

Projeto N.º 1 - *Dots and Boxes*

Inteligência Artificial - Escola Superior de Tecnologia de Setúbal
2022/2023

Manual Técnico

Nuno Martinho, n.º 201901769

João Coelho, n.º 201902001

Índice

- [Introdução](#)
- [Organização do projeto](#)
- [projeto.lisp](#)
- [puzzle.lisp](#)
- [procura.lisp](#)
- [Resultados](#)
- [Limitações](#)

Introdução

Neste manual técnico é abordada a implementação de um programa em *LISP* que tem como objetivo resolver tabuleiros do jogo *Dots and Boxes*.

O objetivo deste é permitir que o utilizador possa receber uma solução possível do número de jogadas num tabuleiro de modo a completar o objetivo de caixas fechadas.

Organização do projeto

O projeto foi implementado no *Visual Studio Code* com recurso ao programa **clisp v2.49**.

Encontra-se organizado em 3 ficheiros de código:

- **projeto.lisp** - carrega os outros ficheiros de código, escreve e lê ficheiros, e trata da interação com o utilizador.
- **puzzle.lisp** - implementação da resolução do problema.
- **procura.lisp** - implementação dos algoritmos de procura.

projeto.lisp

Neste ficheiro encontram-se funções relativas ao carregamento, leitura e escrita de ficheiros, bem como à interação com o utilizador.

O programa é iniciado ao executar a função **iniciar** que apresenta um menu principal com 3 opções:

1. mostrar um tabuleiro entre todos os disponíveis no ficheiro **problemas.dat**
2. resolver um tabuleiro
3. sair do programa.

```
;; Função iniciar
(defun iniciar ()
  "Inicializa o programa"
  (menu)
  (let ((opcao (read)))
    (if
      (or (not (numberp opcao)) (< opcao 1) (> opcao 3))
      (progn (format t "Escolha uma opção válida!") (iniciar))
      (ecase opcao
        ('1 (progn
```

Menu principal

Para que seja possível apresentar os tabuleiros contidos no ficheiro **problemas.dat** é usada uma função **ler-tabuleiros** para ler esse mesmo ficheiro.

Ao seleccionar a primeira opção *Visualizar problemas* o utilizador verá então no ecrã uma lista com todos os tabuleiros lidos a partir da função anterior. Poderá depois seleccionar um dos tabuleiros para poder ver o mesmo impresso no ecrã.

2 / 15

```
(format t "~%| " )
(format t "~%|          0 - Voltar atras      |")
(format t "~%O         o")
(format t "~%~%>> ")
)
)
(T (progn
    (if (= i 1)
        (progn
            (format t "~%O         o")
            (format t "~%|          - Escolha o tabuleiro: - |")
            (format t "~%|          |")
        )
    )
    (format t "~%|          ~a - Tabuleiro ~a           |" i (code-char (+ i 64)))
    (tabuleiros-menu (+ i 1) (cdr problemas))
)
)
)
```

```
;; Função opcao-tabuleiro: permite ao utilizador escolher um tabuleiro
(defun opcao-tabuleiro (&optional (voltar 'iniciar))
  "Recebe um tabuleiro do menu"
  (progn
    (tabuleiros-menu)
    (let ((opcao (read)))
      (cond ((equal opcao '0) (funcall voltar))
            ((not (numberp opcao)) (progn (format t "Escolha uma opção válida~%"))))
      (T
       (let ((lista (ler-tabuleiros)))
         (if (or (< opcao 0) (> opcao (length lista)))
             (progn
              (format t "Escolha uma opcao valida!") (opcao-tabuleiro 'tabuleiros-
menu)
              )
             (list opcao (nth (1- opcao) lista))
            )
        )
      )
    )
  )
)
```

```
0
|
| - Escolha o tabuleiro: -
|
|     1 - Tabuleiro A
|     2 - Tabuleiro B
|     3 - Tabuleiro C
|     4 - Tabuleiro D
|     5 - Tabuleiro E
|     6 - Tabuleiro F
0

0 - Voltar atras

>> █
```

Voltoando ao menu principal, temos a segunda opção do menu **Resolver um problema** que permite ao utilizador escolher um algoritmo, um tabuleiro, o objetivo de caixas fechadas e, se aplicável, profundidade máxima. No fim, é calculada a solução e apresentada no ecrã do

utilizador, voltando de seguida ao menu inicial.

