Manual Técnico

Projeto de Inteligência Artificial

Blokus Uno - Segunda Fase

Autores:

Tiago Farinha (201802235)

Francisco Moura (201802033)

Docente:

Eng. Filipe Mariano

Índice do Manual

Introdução

Este manual técnico conterá os detalhes de implementação da segunda fase do projeto **Blokus Uno**, realizado no âmbito da disciplina de Inteligência Artificial.

Estrutura do Programa

Este projeto estará dividido em 3 ficheiros diferentes, cada um com um propósito distinto.

Estes três ficheiros são os seguintes:

- **jogo.lisp** - ficheiro responsável pela interação com o utilizador, leitura e escrita de ficheiros e estatísticas.
- **algoritmo.lisp** - ficheiro responsável pelo algoritmo de procura, isto é, *NegaMaxAlfaBeta*, tal como funções auxiliares por este usadas.
- **puzzle.lisp** - ficheiro responsável pela adaptação do problema Blokus Uno ao algoritmo de procura genéricos.

jogo.lisp

As funções desenvolvidas neste ficheiro são de interação com o utilizador, estatísticas dos algoritmos implementados e de leitura e escrita em ficheiros externos.

O programa inicia-se com a função **iniciar**. Esta chamará a função **menu-inicial** que imprimirá no ecrã o menu e posteriormente recolherá a escolha do utilizador.

```
(defun iniciar()
"Função que imprime o menu e espera pelo input do utilizador, redirecionando-o
depois à 'página' seguinte"
    (menu-inicial)
    (setf *jogada* (list (tabuleiro-vazio) '(10 10 15) '(10 10 15)))
    (let ((comando (read)))
      (cond
       ((not (numberp comando)) (format t "Insira um número!~%") (iniciar))
       ((= comando 0) t)
       ((or (= comando 1) (= comando 2)) (executar-escolha-tempo-limite-IA
comando))
       (t (format t "Comando Inválido! Insira 1 para resolver um problema ou 0
para sair da aplicação...~%") (iniciar))
      )
    )
  (values)
)
```

A função **executar-escolha-tempo-limite-IA** é para escolher limite de tempo de execução do algoritmo.

```
(defun executar-escolha-tempo-limite-IA (jogo)
"Escolher limite de tempo de execução do algoritmo"
     (format t "Qual o tempo limite para o algoritmo em milésimos de segundo
(Insira 0 para voltar atrás)? ~%")
     (format t "> ")
     (setf *exit* nil)
     (let ((comando (read)))
       (cond
        ((not (numberp comando)) (progn (format t "~%O limite de tempo tem de
ser um NÚMERO ([1000,20000]ms)...~%") (executar-escolha-tempo-limite-IA jogo)))
        ((equal comando 0) (iniciar))
        ((or (< comando 1000) (> comando 20000)) (progn (format t "~%O limite de
tempo tem de estar no intervalo [1000, 20000]ms...~%") (executar-escolha-tempo-
limite-IA jogo)))
        (t
         (if (equal jogo 1)
             (escolha-primeiro-jogador comando)
             (progn (format t "Executando jogo. Por favor aguarde...~%") (AI-vs-
AI comando))

         )
        )
       )
     )
```

```
     (values)
)
```

A função **escolha-primeiro-jogador** é para escolher quem joga primeiro.

```
(defun escolha-primeiro-jogador (tempoLimite)
"Escolher quem joga primeiro"

     (format t "Quem deve começar o jogo (Humano - 1 | PC - 2)? (Insira 0 para
voltar ao menu inicial) ~%")
     (format t "> ")
     (let ((comando (read)) (outputType (make-array 0 :element-type 'character
                                                      :adjustable T
                                                      :fill-pointer 0)))
   (declare (type string outputType))
       (cond
        ((equal comando 0) (iniciar))
        ((or (equal comando 1) (equal comando 2)) (progn (formatar-solucao
*jogada* outputType) (Humano-vs-AI tempoLimite comando)))
        (t (progn (format t "~%O primeiro a jogar tem de ser um NÚMERO (1 ou
2)...~%") (escolha-primeiro-jogador tempoLimite)))
        )
       )
     (values)
)
```

