LYNGK

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo LYNGK_1:
Daniel Pereira da Silva - up201503212
Miguel António Ramalho - up201403027

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

23 de Dezembro de 2017

Resumo

O presente relatório descreve a abordagem utilizada na replicação do jogo de tabuleiro Lyngk numa linguagem de programação em lógica, PROLOG no caso em apreço.

Foram implementados os predicados necessários para a criação do interface com o utilizador, para a especificação das regras de jogo e para a gestão do estado de jogo.

Adicionalmente, foi-nos proposto a implementação de alguns algoritmos de geração de jogadas possíveis, que se tornou uma tarefa satisfatória quando os predicados relevantes já haviam sido implementados de acordo com as boas práticas da linguagem, tornando o processo mais simples e elegante.

Por fim, considera-se que este projeto serviu como rampa de lançamento das nossas capacidades de abordar problemas em linguagens de programação em lógica e que nos despertou o interesse de continuar a encontrar aplicações e formas de usar esta nova ferramenta na resolução de problemas do nosso dia-a-dia.

Conteúdo

1 Introdução

Na Unidade Curricular de Programação em Lógica, foi-nos proposta a implementação, em Prolog, de um jogo de tabuleiro. O grupo escolheu o jogo Lyngk, por ser uma das opções disponíveis aquando da nossa escolha. Não obstante, o jogo mostrou ser um grande desafio, levando-nos a conhecer melhor os conceitos subjacentes à programação em lógica.

Este relatório procura descrever, numa primeira parte, a história do jogo, as suas regras e propriedades, e numa segunda parte, a forma como foi por nós implementado em Prolog, desde a visualização do tabuleiro à especificação das regras e às jogadas por computador.

2 O Jogo Lyngk

2.1 História

Kris Burm (Antuérpia, 1957) é um inventor de jogos Belga, especializado em jogos de tabuleiro abstratos. É conhecido pela série de Jogos GIPF, um dos quais é o Lyngk. O Lyngk é, aliás, o último dos jogos desta série (até agora), e é tido como um agregar das ideias expostas nos outros jogos. Foi lançado em 2017, mas rapidamente se tornou famoso, dada a reputação do seu autor e da série de jogos que integra.

2.2 Regras

2.2.1 Preparação

O jogo deve ser jogado por duas pessoas. No princípio, o tabuleiro deve ser preenchido com 8 peças de cada cor (azul, marfim, vermelho, verde e preto) mais 3 peças *wild*, dispostas aleatoriamente, como mostra a Figura 1.

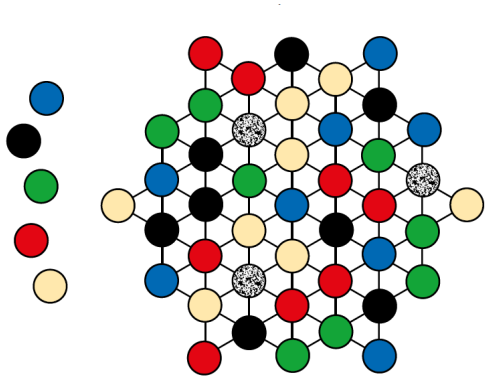

Figura 1: Exemplo de um tabuleiro no estado inicial

5 peças (uma de cada cor) devem ser deixadas de parte, junto ao tabuleiro. Seguidamente, deve ser sorteado o primeiro jogador.

2.2.2 Peças, Reivindicação de Peças & Cores

- As três peças *wild* devem ser consideradas *jokers*, isto é, elas têm a capacidade de representar qualquer peça do tabuleiro. Estas peças são passivas. Não podem ser utilizadas para jogar, e só podem ser movidas como parte de uma pilha.

- No início do jogo, todas as peças no tabuleiro são neutras, e não pertencem a qualquer dos jogadores. Isto significa que ambos os jogadores as podem usar.
- Durante o decorrer do jogo, cada jogador pode reclamar até duas peças. Assim que uma peça é reclamada por um jogador, deixa de ser neutra. Desse momento em diante, só o jogador que reclamou a peça pode jogar com peças dessa cor.

Nota: as peças ao lado do tabuleiro servem para ser reclamadas, e não podem entrar no tabuleiro.

- Os jogadores podem reclamar uma cor em qualquer altura do jogo, mas só durante o seu turno, e antes de fazerem a sua jogada. Para reclamar uma cor, o jogador deve retirar uma peça do lado do tabuleiro para junto de si.
- Um jogador só pode reclamar uma cor de cada vez, isto é, não pode reclamar duas cores no mesmo turno.
- Quando ambos os jogadores tiverem reclamado duas cores, a última cor permanece neutra. Ambos os jogadores podem continuar a jogar com peças dessa cor até ao final do jogo.

