Manual Técnico

Realizado por

- José Pereira nº 150221044
- Lyudmyla Todoriko nº 150221059

Docentes

- Prof. Hugo Silva
- Prof. Joaquim Silva
- Eng. Filipe Mariano

Para a cadeira de Inteligência Artificial

1. Introdução

No âmbito da cadeira de Inteligência Artificial foi nos proposto a criação de toda a lógica do jogo Blokus e de algoritmos para a sua resolução. Uma vez que estamos a trabalhar numa versão limitada do LispWorks e em contexto académico foi nos solicitado utilizar uma versão mais pequena do Blokus apelidada de Blokus Uno, contendo menos peças, um tabuleiro mais pequeno e apenas para um jogador. Os algoritmos que foram utilizados e estudados foram o BFS, DFS, A* e IDA*, com duas heurísticas distintas com objetivos diferentes.

2. Lógica do Puzzle

2.1 Tabuleiro

Uma vez que vamos trabalhar sobre um jogo de tabuleiro será preciso ter funções básicas para manipular o mesmo, seja para selecionar uma linha ou para alterar uma célula.

Para facilitar a obtenção de valores existentes no tabuleiro utilizamos funções para obter linhas, colunas e células baseadas no índice introduzido.

```
(defun line (index board)
  (cond ((not (numberp index)) nil)
        ((< index 0) nil)
        (t (nth index board))))

(defun column (index board)
  (cond ((not (numberp index)) nil)
        ((< index 0) nil)
        (t (mapcar #'(lambda (line &aux (n-column (nth index line))) n-column) board))))

(defun board-cell (x y board)
  (cond ((or (not (numberp x)) (not (numberp y))) nil)
        ((or (< x 0) (< y 0)) nil)
        (t (nth x (line y board)))))

(defun empty-cellp (x y board)
  "Verifies if the x,y cell is empty (contains value 0)"
  (cond ((or (not (numberp x)) (not (numberp y))) nil)
        ((or (< x 0) (< y 0)) nil)
        ((not (eq (board-cell x y board) 0)) nil)
        (t t)))
```

Para podermos visualizar o problema e ser mais fácil de testar criamos uma função que imprime o tabuleiro por linhas, facilitando a sua leitura e mais tarde escrita.

```
(defun board-print (board)
  "Prints board in a easier way to read, in a grid with the board size"
  (cond ((null board) nil)
        ((eq (length board) 1) (format t "~d~%~%" (car board)))
        (t (format t "~d~%" (car board)) (board-print (cdr board)))))
```

Para podermos trocar os valores que existem no tabuleiro e recebermos de volta o tabuleiro alterado foi necessário algumas funções extra.

```
(defun replace-position (index board &optional (value 1))
  "Replace a line in the index of a board"
  (cond ((or(null board) (not (numberp index))) nil)
        ((or (< index 0) (> index (length board))) nil)
        ((= index 0) (cons value (cdr board)))
        (t (cons (car board) (replace-position (- index 1) (cdr board) value)))))

(defun replace-board (x y board &optional (value 1))
  (replace-position y board (replace-position x (line y board) value)))
```

2.2 Nós

Para ser mais fácil a gestão do problema e respetivos cálculos criamos um formato para os nós do problema que consistem num estado, o nó pai, o custo do nó e quando necessário o valor de h e o da função f. Adicionalmente o estado do nosso nó vai conter dois elementos, o tabuleiro e uma lista com as peças restantes no formato de (square-1x1 square-2x2 cross).

Tal como no caso do tabuleiro, foi necessário criar funções básicas de manipulação do nó para receber cada elemento do mesmo e a sua criação.

```
(defun node-create (state parent d g h f)
  (list state parent d g h f))

(defun node-state (node)
  (car node))

(defun node-board (node-state)
  (car node-state))

(defun node-pieces (node-state)
  (second node-state))

(defun node-parent (node)
  (cadr node))

(defun node-depth (node)
  (caddr node))

(defun node-cost (node)
  (cadddr node))


(defun node-h (node)
```

```
  (nth 4 node))

(defun node-f (node)
  (nth 5 node))
```

Para visualizar o nó e facilitar o trabalho de leitura e escrita foi criada também uma função para imprimir o nó e a sua informação necessária

```
(defun node-print (node)
  "Prints node board, pieces remaining, price, h falue and f value"
  (cond ((null node) nil)
        (t (format t "Original:~%")
           (board-print (node-board (node-state (node-original node))))
           (format t "Final:~%")
           (board-print (node-board (node-state node)))
           (format t "~%Pieces: ~d~%Depth:~d~%Cost:~d~%F=~d~%H=~d~%" (node-pieces (node-state
node)) (node-depth node) (node-cost node) (node-f node) (node-h node)))))
```

E para podermos estudar a solução obtida é util saber qual foi o nó de origem e o tamanho da solução existente.

```
(defun node-solution-size (node)
  "Calculates the total solution size of a node (without counting with the original root)"
    (cond ((null (node-parent node)) 0)
          (t (+ 1 (node-solution-size (node-parent node))))))

(defun node-original (node)
  "Gets the node original root"
    (cond ((null (node-parent node)) node)
          (t (node-original (node-parent node)))))
```

Por fim também é util sabermos que jogadas foram tomadas a cada movimento realizado.

```
(defun calculate-made-step (current-position parent-position)
  "Calculates step taken between states"
  (cond ((null parent-position))
        ((< (nth 0 current-position) (nth 0 parent-position)) "Colocou uma peça 1x1")
        ((< (nth 1 current-position) (nth 1 parent-position)) "Colocou uma peça 2x2")
        ((< (nth 2 current-position) (nth 2 parent-position)) "Colocou uma peça cruz")))
```

2.3 Operadores

Tal como foi lecionado na cadeira, partimos o problema nos seus diversos operadores, que na versão pedida é a colocação de um dos três tipos de peças disponíveis (quadrado-1x1, quadrado-2x2 e cruz) numa determinada posição do tabuleiro.

```
(defun operators ()

  "Possible operators to use on Blokus"
```

```
  '(CROSS SQUARE-2X2 SQUARE-1X1))

(defun place-square (x y board)
  "Places a 1x1 square on the given x and y of a board"
  (cond ((verify-empty-cells board (block-occupied-cells x y 'square-1x1))
         (replace-board x y board))
        (t nil)))

(defun update-pieces (pieces type)
  "Subtracts one from the type of piece in the list, returning nil if it wasn't possible"
  (cond ((eq type 'square-1x1) (list (1- (first pieces)) (second pieces) (third pieces)))
        ((eq type 'square-2x2) (list (first pieces) (1- (second pieces)) (third pieces)))
        ((eq type 'cross) (list (first pieces) (second pieces) (1- (third pieces))))
        (t (print "SOMETHING WENT WRONG") nil)))

(defun square-1x1 (x y node)
  "Places a 1x1 square on the board if it is possible and updating the existing pieces on the
node"
  (let ((pieces (node-pieces (node-state node))))
    (cond ((eq (first pieces) 0) nil)
          (t (list (place-square x y (node-board (node-state node)))
                   (update-pieces pieces 'square-1x1))))))

(defun square-2x2(x y node)
  "Places a 2x2 square on the board if it is possible and updating the existing pieces on the
node"
         (labels ((square-aux (x y board cells)
                  (if (null cells) (place-square x y board)
                    (square-aux (first (first cells)) (second (first cells)) (place-square x y
board) (cdr cells)))))
           (let ((pieces (node-pieces (node-state node))))
             (cond ((eq (second pieces) 0) nil)
                   (t (list (square-aux x y (node-board (node-state node)) (block-occupied-cells
x y 'square-2x2))
                            (update-pieces pieces 'square-2x2)))))))

(defun cross (x y node)
  "Places a cross (+) on the board if it is possible and updating the existing pieces on the
node"
         (labels ((cross-aux (x y board cells)
                  (if (null cells) (place-square x y board)
                    (cross-aux (first (first cells)) (second (first cells)) (place-square x y
board) (cdr cells)))))
           (let ((pieces (node-pieces (node-state node))))
             (cond ((eq (third pieces) 0) nil)
                   (t (list (cross-aux (1+ x) (1+ y) (node-board (node-state node)) (block-
occupied-cells x y 'cross))
                            (update-pieces pieces 'cross)))))))
```

No entanto é preciso ter em contas algumas regras:

1. As peças só podem ser colocadas diagonalmente entre si;

2. As peças nunca podem ser colocadas imediatamente adjacente a outra peça do jogador (mas podem estar adjacentes a uma peça já existente);
3. Caso não exista mais nenhuma peça em campo do jogador, a peça apenas pode ser colocada num dos cantos disponíveis;
4. Uma peça não pode ser colocada por cima de outra.