A função **AI-vs-AI** é para correr o jogo no modo computador contra computador. Todas as jogadas e respetivas estatísticas são mostradas e escritas para o ficheiro *log.dat*. É também mostrado no fim o vencedor (ou empate, caso tal tenha ocorrido).

```
(defun AI-vs-AI (tempoLimite &optional (jogador 1))
"Correr o jogo no modo computador vs computador"
 (let ((noJogada (cria-no (first *jogada*) (second *jogada*) (third *jogada*)))
(outputType (make-array 0

             :element-type 'character

             :adjustable T

             :fill-pointer 0)))
  (declare (type string outputType))
  (if (not (and (no-solucaop noJogada 1) (no-solucaop noJogada 2)))
       (progn
            (let ((tempoInicio (GET-INTERNAL-RUN-TIME)))
                 (negamaxAlfaBeta noJogada 10 most-negative-fixnum most-
positive-fixnum jogador (+ (get-internal-real-time) tempoLimite))
                  (let ((tempoFim (GET-INTERNAL-RUN-TIME)))
                        (estatisticas-AI (- tempoFim tempoInicio))
                   )
              )
             (formatar-solucao *jogada* outputType)
             (if (equal jogador 1)
               (AI-vs-AI tempoLimite 2)
```

```
                (AI-vs-AI tempoLimite 1)
              )
        )
    (progn (calcular-vencedor (rest *jogada*) outputType) (iniciar))
  )
 )
)
```

A função **Humano-vs-AI** é para correr o jogo no modo humano contra computador. Todas as jogadas e respetivas estatísticas são mostradas e escritas para o ficheiro *log.dat*. É também mostrado no fim o vencedor (ou empate, caso tal tenha ocorrido).

```
(defun Humano-vs-AI (tempoLimite &optional jogador)
"Correr o jogo no modo humano vs computador"
  (let ((noJogada (cria-no (first *jogada*) (second *jogada*) (third *jogada*)))
(outputType (make-array 0

             :element-type 'character

             :adjustable T

             :fill-pointer 0)))
   (declare (type string outputType))
   (let ((fimJ1 (no-solucaop noJogada 1)) (fimJ2 (no-solucaop noJogada 2)))
     (if (not *exit*)
     (if (not (and fimJ1 fimJ2))
              (if (equal jogador 1)
                  (if (not fimJ1)
                              (progn (jogada-humano)
                                     (formatar-solucao *jogada* outputType)
                                     (Humano-vs-AI tempoLimite 2))
                    (Humano-vs-AI tempoLimite 2)
                  )
                  (if (not fimJ2)
                     (progn (format t "O Computador está pensar! Por favor
aguarde...~%")
                       (let ((tempoInicio (GET-INTERNAL-RUN-TIME)))
                            (negamaxAlfaBeta noJogada 10 most-negative-fixnum
most-positive-fixnum 2  (+ (get-internal-real-time) tempoLimite))
                            (let ((tempoFim (GET-INTERNAL-RUN-TIME)))
                               (estatisticas-AI (- tempoFim tempoInicio))
                            )
                       )

                            (formatar-solucao *jogada* outputType)
                            (Humano-vs-AI tempoLimite 1)
                    )
                    (Humano-vs-AI tempoLimite 1)
                  )
              )
      (progn (calcular-vencedor (rest *jogada*) outputType) (iniciar))
    ))
  )
 )
)
```

A função **jogada-humano** é para escolher a jogada do jogador humano no formato <peça, linha, coluna>. Esta pede um *input* ao jogador e apenas aceita a jogada caso esta seja possível e o comando esteja no formato correto.

```
(defun jogada-humano ()
"Escolher a jogada do jogador humano por input no formato <peca, linha, coluna>"
(if (not *exit*)
(progn (format t "Insira a jogada no formato: Peça linha coluna (ou 0 se desejar
voltar ao menu inicial)~%")
       (format t "> ")
                        (let ((input (read-line)) (outputType (make-array 0
                                                              :element-type
'character
                                                              :adjustable T
                                                              :fill-pointer 0)))
   (declare (type string outputType))
   (if (equal input "0")
     (progn (setf *exit* t)
   (iniciar)))
  (let ((entrada-tratada (tratar-entrada (split-sequence " " input))))
     (if (validar-jogada-utilizador entrada-tratada)
         (if (not (null (validar-peca-jogador-humano (first entrada-tratada))))
           (if (null (verificar-peca-encaixa (second entrada-tratada) (third
entrada-tratada) (first entrada-tratada) (first *jogada*) 1))
               (progn (format t "~%Posição inválida! Tente novamente...~%~%")
(jogada-humano))
             (progn (setf *exit* nil)
               (setf *jogada* (list (colocar-peca (first entrada-tratada)
(second entrada-tratada) (third entrada-tratada) (first *jogada*) 1)
(decrementar-nr-pecas (first entrada-tratada) (second *jogada*)) (third
*jogada*))) (valor-da-posicao outputType entrada-tratada))
           )
           (progn (format t "~%Não tem mais ~a!~%" (first entrada-tratada))
(jogada-humano))
         )
       (if (not *exit*)
           (progn (format t "~%Comando Inválido! Insira o formato
válido...~%~%") (jogada-humano)) )
     )
  ))))
)
```

