
## Inteligência Artificial

## Universidade de Évora

## Curso de Engenharia Informática

# Projeto: Jogo “Ouri”

**Trabalho realizado por:**

**Mafalda Rosa nº40021**

**Miguel Carvalho nº41136**

- Este predicado divide uma lista com base no índice inicial e final dado

```
slice(L,Start,End,R) :-

    once(slice_helper(L,Start,End,R)).
```

- Função auxiliar que lida com o caso em que todos os elementos do índice inicial para o final forem visitados

```
slice_helper(_,0,-1,X) :-
    X = [].
```

- Função auxiliar que lida com o caso em que o final da lista é alcançado, encerrando a recursão

```
slice_helper([],_,_,X) :-
    X = [].
```

- Função auxiliar que lida com o caso em que o primeiro elemento a ser adicionado à lista de resultados (índice inicial) é atingido

```
slice_helper([H|T],0,End,R) :-
    NewEnd is End - 1,
    once(slice_helper(T,0,NewEnd,Z)),
    append([H],Z,R).
```

- Função auxiliar que lida com o caso em que o índice inicial ainda não foi atingido

```
slice_helper([_|T],Start,End,R) :-
    NewStart is Start - 1,
    NewEnd is End - 1,
    once(slice_helper(T,NewStart,NewEnd,R)).
```

- Configura a placa para a curva

```
create_board(Pit,Board,B) :-
    slice(Board,0,Pit-1,First),
    slice(Board,Pit,11,Second),
    append(Second,First,B).
```

- Função auxiliar para distribuir as sementes ao redor do tabuleiro

```
distribute_seeds(S,[],NB,R) :-
    distribute_seeds(S,NB,[],R).


distribute_seeds(0,B,NB,R) :-
    append([0|NB],B,R).


distribute_seeds(S,[H|T],NB,R) :-
    E is H+1,
    append(NB,[E],Board),
    NS is S - 1,
    distribute_seeds(NS,T,Board,R).
```

- Distribui as sementes ao redor do tabuleiro

```
distribute([Seeds|Board],R) :-
    once(distribute_seeds(Seeds,Board,[],R)).
```

- Dado um buraco a ser retirado, retorna o tabuleiro após o turno

```
possible_turn(Pit,Board,R) :-
    Pit =< 5,
    create_board(Pit,Board,Arrangement),
    distribute(Arrangement,SB),
    slice(SB,0,11-Pit,Back),
    slice(SB,12-Pit,11,Front),
    append(Front,Back,R).
```

- Conta as ocorrências de um determinado elemento em uma lista

```
count_elements([],_,0).
count_elements([X|T],X,Y) :-
    !,
    count_elements(T,X,Z),
    Y is 1+Z.
count_elements([_|T],X,Z) :-
    count_elements(T,X,Z).
```

- Dado um tabuleiro calcula quantos pontos devem ser marcados no lado do oponente

```
total_score(Board,S) :-
    slice(Board,6,11,Opponent),
    count_elements(Opponent,2,NumTwos),
    count_elements(Opponent,3,NumThrees),
    A is NumTwos * 2,
    B is NumThrees * 3,
    S is A + B.
```

- Cria uma lista das sementes obtidas na colheita de cada poço possível

```
score_list(Board,FL) :-
    possible_turn(0,Board,Pit1),
    total_score(Pit1,Pit1Score),
    append([Pit1Score],[],L1),
    possible_turn(1,Board,Pit2),
    total_score(Pit2,Pit2Score),
    append(L1,[Pit2Score],L2),
    possible_turn(2,Board,Pit3),
    total_score(Pit3,Pit3Score),
    append(L2,[Pit3Score],L3),
    possible_turn(3,Board,Pit4),
    total_score(Pit4,Pit4Score),
```

```
append(L3,[Pit4Score],L4),
possible_turn(4,Board,Pit5),
total_score(Pit5,Pit5Score),
append(L4,[Pit5Score],L5),
possible_turn(5,Board,Pit6),
total_score(Pit6,Pit6Score),
append(L5,[Pit6Score],FL).
```