Também é criado o ficheiro **resultados.dat** com resultados mais detalhados de todas as execuções realizadas.

Menu Algoritmos

```
o                                     o
|                                     |
| - Escolha o algoritmo -           |
|                                     |
| 1 - Breadth-First                 |
| 2 - Depth-First                   |
| 3 - A*                           |
|                                     |
| 0 - Voltar                        |
|                                     |
o                                     o
>> |
```

Menu Objetivo Caixas Fechadas

```
o                                     o
|                                     |
| - Defina o numero de caixas fechadas - |
|                                     |
| 0 - Voltar                        |
|                                     |
o                                     o
>> |
```

Menu Profundidade

```
o                                     o
|                                     |
| - Defina a profundidade maxima -   |
| - a utilizar -                     |
|                                     |
| 0 - Voltar                        |
|                                     |
o                                     o
>> |
```

Menu Heurística

```
o                                     o
|                                     |
| - Defina a heuristica a utilizar - |
|                                     |
| 1 - Heuristica Enunciado           |
| 2 - Heuristica Personalizada       |
|                                     |
| 0 - Voltar                        |
|                                     |
o                                     o
>> |
```

```
;; Função opcao-objetivo: trata a opção do número de caixas fechadas escolhido do utilizador
(defun opcao-objetivo ()
  "Recebe um valor de caixas fechadas do utilizador"
  (progn
    (objetivo-menu)
    (let ((opcao (read)))
      (cond ((equal opcao '0) (opcao-tabuleiro 'opcao-algoritmo))
            ((or (not (numberp opcao)) (< opcao 0))
             (progn
              (format t "Escolha uma opcao valida!~%")
              (opcao-objetivo)
              )
            )
            (T opcao)
          )
    )
  )
)
```

```

;; Função opcao-profundidade: trata a opção de profundidade máxima do utilizador
(defun opcao-profundidade ()
  "Recebe um valor de profundidade maxima do utilizador"
  (if (not (profundidade-menu))
      (let ((opcao (read)))
        (cond ((equal opcao '0) (opcao-objetivo))
              ((or (not (numberp opcao)) (< opcao 0))
               (progn
                (format t "Escolha uma opcao valida!~%")
                (opcao-profundidade 'profundidade-menu)
                )
              )
              (T opcao)
              )
      )
  )
)

;; Função opcao-heuristica: trata a opção de escolha de heuristica do utilizador
(defun opcao-heuristica ()
  "Recebe um valor que cooresponde a heuristica escolhida pelo utilizador"
  (if (not (heuristica-menu))
      (let ((opcao (read)))
        (cond ((equal opcao '0) (opcao-objetivo))
              ((or (not (numberp opcao)) (< opcao 0) (> opcao 2))
               (progn
                (format t "Escolha uma opcao valida!~%")
                (opcao-heuristica 'heuristica-menu)
                )
              )
              (T (ecase opcao
                   (1 'heuristica-base)
                   (2 'heuristica-top)
                   )
              ))
      )
  )
)

;; Função opcao-algoritmo: trata a opção de algoritmo do utilizador
(defun opcao-algoritmo ()
  "Recebe a opcao de algoritmo do utilizador e executa-o"
  (progn
    (algoritmos-menu)
    (let ((opcao (read)))
      (cond ((equal opcao '0) (iniciar))
            ((or (< opcao 0) (> opcao 4)) (progn (format t "Escolha uma opcao valida!~%")
            (opcao-algoritmo)))
            ((not (numberp opcao)) (progn (format t "Escolha uma opcao valida!~%")))
            (T (let* (
                  (no-tabuleiro (opcao-tabuleiro 'opcao-algoritmo))
                  (objetivo (opcao-objetivo))
                  (id-tabuleiro (code-char (+ (first no-tabuleiro) 64)))
                  (tabuleiro (second no-tabuleiro))
                  (no (list (criar-no tabuleiro nil objetivo)))
                  )
                  (ecase opcao
                    (1
                     (let ((solucao (list id-tabuleiro 'BFS objetivo (hora-atual) (bfs
'expandir-no no) (hora-atual))))
                     (progn

