Manual Técnico - Blokus Uno

Inteligência Artificial - Projeto 2

Grupo:

- André Serrado 201900318
- Gabriel Pais 201900301

Docente:

- Filipe Mariano

Este manual técnico contém a documentação da implementação do projeto, Blokus Uno. Este projeto teve como principal objetivo, a procura de soluções nos tabuleiros, recorrendo à utilização de algoritmos de procura. Este projeto foi também desenvolvido, exclusivamente com a linguagem Lisp.

Para uma melhor organização do projeto, este foi divido em três ficheiros.

- puzzle.lisp
- algoritmo.lisp
- jogo.lisp

Índice

Puzzle

Este ficheiro contém o código relacionado com todo o problema, sendo assim responsável, todas as verificações de restrições de inserção de peças no tabuleiro, inserção das mesmas, verificação das possíveis posições e contagem das peças.

Para uma melhor organização interna, o ficheiro foi dividido pelas seguintes secções, problemas, componentes do tabuleiro, funções secundárias, possíveis movimentos, operações com peças, verificações do tabuleiro e fim do jogo.

Problemas

Para a representação do tabuleiro do problema, foi implementada uma função que retorna uma lista, contendo um conjunto de sublistas. As sublistas representam as várias linhas do tabuleiro e cada átomo das mesmas, representam o valor da posição das colunas. Para representação do conteúdo do tabuleiro, as posições sem peças estão representadas pelo valor 0, com peças, temos representadas pelo valor 1, as que foram inseridas pelo **Jogador 1** e com o valor 2 as que foram inseridas pelo **Jogador 2**. Este tabuleiro é representa o estado inicial e visa facilitar a testagem das restantes funções implementadas.

De seguida, temos a função que retorna o tabuleiro utilizado.

```
;;; Board
;; [0] empty element/cell
;; [1] player one piece
;; [2] player two piece
```

```
;; Problem
;; At least 72 elements fulfilled
;; Empty Board 14x14
;; Returns a 14x14 empty board
(defun empty-board (&optional (dimension 14))
    (make-list dimension :initial-element (make-list dimension :initial-element
'0))
)
```

Componentes do tabuleiro

Row

- A função *row* recebe por parâmetro um índice em conjunto com o respetivo tabuleiro e retorna a linha correspondente ao índice dado por parâmetro.

```
(defun row(index board)
  "[index] must be a number between 0 and the board dimension"
  (nth index board)
)
```

Column

- A função *column* recebe por parâmetro um índice em conjunto com o respetivo tabuleiro e retorna a coluna correspondente ao índice dado por parâmetro.

```
(defun column(index board)
  "[index] must be a number between 0 and the board dimension"
  (mapcar (lambda (x) (nth index x))  board)
)
```

Element

- A função *element* recebe por parâmetro uma linha, uma coluna e um tabuleiro, e retorna o elemento que está nessa mesma posição.

```
(defun element(r col board)
    "[r] and [col] must be numbers between 0 and the board dimension"
  (nth col (row r board))
)
```

Funções secundárias

As funções secundárias, são responsáveis pelas verificações nos vários elementos dos tabuleiros.

Empty-elemp

- A função *empty-elemp*, recebe por parâmetro, a linha, a coluna, o tabuleiro e opcionalmente, um valor inteiro a verficar. Retorna *true* se um elemento for igual ao *val* (por defeito é 0 - elemento vazio) e *nil* caso contrário. Utiliza a função *element* para verificar a casa do tabuleiro.

```
(defun empty-elemp(row col board &optional (val 0))
  "[row] and [col] must be numbers between 0 and the board dimension"
  (cond
  ((or (< row 0) (> row (1- (length board))) (< col 0) (> col (1-(length board))))
nil)
  ((= (element row col board) val) t)
  (t nil)
  )
)
```

Check-empty-elems

- A função *check-empty-elems*, verifica índices. Recebe por parâmetro, o tabuleiro, uma lista de índices e opcionalmente um valor inteiro a verificar. Retorna uma lista com *true* ou *nil,* dependendo se os **índices**

estão **iguais** ao valor passado no **val** ou não. Utiliza a função *empty-elemp* para verificar cada elemento.

```
(defun check-empty-elems(board indexes-list &optional (val 0))
  "Each element(list with row and col) in indexes-list
   must contain a valid number for the row and column < (length board)"
  (mapcar (lambda (index) (empty-elemp (first index) (second index) board val))
indexes-list)
)
```

Replace-pos

- A função *replace-pos*, substitui uma posição no tabuleiro pelo parâmetro *val*. Esta recebe por parâmetro, a linha, a coluna e opcionalmente um valor, que por defeito é "**1**" e retorna uma linha (lista) com o elemento na posição da coluna, substituído pelo *val*.

```
(defun replace-pos (col row &optional (val 1))
    "[Col] (column) must be a number between 0 and the row length"
    (cond
     ((null row) nil)
     ((= col 0) (cons val (replace-pos (1- col) (cdr row) val)))
     (t (cons (car row) (replace-pos (1- col) (cdr row) val)))
    )
)
```