- Verdadeiro se o oponente tiver apenas zeros ao seu lado; nesse caso, o jogador deve se mover de uma maneira que dê sementes ao oponente

```
zeros_only(Board) :-
    slice(Board,6,11,OpponentSide),
    list_to_set(OpponentSide,[0]).
```

- Calcula o número de sementes que o oponente receberia após um turno

```
num_seeds(Board,S) :-
    slice(Board,6,11,Opponent),
    sum_list(Opponent,S).
```

- Dá o oponente se ele não tiver nenhum

```
create_turn(Board,FL) :-
    possible_turn(0,Board,Pit1),
    num_seeds(Pit1,Pit1Score),
    append([Pit1Score],[],L1),
    possible_turn(1,Board,Pit2),
    num_seeds(Pit2,Pit2Score),
    append(L1,[Pit2Score],L2),
    possible_turn(2,Board,Pit3),
    num_seeds(Pit3,Pit3Score),
    append(L2,[Pit3Score],L3),
    possible_turn(3,Board,Pit4),
```

```
num_seeds(Pit4,Pit4Score),
append(L3,[Pit4Score],L4),
possible_turn(4,Board,Pit5),
num_seeds(Pit5,Pit5Score),
append(L4,[Pit5Score],L5),
possible_turn(5,Board,Pit6),
num_seeds(Pit6,Pit6Score),
append(L5,[Pit6Score],FL).
```

- Encontra a jogada que resulta no maior número de sementes conquistadas pelo jogador

```
best_move_(Board,M) :-
  zeros_only(Board),
  create_turn(Board,L),
  max_list(L,MaxScore),
  nth1(M,L,MaxScore),
  nth1(M,Board,Seeds),
  Seeds =\= 0.
best_move_(Board,M) :-
  score_list(Board,L),
  max_list(L,MaxScore),
  nth1(M,L,MaxScore),
  nth1(M,Board,Seeds),
  Seeds =\= 0.
```

- Primeiro algoritmo que determina a melhor jogada com base em quantos pontos em potencial podem ser pontuados

```
best_move_1(Board,M) :-
  once(best_move_(Board,M)).
```

## ➔ CPU vs Human

- Exibe o tabuleiro de jogo na tela (interface)

```
display_board(PlayerScore,CPUScore,Board) :-
   slice(Board,0,5,PlayerPits),
   slice(Board,6,11,CPUPits),
   reverse(CPUPits,PitsFromAbove),
   append([[CPUScore]],PitsFromAbove,CPUSide),
   append(PlayerPits,[[PlayerScore]],PlayerSide),
   write(CPUSide),nl,
   write(PlayerSide),nl.
```

- Dado dois tabuleiros, retorna uma nova lista de cavidades que foram alteradas durante a vez

```
difference([],[],Change,R) :-
   R = Change.
difference([H1|T1],[H2|T2],Change,R) :-
   H1 =:= H2,
   difference(T1,T2,Change,R).
difference([H1|T1],[H2|T2],Change,R) :-
   H1 =\= H2,
   append(Change,[H2],NewChange),
   difference(T1,T2,NewChange,R).
```

- Quando os pontos são marcados, substitua a cova pelo valor zero na lista

```
replace([],[],Update,R) :-
    R = Update.
replace([H1|T1],[2|T2],Update,R) :-
    H1 =\= 2,
    difference([H1|T1],[2|T2],[],Diff),
    (   last(Diff,2) ; last(Diff,3)),
    append(Update,[0],NewUpdate),
    replace(T1,T2,NewUpdate,R).
replace([H1|T1],[3|T2],Update,R) :-
    H1 =\= 3,
    difference([H1|T1],[3|T2],[],Diff),
    (   last(Diff,2) ; last(Diff,3)),
    append(Update,[0],NewUpdate),
    replace(T1,T2,NewUpdate,R).
replace([_|T1],[H2|T2],Update,R) :-
    append(Update,[H2],NewUpdate),
    replace(T1,T2,NewUpdate,R).
clear_pits(Old,New,Update) :-
    slice(Old,6,11,OldOpponent),
    slice(New,6,11,NewOpponent),
    slice(New,0,5,NewPlayerSide),
    once(replace(OldOpponent,NewOpponent,[],R)),
    append(NewPlayerSide,R,Update).
```