```

```

                                (ficheiro-estatisticas solucao)
                                solucao
                                )
                            )
                        )
                    (2
                        (let* (
                            (profundidade (opcao-profundidade))
                            (solucao (list id-tabuleiro 'DFS objetivo (hora-atual) (dfs
'expandir-no profundidade no) (hora-atual) profundidade))
                                )
                                (progn
                                    (ficheiro-estatisticas solucao)
                                    solucao
                                )
                            )
                        )
                    )
                (3
                    (let* (
                        (heuristica (opcao-heuristica))
                        (solucao (list id-tabuleiro 'A* objetivo (hora-atual) (a*
'expandir-no-a* heuristica no) (hora-atual)))
                            )
                            (progn
                                (ficheiro-estatisticas solucao)
                                solucao
                            )
                        )
                    )
                )
            )
        ))
    )
)

```

```

;;; Funções auxiliares para criação do ficheiro output resultados.dat

(defun ficheiro-estatisticas (solucao)
  "Ficheiro de resultados estatísticos (solucao + dados estatísticos sobre a eficiencia)"
  (let* (
    (id-tabuleiro (first solucao))
    (algoritmo (second solucao))
    (objetivo (third solucao))
    (hora-inicio (fourth solucao))
    (caminho-solucao (fifth solucao))
    (hora-fim (sixth solucao))
    (profundidade (seventh solucao))
  )

    (with-open-file (file "resultados.dat" :direction :output :if-does-not-exist :create :if-exists
:append)
      (ecase algoritmo
        ('bfs (estatisticas file id-tabuleiro algoritmo objetivo caminho-solucao hora-inicio
hora-fim))
        ('dfs (estatisticas file id-tabuleiro algoritmo objetivo caminho-solucao hora-inicio
hora-fim profundidade))
        ('a* (estatisticas file id-tabuleiro algoritmo objetivo caminho-solucao hora-inicio
hora-fim))
      )
    )
  )
)

```

```

(defun hora-atual ()
  "Retorna a hora atual (hh mm ss)"
  (multiple-value-bind (s m h)
    (get-decoded-time)
    (format nil "~a:~a:~a" h m s))
)

(defun estatisticas (stream id-tabuleiro algoritmo objetivo caminho-solucao hora-inicio hora-fim
&optional profundidade)
  "Solução e dados de eficiência para os algoritmos"
  (progn
    (format stream "~%Tabuleiro ~a" id-tabuleiro)
    (format stream "~% - Algoritmo: ~a" algoritmo)
    (format stream "~% - Objetivo: ~a caixas" objetivo)
    (format stream "~% - Solucao encontrada")
    (print-tabuleiro (no-solucao caminho-solucao) stream)
    (format stream "~% - Fator de ramificacao media: ~f" (fator-ramificacao-media caminho-solucao))
    (if (eql algoritmo 'DFS)
      (format stream "~% - Profundidade maxima: ~a" profundidade)
    )
    (format stream "~% - Nº nos gerados: ~a" (num-nos-gerados caminho-solucao))
    (if (eql algoritmo 'A*)
      (format stream "~% - Nº nos expandidos: ~a" (num-nos-expandidos-a* caminho-solucao))
      (format stream "~% - Nº nos expandidos: ~a" (num-nos-expandidos caminho-solucao))
    )
    (format stream "~% - Penetrancia: ~f" (penetrancia caminho-solucao))
    (format stream "~% - Inicio: ~a" hora-inicio)
    (format stream "~% - Fim: ~a~%~%" hora-fim)
  )
)

```

puzzle.lisp

Aqui estão presentes as funções relativas à resolução do problema em si.