Replace-

- A função *replace-*, substitui um elemento no tabuleiro. Esta recebe por parâmetro, a linha, a coluna, o tabuleiro e opcionalmente um valor inteiro (por defeito é 1). Retorna o tabuleiro com o elemento substituído, pelo valor passado por parâmetro, ou por "**1**" caso não tenha sido passado nenhum valor. Utiliza a função *replace-pos* para substituir o valor no elemento pretendido.

```
(defun replace- (row col board &optional (val 1))
  "[Row] and [column] must be a number between 0 and the board length"
  (cond
   ((null board) nil)
   ((= row 0) (cons (replace-pos col (car board) val) (replace- (1- row) col (cdr
board) val)))
   (t (cons (car board) (replace- (1- row) col (cdr board) val)))
  )
)
```

Replace_multi_pos

- A função *replace-multi-pos*, substitui várias posições no tabuleiro. Esta recebe por parâmetro uma lista com todas as posições a substituir, o tabuleiro e opcionalmente um valor inteiro (por defeito é 1), retorna o tabuleiro com todos os elementos substituídos pelo valor **"1"**, ou pelo valor passado por parâmetro ao *val*. Utiliza a função *replace-* para substituir cada elemento.

```
(defun replace-multi-pos (pos-list board &optional (val 1))
    (cond
      ((null pos-list) board)
      (t (replace-multi-pos (cdr pos-list) (replace- (first (car pos-list))
(second (car pos-list)) board val) val))
    )
)
```

Remove-nil

- A função *remove-nil* recebe por parâmetro uma lista, remove todos os elementos *nil* e retorna uma lista após efetuar esta operação.

```
(defun remove-nil(list)
   (apply #'append (mapcar #'(lambda(x) (if (null x) nil (list x))) list))
)
```

Verificações do tabuleiro

Nesta secção as funções são responsáveis por verificar, a adjacência entre peças, a primeira posição do tabuleiro, os cantos das peças e se é possível inserir uma peça.

Check-adjacent-elems

- A função *check-adjacent-elems* verifica a adjacência das peças. Esta recebe por parâmetro, a linha, a coluna, o tabuleiro e uma peça. Se a peça for adjacente a outra, retorna *nil,* caso contrário retorna *true*.

```
(defun check-adjacent-elems (row col board piece val)
"
[val] must have the player piece value
"
  (cond
   ((eval (cons 'or (check-empty-elems board (piece-adjacent-elems row col piece)
val))) nil)
   (t t)
   )
)
```

Check-corner

- A função *check-corner* recebe por parâmetro uma lista com as cordenadas da peça e opcionalmente um valor a comparar nos cantos da peça, que por defeito é 0. Retorna *nil* caso a posição esteja vazia, caso contrário retona *true*.

```
(defun check-corner (piece-format &optional (corner 0))
  (let ((list-index (car piece-format)))
    (cond
      ((null piece-format) nil)
      ((= corner (first list-index) (second list-index)) 0)
      (t (check-corner (cdr piece-format) corner))
    )
  )
)
```

Force-move

- A função *force-move*, filtra as possíveis jogadas, se o primeiro elemento do canto superior esquerdo estiver vazio, isto para o **Jogador 1**, caso seja o **Jogador 2**, verifica o canto inferior direito, só permite peças nesse elemento, caso não esteja vazio, só permite movimentos que colocam peças em contacto com outras (apenas nos cantos). Esta recebe por parâmetro, a linha, a coluna, o tabuleiro e a peça, retorna *true* se o movimento for permitido e *nil* caso contrário.

```
(defun force-move (row col board piece &optional (corner2check 0) (pieces-val 1))
"
[corner2check] player1 = 0 || player2 = 13
[pieces-val]   player1 = 1 || player2 = 2
"
  (let ((corner-index (check-corner (piece-taken-elems row col piece)
corner2check)) ; [corner-index] 0 if player's trying to put a piece in his corner
|| nil if he is not
        (corner-state (element corner2check corner2check board))
; [corner-state] 0 if the player corner is empty || 1 or 2 if it already has a
piece
       )
   (cond
    ((and (= corner-state  0) (null corner-index)) nil)
    ((and (= corner-state 0 corner-index)) t)
    ((eval (cons 'or (check-empty-elems board (piece-corners-elems row col piece)
pieces-val))) t)
    (t nil)
   )
  )
)
```