As funções **validar-jogada-utilizador** e **tratar-entrada** verificam se a jogada está num formato válido e limpam e convertem o input para o formato correto, respetivamente.

```
(defun validar-jogada-utilizador (entrada-tratada)
"Função que verifica se a jogada está num formato válido"
  (if (and (equal (length entrada-tratada) 3)
           (verificar-dentro-dos-limites (second entrada-tratada) (third
entrada-tratada)))
            t
  )
)
```

```
(defun tratar-entrada (entrada)
"Função que 'limpa' e converte o input para o formato correto"
  (let ((entradaPreTratada (remove nil entrada)))
    (alisa (list (find-symbol (string-upcase (first entradaPreTratada)))
                 (if (and (not (null (second entradaPreTratada))) (not (null
(third entradaPreTratada))))
                     (list (parse-integer (second entradaPreTratada) :junk-
allowed t)
                           (parse-integer (third entradaPreTratada) :junk-
allowed t)
                     )
                 )
           )
    )
  )
)
```

A função **alisa** converte uma lista com sublista em apenas uma lista.

```
(defun alisa (l)
"Função que converte uma lista com sublista em apenas uma lista"
  (cond ((equal 0 (list-length l)) nil)
        ((if (atom (first l))
             (append (list (first l)) (alisa (rest l)))
             (append  (first l) (alisa (rest l)))
   )))
)
```

A função **escrever-ecra-ficheiro** é a função responsável por imprimir algo para o ecrã e para um ficheiro.

```
(defun escrever-ecra-ficheiro (outputType)
"Função que imprime algo para o ecrã e para um ficheiro"
  (if (not *exit*)
      (progn
  (with-open-file (ficheiro (make-pathname :host "c" :directory '(:absolute "IA
- Projeto 2") :name "log" :type "dat") :direction :output :if-exists :append
:if-does-not-exist :create)
  (format ficheiro outputType))
  (write-line outputType)))
```

A função **formatar-solucao** chama as funções responsáveis por formatar o tabuleiro e as peças adequadamente.

```
(defun formatar-solucao (solucao outputType)
"Chama a função que formata o tabuleiro e as peças adequadamente"
    (if (not (equal solucao nil))
        (progn (formatar-tabuleiro solucao outputType)
           (formatar-pecas (rest solucao) outputType)
           (escrever-ecra-ficheiro outputType)
           (values))
    )
)
```

A função **formatar-tabuleiro** formata o tabuleiro para ser apresentado no ecrã.

```
(defun formatar-tabuleiro (solucao outputType)
"Formata o tabuleiro para ser apresentado no ecrã"
  (format outputType " _____~%")
  (format outputType "|                                           |~%")
  (format outputType "|      A B C D E F G H I J K L M N    |")
  (format outputType "~%")
  (let ((linha 0))
  (mapcar #'(lambda (atual) (progn (setf linha (1+ linha))
                                   (if (< linha 11) (format outputType "|   ~a ~a
|~%" (1- linha) atual)
                                                    (format outputType "|  ~a ~a   |~%"
(1- linha) atual)))) (first solucao)))
  (format outputType "|_____|~%")
  (format outputType "~%")
  (values)
)
```