2.2.3 Movimentos

1. Os jogadores tomam turnos. A cada jogada, o jogador deve mover uma peça ou uma pilha de peças. Para fazer um movimento, podem escolher qualquer peça ou pilha de cor neutra, ou de uma das cores que tenham reclamado.
2. As pilhas de peças devem ser sempre movidas atomicamente. A peça no topo da pilha determina a cor da pilha.
3. Um movimento, seja com uma cor neutra, ou com uma cor reclamada, deve terminar sempre num espaço ocupado, ou seja, em cima de outra peça ou pilha. Um movimento pode terminar em cima de uma peça/pilha adjacente ou em cima de uma peça/pilha que possa ser alcançada jogando em linha reta, cruzando só espaços vazios, isto é, não é permitido saltar por cima de outras peças/pilhas.
4. Há uma outra forma de fazer movimentos, descrita mais abaixo, na secção Regra Lyngk.
5. Uma pilha pode ter até 5 peças. A regra-chave é que uma pilha tem de ter todas as peças de cores diferentes: nunca pode haver duas ou mais peças da mesma cor numa pilha. No entanto, podem haver até 3 peças brancas numa pilha, dado que estas tomam as cores de peças que não estejam já na pilha.
6. Uma única peça neutra só pode ser movida para cima de uma outra peça de qualquer outra cor. Por outras palavras, pode ser movida para cima de uma peça branca, para cima de outra peça neutra, ou para cima de uma peça que tenha sido reclamada por qualquer um dos jogadores. Uma única peça neutra não pode saltar para cima de uma pilha.
7. Uma pilha com uma peça neutra no topo pode saltar patra cima de qualquer peça, ou pilha cuja altura seja igual ou inferior à sua.
8. Uma única peça de uma cor reclamada, ou uma pilha de uma cor reclamada podem ser movidos para cima de qualquer outra peça/pilha, desde que a pilha resultante não fique com um número de peças superior a 5, e desde que todas as cores nessa pilha sejam diferentes.
9. Quando um jogador completa uma pilha de 5 peças, e a peça no topo é de uma cor que esse jogador tenha reclamado, esse jogador deve retirar essa pilha do tabuleiro e colocá-la junto a si, de modo sempre visível ao oponente. Cada pilha removida vale 1 ponto no final do jogo.

10. Quando um jogador completa uma pilha de 5 peças, com uma cor neutra em cima da pilha, a pilha passa a comportar-se como um obstáculo, e não sai de cima do tabuleiro. Esta pilha não vale como ponto para nenhum dos jogadores.

11. Um jogador não pode passar a jogada, a menos que não tenha movimentos possíveis.

12. Se um jogador estiver impossibilitado de jogar, o outro jogador deve continuar a jogar até que este, também, fique sem opções de jogo. Caso um jogador, que anteriormente tenha passado, tenha a possibilidade de jogar, então deve fazê-lo.

2.2.4 A Regra Lyngk

Na secção abaixo, tanto peças como pilhas são designadas como "peças", dado qua a regra Lyngk é válida para ambas. A Figura 2 mostra um diagrama da aplicação da regra Lyngk, durante a jogada do Jogador B.

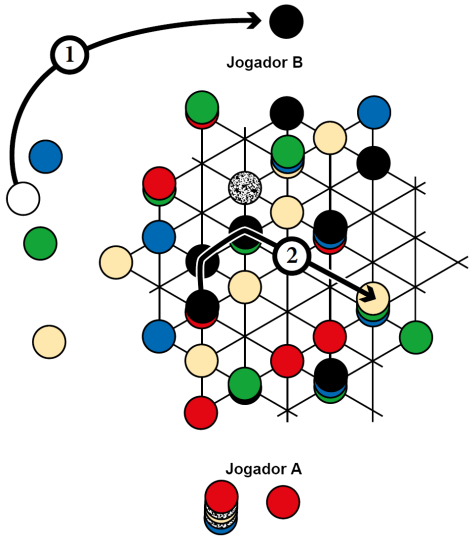

Figura 2: Exemplo da Regra Lyngk

O Jogador A já tem uma pilha, estando em vantagem, contudo o jogador B tem a possibilidade de igualar o seu adversário através da regra Lyngk. Primeiro, deve reclamar a cor preta (1), e depois faz um movimento triplo (2) e completa uma pilha de 5 peças. Para fazer este moviemnto múltiplo, todas os pontos-Lyngk têm de ser da mesma cor, e esta deve ser uma das cores reclamadas.

1. A regra Lyngk só pode ser utilizada quando o jogador estiver a mover uma peça reclamada.

2. A regra diz que as peças de uma dada cor estão ligadas (Lyngk), mas só se estiverem posicionadas de tal modo que seja possível juntá-las com um movimento normal. Um jogador pode usar peças dessa cor reclamada para fazer um movimento duplo, ou até um movimento triplo ou quádruplo, usando-as como ligações para outras peças no tabuleiro.

