SQUEX

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo Squex_1:
David Luís Dias da Silva - up201705373@fe.up.pt
Luís Pedro Pereira Lopes Mascarenhas Cunha - up201706736@fe.up.pt

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

Novembro, 2019

Resumo

O trabalho aqui descrito consiste numa implementação do jogo de tabuleiro *Squex* usando a linguagem de programação *Prolog*, realizado com o objetivo de aplicar os conhecimentos adquiridos nas aulas teóricas e práticas da unidade curricular Programação em Lógica. A implementação consiste num conjunto de predicados capazes de gerar, representar e manipular um estado de jogo. A estes predicados juntam-se outros que servem para desenhar o jogo na consola e pedir *input* ao utilizador, de modo a ser possível construir um programa de fácil utilização, que permite ao utilizador realizar uma partida entre dois humanos, jogar contra o computador e observar uma partida entre dois jogadores controlados pelo computador.

A abordagem utilizada para resolver o problema teve em vista aproveitar a declaratividade própria do *Prolog*, bem como escrever código algo modular, separando diferentes responsabilidades por diferentes ficheiros e criando predicados que, embora coerentes, fossem independentes entre si, de modo a obter uma espécie de *API* que pudesse processar e obter informação sobre vários estados de jogo independentes.

As características previstas foram todas implementadas, tendo sido corretamente implementadas todas as regras do jogo e sendo possíveis 4 modos de utilização (jogador contra jogador, computador contra computador, jogador contra computador e computador contra jogador), com *bots* de dois níveis diferentes de dificuldade, em tabuleiros de tamanho variável.

A realização do trabalho foi interessante, pois estivemos em contacto com um paradigma de programação completamente diferente daquilo a que estávamos habituados. Concluímos que a programação em lógica é bastante poderosa, e que permite resolver facilmente problemas que seriam mais complexos em programação imperativa (um bom exemplo disso é a facilidade com que se obtém uma lista de jogadas válidas), embora possa levar a programas muito ineficientes.

Conteúdo

1 Introdução

Na unidade curricular de Programação em Lógica, foi-nos introduzida a linguagem de programação *Prolog*, cujo paradigma de programação nos era totalmente desconhecido, começando pelo facto de não ser uma linguagem imperativa e, por forma a aplicar os conhecimentos adquiridos nas aulas, foi-nos proposto desenvolver um jogo de tabuleiro. O objetivo era criar uma aplicação intuitiva e suficientemente robusta que permitisse o utilizador jogar *Squex* em tabuleiros de tamanho à escolha, tanto contra um adversário humano, como contra um adversário controlado pelo computador, e ainda observar uma partida entre dois jogadores controlados pelo computador, tendo estes dois níveis diferentes de dificuldade.
Neste relatório, pode encontrar-se informação sobre o jogo e as suas regras na secção 2, bem como informação detalhada sobre a nossa implementação na secção 3. A secção 3 está dividida em subsecções, cada uma referente a um tarefa particular relevante para a implementação do jogo. Na subsecção 3.1 está descrito o modo como representamos o estado do jogo, na subsecção 3.2 está descrita a implementação da interface com o utilizador. Na subsecção 3.3 está descrita a estratégia para obter um as jogadas válidas num determinado estado do jogo. Na subsecção 3.4 está descrito como é realizada uma jogada bem como a atualização do estado do jogo. Na subsecção 3.5 está descrito como detetamos que um determinado estado de jogo corresponde a um estado final. Na subsecção 3.6 descrevemos a estratégia adotada para avaliar um estado de jogo, *i. e.*, o quão favorável ele é para um jogador chegar à vitória. Na subsecção 3.7 descrevemos a implementação das jogadas realizadas pelo computador, sendo estas de dois tipos, uma delas aleatória, e outra procurando maximizar o valor do estado de jogo seguinte, seguindo uma heurística *greedy*. Por fim, na secção 4 encontram-se reflexões sobre o trabalho desenvolvido, nomeadamente sobre aprendizagens e dificuldades.
Para o desenvolvimento do trabalho foi utilizado o ambiente de *SWI-Prolog* pelo que se recomenda o seu uso para testar o código.

2 O Jogo Squex

Squex (*Square Hex*) é um jogo de tabuleiro para dois jogadores, inspirado no *Hex*, jogado num tabuleiro de 8×8 octógonos, ligados entre si por quadrados. Os octógonos estão delimitados por quadrados, acima e abaixo de uma cor, e dos lados de outra. Cada cor corresponde à cor das peças que cada jogador utiliza. No início do jogo, o tabuleiro encontra-se vazio, como na figura 1. Cada jogador joga alternadamente, colocando peças octogonais em qualquer célula octogonal vazia do tabuleiro, com o objetivo de unir os lados opostos do tabuleiro que têm quadrados da cor que lhe corresponde com uma linha contínua de octógonos e quadrados da mesma cor que não possa ser cortada pelo adversário.

2.1 Regras

1. Os jogadores jogam alternadamente, colocando uma peça octogonal da cor que lhe corresponde em qualquer uma das células octogonais do tabuleiro que estejam vazias.

2. Quando uma peça é colocada na diagonal de outra do mesmo jogador,

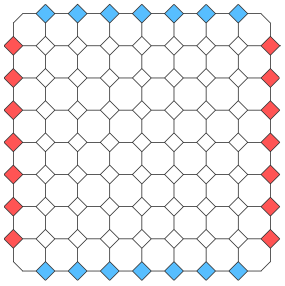
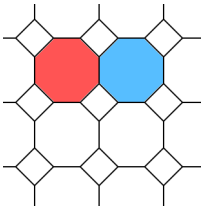
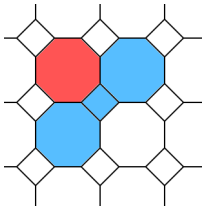


Figura 1: Estado inicial do tabuleiro.

uma peça quadrangular da mesma cor é colocada na célula quadrangular do tabuleiro entre os dois octógonos, como no exemplo da figura 2.



