Título do Trabalho

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo 79:
Flávio Couto - 201303726
Pedro Afonso Castro - 201304205

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

7 de Novembro de 2015

Resumo

Para a unidade curricular de Programação em Lógica, o nosso grupo comprometeu-se a desenvolver na linguagem PROLOG o jogo Syrtis, um jogo de estratégia cujo objetivo é criar uma ilha com todas as peças da nossa cor ou forma, impedindo que o adversário faça o mesmo com as suas.

O objetivo deste projeto foi colocarmos em prática os assuntos abordados nas aulas, bem como compreender um paradigma de programação no qual não tínhamos ainda qualquer experiência. Para tal, fizemos pesquisa, estudo e planeamento, de forma a podermos cumprir não só os objetivos propostos pelos professores, como também os nossos objetivos de aprofundar os nossos conhecimentos acerca de um novo paradigma de programação. Como resultado, ficámos com um jogo, executável em linha de comandos, bastante interessante e robusto, que nos orgulhamos de ter realizado.

Para concluir, entendemos que fomos capazes de cumprir com os objetivos propostos, tanto pelos professores, visto que conseguimos desenvolver todos os objetivos propostos, como por nós, porque conseguimos não só consolidar os conhecimentos aprendidos nas aulas, bem como aprofundar os nossos conhecimentos acerca do PROLOG.

Conteúdo

1 Introdução

O principal objetivo deste projeto é adquirir competências ao nível da Programação em Lógica através do estudo de um jogo de tabuleiro previamente escolhido. Foram-nos dadas várias hipóteses de jogos, tendo nós escolhido o Syrtis, por causa da sua elevada componente estratégica e tática, bem como alguma complexidade, sempre necessária para trazer um maior sentimento de desafio quando nos é colocado um projeto em mãos.

Este relatório contém várias secções. Uma secção a descrever as história e regras do jogo, com imagens para ajudar à sua compreensão, outra secção com a descrição da lógica utilizada para a implementação do jogo, nomeadamente a sua visualização, execução de movimentos, verificação do cumprimento das regras do jogo, teste da condição de vitória e a inteligência artificial do computador quando o utilizador escolhe jogar em modo *single player*, uma descrição da interface com o utilizador, em modo textual, uma conclusão acerca dos objetivos cumpridos e por cumprir, uma lista de fontes utilizadas para a produção deste relatório e do código-fonte, e por último uma série de elementos anexados, nomeadamente o código fonte do projeto.

2 O Jogo Syrtis

Segue-se uma explicação das regras do Syrtis, bem como alguns exemplos para melhor demonstrar alguns aspetos que possam ser de maior dificuldade de compreensão.

2.1 História

O Syris é um jogo de estratégia em que os dois jogadores se encontram numa ilha instável e desconhecida. A paisagem da ilha está sempre em mudança, e o avanço do mar faz com que a ilha esteja a tornar-se cada vez mais pequena... Para complicar ainda mais, este avanço está a fazer com que se formem areias movediças, limitando ainda mais o espaço habitável na ilha... Apenas um dos jogadores poderá sobreviver, quem será capaz de ser o mais forte?

2.2 Objetivo

A cada jogador é atribuída uma forma e uma cor (quadrado e preto ou círculo e branco). O objetivo do jogo é conquistar todas as peças que contenham a sua cor ou a sua forma.

2.3 Equipamento

O jogo é composto por peças quadradas com uma determinada forma e cor. Há 4 combinações possíveis: círculos e quadrados brancos e pretos. Há também quatro torres, duas brancas e circulares e duas pretas e quadradas.

2.4 Preparação

As peças quadradas são inicialmente aleatoriamente dispostas num de dois formatos de ilha. Para um jogo mais longo e estratégico, utiliza-se o formato Syrtis Major. Para um jogo mais curto e tático, utiliza-se o formato Syrtis Minor. As figuras 1 e 2 mostram a estrutura destes dois formatos.

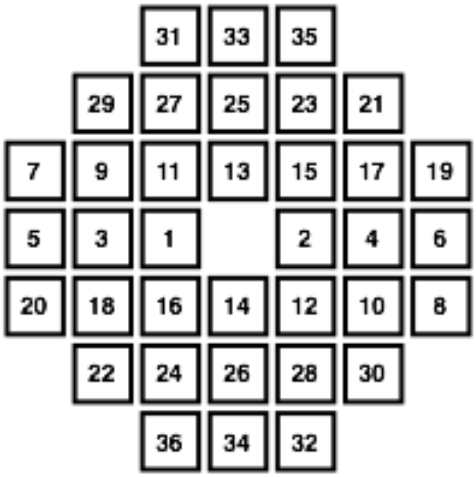

Figura 1: Syrtis Major

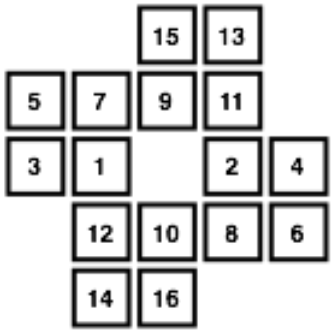

Figura 2: Syrtis Minor

Depois, um dos jogadores coloca as 4 torres por cima de uma das peças à sua escolha, desde que sejam da cor ou da forma dessa peça. O outro jogador decide se quer jogar com as torres brancas circulares ou pretas quadradas e o jogo começa por quem tiver ficado com as peças brancas e circulares.

2.5 Ilhas

Uma ilha é uma peça ou um grupo de peças que partilham a mesma forma ou cor. Para serem consideradas uma ilha, devem estar adjacentes horizontal ou verticalmente. Há quatro tipos de ilhas: ilhas pretas, brancas, quadradas e circulares. Uma peça pode obviamente pertencer a uma ilha de uma cor e a uma ilha de uma forma ao mesmo tempo. As figuras 3 e 4 mostram exemplos de ilhas.

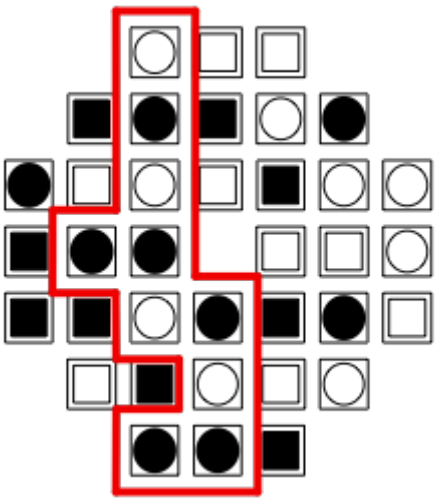

Figura 3: Ilha de círculos

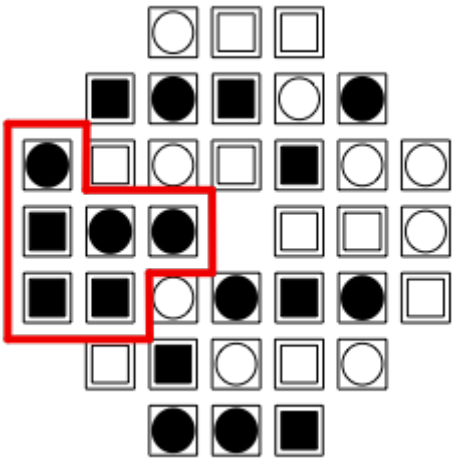

Figura 4: Ilha de pretos

2.6 Acções

Em cada turno cada jogador pode fazer uma de entre 4 acções possíveis: mover uma torre, afundar uma peça, deslocar uma peça, ou passar a sua vez.

2.6.1 Mover uma torre

Cada jogador pode mover uma das suas torres, desde que se mantenha em pelo menos uma das suas duas ilhas atuais (ou seja, ou na ilha respeitante à forma ou na ilha respeitante à cor). As outras torres não bloqueiam este movimento, ou seja, podemos passar por cima de outras torres. A figura 5 mostra um exemplo do movimento de uma torre.

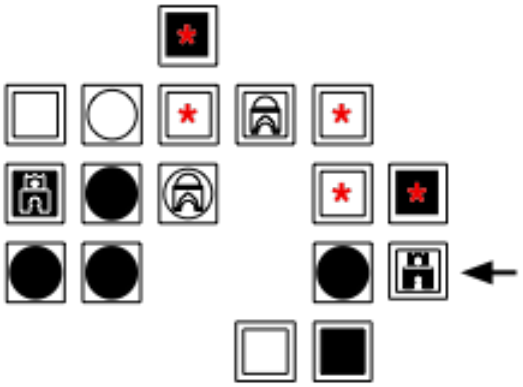

Figura 5: Movimento de uma torre

A torre indicada (preta e quadrada) pode mover-se para qualquer uma das peças marcadas com uma seta, pois estas encontram-se na sua ilha quadrangular. Note-se também que ela não pode ir para a peça à sua esquerda, visto que, apesar de ser preta, esta não se encontra na sua ilha (visto que a peça em que a torre se encontra é branca e quadrada, e a peça em questão é preta e circular).

2.6.2 Afundar uma peça

Cada jogador pode remover uma peça do tabuleiro se:

- Esta for adjacente a uma peça ocupada por uma torre desse jogador;
- Esta estiver desocupada;
- Esta tiver pelo menos um espaço adjacente livre.

A figura 6 mostra um exemplo de quais peças podem ser afundadas.
O jogador preto pode afundar quaisquer peças que estejam marcadas com uma estrela. Nenhuma peça branca está numa posição que lhe permita afundar alguma peça.