A função **formatar-pecas** formata as peças para serem apresentadas no ecrã. Indica por baixo da peça o ID para usar aquando a inserção da peça.

```
(defun formatar-pecas (pecas outputType)
"Formata as peças para serem apresentadas no ecrã"
  (format outputType "             Peças de Jogo:              ~%~%")
  (format outputType " Peça a    Peça b    Peça c1     Peça c2  ~%")
  (format outputType "(peca-a)  (peca-b)  (peca-c-1)  (peca-c-2) ~%")
  (format outputType "   _         ___         ___         _      ~%")
  (format outputType "  |_|       |_|_|       _|_|_|       |_|_    ~%")
  (format outputType "            |_|_|      |_|_|         |_|_|   ~%")
  (format outputType "                                      |_|   ~%~%")

  (format outputType " Peças do Jogador 1: ~%")
  (format outputType " Peca a - ~a~%" (first (first pecas)))
  (format outputType " Peca b - ~a~%" (second (first pecas)))
  (format outputType " Peca c1 e c2 - ~a~%~%" (third (first pecas)))
  (format outputType " Peças do Jogador 2: ~%")
  (format outputType " Peca a - ~a~%" (first (second pecas)))
  (format outputType " Peca b - ~a~%" (second (second pecas)))
  (format outputType " Peca c1 e c2 - ~a~%~%" (third (second pecas)))
  (values)
)
```

A função **calcular-vencedor** calcula o vencedor de um jogo e apresenta-o no ecrã e guarda no ficheiro *log.dat*.

```
(defun calcular-vencedor (pecas outputType)
"Calcula o vencedor de um jogo e apresenta-o (ecrã e ficheiro)"
  (let ((pj1 (calcular-pontos-jogador (first pecas))) (pj2 (calcular-pontos-
jogador (second pecas))))
    (format outputType " _____~%")
    (format outputType "|                              |~%")
    (format outputType "| Pontuação do Jogador 1: ~a |~%" pj1)
    (format outputType "| Pontuação do Jogador 2: ~a |~%" pj2)
    (format outputType "|                              |~%")
    (cond
     ((< pj1 pj2) (format outputType "| O vencedor é o Jogador 1!  |~%"))
     ((< pj2 pj1) (format outputType "| O vencedor é o Jogador 2!  |~%~%"))
     (t (format outputType           "|          Empate!           |~%"))
    )
    (format outputType "|_____|~%~%~%")
    (format outputType "-------------------------------------------------------
------------~%")
    (escrever-ecra-ficheiro outputType)
  )
  (values)
)
```

As funções **valor-da-posicao** e **estatisticas-AI** apresentam as estatísticas pedidas no enunciado, mais especificamente, a posição da jogada, o número de nós, número de cortes e o tempo de execução. O tempo de execução é o tempo que o algoritmo *NegaMaxAlfaBeta* demora desde que é chamado até devolver a jogada.

```
(defun valor-da-posicao (outputType jogada)
"Imprime a posição da jogada no formato <peca, linha, coluna> (ecrã e ficheiro)"
  (format outputType "Posição para onde jogou: ~a ~a ~a~%" (first jogada)
(second jogada) (third jogada))
  (escrever-ecra-ficheiro outputType)
)

(defun estatisticas-AI (tempoExec)
"Imprime as estatísticas: nós explorados, número de cortes e o tempo de execução
(ecrã e ficheiro)"
  (let ((outputType (make-array 0
                                :element-type 'character
                                :adjustable T
                                :fill-pointer 0)))
   (declare (type string outputType))
   (format outputType "~%Número de nós analisados: ~a~%" (first
*nrSucessoresECortes*))
   (format outputType "Número de cortes: ~a~%" (second *nrSucessoresECortes*))
   (format outputType "Tempo de Execução: ~a ms~%" tempoExec)
   (setf *nrSucessoresECortes* '(0 0))
   (escrever-ecra-ficheiro outputType)
  )
)
```

algoritmo.lisp

Algoritmo NegaMaxAlfaBeta

O *MiniMax* é desenhado para determinar a estratégia ótima para o *MAX*.

O nosso algoritmo **negaMaxAlfaBeta** não é mais do que outra formulação do *MiniMax* em que se passa a procurar sempre apenas o máximo, mas se troca o sinal em cada nível, após o backup e ignora os ramos que não interessam.

```
(defun negamaxAlfaBeta(no prof alfa beta jogador endTime &aux (bestValue most-
negative-fixnum))
"Implementação do Algoritmo Negamax com cortes."
  (let ((filhosOrdenados (sortNodes (nos-possiveis-todas-pecas no jogador))))
    (setf *nrSucessoresECortes* (list (+ (list-length filhosOrdenados) (first
*nrSucessoresECortes*)) (second *nrSucessoresECortes*)))
    (if  (or (equal prof 0) (= (list-length filhosOrdenados) 0) (> (get-
internal-real-time) endTime))
      (progn (setf *jogada* (list (first (second (solucaoProblema no))) (fourth
(second (solucaoProblema no))) (fifth (second (solucaoProblema no)))))
             (return-from negamaxAlfaBeta (no-heuristica no))
      )

       (progn
          (mapcar #'(lambda (atual)
                      (if (equal jogador 1)
                         (setf bestValue (max bestValue (- (negamaxAlfaBeta
atual (1- prof) (- beta) (- alfa) 2 endTime))))
                         (setf bestValue (max bestValue (- (negamaxAlfaBeta
atual (1- prof) (- beta) (- alfa) 1 endTime))))
                      )
                      (setf alfa (max alfa bestValue))
                      (if (>= alfa beta)
                        (progn (setf *nrSucessoresECortes* (list (first
*nrSucessoresECortes*) (+ (list-length filhosOrdenados) (second
*nrSucessoresECortes*))))
                        (return-from negamaxAlfaBeta alfa))
                      )
                    )
            filhosOrdenados)

         bestValue
        )
       )
   )
)
```