(a) Antes da jogada.



(b) Depois da jogada.

Figura 2: Exemplo da regra número 2.

3. Quando o descrito pela regra número 2 acontece e uma peça do jogador adversário já se encontra a ocupar o quadrado, a peça é substituída por uma peça do jogador que acabou de jogar. No resto do relatório, esta regra será referida como *cut*.

4. Depois de um *cut*, o jogador que o sofreu recebe duas jogadas consecutivas. Isto também se verifica caso um jogador realize um *cut* na primeira jogada após ter recebido duas, em que o jogador perde a segunda jogada e o adversário recebe dois turnos consecutivos.

5. Um jogador vence o jogo apenas quando consegue criar um caminho entre lados opostos correspondentes que não pode ser cortado (regra *cut*) pelo adversário. A figura 3 é um exemplo do descrito.

3 Lógica do Jogo

3.1 Representação do Estado do Jogo

De modo a representar o estado do jogo, é necessário informação sobre o tamanho do tabuleiro e as peças nele presentes, o tipo de cada jogador, qual o próximo jogador a efetuar uma jogada e sobre a regra *cut*. O código referente ao estado do jogo encontra-se no ficheiro *game_model.pl*.

Apesar do tabuleiro ter, originalmente, 8×8 octógonos, de modo a tornar a implementação mais interessante, decidimos tornar o tamanho do tabuleiro customizável ($m \times n$ octógonos).

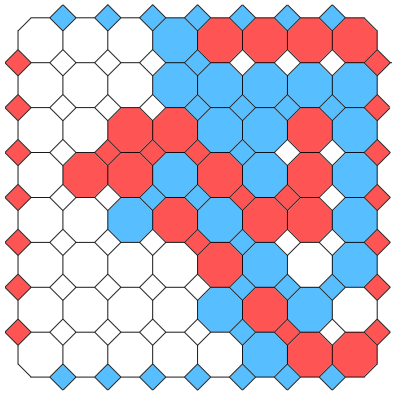


Figura 3: O jogador azul ainda não venceu a partida, pois o jogador vermelho tem possibilidade de cortar o caminho junto ao fundo do tabuleiro.

Para representar o tabuleiro, considerámos duas hipóteses, ambas usando listas de listas. Em ambas, cada elemento das matrizes do tabuleiro representa o estado de cada célula do tabuleiro. O valor 0 representa célula vazia, os valores 1 e 2 correspondem a peças jogadas pelos jogadores 1 e 2, respetivamente. Numa hipótese, o estado do tabuleiro seria guardado apenas numa matriz $(2m+1) \times (n+1)$ com quadrados e octógonos em linhas alternadas. Nesta hipótese, as linhas correspondentes aos octógonos teriam um elemento que não seria utilizado, pois, por linha, o número de quadrados é maior que o de octógonos (daí o número de colunas ser $n + 1$).
A segunda hipótese consiste em guardar duas matrizes diferentes, uma $m \times n$ com informação referente às células octogonais e uma $(m+1) \times (n+1)$ com informação referente às células quadrangulares. Esta hipótese foi escolhida pois, para além de facilitar a tarefa de desenhar o tabuleiro e oferecer alguma modularidade ao separar os conceitos de células de diferentes tipos, faz com que aceder aos quadrados ajdacentes a um octógono seja simples. Como pode ser observado na figura 4, para o octógono no índice (a, b) da matriz de octógonos, os quadrados adjacentes estão, na matriz de quadrados, nos índices (a, b), $(a+1, b)$, $(a, b+1)$ e $(a + 1, b + 1)$.

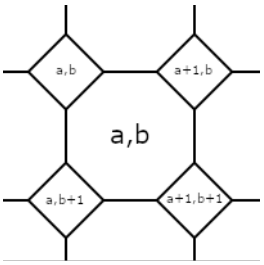


Figura 4: Coordenadas dos quadrados adjacentes a um octógono.

Em relação ao tabuleiro, é ainda guardado a sua altura e largura (número de linhas e colunas, respetivamente, da matriz de octógonos), de modo a não ser necessário calcular caso esta informação seja necessária em algum predicado.

O tipo de cada jogador é representado, no caso de se tratar de um humano,

pelo átomo 'P', e no caso de se tratar de um jogador controlado pelo computador, pelo número referente ao seu nível de dificuldade (1 no caso do nível 1, 2 no caso do nível 2).

O jogador que tem o próximo turno é representado pelo número correspondente (1 para o jogador 1, 2 para o jogador 2).

A informação relativa ao *cut* é essencial de modo a permitir que um jogador faça duas jogadas consecutivas, bem como, pela informação do estado do jogo, perceber qual foi o jogador que jogou anteriormente (*i.e.* se um jogador está na segunda jogada após sofrer um *cut*, ou na sua primeira jogada), necessário no predicado que verifica o fim do jogo. Esta informação é representada por um tuplo do tipo A-B, em que A corresponde ao número de jogadas consecutivas que restam ao próximo jogador ($A \in \{1, 2\}$) e B representa se o próximo jogador sofreu um *cut* na última jogada do adversário ($B \in \{0, 1\}$, 1 representa o valor de verdade, 0 o valor de falso).

O estado do jogo é, de modo a facilitar a sua passagem para os diversos predicados que o usem, representado por uma lista onde se encontram os vários elementos necessários à sua definição. Esta lista é preenchida no início do jogo recorrendo ao predicado *generate_initial_game_state(+Width, +Height, +P1Type, +P2Type, -GameState)*, que, usando os predicados *generate_octagon_board(+Width, +Height, -OctagonBoard)* e *generate_square_board(+Width, +Height, -SquareBoard)*, geram as listas de listas que representam o tabuleiro com tamanho variável. No ficheiro *game_model.pl* existem ainda predicados que servem para, de forma mais legível, obter os atributos ou um atributo específico da lista que representa o estado de jogo, como por exemplo *get_game_current_player_type(+GameState, -Type)* e *get_game_board_size(+GameState, -Height, -Width)*, e também predicados que permitem inserir e obter elementos numa determinada posição numa lista de listas.