Filter-player-move

- A função *filter-player-move* recebe por parâmetro o jogador, a linha, a coluna, o tabuleiro e a peça. Com esta informação filtra as jogadas do utilizador, caso seja o **Jogador 1**, este deverá jogar no canto superior esquerdo e o valor da peça será 1, para o **Jogador 2** a peça terá o valor 2 e este terá que jogar no canto inferior direito. Retorna o valor da função *force-move*, *true* se a jogada for possível e *nil* caso contrário.

```
(defun filter-player-move (player row col board piece)
"
[player] player1 = 1 || player2 = -1
"
  (cond
    ((equal player 1) (force-move row col board piece))
    ((equal player -1) (force-move row col board piece 13 2))
    (t nil)
  )
)
```

Can-placep

- A função *can-placep*, reúne as funções de verificações. Esta recebe por parâmetro uma lista com os elementos a ocupar pela peça, o tabuleiro, a linha, a coluna, a peça e o jogador, retorna *true* caso seja possível inserir a peça e *nil* caso contrário. Utiliza as funções *pieces-left-numb*, *force-move*, *check-adjacent-elems*, *check-empty-elems* e *piece-taken-elems* para fazer todas a verificações necessárias.

```
(defun can-placep (pieces-list board row col piece player)
"
[player] player1 = 1 || player2 = -1
"
(let ((pieces-val (cond ((= player 1) 1) (t 2))))
  (cond
    ((= 0 (pieces-left-numb pieces-list piece)) nil)
    ((or (> row (length board)) (< row 0) (< col 0) (> col (length board))) nil)
    ((not (filter-player-move player row col board piece)) nil)
    ((not (check-adjacent-elems row col board piece pieces-val)) nil)
    ((eval (cons 'and (check-empty-elems board (piece-taken-elems row col
piece))))t)
    (t nil)
  )
)
)
```

Possíveis movimentos

As funções que verificam os possíveis movimentos, são responsáveis pelas verificações que melhoram a eficiência do algoritmo, estas evitam a verificação de posições desnecessárias do tabuleiro.

Filter-possible-moves

- A função *filter-possible-moves* verifica onde a peça pode ser colocada no tabuleiro a partir da lista de índices recebida por parâmetro. Retorna uma lista com todos os possíveis movimentos, com base nos índices (coordenadas) fornecidos.

```
(defun filter-possible-moves(pieces-list piece board indexes-list player)
  (remove-nil(mapcar (lambda (index) (cond ((can-placep pieces-list board (first
index) (second index) piece player) index) (t nil))) indexes-list))
)
```

All-spaces

- A função *all-spaces* verifica onde a peça pode ser colocada no tabuleiro a partir de uma lista de índices. Retorna uma lista com todos os movimentos possíveis com base em todos os índices fornecidos.

```
(defun all-spaces(pieces-list piece board indexes-list player)
  (remove-nil(mapcar (lambda (index) (cond ((can-placep pieces-list board (first
index) (second index) piece player) index) (t nil))) indexes-list))
)
```

Possible-moves

- A função *possible-moves* verifica onde a peça pode ser colocada no tabuleiro. Retorna uma lista com índices para todos os movimentos possíveis dessa mesma peça, ordenada de cima para baixo.

```
(defun possible-moves(pieces-list piece board player)
  (reverse (all-spaces pieces-list piece board (check-all-board board (1- (length
board)) (1- (length (car board)))) player))
)
```

Check-all-board

- A função *check-all-board* retorna uma lista com todos os índices do tabuleiro

```
(defun check-all-board(board row col)
  (cond
    ((< row 0) nil)
    ((< col 0) (check-all-board board (1- row) (1- (length board))))
    (t (cons (list row col) (check-all-board board row (1- col))))
  )
)
```

Piece-taken-elems

- A função *piece-taken-elems*, recebe por parâmetro, a linha, a coluna e uma peça. Retorna uma lista com todos os elementos, que essa mesma peça ocupa no tabuleiro.

```
(defun piece-taken-elems (row col piece)
  (cond
   ((equal piece 'piece-a) (cons (list row col) nil))
   ((equal piece 'piece-b)
    (list (list row col) (list row (1+ col)) (list (1+ row) col) (list (1+ row)
(1+ col)))
    )
   ((equal piece 'piece-c-2)
    (list (list row col) (list (1+ row) col) (list (1+ row) (1+ col)) (list (+ row
2) (1+ col)))
    )
   ((equal piece 'piece-c-1) (list (list row col) (list row (1+ col)) (list (1-
row) (1+ col)) (list (1- row) (+ col 2))))
   (t nil)
   )
)
```