3. A regra aplica-se do seguinte modo: um jogador pode mover uma peça de uma cor reclamada até uma outra peça da mesma cor, onde não a empilha, antes a usa como um ponto-Lyngk, o que significa que deve fazer um segundo movimento a partir daí. Assim, da peça alcançada, o jogador deve mover (a peça original) para uma peça adjacente, ou para uma peça acessível em linha reta. A peça jogada deve ser empilhada na peça de destino. No entanto, se esta

segunda peça for da mesma cor da peça jogada oiginalmente, o jogador deverá fazer um terceiro movimento a partir daí, e assim por diante, até que a peça chegue a um sítio onde possa aterrar. (Ver Figura 2)

3 Lógica do jogo

3.1 Representação do Estado do Jogo

Dada a natureza não tabular do tabuleiro, optou-se por mapear o mesmo para uma lista de listas, que se pode abstrair para uma matriz, que contem algumas células cuja existência serve apenas para facilitar a gestão e alteração do estado do jogo, não sendo utilizadas para guardar informação relevante para o jogo em si. Cada uma das células desta matriz será também uma lista, devendo-se isto ao facto de, em cada posição do tabuleiro, poder haver uma pilha de peças.

De forma a aumentar a legibilidade dos tabuleiros, criou-se um predicado que especifica as equivalências no mapeamento de átomos para a consola `translate(+Name, -Representation)` em que `Name` é o átomo usado para descrever uma dada cor ou entidade do jogo e `Representation` é a string para a qual essa entidade vai ser traduzida aquando da escrita para a consola. O Excerto de Código Prolog 1 apresenta as traduções implementadas.

```
translate(black, 'D'). % preto
translate(red,   'R'). % vermelho
translate(ivory, 'I'). % marfim
translate(green, 'G'). % verde
translate(blue,  'B'). % azul
translate(wild,  'W').
```

Excerto de Código Prolog 1: Traduções utlizadas para identificar as cores através de átomos

Assim sendo, a representação escolhida para o tabuleiro será uma lista de listas de listas, cujo estado vazio se apresenta no Excerto de Código Prolog 2.

```
/*The board is a matrix of 13 lines and 9 columns, not all are used */
emptyBoard([
    [[], [], [], [], [], [], [], [], []],
    [[], [], [], [], [], [], [], [], []],
    [[], [], [], [], [], [], [], [], []],
    [[], [], [], [], [], [], [], [], []],
    [[], [], [], [], [], [], [], [], []],
    [[], [], [], [], [], [], [], [], []],
    [[], [], [], [], [], [], [], [], []],
    [[], [], [], [], [], [], [], [], []],
    [[], [], [], [], [], [], [], [], []],
    [[], [], [], [], [], [], [], [], []],
    [[], [], [], [], [], [], [], [], []],
    [[], [], [], [], [], [], [], [], []],
    [[], [], [], [], [], [], [], [], []]
]).
```

Excerto de Código Prolog 2: Mapeamento do tabuleiro numa lista de listas de listas

3.1.1 Estado Inicial

Sendo que, como referido na secção Preparação, o estado incial do tabuleiro é aleatório, a disposição incial das peças no tabuleiro não é única, a constante é, apenas, o número de peças de cada cor que está inicialmente no tabuleiro - 8 de cada cor, exceto da cor *wild*, que são apenas 3.

Apresenta-se no Excerto de Código Prolog 3 a representação correspondente ao tabuleiro da Figura 1, um exemplo do tabuleiro no seu estado inicial.

```
/* 8 pieces of each main color (black, red, ivory, green and blue) and
        3 of the wild color (W) */
ExampleBoard([
    [[], [], [red], [], [black], [], [blue], [], []],
    [[], [], [], [red], [], [ivory], [], [], []],
    [[], [], [green], [], [ivory], [], [black], [], []],
    [[], [green], [], [wild], [], [blue], [], [blue], []],
    [[], [], [black], [], [ivory], [], [green], [], []],
    [[], [blue], [], [green], [], [red], [], [wild], []],
    [[ivory], [], [black], [], [blue], [], [red], [], [ivory]],
    [[], [black], [], [ivory], [], [black], [], [green], []],
    [[], [], [red], [], [ivory], [], [blue], [], []],
    [[], [blue], [], [wild], [], [red], [], [green], []],
    [[], [], [ivory], [], [red], [], [black], [], []],
    [[], [], [], [black], [], [green], [], [], []],
    [[], [], [red], [], [green], [], [blue], [], []]
]).
```

Excerto de Código Prolog 3: Exemplo de um tabuleiro no estado inicial

3.1.2 Posição Intermédia

Apresenta-se no Excerto de Código Prolog 4 a representação correspondente ao tabuleiro da Figura 2.

```
ExampleBoard([
  [[], [], [green, red], [], [black], [], [blue], [], []],
  [[], [], [], [], [], [ivory], [], [], []], [], []],
  [[], [red, green], [], [white], [], [], [], [], []],
  [[], [], [], [], [ivory], [], [], [], []],
  [[], [blue], [], [black, green], [], [black, blue, red], [], [], []],
  [[ivory], [], [black], [], [], [], [], [], []],
  [[], [], [], [ivory], [], [], [], [], []],
  [[], [], [black, red], [], [], [],  [ivory, green, blue], [], []],
  [[], [blue], [], [], [], [red], [], [green], []],
  [[], [], [ivory], [], [red], [], [], [], []],
  [[], [], [], [green, black], [], [black, blue, green], [],  [], []],
  [[], [], [red], [], [], [], [], [], []]
]).
```