2.6.3 Deslocar uma peça

Um jogador poderá deslocar uma peça ocupada por uma das suas torres através de quaisquer espaços vazios que estejam conectados. A peça pode ser movida em qualquer direcção, quantas vezes quisermos, desde que a largura ou

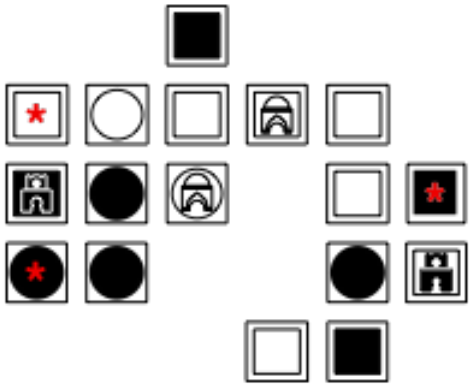

Figura 6: Peças afundáveis

altura do tabuleiro não seja aumentada (nem mesmo a meio do movimento) e as peças do tabuleiro continuem conectadas. A figura 7 mostra um exemplo disto.

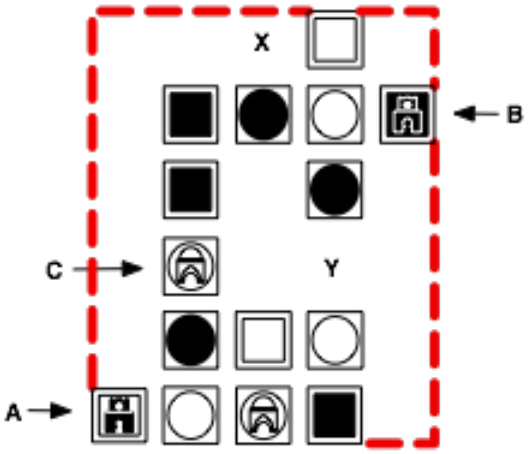

Figura 7: Peças deslocáveis

A torre A pode mover a sua peça para a posição X, mas a torre B não pode (visto que teria de aumentar o tamanho do tabuleiro a meio da jogada para o fazer). A torre C apenas se pode deslocar para a posição Y, porque é a unica posição em que o tabuleiro continua todo conectado. A outra torre não se pode mexer.

2.6.4 Passar a vez

Basta não fazer nada e passar a vez ao outro jogador. Um jogador que não pode fazer nada é obrigado a passar.

2.7 Fim do jogo

Há três formas de ganhar o jogo. Uma já foi dita anteriormente (ter uma ilha completa). As outras duas são usadas principalmente para evitar empates

e que os jogos durem demasiado tempo.

2.7.1 Ilha completa

Um jogador vence quando todas as peças restantes da sua cor, ou da sua forma, estão conectadas. Se todas as peças de uma cor ou forma estão conectadas quando o tabuleiro é inicialmente construído, o jogador que colocar as torres deve inicalmente trocar duas peças cuja numeração no formato de jogo usado (ver figuras 1 e 2) sejam seguidas. Por exemplo, trocar a peça 11 com a peça 12. Deve continuar a usar-se este método até que tal deixe de acontecer.

2.7.2 Areias movediças

Se um jogador afundar quatro peças sem o outro jogador ter afundado nenhuma, o segundo perde, sendo engolido pelas areias movediças. Os jogadores devem controlar quantas peças afundaram desde que o outro jogador afundou a última peça.

2.7.3 Iniciativa

Esta regra existe para evitar empates. O jogo termina se alguma das seguintes situações ocorrer:

- Ambos os jogadores conseguem uma ilha completa ao mesmo tempo;
- Ambos os jogadores passam a sua vez em jogadas consecutivas (ou seja, ambos passam duas vezes cada um);
- Um jogador passa quatro vezes seguidas.

Em qualquer um dos casos, o último jogador a afundar uma peça ganha. Se nenhum dos jogadores tiver afundado uma peça, o jogador que jogou primeiro ganha.

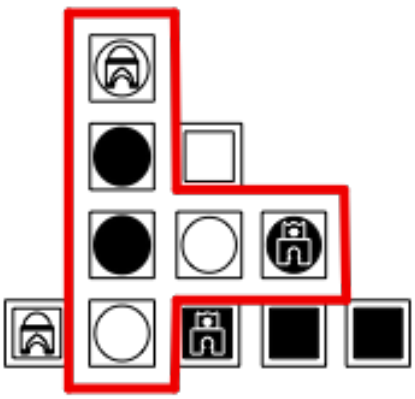

Figura 8: Uma situação em que o jogador branco ganhou ao completar uma ilha de círculos.

3 Lógica do Jogo

3.1 Representação do Estado do Jogo

O estado do jogo é guardado recorrendo à base de dados do PROLOG. Cada peça é guardada sob a forma de um predicado, utilizando o formato [Torre,Cor,Forma].

- **Torre:** Representa a existência ou não de uma torre no quadrado. Em caso afirmativo, utiliza a letra 'L' para representar uma torre branca, circular, e uma letra 'T' para uma torre preta, quadrangular.
- **Cor:** Representa a cor da do quadrado. Se for preto, utiliza-se a letra 'P', se for branco utiliza-se a letra 'B'.
- **Forma:** Representa a forma da peça do quadrado. Se for um círculo, usa-se a letra 'C', se for um quadrado usa-se a letra 'Q'.

Para todos os casos, a não existência do objeto implica a representação através de um espaço (' ').

Segue-se um exemplo de representação do board quando o jogo começa:

```
beggining_board(Board) :- Board =
[       [[' ',' ',' '], [' ',' ',' '],[' ','B','C'], [' ','B','Q'],
         [' ','B','Q'], [' ',' ',' '],[' ',' ',' ']],
        [[' ',' ',' '], [' ','P','Q'],[' ','P','C'], [' ','P','Q'],
         ['L','B','C'], [' ','P','C'], [' ',' ',' ']],
        [[' ','P','C'], [' ','B','Q'], ['L','B','C'], [' ','B','Q'],
         [' ','P','Q'], [' ','B','C'], [' ','B','C']],
        [[' ','P','Q'], [' ','P','C'], [' ','P','C'], [' ',' ',' '],
         ['T','B','Q'], [' ','B','Q'], [' ','B','C']],
        [[' ','P','Q'], [' ','P','Q'], [' ','B','C'], [' ','P','C'],
         [' ','P','Q'], [' ','P','C'], [' ','B','Q']],
        [[' ',' ',' '], ['T','B','Q'], [' ','P','Q'], [' ','B','C'],
         [' ','B','Q'], [' ','B','C'], [' ',' ',' ']],
        [[' ',' ',' '], [' ',' ',' '], [' ','P','C'], [' ','P','C'],
         [' ','P','Q'], [' ',' ',' '], [' ',' ',' ']]].
```

Quando o jogo está a meio:

```
middle_board(Board) :- Board =
[       [[' ',' ',' '], [' ',' ',' '], [' ',' ',' '], [' ',' ',' '],
         [' ','B','Q'], [' ',' ',' '], [' ',' ',' ']],
        [[' ',' ',' '], [' ',' ',' '], [' ','P','Q'], [' ','P','C'],
         [' ','B','C'], ['T','P','Q'], [' ',' ',' ']],
        [[' ',' ',' '], [' ',' ',' '], [' ','P','Q'], [' ',' ',' '],
         [' ','P','Q'], [' ',' ',' '], [' ',' ',' ']],
        [[' ',' ',' '], [' ',' ',' '], ['L','B','C'], [' ',' ',' '],
         [' ',' ',' '], [' ',' ',' '], [' ',' ',' ']],
        [[' ',' ',' '], [' ',' ',' '], [' ','P','C'], [' ','B','Q'],
         [' ',' ',' '], [' ',' ',' '], [' ',' ',' ']],
        [[' ',' ',' '], ['T','B','Q'], [' ','B','C'], ['L','B','C'],
         [' ','P','Q'], [' ',' ',' '], [' ',' ',' ']],
        [[' ',' ',' '], [' ',' ',' '], [' ',' ',' '], [' ',' ',' '],
         [' ',' ',' '], [' ',' ',' '], [' ',' ',' ']]].
```

E quando o jogo acaba:

```
end_board(Board) :- Board =
[       [[' ',' ',' ',' '], [' ',' ',' ',' '], [' ',' ',' ',' '], [' ',' ',' ',' '],
         [' ',' ',' ',' '], [' ',' ',' ',' '], [' ',' ',' ',' ']],
        [[' ',' ',' ',' '], [' ',' ',' ',' '], [' ',' ',' ',' '], [' ',' ',' ',' '],
         [' ',' ',' ',' '], [' ',' ',' ',' '], [' ',' ',' ',' ']],
        [[' ',' ',' ',' '], [' ',' ',' ',' '], ['L','B','C'], [' ',' ',' ',' '],
         [' ',' ',' ',' '], [' ',' ',' ',' '], [' ',' ',' ',' ']],
        [[' ',' ',' ',' '], [' ',' ',' ',' '], [' ','P','C'], [' ','Q','B'],
         [' ',' ',' ',' '], [' ',' ',' ',' '], [' ',' ',' ',' ']],
        [[' ',' ',' ',' '], [' ',' ',' ',' '], [' ','P','C'], [' ','B','C'],
         ['T','B','C'], [' ',' ',' ',' '], [' ',' ',' ',' ']],
        [['L','B','Q'], [' ','B','C'], ['T','P','Q'], [' ','P','Q'],
         [' ','P','Q'], [' ',' ',' ',' '], [' ',' ',' ',' ']],
        [[' ',' ',' ',' '], [' ',' ',' ',' '], [' ',' ',' ',' '], [' ',' ',' ',' '],
         [' ',' ',' ',' '], [' ',' ',' ',' '], [' ',' ',' ',' ']]].
```

Para além do tabuleiro, outros aspetos importantes para a representação do estado de jogo, como o jogador que está atualmente a jogar, ou quem tem a sequência de peças afundadas atual, são também guardadas na base de dados do PROLOG.