Foram implementadas algumas funções de tabuleiros de teste mais básicos para ir retificando a implementação das funções.

```

;; ===== TABULEIROS PARA TESTE =====
;; '( ((0 0 0) (0 0 1) (0 1 1) (0 0 1)) ((0 0 0) (0 1 1) (1 0 1) (0 1 1)) )
(defun tabuleiro-teste ()
  "Retorna um tabuleiro 3x3 (3 arcos na vertical por 3 arcos na horizontal)"
  '(
    ((0 0 0) (0 0 1) (0 1 1) (0 0 1))
    ((0 0 0) (0 1 1) (1 0 1) (0 1 1))
  )
)

(defun tabuleiro-teste-simples ()
  "Retorna um tabuleiro 2x2 (2 arcos na vertical por 2 arcos na horizontal)"
  '(
    ((0)(0))
    ((0)(1))
  )
)

(defun tabuleiro-caixa-fechada ()
  "Retorna um tabuleiro 2x2 (2 arcos na vertical por 2 arcos na horizontal)"
  '(
    ((1)(1))
    ((1)(1))
  )
)

```

Funções seletoras e auxiliares

Estão também implementadas algumas funções seletoras e auxiliares, tais como:

```
;; ===== SELETORES =====

(defun get-arcos-horizontais (tabuleiro)
  "Retorna a lista dos arcos horizontais de um tabuleiro."
  (car tabuleiro)
)

(defun get-arcos-verticais (tabuleiro)
  "Retorna a lista dos arcos verticais de um tabuleiro."
  (car(cdr tabuleiro))
)

(defun get-arco-na-posicao (nLista pos listaArcos)
  "Função que retorna o arco que se encontra numa posicao da lista de arcos horizontais ou verticais.
  (começa no 0 o index)"
  (if (or (< nLista 0) (< pos 0))
      NIL
      (nth pos (nth nLista listaArcos)))
)

;; ===== AUXILIARES =====

(defun substituir (index arcsList &optional (x 1))
  "Função que recebe um índice (começa no 1), uma lista e valor x e deverá substituir o elemento nessa
  posição pelo valor x"
  (cond
    ((= (- index 1) 0) (cons x (cdr arcsList)))

    (T (cons (car arcsList) (substituir (- index 1) (cdr arcsList) x)))
  )
)

(defun arco-na-posicao (listPos arcPos arcsList &optional (x 1))
  "Insere um arco numa lista que representa o conjunto de arcos horizontais ou verticais de um tabuleiro.
  (Começa no indice 1)"
  (cond
    ( (= listPos 1) (cons (substituir arcPos (nth (- listPos 1) arcsList) x) (cdr arcsList)))

    (T (cons (car arcsList) (arco-na-posicao (- listPos 1) arcPos (cdr arcsList) x)) )
  )
)

(defun count-colunas (tabuleiro)
  "Contagem de colunas do tabuleiro"
  (length (car (get-arcos-horizontais tabuleiro)))
)

(defun count-linhas (tabuleiro)
  "Contagem de linhas do tabuleiro"
  (length (get-arcos-horizontais tabuleiro))
)
```

