Manual Técnico

Jogo do Cavalo

Índice

Introdução

Este documento tem a finalidade de fornecer a documentação técnica relacionada com o programa desenvolvido no âmbito da disciplina de Inteligência Artificial, escrito em *Lisp,* do qual o objetivo é desenvolver uma variação do Jogo do Cavalo de 2 jogadores com a utilização do algoritmo MiniMax com cortes Alfa-Beta.

> Este projeto foi produzido e desenvolvido no IDE ***Visual Studio Code*** com a utilização do interpretador de ***Lisp*** *CLisp*. **Tedo sido apenas compilado e testado neste ambiente.**

Os ficheiros de código foram divididos e organizados da seguinte forma:

- **algoritmo.lisp** → Contem a implementação do algoritmo MiniMax com cortes AlfaBeta e todas as suas funções auxiliares.
- **jogo.lisp** → Contem os operadores de jogo e funções auxiliares dos mesmos.
- **interact.lisp** → Contem as funções de ler e escrever em ficheiros e as funções que tratam da interação com o utilizador.

Regras

As regras impostas para as jogadas são:

- Se a casa escolhida tiver um número com dois dígitos diferentes, por exemplo 57, então, em consequência, o número simétrico 75 é apagado do tabuleiro, tornando esta casa inacessível durante o resto do jogo. Ou seja, nenhum cavalo pode terminar outra jogada nessa casa.
- Se um cavalo for colocado numa casa com um número "duplo", por exemplo 66, então qualquer outro número duplo pode ser removido e o jogador deve escolher qual em função da sua estratégia (por *default* remover a de maior valor).
- Depois de um jogador deixar a casa para se movimentar para outra, a casa onde estava fica também inacessível para o jogo, ficando o numero da casa apagado.
- Os cavalos não podem acabar uma jogada numa casa nula ou numa casa ameaçada pelo cavalo do adversário.

- Os jogadores só ganham pontos com as casas visitadas e não com as casas bloqueadas via regra do simétrico ou do numero duplo.
- Caso um dos jogadores não tenha jogadas possíveis a sua vez é passada.

Ficheiro Algoritmo

Neste ficheiro podemos encontrar a estrutura dos nós, a implementação do algoritmo e as funções de manipulação de nós.

- Esta é a estrutura criada para um nó, esta contem a pontuação do jogador 1, a pontuação do jogador 2, qual o jogador a fazer a jogada, a profundidade do nó, a avaliação, o tabuleiro, e o nó pai.

```
(defstruct node
  score-one
  score-two
  turn-player
  depth
  board
  evaluation
  parent
)
```

- Esta é a função responsável por criar um novo, sendo a base de todo o funcionamento. Esta função recebe a pontuação do jogador 1, a pontuação do jogador 2, qual o jogador a jogar (-1 ou -2), o tabuleiro atual e opcionalmente a profundidade que por defeito é 0, a avaliação que por defeito é programaticamente menos infinito e o nó pai que por defeito é nil.

```
(defun create-node (score-one score-two turn-player board &optional (depth
0) (evaluation most-negative-fixnum) (parent nil))
(make-node
  :score-one score-one
  :score-two score-two
  :turn-player turn-player
  :depth depth
  :evaluation evaluation
  :board board
  :parent parent
)
```

) ```

- Esta função serve para identificar o nó da primeira jogada, esta recebe um nó e verifica se tem nó pai se o nó pai for nil quer dizer que este é o estado inicial e retorna nil. Se não for nil ele verifica o pai do nó pai, se for nil quer dizer que o pai é o estado inicial e o nó é a primeira jogada.

```
```

```
(defun node-to-first-move-node (node)
  (cond
    ((null (node-parent node)) nil)
    ((null (node-parent (node-parent node))) node)
    (t (node-to-first-move-node (node-parent node)))
  )
)
```

- Esta função tem como objetivo mudar o tabuleiro de um nó, esta recebe o nó a ser alterado e o novo tabuleiro. O tabuleiro é posteriormente alterado.

```
(defun node-change-board(node new-value)
    (setf (node-board node) new-value)
)
```

