**3 Lógica de Jogo**

Irá ser descrita nesta secção a nossa implementação do jogo em Prolog, falando sobre diferentes tópicos desde a visualização do estado de jogo até ao cálculo e geração de jogadas do computador.

O predicado principal do jogo é denominado **play**. Este inicializa o estado do jogo e começa-o (**startGame**), e depois contém o ciclo principal do jogo. Este é formado por 3 tarefas principais: **getState**, que extrai o estado de jogo atual, **changeState**, que gera uma jogada e modifica o estado de jogo, e **saveState**, que tendo em conta essa mudança guarda o estado de jogo modificado, para a próxima iteração do ciclo. Os predicados built-in **repeat** e **once** são utilizados de modo a criar um ciclo, sendo que cada iteração caracteriza um turno/jogada.

O predicado **game\_over** é também chamado em cada iteração, sendo verdadeiro apenas se a condição de terminação do jogo foi alcançada; se este for o caso, é executada a saída do ciclo de jogo, e é chamado o predicado **showWinnerAndReset**, que mostra o vencedor do jogo (ou se houve empate), e que restabelece os valores iniciais do estado de jogo, para a próxima vez queo predicado **play** for executado.

**3.3 Lista de Jogadas Válidas (valid\_moves)**

O predicado em questão recebe uma variável denominada **Player**, que pode ser ‘A’ ou ‘B’, representando o jogador que deve jogar a seguir; uma variável chamada **Board**, contém o estado atual do jogo (tabuleiro); e retorna **ListOfValidMoves**, uma lista de estruturas que representam um dado movimento que o jogador poderá efetuar (são tuplos do género OldLine-OldColumn-NewLine-NewColumn, sendo que as variáveis Old referem-se ao sítio onde a peça se encontrava, e as variáveis New referem-se à nova posição que a peça irá ter). Este predicado é utilizado na lógica das jogadas do computador, de modo a gerar todas as jogadas possíveis de modo a escolher a mais vantajosa, e também na lógica das jogadas do humano, de modo a saber se há alguma jogada válida que este poderá fazer (sendo que, se não existir, o seu turno é passado à frente).

O predicado utiliza o predicado built-in **findall**, que chama um predicado feito por nós denominado **findMove**. Este último retorna uma possível jogada que o Player poderá fazer, sendo que se o utilizador pedir outra solução (com ponto e vírgula), o predicado irá gerar outra jogada possível (se existir). Desta maneira, utilizando findall é possível gerar todas as jogadas válidas possíveis.

Relativamente ao funcionamento do predicado **findMove**, este, tendo em conta o jogador atual a jogar, tenta encontrar uma peça do mesmo que esteja no tabuleiro, gerando depois um possível movimento para essa peça (por exemplo, 1 casa para a esquerda). De seguida, verifica-se a validade dessa jogada, chamando o predicado **checkValidMove**, de modo ao predicado apenas ser verdadeiro se o movimento for válido, tendo em conta o conteúdo do tabuleiro atual. Com a estratégia adotada de gerar uma solução e de seguida verificar a validade da mesma, tira-se partido do mecanismo de retrocesso do Prolog, uma vez que, se o movimento não for válido, o retrocesso faz com que se escolha outro movimento e/ou outra peça do tabuleiro para mover (se for possível). O **once** é utilizado quando se chama **checkValidMove** de modo a evitar retrocesso dentro deste predicado.

**3.4 Execução de jogadas (move)**

O predicado **move** recebe: **Player**, que indica o jogador que está a fazer o movimento; a posição da peça que este pretende mover, através dos argumentos **OldLine** e **OldColumn**; a posição para a qual se pretende mover a peça, com **NewLine** e **NewColumn**; e **Board**, representando o tabuleiro atual. O predicado valida o movimento proposto pelo jogador, e se este for válido então o predicado modifica o tabuleiro, executando o movimento, sendo que o tabuleiro resultante é retornado através do argumento **NewBoard**.

O predicado começa por saber qual é o tipo de peça que corresponde ao jogador, através de **getMicrobeType.** De seguida, chama **checkValidMove** de modo a verificar se o movimento proposto pelo jogador é valido; este predicado verifica se há uma peça do jogador na posição inicial, se a posição final se encontra livre, e se o movimento entre as duas posições é válido.