Funções operadores

Estão também implementadas funções que permitem inserir arcos no tabuleiro, verificar se existem caixas fechadas e calcular o número das mesmas.


```

(defun arco-horizontal (listPos arcPos tabuleiro &optional (x 1))
  "Função que recebe dois índices e o tabuleiro e coloca um arco horizontal nessa posição.(Começa no índice 1)"
  (cond
    ( (> listPos (length (get-arcos-horizontais tabuleiro)) ) NIL)
    ( (> arcPos (length (car (get-arcos-horizontais tabuleiro))) ) NIL)
    ( (= (get-arco-na-posicao (1- listPos) (1- arcPos) (get-arcos-horizontais tabuleiro)) 1) NIL)
    (T
      (list (arco-na-posicao listPos arcPos (get-arcos-horizontais tabuleiro) x)
            (get-arcos-verticais tabuleiro))
      )
    )
  )

(defun arco-vertical (arcPos listPos tabuleiro &optional (x 1))
  "Função que recebe dois índices e o tabuleiro e coloca um arco vertical nessa posição.(Começa no índice 1)"
  (cond
    ( (> listPos (length (get-arcos-verticais tabuleiro)) ) NIL)
    ( (> arcPos (length (car (get-arcos-verticais tabuleiro))) ) NIL)
    ( (= (get-arco-na-posicao (1- listPos) (1- arcPos) (get-arcos-verticais tabuleiro)) 1) NIL)
    (T
      (list (get-arcos-horizontais tabuleiro)
            (arco-na-posicao listPos arcPos (get-arcos-verticais tabuleiro) x))
      )
    )
  )

(defun existe-caixa-fechada (linha coluna tabuleiro)
  "Verifica num determinado arco com as suas coordenadas, se existe uma caixa fechada num tabuleiro"

  (and
    (=
      (if (not (get-arco-na-posicao linha coluna (get-arcos-horizontais tabuleiro)))
        0 (get-arco-na-posicao linha coluna (get-arcos-horizontais tabuleiro)))
      ) 1
    )
    (=
      (if (not (get-arco-na-posicao (1+ linha) coluna (get-arcos-horizontais tabuleiro)))
        0 (get-arco-na-posicao (1+ linha) coluna (get-arcos-horizontais tabuleiro)))
      ) 1
    )
    (=
      (if (not (get-arco-na-posicao coluna linha (get-arcos-verticais tabuleiro)))
        0 (get-arco-na-posicao coluna linha (get-arcos-verticais tabuleiro)))
      ) 1
    )
    (=
      (if (not (get-arco-na-posicao (1+ coluna) linha (get-arcos-verticais tabuleiro)))
        0 (get-arco-na-posicao (1+ coluna) linha (get-arcos-verticais tabuleiro)))
      ) 1
    )
  )

(defun calcular-caixas-fechadas (tabuleiro &optional (linha 0) (col 0))
  "Devolve o numero de caixas fechadas num tabuleiro. (começa no index 0)"
  (cond
    ( (>= col (count-colunas tabuleiro)) (calcular-caixas-fechadas tabuleiro (1+ linha)))
    ( (>= linha (count-linhas tabuleiro)) 0)
    (T
      (+
        (if (existe-caixa-fechada linha col tabuleiro) 1 0)

        (calcular-caixas-fechadas tabuleiro linha (1+ col))
      )
    )
  )

```

```
)
)
)
)
```

procura.lisp

Aqui estão as funções relativas à resolução do problema em si: funções algorítmicas, funções de manipulação nós e suas auxiliares que efetuam a procura da solução no tabuleiro.

A estrutura utilizada para representar o estado atual do tabuleiro é a seguinte:

```
< no >::= (< tabuleiro > < pai > < caixas-objetivo > < g > < h >)
```

Sendo que o < tabuleiro > representa o estado atual do tabuleiro, < pai > o nó antecessor, < caixas-objetivo > o número de caixas fechadas a atingir, < g > a profundidade e < h > o valor da heurística do nó.