- Esta funções de print tem apenas de mostrar para o terminal ou para o documento o node fornecido.

```
(defun node-print (node &optional (stream-to-write-to t))
  (if (node-parent node) (node-print (node-parent node) stream-to-write-to))
  (format stream-to-write-to "~%SCORE: ~5a | ~5a|TURN: ~5a~%~%" (node-score-
one node) (node-score-two node) (node-turn-player node))
  (print-board (node-board node) stream-to-write-to)
)

(defun node-print-singular (node &optional (stream-to-write-to t))
  (format stream-to-write-to "~%SCORE: ~5a | ~5a|TURN: ~5a~%~%" (node-score-
one node) (node-score-two node) (node-turn-player node))
  (print-board (node-board node) stream-to-write-to)
)
```

- A estrutura do nó solução tem como objetivo guardar o nó em si, os cortes Alfa, cortes Beta, tempo e nós analisados durante o processo de procura do *minimax*.

```
(defstruct solution-node
  optimal-move
  alpha-cuts
  beta-cuts
  time-elapsed
    analized-nodes
)
```

- Esta função aplica uma outra função de expansão de um no recebida por argumento. Gera os seus filho e cria outros nós a partir dos filhos.

```
(defun apply-expanding-function (expanding-function node player)
  (let* (
      (children (funcall expanding-function (node-board node) player))
    )
    (when children
      (remove nil
        (mapcar (lambda (movement-result)
          (let* (
              (current-board (movement-result-board movement-result))
              (val (car (last (find-double-digit-numbers current-board))))
              (new-board
                (if val
                  (remove-double-number current-board (cell current-board
(car val) (cadr val)))
                  current-board
                )
              )
            )
            (when new-board
              (let* (
                  (score-one (if (= (node-turn-player node) *player-one*)
(movement-result-score movement-result) 0))
                  (score-two (if (= (node-turn-player node) *player-two*)
(movement-result-score movement-result) 0))
                  (current-node (create-node
                    (+ (node-score-one node) score-one)
                    (+ (node-score-two node) score-two)
                    (oposite-player (node-turn-player node))
                    new-board
                    (1+ (node-depth node))
                    most-negative-fixnum
                    node
                  ))
                )
                current-node
              )
            )
          )
        )children
...)
```

- A função principal do algoritmo *minimax* com cortes alfa beta pretende disponibilizar no final do tempo requisitado o melhor nó encontrado.
    - Primeiramente são criadas as variáveis necessárias para guardar as informações referentes ao algoritmo assim como a tabela de memoização.
    - É criada uma função "alfabeta-helper" que efetua todo o processo e é chamada apenas no final quando é efetivamente criado o nó solução guardado na variável *result-final* com o resto da informação relevante.