Excerto de Código Prolog 4: Exemplo de um tabuleiro no estado intermédio

3.1.3 Posição Final

Apresenta-se no Excerto de Código Prolog 5 um exemplo de um tabuleiro em que já não há movimentos possíveis para nenhum dos jogadores, de forma que o jogo está terminado.

```
ExampleBoard([
  [[], [], [], [], [], [], [], [], []],
  [[], [], [], [], [], [black, ivory, blue, green], [], [], []],
  [[], [], [], [], [], [], [blue, black, red, green], [], []],
  [[], [], [], [], [], [], [], [], []],
  [[], [], [], [], [green, red, blue, ivory], [], [], [], []],
  [[], [], [], [], [], [], [], [], []],
  [[], [], [], [], [], [], [], [], []],
  [[], [], [], [], [], [], [], [], []],
  [[], [], [], [], [ivory, green, blue], [], [], [], []],
  [[], [], [], [], [], [], [], [blue], []],
  [[], [], [], [], [], [], [], [], []],
  [[], [], [], [green, blue], [], [], [], [], []],
  [[], [], [], [], [], [], [], [], []]
]).
```

Excerto de Código Prolog 5: Exemplo de um tabuleiro no estado final

3.2 Visualização do Tabuleiro

Apresenta-se na Figura 3 um exemplo do tabuleiro durante um jogo. Nesta figura, as linhas impressas representam os caminhos navegáveis, sendo o resultado de tentar tornar a visualização inicialmente proposta em algo mais intuitivo.

Os predicados desenvolvidos para a visualização do tabuleiro na consola encontram-se no ficheiro `displayBoard.pl`. Está transcrito, no Excerto de Código Prolog 6, o predicado `displayBoard`, principal responsável pela impressão.

Este predicado invoca ainda o predicado `displayLine(+Board, +LineIndex)` que trata de percorrer cada uma das linhas do tabuleiro que recebe e de, para cada uma, invocar o predicado `stateReceiveValidCell(+Stack, +LineIndex, +ColumnIndex)` que se encarrega de traduzir as cores presentes no stack, como por exemplo `[red, green, blue]` para as respetivas letras, no caso: `RGB`.

De realçar que a visualização do tabuleiro é um pouco mais complexa do que esta recursividade direta, dado que há que alternar linhas de barras e linhas de stacks, mais ainda de alinhar e não colocar barras nas células que não são válidas. Isto é alcançado através de verificações do predicado `isValid(+X, +Y)` que descreve todas as posições válidas do tabuleiro.

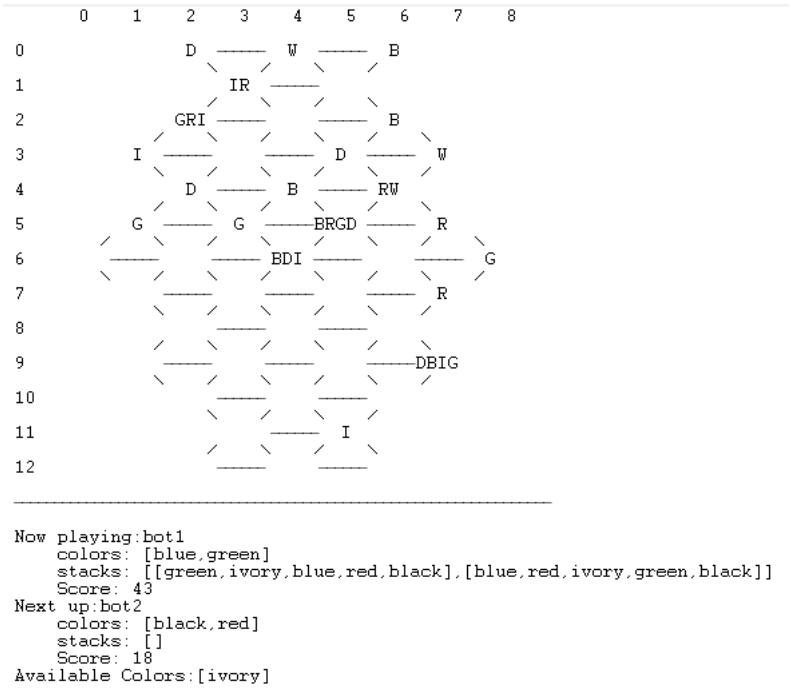

Figura 3: Exemplo do estado da consola após um displayBoard

```
%clears the screen and ouputs the board and the game state
displayBoard:-
  clear,
  write('      0    1    2    3    4    5    6    7    8\n\n'),
  board(Board),
  displayLine(Board, 0),
  wd(66),
  nl, !,

  player(CurrentPlayer),
  write('\nNow playing:'),
  displayPlayerStats(CurrentPlayer),

  nextPlayer(NextPlayer),
  write('\nNext up:'),
  displayPlayerStats(NextPlayer),

  toClaim(C),
  write('\nAvailable Colors:'), write(C), nl, nl, nl.
```