Funções-algoritmo

Funções de procura a serem utilizadas nos tabuleiros de modo a encontrar a solução.

```
;; Função-algoritmo BFS (Breadth-First-Search)
(defun bfs (fnExpandir abertos &optional (fechados '()))
  "Algoritmo de procura em largura primeiro: Breadth-First-Search."
  (cond
    ((= (length abertos) 0) NIL)
    (T
     (let*
       (
        (no-atual (car abertos))
        (sucessores (funcall fnExpandir no-atual))
        ;;verificar se ha solucao
        (cond
          (
            (or
             (= (- (get-no-objetivo no-atual) (calcular-caixas-fechadas (get-no-estado
no-atual))) 0)
             (= (length sucessores) 0)
            )
            (list (get-caminho-solucao no-atual) (length abertos) (length fechados))
          )
          (T
           (bfs
            fnExpandir
            (append (cdr abertos) (remover-nil (remover-duplicados (remover-duplicados
sucessores abertos) fechados)))
            (append fechados (list no-atual))
           )
          )
        )
      )
    )
  )

;; Função-algoritmo DFS (Depth-First-Search)
(defun dfs (fnExpandir maxProfundidade abertos &optional (fechados '()))
  "Algoritmo de procura em profundidade primeiro: Depth-First-Search."
  (cond
    ((= (length abertos) 0) NIL)
```

```

    ( (> (get-no-g (car abertos) ) maxProfundidade) (dfs fnExpandir maxProfundidade (cdr abertos)
(append fechados (list (car abertos)))))
    (T
      (let*
        (
          (no-atual (car abertos))
          (sucessores (funcall fnExpandir no-atual))
        )
        (cond
          (
            (or
              (= (- (get-no-objetivo no-atual) (calcular-caixas-fechadas (get-no-estado
no-atual))) 0)
              (= (length sucessores) 0)
            )
            (list (get-caminho-solucao no-atual) (length abertos) (length fechados))
          )
          (T
            (dfs fnExpandir maxProfundidade (append sucessores (cdr abertos)) (append
fechados (list no-atual)))
          )
        )
      )
    )
  )
)

;; Função-algoritmo A*
(defun a* (fnExpandir fnHeuristica abertos &optional (fechados '()) (numeroExpandidos 0))
  "Algoritmo A*"
  (cond
    ((= (length abertos) 0) NIL)
    (T
      (let*
        (
          (no-atual (substituir '5 (get-f-mais-baixo abertos) (funcall fnHeuristica (get-f-
mais-baixo abertos))) )
          (sucessores (funcall fnExpandir no-atual fnHeuristica))
          (novos-fechados (recalcular-fechados fechados sucessores no-atual) ) ;;
recalcular f dos abertos
          (novos-abertos (recalcular-abertos (cdr abertos) sucessores no-atual) ) ;;
recalcular f dos fechados
          (abertos-com-novos-fechados (remove-nil (append novos-abertos (remove-duplicados
sucessores novos-abertos) novos-fechados))) ;;passar os novos fechados para abertos
        )
        (if (verificar-solucao no-atual sucessores)
          (list (get-caminho-solucao no-atual) (length abertos) (length fechados)
numeroExpandidos)
          (a* fnExpandir fnHeuristica abertos-com-novos-fechados (remove-duplicados (append
fechados (list no-atual)) novos-fechados) (1+ numeroExpandidos) )
        )
      )
    )
  )
)

```

Heurísticas

Heurística base (fornecida no enunciado)

```
(defun heuristica-base (no)
  "Heurística dada no enunciado:  $h(x) = o(x) - c(x)$  :  $o(x)$ : objetivo de caixas do tabuleiro,  $c(x)$ : numero de caixas fechadas"
  (- (get-no-objetivo no) (calcular-caixas-fechadas (get-no-estado no)) )
)
```

Funções nó

Funções que permitem criar uma nova estrutura de um nó e manipulá-la.

```
;; Função que cria um novo nó
(defun criar-no (tabuleiro pai caixas-objetivo &optional (g 0) (h 0))
  "Constroi a estrutura do no."
  (list tabuleiro pai caixas-objetivo g h)
)

;; Função que devolve o estado atual do tabuleiro
(defun get-no-estado (no)
  "Devolve o estado (tabuleiro) de um no."
  (car no)
)

;; Função que devolve o nó antecessor do nó atual
(defun get-no-pai (no)
  "Devolve o no pai deste no."
  (cadr no)
)

;; Função que devolve o número-objetivo de caixas fechadas
(defun get-no-objetivo (no)
  "Devolve o numero de caixas fechadas deste estado."
  (nth 2 no)
)

;; Função que devolve a profundidade do nó
(defun get-no-g (no)
  "Devolve o g (profundidade) de um no."
  (nth 3 no)
)

;; Função que devolve a heurística de um nó
(defun get-no-h (no)
  "Devolve a heurística de um no."
  (nth 4 no)
)

;; Função que devolve a função avaliação de um nó
(defun get-no-f (no)
  "Calcula o valor de f (funcao avaliacao) de um no."
  (+ (get-no-g no) (get-no-h no))
)
```