3.2 Visualização do Tabuleiro

O tabuleiro é gerado aleatoriamente, começando por se inicializar os predicados na base de dados recorrendo ao seguinte predicado:

```
create_database(+Length).
```

Em que Length é o tamanho do tabuleiro, que terá o tamanho 5 se for um tabuleiro do tipo Syrtis Minor ou 7 se for um do tipo Syrtis Major. Depois disso, as peças são colocadas de forma aleatória, sendo chamadas um dos seguintes predicados consoante do tipo de tabuleiro:

```
randomize_board_major.
```

ou

```
randomize_board_minor.
```

Tal como foi referido na secção 3.1, nós recorremos à base de dados do PROLOG para guardar o estado do tabuleiro, pelo que não se torna necessário retornar informação nenhuma acerca deste.

A figura 9 mostra uma imagem com o output produzido.

3.3 Lista de Jogadas Válidas

Para se obter uma lista de jogadas válidas para cada jogador, recorre-se ao seguinte predicado:

```
available_actions(+Player, -Actions).
```

Em Actions é retornada uma lista de listas, cada lista contendo informação acerca do tipo de jogada (Afundar uma peça, mover uma peça, mover uma torre ou passar a vez), bem como as coordenadas necessárias para cada tipo de jogada.

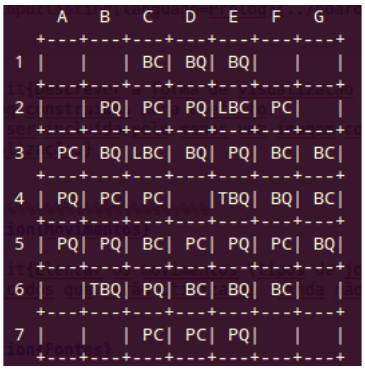

Figura 9: Output produzido no terminal para o tabuleiro

3.4 Execução de Jogadas

Em cada jogada, é pedido ao jogador para decidir qual a jogada, de entre as 4 possíveis, que pretende fazer, bem como as coordenadas necessárias para fazer a jogada. Cada uma destas jogadas passa por um processo de validação, que, dependendo do tipo de jogada, passa por um destes 3 predicados:

```
valid_slide(+X,+Y,+NX,+NY).
valid_sink(+X,+Y).
valid_move(+X,+Y,+NX,+NY).
```

Passar a vez não envolve qualquer tipo de verificação. Caso o predicado falhe, o *backtracking* do PROLOG levará a que seja apresentado ao jogador a mensagem inicial de escolher uma jogada, fazendo com que este possa tentar outra vez, sucessivamente, até fazer uma jogada válida.

A figura 10 mostra um exemplo de um jogador a realizar uma jogada.

3.5 Avaliação do Tabuleiro

A avaliação do tabuleiro é feita recorrendo ao seguinte predicado:

```
evaluate_board(+Player, -Score).
```

Esta avaliação é feita recorrendo a uma função de avaliação, que utiliza os seguintes critérios:

3.5.1 Critério do número de peças

Este critério baseia-se no facto de que quanto mais peças da nossa cor/forma e quanto menos peças da cor/forma do adversário há no tabuleiro, melhor. O cálculo deste critério é feito recorrendo à seguinte formula:

$$Resultado = numPecasCorBot + numPecasFormaBot - (numPecasCorJogador + numPecasFormaBot)$$

3.5.2 Critério da proximidade ao centro

Este critério baseia-se no facto de que quanto mais próxima uma peça está do centro, mais valiosa é, ou seja, devemos dar prioridade a manter as nossas peças no centro, enquanto afundamos as peças do adversário que estejam

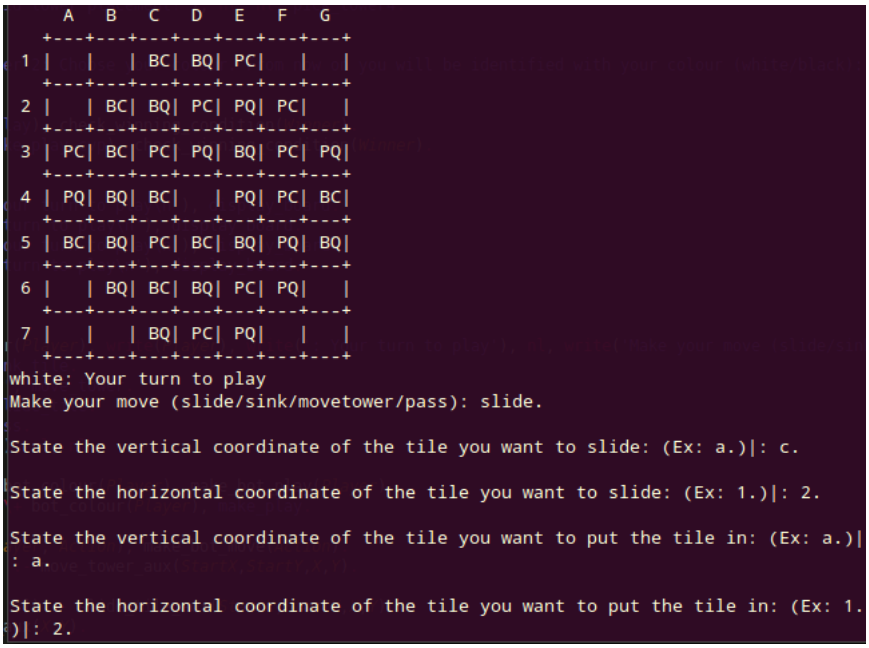

Figura 10: Exemplo de uma jogada.

mais próximas do centro. Infelizmente este é um critério muito pesado computacionalmente, visto que envolve uma elevada quantidade de cálculos, pelo que restringimos esta avaliação apenas às peças que é possível afundar tendo em conta o estado atual do jogo. O cálculo deste critério é feito recorrendo à seguinte fórmula:

$$Resultado = \sum_{i=1}^{pBot} 10 * ((MaxDist - Dist)/MaxDist) - \sum_{i=1}^{pJog} 10 * ((MaxDist - Dist)/MaxDist)$$

Em que *pBot* e *pJog* representam as peças do Bot e do Jogador, respetivamente, *MaxDist* a distância máxima possível de uma peça ao centro e *Dist* a distância da peça ao centro.

3.5.3 Critério das ilhas

Este critério baseia-se no facto de que quanto mais peças tiver uma ilha em relação ao número total das peças do tipo dessa ilha, mais perto estamos de ganhar o jogo e, consequentemente, o tabuleiro tem mais valor. Mais uma vez, por este também ser um critério pesado computacionalmente, restringimos este cálculo apenas às ilhas onde se encontram as quatro torres. A fórmula para o cálculo deste critério é a seguinte:

$$Resultado = 50 * ((TamIlhaCorBot1 + TamIlhaCorBot2)/NCorBot) + ((TamIlhaFormaBot1 + TamIlhaFormaBot2)/NFormaBot) - (((TamIlhaCorJog1 + TamIlhaCorJog2)/NCorJog) + ((TamIlhaFormaJog1 + TamIlhaFormaJog2)/NFormaJog))$$

Os números 1 e 2 representam as torres 1 e 2 de cada jogador.

3.5.4 Critério da sequência de peças afundadas

Este critério baseia-se no facto de que quanto maior estiver uma sequência de peças afundadas para um jogador, mais valioso está o tabuleiro para esse jogador, já que se um jogador afundar 4 peças sem o outro afundar nenhuma, esse jogador ganha o jogo. A fórmula para o cálculo deste critério é a seguinte:

Se o bot estiver com a sequência atual:

$$Resultado = 20 * NPecasAfundadas$$

Se o jogador estiver com a sequência atual:

$$Resultado = -20 * NPecasAfundadas$$

3.5.5 Critério de vitória

Este critério testa se alguma jogada leva a alguma condição de vitória, e se encontrar alguma, atribui um valor de 10000 a essa jogada, fazendo com que seja sempre a jogada escolhida. No caso em que alguma jogada leve à vitória do jogador, atribui-lhe um valor de -10000.

3.6 Final do Jogo

Sempre que o ciclo de jogo é executado, o programa testa se alguma das condições de vitória foi atingida, chamando o predicado

```
check_winning_condition(-Winner)
```

Se este predicado suceder, então o vencedor estará em Winner. Caso contrário, o ciclo continua. Este predicado testa as 3 condições de vitória descritas na secção 2.7.

3.7 Jogada do Computador

O computador tem dois modos possíveis de funcionamento: um modo fácil e um modo difícil. No modo fácil, ele simplesmente escolhe uma jogada aleatoriamente, não se preocupando se a jogada é prejudicial ou não. No modo difícil, o computador avalia os critérios descritos na secção 3.5 para cada jogada possível, e escolhe a jogada que lhe permite obter um tabuleiro mais benéfico.

