Manual técnico: Projeto 2

Inteligência Artificial - Escola Superior de Tecnologia de Setúbal 2019/2020

Prof. Joaquim Filipe Eng. Filipe Mariano

Projeto realizado por:

- Sara Batista, nº 170221054
- Carolina Castilho, nº 180221071

1. Arquitetura do sistema

O projeto é dividido em três ficheiros: **interact.lisp**, **jogo.lisp** e **algorithm.lisp**.

O ficheiro **interact.lisp** trata da interação com o utilizador. Carrega os outros ficheiros de código, escreve e lê ficheiros.

O ficheiro **jogo.lisp** tem a implementação do código relacionado com o jogo do cavalo: seletores, operadores e todas as funções necessárias para jogar (jogada-humano e jogada-computador).

O ficheiro **algorithm.lisp** tem a implementação do algoritmo Negamax com cortes Alfa-Beta.

No ficheiro **log.dat** é escrito o output do programa, isto é, o número de nós analisados, o número de cortes efetuados (de cada tipo), o tempo gasto em cada jogada e o tabuleiro atual. Mostramos ainda as pontuações de ambos os cavalos à medida que o jogo se desenrola.

interact.lisp

Neste ficheiro é onde o programa é iniciado através da função **iniciar**. Esta chama as funções **menu** e **jogar**.

A função **menu** é responsável por chamar as funções que fazem as perguntas ao utilizador e guardam as respetivas respostas. O utilizador terá de indicar o **modo de jogo** (Humano VS Computador / Computador VS Computador), o **primeiro jogador** e ainda o **tempo limite** para a jogada do computador.

A função **jogar** é responsável por chamar as funções **jogada-humano**, **jogada-computador** ou **jogada-computador-computador**, consoante os parâmetros inseridos pelo utilizador.

jogo.lisp

É neste ficheiro que são definidas todas as funções relacionadas com o problema, neste caso com o jogo do cavalo.

- **Seletores**: de modo a aumentarmos a nomenclatura do LISP, criámos **funções seletoras**, que retornam determinadas características de um nó passado como argumento: o estado (tabuleiro, pontuações do

cavalo branco, pontuações do cavalo preto, jogador) e a profundidade do nó.

- **Operadores**: de acordo com os movimentos possíveis para o cavalo (movimenta-se em "L"), definimos 8 operadores e um operador default.
- **"Operador 0"**: definimos as funções necessárias para posicionar os cavalos branco e preto no tabuleiro no inicio do jogo.
- **Posições no tabuleiro**: definimos funções para obter uma linha do tabuleiro, obter o valor de uma determinada casa do tabuleiro, procurar uma casa com um determinado valor, procurar a casa onde se encontra o cavalo e substituir valores de casas do tabuleiro.
- **Gerar um tabuleiro aleatório**: implementámos as funções necessárias para gerar um tabuleiro aleatório.
- **Funções sucessores**: implementámos as funções para gerar os sucessores do nó, aplicando os operadores.
- **Funções de verificação do estado do jogo**: implementámos ainda funções de verificação do estado do jogo, isto é, verificar se o jogo já terminou (se ambos os jogadores não têm mais sucessores) e determinar o vencedor.
- **Jogadas**: definimos as funções para os modos de jogo (Humano VS Computador e Computador VS Computador), assim como todas as funções auxiliares necessárias para validar os movimentos para ambos os jogadores.
    - Na **jogada-humano** apresentamos as jogadas possíveis para o utilizador, aplicando os operados à casa onde se encontra. Pedimos ao utilizador para indicar a casa para a qual pretende mover o cavalo, validamos este movimento e devolvemos a jogada (tabuleiro e pontuações atualizadas.
    - Na **jogada-computador** aplicamos o algoritmo negamax se modo a selecionar a melhor jogada possível.
- **Funções de avaliação**: implementámos duas funções de avaliação, uma aplicada aos **nós folha** e outra aos **nós pseudo-folha**. Estas funções tiram proveito da diferença pontual entre os dois jogadores.
    - **Nós folha**: só existem 3 valores possíveis: -1 caso tenha perdido, 0 em caso de empate e 1 caso seja o vencedor.
    - **Nós pseudo-folha**: valores no intervalo de **[-1, 1]**. Considerámos o máximo de pontos **5050** (soma de todos os valores de 1 até 100, o que equivale à soma das pontuações de todas as casas do tabuleiro). Nesta função, convertemos o valor da diferença de pontuação dos dois jogadores num valor entre -1 e 1, aplicando uma simples "*regra de 3 simples*". Atenção que um valor mais próximo de **1** significa que está mais perto da vitória, ao invés de um valor próximo de **-1**, que significa estar mais próximo da derrota.
- **Escrita dos dados**: por último, definimos funções de escrita de dados e métricas, num ficheiro e no ecrã.

algorithm.lisp

Neste ficheiro é implementado o algoritmo Negamax, com cortes Alfa-Beta de forma indepedente do domínio de aplicação.