Funções que permitem gerar os sucessores de um nó

```
(defun gerar-nos-horizontal (no &optional (linha 1) (col 1))
  "Devolve os sucessores de um no, da parte horizontal do tabuleiro. (Começa no index 1)"
  (cond
    ((> col (count-colunas (get-no-estado no))) (gerar-nos-horizontal no (1+ linha)))
    ((> linha (count-linhas (get-no-estado no))) '())
    ((= (get-arco-na-posicao (1- linha) (1- col)) (get-arcos-horizontais (get-no-estado no)) ) 1)
```

```

(gerar-nos-horizontal no linha (1+ col)))

    (T
      (cons
        (criar-no (arco-horizontal linha col (get-no-estado no)) no (get-no-objetivo no) (1+
(get-no-g no)))
        (gerar-nos-horizontal no linha (1+ col)))
      )
    )
  )
)

(defun gerar-nos-vertical (no &optional (linha 1) (col 1))
  "Devolve os sucessores de um no, da parte vertical do tabuleiro. (Começa no index 1)"
  (cond
    ( (> col (count-colunas (get-no-estado no))) (gerar-nos-vertical no (1+ linha)))
    ( (> linha (count-linhas (get-no-estado no))) '())
    ( (= (get-arco-na-posicao (1- linha) (1- col) (get-arcos-verticais (get-no-estado no) )) 1)
      (gerar-nos-vertical no linha (1+ col)))
  )

  (T
    (cons
      (criar-no (arco-vertical col linha (get-no-estado no)) no (get-no-objetivo no) (1+ (get-
no-g no)))
      (gerar-nos-vertical no linha (1+ col)))
    )
  )
)

(defun expandir-no (no)
  "Expande um no e devolve os seus sucessores."
  (append (gerar-nos-horizontal no) (gerar-nos-vertical no))
)

```

Função que permite obter o caminho solução

```

(defun get-caminho-solucao (no)
  "Devolve uma lista de estados do no inicial ate ao no da solucao."
  (cond
    ( (null (get-no-pai no)) (list (get-no-estado no)))
    (T
      (append (get-caminho-solucao (get-no-pai no)) (list (get-no-estado no)) )
    )
  )
)

```

Funções que permitem obter medidas de desempenho da solução

```

;; <lista>::= (<caminho-solucao> <n-abertos> <n-fechados>)

;; Fator de ramificação média com recurso ao método da bissecção
(defun fator-ramificacao-media (lista &optional (L (tamanho-solucao lista)) (valor-T (num-nos-gerados
lista)) (B-min 0) (B-max valor-T) (margem 0.1))
  "Retorna o fator de ramificacao media (metodo bisseccao)"
  (let ((B-avg (/ (+ B-min B-max) 2)))
    (cond ((< (- B-max B-min) margem) (/ (+ B-max B-min) 2))
          ((< (aux-ramificacao B-avg L valor-T) 0) (fator-ramificacao-media lista L valor-T B-avg B-
max margem))
          (T (fator-ramificacao-media lista L valor-T B-min B-avg margem)))
  )
)

```

```

)

;; B + B^2 + ... + B^L = T
(defun aux-ramificacao (B L valor-T)
  (cond
    ((= 1 L) (- B valor-T))
    (T (+ (expt B L) (aux-ramificacao B (- L 1) valor-T)))
  )
)

(defun tamanho-solucao (lista)
  "Retorna o tamanho da solucao"
  (length (car lista))
)

(defun num-nos-gerados (lista)
  "Retorna o numero de nos gerados"
  (+ (second lista) (third lista))
)

(defun num-nos-expandidos (lista)
  "Retorna o numero de nos expandidos"
  (third lista)
)

(defun num-nos-expandidos-a* (lista)
  "Retorna o numero de nos expandidos (a*)"
  (fourth lista)
)

(defun penetrancia (lista)
  "Calcula a penetrancia"
  (/ (length (car lista)) (num-nos-gerados lista))
)

(defun no-solucao (lista)
  "Retorna o no solucao"
  (nth (1- (length (car lista))) (car lista))
)

```

Resultados

Os resultados a seguir apresentados calculados com o algoritmo A* utilizaram a heurística base fornecida no enunciado.

Tabuleiro	Algoritmo	Objetivo	Fator de Ramificação Média	Nós gerados	Nós expandidos	Profundidade máxima	Penetrância	Início	Fim
A	BFS	3	6.9085693	385	68	N.A.	0.007792208	18:20:11	18:20:11
A	DFS	3	1.139375	96	10	1000	0.114583336	18:21:10	18:21:10
A	A*	3	2.6523437	28	2	N.A.	0.10714286	22:23:13	22:23:13
B	BFS	7	10.458984	120	14	N.A.	0.016666668	18:29:13	18:29:13
B	DFS	7	1.5356445	85	7	1000	0.09411765	18:29:34	18:29:34
B	A*	7	3.53125	16	1	N.A.	0.125	22:34:11	22:34:11
C	BFS	10	O.M.	-	-	-	-	-	-
C	DFS	10	1.3051758	162	14	1000	0.09259259	18:37:24	18:37:24
C	A*	10	1.4277344	136	108	N.A.	0.080882356	22:35:10	22:35:10

Tabuleiro	Algoritmo	Objetivo	Fator de Ramificação Média	Nós gerados	Nós expandidos	Profundidade máxima	Penetrância	Início	Fim
D	BFS	10	O.M.	-	-	-	-	-	-
D	DFS	10	1.0974121	1240	42	1000	0.03467742	18:45:6	18:45:6
D	A*	10	O.M.	-	-	-	-	-	-
E	BFS	20	O.M.	-	-	-	-	-	-
E	DFS	20	1.1174316	796	30	1000	0.038944725	22:38:8	22:38:8
E	A*	20	1.343811	537	16	N.A.	0.031657357	22:39:12	22:39:12
F	BFS	35	O.M.	-	-	-	-	-	-
F	DFS	35	1.1174316	796	30	1000	0.038944725	18:51:0	18:51:0
F	A*	35	O.M.	-	-	-	-	-	-

N.A. - não aplicável

O.M. - out of memory (**stack overflow**)

Com estes resultados podemos ver que a aplicação mostra uma solução possível para cada tabuleiro, excepto onde estes apresentam uma exceção *stack overflow* devido às limitações do *LispWorks* / *clisp*. Na maioria dos casos, a solução é apresentada num tempo de execução bastante reduzido e inferior a 1 segundo.

A heurística que nos foi fornecida não é eficiente para ser utilizada no algoritmo A * pois faz com que este se comporte como o algoritmo *BFS*.

Limitações

A principal limitação é a falta de memória do *LispWorks* / *clisp* que impossibilita a resolução, através do algoritmo *BFS* e A * (com a heurística que nos foi fornecida), dos tabuleiros maiores com poucos ou nenhuns arcos preenchidos por defeito.

Outra limitação é o utilizador não poder introduzir um número maior de caixas a fechar do que aquele proposto no enunciado para cada tabuleiro.

Todos os requisitos base enumerados no enunciado foram implementados à exceção de:

- Heurística personalizada
- Bónus
 1. Implementação da estratégia SMA*;
 2. Implementação da estratégia IDA*;
 3. Implementação da estratégia RBFS.