é passado ao jogador seguinte utilizando o predicado *next_player/3*, que tem em conta o modo de jogo.

6 Conclusões e Perspectivas de Desenvolvimento

Em suma, o trabalho elaborado foi um desafio para o grupo e algumas partes do processo de desenvolvimento mostraram-se complicadas de concluir. O grupo encontrou alguma dificuldade a lidar com a linguagem PLOG apresentada.

Estes problemas levaram a que o trabalho não estivesse acabado na sua totalidade mas a maior parte deste está finalizada.

Bibliografia

[1] cequis.ca, http://www.cequis.ca/variations/Sumo.php/, 13 10 2013.

A Código Prolog

```
%%  Prolog implementation of Cequis - Sumo   %%
%%        FEUP - MIEIC - 2013/2014           %%
%%                                           %%
%% Authors:                                  %%
%% - Duarte Duarte - ei11101@fe.up.pt        %%
%% - Hugo Freixo   - ei11086@fe.up.pt        %%
%%                                           %%
%% Instructions:                             %%
%% - Call play predicate (e.g "play.")       %%
%% - ...                                     %%
%% - Profit!                                 %%
%%                                           %%
%%      a   b   c   d   e   f   g            %%
%%     ___     ___     ___     ___           %%
%% 8  /   \___/   \___/   \___/   \          %%
%%    \___/   \___/   \___/   \___/          %%
%% 7  /   \___/   \___/   \___/   \          %%
%%    \___/   \___/   \___/   \___/          %%
%% 6  /   \___/   \___/   \___/   \          %%
%%    \___/   \___/ w \___/   \___/          %%
%% 5  /   \___/ w \___/ w \___/   \          %%
%%    \___/   \___/ o \___/   \___/          %%
%% 4  /   \___/ b \___/ b \___/   \          %%
%%    \___/   \___/ b \___/   \___/          %%
%% 3  /   \___/   \___/   \___/   \          %%
%%    \___/   \___/   \___/   \___/          %%
%% 2  /   \___/   \___/   \___/   \          %%
%%    \___/   \___/   \___/   \___/          %%
%% 1  /   \___/   \___/   \___/   \          %%
%%    \___/   \___/   \___/   \___/          %%
%%                                           %%
%% [[e,    e,    e,    e],                   %%
%%  [e, e, e, e, e, e, e],                   %%
%%  [e, e, e, e, e, e, e],                   %%
%%  [e, e, w, w, w, e, e],                   %%
%%  [e, e, b, o, b, e, e],                   %%
%%  [e, e, e, b, e, e, e],                   %%
%%  [e, e, e, e, e, e, e],                   %%
%%  [e, e, e, e, e, e, e]]                   %%
%%                                           %%
%%  b - black (P1 pieces)                    %%
%%  w - white (P2 pieces)                    %%
%%  e - empty (nothing)                      %%
%%  o - objective (objective piece)          %%
%%                                           %%
%% http://www.cequis.ca/variations/Sumo.php  %%
%%                                           %%

library(lists).
% use_module(library(lists)).
```

```
empty_piece(e).
p1_piece(w).
p2_piece(b).
obj_piece(o).

player(player1).
player(player2).
player(computer1).
player(computer2).

piece(player1, w).
piece(player2, b).
piece(computer1, w).
piece(computer2, b).

gmode(pVSp). %% player vs player %%
gmode(pVSc). %% player vs computer %%
gmode(cVSc). %% computer vs computer %%

%% board creation - main method is create_board(N, B) - N has to be even %%

create_board(N, B) :-
    length(B, N),
    create_board_aux(N, 0, B),
    init_board(N, B).