O predicado que determina a jogada do computador é o seguinte:

```
bot_action(+Difficulty, +Player, -Action).
```

Em que Difficulty é 0 se for o modo fácil e 1 se for o modo difícil, player é a cor do bot e Action a acção determinada.

4 Interface com o Utilizador

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

5 Conclusões

O jogo exigiu ao grupo bastante tempo e dedicação para a sua implementação. No entanto, estávamos cientes, quando o escolhemos, de que assim seria, pois era para nós bastante claro que era um dos jogos mais complexos da lista disponibilizada pelos professores, pelo que isto não nos surpreendeu.

As diversas dificuldades que as regras do jogo nos proporcionaram, dada a sua complexidade, foi superada com sucesso. A maior dificuldade terá mesmo sido a implementação da validação das jogadas possíveis.

O Syrtis é um jogo com uma componente estratégica bastante apelativa, e, na nossa opinião, conseguimos criar uma representação deste jogo bastante apelativa, apesar das dificuldades que o seu tabuleiro proporciona a uma representação textual.

Em suma, o grupo gostou bastante da experiência de desenvolver um jogo em PROLOG. O raciocínio lógico necessário apresentou-se uma dificuldade no início, visto que não estávamos familiarizados com este paradigma de programação, mas a pouco e pouco fomo-nos ambientando cada vez melhor e saímos deste projeto definitivamente bastante mais experientes no que a ele toca.