Se o movimento for válido, é adicionada uma nova peça na posição final (**playMicrobe**), sendo que a peça original é retirada se o movimento não for adjacente, e mantida caso seja (**handleIsAdjacent**). Por fim, chama um predicado que “contamina” todas as peças do oponente que são adjacentes à nova peça, que passam a ser peças do jogador que fez o movimento (**contaminateAdjacent**).

**3.5 Final do jogo (game\_over)**

Este predicado recebe o estado atual do jogo, através de **Board**, verifica se o jogo acabou e, se tal ocorreu, identifica qual o jogador que venceu o jogo, retornando-o através de **Winner**.

Como descrito anteriormente, o jogo termina se apenas existem peças de um tipo em campo, ou se todas as posições do campo estão preenchidas, sendo que neste caso o vencedor é quem tem o maior número de peças no tabuleiro.

O predicado começa por percorrer o tabuleiro, verificando se existem peças do jogador B (se não houver, o vencedor é A). Verifica depois se existem peças do jogador A (se não houver, o vencedor é B). Por fim, verifica se existem espaços livres no tabuleiro. Caso esta última condição não seja verdadeira, o predicado compara o número de pontos atual de cada jogador, sendo que o vencedor é quem tem a pontuação mais alta (uma vez que o número de pontos de um jogador é igual ao número de peças que este tem em campo).

**3.6 Avaliação do tabuleiro (value)**

Este predicado avalia o estado de jogo que lhe é passado como argumento, através de **Board**, retornando em **Value** um valor inteiro que caracteriza o valor desse estado, sendo que quanto maior for esse valor, mais “vantajoso” é esse estado relativamente ao **Player** passado como argumento.

Optamos por desenvolver e integrar 2 níveis de dificuldade do “AI” do nosso jogo: easy (**nível 1**) e medium (**nível 2**). O predicado **value** apenas é chamado quando um computador se encontra no nível de dificuldade 2 (no nível 1 é gerado um value aleatório; irá ser visto mais a frente).

A função value retorna a diferença entre o número de peças do utilizador e o número de peças do oponente. É uma estratégia gananciosa que faz com que um computador a jogar no nível 2 escolha sempre a jogada que é mais vantajosa, nessa altura, para ele, uma vez que tendo uma lista de jogadas válidas, a jogada escolhida é a que adiciona mais peças do jogador ao tabuleiro e/ou a que retira mais peças do oponente.

**3.7 Jogada do Computador (choose\_move)**

Este predicado apenas é chamado no turno de um computador, sendo que não é utilizado quando se trata de um humano a jogar. Este gera todos os possíveis movimentos/jogadas que o computador pode fazer, através de **valid\_moves**, e através de **chooseBestMove** avalia cada jogada, e retorna aquela que é considerada a melhor.

A maneira como as jogadas são avaliadas varia com o nível de dificuldade do computador que chama o predicado (passado em **Level**).

Para o nível 1, e tal como foi dito anteriormente, a cada possível jogada irá ser atribuída um value aleatório, de 0 a 99 (números arbitrários, podiam ser outros), fazendo com que a jogada escolhida seja completamente aleatória.

Para o nível 2, cada jogada é “simulada”, sendo chamado o predicado **move** para construir o Board resultante da aplicação dessa jogada. Esse Board é então avaliado recorrendo à função **value** descrita anteriormente. Deste modo, a jogada retornada é a que está associada ao maior valor retornado por esse função **value**.

Se o jogador que invoca o predicado não tiver movimentos válidos que possa fazer (**valid\_moves** retorna uma lista vazia), então o movimento retornado é **movement(0, 0, 0, 0)**, o que indica a outro setor do código que o jogador não tem jogadas válidas e que o seu turno deverá ser passado à frente.

O predicado recebe como argumentos o **Level,** o **Player** que está a jogar, e o **Board** atual. Retorna o melhor movimento/jogada, em **movement(OldLine, OldColumn, NewLine, NewColumn).**