Para validar todos estes casos foi preciso uma longa lista de condições, garantindo o bom funcionamento dos nossos operadores.

```
(defun verify-empty-cells (board positions)
  "Verifies if all the given positions are empty"
  (mapcar #' (lambda (board-cell) (empty-cellp (first board-cell) (second board-cell) board))
positions))

(defun empty-positions (board positions)
  "Returns the positions that are empty from the given position list"
  (apply #'append
  (mapcar #' (lambda (board-cell &aux (empty (empty-cellp (first board-cell) (second board-cell)
board)))
               (if (and empty (cell-inbounds (first board-cell) (second board-cell) board))
(list board-cell))) positions)))

(defun block-occupied-cells (x y block-type)
  "Calculates the coordinates (x y) that will be occupied by a given block type on a give x and
y"
  (cond ((eq block-type 'square-1x1) (list (list x y)))
        ((eq block-type 'square-2x2) (list (list x y) (list x (+ y 1))(list (+ 1 x) y) (list (+
x 1) (+ y 1))))
        ((eq block-type 'cross) (list (list x (+ y 1)) (list (+ x 1) (+ y 1)) (list (+ x 2) (+ y
1))(list (+ x 1) y) (list (+ x 1)(+ y 2))))
        (t nil)))

(defun cell-inbounds (x y board)
  "Verifies if a cell is in a valid board position (between 0 and maximum line length"
  (cond ((and (and (>= y 0) (<= y (1- (length (line 0 board))))) (and (>= x 0) (<= x (1- (length
(column 0 board)))))) t)
        (t nil)))

(defun possible-diagonals (x y board block-type)
  "Calculates all the possible diagonals from a specific block on a given x and y"
  (cond ((eq block-type 'square-1x1) (empty-positions board (list (list (1- x) (1- y)) (list (1-
x) (1+ y))
                                                                  (list (1+ x) (1- y)) (list (1+
x) (1+ y)))))
        ((eq block-type 'square-2x2) (empty-positions board (list (list (- x 2) (- y 2)) (list
(+ x 1) (- y 2))
                                                                  (list (- x 2) (+ y 1)) (list
(+ x 1) (+ y 1)))))
        ((eq block-type 'cross) (empty-positions board (list (list (- x 3) (- y 2)) (list x (- y
3)) (list (+ x 1) y)
                                                               (list (- x 2) (+ y 1)) (list (- x
3) y)
                                                               (list x (+ y 1)) (list x (+ y 1))
```

```
(list (+ x 1) (- y 2))
                                                        (list (- x 2) (- y 3)))))))

(defun not-adjacent-pos (x y board block-type)
  "Verifices if there is any adjacent player piece on a given x and y"
  (cond ((and (eq block-type 'square-1x1)
              (not (eq (board-cell x (1- y) board) 1))
              (not (eq (board-cell x (1+ y) board) 1))
              (not (eq (board-cell (1- x) y board) 1))
              (not (eq (board-cell (1+ x) y board) 1))) t)
        ((and (eq block-type 'square-2x2)
              (not (eq (board-cell x (- y 1) board) 1))
              (not (eq (board-cell (+ x 1) (- y 1)board) 1)) ;;
              (not (eq (board-cell (+ x 2) y board) 1)) ;;
              (not (eq (board-cell (+ x 2) (+ y 1) board)1))
              (not (eq (board-cell (+ x 1) (+ y 2) board) 1))
              (not (eq (board-cell x (+ y 2) board) 1))
              (not (eq (board-cell (- x 1) (+ y 1) board) 1)) ;;
              (not (eq (board-cell (- x 1) y board) 1))
              (eq (length (empty-positions board (block-occupied-cells x y block-type)))
                  (length (block-occupied-cells x y block-type)))) t)
        ((and (eq block-type 'cross)
              (not (eq (board-cell x y board) 1))
              (not (eq (board-cell (+ x 1) (- y 1) board) 1))
              (not (eq (board-cell (+ x 2) y board) 1))
              (not (eq (board-cell (- x 1) (+ y 1) board) 1))
              (not (eq (board-cell (+ x 3) (+ y 1) board) 1))
              (not (eq (board-cell x (+ y 2) board) 1))
              (not (eq (board-cell (+ x 2) (+ y 2) board) 1))
              (not (eq (board-cell (+ x 1) (+ y 3) board) 1))
              (eq (length (empty-positions board (block-occupied-cells x y block-type)))
                  (length (block-occupied-cells x y block-type)))) t)))

(defun valid-diagonals (diagonal-positions board block-type)
  "Returns the valid diagonals"
  (apply #'append(mapcar #'(lambda (position) (cond ((not-adjacent-pos (first position) (second
position) board block-type) (list position)))) diagonal-positions)))


(defun possible-block-positions (board block-type)
  "Returns all the possible position to place a block type on the board, following the game
rules"
  (labels ((possible-pos-aux (x y board)
             (cond ((= x 14) (possible-pos-aux 0 (1+ y) board))
                   ((= y 14) nil)
                   ((eq (board-cell x y board) 1) (append (list (valid-diagonals (possible-
diagonals x y board block-type) board block-type)) (possible-pos-aux (1+ x) y board)))
                   (t (possible-pos-aux (1+ x) y board)))))
    (cond ((empty-boardp board) (valid-corner board block-type))
          (t (remove-duplicates (apply #'append (possible-pos-aux 0 0 board)) :test #'equal-
coords)))))


(defun valid-corner (board block-type)
```

```
  (cond ((eq block-type 'square-1x1) (empty-positions board '((0 0) (0 13) (13 0) (13 13))))
        ((eq block-type 'square-2x2) (apply #'append (mapcar #'(lambda (pos &aux (x (first pos))
(y (second pos)))
                                                                 (if (= (length (empty-positions
board (block-occupied-cells x y 'square-2x2))) 4) (list (list x y))))
                                               '((0 0) (0 12) (12 0) (12 12)))))
        (t nil)))

(defun equal-coords (coorda coordb)
  (and (= (car coorda) (car coordb)) (= (cadr coorda) (cadr coordb))))

;;Checks if there is any "1" piece on the board
(defun empty-boardp (board)
  "Verifies that the board is empty"
  (labels ((possible-pos-aux (x y board)
             (cond ((= x 14) (possible-pos-aux 0 (1+ y) board))
                   ((= y 14) t)
                   ((eq (board-cell x y board) 1) nil )
                   (t (possible-pos-aux (1+ x) y board)))))
    (possible-pos-aux 0 0 board)))
```