A Código Fonte

```
%MODULES
:- use_module(library(random)).
:- use_module(library(lists)).
:- dynamic board_cell/3.
:- dynamic board_length/1.
:- dynamic sink_streak/2.
:- dynamic current_player/1.
:- dynamic number_squares/1.
:- dynamic number_circles/1.
:- dynamic number_blacks/1.
:- dynamic number_whites/1.
:- dynamic number_pass/2.
:- dynamic bot_colour/1.

%Database manipulation
purge_database(N) :-    N > 0, purge_database_aux(0,0),
    retract(board_length(N)), retract(sink_streak(_,_)),
    retract(current_player(_)),
retract(number_circles(_)), retract(number_squares(_)),
    retract(number_blacks(_)), retract(number_whites(_)),
retract(number_pass('white',_)), retract(number_pass('
    black',_)), (bot_colour(_) -> retract(bot_colour(_));
    true).
purge_database_aux(Row, Col) :- board_length(Length), Row
    < Length, Col < Length, !, retract(board_cell(Row,
    Col, _)), NCol is Col + 1, purge_database_aux(Row,NCol
    ).
purge_database_aux(Row, _) :- board_length(Length), Row <
    Length, !, NRow is Row + 1, purge_database_aux(NRow,
    0).
purge_database_aux(Row, _) :- board_length(Row).

create_database(N) :-    N > 0, assert(number_squares(0)),
    assert(number_circles(0)), assert(number_blacks(0)),
    assert(number_whites(0)),
assert(sink_streak('white', 0)), assert(current_player('
    white')), assert(number_pass('white', 0)), assert(
    number_pass('black', 0)),
assert(board_length(N)), create_database_aux(0, 0).
create_database_aux(Row, Col) :- board_length(Length),
    Row < Length, Col < Length, !, assert(board_cell(Row,
    Col, ['␣', '␣', '␣'])), NCol is Col + 1,
    create_database_aux(Row,NCol).
create_database_aux(Row, _) :- board_length(Length), Row
    < Length, !, NRow is Row + 1, create_database_aux(NRow
    , 0).
create_database_aux(Row, _) :- board_length(Row).

%Board display
display_board :- write_col_coords, display_board_row(0).
```

```
display_board_row(Row) :- board_length(Length), Row <
    Length, !, write_border, write_line(Row), NRow is Row
    + 1, display_board_row(NRow).
display_board_row(Row) :- board_length(Row), !,
    write_border.

write_border :- write('␣␣␣'), write_border_aux(0).
write_border_aux(Col) :- board_length(Col), !, write('+\n
    ').
write_border_aux(Col) :- board_length(Length), Col <
    Length, !, write('+--'), NCol is Col + 1,
    write_border_aux(NCol).

write_aux_line(_, Col) :- board_length(Col), !, write('|\
    n').
write_aux_line(Row, Col) :- board_length(Length), Col <
    Length, !, write('|'), write_elem(Row, Col), NCol is
    Col + 1, write_aux_line(Row, NCol).

write_line(Row) :- Number is Row + 1, format('␣~d␣', [
    Number]), write_aux_line(Row, 0).

write_elem(Row,Col) :- board_cell(Row, Col, [Tower,
    Colour, Shape]), write(Tower), write(Colour), write(
    Shape).

write_col_coords :- write('␣␣␣'), write_col_coords_aux(0)
    .
write_col_coords_aux(Col) :- board_length(Length), Col <
    Length, !, Charcode is Col + 65, format('␣␣~c␣', [
    Charcode]), NCol is Col + 1, write_col_coords_aux(NCol
    ).
write_col_coords_aux(Col) :- board_length(Length), Col is
     Length, !, nl.

%Insert and remove a tower in a given place
insert_tower(X,Y,'L') :- board_cell(X,Y,['␣',_,_]), \+
    board_cell(X,Y,[_,'P','Q']), \+ board_cell(X,Y,[_,'␣'
    ,'␣']), board_cell(X,Y,[_,Colour,Shape]), change_tile
    (X,Y,['L',Colour,Shape]).
insert_tower(X,Y,'T') :- board_cell(X,Y,['␣',_,_]), \+
    board_cell(X,Y,[_,'B','C']), \+ board_cell(X,Y,[_,'␣'
    ,'␣']), board_cell(X,Y,[_,Colour,Shape]), change_tile
    (X,Y,['T',Colour,Shape]).
remove_tower(X,Y) :- board_cell(X,Y,[_,Colour,Shape]),
    change_tile(X,Y,['␣', Colour, Shape]).

%Start game
start_game :- pick_play_mode.
pick_play_mode :- write('Please␣state␣the␣desired␣play␣
    mode␣(1-␣Player␣vs␣Computer␣2-␣Player␣vs␣Player)'),
    read(Mode), test_mode(Mode).
```

```
test_mode(1) :- game_cvp.
test_mode(2) :- game_pvp.
test_mode(_) :- nl, write('Invalid mode!'), nl,
    pick_play_mode.
game_pvp :- once(ask_board), once(pick_tower), !,
    pick_colour, game_cycle(Winner), end_game(Winner).
game_cvp :- once(ask_board), once(pick_tower), !,
    bot_pick_colour(Colour), assert(bot_colour(Colour)),
    nl, write('Bot picked the '), write(Colour), write('
    colour'), nl, game_cycle(Winner), end_game(Winner).

%Creates the chosen board.
create_board(minor) :- create_database(5),
    randomize_board_minor.
create_board(major) :- create_database(7),
    randomize_board_major.
create_board(_) :- write('Invalid type of board!'), nl,
    ask_board.

%Randomize board
randomize(N) :- random(0,4,N).

/***************

Database modifiers

****************/

%Changes a tile's content
add_tile(X,Y,[Tower,Colour, Shape]) :- board_cell(X,Y,['
    ',' ',' ']), change_tile(X, Y, [Tower,Colour,Shape]),
    add_colour_shape(Colour, Shape).
change_tile(X,Y,[Tower,Colour,Shape]) :- retract(
    board_cell(X,Y,_)), assert(board_cell(X,Y,[Tower,
    Colour,Shape])).
remove_tile(X, Y) :- board_cell(X, Y, [_,Colour,Shape]),
    change_tile(X,Y,[' ',' ',' ']), remove_colour_shape(
    Colour, Shape).

%Adds a colour/shape to the colour/shape counter
add_colour_shape(Colour, Shape) :- add_colour(Colour),
    add_shape(Shape).
add_colour('B') :- number_whites(N), NW is N+1, retract(
    number_whites(N)), assert(number_whites(NW)).
add_colour('P') :- number_blacks(N), NB is N+1, retract(
    number_blacks(N)), assert(number_blacks(NB)).
add_shape('Q') :- number_squares(N), NS is N+1, retract(
    number_squares(N)), assert(number_squares(NS)).
add_shape('C') :- number_circles(N), NC is N+1, retract(
    number_circles(N)), assert(number_circles(NC)).

%Removes a colour/shape to the colour/shape counter
```

```
remove_colour_shape(Colour, Shape) :- remove_colour(
    Colour), remove_shape(Shape).
remove_colour('B') :- number_whites(N), NW is N-1,
    retract(number_whites(N)), assert(number_whites(NW)).
remove_colour('P') :- number_blacks(N), NB is N-1,
    retract(number_blacks(N)), assert(number_blacks(NB)).
remove_shape('Q') :- number_squares(N), NS is N-1,
    retract(number_squares(N)), assert(number_squares(NS))
    .
remove_shape('C') :- number_circles(N), NC is N-1,
    retract(number_circles(N)), assert(number_circles(NC))
    .

%Tiles sinked counter - used to check win condition
sink_count(Player) :- sink_streak(Player,Streak), NStreak
     is Streak+1, retract(sink_streak(_, Streak)), assert(
    sink_streak(Player,NStreak)).
sink_count(Player) :- sink_streak(OPlayer,_), Player \=
    OPlayer, retract(sink_streak(_,_)), assert(sink_streak
    (Player, 1)).

%Change current Player
change_player :- retract(current_player('white')), assert
    (current_player('black')).
change_player :- retract(current_player('black')), assert
    (current_player('white')).

%Change pass
increment_pass(Player) :- retract(number_pass(Player,
    Pass)), NPass is Pass + 1, assert(number_pass(Player,
    NPass)).
reset_pass(Player) :- retract(number_pass(Player, _)),
    assert(number_pass(Player, 0)).

% Board randomizer
randomize_board_major :- randomize(N), replace_board
    (3,2,3,4,N), randomize_board_major_3.
randomize_board_major_3 :- randomize(N), replace_board
    (3,1,3,5,N), randomize_board_major_5.
randomize_board_major_5 :- randomize(N), replace_board
    (3,0,3,6,N), randomize_board_major_7.
randomize_board_major_7 :- randomize(N), replace_board
    (2,0,4,6,N), randomize_board_major_9.
randomize_board_major_9 :- randomize(N), replace_board
    (2,1,4,5,N), randomize_board_major_11.
randomize_board_major_11 :- randomize(N), replace_board
    (2,2,4,4,N), randomize_board_major_13.
randomize_board_major_13 :- randomize(N), replace_board
    (2,3,4,3,N), randomize_board_major_15.
randomize_board_major_15 :- randomize(N), replace_board
    (2,4,4,2,N), randomize_board_major_17.
randomize_board_major_17 :- randomize(N), replace_board
```

```
    (2,5,4,1,N), randomize_board_major_19.
randomize_board_major_19 :- randomize(N), replace_board
    (2,6,4,0,N), randomize_board_major_21.
randomize_board_major_21 :- randomize(N), replace_board
    (1,5,5,1,N), randomize_board_major_23.
randomize_board_major_23 :- randomize(N), replace_board
    (1,4,5,2,N), randomize_board_major_25.
randomize_board_major_25 :- randomize(N), replace_board
    (1,3,5,3,N), randomize_board_major_27.
randomize_board_major_27 :- randomize(N), replace_board
    (1,2,5,4,N), randomize_board_major_29.
randomize_board_major_29 :- randomize(N), replace_board
    (1,1,5,5,N), randomize_board_major_31.
randomize_board_major_31 :- randomize(N), replace_board
    (0,2,6,4,N), randomize_board_major_33.
randomize_board_major_33 :- randomize(N), replace_board
    (0,3,6,3,N), randomize_board_major_35.
randomize_board_major_35 :- randomize(N), replace_board
    (0,4,6,2,N).

randomize_board_minor :- randomize(N), replace_board
    (2,1,2,3,N), randomize_board_minor_3.
randomize_board_minor_3 :- randomize(N), replace_board
    (2,0,2,4,N), randomize_board_minor_5.
randomize_board_minor_5 :- randomize(N), replace_board