- A função “alfabeta-helper” começa por guardar o valor da chave e do cache para o processo de memoização.
- A cada iteracao é procurada na tabela de memoização o nó que estamos a tentar gerar, se for encontrada poupa o processamento, caso contrario regista esse novo nó na tabela.
- A função tem o objetivo de organizar os filhos e gerir a sua expansão, aplicar os cortes e regista-los até encontrar o caminho mais eficiente encontrado.

```
(defun alfabeta (node expanding-function max-depth max-time &optional
(maximizing-player (node-turn-player node)))
  (let (
      (start-time (current-time-milliseconds))
      (alfa-cuts 0)
      (beta-cuts 0)
      (result-final nil)
      (analized-nodes 0)
      (memo-table (make-hash-table :test 'equalp)) ;memoization table
    )
    (labels
      ((alfabeta-helper (current-node current-depth &optional (alfa most-
negative-fixnum) (beta most-positive-fixnum))
        (let* (
            (key (list current-node current-depth)) ; for memoization
            (cached-value (gethash key memo-table)) ; for memoization
          )
          (if cached-value ; for memoization
            cached-value ; for memoization
            (let ((r ; for memoization
                (progn (setq analized-nodes (+ 1 analized-nodes))
                  (cond
                    ((or (>= current-depth max-depth) (time-limit-exceeded-p
start-time max-time))
                      (evaluate-node current-node maximizing-player)
                    )
                    (t
                      (let* (
                          (maximizing (= (node-turn-player current-node)
maximizing-player))
                          (symb (if maximizing #'>= #'<=))
                          (children (apply-expanding-function expanding-
function current-node))
                          (ordered-children (order-queue-of-nodes children
symb maximizing-player))
                          (initial-value (if maximizing most-negative-fixnum
most-positive-fixnum))
                          (result (create-node 0 0 0 nil current-depth
initial-value nil))
                        )
                        (if ordered-children
                          (loop
                            for child in ordered-children
                            do
                              (let* (
```

```
                                    (ab-node (alfabeta-helper child (1+
current-depth) alfa beta))
                                    (v (pick-best-node ab-node result symb
maximizing-player))
                                    (v-eval (node-evaluation v))
                                  )
                                  (setq result v)
                                  (if maximizing
                                    (progn
                                      (setq alfa (if (funcall symb alfa v-
eval) alfa v-eval))
                                      (when (>= alfa beta) (progn (setf alfa-
cuts (+ 1 alfa-cuts)) (return-from alfabeta-helper v)))
                                    )
                                    (progn
                                      (setq beta (if (funcall symb beta v-
eval) beta v-eval))
                                      (when (>= alfa beta) (progn (setf beta-
cuts (+ 1 beta-cuts)) (return-from alfabeta-helper v)))
                                    )
                                  )
                                )
                              finally (return result)
                            )
                            (evaluate-node current-node maximizing-player)
                          )
                        )
                      )
                    )
                  )
                ))
                (setf (gethash key memo-table) r) r ;memoization
              )
            )
          )
        ))
        (setq result-final
          (create-solution-node
              (alfabeta-helper node 0 most-negative-fixnum most-positive-
fixnum)
            alfa-cuts beta-cuts (time-elapsed start-time) analized-nodes
          )
        ))
    result-final
)
```

- Estas são as funções criadas para fazer a gestão do tempo da jogada do computador.

```
(defun time-limit-exceeded-p (start-time max-time)
  (let ((leeway (/ max-time 10)))
    (>= (+ leeway (time-elapsed start-time)) max-time)
  )
```

```
)

(defun current-time-milliseconds ()
  (* 1000 (get-universal-time))
)

(defun time-elapsed (start-time)
  (- (current-time-milliseconds) start-time)
)
```

Ficheiro Jogo

No ficheiro **Jogo** teremos todo o código relacionado com as operações de jogo, aplicação das regras e gestão de tabuleiro.

- Esta é uma estrutura de dados criada para cada movimento com a pontuação da jogada, o jogador que fez a jogada e o tabuleiro atual. Temos também a função para criar um novo movimento que recebe a pontuação, o jogador que a realizou e o tabuleiro.

```
(defstruct movement-result
  score
  player
  board
)

(defun create-movement-result (score player board)
  (make-movement-result
    :score score
    :player player
    :board board
  )
)
```

- Esta é a função desenvolvida com o propósito de mostrar na consola uma jogada, esta recebe um *movement-result e* opcionalmente uma *stream* de modo a guardar os dados num ficheiro. Caso não seja passada nenhuma *stream* esta por defeito toma o valor lógico de *True* de forma a dar print na consola.

```
(defun movement-result-print (movement-result &optional (stream-to-write-to
T))
  (progn
    (format stream-to-write-to
"~%_____________________________________________________~%~%")
    (format stream-to-write-to "SCORE: ~15a~t~15aPLAYER: ~a~%~%"
      (movement-result-score movement-result) (code-char 124) (movement-
result-player movement-result))
    (print-board (movement-result-board movement-result))
```

```
    )
  )
```

- A função seguinte tem o propósito de apresentar visualmente um tabuleiro. Esta função tem como argumento o tabuleiro a ser apresentado e tal como previamente observado temos o argumento *stream-to-write-to* que é opcional no caso de se querer guardar os dados num ficheiro.

```
(defun print-board (board &optional (stream-to-write-to t))
"Dá print do tabuleiro de maneira limpa e legivel"
  (cond
    ((not (car board)) (format stream-to-write-to
"_____________________________________________________~%"))
    (T
      (progn
        (if (= 10 (length board))
          (progn
            (format stream-to-write-to
"~5t~4,2a~4,2a~4,2a~4,2a~4,2a~4,2a~4,2a~4,2a~4,2a~%"
                0 1 2 3 4 5 6 7 8 9)
            (format stream-to-write-to
"~5,9t_____________________________________________~%")
          )
        )
        (format stream-to-write-to "~4a|" (- 9 (1- (length board))))
        (print-row (car board) stream-to-write-to)
        (print-board (cdr board) stream-to-write-to)
...)
```