3.1.1 Exemplos de Estados do Jogo

No estado inicial, a matriz de octógonos tem todos os elementos a 0 e a matriz de quadrados tem a primeira e última linhas preenchidas com 1's e a primeira e última colunas estão preenchidas com 2's. É importante notar que as posições inicialmente preenchidas com 1's e 2's na matriz de quadrados não mudam com o decorrer do jogo e que os cantos da matriz estão sempre a 0 e são irrelevantes para o jogo. A informação relativa ao *cut*, no início do jogo, é $1 - 0$. O 1 significa que o jogador vai realizar apenas uma jogada e o 0 significa que não sofreu *cut* na jogada imediatamente anterior do outro jogador. No caso de um jogo entre um humano e um *bot* nível 2 num tabuleiro 8×8, a lista que representa o estado inicial de jogo seria a seguinte:

```
1     [[[0, 0, 0, 0, 0, 0, 0, 0], % Tabuleiro de octógonos
2       [0, 0, 0, 0, 0, 0, 0, 0],
3       [0, 0, 0, 0, 0, 0, 0, 0],
4       [0, 0, 0, 0, 0, 0, 0, 0],
5       [0, 0, 0, 0, 0, 0, 0, 0],
6       [0, 0, 0, 0, 0, 0, 0, 0],
7       [0, 0, 0, 0, 0, 0, 0, 0],
```

```
      [0, 0, 0, 0, 0, 0, 0, 0]],
     [[0, 1, 1, 1, 1, 1, 1, 1, 0], % Tabuleiro de quadrados
      [2, 0, 0, 0, 0, 0, 0, 0, 2],
      [2, 0, 0, 0, 0, 0, 0, 0, 2],
      [2, 0, 0, 0, 0, 0, 0, 0, 2],
      [2, 0, 0, 0, 0, 0, 0, 0, 2],
      [2, 0, 0, 0, 0, 0, 0, 0, 2],
      [2, 0, 0, 0, 0, 0, 0, 0, 2],
      [2, 0, 0, 0, 0, 0, 0, 0, 2],
      [0, 1, 1, 1, 1, 1, 1, 1, 0]],
      8,    % Altura do tabuleiro
      8,    % Largura do tabuleiro
      'P',  % Tipo do jogador 1
      2,    % Tipo do jogador 2
      1,    % Próximo jogador
      1-0]. % Informação sobre o cut
```

Código fonte 1: Estado inicial do jogo.

Um estado intermédio possível é o de uma partida entre dois humanos num tabuleiro 5×6, após o jogador 1 fazer um *cut* ao jogador 2. A informação relativa ao *cut* é $2 - 1$. O 2 significa que o jogador 2 vai realizar duas jogadas consecutivas e o 1 significa que sofreu *cut*. A representação é a seguinte:

```
    [[[1, 0, 0, 0, 0, 0], % Tabuleiro de octógonos
      [1, 0, 0, 0, 0, 0],
      [1, 0, 0, 0, 0, 0],
      [0, 1, 2, 2, 1, 0],
      [0, 2, 1, 0, 2, 2]],
     [[0, 1, 1, 1, 1, 1, 0], % Tabuleiro de quadrados
      [2, 0, 0, 0, 0, 0, 2],
      [2, 0, 0, 0, 0, 0, 2],
      [2, 1, 0, 0, 0, 0, 2],
      [2, 0, 1, 0, 2, 0, 2],
      [0, 1, 1, 1, 1, 1, 0]],
      5,    % Altura do tabuleiro
      6,    % Largura do tabuleiro
      'P',  % Tipo do jogador 1
      'P',  % Tipo do jogador 2
      2,    % Próximo jogador
      2-1]. % Informação sobre o cut
```

Código fonte 2: Um estado intermédio possível do jogo.

Um estado final possível, no seguimento da partida descrita no exemplo anterior, em que o jogador 1 vence, é o seguinte:

```
    [[[1, 0, 1, 0, 2, 0], % Tabuleiro de octógonos
      [1, 1, 2, 0, 0, 0],
      [1, 2, 1, 0, 0, 0],
      [2, 1, 2, 2, 1, 0],
      [0, 2, 1, 1, 2, 2]],
```

```
6       [[0, 1, 1, 1, 1, 1, 0], % Tabuleiro de quadrados
7        [2, 1, 1, 0, 0, 0, 2],
8        [2, 1, 1, 0, 0, 0, 2],
9        [2, 2, 1, 0, 0, 0, 2],
10       [2, 2, 1, 0, 2, 0, 2],
11       [0, 1, 1, 1, 1, 1, 0]],
12       5,    % Altura do tabuleiro
13       6,    % Largura do tabuleiro
14       'P',  % Tipo do jogador 1
15       'P',  % Tipo do jogador 2
16       2,    % Próximo jogador
17       1-0]. % Informação sobre o cut
```

Código fonte 3: Um estado final possível do jogo, com vitória do jogador 1.