    (1,0,3,4,N), randomize_board_minor_7.
randomize_board_minor_7 :- randomize(N), replace_board
    (1,1,3,3,N), randomize_board_minor_9.
randomize_board_minor_9 :- randomize(N), replace_board
    (1,2,3,2,N), randomize_board_minor_11.
randomize_board_minor_11 :- randomize(N), replace_board
    (1,3,3,1,N), randomize_board_minor_13.
randomize_board_minor_13 :- randomize(N), replace_board
    (0,3,4,1,N), randomize_board_minor_15.
randomize_board_minor_15 :- randomize(N), replace_board
    (0,2,4,2,N).

% Auxiliary function of the randomizer. 0-BC 1-PC 2-BQ 3-
    PQ
replace_board(X1,Y1,X2,Y2,0) :- change_tile(X1,Y1,['␣','B
    ','C']), change_tile(X2,Y2,['␣','P','Q']),
    add_colour_shape('B', 'C'), add_colour_shape('P', 'Q')
    .
replace_board(X1,Y1,X2,Y2,1) :- change_tile(X1,Y1,['␣','P
    ','C']), change_tile(X2,Y2,['␣','B','Q']),
    add_colour_shape('P', 'C'), add_colour_shape('B', 'Q')
    .
replace_board(X1,Y1,X2,Y2,2) :- change_tile(X1,Y1,['␣','B
    ','Q']), change_tile(X2,Y2,['␣','P','C']),
    add_colour_shape('B', 'Q'), add_colour_shape('P', 'C')
    .
replace_board(X1,Y1,X2,Y2,3) :- change_tile(X1,Y1,['␣','P
```

```
    ','Q']), change_tile(X2,Y2,[' ','B','C']),
    add_colour_shape('P', 'Q'), add_colour_shape('B', 'C')
    .

%Asks the type of board
ask_board :- write('Please state the board you want (
    major/minor): '), read(X), create_board(X).

%After a winner is found, end the game
end_game(Winner) :- nl, write('Player '), write(Winner),
    write(' has won the game!'), board_length(Length),
    purge_database(Length).

% Player 1 picks the towers
pick_tower_aux(Character, Number, Tower) :- Tower == 'L',
     char_code(Character,Charcode), write('\n'), Y is
    Charcode-97, X is Number-1, insert_tower(X, Y, Tower).
pick_tower_aux(Character, Number, Tower) :- Tower == 'T',
     char_code(Character,Charcode), write('\n'), Y is
    Charcode-97, X is Number-1, insert_tower(X, Y, Tower).

pick_tower :- display_board, write('Player 1: State the
    vertical coordinate of the first white tower: (Ex: a.)
    '), read(Character), write('State the horizontal
    coordinate of the first white tower: (Ex: 1.)'), read(
    Number), validate_pick_tower(Character, Number), once(
    pick_tower2).
validate_pick_tower(Character, Number) :- integer(Number)
    , board_length(Length), Number =< Length,
    pick_tower_aux(Character, Number, 'L'), !.
validate_pick_tower(_,_) :- write('Invalid tower
    placement!'), nl, nl, pick_tower.
pick_tower2 :- display_board, write('Player 1: State the
    vertical coordinate of the second white tower: (Ex: a
    .)'), read(Character), write('State the horizontal
    coordinate of the second white tower: (Ex: 1.)'), read
    (Number), validate_pick_tower2(Character, Number),
    once(pick_tower3).
validate_pick_tower2(Character, Number):- integer(Number)
    , board_length(Length), Number =< Length,
    pick_tower_aux(Character, Number, 'L'), !.
validate_pick_tower2(_,_) :- write('Invalid tower
    placement!'), nl, nl, pick_tower2.
pick_tower3 :- display_board, write('Player 1: State the
    vertical coordinate of the first black tower: (Ex: a.)
    '), read(Character), write('State the horizontal
    coordinate of the first black tower: (Ex: 1.)'), read(
    Number), validate_pick_tower3(Character, Number), once
    (pick_tower4).
validate_pick_tower3(Character, Number) :- integer(Number
    ), board_length(Length), Number =< Length,
    pick_tower_aux(Character, Number, 'T'), !.
```

```
validate_pick_tower3(_,_) :- write('Invalid tower
    placement!'), nl, nl, pick_tower3.
pick_tower4 :- display_board, write('Player 1: State the
    vertical coordinate of the second black tower: (Ex: a
    .)'), read(Character), write('State the horizontal
    coordinate of the second black tower: (Ex: 1.)'), read
    (Number), validate_pick_tower4(Character, Number).
validate_pick_tower4(Character, Number) :- integer(Number
    ), board_length(Length), Number =< Length,
    pick_tower_aux(Character, Number, 'T'), !.
validate_pick_tower4(_,_) :- write('Invalid tower
    placement!'), nl, nl, pick_tower4.

% Player two picks the colour
pick_colour :- display_board, write('Player 2: Choose
    your colour. From now on you will be identified with
    your colour (white/black): '), read(Colour),
    colour_picked(Colour).

% Game cycle
game_cycle(Winner) :- repeat, once(make_play),
    check_winning_condition(Winner).
game_cycle_cvp(Winner) :- repeat, once(make_play_cvp),
    check_winning_condition(Winner).

% Play time!
colour_picked('white') :- write('White: Your turn to play
    \n'), display_board.
colour_picked('w') :- write('White: Your turn to play\n')
    , display_board.
colour_picked('black') :- write('White: Your turn to play
    \n'), display_board.
colour_picked('b') :- write('White: Your turn to play\n')
    , display_board.

% Make play

make_play :- display_board, current_player(Player), write
    (Player), write(': Your turn to play'), nl, write('
    Make your move (slide/sink/movetower/pass): '), read(
    Move), make_play_aux(Move).
make_play_aux(Move) :- Move == 'sink', sink_tile.
make_play_aux(Move) :- Move == 'movetower', move_tower.
make_play_aux(Move) :- Move == 'slide', slide_tile.
make_play_aux(Move) :- Move == 'pass', pass.
make_play_aux(_) :- write('Invalid move!'), nl, nl.

make_play_cvp :- current_player(Player), bot_colour(
    Player), make_bot_play(Player).
make_play_cvp :- current_player(Player), \+ bot_colour(
    Player), make_play.
```

```
make_bot_play(Player) :- bot_action(1, Player, Action),
    make_bot_move(Action).
make_bot_move(['move',StartX,StartY,X,Y]) :-
    move_tower_aux(StartX,StartY,X,Y).
make_bot_move(['pass']) :- pass.
make_bot_move(['slide',StartX, StartY, X, Y]) :-
    slide_tile_aux(StartX,StartY,X,Y).
make_bot_move(['sink',X,Y]) :- sink_tile_aux(X,Y).

% Treat each play individually
slide_tile :-    write('\nState the vertical coordinate of
    the tile you want to slide: (Ex: a.)'), read(
    Character),
write('\nState the horizontal coordinate of the tile you
    want to slide: (Ex: 1.)'), read(Number),
write('\nState the vertical coordinate of the tile you
    want to put the tile in: (Ex: a.)'), read(NCharacter),
write('\nState the horizontal coordinate of the tile you
    want to put the tile in: (Ex: 1.)'), read(NNumber),
char_code(Character,Charcode), write('\n'), Y is Charcode
    -97, X is Number-1,
char_code(NCharacter,NCharcode), write('\n'), NY is
    NCharcode-97, NX is NNumber-1,
slide_tile_aux(X,Y,NX,NY).

sink_tile :-    write('\nState the vertical coordinate of
    the tile you want to remove: (Ex: a.)'), read(
    Character),
write('\nState the horizontal coordinate of the tile you
    want to remove: (Ex: 1.)'), read(Number),
char_code(Character,Charcode), write('\n'), Y is Charcode
    -97, X is Number-1, write(' '),
sink_tile_aux(X,Y).

move_tower :-    write('\nState the vertical coordinate of
    the tower you want to move: (Ex: a.)'), read(
    Character),
write('\nState the horizontal coordinate of the tower you
    want to move: (Ex: 1.)'), read(Number),
write('\nState the vertical coordinate of the tower you
    want to move the tower to: (Ex: a.)'), read(NCharacter
    ),
write('\nState the horizontal coordinate of the tower you
    want to move the tile to: (Ex: 1.)'), read(NNumber),
char_code(Character,Charcode), write('\n'), Y is Charcode
    -97, X is Number-1,
char_code(NCharacter,NCharcode), write('\n'), NY is
    NCharcode-97, NX is NNumber-1,
move_tower_aux(X,Y,NX,NY).

slide_tile_aux(X,Y,NX,NY) :- valid_slide(X,Y,NX,NY),
    board_cell(X,Y,Elem), change_tile(NX,NY,Elem),
```

```
    change_tile(X,Y,['␣', '␣', '␣']), current_player(
    Player), reset_pass(Player), once(change_player).
slide_tile_aux(_,_,_,_) :- write('Invalid␣move!'), nl, nl
    .
sink_tile_aux(X,Y) :- valid_sink(X,Y), board_cell(X, Y, [
    '␣',C,S]), remove_colour_shape(C,S), change_tile(X,Y,[
    '␣', '␣', '␣']), current_player(Player), sink_count(
    Player), reset_pass(Player), once(change_player).
sink_tile_aux(_,_) :- write('Invalid␣move!'), nl, nl.
move_tower_aux(X,Y,NX,NY) :- valid_move(X,Y,NX,NY),
    board_cell(X,Y,[Tower|_]), insert_tower(NX, NY, Tower)
    , remove_tower(X,Y), current_player(Player),
    reset_pass(Player), once(change_player).
move_tower_aux(_,_,_,_) :- write('Invalid␣move!'), nl, nl
    .
pass :- current_player(Player), increment_pass(Player),
    once(change_player).

player_tower('white', 'L').
player_tower('black', 'T').

% Check end game condition
check_winning_condition(Winner) :- sink_streak(Winner, 4)
    .
check_winning_condition(Winner) :- completed_island(
    Player1), completed_island(Player2), Player1 \=
    Player2, !, resolve_initiative(Winner).
check_winning_condition(Winner) :- completed_island(
    Winner), !.
check_winning_condition(Winner) :- number_pass(Player1,
    1), number_pass(Player2, 1), Player1 \= Player2, !,
    resolve_initiative(Winner).
check_winning_condition(Winner) :- number_pass(_, 4),
    number_pass(Winner, 0), !.