Excerto de Código Prolog 6: Predicados necessários para a apresentação do estado do jogo na consola

3.3 A pilha de jogo

De forma a tornar o desenvolvimento de algoritmos de pesquisa no código mais intuitiva, implementou-se um conjunto de predicados que tratam de gerir o estado atual do jogo - caracterizado pelo tabuleiro, pelas cores por reivindicar, pelas cores reivindicadas por cada jogador, pelas pilhas de 5 elementos de cada jogador, entre outras flags - de forma que permite simular múltiplas alterações ao jogo em sequência, sem perder o estado real do jogo.

Para tal desenvolveram-se os predicados `pushGame` e `popGame` que permitem a inserção, ainda que virtual, do estado atual de jogo numa pilha e a complementar ação de remoção dessa pilha.

Qualquer alteração exploratória ou destrutiva feita dentro da vigência destes dois predicados acontece sobre uma cópia do estado do jogo. Esta lógica permite ainda encadear inserções e remoções na stack de forma a simular o estado do jogo após n jogadas. Na secção 3.8, esta abordagem é muito proveitosa para a exploração da árvore do jogo.

Apesar da característica de backtracking do PROLOG, considerou-se que não seria suficiente para passar o estado de jogo entre predicados de forma limpa e intuitiva, dada a grande quantidade de informação que é necessária para caracterizar o jogo num dado momento.

3.4 Lista de Jogadas Válidas

A obtenção da lista de jogadas válidas é efetuada através de um predicado que lista as regras necessárias para uma dada jogada ser válida, entre a pilha `Xf,Yf` e a pilha `Xt,Yt`; este é o predicado `getFullValidMove(+MoveableColors, +Xf, +Yf, +Xt, +Yt, +ClaimedColor)`, transcrito no Excerto de Código Prolog 7.

O uso do predicado `once` justifica-se com uma melhoria na eficiência do programa quando este é usado com algum predicado de listagem exaustiva, como sendo `setof` ou `findall`.

Este predicado revelou-se uma forma bastante elegante de lidar com a questão de encontrar a próxima jogada válida, tendo em conta que não passa apenas por encontrar ligação entre dois pontos. Passa por o jogador, antes de fazer um movimento, poder ou não reivindicar uma cor (`ClaimedColor`), o que expande o número de jogadas possíveis e também dificulta a forma de as enumerar.

```
1  %get all the stacks a given stack can be moved to
2  getFullValidMove(MoveableColors, Xf, Yf, Xt, Yt, none):-%no color claimed
3    isValid(Xf, Yf), %valid coordinates for the move origin (from)
4    once(checkStackNotEmpty(Xf, Yf)), %make sure the from stack is not empty
5    isValid(Xt, Yt), %valid coordinates for the move destination (to)
6    once(checkStackNotEmpty(Xt, Yt)), %make sure the to stack is not empty
7    isColorInStackPlayable(Xf, Yf, MoveableColors), %the player can move this stack
8    once(hasNoDuplicateColors(Xf, Yf, Xt, Yt)), %no duplicate color in the stacks
9    once(canPileStacks(Xf, Yf, Xt, Yt)), %stacks can be piled (height restrictions)
10   once(checkValidMove(Xf, Yf, Xt, Yt)). %there is a valid path between the stacks
11
12 %get all the stacks a given stack can be moved to
13 getFullValidMove(MoveableColors, Xf, Yf, Xt, Yt, ClaimedColor):-%with claim
14   once(validClaim), %the player has claimed less than 2 colors
15   isClaimableColor(ClaimedColor), %is this a valid color to claim
16   %the claimed color can also be moved
17   once(append([MoveableColors, [ClaimedColor]], NewMoveableColors)),
18   %call the predicate with "none" as the claimed color
19   getFullValidMove(NewMoveableColors, Xf, Yf, Xt, Yt, none).
```

Excerto de Código Prolog 7: Predicado para obter uma jogada válida

Posto este predicado, obter a lista das jogadas válidas torna-se tão simples como implementar o predicado `findall` para pesquisar sobre o universo das jogadas válidas. De facto, isto foi feito e isolou-se este raciocínio num novo predicado que devolve a lista de todas as jogadas válidas, para um dado jogador, num dado momento - `getPossibilities(-Possibilities)` - cuja implementação se encontra no Excerto de Código Prolog 7.

```
1 getPossibilities(Possibilities):-
2     getMoveableColorsByPlayer(MoveableColors),
3     findall(Xf-Yf-Xt-Yt-Color,
4             getFullValidMove(MoveableColors, Xf, Yf, Xt, Yt, Color),
5             Possibilities).
```

Excerto de Código Prolog 8: Predicados para obter a lista de movimentos válidos

3.5 Execução de Jogadas

Durante cada jogada, existem duas ações possíveis: reivindicar uma cor (`claim`) e movimentar uma pilha de uma posição para outra (`move`).