init_board(N, B) :-
    obj_piece(ObjPiece),
    p1_piece(P1Piece),
    p2_piece(P2Piece),
    empty_piece(Piece),
    No2m2 is N // 2 - 2,
    No2m1 is N // 2 - 1,
    No2 is N // 2,
    No2p1 is N // 2 + 1,
    nth0(No2, B, MidRow), nth0(No2m1, MidRow, ObjPiece),
                          nth0(No2, MidRow, P2Piece),
                          nth0(No2m2, MidRow, P2Piece),
    nth0(No2p1, B, MidRowp1), nth0(No2m1, MidRowp1, P2Piece),
    nth0(No2m1, B, MidRowm1), nth0(No2, MidRowm1, P1Piece),
                              nth0(No2m1, MidRowm1, P1Piece),
                              nth0(No2m2, MidRowm1, P1Piece),
    init_row(B, Piece).

create_board_aux(N, N, _).
create_board_aux(N, NL, B) :-
    create_row(N, NL, Row),
    nth0(NL, B, Row),
    NL1 is NL + 1,
    create_board_aux(N, NL1, B).
```

```
create_row(N, 0, Row) :-
    !, No2 is N // 2,
    length(Row, No2).
create_row(N, _, Row) :-
    Nm1 is N - 1,
    length(Row, Nm1).

init_row([], _).
init_row([B|BT], Piece) :-
    init_row_aux(B, Piece),
    init_row(BT, Piece).

init_row_aux([], _).
init_row_aux([H|HT], Piece) :-
        nonvar(H),
        init_row_aux(HT, Piece).
init_row_aux([Piece|HT], Piece) :-
    init_row_aux(HT, Piece).

%% board representation - main method is show_board(B) %%

show_board(B) :-
    length(B, N),
    Nm1 is N - 1,
    No2 is N // 2,
    write('    '), draw_letters(Nm1), nl,
    write('  '), draw_slashes(No2), nl,
    show_line(B, 0),
    write('  '), draw_lower_cell(No2), nl, !.

draw_piece(x) :- write('err').
draw_piece(b) :- write(' b ').
draw_piece(w) :- write(' w ').
draw_piece(o) :- write(' o ').
draw_piece(e) :- write('   ').
draw_piece(N) :- write(' '), write(N), write(' ').

letters([a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z]).

draw_letters(N) :-
    letters(L),
    sublist(L, SL, 0, N, _),
    draw_letters_aux(SL).

draw_letters_aux([]).
draw_letters_aux([H|T]) :-
    write(' '), write(H), write('  '),
    draw_letters_aux(T).

draw_line_number(N) :-
    N >= 0,
    N < 10,
```

```
    write(N), write('  ').
draw_line_number(N) :-
    N >= 10,
    N < 100,
    write(N), write(' ').
draw_line_number(N) :-
    N >= 100,
    N < 1000,
    write(N).

draw_slashes(0).
draw_slashes(N) :-
    write('  ___   '),
    N1 is N - 1,
    draw_slashes(N1).

draw_lower_cell(0).
draw_lower_cell(N) :-
    write(' \\___/  '),
    N1 is N - 1,
    draw_lower_cell(N1).

show_line([], _).
show_line([BH|BT], LINEN) :-
    even(LINEN),
    LINEN1 is LINEN + 1,
    length([BH|BT], X),
    draw_line_number(X),
    show_piece(BH, LINEN, 0), nl,
    show_line(BT, LINEN1).
show_line([BH|BT], LINEN) :-
    odd(LINEN),
    LINEN1 is LINEN + 1,
    write('   '),
    show_piece(BH, LINEN, 0), nl,
    show_line([BH|BT], LINEN1).