- A função seguinte tem como objetivo apresentar visualmente uma determinada linha do tabuleiro, tal como nas funções de *print* anteriormente apresentadas, esta tem como argumentos a linha e opcionalmente o *stream-to-write-to* que toma o valor lógico de *True* por defeito e a coluna que toma o valor 9 por defeito.

```
(defun print-row (row &optional (stream-to-write-to t) (column 9) )
"Dá print da linha de um tabuleiro de maneira limpa e legivel"
  (cond
    ((> 0 column) (format stream-to-write-to "~%"))
    (t
      (progn
        (format stream-to-write-to "~5a" (car row))
        (print-row (cdr row) stream-to-write-to (1- column) )
...)
```

- Esta função tem como objetivo baralhar os elementos de uma lista e tem como argumento a lista a ser baralhada. O objetivo é atingido pegando num valor aleatoriamente da lista, adicionando-o ao inicio e removendo-o do sitio anterior. Este processo é repetido para todos os elemento da lista.

```
(defun shuffle (board)
"Mistura a lista de posições"
  (cond
        ((null board) '())
    ((car board) (let (
          (n (nth (random (length board)(make-random-state t)) board))
        )
        (append (cons n nil) (shuffle (remove n board)))
      )
    )
    (T '())
  )
)
```

- Esta função tem como objetivo criar um tabuleiro com o valor das casas distribuído aleatoriamente.

```
(defun random-board (&optional (board (shuffle (list-numbers))) (n 10))
"Cria um tabuleiro aleatório"
  (cond
    ((null board) nil)
    (t (cons (subseq board 0 n) (random-board (subseq board n) n)))
  )
)
```

- A função seguinte tem o propósito de criar uma lista ordenada decrescente de “n” exclusive a 0 inclusive.

```
(defun list-numbers (&optional (n 100))
"Cria uma lista de numeros de 0 até 100"
  (cond
    ((= n 1) (list 0))
    (T (append (list (1- n)) (list-numbers (1- n))))
  )
)
```

- Esta função tem como objetivo devolver uma determinada casa do tabuleiro. A função tem como argumentos o tabuleiro em questão, a linha e a coluna da casa.

```
(defun cell (board row column)
"Devolve o atomo que está numa certa coluna, no numero da linha do
tabuleiro"
  (if (or (> row 9) (< row 0)(> column 9)(< column 0))
    nil
    (nth column (nth row board))
  )
)
```

- Esta função serve para encontrar as coordenadas no tabuleiro de um cavalo. A função tem como argumentos o tabuleiro atual, opcionalmente o jogador (cavalo branco ou preto) que por defeito é o jogador 1, a linha e coluna que por defeito são ambos 0. O objetivo é atingido percorrendo o tabuleiro horizontalmente ate encontrar o cavalo desejado, caso o cavalo não seja encontrado é retornado nil.

```
(defun find-value (board &optional (value-to-find *player-one*) (row 0)
(column 0) )
"Devolve a posição atual do cavalo, onde está no tabuleiro (T)"
  (cond
    ((null board) nil)
    ((eq value-to-find (cell board row column)) (list row column))
    ((< 9 row) nil)
    ((< 9 column) (find-value board  value-to-find (1+ row)))
    (t (find-value board value-to-find row (1+ column)))
  )
)
```

- Esta função que ,tem como argumentos uma linha, uma coluna, um tabuleiro e opcionalmente um valor que por defeito é nil, tem como finalidade colocar o valor passado no tabuleiro fornecido nas coordenadas dadas.

```
(defun place (row column board &optional (value nil))
"Coloca um certo valor no tabuleiro na posição fornecida (linha, culuna) em
origem no topo esquerdo"
  (cond
    ((null (car board)) nil)
    ((eq row 0) (cons (place-list column (car board) value) (cdr board)))
    (T (cons (car board) (place (1- row) column (cdr board) value)))
  )
)
```