3.2 Visualização do Tabuleiro

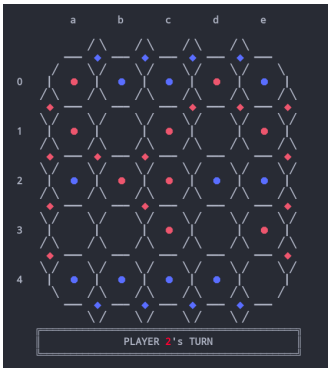
A interface do jogo com o utilizador consiste nos predicados que desenham o estado do jogo e outras informações na consola e os predicados que pedem dados de entrada ao utilizador, que se encontram, respetivamente, nos ficheiros *display.pl* e *input.pl*.
O predicado de desenho encarregue de desenhar o estado de jogo é o *display_game(+GameState)*. Este predicado imprime uma mensagem que informa se na jogada anterior ocorreu um *cut*, desenha o eixo de coordenadas horizontal, desenha o tabuleiro e imprime uma mensagem que indica qual o próximo jogador a jogar.

```
1   display_game([OctagonBoard, SquareBoard, Height, Width, _, _, Player,
    ↪ CutHappened |[]]) :-
2       display_cut_message(CutHappened), nl, nl,
3       display_horizontal_coordinates(a, Width), nl,
4       display_board(OctagonBoard, SquareBoard, 0, Height, Width), nl,
5       display_turn_message(Player, Width), nl.
```

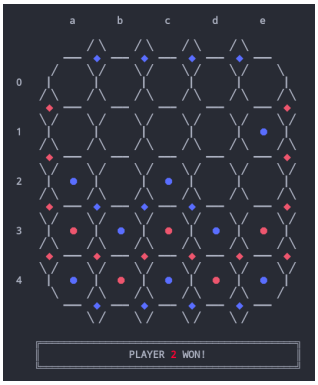
Código fonte 4: Predicado responsável por imprimir o estado de jogo.

O predicado *display_board(+OctagonBoard, +SquareBoard, +CurrentY, +Height, +Width)* é o predicado que, recursivamente e chamando alternadamente os predicados responsáveis por desenhar linhas de octógonos e linhas de quadrados, bem como os caracteres que separam as linhas, é responsável por desenhar a totalidade do tabuleiro, sendo que este pode ter tamanho variável. Cada linha é desenhada recursivamente, elemento a elemento. O terceiro argumento é utilizado para imprimir a coordenada vertical em cada linha. Cada elemento do tabuleiro é desenhado recorrendo aos predicados *display_square_piece(+Player)* e *display_octagon_piece(+Player)*, que desenham o caracter certo na correspondente ao jogador. O predicado *display_game_over(+GameState, +Player)* é semelhante ao predicado *display_board/2)*, com a diferença que em vez de imprimir uma mensagem relativamente ao próximo turno, imprime uma mensagem indicado que o jogador correspondente à variável *Player* venceu. Foram escritos alguns predicados auxiliares, como por exemplo o predicado *display_box_message/3*, utilizado para desenhar uma caixa da largura do tabuleiro

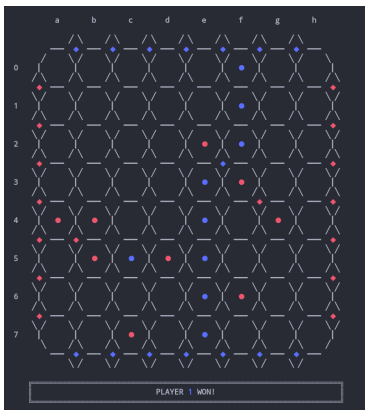
com uma mensagem centrada dentro.



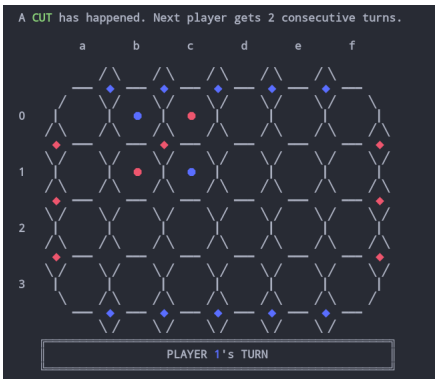
(a) Desenho do estado de jogo.



(b) Desenho de fim do jogo.



(c) Tabuleiro 8×8.



(d) Informação de *cut*.

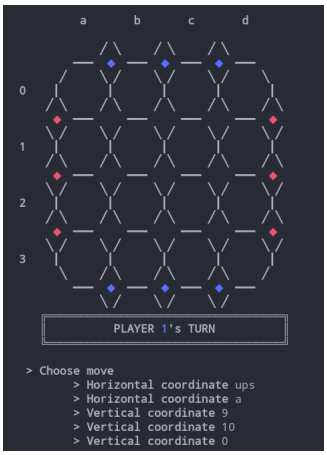
Figura 5: Validação de dados de entrada.

O predicado *display_main_screen/0* é responsável por desenhar um menu principal, que consiste em um logo feito em *ASCII art* e os modos de jogo que o utilizador pode escolher de modo a começar a jogar, como mostrado na figura 6.

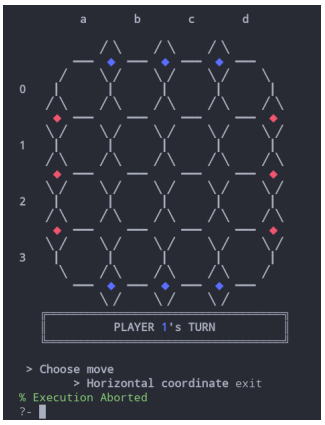
Figura 6: Menu inicial do jogo.