completed_island('white') :- completed_light_island.
completed_island('white') :- completed_circle_island.
completed_island('black') :- completed_dark_island.
completed_island('black') :- completed_square_island.

resolve_initiative(Winner) :- sink_streak(Winner,_).

completed_dark_island :- board_cell(X,Y,[_,'P',_]),
    dark_island(X,Y,Island), number_blacks(N), length(
    Island, N).
completed_light_island :- board_cell(X,Y,[_,'B',_]),
    light_island(X,Y,Island), number_whites(N), length(
    Island, N).
completed_square_island :- board_cell(X,Y,[_,_,'Q']),
    square_island(X,Y,Island), number_squares(N), length(
    Island, N).
completed_circle_island :- board_cell(X,Y,[_,_,'C']),
```

```
    circle_island(X,Y,Island), number_circles(N), length(
    Island, N).

valid_slide(X, Y, FinalX, FinalY) :- board_cell(X,Y,[
    Tower,_,_]), current_player(Player), player_tower(
    Player,Tower),slidable_tiles(X,Y,Tiles), member([
    FinalX,FinalY], Tiles).
valid_move(X,Y,NX,NY) :-          board_cell(X,Y,[Tower,_,_
    ]), current_player(Player), player_tower(Player,Tower)
    ,
board_cell(NX,NY, ['␣',_,_]), [X, Y] \= [NX, NY],
    valid_move_aux(X,Y,NX,NY).

valid_move_aux(X,Y,NX,NY) :- board_cell(X,Y,['L','B',_]),
     light_island(X,Y,Island), member([NX,NY],Island).
valid_move_aux(X,Y,NX,NY) :- board_cell(X,Y,['L',_,'C']),
     circle_island(X,Y,Island), member([NX,NY],Island).
valid_move_aux(X,Y,NX,NY) :- board_cell(X,Y,['T','P',_]),
     dark_island(X,Y,Island), member([NX,NY],Island).
valid_move_aux(X,Y,NX,NY) :- board_cell(X,Y,['T',_,'Q']),
     square_island(X,Y,Island), member([NX,NY],Island).

valid_sink(X, Y) :- current_player(Player), player_tower(
    Player, Tower), tower_positions(Tower, [[X1, Y1], [X2,
     Y2]]),
sinkable_tiles(X1, Y1, Tiles1), sinkable_tiles(X2, Y2,
    Tiles2), append(Tiles1, Tiles2, Sinkable),
member([X,Y], Sinkable).

connected_board :-        board_cell(X, Y, [_, 'P', _]),
    reachable_tiles(X, Y, Tiles), length(Tiles, L),
    number_blacks(B), number_whites(W), !, L is B + W.

reachable_tiles(X, Y, Tiles) :- reachable_tiles_aux([[X,Y
    ]],[], Tiles).
reachable_tiles_aux([],_, []).
reachable_tiles_aux([Next|T], Visited, Reachable) :-
    member(Next, Visited), !, reachable_tiles_aux(T,
    Visited, Reachable).
reachable_tiles_aux([[X, Y]|T], Visited, [[X, Y]|
    Reachable]) :- board_cell(X, Y, Cell), Cell \= ['␣', '
    ␣', '␣'], !,
neighbour_tiles(X,Y,Neighbours), append(T, Neighbours, NT
    ),
reachable_tiles_aux(NT, [[X,Y]|Visited], Reachable).
reachable_tiles_aux([Next|T], Visited, Reachable) :-
    reachable_tiles_aux(T, [Next|Visited], Reachable).

%neighbour positions
neighbour_tiles(X, Y, [Alt1, Alt2, Alt3, Alt4]) :-
NX is X + 1, PX is X - 1, NY is Y + 1, PY is Y - 1,
Alt1 = [NX, Y], Alt2 = [PX, Y], Alt3 = [X, NY], Alt4 = [X
```

```
    , PY].

%searching islands
dark_island(X, Y, Island) :- board_cell(X,Y,[_,'P',_]),
    !, dark_island_search([[X,Y]], [], Island).
dark_island(X, Y, []) :- \+ board_cell(X,Y,[_,'P',_]), !.
dark_island_search([], _, []).
dark_island_search([Tile|T], Visited, Island) :- member(
    Tile, Visited), !, dark_island_search(T,Visited,Island
    ).
dark_island_search([[X,Y]|T], Visited, [[X,Y]|Island]) :-
     \+ member([X,Y],Visited), board_cell(X,Y,[_,'P',_]),
    !,
neighbour_tiles(X,Y,Neighbours), append(T, Neighbours, NT
    ),
dark_island_search(NT, [[X,Y]|Visited], Island).
dark_island_search([[X,Y]|T], Visited, Island) :- \+
    member([X,Y], Visited), \+ board_cell(X,Y,[_,'P',_]),
    !,
dark_island_search(T,[[X,Y]|Visited], Island).

light_island(X, Y, Island) :- board_cell(X,Y,[_,'B',_]),
    !, light_island_search([[X,Y]], [], Island).
light_island(X, Y, []) :- \+ board_cell(X,Y,[_,'B',_]),
    !.
light_island_search([], _, []).
light_island_search([Tile|T], Visited, Island) :- member(
    Tile, Visited), !, light_island_search(T,Visited,
    Island).
light_island_search([[X,Y]|T], Visited, [[X,Y]|Island])
    :- \+ member([X,Y],Visited), board_cell(X,Y,[_,'B',_])
    , !,
neighbour_tiles(X,Y,Neighbours), append(T, Neighbours, NT
    ),
light_island_search(NT, [[X,Y]|Visited], Island).
light_island_search([[X,Y]|T], Visited, Island) :- \+
    member([X,Y], Visited), \+ board_cell(X,Y,[_,'B',_]),
    !,
light_island_search(T,[[X,Y]|Visited], Island).

circle_island(X, Y, Island) :- board_cell(X,Y,[_,_,'C']),
     !, circle_island_search([[X,Y]], [], Island).
circle_island(X, Y, []) :- \+ board_cell(X,Y,[_,_,'C']),
    !.
circle_island_search([], _, []).
circle_island_search([Tile|T], Visited, Island) :- member
    (Tile, Visited), !, circle_island_search(T,Visited,
    Island).
circle_island_search([[X,Y]|T], Visited, [[X,Y]|Island])
    :- \+ member([X,Y],Visited), board_cell(X,Y,[_,_,'C'])
    , !,
neighbour_tiles(X,Y,Neighbours), append(T, Neighbours, NT
```

```
    ),
circle_island_search(NT, [[X,Y]|Visited], Island).
circle_island_search([[X,Y]|T], Visited, Island) :- \+
    member([X,Y], Visited), \+ board_cell(X,Y,[_,_,'C']),
    !,
circle_island_search(T,[[X,Y]|Visited], Island).

square_island(X, Y, Island) :- board_cell(X,Y,[_,_,'Q']),
     !, square_island_search([[X,Y]], [], Island).
square_island(X, Y, []) :- \+ board_cell(X,Y,[_,_,'Q']),
    !.
square_island_search([], _, []).
square_island_search([Tile|T], Visited, Island) :- member
    (Tile, Visited), !, square_island_search(T,Visited,
    Island).
square_island_search([[X,Y]|T], Visited, [[X,Y]|Island])
    :- \+ member([X,Y],Visited), board_cell(X,Y,[_,_,'Q'])
    , !,
neighbour_tiles(X,Y,Neighbours), append(T, Neighbours, NT
    ),
square_island_search(NT, [[X,Y]|Visited], Island).
square_island_search([[X,Y]|T], Visited, Island) :- \+
    member([X,Y], Visited), \+ board_cell(X,Y,[_,_,'Q']),
    !,
square_island_search(T,[[X,Y]|Visited], Island).

%sinkable tiles

tower('L').
tower('T').
tower_positions(Tower, [[X1, Y1], [X2, Y2]]) :- tower(
    Tower), board_cell(X1,Y1,[Tower,_,_]), board_cell(X2,
    Y2,[Tower,_,_]), [X1,Y1] \= [X2,Y2].

sinkable_tiles(X,Y,Tiles) :- \+board_cell(X,Y,['␣',_,_]),
     !, neighbour_tiles(X,Y,Neighbours), free_edges(
    Neighbours,FreeEdges),
empty_tiles(FreeEdges, EmptyTiles), sinkable_tiles_valid(
    EmptyTiles, Tiles).
sinkable_tiles(_,_,[]).

free_edges([], []).
free_edges([[X,Y]|Tiles], MoreTiles) :- \+board_cell(X,Y,
    _), free_edges(Tiles, MoreTiles).
free_edges([[X,Y]|Tiles], [[X,Y]|MoreTiles]) :- A is X+1,
     \+board_cell(A,Y,_), !, free_edges(Tiles, MoreTiles).
free_edges([[X,Y]|Tiles], [[X,Y]|MoreTiles]) :- B is X-1,
     \+board_cell(B,Y,_), !,free_edges(Tiles, MoreTiles).
free_edges([[X,Y]|Tiles], [[X,Y]|MoreTiles]) :- C is Y+1,
     \+board_cell(X,C,_), !,free_edges(Tiles, MoreTiles).
free_edges([[X,Y]|Tiles], [[X,Y]|MoreTiles]) :- D is Y-1,
     \+board_cell(X,D,_), !,free_edges(Tiles, MoreTiles).
```

```
free_edges([[X,Y]|Tiles], [[X,Y]|MoreTiles]) :- A is X+1,
    board_cell(A,Y,['␣','␣','␣']), !,free_edges(Tiles,
    MoreTiles).
free_edges([[X,Y]|Tiles], [[X,Y]|MoreTiles]) :- B is X-1,
    board_cell(B,Y,['␣','␣','␣']), !,free_edges(Tiles,
    MoreTiles).
free_edges([[X,Y]|Tiles], [[X,Y]|MoreTiles]) :- C is Y+1,
    board_cell(X,C,['␣','␣','␣']), !,free_edges(Tiles,
    MoreTiles).
free_edges([[X,Y]|Tiles], [[X,Y]|MoreTiles]) :- D is Y-1,
    board_cell(X,D,['␣','␣','␣']), !,free_edges(Tiles,
    MoreTiles).
free_edges([_|Tiles], MoreTiles) :- free_edges(Tiles,
    MoreTiles).

empty_tiles([],[]).
empty_tiles([[X,Y]|Tiles], EmptyTiles) :- board_cell(X, Y
    , ['␣','␣','␣']), !, empty_tiles(Tiles, EmptyTiles).
empty_tiles([[X,Y]|Tiles], [[X,Y]|EmptyTiles]) :-
    board_cell(X, Y, ['␣', _, _]), !, empty_tiles(Tiles,
    EmptyTiles).
empty_tiles([[X,Y]|Tiles], EmptyTiles) :- \+ board_cell(X
    , Y, ['␣', _, _]), !, empty_tiles(Tiles, EmptyTiles).

%% change colour and shape counts
sinkable_tiles_valid([],[]).
sinkable_tiles_valid([[X,Y]|Tiles],ValidTiles) :-
    sinkable_tiles_valid(Tiles, VTiles), board_cell(X,Y,
    Cell), remove_tile(X, Y),
(connected_board -> ValidTiles = [[X,Y]|VTiles];
    ValidTiles = VTiles), add_tile(X, Y, Cell).

%searching slidable positions
slidable_tiles(X, Y, Tiles) :- \+ board_cell(X, Y, ['␣',_
    ,_]),
neighbour_tiles(X,Y,Neighbours), slidable_tiles_search(
    Neighbours, [[X,Y]], PTiles),
slidable_tiles_valid(X, Y, PTiles, Tiles).

slidable_tiles_search([],_,[]).
slidable_tiles_search([Tile|T], Visited, PTiles) :-
    member(Tile, Visited), !, slidable_tiles_search(T,
    Visited, PTiles).
slidable_tiles_search([[X,Y]|T], Visited, [[X,Y]|PTiles])
    :- \+ member([X,Y], Visited), board_cell(X, Y, ['␣','
    ␣','␣']), within_board_limits(X,Y), !,
neighbour_tiles(X, Y, Neighbours), append(T, Neighbours,
    NT),
slidable_tiles_search(NT, [[X,Y]|Visited], PTiles).
slidable_tiles_search([[X,Y]|T], Visited, PTiles) :- \+
    member([X,Y], Visited), !,
slidable_tiles_search(T,[[X,Y]|Visited], PTiles).
```

```
slidable_tiles_valid(_,_,[], []).
slidable_tiles_valid(StartX, StartY, [[X,Y]|PTiles],Tiles
    ) :- slidable_tiles_valid(StartX, StartY, PTiles,