- Dado um tabuleiro antes do turno e o tabuleiro após um turno, determina qual deve ser a nova pontuação do jogador

```
get_score(OldBoard,NewBoard,Score,NewScore) :-
   slice(OldBoard,6,11,OldOpponentSide),
   slice(NewBoard,6,11,NewOpponentSide),
   difference(OldOpponentSide,NewOpponentSide,[],R),
   (   last(R,2) ; last(R,3)),
   count_elements(R,2,Twos),
   count_elements(R,3,Threes),
   A is 3 * Threes,
   B is 2 * Twos,
   C is A + B,
   NewScore is Score + C.
get_score(_,_,Score,NewScore) :-
   NewScore = Score.
```

- Vira o tabuleiro para que o outro jogador possa jogar

```
flip_board(Board,Result) :-
   slice(Board,6,11,Front),
   slice(Board,0,5,Back),
   append(Front,Back,Result).
```

## ➔ Player vs. CPU game of ouri

```
play_game(_,CPUScore,_,_) :-
   CPUScore >= 25,
   once(write("CPU Wins!")).
play_game(_,PlayerScore,_,_) :-
   PlayerScore >= 25,
   once(write("Player wins!")).
```

- Jogo onde o programa fica em primeiro

```
play_game(PlayerScore,CPUScore,CurrentBoard,p) :-
   display_board(PlayerScore,CPUScore,CurrentBoard),nl,
   write("CPU Turn: "),nl,
   sleep(2),
   flip_board(CurrentBoard,FlippedBoard),
   best_move_1(FlippedBoard,Move),
   write("CPU Chose Pit "),
   write(Move),nl,nl,
   possible_turn(Move-1,FlippedBoard,UpdatedBoard),
   get_score(FlippedBoard,UpdatedBoard,CPUScore,NewCPUScore),
   clear_pits(FlippedBoard,UpdatedBoard,NextBoard),
   flip_board(NextBoard,NewFlippedBoard),nl,
   write("Player's Turn: "),nl,
   display_board(PlayerScore,NewCPUScore,NewFlippedBoard),nl,
   write("Enter a Pit: "),
   read(Pit),nl,
   possible_turn(Pit-1,NewFlippedBoard,NewUpdatedBoard),
   get_score(NewFlippedBoard,NewUpdatedBoard,PlayerScore,NewPlayerScore),
   clear_pits(NewFlippedBoard,NewUpdatedBoard,FinalBoard),
   play_game(NewPlayerScore,NewCPUScore,FinalBoard,p).
```

- Jogo em que o programa fica em segundo

```
play_game(PlayerScore,CPUScore,CurrentBoard,s) :-
  display_board(PlayerScore,CPUScore,CurrentBoard),nl,
  write("Enter a Pit: "),
  read(Pit),nl,
  possible_turn(Pit-1,CurrentBoard,NewBoard),
  get_score(CurrentBoard,NewBoard,PlayerScore,NewPlayerScore),
  clear_pits(CurrentBoard,NewBoard,UpdatedBoard),
  display_board(NewPlayerScore,CPUScore,UpdatedBoard),nl,
  write("CPU Turn: "),nl,
  sleep(2),
  flip_board(UpdatedBoard,FlippedBoard),
  best_move_1(FlippedBoard,Move),
  possible_turn(Move-1,FlippedBoard,NewUpdatedBoard),
  get_score(FlippedBoard,NewUpdatedBoard,CPUScore,NewCPUScore),
  clear_pits(FlippedBoard,NewUpdatedBoard,NextBoard),
  flip_board(NextBoard,FinalBoard),nl,
  write("CPU Chose Pit "),
  write(Move),nl,nl,
  play_game(NewPlayerScore,NewCPUScore,FinalBoard,s).
```