De modo a que o utilizador possa usufruir das funcionalidades do programa, é necessário o utilizador poder introduzir números (para escolher a opção do menu principal, nível de dificuldade do computador, as coordenadas do tabu-

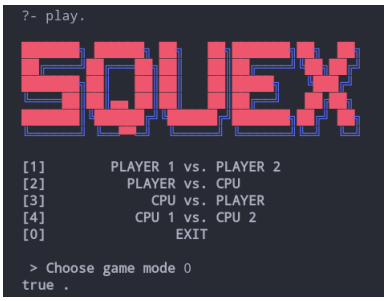
leiro onde jogar e tamanho do tabuleiro) e uma letra (que representa a coordenada horizontal de uma jogada). De modo a que o utilizador possa introduzir os dados naturalmente, e não estrague a execução do programa caso se engane, os predicados que leem um valor inteiro (*get_int/1*) e uma letra (*get_letter/1*) fazem-no carácter a carácter, usando o predicado *get_char/1* do *Prolog*. Ambos os predicados usam o predicado auxiliar *get_char_list(+CharList)*, que lê uma lista de caracteres da entrada até encontrar o carácter '
n'. Após obter a lista, esta é convertida num valor inteiro ou numa letra, respetivamente, nos predicados *get_int/1* e *get_letter/1*, fazendo a devida validação. Quando algum dado introduzido não cumpre com os requisitos, através da utilização do predicado *repeat/0*, o *prompt* é imprimido novamente e o utilizador pode reintroduzir os dados.
De modo ao utilizador poder acompanhar a execução do programa quando está a ocorrer uma partida entre dois jogadores controlados pelo computador, foi criado um predicado *press_enter_to_continue*, que pausa a execução do programa antes de cada jogada do computador até o utilizador pressionar a tecla *enter*. Após uma partida, o programa regressa ao menu principal, onde o utilizador pode sair do programa através de uma opção do menu principal. Caso pretenda sair do programa em qualquer estado da sua execução, basta introduzir a palavra *exit* quando aparece um *prompt*.



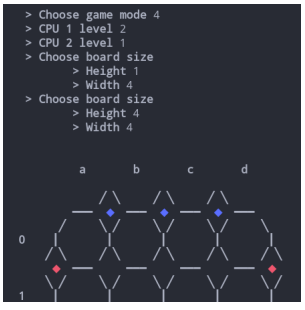
(a) Validação das coordenadas de uma jogada.



(b) Terminar a execução através de *exit*.



(c) Terminar a execução no menu principal.



(d) Validação do tamanho de um tabuleiro.

Figura 7: Desenho do tabuleiro de jogo.

3.3 Lista de Jogadas Válidas

A obtenção de uma lista com todas as jogadas válidas é uma tarefa relativamente simples, pois a única restrição para uma jogada ser válida, para além de esta se referir uma posição válida do tabuleiro, é que a célula octogonal onde se pretende inserir uma peça esteja livre. As jogadas são representadas por um tuplo na forma X-Y, sendo X e Y os índices da coluna e linha, respetivamente, da matriz de octógonos onde se pretende colocar a peça. O predicado *valid_moves(+GameState, -ListOfMoves)* recebe o estado de jogo (variável de entrada *GameState*), de onde retira a matriz que representa as células octogonais do tabuleiro, e devolve uma lista com todos os movimentos válidos (variável de saída *ListOfMoves*). O predicado *validate_move(+OctagonBoard, ?Move)* sucede caso a jogada *Move* seja válida num estado de jogo cuja matriz de octógonos seja *OctagonBoard*, e pode ser usado tanto para validar uma jogada como para obter jogadas válidas. Assim, uma maneira de obter uma lista de todas as jogadas válidas para um determinado estado de jogo consiste em usar o predicado *validate_move* como *Goal* do predicado *findall*.

```
1    valid_moves(GameState, Moves) :-
2        get_game_octagon_board(GameState, OctagonBoard),
3        findall(Move, validate_move(OctagonBoard, Move), Moves).
4
5    validate_move(OctagonBoard, Move) :-
6        board_get_element_at(OctagonBoard, Move, 0).
```

Código fonte 5: Obter lista de jogadas válidas.

3.4 Execução de Jogadas

O predicado *move(+Move, +GameState, -NewGameState)* é o predicado responsável por atualizar o estado de jogo. Para o predicado move suceder é necessário que o argumento passado na variável *Move* seja uma jogada válida para o tabuleiro do *GameState*. Seguidamente, o predicado encarrega-se de atualizar o tabuleiro que contém informação relativa às peças octogonais inserindo o número corresponde ao jogador que efetua a jogada na posição correspondente, atualizar as células quadrangulares caso ocorra o descrito na regra número 2 e atualizar a informação referente ao próximo turno.

```
1    move(Move,GameState,NewGameState) :-
2        get_game_attributes(GameState, OctagonBoard, SquareBoard, Height, Width,
         ↪  P1Type, P2Type, Player, CutInfo),
3        validate_move(OctagonBoard, Move),
4        board_insert_element_at(OctagonBoard, Move, Player, NewOctagonBoard),
5        update_squares(Player, Move, OctagonBoard, SquareBoard, NewSquareBoard,
         ↪  Height, Width, NumCuts),
6        update_next_player(Player, CutInfo, NewPlayer, NewCutInfo, NumCuts),
7        get_game_attributes(NewGameState, NewOctagonBoard, NewSquareBoard,
         ↪  Height, Width, P1Type, P2Type, NewPlayer, NewCutInfo).
```

Código fonte 6: Atualização do estado de jogo.

A verificação da legalidade de uma jogada é realizada recorrendo ao predicado *validate_move(+OctagonBoard, ?Move)* descrito na secção 3.3.

A atualização do tabuleiro de octógonos é feita usando o predicado *board_insert_element_at(+Board, +Pos, +Element, -NewBoard)* existente no ficheiro *game_model.pl*, que cria uma nova lista de listas (*NewBoard*) igual à recebida na variável *Board*, mas com o elemento recebido na variável *Element* inserido na posição *Pos* (tuplo $X - Y$).