2.4 Expansão

Depois de termos a nossa estrutura de nós, tabuleiro e operadores a funcionar e a respeitar as regras propostas precisamos de preparar as funções que serão usadas na exploração do nosso problema pelos algoritmos de procura.

```
(defun solution-nodep (node)
  "Verifies if a node is a solution node"
  (cond ((equal (node-pieces node) '(0 0 0)) t)
        ((null (node-expandp node)) t)
        (t nil)))

(defun node-expand (node operators search &optional (d 0))
  "Expands a node based on given operators and search string, only used for bfs and dfs"
  (labels ((place-nodes (node operation positions)
             (cond ((null positions) nil)
                   ;Check if there was any problem or if out of pieces
                   ((null (funcall operation (first (car positions)) (second (car positions))
node))
                    (place-nodes node operation (cdr positions)))
                   ((and (eq search 'dfs) (= (node-depth node) d)) nil)
                   (t (cons (node-create
                             (funcall operation (first (car positions)) (second (car positions))
node)
                             node (1+ (node-depth node)) 0 0 0)
                            (place-nodes node operation (cdr positions)))))))
    (flet ((expand-node (node operation)
             (place-nodes node operation (possible-block-positions (node-board (node-state
node)) operation))))
      (apply #'append (mapcar #'(lambda(operation) (expand-node node operation)) operators)))))
```

```
(defun create-node-from-state (state parent h)
  "Used to create a node based on a specific state, used to shorten the code on node-expand-a"
            (let ((g (1+ (node-cost parent))) (h-v (funcall h state)))
              (node-create state parent (1+ (node-depth parent)) g h-v (+ g h-v))))

(defun node-expand-a (node operators h)
  "Expands a node based on given operators and heuristic, used by A* and IDA*"
  (labels ((place-nodes (node operation positions)
             (cond ((null positions) nil)
                   ;Check if there was any problem or if out of pieces
                   ((null (funcall operation (first (car positions)) (second (car positions))
node))
                    (place-nodes node operation (cdr positions)))
                   (t (cons (create-node-from-state
                             (funcall operation (first (car positions)) (second (car positions))
node) node h)
                            (place-nodes node operation (cdr positions)))))))
    (flet ((expand-node (node operation)
             (place-nodes node operation (possible-block-positions (node-board (node-state
node)) operation))))
      (apply #'append (mapcar #'(lambda(operation) (expand-node node operation)) operators)))))

(defun node-expandp (node)
  "Faster way to confirm is a node can expand, verifying if there is no possible position to go
to"
  (labels ((place-nodes (node operation positions)
             (cond ((null positions) nil)
                   ((null (funcall operation (first (car positions)) (second (car positions))
node))
                    (place-nodes node operation (cdr positions)))
                   (t '(t)))))
    (flet ((expand-node (node operation)
             (place-nodes node operation (possible-block-positions (node-board (node-state
node)) operation))
             ))
      (apply #'append (mapcar #'(lambda(operation) (expand-node node operation)) '(square-1x1
square-2x2 cross))))))
```

3. Procura

Abstrato da lógica do problema proposto, foi também solicitado a implementação de diversos algoritmos de procura para conseguirem resolver os tabuleiros (mas mantendo-se abstratos) e o respetivo estudo dos resultados obtidos, tais como o tempo necessário para resolver o tabuleiro, nós gerados, nós explorados, penetrância e ramificação média

3.1 Breadth-first search (BFS)

O BFS, tal como o seu nome indica, procura numa árvore por largura, percorrendo todos os níveis até encontrar um nó que seja solução. Uma vez que não é um algoritmo de procura informado, não garante a solução ótima e como no pior dos casos pode explorar todos os nós existentes, este torna-se muito lento, pouco eficaz e pouco eficiente.

Este foi implementado da seguinte forma (iterativa).

```
(defun open-bfs (open child)
  (append open child))

(defun filter-nodes (node-list open-list close-list)
  (cond ((null node-list) nil)
        ((or (node-existsp (car node-list) open-list) (node-existsp (car node-list) close-list))
(filter-nodes (cdr node-list) open-list close-list))
        (t (cons (car node-list) (filter-nodes (cdr node-list) open-list close-list)))))

(defun bfs-it (node solution expand operators &optional (d nil))
  "Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data
structures.
   It starts at the tree root and explores the neighbor nodes first, before moving to the next
level neighbours."
  (start-performance)
  (setq *open* (list node))
  (setq *close* nil)
  (loop while (not (null *open*)) do
        (let* ((current-node (car *open*)) (expanded-nodes (filter-nodes (funcall expand
current-node operators 'bfs d) *open* *close*)))
          (add-explored 1)
          (add-generate (length expanded-nodes))
          ;Add currentNode to closed list
          (setq *close* (append *close* (list current-node)))
          ;Remove current node from open
          (setq *open* (cdr *open*))
          ;Add expanded nodes to open
          (setq *open* (open-bfs *open* expanded-nodes))
          ;Check if a node is a possible solution and return it
          (mapcar #'(lambda (expanded-node) (cond ((funcall solution expanded-node) (stop-
performance expanded-node)(return expanded-node))))expanded-nodes))))
```

Ignorando as funções relacionadas com *performance* que serão mostradas e explicadas mais à frente, obtivemos os seguintes resultados através deste algoritmo sobre os tabuleiros fornecidos.

	A	B	C	D	E	F
Nós Gerados	6	16	47	85	N/A	N/A
Nós Explorados	1	2	9	13	N/A	N/A
Penetrância	0.17	0.13	0.06	0.04	N/A	N/A
Ramificação	6	3.531	3.221	4.017	N/A	N/A
Tempo(ms)	2	4	18	48	N/A	N/A

É necessário salientar que onde está assinalado "N/A" foram tabuleiros com mais jogadas possiveis, e uma vez que estamos a trabalhar numa versão gratuita de *LispWorks* estavamos a exceder a memória disponivel antes de encontrar uma solução consistente.

Também é importante notar que ordem de colocação de peças na nossa implementação depende da ordem em que os operadores se encontram, pois ele irá tenter colocar os operados pela ordem fornecida ao logo de todos os nós explorados.

As soluções obtidas foram, tal como esperadas, não ótimas dentro do que estavamos à procura e com uma penetrância e ramificação muito altas.

3.2 Depth-first search (DFS)

Ao contrário do algoritmo anterior, BFS, o DFS explora a árvore em profundidade, tentando sempre explorar o nó mais à esquerda possível até não ser possivel expandir mais ou chegar ao limite de profundida. No nosso caso de uso, dada uma profundidade grande o suficiente, mostrou ser mais rápido que o BFS e até consegue obter soluções, embora longe de ótimas, para todos os tabuleiros.

Este foi implementado da seguinte forma (iterativa).

```
(defun open-dfs (open child)
  (append child open))

(defun dfs-it (node solution expand operators p)
  "Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data
structures.
   One starts at the root and explores as far as possible along each branch before
backtracking."
  (start-performance)
  (setq *open* (list node))
  (setq *close* nil)
  (loop while (not (null *open*)) do
        (let* ((current-node (car *open*)) (expanded-nodes (filter-nodes (funcall expand
current-node operators 'dfs p) *open* *close*)))
          (add-explored 1)
          (add-generate (length expanded-nodes))
          ;Add currentNode to closed list
          (setq *close* (append *close* (list current-node)))
          ;Remove current node from open
          (setq *open* (cdr *open*))
          ;Add expanded nodes to open
          (setq *open* (open-dfs *open* expanded-nodes))
          ;Check if a node is a possible solution and return it
          (mapcar #'(lambda (expanded-node)
                      (cond ((funcall solution expanded-node)
                             (stop-performance expanded-node)(return expanded-node)))) expanded-
nodes))))
```

Ignorando as funções relacionadas com *performance* que serão mostradas e explicadas mais à frente, obtivemos os seguintes resultados através deste algoritmo sobre os tabuleiros fornecidos.

	A	B	C	D	E	F
Nós Gerados	6	16	30	24	83	455
Nós Explorados	1	2	9	3	14	26
Penetrância	0.17	0.13	0.10	0.13	0.17	0.06
Ramificação	6	3.531	2.712	2.485	1.218	1.824
Tempo(ms)	1	6	7	13	32	258

Neste caso já conseguimos encontrar soluções para todos os tabuleiros, dado uma profundidade que fosse grande o suficiente para explorar um ramo até ao fim, ou chegar ao ramo mais curto. Se a profundidade for reduzida entramos no problema do BFS e não é possivel encontrar uma solução através do *LispWorks* ou, se a profundidade for demasiado baixa, não é possivel encontrar uma solução de todo.

Tal como no BFS a ordem em que as peças são colocadas, e podendo afetar o resultado obtivo, depende da ordem dos operadores.

As soluções obtidas foram, tal como esperadas, não ótimas dentro do que estavamos à procura mas um pouco melhor que no BFS que não conseguiu analisar todos os tabuleiros.

3.3 Algoritmos de procura informados

Os algoritmos de procura informados usam conhecimento representado por **heurísticas** que irão "guiar" a nossa procura para uma solução ótima (se esta for admissível) e um caminho mais eficiente que o obtido nas procuras anteriores.

As heurísticas observadas foram duas heurísticas diferentes, com objetivos diferentes onde uma delas foi fornecida no projeto e a outra foi obtida por uma ideia obtida com os docentes.