Foram criadas várias outras funções auxiliares de apoio aos algoritmos.

A função **sortNodes** ordena os nós por ordem crescente da heurística.

```
(defun sortNodes (nodesList)
"Ordena uma lista de nós"
      (sort nodesList #'> :key #'sixth)
)
```

A função **solucaoProblema** devolve o caminho da solução ao estado inicial do problema.

```
(defun solucaoProblema (noSolucao)
"Função que retorna a lista do caminho da solução."
  (cond
   ((equal (no-pai noSolucao) nil) (list noSolucao))
   (t (append (solucaoProblema (no-pai noSolucao)) (list noSolucao)))
  )
)
```

A função **calcular-pontos-jogador** devolve a pontuação final de um jogador (menor é melhor).

```
(defun calcular-pontos-jogador (pecasJogador)
"Calcula a pontuação final de um jogador (numero de peças vezes o seu valor)"
    (+ (first pecasJogador) (* (+ (second pecasJogador) (third pecasJogador))
4))
)
```

A função **cria-no** representa a estrutura de um nó. As funções seguintes devolvem os elementos individuais de um nó (estado/tabuleiro, profundidade, o nó antecessor, o número de peças do jogador 1 e 2 e o valor heurístico, respetivamente).

```
(defun cria-no (tray nrPecasJ1 nrPecasJ2 &optional (prof 0) (pai nil)
(heuristica nil))
"Cria a estrutura do nó"
  (list tray prof pai nrPecasJ1 nrPecasJ2 heuristica)
)

(defun no-estado (no)
"Devolve o tabuleiro atual"
  (first no)
)

(defun no-profundidade (no)
"Devolve o nível de profundidade no grafo"
  (second no)
)

(defun no-pai (no)
"Devolve o pai do nó atual"
  (third no)
)

(defun nr-pecasJ1 (no)
"Devolve a lista com a quantidade de peças"
```

```
  (fourth no)
)

(defun nr-pecasJ2 (no)
"Devolve a lista com a quantidade de peças"
  (fifth no)
)

(defun no-heuristica (no)
"Devolve a lista com a quantidade de peças"
  (sixth no)
)
```

A função **no-solucaop** devolve verdadeiro caso o nó que recebe não tem mais sucessores possíveis, indicando que esta é uma solução.

```
(defun no-solucaop (no)
"Função que retorna t se o nó recebido por parâmetro é nó solução. Nil caso
contrário."
  (if (equal (length (nos-possiveis-todas-pecas  no)) 0) t nil)
)
```