O predicado *update_squares(+Player, +Pos, +OctagonBoard, +SquareBoard, -NewSquareBoard, +Height, +Width, -NumCuts* é responsável por atualizar a informação referente ao tabuleiro de quadrados e verificar a existência de *cuts*. O predicado começa por calcular as posições dos octógonos diagonais ao octógono onde o jogador acabou de jogar. Em seguida, e com a informação acabada de obter, calcula os quadrados adjacentes ao octógono que estão em condições de ser preenchidos, ou seja, que se situam entre o octógono onde se jogou e um octógono que tinha uma peça do mesmo jogador. Antes de efetivamente colocar as peças nos quadrados e de modo a verificar a existência de *cuts*, é preciso verificar se algum elemento da lista de quadrados a utilizar pertence ao outro jogador. Por fim, o tabuleiro de quadrados é atualizado, sendo preenchidas as posições referentes aos quadrados obtidos nos cálculos anteriores com o elemento correspondente ao jogador.

```
1    update_squares(Player, X-Y, OctagonBoard, SquareBoard, NewSquareBoard,
     ↪ Height, Width, NumCuts) :-
2        get_diagonals_pos(X-Y, Height, Width, DiagonalsPos),
3        get_squares_pos(Player, OctagonBoard, X-Y, DiagonalsPos, SquaresPos),
4        check_cut(Player, SquareBoard, SquaresPos, NumCuts),
5        place_squares(Player, SquareBoard, SquaresPos, NewSquareBoard).
```

Código fonte 7: Atualização do tabuleiro de quadrados.

Resta, de modo a concluir a atualização do estado de jogo, atualizar a informação referente ao próximo jogador a jogar e ao *cut*, utilizando o predicado *update_next_player(+Player, +CutInfo, -NewPlayer, -NewCutInfo, +NumCuts)*. Isto é efetuado tendo em conta se ocorreram ou não *cuts* nesta jogada e se o jogador está na primeira ou segunda jogada após a ocorrência de um *cut*.

3.5 Final do Jogo

Tal como é descrito no ponto 6 da secção 2.1: *"Um jogador vence o jogo apenas quando consegue criar um caminho entre lados opostos correspondentes que não pode ser cortado (regra cut) pelo adversário"*. Assim sendo, para verificar que o jogo chegou ao fim implementamos o predicado *game_over(+GameState, -Winner)*,presente no ficheiro *gameover.pl*, que testa se o jogador que efetuou a ultima jogada(*Player*) possui tal caminho. Para testar a condição, o predicado constrói um grafo que representa todas as peças octagonais pertencentes a *Player* bem como todas as suas ligações, procurando nele a existência de tal caminho. Na secção seguinte especificação-se os detalhes acerca do grafo e dos principais predicados que o implementam.

3.5.1 Grafo de ligações

O grafo de ligações de um jogador é um grafo não-dirigido e não-pesado cujos nós representam as suas peças octagonais e as arestas as ligações existentes entre octógonos adjacentes. Um octógono está adjacente a outro da mesma cor se este se encontra diretamente à sua esquerda, direita, cima ou baixo, ou seja, se os octógonos estão em contacto, ou caso se encontre na sua diagonal unido por um quadrado da mesma cor. Um exemplo destas adjacências pode ser vista na figura 8.

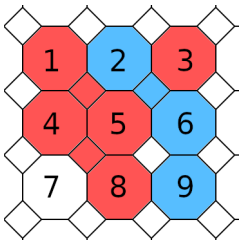


Figura 8: Adjacências de um octógono no grafo de ligações. Os octógnos adjacentes ao octógono 5 são os octógonos 1,4 e 8.

O predicado *build_graph(+BoardState,+Player,-Graph)* constrói o grafo em duas fases. A primeira, através do predicado *build_edges(+OctagonBoard, +SquareBoard, +Height, +Width, +Player, -Edges)*, processa o tabuleiro de cima para baixo e adiciona, para cada linha, as arestas das adjacências entre os octógonos da mesma linha e entre os octógonos dessa linha e da seguinte. Assim, para cada octógono de uma linha (com exceção do primeiro e do último da linha) são verificados os octógonos da direita, da esquerda, de baixo e das diagonais inferiores esquerda e direita, através de predicados como *get_child_from_belowL/7* , *get_child_from_belowC/7*, *get_child_from_belowR/7*, *get_child_from_levelL/6* e *get_child_from_levelR/6*. A segunda fase consiste me utilizar o predicado *vertices_edges_to_ugraph(+Vertices, +Edges, -Graph)* da bilbioteca *ugraphs*, para construir o grafo propriamente dito. Dado os caminhos do jogador 2 progredirem na horizontal e a construção do grafo ser na vertical, é feita a transposição do tabuleiro do mesmo antes da chamada ao predicado. Cada peça octogonal do tabuleiro é representado por um *ID* sendo que o *ID* é atribuído de cima para baixo, da esquerda para a direita começando em 0. Por exemplo, num tabuleiro 8×8 os *ID's* estão no intervalo de 0 a 63.

3.5.2 Caminho não cortável

Para averiguar se um caminho é ou não cortável a predicado, antes da construção do grafo, são "retirados"do tabuleiro de quadrados do *GameState* todos os quadrados que são cortáveis pelo jogador adversário. Esta remoção é feita a partir do predicado *remove_cuttable_squares(+OctagonBoard, +SquareBoard, +Player, -NewSquareBoard)*, que percorre o tabuleiro de quadrados de cima para baixo, aplicando o predicado *is_cuttable(+OctagonBoard, +Player, +SquareX, +SquareY)* e removendo os quadrados que validam esse mesmo predicado.

Depois da construção do grafo, é chamado o predicado *reachable_from_list(+Graph,+List,+Destinations)* com *List* contendo a lista dos ID's dos octógonos da primeira fila do tabu-

leiro com a cor do jogador e *Destinations* contendo a lista dos ID's das células da última fila do tabuleiro. Este predicado efetua uma chamada a *reachable(+Vertex, +Graph, -Vertices)* da biblioteca *ugraphs*, por cada elemento de *List* e verifica de algum elemento de *Destinations* pertence a *Vertices*. Se este predicado suceder então *game_over/2* também sucede e *Winner* guarda o número do jogador que ganhou o jogo, caso contrário o predicado falha.