3.5.1 Mover

Este é o movimento mais comum: pegar numa peça ou numa pilha e colocar em cima de outra peça ou pilha. O cabeçalho corresponderá a `move(+Xf, +Yf, +Xt, +Yt)`. Neste cabeçalho, `Xf` e `Yf` identificam a linha e coluna da peça a mover, respetivamente; `Xt` e `Yt` identificam a posição de destino, da mesma forma.

Ao contrário do que tínhamos previamente planeado, a jogada `lyngk` acabou por ser absorvida pela jogada `move`. Apesar de consistir numa sequência de saltos, esta jogada pode ser feita indicando meramente a posição inicial e a posição final, encarregando-se o motor de jogo de aferir a validade das jogadas.

Para além de nem todas as posições da nossa matriz serem válidas, no contexto do jogo, há ainda movimentos que não são válidos por não respeitarem as regras. É por isso necessário verificar uma série de condições. No caso do movimento ser do jogador criou-se um novo predicado `processMove(+Xf, +Yf, +Xt, +Yt)` que trata de fazer as mesmas verificações do predicado `getFullValidMove` (cuja implementação se encontra no Excerto de Código Prolog 7) mas com predicados que escrevem na consola quais as regras inválidas, caso as haja, para que o utilizador saiba porque razão um dado movimento que tentou fazer é inválido.

Os predicados mais relevantes para esta interação com o utilizador e alteração do tabuleiro encontram-se no Excerto de Código Prolog 9

```
% move a stack from a cell in the board into another
move(Xf, Yf, Xt, Yt):-
    board(B),
    getBoardStack(Xf, Yf, StackTop),
    getBoardStack(Xt, Yt, StackBottom),
    append([StackTop, StackBottom], Stack), % RULE E-2
    replaceBoardStack(B, Xt, Yt, Stack, B2),
    replaceBoardStack(B2, Xf, Yf, [], NewBoard),
    saveBoard(NewBoard).
%if the move is valid executes it, prints an error otherwise
processMove(Xf, Yf, Xt, Yt):-
    checkValidCell(Xf, Yf),
    checkValidCell(Xt, Yt),
    checkStackNotEmpty(Xt, Yt), !, % RULE E-3
    checkBelongsToPlayer(Xf, Yf), !, % RULE E-1, RULE E-2
    checkStacksPile(Xf, Yf, Xt, Yt), !, % RULE E-5, RULE E-8
    checkNeutralStackJumpTo(Xf, Yf, Xt, Yt), !, % RULE E-6, RULE E-7
    checkDuplicateColors(Xf, Yf, Xt, Yt), !,  % RULE E-5
    checkValidMove(Xf, Yf, Xt, Yt),
    move(Xf, Yf, Xt, Yt).
```

Excerto de Código Prolog 9: Predicado para verificar se um movimento pode ser feito.

3.5.2 Reivindicar uma cor

Ação que cada jogador pode fazer até um máximo de duas vezes no total, uma durante a sua jogada e necessariamente antes de mover alguma peça. O cabeçalho corresponde a `claim(+Color)`. Neste predicado, `Color` identifica a cor que o jogador quer reivindicar. O turno durante o qual este predicado é chamado determina o jogador que reclama a cor passada.

```
%changes the game state by claiming a color
claim(Color):-
    getColors(ChosenColors), %get the current player player's colors
    append([ChosenColors, [Color]], Result),
    saveGetColors(Result),
    saveHasClaimed(true),%set the flag hasClaimed to true
    toClaim(ToClaim),
    nth0(_, ToClaim, Color, NewToClaim),%if the chosen color is inside ToClaim
    saveToClaim(NewToClaim). %updated available colors
```

Excerto de Código Prolog 10: Predicado para reivindicar uma dada cor e alterar o estado do jogo

Como há restrições à forma como se pode reclamar uma cor, é necessário verificar que todas as regras são cumpridas. Para tal é usada um predicado `hasClaimed` como uma *flag* booleana que indica se, na jogada atual, o jogador já reivindicou alguma cor. Adicionalmente, fazem-se verificações como a do Excerto de Código Prolog 11 onde se garante que o jogador a tentar reclamar uma cor ainda não o fez a duas.

```
%fails if the user cannot claim
validClaim:-
    getColors(ChosenColors), %get the current player player's colors
    length(ChosenColors, Len), %get the length of this player's colors
    Len < 2. %if the length is less than 2, then read color
```

Excerto de Código Prolog 11: Predicado para verificar se jogador pode reclamar mais uma cor.

3.6 Avaliação do Tabuleiro

Por forma a implementar as heurísticas responsáveis pela determinação da jogada feita pelo computador, há necessidade de determinar um valor de comparação do estado do tabuleiro. Este valor é calculado à custa das pilhas obtíveis e do número de cores por reclamar.