A função **nos-posicoes-possiveis** devolve uma lista com os nós sucessores de um nó para uma peça e jogador especificos.

```
(defun nos-posicoes-possiveis(no peca jogador)
"Função que verifica e coloca as jogadas possíveis (em nós - (cria-no no)) numa
lista a ser usada, mais tarde, nos algoritmos."
        (if (equal jogador 1)
          (if (> (cond
                ((equal peca 'peca-a) (first (nr-pecasJ1 no)))
                ((equal peca 'peca-b) (second (nr-pecasJ1 no)))
                ((or (equal peca 'peca-c-1) (equal peca 'peca-c-2)) (third (nr-
pecasJ1 no)))) 0)

        (mapcar #'(lambda(sucessorAtual)
                                (cria-no sucessorAtual (decrementar-nr-pecas
peca (nr-pecasJ1 no)) (nr-pecasJ2 no) (1+ (no-profundidade no)) no (avaliar-no
(list (decrementar-nr-pecas peca (nr-pecasJ1 no)) (nr-PecasJ2 no)) jogador))
                          )

                (mapcar #'(lambda(coordAtual)
                            (verificar-peca-encaixa-e-colocar  (first
coordAtual) (second coordAtual) peca (no-estado no) jogador))
                      (sucessores (no-estado no) 0 0 peca jogador)
         ))
          nil)
         (if (> (cond
                ((equal peca 'peca-a) (first (nr-pecasJ2 no)))
                ((equal peca 'peca-b) (second (nr-pecasJ2 no)))
                ((or (equal peca 'peca-c-1) (equal peca 'peca-c-2)) (third (nr-
pecasJ2 no)))) 0)
```

```
        (mapcar #'(lambda(sucessorAtual)
                                (cria-no sucessorAtual (nr-pecasJ1 no)
(decrementar-nr-pecas peca (nr-pecasJ2 no)) (1+ (no-profundidade no)) no
(avaliar-no (list (nr-PecasJ1 no) (decrementar-nr-pecas peca (nr-pecasJ2 no)))
jogador))
                            )
                (mapcar #'(lambda(coordAtual)
                            (verificar-peca-encaixa-e-colocar  (first
coordAtual) (second coordAtual) peca (no-estado no) jogador))
                    (sucessores (no-estado no) 0 0 peca jogador)
        ))
          nil)
       )
)
```

A função **sucessores** devolve a lista de sucessores de um nó.

```
(defun sucessores (tray line row peca jogador &optional (listaSucessores '()))
"Devolve a lista de sucessores de um nó"
  (if (equal (verificar-dentro-dos-limites line row) nil)
      (reverse listaSucessores)
      (if (equal (verificar-peca-encaixa line row peca tray jogador) t)
          (if (> row 12) (sucessores tray (1+ line) 0 peca jogador (append (list
(list line row)) listaSucessores )) (sucessores tray line (1+ row) peca jogador
(append (list (list line row)) listaSucessores)))
          (if (> row 12) (sucessores tray (1+ line) 0 peca jogador
listaSucessores) (sucessores tray line (1+ row) peca jogador listaSucessores))
      )
  )
)
```

A função **decrementar-nr-pecas** decrementa o número de uma peça e devolve a lista das peças atualizada.

```
(defun decrementar-nr-pecas (peca nrPecas)
"Decrementa o numero de uma peça"
      (cond
       ((equal peca 'peca-a) (list (1- (first nrPecas)) (second nrPecas) (third
nrPecas)))
       ((equal peca 'peca-b) (list (first nrPecas) (1- (second nrPecas)) (third
nrPecas)))
       ((or (equal peca 'peca-c-1) (equal peca 'peca-c-2)) (list (first nrPecas)
(second nrPecas) (1- (third nrPecas))))
      )
)
```

A função **avaliar-no** é a heurística que escolhemos implementar, consistindo na subtração dos pontos das peças que sobram do jogador *max* pelos do *min* (ou vice versa).

```
(defun avaliar-no (l jogador)
"Função Heurística"
  (if (equal jogador 1)
    (- (+ (first (first l)) (* (+ (second (first l)) (third (first l))) 4)) (+
(first (second l)) (* (+ (second (second l)) (third (second l))) 4)))
    (- (+ (first (second l)) (* (+ (second (second l)) (third (second l))) 4))
(+ (first (first l)) (* (+ (second (first l)) (third (first l))) 4)))
  )
)
```

puzzle.lisp

Este ficheiro contém as funções referentes às peças em si, validações dessas peças e modificações ao tabuleiro. É o ficheiro responsável por adaptar o problema aos algoritmos.

A função **linha** recebe um Índice e o tabuleiro e retorna uma lista que representa essa linha do tabuleiro, a **coluna** recebe um índice e o tabuleiro e retorna uma lista que representa essa coluna do tabuleiro e a **celula** recebe dois índices (linha e coluna) e o tabuleiro e retorna o valor presente nessa calcula do tabuleiro.

```
(defun linha (index tray)
  "Função que recebe um Índice e o tabuleiro e retorna uma lista que representa
essa linha do tabuleiro"
  (nth index tray)
)


(defun coluna (index tray)
  "Função que recebe um índice e o tabuleiro e retorna uma lista que representa
essa coluna do tabuleiro."
  (mapcar #'(lambda (curr) (nth index curr)) tray)
)


(defun celula (line row tray)
  "Função que recebe dois índices (linha e coluna) e o tabuleiro e retorna o
valor presente nessa calcula do tabuleiro."
  (if (equal (verificar-dentro-dos-limites line row) t)
      (nth row (nth line tray))
  )
)
```

As funções **casa-vaziap** e **verifica-casas-vazias** são responsáveis por verificar se as peças não estão a ser inseridas em cima de outra peça.

```
(defun casa-vaziap (line row tray)
  "Função que recebe dois índices e o tabuleiro e verifica se a casa
correspondente a essa posição se encontra vazia, ou seja, igual a 0."
  (if (equal (celula line row tray) 0)
      T NIL
  )
)

(defun verifica-casas-vazias (tray positions)
  "Função que recebe o tabuleiro e uma lista de pares de índices linha e coluna
e devolve uma lista com T ou NIL caso se encontrem vazias."
  (mapcar #'(lambda (curr)
              (if (equal (casa-vaziap (first curr) (second curr) tray) T) T NIL)
            ) positions)
)
```

As funções **substituir-posicao** e **substituir** são responsáveis por colocar no tabuleiro uma peça, dada a coordenada inicial de uma peça.

```
(defun substituir-posicao (row lineList jogador)
  "Função que recebe um índice, uma lista e um valor (por default o valor é 1) e
substitui pelo valor pretendido nessa posição"
  (let ((currPos 0))
    (mapcar #'(lambda (curr)
                (if (and (= row currPos) (= curr 0)) (progn (setf currPos (1+
currPos)) jogador)
                                      (progn (setf currPos (1+ currPos)) curr)
                )
              )
   lineList)
  )
)

(defun substituir (line row tray jogador)
  "Função que recebe 2 índices, o tabuleiro e um valor (por default = 1) e
retorna o tabuleiro com a substituição pelo valor pretendido."
  (let ((currLine 0))
        (mapcar #'(lambda (curr)
                (if (= line currLine) (progn (setf currLine (1+ currLine))
(substituir-posicao row (linha line tray) jogador))
                                        (progn (setf currLine (1+ currLine)) curr)
                )
              )
         tray)
  )
)
```

A função **peca-casas-ocupadas** recebe dois índices e um tipo de peça (peca-a, peca-b, peca-c-1 ou peca-c-2) e retorna uma lista com os pares de índices correspondentes às posições em que irá ser colocada a peça. As restantes funções colocam uma peça de acordo com os índices fornecidos.

```
(defun peca-casas-ocupadas (line row piece)
"Função que recebe dois índices e um tipo de peça (peca-a, peca-b, peca-c-1 ou
peca-c-2) e retorna uma lista com os pares de índices correspondentes às posições
em que irá ser colocada a peça."
  (cond
     ((equal piece 'peca-a) (list (list line row)))
     ((equal piece 'peca-b) (list (list line row) (list line (1+ row)) (list (1+
line) row) (list (1+ line) (1+ row))))
     ((equal piece 'peca-c-1) (list (list line row) (list line (1+ row)) (list
(1- line) (1+ row)) (list (1- line) (+ row 2))))
     ((equal piece 'peca-c-2) (list (list line row) (list (1+ line) row) (list
(1+ line) (1+ row)) (list (+ line 2) (1+ row))))
  )
)


(defun peca-a (line row tray jogador)
  "Função que recebe dois índices e o tabuleiro e coloca um quadrado de 1x1 no
tabuleiro"
  (substituir line row tray jogador)
)


(defun peca-b (line row tray jogador)
  "Função que recebe dois índices e o tabuleiro e coloca o quadrado 2x2 no
tabuleiro tendo como ponto de referência o índice passado como argumento."
  (progn (substituir (1+ line) (1+ row) (substituir (1+ line) row (substituir
line (1+ row) (substituir line row tray jogador) jogador) jogador) jogador)
  )
)

(defun peca-c-1 (line row tray jogador)
  "Função que recebe dois índices e o tabuleiro e coloca a peça 'esse' na
posição de lado no tabuleiro tendo como ponto de referência o índice passado como
argumento."
  (progn (substituir (1- line) (+ row 2) (substituir (1- line) (1+ row)
(substituir line (1+ row) (substituir line row tray jogador) jogador) jogador)
jogador)
  )
)


(defun peca-c-2 (line row tray jogador)
  "Função que recebe dois índices e o tabuleiro e coloca a peça 'esse' na
posição para cima no tabuleiro tendo como ponto de referência o índice passado
como argumento."
  (progn (substituir (+ line 2) (1+ row) (substituir (1+ line) (1+ row)
(substituir (1+ line) row (substituir line row tray jogador) jogador) jogador)
jogador)
    )
)
```