3.3.1 Heurísticas

Fornecida pelo professor

A heurística fornecida originalmente utiliza a fórmula `h(x) = o(x) - c(x)`

Onde temos

- `o(x) = número de quadrados a preencher`
- `c(x) = número de quadrados preenchidos`

Sendo esta a fornecida, foi nos proposto a criação de uma heurística melhor que a mencionada. Depois de realizado um estudo sobre esta e ter sido discutido com os docentes, decidimos criar uma heurística com um objetivo diferente, uma vez que esta visava ocupar o máximo de espaço no tabuleiro possível.

```
(defun heuristic-squares (state)
  (apply '+ (apply #'append
                   (mapcar #'(lambda(line)
                               (mapcar #'(lambda(position)
                                           (cond ((= position 0) 1)
                                                 ((= position 1) -1)
                                                 (t 0))) line)) (node-board state)))))
```

A nossa alteração

Em vez de tentarmos preencher o máximo número de quadrados, decidimos que o objetivo seria acabar o tabuleiro o mais rápido possível, ou seja, chegar a uma posição onde não nos é possível realizar mais nenhuma jogada, uma vez que uma das condições de nó solução é não ser possível colocar mais peças.

Para isso usamos a fórmula `h(x) = 3a(x) + 2b(x) + c(x)`

Onde temos

- `a(x) = número de peças 1x1 que podemos colocar`
- `b(x) = número de peças 2x2 que podemos colocar`
- `c(x) = número de peças cruz que podemos colocar`
- E os respetivos pesos a usar para cada uma, que podem ser alterados para melhorar/piorar a solução em casos especificos ou em futuras alterações sobre o tabuleiro e/ou peças.

```
(defun heuristic-custom-complex (state &optional (sq1mod 3) (sq2mod 2) (crossmod 1))
  "Heuristic used to benefid boards with less possible places"
  (flet ((count-squares-aux (board operation)
           (cond ((eq operation 'square-1x1) (* (length (possible-block-positions board
operation)) sq1mod))
                 ((eq operation 'square-2x2) (* (length (possible-block-positions board
operation)) sq2mod))
                 ((eq operation 'cross) (* (length (possible-block-positions board operation))
crossmod))
                 (t 0))))
    (apply #'+ (mapcar #'(lambda (operation)
                           (count-squares-aux (node-board state) operation)) (operators)))))
```

Ao inicio ainda tentamos uma abordagem sem pesos, mas mostrou ser pouco eficaz para o nosso objetivo e não era muito mais rápida a nivel de calculos com o seguinte código:

```
(defun heuristic-custom (state)
    (length (apply #'append (mapcar #'(lambda (operation)
                           (possible-block-positions (node-board state) operation))
(operators)))))
```

Para comprovar a eficácia desta heurística perante a fornecida pelos docentes, com o objetivo de ficar sem jogadas mais rápido possível, estudamos a profundidade das soluções obtidas que se traduz no comprimento da solução e número de jogadas realizadas pelo algoritmo.

Número de Jogadas	A	B	C	D	E	F
Heurística Fornecida	2	4	5	6	17	25
Nova heurística	1	2	3	5	12	8

Como podemos observar, a nossa heurística tem uma melhoria significativa, principalmente quando o tabuleiro tem vários caminhos possíveis. Estudos sobre a *performance* das heurísticas serão mostrados no próximo capitulo.

3.3.2 A-Star Search (A*)

O A* é o nosso primeiro algoritmo de procura informada, este irá explorar a árvore através do menor custo F de um nó, obtido por `f(x) = g(x) + h(x)`, percorrendo ordenadamente, do menor para o maior, a lista de nós explorados e apenas devolvendo um nó como solução quando este for o nó aberto de menor custo na árvore. Assim garante que se for admissível, a solução encontrada é ótima.

Este foi implementado da seguinte forma (iterativa).

```
(defun filter-nodes-update-open (node-list open-list &optional (cost 'node-f))
  "Verifies if any of the duplicate nodes are better than the current nodes"
  (cond ((null node-list) open-list)
        ((node-existsp (car node-list) open-list)
         (let* ((new-node (car node-list)) (existing-node (car (member new-node open-list :test
'equal-node))))
           (if (>= (funcall cost new-node) (funcall cost existing-node))
               (filter-nodes-update-open (cdr node-list) open-list)
             (filter-nodes-update-open (cdr node-list) (substitute new-node existing-node open-
list :count 1 :test #'equal)))))
        (t (filter-nodes-update-open (cdr node-list) open-list))))

(defun ida-star (node solution expand operators heuristic cost &optional (bound (funcall cost
node)))
  "Iterative-deepening-A* works as follows: at each iteration, perform a depth-first search,
   cutting off a branch when its total cost f(n)=g(n)+h(n) exceeds a given threshold.
   This threshold starts at the estimate of the cost at the initial state, and increases for
each iteration of the algorithm.
   At each iteration, the threshold used for the next iteration is the minimum cost of all
values that exceeded the current threshold."
  (labels ((ida-star-aux (node solution expand operators heuristic cost &optional (bound
(funcall cost node)))
             (setq *open* (list node))
             (setq *close* nil)
             (loop while (not (null *open*)) do
                   ;Node cost is bigger than the current bound, start a new search with a bigger
bound
                   (if (> (funcall cost (car *open*)) bound) (return (ida-star node solution
expand operators heuristic cost (funcall cost (car *open*)))))
                   ;Nodes cost is lesser than the current bound, keep searching with the current
bound

                           (let* ((current-node (car *open*))
```

```
                                    (unfiltered-nodes (funcall expand current-node operators
heuristic))
                                    (expanded-nodes (filter-nodes unfiltered-nodes *open*
*close*)))
                              (add-explored 1)
                              (add-generate (length expanded-nodes))
                              (setq *close* (append *close* (list current-node)))
                              (cond ((funcall solution current-node) (stop-performance current-
node)(return current-node)))
                              (setq *open* (ordered-insert-list (cdr *open*) expanded-nodes))
                              (setq *open* (filter-nodes-update-open unfiltered-nodes *open*))
                              ;Failsafe
                              (stable-sort *open* #'< :key cost)))))
    (start-performance)
    ;auxiliar function to avoid restarting performance operations
    (ida-star-aux node solution expand operators heuristic cost bound)))
```

Ignorando as funções relacionadas com *performance* que serão mostradas e explicadas mais à frente, obtivemos os seguintes resultados através deste algoritmo sobre os tabuleiros fornecidos.

Heurística Docente	**A**	**B**	**C**	**D**	**E**	**F**
Nós Gerados	10	27	45	46	112	346
Nós Explorados	3	5	6	7	18	26
Penetrância	0.20	0.15	0.11	0.13	0.15	0.07
Ramificação	2.702	1.936	1.850	1.631	1.890	1.172
Tempo(ms)	2	3	7	13	16	35

Comparativamente aos algoritmos anteriores, a diferença da solução obtida é muito elevada, e em média a penetrância também é superior no A*.

Vamos agora testar o mesmo algoritmo, mas com a nova heurística.

Heurística Original	**A**	**B**	**C**	**D**	**E**	**F**
Nós Gerados	6	11	20	22	125	281
Nós Explorados	2	3	5	5	47	66
Penetrância	0.17	0.18	0.15	0.18	0.10	0.05
Ramificação	6	2.854	2.311	1.817	1.890	1.403
Tempo(ms)	3	6	9	14	80	169

A penetrância embora situacional, foi melhorada mas a ramificação aumentou juntamente com o tempo necessário para chegar a uma solução, no entanto, como demonstrado no capitulo anterior das heurísticas foi possível chegar a soluções com um número de jogadas muito inferior à heurística dos docentes, assim podemos afirmar que a heurística foi alterada e melhorada.

Sobre o algoritmo, o tempo e soluções obtidas foram muito melhores que a dos algoritmos anteriores uma vez que a procura é informada, obtendo soluções ótimas.

3.3.3 Iterative Deepening A-Star Search (IDAA*)

A procura do algoritmo A* consegue encontrar a solução ótima e num espaço tempo bastante satisfatório para o problema em questão e tendo em conta a heurística, no entanto existe outro problema que devemos considerar, a memória não é infinita.

Para colmatar a memória limitada, podemos utilizar o IDA*, posit este não carrega toda a árvore de uma só vez, utilizando limites para saltar nós em favor de nós de menor custo. Em contra partida, como não carregamos toda a árvore de uma só vez, cada vez que formos aumentar o nosso limite, iremos gerar a árvore a partir da raiz, que irá aumentar o número de nós gerados, explorados e o tempo de execução.

Este foi implementado da seguinte forma (iterativa).

```
(defun ida-star (node solution expand operators heuristic cost &optional (bound (funcall cost
node)))
  "Iterative-deepening-A* works as follows: at each iteration, perform a depth-first search,
   cutting off a branch when its total cost f(n)=g(n)+h(n) exceeds a given threshold.
   This threshold starts at the estimate of the cost at the initial state, and increases for
each iteration of the algorithm.
   At each iteration, the threshold used for the next iteration is the minimum cost of all
values that exceeded the current threshold."
  (labels ((ida-star-aux (node solution expand operators heuristic cost &optional (bound
(funcall cost node)))
             (setq *open* (list node))
             (setq *close* nil)
             (loop while (not (null *open*)) do
                   ;Node cost is bigger than the current bound, start a new search with a bigger
bound
                   (if (> (funcall cost (car *open*)) bound) (return (ida-star node solution
expand operators heuristic cost (funcall cost (car *open*)))))
                   ;Nodes cost is lesser than the current bound, keep searching with the current
bound
                           (let* ((current-node (car *open*))
                                  (unfiltered-nodes (funcall expand current-node operators
heuristic))
                                  (expanded-nodes (filter-nodes unfiltered-nodes *open*
*close*)))
                             (add-explored 1)
                             (add-generate (length expanded-nodes))
                             (setq *close* (append *close* (list current-node)))
                             (cond ((funcall solution current-node) (stop-performance current-
node)(return current-node)))
                             (setq *open* (ordered-insert-list (cdr *open*) expanded-nodes))
                             (setq *open* (filter-nodes-update-open unfiltered-nodes *open*))

                             ;Failsafe
```

```
                            (stable-sort *open* #'< :key cost)))))
    (start-performance)
    ;auxiliar function to avoid restarting performance operations
    (ida-star-aux node solution expand operators heuristic cost bound)))
```

Ignorando as funções relacionadas com *performance* que serão mostradas e explicadas mais à frente, obtivemos os seguintes resultados através deste algoritmo sobre os tabuleiros fornecidos.

Heurística Docente	**A**	**B**	**C**	**D**	**E**	**F**
Nós Gerados	10	27	45	46	112	346
Nós Explorados	3	5	6	7	18	26
Penetrância	0.20	0.15	0.11	0.13	0.15	0.07
Ramificação	2.702	1.936	1.850	1.631	1.890	1.172
Tempo(ms)	1	2	4	7	12	33

Comparativamente aos algoritmos não informados, a diferença da solução obtida é muito elevada, e em média a penetrância também é superior no IDA*.

Vamos agora testar o mesmo algoritmo, mas com a nova heurística.

Heurística Original	**A**	**B**	**C**	**D**	**E**	**F**
Nós Gerados	6	11	20	22	125	281
Nós Explorados	2	3	5	5	47	66
Penetrância	0.17	0.18	0.15	0.18	0.10	0.05
Ramificação	6	2.854	2.311	1.817	1.890	1.403
Tempo(ms)	2	4	8	14	62	140

Podemos perceber que os resultados, quer de solução quer de *performance* foram muito similares aos encontrados no A*.

A solução ser igual ao de A* é esperado, pois ambos vão obter a mesma solução dada uma heurística igual.

No entanto, neste caso não podemos comprovar um aumento de nós gerados nem explorados, pois as nossas heurísticas não têm tendência a aumentar de valor, logo a árvore não irá ser gerada mais que uma vez.

3.4 Ciclicidade

Todos os algoritmos de procura que implementamos tiveram em conta a possível ciclicidade existente nos grafos de jogo, pois existem vários caminhos para chegar a uma certa posição do tabuleiro.

No caso do BFS e DFS é simples de resolver, fazemos uma *filtragem* dos nós expandidos que existem na lista de abertos e/ou fechados e estes são descartados.

```
(defun filter-nodes (node-list open-list close-list)
  (cond ((null node-list) nil)
        ((or (node-existsp (car node-list) open-list) (node-existsp (car node-list) close-list))
(filter-nodes (cdr node-list) open-list close-list))
        (t (cons (car node-list) (filter-nodes (cdr node-list) open-list close-list)))))
```

Mas no caso do A* e IDA* foi preciso, adicionalmente à função anterior, ter em conta que podemos encontrar um nó que embora tenha um estado já existente na lista de abertos e/ou fechados, este novo nó pode ser melhor que os existentes. Nesse caso é necessário atualizar o nó que se encontra na lista de abertos pelo nó encontrado, e descartar o nó de maior custo.

```
(defun filter-nodes-update-open (node-list open-list &optional (cost 'node-f))
  "Verifies if any of the duplicate nodes are better than the current nodes"
  (cond ((null node-list) open-list)
        ((node-existsp (car node-list) open-list)
         (let* ((new-node (car node-list)) (existing-node (car (member new-node open-list :test
'equal-node))))
           (if (>= (funcall cost new-node) (funcall cost existing-node))
               (filter-nodes-update-open (cdr node-list) open-list)
             (filter-nodes-update-open (cdr node-list) (substitute new-node existing-node open-
list :count 1 :test #'equal)))))
        (t (filter-nodes-update-open (cdr node-list) open-list))))
```

Esta devolve uma cópia da lista de abertos atualizada com os nós de menor custo.

3.5 Cálculos de *Performance*

Para obter as tabelas anteriormente usadas foi preciso desenvolver funções capazes de calcular os dados pretendidos. Isto foi tudo calculado dentro de um espaço lexical criado para os algoritmos de procura.

```
(let ((results (list 0 -1 0 nil)))
     ;Algoritmos de procura
     ;Funções de cálculo
     ...)
```

Esta lista de *results* controla os nós explorados, gerados, tempo decorrido e nó solução. Uma vez que iremos usar funções destrutivas, e depois do docente ter aprovado esta ideia, foi necessário utilizar este espaço. Os resultados a obter foram os seguintes:

Nós gerados e explorados

Para os nós gerados e explorados apenas foi preciso utilizar funções que somam valores aos que existem nos resultados.

```
(defun add-generate (value) (setf results (list (+ (nth 0 results) value) (nth 1 results) (nth 2
results) (nth 3 results))))

(defun add-explored (value) (setf results (list (nth 0 results) (+ (nth 1 results) value) (nth 2
results) (nth 3 results))))
```

Tempo de Execução

Para calcular o tempo de execução **em milissegundos** foram usadas duas funções, uma para atualizar o tempo dos *results* para o tempo em milissegundos em que a procura começou e outro que calcula a diferença entre o tempo quando acabou e quando começou.

```
(defun start-timer ()
  "Starts timer with the internal-real-time function for better precision, the time is counted
in ms"
  (setf results (list (nth 0 results) (nth 1 results) (get-internal-real-time) (nth 3
results))))

(defun stop-timer ()
  "Calculates the difference between the current time and the timer registered on start in ms"
  (setf results (list (nth 0 results) (nth 1 results) (- (get-internal-real-time)(nth 2
results)) (nth 3 results))))
```

Penetrância

A fórmula da penetrância é dada por $Penetrância = ComprimentoSolução/Totaldenósgerados$ e para isso recorremos a utilizar o número de nós gerados no *results* e o comprimento da solução do nó guardado no *results* através de uma função mostrada no capitulo dos nós.

```
(defun calculate-pen(node)
  "Calculates penetration with L/T"
  (/ (node-solution-size node) (nth 0 results)))
```

Ramificação Média

Este é o elemento mais complexo de calcular, pois a expressão da ramificação média é $B + B^1 + B^2 ... + B^L = T$ onde **B** é a nossa ramificação média, **L** comprimento da solução e **T** o total de nós gerados. Uma vez que temos uma equação que pode atingir vários niveis, foi nos sugerido ([e dado em aula teórica*) uma função que implementa a fórmula de Newton Raphson.

```
(defun Newton-Raphson
       (f
        f-prime
        x-left
        x-right
        &key
        (accuracy 0.01)
        (maximum-number-of-iterations 100)
        (prevent-bracket-jumping-p t))
  "given
   [1] f (required)
       ==> a function with a single argument
   [2] f-prime (required)
       ==> another function with a single argument,
           this one being the first derivative of f
   [3] x-left (required)

       ==> left-hand bracket for the desired root;
```

```
          i.e., left-hand bracket <= desired root
   [4] x-right (required)
       ==> right-hand bracket for the desired root;
           i.e., desired root <= right-hand bracket"
  (assert (< x-left x-right))
  (let ((x (* 0.5 (+ x-left x-right)))
        delta-x denom-for-accuracy-test)
    (dotimes (j maximum-number-of-iterations
                (if (not (cerror "returns solution so far"
                                 "exceeding maximum number of iterations"))
                    (values x)))
      (setf delta-x (/ (funcall f x)  (funcall f-prime x)))
      (setf denom-for-accuracy-test (+ (abs x)
                                       (abs (decf x delta-x))))
      (cond
       (prevent-bracket-jumping-p
        (if (< x x-left) (setf x x-left))
        (if (> x x-right) (setf x x-right))
        (if (< (/ (abs delta-x) denom-for-accuracy-test) accuracy)
          (return (values x ))))
       ((<= x-left x x-right)
        (if (< (/ (abs delta-x) denom-for-accuracy-test) accuracy)
          (return (values x ))))
       (t
        (error "jumped out of brackets"))))))
```

No entanto para utilizar a fórmula é preciso termos a equação em função e a sua derivada.

```
;;Base function to calculate ramification
;;We have b+b^1+b^2...+b^L = T and we can convert it into
;;b+b^1+b^2...+b^L - T = 0
(defun base-ramification (b)
  "Ramification expression based on a variable B and the solution node found in results"
  (labels ((build-expression(b l)
             (cond ((= l 1) b)
                   (t (+ (expt b l) (build-expression b (1- l)))))))
    (- (build-expression b (node-solution-size (get-node))) (get-generate))))

;;To be used on Raphson, we need the first derivative of our ramification expression
(defun derivative-ramification (b)
  "First derivative expression of the ramification formula used on base-ramification"
  (labels ((build-expression(b l)
             (cond ((= l 1) 1)
                   (t (+ (* l (expt b (1- l))) (build-expression b (1- l)))))))
    (build-expression b (node-solution-size (get-node)))))
```

E por fim ainda é preciso ter cuidado com um caso de exceção, quando a solução apenas tem 1 elemento temos uma situação onde $B = T$ onde não precisamos de utilizar o método de Raphson e podemos diretamente devolver T.

```
(defun calculate-average-ramification()
  "Used to calculate our average ramification base on Newton-Raphson and testing if a solution
only contains one node"
  (if (= (node-solution-size (get-node)) 1)
      ;If the solution only contains 1 node, then we have ramification = generate nodes (b^1 =
t)
      (get-generate)
    ;Else we can apply the normal formula b+b^1+b^2....b^L = T
    (Newton-Raphson #'base-ramification #'derivative-ramification 0.0 5.0)))
```

Inicio e fim

Por fim é preciso controlar o começo e o fim do algoritmo, pois uma vez que estamos a trabalhar num espaço léxico com funções destrutivas, precisamos de garantir que quando o algoritmo começa, o *results* é reiniciado e no fim que o *results* esteja atualizado com o nó solução.

```
(defun start-performance ()
  "Starts performance with the results at 0"
  (setf results (list 0 0 0 nil))
  (start-timer))

(defun stop-performance (node)
  "Stops timer and prints the results with print-results function"
  (stop-timer)
  (setf results (list (nth 0 results) (nth 1 results) (nth 2 results) node))
  (print-results))
```

Os sítios onde estas funções são usadas já foi mostrado anteriormente nos capítulos dos respetivos algoritmos.

3.6 Resultados

Apenas a titulo de exemplo iremos mostrar alguns resultados obtidos nos diferentes algoritmos implementados sobre os diferentes tabuleiros.

Tabuleiro F

BFS

N/A

DFS com P = 30

```
(1 1 0 0 1 0 1 0 1 0 1 0 2 2)
(1 1 0 1 0 1 0 1 1 1 0 1 2 2)
(0 0 1 1 1 0 1 0 1 0 2 2 1 1)
(0 1 0 1 0 1 1 1 0 1 2 2 1 1)
(1 0 1 0 2 2 1 0 2 2 0 1 0 0)
(0 1 1 1 2 2 0 1 2 2 1 1 1 0)
(1 0 1 0 1 1 2 2 1 1 0 1 0 0)
(0 1 0 0 1 1 2 2 1 1 0 0 1 0)
(1 1 1 0 2 2 1 1 2 2 0 1 1 1)
(0 1 0 0 2 2 1 1 2 2 0 0 1 0)
(0 0 2 2 0 1 0 0 0 0 0 1 0 0)
(0 0 2 2 1 1 1 0 1 0 1 1 1 0)
(2 2 1 1 0 1 0 1 1 1 0 1 0 0)
(2 2 1 1 0 0 0 0 1 0 0 0 0 0)

Pieces: (0 4 5)
```

A* com heurística fornecida (similar ao IDA*)

```
(0 1 0 0 0 0 0 0 0 1 0 0 2 2)
(1 1 1 0 0 0 1 0 1 1 1 0 2 2)
(0 1 0 1 0 1 1 1 0 1 2 2 0 0)
(0 0 1 1 1 0 1 0 0 0 2 2 0 0)
(0 1 0 1 2 2 0 0 2 2 0 0 1 0)
(1 1 1 0 2 2 0 0 2 2 0 1 1 1)
(0 1 0 0 1 1 2 2 1 0 1 0 1 0)
(1 0 1 0 1 1 2 2 0 1 1 1 0 1)
(0 1 1 1 2 2 1 0 2 2 1 0 1 0)
(0 0 1 0 2 2 0 1 2 2 0 1 0 1)
(1 1 2 2 1 0 1 1 1 0 1 1 1 0)
(1 1 2 2 0 1 0 1 0 1 0 1 0 0)
(2 2 1 0 1 1 1 0 1 1 1 0 1 1)
(2 2 0 1 0 1 0 1 0 1 0 0 1 1)

Pieces: (0 7 3)
```

A* com a nova heurística (similar ao IDA*)

```
(0 0 0 0 0 0 0 0 0 0 0 0 2 2)
(0 0 0 0 0 0 0 0 0 0 0 0 2 2)
(0 0 0 0 0 0 0 0 0 0 2 2 0 0)
(0 0 0 0 0 0 0 0 0 0 2 2 0 0)
(0 0 0 0 2 2 0 0 2 2 0 0 0 0)
(0 0 0 0 2 2 0 0 2 2 0 0 0 0)
(0 0 0 0 0 0 2 2 0 0 0 0 0 0)
(0 0 0 0 0 0 2 2 1 0 0 0 0 0)
(0 0 0 0 2 2 0 1 2 2 0 0 0 0)
(0 0 0 0 2 2 1 0 2 2 0 0 0 0)
(0 0 2 2 0 1 0 1 0 0 0 0 0 0)
(0 0 2 2 1 1 1 0 1 1 0 1 1 0)
(2 2 0 1 0 1 0 0 1 1 0 1 1 0)
(2 2 1 0 1 0 1 0 0 0 1 0 0 1)

Pieces: (0 8 14)
```

Podemos verificar que o método que colocou menos peças foi a nossa heurística, pois o seu objetivo era acabar mais rápido, que se confirmou. E podemos verificar um caso particular em que o DFS conseguiu ocupar mais quadrado que a heurística fornecida (com um tempo de procura maior).

Tabuleiro E

DFS com P=30

```
(2 2 1 0 2 2 0 0 2 0 0 0 2 2)
(2 2 2 1 2 2 0 2 2 2 0 0 2 2)
(1 2 1 2 1 0 2 0 2 0 2 2 0 0)
(0 1 2 2 2 0 2 2 0 0 2 2 0 0)
(2 2 1 2 0 2 0 0 0 0 0 0 0 0)
(2 2 0 1 2 2 2 0 0 0 0 0 0 0)
(0 0 2 0 1 2 0 0 0 0 0 0 0 0)
(0 2 2 2 0 1 2 0 2 0 0 0 2 0)
(0 0 2 0 1 1 1 2 2 2 0 2 2 2)
(0 2 0 2 0 1 0 1 2 0 2 0 2 0)
(0 0 2 2 2 0 1 1 1 2 2 2 0 0)
(0 2 0 2 0 2 0 1 0 1 2 1 0 0)
(2 2 2 0 2 2 2 0 1 1 1 0 1 1)
(2 2 0 0 0 2 0 0 0 1 0 0 1 1)


Pieces: (0 9 12)
```

A* com heurística fornecida (similar ao IDA*)

```
(2 2 0 0 2 2 0 0 2 0 0 0 2 2)
(2 2 2 0 2 2 0 2 2 2 0 0 2 2)
(0 2 0 2 0 0 2 0 2 0 2 2 0 0)
(0 0 2 2 2 0 2 2 0 0 2 2 0 0)
(2 2 0 2 0 2 0 0 1 0 0 0 1 0)
(2 2 0 0 2 2 2 1 1 1 0 1 1 1)
(0 0 2 0 1 2 1 0 1 0 1 0 1 0)
(0 2 2 2 0 1 2 1 2 1 1 1 2 1)
(0 0 2 0 1 1 1 2 2 2 1 2 2 2)
(0 2 0 2 0 1 0 1 2 1 2 1 2 1)
(0 0 2 2 2 0 1 1 1 2 2 2 1 0)
(0 2 0 2 0 2 0 1 0 1 2 1 0 0)
(2 2 2 0 2 2 2 0 1 1 1 0 1 1)
(2 2 0 0 0 2 0 1 0 1 0 0 1 1)


Pieces: (0 9 9)
```

A* com a nova heurística (similar ao IDA*)

```
(2 2 0 0 2 2 0 0 2 0 0 0 2 2)
(2 2 2 0 2 2 0 2 2 2 0 0 2 2)
(0 2 0 2 0 0 2 0 2 0 2 2 0 0)
(0 0 2 2 2 0 2 2 0 0 2 2 0 0)
(2 2 0 2 0 2 0 0 0 0 0 0 0 0)
(2 2 0 0 2 2 2 0 0 0 0 0 0 0)
(0 0 2 0 0 2 0 0 0 0 0 0 0 0)
(0 2 2 2 0 0 2 0 2 0 0 0 2 0)
(0 0 2 0 0 1 0 2 2 2 0 2 2 2)
(0 2 0 2 1 0 1 1 2 0 2 0 2 0)
(0 0 2 2 2 0 1 1 0 2 2 2 0 1)
(0 2 0 2 0 2 0 0 1 0 2 1 1 0)
(2 2 2 0 2 2 2 1 0 1 0 1 1 0)
(2 2 0 0 0 2 1 0 1 0 1 0 0 1)


Pieces: (0 8 15)
```

Neste exemplo podemos novamente reparar que a nossa heurísticas cumpre o seu critério, e que desta vez a heurística fornecida colocou mais peças que o DFS e num espaço de tempo mais reduzido, verificando assim o seu bom funcionamento.

Restantes tabuleiros

O padrão verificado do tabuleiro A até E manteve-se, logo não vimos necessidade de repetir o descrito no ultimo parágrafo.

3.7 Métodos Recursivos

Embora o enunciado tenha permitidoque fossem feitas procuras iterativamente, ou seja através de um ciclo loop,achamos relevante incluir uma versão recursiva de cada procura ao projeto. Denotar que estas versões foram as ultimas a ser feitas e não tiveram tempo paraser devidamente testadas e estudadas e foram feitas com objetivo académico demostrar o domínio da linguagem.

Todas estão funcionais,mas podem ser otimizadas em alguns campos para reduzir iterações extra quepossam estar a ser realizadas. O código recursivo implementado é o seguinte.

```
(defun bfs-rec (node solution expand operatores)
  (labels ((bfs-aux (node solution expand operatores open close)
             (let ((expanded-nodes (filter-nodes (funcall expand node operatores 'bfs nil) open
close)))
             (cond ((null open) nil)
                   ((funcall solution node) (stop-performance node) node)
                   (t (add-explored 1)
                      (add-generate (length expanded-nodes))
                      (bfs-aux (car open) solution expand operatores (open-bfs (cdr open)
expanded-nodes) (append close (list node))))))))
    (start-performance)
    (bfs-aux node solution expand operatores (list node) nil)))

(defun dfs-rec (node solution expand operatores p)
  (labels ((dfs-aux (node solution expand operatores open close p)
             (let ((expanded-nodes (filter-nodes (funcall expand node operatores 'dfs p) open
close)))
             (cond ((null open) nil)
                   ((funcall solution node) (stop-performance node) node)
                   (t (add-explored 1)
                      (add-generate (length expanded-nodes))
                      (dfs-aux (car open) solution expand operatores (open-dfs (cdr open)
expanded-nodes) (append close (list node)) p))))))
    (start-performance)
    (dfs-aux node solution expand operatores (list node) nil p)))

(defun a-star-rec (node solution expand operators heuristic cost)
  (labels ((a-aux (node solution expand operators heuristic cost open close)
             (let* ((unfiltered-nodes (funcall expand node operators heuristic))
                    (expanded-nodes (filter-nodes unfiltered-nodes *open* *close*)))
               (add-explored 1)
               (add-generate (length expanded-nodes))
               (cond ((null open) nil)
                     ((funcall solution node) (stop-performance node) node)
                     (t (a-aux (car open) solution expand operators heuristic cost
                               (stable-sort (filter-nodes-update-open unfiltered-nodes (ordered-
insert-list (cdr open) expanded-nodes)) #'< :key cost)
                               (append close (list node))))))))
```

```
    (start-performance)
    (a-aux node solution expand operators heuristic cost (list node) nil)))

(defun ida-star-rec (node solution expand operators heuristic cost)
  (labels ((ida-aux (node solution expand operators heuristic cost open close root &optional
(bound (funcall cost node)))
             (let* ((unfiltered-nodes (funcall expand node operators heuristic))
                    (expanded-nodes (filter-nodes unfiltered-nodes *open* *close*)))
               (cond ((> (funcall cost node) bound)
                      (ida-aux root solution expand operators heuristic cost
                               (list root) nil root (funcall cost node)))
                     (t
                      (add-explored 1)
                      (add-generate (length expanded-nodes))
                      (cond ((null open) nil)
                            ((funcall solution node) (stop-performance node) node)
                            (t (ida-aux (car open) solution expand operators heuristic cost
                                        (stable-sort (filter-nodes-update-open unfiltered-nodes
(ordered-insert-list (cdr open) expanded-nodes)) #'< :key cost)
                                        (append close (list node)) root bound))))))))
    (start-performance)
    (ida-aux node solution expand operators heuristic cost (list node) nil node)))
```

Uma vez que é experimental, não é certo se irá funcionar com tabuleiros muito extensos, uma vez que o *LispWorks* gratuito tem um limite de stack e heap muito baixo para operações mais complexas.

4. Input/Output

Adicionalmente foi pedido que o utilizador pudesse interagir com o programa e que este fizesseoutput dos resultados para ficheiros de texto.

Embora entremos em detalhes de como interagir com a aplicação no Manual de Utilizador, o código e respetiva documentação irão ser deixados aqui como referência.

Para juntar todos os ficheiros lisps, decidimos solicitar ao utilizador que introduza os ficheiros a compilar, se a aplicação detetar que os ficheiros já foram compilados, irá saltar este pedido e passar para o decorrer normal.