show_piece([], _, _).
show_piece(L, 0, PIECEN) :-
    odd(PIECEN),
    write('___'),
    PIECEN1 is PIECEN + 1,
    show_piece(L, 0, PIECEN1).
show_piece([LH|LT], LINEN, PIECEN) :-
    even(LINEN),
    even(PIECEN),
    write('/'), draw_piece(LH), write('\\'),
    PIECEN1 is PIECEN + 1,
    show_piece(LT, LINEN, PIECEN1).
show_piece([_|LT], LINEN, PIECEN) :-
    even(LINEN),
    odd(PIECEN),
```

```
    write('___'),
    PIECEN1 is PIECEN + 1,
    show_piece(LT, LINEN, PIECEN1).
show_piece([_|LT], LINEN, PIECEN) :-
    odd(LINEN),
    even(PIECEN),
    write('\\___/'),
    PIECEN1 is PIECEN + 1,
    show_piece(LT, LINEN, PIECEN1).
show_piece([LH|LT], LINEN, PIECEN) :-
    odd(LINEN),
    odd(PIECEN),
    draw_piece(LH),
    PIECEN1 is PIECEN + 1,
    show_piece(LT, LINEN, PIECEN1).

%% game progression - main method is play %%

play :- %% TODO: (8, player1, pVSp), ask board size, starting player and game mode %%
    create_board(8, Game),
    show_board(Game),
    play(Game, player1, pVSp, _).

game_over([], Player, Result) :-
    Result = Player.
game_over([Line|_], _, _) :-
    obj_piece(ObjPiece),
    member(ObjPiece, Line), !, fail.
game_over([_|Lines], Player, Result) :-
    game_over(Lines, Player, Result).

announce(Result) :-
    write('Game over, '),
    write(Result),
    write(' won!'), nl.

play(Game, Player, _, Result) :-
    game_over(Game, Player, Result), !, announce(Result).
play(Game, Player, Mode, Result) :-
    write(Player), write(' turn!'), nl,
    choose_move(Game, Player, MoveSrc, MoveDest),
    move(Game, MoveSrc, MoveDest, Game1),
    show_board(Game1),
    next_player(Mode, Player, NextPlayer),
    !, play(Game1, NextPlayer, Mode, Result).

next_player(pVSp, player1, player2).
next_player(pVSp, player2, player1).
next_player(pVSc, player1, computer1).
next_player(pVSc, computer1, player1).
next_player(cVSc, computer1, computer2).
next_player(cVSc, computer2, computer1).
```

```
%% movement / validation %%

position(Position, Line, Column) :-
    nth0(0, Position, Line),
    nth0(1, Position, Column).

object(Game, Position, X) :-
    position(Position, LN, CN),
    nth0(LN, Game, Line),
    nth0(CN, Line, X).

replace_piece(Game, Piece, Position, Game1) :-
    position(Position, L, C),
    nth0(L, Game, Line),
    replace(Line, C, Piece, Line1),
    replace(Game, L, Line1, Game1).

swap_piece(Game, Position1, Position2, Game1) :-
    object(Game, Position1, Piece1),
    object(Game, Position2, Piece2),
    replace_piece(Game,   Piece1, Position2, Game11),
    replace_piece(Game11, Piece2, Position1, Game1).

move(Game, MoveSrc, MoveDest, Game1) :-
    object(Game, MoveDest, PieceDest),
    PieceDest = e,
    swap_piece(Game, MoveSrc, MoveDest, Game1).

choose_move_player(Game, Player, MoveSrc, MoveDest) :-
    write('Move Source Line (number): '),
    get_char(LSrc), skip_line,
    write('Move Source Column (letter): '),
    get_char(CSrc), skip_line,
    write('Move Dest Line (number): '),
    get_char(LDest), skip_line,
    write('Move Dest Column (letter): '),
    get_char(CDest), skip_line,
    choose_move_player_aux(Game, Player, LSrc, CSrc, LDest, CDest, MoveSrc, MoveDest).

choose_move_player_aux(Game, Player, LSrc, CSrc, LDest, CDest, MoveSrc, MoveDest) :-
    \+valid_move(Game, Player, LSrc, CSrc, LDest, CDest, _, _, _, _),
    write('Invalid '), write(Player), write(' move!!!'), nl,
    choose_move_player(Game, Player, MoveSrc, MoveDest).
choose_move_player_aux(Game, Player, LSrc, CSrc, LDest, CDest, MoveSrc, MoveDest) :-
    valid_move(Game, Player, LSrc, CSrc, LDest, CDest, LiSrc, CiSrc, LiDest, CiDest),
    MoveSrc = [LiSrc, CiSrc],
    MoveDest = [LiDest, CiDest].