- Este é o principal predicado usado para iniciar um jogo em que o programa se move primeiro

```
start_game(-p) :-
  nl,
  write("Welcome to Ouri! The CPU Goes First."),nl,nl,
  play_game(0,0,[4,4,4,4,4,4,4,4,4,4,4,4],p).
```

- Usado para iniciar um jogo em que o jogador se move primeiro

```
start_game(-s) :-
   nl,
   write("Welcome to Ouri! The Player Goes First."),nl,nl,
   play_game(0,0,[4,4,4,4,4,4,4,4,4,4,4,4],s).
```

## ➔ 2º Algoritmo

Este algoritmo determina de quais cavidades ele pode extrair e seleciona uma aleatoriamente.

- Determina quais covas têm sementes (não zero)

```
get_valid_moves(Board,L):-
   slice(Board,0,5,Player),
   delete(Player,0,L).
```

- Seleciona uma cova aleatória com sementes

```
best_move_2(Board,M) :-
   get_valid_moves(Board,L),
   random_member(Seeds,L),
   slice(Board,0,5,Player),
   nth1(M,Player,Seeds).
```

## ➔ 6 torneios de jogos entre o algoritmo 1 e o algoritmo 2

p = CPU 1 vai em primeiro

s = CPU 2 vai em primeiro

- Em uma situação em que o jogo se torna um loop infinito e as cpus estão empatadas, o resultado do jogo é 0 (empate)

```
play_cpu_game(CPU1Score,CPU2Score,Board,_,Result):-
   sum_list(Board,S),
   S =< 4,
   CPU1Score =:= CPU2Score,
   append([],0,Result).
```

- Em uma situação em que o jogo se torna um loop infinito e a cpu2 tem uma pontuação mais alta, o resultado do jogo é 2 (CPU2 vence)

```
play_cpu_game(CPU1Score,CPU2Score,Board,_,Result):-
    sum_list(Board,S),
    S =< 4,
    CPU2Score > CPU1Score,
    append([],2,Result).
```

- Em uma situação em que o jogo se torna um loop infinito e a cpu1 tem uma pontuação mais alta, o resultado do jogo é 1 (CPU1 vence)

```
play_cpu_game(CPU1Score,CPU2Score,Board,_,Result):-
    sum_list(Board,S),
    S =< 3,
    CPU1Score > CPU2Score,
    append([],1,Result).
```

- Se a CPU1 chegar aos 25 primeiro, 1 é o resultado

```
play_cpu_game(CPU1Score,_,_,_,Result) :-
    CPU1Score >= 25,
    append([],1,Result).
```

- Se CPU2 chegar a 25 primeiro, 2 é o resultado

```
play_cpu_game(_,CPU2Score,_,_,Result) :-
    CPU2Score >= 25,
    append([],2,Result).
```

- CPU vs CPU game onde CPU1 se move primeiro

```
play_cpu_game(CPU1Score,CPU2Score,CurrentBoard,p,R) :-
  best_move_1(CurrentBoard,Move),
  possible_turn(Move-1,CurrentBoard,UpdatedBoard),
  get_score(CurrentBoard,UpdatedBoard,CPU1Score,NewCPU1Score),
  clear_pits(CurrentBoard,UpdatedBoard,NextBoard),
  flip_board(NextBoard,FlippedBoard),
  best_move_2(FlippedBoard,Move2),
  possible_turn(Move2-1,FlippedBoard,NewUpdatedBoard),
  get_score(FlippedBoard,NewUpdatedBoard,CPU2Score,NewCPU2Score),
  clear_pits(FlippedBoard,NewUpdatedBoard,SemiFinalBoard),
  flip_board(SemiFinalBoard,FinalBoard),
  play_cpu_game(NewCPU1Score,NewCPU2Score,FinalBoard,p,R).
```