As principais funções deste ficheiro são **verificar-peca-encaixa-e-colocar** e **verificar-peca-encaixa**. A primeira chama a função **colocar-peca** caso se tenha verificado que a peça efetivamente pode ser colocada inteiramente nessa posição sem violar as regras do jogo.

```
(defun colocar-peca(peca line row tray jogador)
"Função que chama uma das funções de colocação de uma peça"
  (cond
   ((equal peca 'peca-a) (peca-a line row tray jogador))
   ((equal peca 'peca-b) (peca-b line row tray jogador))
   ((equal peca 'peca-c-1) (peca-c-1 line row tray jogador))
   ((equal peca 'peca-c-2) (peca-c-2 line row tray jogador))
  )
)

(defun verificar-peca-encaixa-e-colocar (line row peca tray jogador)
"Função que coloca a peça no tabuleiro, verificando se estas podem ser inseridas.
Se todas as validações forem positivas, adicionar peça ao tabuleiro. NIL caso
contrário."
    (if (equal (verificar-peca-encaixa line row peca tray jogador) nil)
        nil
        (colocar-peca peca line row tray jogador)
    )
)

(defun verificar-peca-encaixa (line row peca tray jogador)
"Função que coloca a peça no tabuleiro, verificando se estas podem ser inseridas.
Se todas as validações forem positivas, adicionar peça ao tabuleiro. NIL caso
contrário."
  (let ((piece (peca-casas-ocupadas line row peca)))
    (if (not (member nil (mapcar #'(lambda (atual)
                (if
                  (and
                        (not (equal (celula (first atual) (1- (second atual))
tray) jogador))
                        (not (equal (celula (first atual) (1+ (second atual))
tray) jogador))
                        (not (equal (celula (1- (first atual)) (second atual)
tray) jogador))
                        (not (equal (celula (1+ (first atual)) (second atual)
tray) jogador))
                        (verificar-dentro-dos-limites (first atual)(second
atual))
                        (casa-vaziap (first atual) (second atual) tray))
                  t nil
                )
              )piece)))
        (if (or (member t (mapcar #'(lambda (atual)
                (if
                  (and
                        (or (equal (celula (1- (first atual)) (1- (second
atual)) tray) jogador)
                            (equal (celula (1+ (first atual)) (1- (second
atual)) tray) jogador)
                            (equal (celula (1- (first atual)) (1+ (second
atual)) tray) jogador)
                            (equal (celula (1+ (first atual)) (1+ (second
atual)) tray) jogador)
                        )
                   )
                   t nil
                )
```

```
                ) piece))
      (member t (mapcar #'(lambda (atual)
                            (if (or (and (equal (first atual) 0) (equal (second
atual) 0) (equal jogador 1)) (and (equal (first atual) 13) (equal (second atual)
13) (equal jogador 2)))
                              t
                            )
              ) piece)))
            t
)
     )
  )
)
```

A função **verificar-dentro-dos-limites** verifica se a linha e coluna passadas como argumento estão dentro dos limites do tabuleiro. A função **validar-peca-jogador-humano** verifica se o jogador tem peças suficientes para jogar.

```
(defun verificar-dentro-dos-limites (line row)
"Verifica se alguma das coordenada de uma calccula está fora do tabuleiro. T se
todas estiverem dentro do tabuleir, NIL caso contrário."
  (if (and (numberp line) (numberp row))
      (if
          (and (>= line 0)
               (<= line 13)
               (>= row 0)
               (<= row 13)) t nil
       )
  )
)

(defun validar-peca-jogador-humano (peca)
"Verifica se o jogador tem peças suficientes para jogar"
  (cond
   ((equal peca 'peca-a) (if (<= (first (second *jogada*)) 0) nil t))
   ((equal peca 'peca-b) (if (<= (second (second *jogada*)) 0) nil t))
   ((or (equal peca 'peca-c-1) (equal peca 'peca-c-2)) (if (<= (third (second
*jogada*)) 0) nil t))
  )
)
```

Limitações

O IDE fornecido limita seriamente a capacidade de memória que podemos usar, tendo resultado muitas vezes no fecho repentino do mesmo.

Não fomos capazes de implementar a memoização e quiescência apenas pela limitada disponibilidade de tempo.