Manual Técnico do Projeto Nº2: Época de Recurso

Inteligência Artificial - Escola Superior de Tecnologia de Setúbal 2024/2025
Estudante: **Rodrigo Baptista**, número **202200217**

1. Introdução

Este manual técnico documenta o desenvolvimento e implementação do projeto Adji-boto, um jogo estratégico da família Mancala, no âmbito da disciplina de Inteligência Artificial da Escola Superior de Tecnologia de Setúbal.
O objetivo do projeto é desenvolver um programa em Lisp, utilizando algoritmos de procura (no caso desto projeto, Negamax com cortes alfa-beta) baseados na Teoria de Jogos lecionada.
O manual detalha a estrutura do código, os algoritmos implementados, os tipos abstratos utilizados e as decisões técnicas adotadas. Além disso, inclui uma análise crítica do desempenho do programa, as suas limitações e estatísticas obtidas a partir das partidas jogadas (Humano *vs* Computador e Computador *vs* Computador).

2. Estrutura do Código

O código divide-se em três ficheiros separados, com as respetivas responsabilidades:
• **jogo.lisp**: Carrega os outros ficheiros de código, escreve e lê de ficheiros e trata da interação com o utilizador.
• **puzzle.lisp**: Código relacionado com o problema.
• **algoritmo.lisp**: Implementação do algoritmo de jogo (Negamax com cortes alfa-beta) independente do domínio.

3. Algoritmo Implementado e Funções Auxiliares

O algoritmo implementado foi o Negamax com cortes alfa-beta.

3.1 Algoritmo

1. Verificamos os casos base da recursividade, que é a profundidade limite de procura (no caso desta implementação, é uma constante definida como 10), se o nó atual é um nó terminal, ou seja, é um nó objetivo (no contexto do problema, o nó terminal é um nó cujo estado é o tabuleiro vazio), se o tempo limite de procura foi atingido (um valor, em segundos, entre 1 e 20, inclusive).
2. Inicializamos o valor máximo e geramos os descendentes (sucessores), que são posteriormente ordenados de forma descrescente tendo em conta a sua avaliação.
3. Para cada descendente (sucessor), computamos valor Negamax de forma recursiva, ou seja, para cada criança voltamos a repetir estes passos.
4. São gerados cortes beta caso o valor de alfa seja superior ao valor de beta, e cortes alfa caso seja encontrada uma melhor solução para alfa.

Código:

```
(defun negamax (node depth alpha beta player generator objective evaluation game-
operator &optional (start-time -1) (time-limit -1))
  "Receives a NODE, a max search DEPTH, value of ALPHA, BETA, current PLAYER, the
GENERATOR function for the children, the OBJECTIVE node, an EVALUATION function and
```

```
the GAME-OPERATOR. 34 - 4."
  (if (or (= depth 0) (funcall objective node) (and (not (= time-limit -1)) (elapsed-
sufficient start-time time-limit)))
    (* player (funcall evaluation node)) ; Objective node or final search depth. Human
VS Computer time limit.
    (let
      (
        (max-value -1.0e+9)
        (children (funcall generator node game-operator player)))
      (progn
        (setf children (sort children #'> :key (lambda (child) (* player (funcall
evaluation child)))))
        (dolist (child children)
          (let
            ((score (- (negamax child (- depth 1) (- beta) (- alpha) (- player)
generator objective evaluation game-operator start-time time-limit))))
            (set-nodes-analyzed (+ (get-nodes-analyzed) 1))
            (when (> score max-value) (setf max-value score))
            (when (> score alpha)
              (setf alpha score)
              (set-alpha-cuts (+ (get-alpha-cuts) 1)))
            (when (>= alpha beta)
              (set-beta-cuts (+ (get-beta-cuts) 1))
              (return max-value) ; Prune.
            )))
        max-value))))
```

3.2 Funções de Geração

Funções responsáveis por gerar os sucessores de um dado nó:

```
(defun new-child (node operator index1 index2)
  "Generates the child of a given NODE according to OPERATOR, called at INDEX1 and
INDEX2."
  (when node
    (let
      ((child (funcall operator index1 index2 node))) ; Calls the operator, which
returns a node with the new board and score.
      (if (not (null child))
        child
        nil))))

(defun generate-children (node operator current-player)
  "Generates the children of a given NODE using OPERATOR according to CURRENT-PLAYER."
  (when
    (and (not (null node)))
    (let
      (
        (children '())
        (i1 (if (= current-player 1) 1 0)))
      (loop for i2 from 0 below 6 do ; Board columns.
        (let
          ((child (new-child node operator i1 i2)))
```

```
          (when child (push child children))))
      (nreverse children))))
```

3.3 Função de Verificação do Nó Objetivo

Função para verificar se um nó é um nó terminal:

```
(defun node-solutionp (node)
  "Receives a NODE and checks if it's state is the problem solution."
  (if (or (null node) (null (node-state node)))
    nil
    (compare-state (create-node (board-empty) 0 0 0 nil) node) ; Creates a node with
an empty board to check if the passed board has the same state.
  )
)
```

3.4 Função de Avaliação

Função de avaliação de um nó, em que se o jogador 1 estiver a ganhar a avaliação é positiva e se estiver a perder é negativa:

```
(defun evaluate-node (node)
  "Evaluates the score of a given NODE."
  (- (node-score-p1 node) (node-score-p2 node)))
```

3.5 Operador de Jogo

Operador do jogo, responsável por efetuar as jogadas:

```
(defun game-operator (line-index position-index node)
  "Denotes the play is to be made at line INDEX1 and index INDEX2 of the board in
NODE."
  (if (and (valid-linep line-index) (valid-line-indexp position-index) (not (null
node)))
    (let*
      (
        (total-pieces (cell line-index position-index (node-state node)))
        (holes (distribute-pieces total-pieces line-index position-index)))
      (if (= total-pieces 0)
        nil ; Invalid operation.
        (let
          ((new-node (change-board holes node line-index))) ; The new node, with
updated state & scores.
          (create-node ; Turns the moved value to zero and returns the new node.
            (replace-value line-index position-index (node-state new-node) 0)
            (node-score-p1 new-node)
            (node-score-p2 new-node)
            (+ (node-depth node) 1)
            node))))
    nil))
```

3.6 Lógica da jogada do Computador (com Memoização)

Lógica que trata de ler um movimento do computador com memoização, se a jogada já tiver sido feito anteriormente vai-se buscar o melhor movimento à tabela de transposição, senão corre-se o Negamax:

```
(defun read-computer-move (current-player current-node &optional (computer-time-limit
-1))
  "Reads a computer move according to CURRENT-PLAYER and CURRENT-NODE, receives an
optional COMPUTER-TIME-LIMIT."
  (progn
    (format t "Computer thinking...~%")
    (let*
      (
        (alpha -1.0e+9)
        (beta 1.0e+9)
        (best-move nil)
        (best-score -1.0e+9)
        (line (if (= current-player 1) 1 0))
        (hash-key (list current-player (node-state current-node)))
        (start-time (get-internal-real-time)))
      (progn
        (if (gethash hash-key *hash-table*)
          (progn ; In hash table.
            (setf best-move (gethash hash-key *hash-table*))
            (setq *hash-table-hit-rate* (+ *hash-table-hit-rate* 1)))
          (progn ; Not in hash table.
            (loop for pos from 0 below 6 do
              (let
                ((child (game-operator line pos current-node)))
                (if (not (null child))
                  (let
                    ((score (- (negamax child *search-depth* (- beta) (- alpha) (-
current-player) 'generate-children 'node-solutionp 'evaluate-node 'game-operator
start-time computer-time-limit))))
                    (if (>= score best-score)
                      (progn
                        (setf best-score score)
                        (setf best-move pos)))))))
            (setf (gethash hash-key *hash-table*) best-move) ; Memorize the best move.
            (setq *hash-table-miss-rate* (+ *hash-table-miss-rate* 1))))
        (print-negamax-info)
        (print-time-elapsed start-time)
        (setq *alpha-cuts-total* (+ *alpha-cuts* *alpha-cuts-total*))
        (setq *alpha-cuts* 0)
        (setq *beta-cuts-total* (+ *beta-cuts* *beta-cuts-total*))
        (setq *beta-cuts* 0)
        (setq *nodes-analyzed-total* (+ *nodes-analyzed* *nodes-analyzed-total*))
        (setq *nodes-analyzed* 0)
        best-move))))
```

3.7 Lógica de um Jogo

Como se efetuam os turnos de jogo e qual a lógica de um turno, leitura e escrita para interação com o utilizador:

```
(defun game (first-player mode &optional (computer-time-limit 0))
  "Receives a FIRST-PLAYER, the game MODE and a COMPUTER-TIME-LIMIT. Is responsible
for the game logic (turns / win condition)."
  (let
    (
      (current-player first-player)
      (current-node (create-node (board-initial) 0 0 0 nil)))
    (loop while t do
      (progn
        (print-game-turn)
        (print-board (node-state current-node)) ; Before.

        (cond
          ((and (= current-player 1) (= (line-piece-count 1 (node-state current-node))
0)) (print-cant-move current-player)) ; Player 1 can't move.
          ((and (= current-player -1) (= (line-piece-count 0 (node-state current-
node)) 0)) (print-cant-move current-player)) ; Player 2 can't move.
          (t
            (if (eq mode 'human-vs-computer)
              (cond ; Human VS Computer
                ((= current-player 1) ; Human to play.
                  (let
                    ((human-move (read-human-move (node-state current-node) current-
player))) ; Only valid moves get returned.
                    (progn
                      (setf current-node (game-operator 1 human-move current-node))
                      (print-move current-player human-move))))
                ((= current-player -1) ; Computer to play.
                  (let
                    ((best-move (read-computer-move current-player current-node
computer-time-limit)))
                    (if (not (null best-move))
                      (progn
                        (setf current-node (game-operator 0 best-move current-node))
                        (print-move 0 best-move))))))
              (cond ; Computer VS Computer
                ((= current-player 1)
                  (let
                    ((best-move (read-computer-move current-player current-node
computer-time-limit)))
                    (if (not (null best-move))
                      (progn
                        (setf current-node (game-operator 1 best-move current-node))
                        (print-move 1 best-move)))))
                ((= current-player -1)
                  (let
                    ((best-move (read-computer-move current-player current-node
computer-time-limit)))
                    (if (not (null best-move))
                      (progn
                        (setf current-node (game-operator 0 best-move current-node))
                        (print-move 0 best-move)))))))))))
```

```
        (setq *current-turn* (+ *current-turn* 1))
        (setf current-player (* current-player -1))

        (print-board (node-state current-node)) ; After.
        (print-score (node-score-p1 current-node) (node-score-p2 current-node))

        (if (board-emptyp (node-state current-node))
          (progn
            (print-game-over current-node computer-time-limit)
            (return)))))))
```

4. Tipos Abstratos de Dados

4.1 Estado do Jogo (Tabuleiro) e Jogadores

Uma lista com duas *nested lists*, em que as células representam o número de peças numa dada posição e a linha de índice 0 o jogador 2, a linha de índice 1 o jogador 1 (de forma a que na interface de jogo, o jogador humano fique com o tabuleiro "virado" para si). O jogador 1 é representado com o valor 1, o jogador 2 com o valor -1.

```
(defun board-initial ()
  "Returns a 2x6 board that corresponds initial game state."
  '((8 8 8 8 8 8)
    (8 8 8 8 8 8))
)
```

4.2 Especificação do Nó

O nó utilizado no algoritmo de procura é uma lista com 5 elementos, o primeiro elemento representa o estado do jogo (tabuleiro), o segundo elemento a pontuação do jogador 1, o terceiro elemento a pontuação do jogador 2, o quarto elemento a profundidade do grafo de jogo e o quinto elemento o nó anterior. Não foi necessário implementar uma célula com o valor do jogador atual, pois a memoização foi feita segundo as jogadas (fora do algoritmo Negamax, dentro da leitura de jogadas, como referido no capítulo da estrutura do código).

```
(defun create-node (state score-p1 score-p2 depth previous)
  (list state score-p1 score-p2 depth previous)
)
```

5. Limitações e Opções Técnicas

Não foi implementada a procura quiescente e o método de memoização utilizado necessita de armazenar as jogadas fora do algoritmo negamax e na memória interna do computador, o que pode ser considerada uma implementação "incorreta", a lógica é chamar o negamax apenas quando uma jogada não se encontra na tabela de transposição. Se a jogada se encontrar na tabela de transposição (chave sendo o estado e o jogador atual), então retornamos a posição da melhor jogada. No entanto, isto leva a que o primeiro jogo a ser jogado tenha *hit rate* sempre de 0 pois não se repetem jogadas. De resto, não existem limitações para o programa.

6. Análise Estatística

Nota: Na mesma pasta deste relatório estão os ficheiros .txt *raw* com os dados de execução do programa. Em jogos Computador VS Computador o jogador que inicia o jogo é o jogador 1.

6.1 Computador VS Computador

Indicadores de desempenho por unidade de tempo pensado pelo computador (duração da pesquisa do Negamax).

Indicadores de Desempenho	1 Segundo	2 Segundos	5 Segundos	20 Segundos
Pontuação Jogador 1	58	63	66	48
Pontuação Jogador 2	38	33	30	48
Vencedor	Jogador 1 por 20 pontos.	Jogador 1 por 30 pontos.	Jogador 1 por 36 pontos.	Empate
Jogadas (Profundidade) e Turnos	114 Jogadas, 134 Turnos	136 Jogadas, 158 Turnos	146 Jogadas, 160 Turnos	151 Jogadas, 168 Turnos
Hit Rate da Hash Table	0 / 114	0 / 136	0 / 146	0 / 151
Cortes Alfa	76940	244189	451760	1492245
Cortes Beta	67093	219295	411294	1376446
Número de Nós Analisados	223431	672941	1326956	4200450
Profundidade Máxima de Pesquisa	10	10	10	10

Podemos ver que o número de cortes gerado é muito alto e o número de nós analisados também, o que significa que existem muitos caminhos que acabam por não ser percorridos na árvore de pesquisa e decisão, este número poderia ser diminuido com a utilização da memoização dentro do algoritmo Negamax, sendo assim mais eficiente, no entanto, o algoritmo funciona de forma correta tendo em conta a profundidade de pesquisa.

6.2 Humano VS Computador

O primeiro a jogar foi o jogador Humano.
A profundidade de pesquisa foi 10 e o tempo de pensamento do computador foi 1 segundo.
Pontuação do Humano: 27 Pontos.
Pontuação do Computador: 69 Pontos.
Foram feitas um total de 105 jogadas (profundidade do grafo de jogo) em 122 turnos.
Hit Rate da Hash Table: 0/44.

Número de cortes Alfa: 55657 cortes.
Número de cortes Beta: 49136 cortes.
Número de Nós Analisados: 150725 nós.

7. Conclusões

É possível ver que o algoritmo Negamax está a funcionar corretamente segundo as especificações do mesmo, tais como tempo de pensamento (tempo de pesquisa) e profundidade máxima de pesquisa (que no caso deste projeto, foi 10). No entanto, os valores de cortes Alfa e cortes Beta encontram-se bastante inflacionados.
Com isto dito, o projeto encontra-se completamente funcional sem nenhuns erros encontrados e o único requisito que não foi implementado é a procura quiescente.