    NTiles),
board_cell(StartX,StartY,Cell), remove_tile(StartX,
    StartY), add_tile(X,Y,Cell),
(connected_board -> Tiles = [[X,Y]|NTiles]; Tiles =
    NTiles), add_tile(StartX, StartY, Cell), remove_tile(X
    , Y).

within_board_limits(X, Y) :- board_limits(MinX, MaxX,
    MinY, MaxY), X >= MinX, X =< MaxX, Y >= MinY, Y =<
    MaxY.

tiles_in_X(X, Tiles) :- var(Tiles), tiles_in_X_aux(X, 0,
    Tiles).
tiles_in_X_aux(_, Y, []) :- board_length(Y).
tiles_in_X_aux(X, Y, [[X,Y]|Tiles]) :- board_cell(X, Y, [
    _,Colour,_]), Colour \= '␣', !, NY is Y + 1,
    tiles_in_X_aux(X,NY, Tiles).
tiles_in_X_aux(X, Y, Tiles) :- NY is Y + 1,
    tiles_in_X_aux(X, NY,Tiles).

tiles_in_Y(Y, Tiles) :- var(Tiles), tiles_in_Y_aux(0, Y,
    Tiles).
tiles_in_Y_aux(X, _, []) :- board_length(X).
tiles_in_Y_aux(X, Y, [[X,Y]|Tiles]) :- board_cell(X, Y, [
    _,Colour,_]), Colour \= '␣', !, NX is X + 1,
    tiles_in_Y_aux(NX,Y, Tiles).
tiles_in_Y_aux(X, Y, Tiles) :- NX is X + 1,
    tiles_in_Y_aux(NX, Y,Tiles).

board_limits(MinX, MaxX, MinY, MaxY) :- min_x(MinX),
    max_x(MaxX), min_y(MinY), max_y(MaxY).
min_x(MinX) :- min_x_aux(0, MinX).
min_x_aux(X, MinX) :- once(tiles_in_X(X, Tiles)), Tiles =
     [], !,NX is X + 1, min_x_aux(NX, MinX).
min_x_aux(X, MinX) :- var(MinX), MinX = X.

max_x(MaxX) :- board_length(Length), X is Length - 1,
    max_x_aux(X,MaxX).
max_x_aux(X, MaxX) :- once(tiles_in_X(X, Tiles)), Tiles =
     [], !, PX is X - 1, max_x_aux(PX, MaxX).
max_x_aux(X, MaxX) :- var(MaxX), MaxX = X.

min_y(MinY) :- min_y_aux(0, MinY).
min_y_aux(Y, MinY) :- once(tiles_in_Y(Y, Tiles)), Tiles =
     [], !, NY is Y + 1, min_y_aux(NY, MinY).
min_y_aux(Y, MinY) :- var(MinY), MinY = Y.
```

```
max_y(MaxY) :- board_length(Length), Y is Length - 1,
    max_y_aux(Y,MaxY).
max_y_aux(Y, MaxY) :- once(tiles_in_Y(Y, Tiles)), Tiles =
     [], !, PY is Y - 1, max_y_aux(PY, MaxY).
max_y_aux(Y, MaxY) :- var(MaxY), MaxY = Y.



%Available Actions
available_actions(Player, Actions) :-    pass_actions(
    Player, PassActions), move_actions(Player, MoveActions
    ),
slide_actions(Player, SlideActions), sink_actions(Player,
     SinkActions),
append(PassActions, MoveActions, L1), append(L1,
    SlideActions, L2),
append(L2, SinkActions, Actions).

pass_actions(_, [['pass']]).

move_actions('white', MoveActions) :-    player_tower('
    white', Tower), tower_positions(Tower, [[X1,Y1],[X2,Y2
    ]]), !,
light_island(X1,Y1,LightIsland1), circle_island(X1,Y1,
    CircleIsland1),
append(LightIsland1, CircleIsland1, Island1), list_moves(
    X1, Y1, Island1, MoveActions1),
light_island(X2,Y2,LightIsland2), circle_island(X2,Y2,
    CircleIsland2),
append(LightIsland2, CircleIsland2, Island2), list_moves(
    X2, Y2, Island2, MoveActions2),
append(MoveActions1, MoveActions2, MoveActions).
move_actions('black', MoveActions) :-    player_tower('
    black', Tower), tower_positions(Tower, [[X1,Y1],[X2,Y2
    ]]), !,
dark_island(X1,Y1,DarkIsland1), square_island(X1,Y1,
    SquareIsland1),
append(DarkIsland1, SquareIsland1, Island1), list_moves(
    X1, Y1, Island1, MoveActions1),
dark_island(X2,Y2,DarkIsland2), square_island(X2,Y2,
    SquareIsland2),
append(DarkIsland2, SquareIsland2, Island2), list_moves(
    X2, Y2, Island2, MoveActions2),
append(MoveActions1, MoveActions2, MoveActions).
list_moves(_, _, [], []).
list_moves(StartX, StartY, [[StartX, StartY]|EndList],
    MoveList) :- !, list_moves(StartX, StartY, EndList,
    MoveList).
list_moves(StartX, StartY, [[X, Y]|EndList], [['move',
    StartX,StartY,X,Y]|MoveList]) :- [StartX, StartY] \= [
    X, Y], !,
list_moves(StartX, StartY, EndList, MoveList).
```

```
slide_actions(Player, SlideActions) :- player_tower(
    Player, Tower), tower_positions(Tower, [[X1,Y1],[X2,Y2
    ]]), !,
slidable_tiles(X1,Y1,Slides1), list_slides(X1, Y1,
    Slides1, SlideActions1),
slidable_tiles(X2,Y2,Slides2), list_slides(X2, Y2,
    Slides2, SlideActions2),
append(SlideActions1, SlideActions2, SlideActions).
list_slides(_, _, [], []).
list_slides(StartX, StartY, [[X,Y]|EndList], [['slide',
    StartX, StartY, X, Y]|SlideList]) :- list_slides(
    StartX, StartY, EndList, SlideList).

sink_actions(Player, SinkActions) :- player_tower(
    Player, Tower), tower_positions(Tower, [[X1,Y1],[X2,Y2
    ]]), !,
sinkable_tiles(X1, Y1, Sinks1), list_sinks(Sinks1,
    SinkActions1),
sinkable_tiles(X2, Y2, Sinks2), list_sinks(Sinks2,
    SinkActions2),
append(SinkActions1, SinkActions2, SinkActions).
list_sinks([],[]).
list_sinks([[X,Y]|Tiles], [['sink',X,Y]|SinkList]) :-
    list_sinks(Tiles, SinkList).

%Evaluation funtions
evaluate_board(Player, Score) :- number_tiles_criteria(
    Score1),
sinkable_tiles_criteria(Score2), islands_criteria(Score3)
    ,
sink_streak_criteria(Score4), winning_criteria(Score5),
Calc is Score1 + Score2 + Score3 + Score4 + Score5,
(Player \= 'white' -> Score = Calc; Score is -Calc).

number_tiles_criteria(Score) :- number_blacks(B),
    number_whites(W), number_circles(C), number_squares(S)
    ,
Score is W + C - (B + S).

sinkable_tiles_criteria(Score) :- player_tower('
    white', WTower), tower_positions(WTower, [[WX1,WY1],[
    WX2,WY2]]),
sinkable_tiles(WX1, WY1, WSinks1), sinkable_tiles(WX2,
    WY2, WSinks2),
append(WSinks1, WSinks2, WSinks),
    sinkable_tiles_criteria_aux(WSinks, WScore),
player_tower('black', BTower), tower_positions(BTower, [[
    BX1,BY1],[BX2,BY2]]),
sinkable_tiles(BX1, BY1, BSinks1), sinkable_tiles(BX2,
    BY2, BSinks2),
append(BSinks1, BSinks2, BSinks),
```

```
    sinkable_tiles_criteria_aux(BSinks, BScore),
Score is WScore - BScore.

sinkable_tiles_criteria_aux([], 0).
sinkable_tiles_criteria_aux([[X,Y]|Sinks], Score) :-
    sinkable_tiles_criteria_aux(Sinks, Score1),
sinkable_tiles_score(X, Y, Score2),
Score is Score1 + Score2.
sinkable_tiles_score(X, Y, Score) :-    board_length(
    Length), MaxDist is Length/sqrt(2),
CX is truncate(Length / 2), CY = CX,
Dist is sqrt((CX - X)^2+(CY - Y)^2),
Score is (10 * (MaxDist - Dist) / MaxDist).

islands_criteria(Score) :-      island_score('white',
    WScore), island_score('black', BScore),
Score is WScore - BScore.

island_score('white', Score) :- player_tower('white',
    Tower), tower_positions(Tower,[[X1,Y1],[X2,Y2]]),
light_island(X1,Y1,LightIsland1), circle_island(X1,Y1,
    CircleIsland1),
light_island(X2,Y2,LightIsland2), circle_island(X2,Y2,
    CircleIsland2),
number_whites(Whites), number_circles(Circles),
length(LightIsland1, Length1), length(LightIsland2,
    Length2), length(CircleIsland1, Length3), length(
    CircleIsland2, Length4),
Score is 50*(((Length1 + Length2)/ Whites) + ((Length3+
    Length4) / Circles)).

island_score('black', Score) :- player_tower('black',
    Tower), tower_positions(Tower,[[X1,Y1],[X2,Y2]]),
dark_island(X1,Y1,DarkIsland1), square_island(X1,Y1,
    SquareIsland1),
dark_island(X2,Y2,DarkIsland2), square_island(X2,Y2,
    SquareIsland2),
number_blacks(Blacks), number_squares(Squares),
length(DarkIsland1, Length1), length(DarkIsland2, Length2
    ), length(SquareIsland1, Length3), length(
    SquareIsland2, Length4),
Score is 50*(((Length1 + Length2)/ Blacks) + ((Length3+
    Length4) / Squares)).

sink_streak_criteria(Score) :- sink_streak('white', Sinks
    ), !, Score is 20 * Sinks.
sink_streak_criteria(Score) :- sink_streak('black', Sinks
    ), !, Score is -20 * Sinks.

winning_criteria(Score) :- check_winning_condition(Winner
    ), !,
(Winner = 'white' -> Score is 10000; Score is -10000).
```

```
winning_criteria(Score) :- Score is 0.


bot_pick_colour(Colour) :- evaluate_board('white', Score)
    , bot_pick_colour_aux(Score, Colour).
bot_pick_colour_aux(Score, Colour) :- Score < 0, Colour =
    'black'.
bot_pick_colour_aux(_, Colour) :- Colour = 'white'.

bot_action(0, Player, Action) :- available_actions(Player
    , Actions), length(Actions, Length),
 random(0, Length, Index), nth0(Index,Actions, Action).

bot_action(1, Player, Action) :- available_actions(Player
    , Actions), bot_action_helper(Player, Actions, -10000,
    [], Action).

bot_action_helper(_,[], _, BestAction, BestAction).
bot_action_helper(Player, [Action|Actions], BestScore,
   BestAction, SelectedAction) :-
evaluate_action(Player, Action, Score),
(Score > BestScore -> (NScore = Score, NAction = Action);
    (NScore = BestScore, NAction = BestAction)),
bot_action_helper(Player, Actions, NScore, NAction,
   SelectedAction).

evaluate_action(Player, ['pass'], Score) :-
evaluate_board(Player, Score).

evaluate_action(Player, ['move', X, Y, NX, NY], Score) :-
player_tower(Player, Tower), remove_tower(X, Y),
   insert_tower(NX, NY, Tower),
evaluate_board(Player, Score),
remove_tower(NX, NY), insert_tower(X, Y, Tower).

evaluate_action(Player, ['slide', X, Y, NX, NY], Score)
   :-
board_cell(X, Y, Cell), remove_tile(X,Y), add_tile(NX, NY
    , Cell),
evaluate_board(Player, Score),
remove_tile(NX, NY), add_tile(X, Y, Cell).

evaluate_action(Player, ['sink', X, Y], Score) :-
board_cell(X, Y, Cell), remove_tile(X, Y),
evaluate_board(Player, Score),
add_tile(X, Y, Cell).
```