- Esta função coloca um valor numa lista na posição fornecida.

```
(defun place-list (position row value)
"Coloca um valor na lista (linha) na posição dada"
  (cond
    ((eq position 0) (cons value (cdr row)))
    (T
      (append
        (list (car row))
        (place-list (1- position) (cdr row) value)
...)
```

- Esta função tem como objetivo aplicar a regra do número duplo, mais especificamente verificar se é duplo e remove-lo do tabuleiro.

```
(defun remove-double-number (board val)
"Remove o *val* do tabuleiro"
  (cond
    ((not (numberp val)) nil)
    ((= val (reverse-digits val))
      (let ((position-to-remove (find-value board val)))
        (if position-to-remove
          (place (car position-to-remove) (cadr position-to-remove) board)
          nil
        )
      )
    )
    (t nil)
  )
)
```

> Esta função utiliza o ***reverse-digits*** para transformar o número no seu simétrico de modo a verificar se este é um número duplo, ***find-value*** para verificar e encontrar o valor no tabuleiro e ***place*** para colocar nas coordenadas do ***"val"*** o valor nil.

- A função abaixo serve para aplicar a regra do simétrico, removendo assim, o simétrico do número passado por argumento.

```
(defun remove-reverse-number (board val)
"Remove da tabela o numero contrario ao recebido no parametro e troca por
NIL"
  (let ((pos (find-value board (reverse-digits val))))
    (cond
      ((null pos) board)
      (t (place (car pos) (cadr pos) board))
    )
  )
)
```

> Esta função utiliza o ***reverse-digits*** para transformar o número no seu simétrico , ***find-value*** para obter a posição do valor no tabuleiro e ***place*** para colocar nas coordenadas do ***"val"*** o valor nil.

- Esta função tem como objetivo criar uma lista de sublistas onde as sublistas são as coordenadas dos números duplos.

```
(defun find-double-digit-numbers (board &optional (value 1))
"Encontra números no tabuleiro com o mesmo numero"
  (cond
    ((> value 9) '())
    ((find-value board (* value 11))
      (append
```

```
          (list (find-value board (* value 11)))
          (find-double-digit-numbers board (1+ value))
        )
      )
      (t  (find-double-digit-numbers board (1+ value)))
    )
  )
```

- Esta função tem como objetivo auxiliar outras funções mais abaixo, esta recebe um número de 1 a 8, um tabuleiro, um jogador, e uma função a aplicar.

```
(defun operator (num board player expanding-func)
  (case num
    (1 (funcall expanding-func board  2   -1  player))
    (2 (funcall expanding-func board  2   1   player))
    (3 (funcall expanding-func board  1   2   player))
    (4 (funcall expanding-func board  -1  2   player))
    (5 (funcall expanding-func board  -2  1   player))
    (6 (funcall expanding-func board  -2  -1  player))
    (7 (funcall expanding-func board  -1  -2  player))
    (8 (funcall expanding-func board  1   -2  player))
  )
)
```

- Esta função vai devolver uma lista com todas as jogadas possíveis de um jogador num determinado tabuleiro. O jogador e o tabuleiro são passados por argumento. Caso a posição do jogador não seja encontrada significa que estamos no estado inicial e é apresentada a lista de jogadas possíveis nesse caso.

```
(defun list-possible-movements (board player)
  (remove nil (if (find-value board player)
    (mapcar (lambda (n) (operator (1+ n) board player 'possible-movement))
(list-numbers 8))
    (mapcar (lambda (movement-result) (find-value (movement-result-board
movement-result) player)) (all-horse-start-positions board player))
  ))
)
```

> Esta função faz recurso da função ***operator*** aplica a função ***possible-movement*** a uma lista de números de 1 a 8 para verificar qual das jogadas disponiveis são válidas.