A heurística de avaliação de tabuleiro está especificada no Excerto de Código Prolog 12, através do predicado `evaluateBoard(+Player, -Score)`. Foram testadas várias valorizações para os diferentes pontos da heurística, sendo que a decisão final se caracteriza por:

1. 10 pontos por cor não reivindicada;
2. 20 pontos por pilha de tamanho 5 que pertence ao jogador;
3. número de pontos igual ao tamanho de cada pilha pertencente ao jogador, excetuando as pilhas de tamanho 5.

```
evaluateBoard(Player, Score):-
    getColors(Player, Colors),
    getPlayerStackScore(Player, Colors, 5, Score).
%reached the end, return the score of the stacks
getPlayerStackScore(Player, Colors, Height, Score):-
    Height =< 0,
    getStacks(Player, Stacks),
    length(Stacks, TempScore), %count the height of the stacks
    length(Colors, LenColors), %count the number of claimed colors
    %each stack is worth 20 points, each unclaimed color 10
    Score is ((TempScore * 20) + ((2- LenColors) * 10)).

%if there are still heights to evaluate
getPlayerStackScore(Player, Colors, Height, StackScore):-
    countStacksByColorAndHeight(Colors, Height, Count),
    NewHeight is Height - 1,
    getPlayerStackScore(Player, Colors, NewHeight, TempScore),
    %each extra stack is worth it's height (1 to 4)
    StackScore is ((Count * Height) + TempScore).
```

Excerto de Código Prolog 12: Predicado para avaliar o tabuleiro

3.7 Final do Jogo

O final de jogo só ocorre quando já nenhum dos jogadores pode efetuar qualquer movimento válido. Para fazer esta verificação voltou-se a usar o predicado `getFullValidMove`, mas desta vez

apenas numa instância, dado que para o jogo não acabar basta haver um único movimento por fazer. De acordo com as regras do Lyngk, se o jogador atual não tiver jogadas por fazer, o próximo jogador joga, só quando nenhum tem jogadas possíveis é que o jogo se dá por terminado. Esta lógica de jogo é garantida com os predicados implementados no Excerto de Código Prolog 13

```
%fails if current player has no move left
currentPlayerHasMoves:-
    getMoveableColorsByPlayer(MoveableColors),
    getFullValidMove(MoveableColors, _Xf, _Yf, _Xt, _Yt, _Color).

%check if the current player has moves
%if not check if the next player has moves
%if not endGame
assertBoard:-currentPlayerHasMoves.
assertBoard:-%current player has no moves, invert players and test
    invertPlayers,
    currentPlayerHasMoves.
assertBoard:-%if no player has moves, end the game
    displayBoard,
    write('no more valid moves'), nl,
    getWinner(Winner), %calculate the winner of the game (draw) if there is no winner
    displayWinner(Winner),
    exit.
```

Excerto de Código Prolog 13: Predicado para determinar o fim de jogo e o vencedor.

3.8 Jogada do Computador

A escolha do movimento a ser efetuado pelo computador é feita com base no critério selecionado no início do jogo. Há vários níveis disponíveis: aleatório, ganancioso e Alpha-beta.

No Excerto de Código Prolog 14 econtra-se o predicado `playBotByLevel(+BotType, -Move)` que devolve a jogada do computador para cada um dos algoritmos de escolha. A descrição da jogada segue o formato `Xf-Yf-Xt-Yt-Color` que descreve qual a cor (`Color`) a reivindicar (`none` se nenhuma cor for reivindicada) e as coordenadas das pilhas inicial `Xf-Yf` e final `Xt-Yt`.

```
playBotByLevel(random, Move):-%random move
    getMoveableColorsByPlayer(MoveableColors),
    findall(Xf-Yf-Xt-Yt-Color,
        getFullValidMove(MoveableColors, Xf, Yf, Xt, Yt, Color),
        Possibilities),
    random_select(Move, Possibilities, _).

playBotByLevel(greedy, Move):-%greedy move
    getMoveableColorsByPlayer(MoveableColors),
    setof(Score:Xf-Yf-Xt-Yt-Color,
        (getFullValidMove(MoveableColors, Xf, Yf, Xt, Yt, Color),
                evaluateMove(Xf-Yf-Xt-Yt-Color, Score)),
        Possibilities),
    last(Possibilities, Score:Move).

playBotByLevel(Number, Move):-%alpha-beta
    integer(Number),
    startAlphaBeta(Number, Value:Move),
    write('alphabeta: '), write(Value), nl.
```

Excerto de Código Prolog 14: Predicados de jogo do computador

3.8.1 *random*

O computador joga de forma aleatória, executando uma pesquisa das jogadas possíveis e escolhendo uma aleatoriamente, através do predicado `random_select`.