valid_move(Game, Player, LSrc, CSrc, LDest, CDest, LiSrc, CiSrc, LiDest, CiDest) :-
    line_to_index(Game, LSrc, LiSrc),
    column_to_index(Game, LiSrc, CSrc, CiSrc),
```

```
    line_to_index(Game, LDest, LiDest),
    column_to_index(Game, LiDest, CDest, CiDest),
    adjacent(LiSrc, CiSrc, LiDest, CiDest),
    object(Game, [LiSrc, CiSrc], Piece),
    piece(Player, Piece).                   %% todo: stuff missing here %%

column_to_index(Game, 0, C, I) :-
    letters(L),
    nth0(Ii, L, C),
    I is Ii // 2,
    length(Game, Len),
    I < Len // 2.
column_to_index(Game, _, C, I) :-
    letters(L),
    nth0(I, L, C),
    length(Game, Len),
    I < Len.

line_to_index(Game, L, I) :-
    number_chars(Li, [L]),
    length(Game, Len),
    I is Len - Li,
    I > 0.

adjacent(Line, Column, LineTo, Column) :-
LineTo is Line + 1.
adjacent(Line, Column, LineTo, Column) :-
LineTo is Line - 1.
adjacent(Line, Column, Line, ColumnTo) :-
ColumnTo = Column + 1.
adjacent(Line, Column, Line, ColumnTo) :-
ColumnTo is Column - 1.
adjacent(Line, Column, LineTo, ColumnTo) :-
odd(Column),
LineTo is Line + 1,
ColumnTo is Column + 1.
adjacent(Line, Column, LineTo, ColumnTo) :-
odd(Column),
LineTo is Line + 1,
ColumnTo is Column - 1.
adjacent(Line, Column, LineTo, ColumnTo) :-
even(Column),
LineTo is Line - 1,
ColumnTo is Column + 1.
adjacent(Line, Column, LineTo, ColumnTo) :-
even(Column),
LineTo is Line - 1,
ColumnTo is Column - 1.

choose_move(Game, player1, MoveSrc, MoveDest) :-
    choose_move_player(Game, player1, MoveSrc, MoveDest).
```

```
choose_move(Game, player2, MoveSrc, MoveDest) :-
    choose_move_player(Game, player2, MoveSrc, MoveDest).

choose_move(Game, computer1, MoveSrc, MoveDest) :-
    choose_move_computer(Game, computer1, MoveSrc, MoveDest).

choose_move(Game, computer2, MoveSrc, MoveDest) :-
    choose_move_computer(Game, computer2, MoveSrc, MoveDest).

%% choose_move(Position, computer, Move) :-
%%     findall(M, move(Position, M), Moves),
%%     evaluate_and_choose(Moves, Position, (nil, -1000), Move).
%%
%% evaluate_and_choose([Move|Moves], Position, Record, BestMove) :-
%%     move(Move, Position, Position1),
%%     value(Position1, Value),
%%     update(Move, Value, Record, Record1),
%%     evaluate_and_choose(Moves, Position, Record1, BestMove).
%% evaluate_and_choose([], Position, (Move, Value), Move).
%%
%% update(Move, Value, (Move1, Value1), (Move1, Value1)) :-
%%     Value =< Value1.
%% update(Move, Value, (Move1, Value1), (Move, Value)) :-
%%     Value > Value1.

%% helpers %%

even(N) :- N mod 2 =:= 0.
odd(N) :- \+ even(N).

replace([_|T], 0, X, [X|T]).
replace([H|T], I, X, [H|R]):- I > 0, I1 is I-1, replace(T, I1, X, R).

%% show_board([[e, e, e, e], [e, e, e, e, e, e, e], [e, e, e, e, e, e, e], [e, e, w, w, w,
```