- CPU vs CPU game onde CPU2 se move primeiro

```
play_cpu_game(CPU1Score,CPU2Score,CurrentBoard,s,R) :-
  flip_board(CurrentBoard,FlippedBoard),
  best_move_2(FlippedBoard,Move2),
  possible_turn(Move2-1,FlippedBoard,UpdatedBoard),
  get_score(FlippedBoard,UpdatedBoard,CPU2Score,NewCPU2Score),
  clear_pits(FlippedBoard,UpdatedBoard,NextBoard),
  flip_board(NextBoard,NewFlippedBoard),
  best_move_1(NewFlippedBoard,Move),
  possible_turn(Move-1,NewFlippedBoard,NewUpdatedBoard),
  get_score(NewFlippedBoard,NewUpdatedBoard,CPU1Score,NewCPU1Score),
  clear_pits(NewFlippedBoard,NewUpdatedBoard,FinalBoard),
  play_cpu_game(NewCPU1Score,NewCPU2Score,FinalBoard,s,R).
```

- Inicia o jogo CPU vs CPU onde CPU1 se move primeiro

```
start_cpu_game(-p,Result) :-
    once(play_cpu_game(0,0,[4,4,4,4,4,4,4,4,4,4,4,4],p,Result)).
```

Inicia o jogo CPU vs CPU onde CPU2 se move primeiro

```
start_cpu_game(-s,Result) :-
    once(play_cpu_game(0,0,[4,4,4,4,4,4,4,4,4,4,4,4],s,Result)).
```

- Inicia um torneio de 6 jogos CPU vs CPU Ouri, com as duas CPUs alternando em relação a quem se move primeiro. Os resultados são exibidos no final de uma lista em que o índice 0 é o primeiro jogo, o índice 1 é o segundo jogo etc.

```
start_cpu_tournament :-
    start_cpu_game(-p,Game1),
    start_cpu_game(-s,Game2),
    start_cpu_game(-p,Game3),
    start_cpu_game(-s,Game4),
    start_cpu_game(-p,Game5),
    start_cpu_game(-s,Game6),
    write("Results"),nl,
    write([Game1,Game2,Game3,Game4,Game5,Game6]).
```

O jogo Ouri é exibido usando a seguinte interface:

[[0], 4,4,4,4,4,4] [4,4,4,4,4,4, [0]]

O predicado best_move_1 (Quadro, M) pega um tabuleiro de jogo como argumento e retorna M, a melhor jogada para o jogador. O predicado best_move_1 usa um algoritmo que maximiza o número de pontos marcados com a movimento seleciona.

Nota: No predicado best_move_1, o quadro deve ser inserido no formato de uma lista de comprimento 12, onde os 6 primeiros elementos representam as covas do jogador e os últimos 6 elementos representam as covas do adversário.

Exemplo:

[0,1,4,6,2,1,0,0,8,1,4,6]

Traduz para: [[Score], 6,4,1,8,0,0]

[0,1,4,6,2,1, [Pontuação]]

Para jogar Ouri no computador, execute o predicado start_game (-p) se desejar que o computador se mova primeiro e execute o predicado start_game (-s) se quiser que o computador seja o segundo.

Para executar o segundo algoritmo de melhor movimento, execute best_move_2 (Placa, M) substituindo o tabuleiro por um tabuleiro de jogo no formato mencionado acima.

Esse algoritmo faz uma lista de covas que possuem sementes e seleciona aleatoriamente um deles.

Para simular um torneio de 6 jogos entre os dois melhores algoritmos de movimento, execute start_cpu_tournament.

Os resultados serão exibidos em uma lista de comprimento 6, onde o índice da lista representa o jogo e o resultado pode ser um dos seguintes:

1 = CPU1 ganha

2 = CPU2 ganha

0 = Empate