3.8.2 *greedy*

O computador escolhe sempre o movimento que lhe é imediatamente mais vantajoso. O que passa por identificar as jogadas possíveis, atribuir-lhes uma pontuação com o predicado `evaluateMove` fazendo a listagem através do predicado `setof` que devolve uma lista ordenada por pontuação crescente da jogadas possíveis, da qual se escolhe a última, a melhor portanto.

3.8.3 Alpha-beta Pruning

O computador percorre a árvore de jogadas até uma profundidade máxima de n níveis e aplica um algoritmo derivado de Minimax para escolher a melhor jogada. Optou-se por fazer uma experiência sobre o trabalho inical desenvolvido. Esta experiência passou pela implementação de um algoritmo de pesquisa que não fosse meramente aleatório ou ganancioso. Dado que apresenta alguma melhoria de performance em relação ao algoritmo MiniMax, opotu-se por implementar o algoritmo Alpha-beta que passa por descartar, de antemão, os ramos da árvore que já se sabem não levar a uma jogada melhor do que as até aí obtidas, aquando da pesquisa em profundiade feita pela árvore de todas as jogadas possíveis.

Dada a dimensão do tabuleiro e a quantidade de jogadas imediatas possíveis, que atinje valores como 600 ou mais, era claro à partida que o algoritmo não teria uma prestação eficiente. Posto isto opotu-se por fazer uma modificação ao mesmo adicionando uma nova condição de paragem - atingido um nível de profundidade máximo na árvore de pesquisa não se continua a pesquisa em profundidade.

Com esta adição e, se o nível máximo escolhido para uma dada pesquisa for 1, podemos observar que os resultados desta pesquisa seriam idênticos a um algoritmo ganancioso.

Empiricamente, a pesquisa sobre uma árvore tão grande é inviável quando a distância até um tabuleiro terminal é superior a 2. Portanto, apesar de ter sido uma cruzada enriquecedora e interessante, a utilização desta pesquisa no comportamento de um *bot* (computador) não traz uma jogabilidade mais desafiante para o motor de jogo construído.

Isolou-se num ficheiro `alphaBeta.pl` os predicados necessários para este algoritmo.

4 Interface com o utilizador

Para arrancar o jogo deverá invocar o predicado `init`.

Inicialmente, o utilizador é saudado com um ecrã que permite escolher o modo de jogo (Humano contra humano, humano contra computador, computador contra computador) ou ver uma breve descrição de como interagir com a interface (Figura 4).

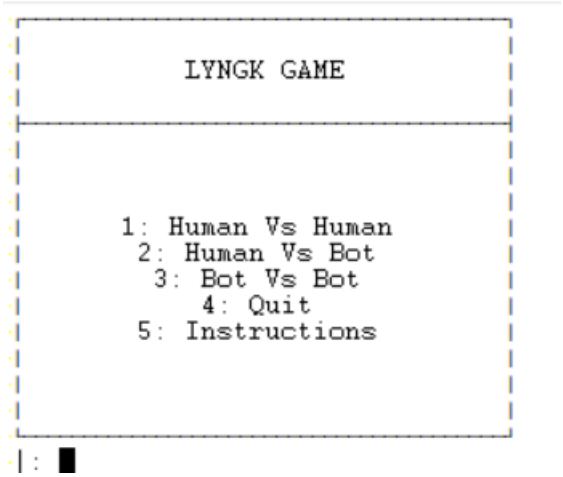

Figura 4: Menu inicial

Uma vez escolhido o modo de jogo, o jogo começa.

Nas jogadas to tipo *move*, devem introduzir-se as coordenadas do seguinte modo: `Xde-Yde:Xpara-Ypara`.

Nas jogadas do tipo claim, deve introduzir-se a cor a reclamar, é adicionalmente mostrada uma lista das cores disponíveis, inicialmente são as seguintes: `black`, `red`, `ivory`, `green`, `blue`.

A interface é, de resto, bastante intuitiva e funcional, tendo sido testada com o compilador `SICSTUS`.

5 Conclusões

Após a finalização do trabalho foi possível reparar que se havia criado um motor de jogo, numa linguagem de programação em lógica, com potencial para ser transposto e reutilizado com interfaces gráficas mais complexas e apelativas. Constatou-se que o resultado final foi uma implementação bastante robusta e escalável do jogo Lyngk.

Como melhorias sobre o trabalho desenvolvido, há apenas a referir a implementação de mais e melhores algoritmos de decisão de jogada para as jogadas geradas pelo computador.

6 URLs consultados

- **Lyngk Game Rules** - `https://www.boardgamegeek.com/filepage/142605/lyngk-rules`

- **BoardGameGeek Lyngk page** - https://www.boardgamegeek.com/boardgame/217083/lyngk
- **BoardGameGeek Review on Lyngk** - https://boardgamegeek.com/thread/1722033/looking-lyngk-review
- **Greedy Algorithm** - https://en.wikipedia.org/wiki/Greedy_algorithm
- **MiniMax** - https://en.wikipedia.org/wiki/Minimax
- **Alpha-beta Pruning** - https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning