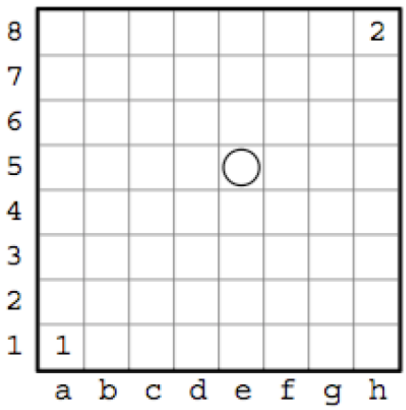
O Jogo do Rastros

O jogo do Rastros é um jogo de tabuleiro para dois jogadores (o *Norte* e o *Sul*) e foi inventado por Bill Taylor em 1992. Joga-se num tabuleiro quadrado 8 por 8. Uma peça branca e pedras pretas em número suficiente (cerca de 60).



Neste tabuleiro a casa a1, marcada com um 1, é a casa final do jogador Norte enquanto a casa h8, marcada com um 2, é a casa final do jogador Sul.
Um jogador ganha se a peça branca estiver na sua casa final ou se for capaz de bloquear o adversário, impedindo-o de jogar. O jogador Sul tenta atingir a casa a1 e o Norte a h8. Cada jogador, alternadamente, desloca a peça branca para um quadrado vazio adjacente (vertical, horizontal ou diagonalmente). A casa onde se encontrava a peça branca recebe uma peça negra. As casas que recebem peças negras não podem ser ocupadas pela peça branca. O jogo começa com a peça branca na casa e5.

Vamos modelizar este jogo. Comecemos pelo tabuleiro. Cada casa do tabuleiro será representada internamente por números. Mais tarde, preocupar-nos-emos com a tradução de letras em números. Cada casa vai ser representada por um par (X,Y). Começamos por definir que o canto esquerdo é a casa (1,1) e o topo direito é a casa (8,8).

O tabuleiro standard do Rastros tem uma dimensão 8 por 8 mas poderíamos ter uma dimensão diferente e para representar essa informação iremos ter um predicado dim/1 (só com 1 argumento) que exprime a dimensão do tabuleiro quadrado.
Este facto pode ser visto como uma variável global.

```
% a dimensão é 8
dim(8).
```

Vamos agora arranjar predicados que nos indiquem a relação entre os jogadores.

```
% o Sul é adversário do Norte e vice versa
adversario(sul,norte).
adversario(norte,sul)
```

Precisamos também de definir quais são as casas a atingir por cada um dos jogadores. O Norte quer chegar ao fundo esquerdo (1,1) e o Sul quer chegar ao topo norte: (dim,dim).

```
% O objectivo do Norte e a casa de fundo à esquerda
objectivo(norte,(1,1)).

% O objectivo do Sul é o topo direito, que depende da dimensão.
objectivo(sul,(D,D)) :-
        dim(D).
```

A localização da peça branca é dada pelo predicado branca/1. Inicialmente não haverá nenhum facto destes pelo que é automaticamente considerado como dinâmico mal seja adicionado pela primeira vez.Mas teremos de saber qual a casa inicial da peça branca e para isso vamos usar um predicado especial. Num jogo a posição da branca muda e precisamos sempre de memorizar onde deve começar a peça branca, se quisermos começar um novo jogo.

```
% A célula inicial da peça branca.
inicial((5,5)).
```

As peças pretas são definidas pelo predicado preta/1. Inicialmente não haverá nenhuma peça preta mas precisaremos de as adicionar sempre que a branca for deslocada, ocupando-se a casa anterior da peça branca com uma peça preta.