O código fonte 8 mostra o predicado *check_for_win(+Gamestate, +Player)*.

```
1  check_for_win([OctagonBoard,SquareBoard,Height,Width | _] ,Player) :-
2
3      remove_cuttable_squares(OctagonBoard, SquareBoard, Player, NewSquareBoard),
4
5      % To allow for checking for both players, since the gameover predicates work
       ↪  form the top edge down, when the Player's number is equal to 2 and, as
       ↪  such,
6      % his paths increase horizontally, the board must be transposed and the
       ↪  height/width must be switched.
7      orient_board(OctagonBoard,
       ↪  NewSquareBoard,Player,OrientedOctagonBoard,OrientedSquareBoard),
8      get_real_side_lengths(Player,Width,Height,RealWidth,RealHeight),
9
10     % Get the octagons where the path may start(there are connected with the
       ↪  Player's top board edge). This function is meant to optimize the
11     % process as there is no need to build the graph(which is an operation that is
       ↪  computationally expensive) if there is no starting pice to begin with.
12     get_valid_starters(OrientedOctagonBoard,Player,0,RealWidth,Starters),
13
14     % Build the player-path graph
15
       ↪  build_graph([OrientedOctagonBoard,OrientedSquareBoard,RealHeight,RealWidth],Player,Graph),
16
17     % Check if the opposing board edge is connected to the starting one via a
       ↪  path
18     LastRowID is RealHeight -1,
19     gen_row_ids(RealWidth,LastRowID,LastRowIDs),!,
20     reachable_from_list(Graph,Starters,LastRowIDs).
```

Código fonte 8: Verificar se um jogador ganhou o jogo

3.6 Avaliação do Tabuleiro

Para avaliar o estado do jogo, é usado o predicado *value(+GameState, -Value)*, que atribui um valor numérico ao estado de jogo do ponto de vista do jogador que jogou anteriormente. Para obter este valor consideramos três fatores.

O primeiro (*SBValue1*) é o comprimento do maior sub-tabuleiro em que o jogador tem um caminho que une os lados opostos correspondentes. Para calcular este valor o predicado *value/2* utiliza o predicado *get_highest_sub_board_value(+OctagonBoard,+Width,+Heigh*
Value) (código fonte 9)que por sua vez percorre todos os sub-tabuleiros desde o comprimento igual à altura do tabuleiro até 1, sucedendo quando encontrar um sub-tabuleiro em que o jogador tem um caminho. A verificação se o jogador tem um caminho nos sub-tabuleiros é feita com auxilio aos predicados

de grafos referidas na secção 3.5.1 e de uma forma semelhante à verificação de *gameover* . É de notar que os sub-tabuleiros descritos apenas vão diminuindo em comprimento, ou seja, na direção em que o jogador progride para vencer. O intuito deste factor é incentivar a escolha de jogadas em que o comprimento do caminho do jogador aumenta na direção que leva à sua vitória.

```
1  % Smallest length of a subboard (1 tile)
2  get_highest_sub_board_value_iter(_,_,_,_,[],_,1) :-!.
3
4  % Value of an empty board is 0
5  get_highest_sub_board_value_iter(_,_,_,_,_,0,0) :-!.
6
7  get_highest_sub_board_value_iter(OctagonBoard,Width,Height,Player,Graph,CurrentBoardSize,Value)
   ↪  :-
8
9      % AlternativeCount designates the number of sub-boards of CurrentBoardSize
       ↪  that need to be checked
10     AlternativeCount is Height - CurrentBoardSize + 1,
11
12     % Check if there are any CurrentBoardSize subboards where Player wins
13
       ↪  \+check_sub_boards(OctagonBoard,Width,Height,Player,Graph,AlternativeCount,AlternativeCount),!,
14
15     NewSize is CurrentBoardSize - 1,
16
       ↪  get_highest_sub_board_value_iter(OctagonBoard,Width,Height,Player,Graph,NewSize,Value).
17
18 get_highest_sub_board_value_iter(_,_,_,_,_,CurrentBoardSize,Value) :-
19
20     % There is a subboard of size CurrentBoardSize where Player Wins
21     Value is CurrentBoardSize.
```

Código fonte 9: Encontrar tamanho do maior sub-tabuleiro em que o jogador tem um caminho que une os lados opostos correspondentes

O segundo fator(*SBValue2*) é muito semelhante ao primeiro, na medida em que também é calculado, usando o mesmo predicado, o comprimento do maior sub-tabuleiro em que o jogador efetivamente venceria o jogo, tendo em conta a regra número 5. O tabuleiro que é avaliado é primeiro processado pelo predicado *remove_cuttable_squares(+OctagonBoard, +SquareBoard, +Player, -NewSquareBoard)* de modo a que o valor de *SBValue2* tem a possibilidade de ser menor que *SBValue1*, significando que o caminho de maior comprimento pode ser cortado. Este fator incentiva jogadas que não deem origem a caminhos que possam ser cortadas pelo adversário.