- Esta função tem como objetivo colocar o cavalo automaticamente na casa com o maior.

```
(defun set-horse (board player)
  (let* (
      (horse-start-positions (all-horse-start-positions board player))
      (initial-value (create-movement-result most-negative-fixnum player
```

```
nil))
      (movement-result
        (reduce
          (lambda (current-max possible-max)
            (if (> (movement-result-score current-max) (movement-result-
score possible-max) )
              current-max
              possible-max
            )
          )
          horse-start-positions
          :initial-value initial-value
        )
      )
    )
    movement-result
  )
)
```

> Esta função faz recurso da função ***all-horse-start-positions*** para gerar todas as posições possíveis de começo de determinado cavalo.

- Esta função tem como objetivo colocar o cavalo do jogador passado como argumento na posição fornecida (só é fornecido a coluna pois a colocação do cavalo tem a linha predefinida).

```
(defun set-horse-start-position (board player &optional (pos 0))
"Coloca o cavalo numa certa posição do tabuleiro"
  (cond
    ((not (numberp player)) board)
    ((find-value board player) board)
    ((= player *player-one*)
      (remove-reverse-number (place 0 pos board player) (cell board 0 pos))
    )
    ((= player *player-two*)
      (remove-reverse-number (place 9 pos board player) (cell board 9 pos))
    )
    (T board)
  )
)
```

- Esta função tem como objetivo listar todas as jogadas possíveis do jogador passado por argumento.

```
(defun all-possible-movements (board player)
"Lista de todas as possiveis movimentações"
  (if (find-value board player)
    (remove nil
      (mapcar (lambda (n) (operator (1+ n) board player 'move-amount))
(list-numbers 8))
    )
```

```
    (all-horse-start-positions board player)
  )
)
```

> Esta função faz uso da função ***move-amount*** e ***operator*** para gerar todas as possibilidades de movimentação e verificar que não estão a ser ameaçadas pelo adversário.

- O objetivo desta função é verificar se uma jogada especifica do jogador humano é válida tanto em termo de posição como se está a ser ameaçado polo adversário.

```
(defun move-amount (board amount-row amount-column player)
"Move o cavalo (player) no tabuleiro pela quantidade de linhas e colunas
possiveis"
  (if (find-value board player)
    (let* (
        (horse-position (find-value board player))
        (new-row (+ amount-row (car horse-position)))
        (new-column (+ amount-column (cadr horse-position)))
      )
      (cond
        ((or (> new-column 9)(< new-column 0)(> new-row 9)(< new-row 0))
nil)
        (T
          (let
            ((value-at (cell board new-row new-column )))
            (cond
              ((not (numberp value-at)) nil)
              ((or (= value-at *player-one*) (= value-at *player-two*)) nil)
              ((some (lambda (e) (equal e (list new-row new-column))) (list-
possible-movements board (oposite-player player)))
                nil
              )
              (T
                (create-movement-result value-at player
                  (remove-reverse-number
                    (place new-row new-column (place (car horse-position)
(cadr horse-position) board) player)
                    value-at
...)
```

Ficheiro Interact

O ficheiro ***Interact*** deve ter como função carregar os outros ficheiros de código, escrever em ficheiros e toda a interação com o utilizador, desde processos do jogo como pedir o próximo movimento como mostrar a jogada a as estatísticas do computador.

- Esta função serve para criar um estado inicial com um tabuleiro aleatório e com os cavalos posicionados nas casa certas.

```
(defun initial-status ()
"Funcao para determinar o estado inicial do primeiro nó com o board gerado e
os cavalos colocados nas melhores posições"
  (let* (
      (movement-result-player-one (set-horse (random-board) *player-one*))
      (score-player-one (movement-result-score movement-result-player-one))
      (board-after-player-one (movement-result-board movement-result-player-
one))
      (movement-result-player-two (set-horse board-after-player-one *player-
two*))
      (final-board (movement-result-board movement-result-player-two))
      (score-player-two (movement-result-score movement-result-player-two))
    )

    (create-node
      score-player-one
      score-player-two
      *player-one*
      final-board
    )
  )
)
```