Vamos precisar de reiniciar o jogo de cada vez que o quisermos. Retiramos todas as pretas, retiramos a branca e vamos buscar a posição inicial da branca, inserindo-a nessa posição inicial. Também retiramos o facto com o próximo jogador e declaramos que começa o Norte. Precisamos também de saber quem é que vai jogar a seguir. O predicado proximo/1 serve para isso. Será o Norte a abrir o jogo e isso tem de ser declarado quando se faz reset.

```
% reset
reset :-
        retractall(preta(_)),
        retract(branca(_)),
        inicial(Inicial),
        assert(branca(Inicial)),
        retract(proximo(_)),
        assert(proximo(norte)).
```

Como uma peça branca só pode deslocar-se para uma casa adjacente é necessário definir quais as casas adjacentes de uma casa a norte, sul, este, oeste e diagonais. Isto é uma relação que não depende da colocação das peças. São as relações espaciais entre as casas do tabuleiro.

```
% adjacente norte
norte((X,Y),(X,NY)) :-
        dim(D),
        Y < D,
        NY is Y + 1.

% adjacente sul
sul((X,Y),(X,NY)) :-
        Y > 1,
        NY is Y - 1.
```

```
% adjacente oeste
sul((X,Y),(NX,Y)) :-
        X > 1,
        NX is X - 1.

% adjacente oeste
oeste((X,Y),(X,NY)) :-
        dim(D),
        Y < D,
        NY is Y + 1.

% adjacente nordeste
nordeste((X,Y),(NX,NY)) :-
        dim(D),
        Y < D,
        X < D,
        NX is X + 1,
        NY is Y + 1.

% adjacente nordeste
nordoeste((X,Y),(NX,NY)) :-
        dim(D),
        Y < D,
        X > 1,
        NX is X - 1,
        NY is Y + 1.

% adjacente nordeste
sudeste((X,Y),(NX,NY)) :-
        Y > 1,
        X < D,
        NX is X + 1,
        NY is Y - 1.

% adjacente nordeste
sudoeste((X,Y),(NX,NY)) :-
        dim(D),
        Y > 1,
        X > 1,
        NX is X - 1,
        NY is Y - 1.
```

Vamos agora precisar de um predicado mais de alto nível que esconda a estratura de dados das casas e que indique a relação de vizinhança entre as casas.

```
vizinha(C, CV):-
        norte(C, CV).
vizinha(C, CV):-
        sul(C, CV).
vizinha(C, CV):-
        leste(C, CV).
vizinha(C, CV):-
        oeste(C, CV).
```

```
vizinha(C, CV):-
        nordeste(C, CV).
vizinha(C, CV):-
        nordoeste(C, CV).
vizinha(C, CV):-
        sudeste(C, CV).
vizinha(C, CV):-
        sudoeste(C, CV).
```

Podemos agora definir um predicado que nos indique se uma determinada casa do tabuleiro é válida para deslocar a branca. Para isso tem de ser vizinha da casa onde está actualmente a peça branca e não pode estar ocupada com peças pretas.

```
% Uma casa é válida para deslocar a branca
valida(C) :-
        branca(B),
        vizinha(B,C),
        \+ preta(C).
```

Podemos agora fazer o predicado que pede ao próximo jogador para escolher uma jogada válida. Temos de ter um ciclo que faça ler do teclado até que o que foi escrito ser válido. Vamos ler elementos do tipo: X-Y.

```
% repete a leitura até ser jogada válida
le(C) :-
        repeat,
        proximo(J),
        write(J),write(': '),
        read(X),
        mapeia(X,C),
        valida(C).
```

O jogador descreve as células do tabuleiro com números para as linhas e letras para as colunas. Precisamos de mapear só as colunas.

```
% mapeia os X's das casas de letras para números
mapeia(X-Y,(NX,Y)) :-
        map(X,NX).

% correspondência entre letras e números
map(a, 1).
map(b, 2).
map(c, 3).
map(d, 4).
map(e, 5).
map(f, 6).
map(g, 7).
map(h, 8).
map(i, 9).
map(j, 10).
```

Temos também de verificar se o jogo já chegou ao fim! Gostaremos também de saber quem ganhou.

```
% fim se a branca é um dos objectivos
final(Vencedor) :-
        objectivo(Vencedor, B),
        branca(B).

% Fim se jogador cercado. Cercado quer dizer que não há casas válidas para jogar.
% Quem ganha é o adversário de quem joga e que está imobilizado.
final(Vencedor) :-
        \+ valida(_),
        proximo(J),
        adversario(J,Vencedor).
```