2. Entidades e sua implementação

- **Nó**: Cada nó é constinuído pelo estado e pela profundidade.
- **Estado**: Cada estado é constituído pelo tabuleiro, pontuação do cavalo branco, pontuação do cavalo preto e o jogador.
- **Operador**: Cada operador representa um movimento do cavalo no tabueiro. Ao todo existem 8 operadores, sendo que num dado estado podem haver operadores que não sejam possíveis de ser realizados (por exemplo, se depois do movimento o cavalo calhar numa casa fora do tabuleiro).
- **Peças**: As peças dos cavalos branco e preto são representadas, respetivamente, por um **"-1"** ou **"-2"** no tabuleiro.

3. Algoritmos e sua implementação

Neste programa foi implementado o algoritmo **Negamax** com cortes **Alfa-Beta**.

Negamax

Este algoritmo é uma variante do Minimax, que tira aproveito de max(a,b)=-min(-a,-b). Isto é, o valor da posição do jogador A num dado jogo é a negação do valor da posição do jogador B. Assim sendo, o jogador A procura uma jogada que maximize a negação do valor que resulta da jogada de B.

É aplicado aos jogos de soma nula, de 2 jogadores.

Implementação em LISP:

````
 ````lisp
 (defun negamax-inicial (no d cor)
    (let* ((reset (reset-jogada))
       (negamax (negamax no d cor)))
     (first *jogada*)))


(defun negamax-sucessores (d cor sucessores alfa beta &optional (bestValue most-
negative-fixnum))
 (cond
 ((null sucessores) alfa)
    (t
      (let* ((valorSucessor (- (negamax (first sucessores) d (- cor) (- beta) (-
alfa))))
          (bestSucessor (max bestValue valorSucessor))
          (aux (cond ((and (> bestSucessor (second *jogada*)) (equal (no-
profundidade (first sucessores)) 1)) (setf *jogada* (list (first sucessores)
bestSucessor)))))
          (newAlfa (max alfa bestSucessor)))
     (cond
````
````

```
     ((>= newAlfa beta) bestSucessor)
     (t (negamax-sucessores d cor (rest sucessores) newAlfa beta
bestSucessor)))))))


(defun negamax (no d cor &optional (alfa -1) (beta 1))
    (let* ((sucessores (sucessores no (operadores)))
         (estado (no-estado no)))
     (cond
        ((null sucessores) (* cor (funcao-avaliacao-nos-folha (second estado)
(third estado))))
        ((equal d 0) (* cor (funcao-avaliacao-pseudo-nos-folha (second estado)
(third estado))))
     (t (negamax-sucessores (- d 1) cor sucessores alfa beta)))))
     ````
```

**Detalhes da implementação:**

O algoritmo foi implementado como se cada nó fosse uma função. (COMPLETAR...)

**Variável global:** Uma vez que precisávamos de obter a jogada feita pelo computador e o Negamax devolve um valor, utilizámos uma variável global **"jogada"** para obtermos a jogada correspondente ao valor retornado pelo algoritmo. Desta forma mantivemo-nos fiéis à implementação do algoritmo.

**Tempo**: (COMPLETAR...)

# 5.Opções técnicas:

- Tendo em conta que as nossas funções de avaliação devolvem valores no intervalo [-1, 1], assumimos os valores de Alfa e Beta respetivamente -1 e 1.
- Definimos ainda duas variáveis globais **jogador1** e **jogador2**, com os valores -1 e -2 respetivamente, de modo a aplicar boas práticas de programação.

# 6.Limitações técnicas e ideias para desenvolvimento futuro

Apesar dos conceitos terem sido compreendidos, neste programa não se encontra implementado a procura quiescente, a memoização e a ordenação de nós.
```