- Para dar inicio ao jogo inteiro é utilizada a função ***(start)*** que chama toda a mensagem inicial e espera por input do utilizador, caso o input seja correto vai para para a uma das opções dependendo, caso não passe na verificação será pedido novamente o input.

```
(defun start-message ()
"Mostra as opções iniciais"
  (progn
    (run-program "clear")
    (format t "~%╔═══════════════════════════════════════╗")
    (format t "~%║           Jogo do Cavalo              ║")
    (format t "~%║                                       ║")
    (format t "~%║   1. Jogador VS Computador            ║")
    (format t "~%║   2. Computador VS Computador         ║")
    (format t "~%║   0. Sair                             ║")
    (format t "~%║                                       ║")
    (format t "~%╚═══════════════════════════════════════╝   ~%~%> ")
  )
)

(defun start ()
"Funcao que inicia todo o processo do programa, apresenta as opcoes iniciais
e pede ao utilizador para escolher
  a opcao para avancar para o proximo passo"
  (progn
    (start-message)
    (let ((in (read)))
      (if (or (not (numberp in)) (> in 2) (< in 0)) (start))
```

```
      (cond
        ((eq in 1) (first-player))
        ((eq in 2) (time-play))
        ((eq in 0) (progn (format t "Obrigado!")(quit)))
      )
...
```

- Caso o utilizador escolha a opção "jogador vs computador" será direcionado para o ecrã de escolha para quem joga primeiro, da mesma maneira terá de colocar um valor válido, caso o faça será direcionado para o próximo ecrã.

```
(defun first-player()
"Funcao para determinar qual o primeiro a fazer a jogada, entre o jogador e
o computador"
  (progn
    (first-player-message)
    (let ((in (read)))
      (if (or (not (numberp in)) (> in 2) (< in 0)) (first-player-message))
      (cond
        ((eq in 1) (time-play 1))

        ((eq in 2) (time-play 2))

        ((eq in 0) (start))
      )
...
```

- A função de escolha de tempo vem com o argumento *first* que é quem joga primeiro no caso optional com *default nil* para caso o jogo seja Computador vs Computador, neste ecrã o utilizador deve introduzir o tempo que o computador tem para fazer a pesquisa de jogada, colocando um valor válido começa o jogo.

```
(defun time-play(&optional (first nil))
"Funcao para escolher quanto tempo o computador deve ter para fazer a sua
jogada"
  (progn
    (time-play-message)
    (let ((in (read)))
    (if (and (numberp in) (= 0 in)) (start)
      (if (or (not (numberp in)) (> in 5000) (< in 1000)) (time-play first)
        (start-game in first)
      ))
    )
  )
...
```

- A função ***(start-game)*** tem o objetivo de começar o jogo mas também de criar o loop entre jogador e computador na sua vez de jogar.

```
(defun start-game (time first)
"Funcao para começar e criar o loop de jogo caso seja jogador vs computador
ou so mesmo computador"
  (let (
      (current-node (initial-status))
      (user (when first (if (= first 1) *player-one* *player-two*)))
    )
    (node-print current-node)
    (loop
      (if (all-possible-movements (node-board current-node) (node-turn-
player current-node))
        (setf current-node
          (if (or (null user) (/= (node-turn-player current-node) user) )
            (let (
                (pc-move (alfabeta current-node 'all-possible-movements
*max-depth* time))
              )
              (print-move pc-move)
              (setf (node-parent (solution-node-optimal-move pc-move)) nil)
              (solution-node-optimal-move pc-move)
            )
            (player-move current-node time)
          )
        )
        (progn
          (setf (node-turn-player current-node) (oposite-player (node-turn-
player current-node)))
          current-node
        )
      )
      (when (no-possible-movements (node-board current-node)) (return
(final-results current-node)))
    )
  )
...
```

> Primeiramente cria e guarda em "current-node" o primeiro node com a função (initial-status) que cria um node com o tabuleiro aleatório já pronto com os cavalos colocados, guarda também qual o primeiro a jogar em "user". A partir dai entra num loop dependendo nas escolhas do jogador, estas sendo:
>
> - User nulo: Significa que o computador ***(minimax)*** será sempre chamado apenas alterando a vez do jogador (-1>-2>-1>-2...)
> - Vez do jogador: caso seja a vez do jogador vai ser chamada a função ***(player-move)*** que vai fazer a jogada do utilizador.
> - Vez do computador: caso seja a vez do computador vai ser chamada a função ***(minimax)*** para executar a jogada.

> O jogo vai ficar nesse *loop* até nenhum dos participantes tiver mais jogadas possíveis, quando acabar vai devolver o node final e mandar uma mensagem a apresentar o vencedor.