Quando se faz uma jogada, a branca desloca-se e a uma nova peça preta é colocada. Temos de retirar e adicionar novos factos. Para isso usaremos os predicados built-in assert/1 e retract/1.

```
% A casa com branca passa a casa com preta e a nova casa fica com a branca
move_branca(NB) :-
        retract(branca(B)),
        assert(preta(B)),
        assert(branca(NB)).
```

Vamos agora fazer o predicado principal que alterna entre as jogadas dos jogadores até que o jogo acabe. Mostra-se o jogo antes das jogadas.

```
% o jogo acabou e mostra o vencedor
jogo :-
        final(V),
        show,
        write('Fim! O vencedor é '),write(V).

% O jogo ainda não acabou
jogo :-
        show,
        proximo(J),
        le(C),
        move_branca(C),
        alterna_jogador,
        jogo.
```

O predicado show ainda não foi feito e é mais elaborado. Fica para o fim. Começamos por mudar de linha e mostrar todas as linhas, terminando imprimindo a fila de letras, uma por coluna

```
show :-
        nl,
        dim(Dim),
        mostra_linhas(Dim),
        tab(34),mostra_colunas(1,Dim).
```

O processo de escrita das linhas começa na linha igual à dimensão do tabuleiro e acaba na

linha 1. Começamos sempre por escrever o número da linha e a seguir o cursor vai de 1 até Dim, imprimindo vazio ou ocupada com preta ou branca. Este predicado é recursivo e termina quando quisermos imprimir a linha 0 (que é menor do que 1).

```
% base da recursão, para quando a linha for menor do que 0.
mostra_linhas(C) :-
        C < 1, nl, !.

% predicado recursivo, imprime uma linha e continua a mostrar as linhas seguintes
mostra_linhas(Y) :-
        mostra1linha(Y),
        NY is Y - 1,
        mostra_linhas(NY).
```

Vamos mostrar uma linha Y. Começamos com um conjunto de espaços, escrevemos o identificador da linha, o seu número e depois de mais uns espaços vamos imprimir todas as células dessa linha do tabuleiro.

```
% Mostra uma linha
mostra1linha(Y) :-
        tab(30),
        write(Y),
        tab(2),
        dim(Dim),
        mostra_linha(Y,1,Dim).
```

O predicado que mostra uma linha do tabuleiro, vai de 1 a Dim, imprimindo todas as células dessa linha.

```
% chegámos ao fia da linha
mostra_linha(_,X,Dim) :-
        X > Dim,!,nl,nl.

% mostra o conteúdo do cursor e faz avançar o cursor (X,Y) para a direita
mostra_linha(Y,X,Dim) :-
        mostra_quadrado(X,Y),
        NX is X+1,
        mostra_linha(Y,NX,Dim).
```

O predicado que mostra uma célula do tabuleiro. Precisamos de conhecer as coordenadas dessa célula e uma célula vazia fica com '.', uma casa com peça branca fica com '0' e com uma peça preta será '*'.

```
% Mostra o conteudo de uma casa com peca branca
mostra_quadrado(X,Y) :-
        branca((X,Y)),
        write(' 0 '),!.

% Mostra o conteudo de uma casa com peca preta
mostra_quadrado(X,Y) :-
        preta((X,Y)),
        write(' * '),!.
```

% Mostra o conteudo de uma casa sem pecas
```
mostra_quadrado(_,_) :-
        write(' . ').
```

Temos de mostrar a fila de letras em baixo para guiar o utilizador na identificação das casas jogadas. Para isso temos de imprimir as letras.

% mostra as letras em baixo
```
mostra_colunas(X,Lim) :-
        X > Lim,!.
mostra_colunas(X,Lim) :-
        map(MX,X),
        write(MX),
        write('  '),
        NX is X + 1,
        mostra_colunas(NX,Lim).
```