O utilizador também terá a opção de escrever exit caso deseje sair da aplicação.

```
(defun read-file (filename)
  (with-open-file (stream filename)
    (loop for line = (read stream nil)
          while line
          collect line)))

;;Start function. To start the simulation, this function needs to be called in the listener
(defun start(&optional boards-value)
  "Start function. To start the simulation, this function needs to be called in the listener"
  (let ((boards (if (null boards-value) (initialize) boards-value)))
    (if (equal boards 'exit) (format t "~%Ended Session")
      (labels ((read_board ()
                 (let ((board (choose-board)))
```

```
                    (cond
                     ((not (numberp board)) nil)
                     ((or(< board 0) (> board (length boards))) (read_board))
                     (t (choose-algorithm (nth board boards) boards))))))(read_board)))))

;;Function that introduces simulation to user, greeting him and asking him for filepath
(defun initialize ()
  "Function that introduces simulation to user, greeting him and asking him for filepath"
  (format t "~%Welcome to BlocksAI!~%Please insert the filepath you want to simulate or type
exit to leave.~%")
  (let ((filepath (read)))
    (cond ((eq filepath 'exit) 'exit)
          ((or (pathnamep filepath) (stringp filepath))
           (read-file filepath) ;test file existence
           (if (eq (if (not (fboundp 'node-create)) (read-compile 'puzzle)) 'exit) 'exit
             (if (eq (if (not (fboundp 'bfs-it)) (read-compile 'procura)) 'exit) 'exit (read-
file filepath))))
          (t (format t "~%Invalid file path!~%") (initialize)))))


(defun read-compile (filename)
  (format t "~%Please insert the filepath to the file with ~d logic:~%" filename)
   (let ((filepath (read)))
   (cond ((eq filepath 'exit) 'exit)
         ((or (pathnamep filepath) (stringp filepath)) (compile-file filepath :load t))
         (t (format t "~%Invalid file path!~%") (read-compile filename)))))

;;function that asks user which board from file he wants to test
(defun choose-board()
  "Function that asks user which board from file he wants to test"
  (format t "~%Type restart to go back to begining.~%Type exit to end program.~%Choose the board
you want to test:~%")
  (let ((input (read)))
    (cond
     ((eq input 'exit) (format t "~%Ended Session"))
     ((eq input 'restart) (start))
     (t (*(- input 1) 2)))))

;;Function that asks user which algorithm he wants to test
(defun algorithm-number ()
  "Function that asks user which algorithm he wants to test"
  (format t "~%Please choose the algorithm you want to work with.~%1 - BFS ~%2 - DFS~%3 - A*~%4
- IDA* ~%")
  (read))

(defun state-calculator (board)
  (list
   (- 10 (block-count board 'square-1x1))
   (- 10 (block-count board 'square-2x2))
   (- 15 (block-count board 'cross))))

;;Auxiliar function tho execute bfs algorithm

(defun bfs-io-aux(board)
```

```
 (bfs-it (node-create (list board (state-calculator board)) nil 0 0 0 0 )
                     'solution-nodep 'node-expand (operators)))

;;Auxiliar function to execute dfs algorithm asking user to insert the maximum depth
(defun dfs-io-aux (board)
  (format t "~%Please insert maximum depth ~%")
  (let ((max-depth (read)))
    (cond ((numberp max-depth)
           (dfs-it (node-create(list board (state-calculator board)) nil 0 0 0 0 )
                     'solution-nodep 'node-expand (operators) max-depth))
        (t (dfs-io-aux board)))))

;;Auxiliar function to execute a-star with certain heuristic
(defun a-star-io-aux (board)
  (format t "~%Please insert wanted heuristic ~%1 - Default ~%2 - Student Custom~%")
  (let ((heuristic (read)))
    (cond ((and (numberp heuristic) (= heuristic 1))
           (a-star (node-create (list board (state-calculator board)) nil 0 0 0 0 )
                   'solution-nodep 'node-expand-a (operators) 'heuristic-squares 'node-f))
          ((and (numberp heuristic) (= heuristic 2))
           (a-star (node-create (list board (state-calculator board)) nil 0 0 0 0 )
                   'solution-nodep 'node-expand-a (operators) 'heuristic-custom-complex 'node-
f))
        (t (a-star-io-aux board)))))

;;Auxiliar function to execute ida-star with certain heuristic
(defun ida-star-io-aux (board)
  (format t "~%Please insert wanted heuristic ~%1 - Default ~%2 - Student Custom~%")
  (let ((heuristic (read)))
    (cond ((and (numberp heuristic) (= heuristic 1))
           (ida-star (node-create (list board (state-calculator board)) nil 0 0 0 0 )
                     'solution-nodep 'node-expand-a (operators) 'heuristic-squares 'node-f))
          ((and (numberp heuristic) (= heuristic 2))
           (ida-star (node-create (list board (state-calculator board)) nil 0 0 0 0 )
                     'solution-nodep 'node-expand-a (operators) 'heuristic-custom-complex 'node-
f))
        (t (ida-star-io-aux board)))))

(defun insert-record-file ()
  "Function that asks user to type the file path in which the program will record the results"
    (format t "~%Please insert the filepath where you want to save the results~%")
     (read))

(defun choose-algorithm (board boards)
  "Function that asks user to choose the algorithm to work with"
  (let ((algorithm (algorithm-number)))
    (cond ((eq algorithm 'exit) (format t "~% Ended Session"))
          ((= algorithm 1) (bfs-io-aux board)); BFS
          ((= algorithm 2) (dfs-io-aux board)); DFS
          ((= algorithm 3) (a-star-io-aux board)); A*
          ((= algorithm 4) (ida-star-io-aux board)); IDA*
          (t (choose-algorithm board boards))

          ))
```

```
  (results-write-file (insert-record-file))
  (start boards))
```

Na parte da escrita de ficheiros, esta ficou no espaço lexical da *performance*, pois é esta que tem acesso aos valores que queremos imprimir, de notar que deixamos alguns valores a serem imprimidos para a consola por uma questão de **feeback** ao utilizador.

```
(defun results-write-file (pathname)
    (with-open-file (out pathname :direction :output :if-exists :supersede :if-does-not-exist
:create)
    (format out "Tamanho da solução: ~d~%
Nós gerados: ~d~%
Nós explorados: ~d~%
Tempo de execução: ~dms~%
Penetrância: ~d~%
Ramificação média: ~s~%
Peças finais:
        1x1 = ~d
        2x2 = ~d
     Cruzes = ~d~%~%"
          (node-solution-size (get-node))
          (get-generate)
          (get-explored)
          (get-timer)
          (calculate-pen (get-node))
          (calculate-average-ramification)
          (first (node-pieces (node-state (get-node))))
          (second (node-pieces (node-state (get-node))))
          (third (node-pieces (node-state (get-node))))))
    (board-write-file (node-board (node-state (node-original (get-node)))) pathname "Inicio:~%")
    (board-action-write (node-steps (get-node)) pathname)
    (board-write-file (node-board (node-state (get-node))) pathname "Fim:~%")
    (format t "~%Results were saved in ~s~%~%" pathname))

(defun board-write-file (board pathname &optional message)
  "Prints a whole board to a file"
  (with-open-file (out pathname :direction :output :if-exists :append :if-does-not-exist
:create)
    (format out message))
  (mapcar #'(lambda (line) (board-line-write line pathname)) board)
  (with-open-file (out pathname :direction :output :if-exists :append :if-does-not-exist
:create)
    (format out "~%")))

(defun board-line-write (line pathname)
  "Prints a board line to a file"
  (with-open-file (out pathname :direction :output :if-exists :append :if-does-not-exist
:create)
    (mapcar #'(lambda (cell) (format out "~d " cell)) line)
    (format out "~%")))

(defun board-action-write (positions pathname)
  "Prints all steps made on a board to file"
```

```
  (labels ((print-aux (positions stream &optional number)
             (cond ((null positions) (format stream "~%"))
                   ((eq (car positions) "") (print-aux (cdr positions) stream number))
                   (t (format stream "~d- ~d~%" number (car positions)) (print-aux (cdr
positions) stream (1+ number))))))
    (with-open-file (out pathname :direction :output :if-exists :append :if-does-not-exist
:create)
      (print-aux positions out 1))))
```

Por fim, para auxiliar na leitura, uma vez que queremos adicionar tabuleiros novos ao jogo quando desejarmos, utilizamos uma função que calcula a quantidade de peças existentes num tabuleiro de um determinado tipo.

```
(defun ordered-insert (list ele)
  "Used by A* and IDA*"
  (cond ((null list) (list ele))
        ((> (node-h (car list)) (node-h ele)) (cons ele list))
        ((= (node-h (car list)) (node-h ele)) (if (> (node-depth(car list)) (node-depth ele))
(cons ele list) (cons (car list) (ordered-insert (cdr list) ele))))
        (t (cons (car list) (ordered-insert (cdr list) ele)))))

(defun ordered-insert-list (list elelist)
  "Used by A* and IDA*"
  (cond ((= (length elelist) 1) (ordered-insert list (car elelist)))
        (t (ordered-insert (ordered-insert-list list (cdr elelist)) (car elelist)))))
```

5. Conclusão e Aspetos Finais

Este projeto serviu para consolidar conhecimentos relacionados com a programação funcional, o paradigma de inteligência artificial e CommonLisp. Conseguimos realizar tudo o que é pedido no enunciado.

No entanto o algoritmo BFS não consegue finalizar os tabuleiros mais complexos dado as limitações existentes na versão gratuita do *LispWorks*. As versões recursivas também necessitam de testes mais extensos, pois é possivel que esgotem o stack disponivel.