- A função de movimentação do jogador primeiramente verifica se o jogador pode sequer fazer um movimento, se não acaba logo a vez do utilizador de jogar e passa para o adversário.

  Caso seja possível pede ao utilizador para introduzir a linha e a coluna do movimento, este caso seja válido cria o node com esse movimento caso contrário pede ao utilizador um novo input.

```
(defun player-move (node time)
"Input e movimentação do jogador"
  (let ((list (list-possible-movements (node-board node) (node-turn-player
node))))
  (if list
    (progn
      (format t "Escolha a proxima posição do cavalo~%")
      (let* (
          (l (read-position-input "Linha"))
          (c (read-position-input "Coluna"))
        )
        (if (some (lambda (e) (equal e (list l c))) list)
          (next-node-player node l c time)
          (progn (format t "Movimento não é possivel, escolha outro!~%")
(player-move node time))
        )
      )
    )
    (progn
      (format t "Turno passado por não haver jogada possivel!")
      (setf (node-turn-player node) (oposite-player (node-turn-player
node)))
      node
    )
)
```

) ... ```

```
> A função ***(read-position-input)*** pede o input do utilizador mudando apenas a
palavra linha e coluna evitando código duplicado.
>
```

- A função ***(next-node-player)*** primeiramente atualiza os *scores* dos jogadores dependendo em quem faz a jogada, verifica se o número escolhido é um número duplo, se sim chama a função ***(double-n)*** em cima do novo node criado, se não for número duplo procede a criar apenas a node atualizado com a jogada.

```
(defun next-node-player (node l c time)
"Movimentação e gestao do tipo de movimento"
(let ((result (move (node-board node) l c (node-turn-player node))))
(if result
  (let* ((score-one (if (= (node-turn-player node) *player-one*)
            (+ (movement-result-score result) (node-score-one node))
            (node-score-one node)))
          (score-two (if (= (node-turn-player node) *player-two*)
            (+ (movement-result-score result) (node-score-two node))
            (node-score-two node)))
          )
    (if (and (/= 0 (movement-result-score result))(= (movement-result-score
result) (reverse-digits (movement-result-score result))))
      (double-n (create-node score-one score-two (oposite-player (node-turn-
player node)) (movement-result-board result) (node-depth node) most-
negative-fixnum node))
      (create-node score-one score-two (oposite-player (node-turn-player
node)) (movement-result-board result) (node-depth node) most-negative-fixnum
node)
    )
  )
  (progn (format t "Movimento não é possivel, escolha outro!") (player-move
node time))
 )
)
```

) ```

```
> A função ***(double-n)*** pede ao utilizador para remover um dos números duplos
ainda presentes no tabuleiro.
>
```

- As seguintes funções tem como objetivo darem print no ecrã e no documento ***log.dat*** das estatísticas da jogada do computador a cada jogada, com a seguinte informação:
    - número de nós analisados
    - número de cortes efetuados (alfa e beta)
    - tempo gasto na jogada
    - tabuleiro atual

```
(defun print-move (solution-node)
"Funcao de gestao de print das estatisticas"
(progn
    (with-open-file (stream "./log.dat" :direction :output :if-exists
:append :if-does-not-exist :create)
      (solution-node-print solution-node stream)
      (close stream)
    )
```

```
      (solution-node-print solution-node t)
    )

  )
```

Observações Finais

Como pode ser observado nas jogadas o algoritmo ***MiniMax*** com cortes ***AlfaBeta*** tende em optar sempre pela casa de maior valor como esperado. Apesar de as escolhas se comportarem como esperado o tempo de processamento tende em aumentar quando a profundidade é aumentada.

Quanto ao tempo da jogada do computador, devido ao ***lisp*** não suportar operações em milissegundos o numero é arredondado por excesso (EX: 5250ms = 6s), para resolver este problema encontramos ***packages*** externas mas não foram implementada devido á dependência de instalação. Adicionamos também um tempo de desconto extra para o algoritmo poder retornar os valores.

No que toca á pesquisa quiescente não foi totalmente implementado devido á falta de tempo e abundancia de trabalhos.