O último fator, denominado de *NextPlayerValue*, é o resultado do cálculo do melhor valor da soma de *SBValue1* e *SBValue2* para todas as possíveis jogadas do jogador seguinte. Este fator pode ser positivo caso o jogador seguinte seja o jogador atual. Esta medida leva a uma tentativa de jogadas que diminuam o valor da próxima jogada do adversário (ou aumentem o valor, caso o próximo jogador seja o mesmo que o atual). O cálculo de *NextPlayerValue* pode ser visto no

código fonte 10.

```
/**
* compute_next_player_value(+GameState, +ListOfMoves, -NextPlayerValue)
*
* Find all pairs of valid moves and their corresponding values. The difference between this segment and the similar segment in greedy_move(+GameState, -Move) is that this segment uses value_next(+GameState,Value) to obtain it's values. The essential difference is that value_next does not only values according to it's state and doesn't take the next player's  move into account.
*/
compute_next_player_value(_, [], 0).
compute_next_player_value(GameState, ListOfMoves, NextPlayerValue) :-
    findall(Val-Move, (member(Move, ListOfMoves),move(Move, GameState, NewGameState), value_next(NewGameState, Val)), Result),
    keysort(Result,SortedResultAsc),
    reverse(SortedResultAsc,SortedResultDsc),
    nth0(0,SortedResultDsc,NextPlayerValue-_).
```

Código fonte 10: Calcular o valor de NextPlayerValue. O predicado *value_next* efetua os mesmos calculos que *value* no entanto apenas calcula *SBValue1* e *SBValue2*

Assim, o valor do GameState é calculado como *SBValue1 + SBValue2 - NextPlayerValue* e o predicado *value(+GameState,-Value)* pode ser visto no código fonte 11.

```
    value(GameState, Value) :- !,

    % Setup needed variables. Extract variables from gamestate, transpose the board  and switch the height/width values if needed
    GameState = [OctagonBoard,SquareBoard, DefaultHeight, DefaultWidth,_,_,_, _-_ | []],
    get_game_previous_player(GameState,PrevPlayer),
    get_real_side_lengths(PrevPlayer,DefaultWidth,DefaultHeight,Width,Height),
    orient_board(OctagonBoard, SquareBoard,PrevPlayer,OrientedOctagonBoard,OrientedSquareBoard),

   % Get SBValue1 - Highest length of the sub-board in which Player has a path between the margins

    % Build the connection graph
    build_graph([OrientedOctagonBoard, OrientedSquareBoard, Height, Width], PrevPlayer, Graph1),!,

    get_highest_sub_board_value(OrientedOctagonBoard,Width,Height,PrevPlayer,Graph1,SBValue1),

    % Get SBValue2 - Highest length of the sub-board in which Player wins the game

    % Remove cuttable squares
```

```
18
        ↪ remove_cuttable_squares(OrientedOctagonBoard,OrientedSquareBoard,PrevPlayer,NewSquareBoard),
19
20      % Build the connection graph
21      build_graph([OrientedOctagonBoard, NewSquareBoard, Height, Width],
        ↪ PrevPlayer, Graph2),!,
22
        ↪ get_highest_sub_board_value(OrientedOctagonBoard,Width,Height,PrevPlayer,Graph2,SBValue2),
23
24      % Calculate the value of the next player's move
25
26      % Get all the valid moves
27      valid_moves(GameState,ListOfMoves),
28
29      compute_next_player_value(GameState, ListOfMoves, NextPlayerValue),
30
31      % Modifier used to make the value positive or negative in case of a double
        ↪ play
32      get_modifier(PrevPlayer,NextPlayer,Modifier),
33
34      Value is SBValue1 + SBValue2 + NextPlayerValue * Modifier.
```

Código fonte 11: Calcular o valor de um estado de jogo

3.7 Jogada do Computador

Escolha da jogada a efetuar pelo computador, dependendo do nível de dificuldade. O predicado deve chamar-se *choose_move(+Board, +Level, -Move)*. O jogo implementado possui dois niveis de jogada de computador. O predicado *choose_move(+GameState, +Level, -Move)* recebe o estado de jogo e o tipo de jogada que se pretende obter ('P' para uma jogada introduzida por um jogador humano, 1 para o *bot* de nivel 1 e 2 para o *bot* de nivel 2.

O *bot* de nível 1 escolhe a sua jogada aleatoriamente de entre de uma das jogadas que lhe são possíveis (código fonte 12).

```
1   random_move(GameState, Move) :-
2       valid_moves(GameState, ListOfMoves),
3       random_member(Move, ListOfMoves).
```

Código fonte 12: Predicado que escolhe a jogada aleatória

O *bot* de nível 2 usa uma estratégia gananciosa e, assim sendo, procura jogar, de entre todas as jogadas que lhe são possíveis, a jogada que leve a um tabuleiro de maior valor. Para isso, utiliza o predicado *value/2* que é descrita na secção 3.6. O código fonte 13 descreve o predicado *greedy_move(+GameState, -BestMove)* que efetua a escolha.

```
1   greedy_move(GameState, BestMove) :-
2
3       % Get all the valid moves
```

```
4       valid_moves(GameState, ListOfMoves),
5
6       % Find all pairs of valid moves and their corresponding values
7       findall(Value-Move, (member(Move, ListOfMoves),move(Move, GameState,
        ↪  NewGameState), print(NewGameState), value(NewGameState, Value)), Result),
8
9       % Sort the pairs Value-Move in descending order according to Value
10      keysort(Result,SortedResultAsc),
11      reverse(SortedResultAsc,SortedResultDsc),
12      print(SortedResultDsc),
13
14      % Get the moves that lead to the best value and randomly choose one of them
15      group_pairs_by_key(SortedResultDsc, [_-BestMoves|_]),
16      length(BestMoves,Length),
17      random_between(1,Length,Rnd),
18      nth1(Rnd,BestMoves,BestMove).
```

Código fonte 13: Predicado que escolhe a jogada gananciosa

4 Conclusões

A linguagem Prolog embora se apresente um desafio devido à mudança brusca de paradigma em relação às linguagens mais populares (como o Java, C/C++, Python, JavaScript, entre outros) é muito poderosa pois a sua característica declarativa leva à concepção de soluções para um dado problema num espaço de tempo reduzido, apesar de, muitas vezes, à custa da eficiência da solução.

Consideramos que a implementação feita do *Squex* é completa e que serve como uma boa demonstração das capacidades do Prolog e do uso da linguagem. Existe lugar para melhorias, nomeadamente na eficiência de alguns predicados utilizados e também nas heurísticas usadas para avaliar o tabuleiro e, por consequência, na *performance* do *bot* de nível 2.