Piece-adjacent-elems

- A função *piece-adjacent-elems* cria uma lista para cada peça, das casas adjacentes. Esta recebe por parâmetro, a linha, a coluna e uma peça. Retorna uma lista, referente à peça passado por parâmetro com todas os elementos adjacentes da mesma.

```
(defun piece-adjacent-elems (row col piece)
  (cond
   ((equal piece 'piece-a) (list (list row (1+ col)) (list row (1- col)) (list (1+
row) col) (list (1- row) col)))
   ((equal piece 'piece-b) (list (list row (1- col)) (list (1+ row) (1- col))
(list (1- row) col) (list (1- row) (1+ col)) (list (+ row 2) col) (list (+ row 2)
(1+ col)) (list row (+ col 2)) (list (1+ row) (+ col 2))))
   ((equal piece 'piece-c-1)
    (list (list row (- col 2)) (list (1- row) col) (list (- row 2) (1+ col)) (list
(- row 2) (+ col 2)) (list (1- row) (+ col 3)) (list (1+ row) col) (list (1+ row)
(1+ col)) (list row (+ col 2))))
   ((equal piece 'piece-c-2) (list (list (1- row) col) (list row (1- col)) (list
row (1+ col)) (list (1+ row) (1- col)) (list (1+ row) (+ col 2)) (list (+ row 2)
col) (list (+ row 2) (+ col 2)) (list (+ row 3) (1+ col))))
   (t nil)
   )
)
```

Piece-corners-elems

- A função *piece-corners-elems*, cria uma lista para cada peça, das casas referentes aos cantos exteriores. Esta recebe por parâmetro, a linha, a coluna e uma peça. Retorna uma lista, referente à peça passada

por parâmetro com todas os elementos que dizem respeito aos cantos exteriores da mesma.

```
(defun piece-corners-elems (row col piece)
  (cond
   ((equal piece 'piece-a)
    (list (list (1- row) (1- col)) (list (1+ row) (1- col)) (list (1- row) (1+
col)) (list (1+ row) (1+ col))))
   ((equal piece 'piece-b)
    (list (list (1- row) (1- col)) (list (+ row 2) (1- col)) (list (1- row) (+ col
2)) (list (+ row 2) (+ col 2))))
   ((equal piece 'piece-c-1)
    (list (list (- row 2) col) (list (1- row) (1- col)) (list (1+ row) (1- col))
(list (- row 2) (+ col 3)) (list row (+ col 3)) (list (1+  row) (+ col 2))))
   ((equal piece 'piece-c-2)
    (list (list (1- row) (1- col)) (list (1- row) (1+ col)) (list row (+ col 2))
(list (+ row 2) (1- col)) (list (+ row 3) col) (list (+ row 3) (+ col 2))))
   )
)
```

Insert-piece

- A função *insert-piece*, insere peças no tabuleiro. Esta recebe por parâmetro uma lista com os elementos a ocupar, a linha, a coluna, o tabuleiro, uma peça e opcionalmente o jogador, que por defeito é o **Jogador 1**. Com recurso à função *can-placep*, retorna o tabulerio caso seja possível inserir a peça no mesmo, ou *nil* caso contrário. Utiliza ainda as funções *replace-multi-pos* e *piece-taken-elems* para inserir a as peças com a suas formas corretas, respetivamente.

```
(defun insert-piece (pieces-list row col board piece &optional (player 1))
"
[player] player1 = 1 || player2 = -1
"
  (let ((val (cond ((= player 1) 1) (t 2))))
    (cond
      ((null (can-placep pieces-list board row col piece player)) nil)
      (t (replace-multi-pos (piece-taken-elems row col piece) board val)))
  )
)
```

Pieces-left-numb

- A função *pieces-left-numb*, recebe por parâmetro uma lista com todas as peças restantes por jogar e tipo da peça, retorna quantas peças sobraram por tipo.

```
(defun pieces-left-numb(pieces-list piece-type)
  (cond
    ((equal piece-type 'piece-a)(first pieces-list))
    ((equal piece-type 'piece-b) (second pieces-list))
```

```
    (t (third pieces-list))
  )
)
```

Removed-used-piece

- A função *removed-used-piece*, recebe por parâmetro uma lista com todas as peças e o tipo da peça, e remove uma peça da lista de peças restantes a jogar.

```
(defun remove-used-piece(pieces-list piece-type)
  (cond
    ((equal piece-type 'piece-a) (list (1- (first pieces-list)) (second pieces-
list) (third pieces-list)))
    ((equal piece-type 'piece-b) (cadr pieces-list) (list (first pieces-list) (1-
(second pieces-list)) (third pieces-list)))
    (t (list (first pieces-list) (second pieces-list) (1- (third pieces-list))))
  )
)
```

Operações com Peças

Esta secção contém as funções referentes à inserção das peças, número de peças a inserir, atualização do número de peças a inserir e verificação de todas as possíveis inserções de peças no tabuleiro.

Nesta secção definimos as funções referentes à quantidade de peças iniciais, lista de todas as operações e a inserção das peças no tabueleiro.

Init-pieces

- A função *init-pieces*, cria uma lista com o número total de peças, por cada tipo de peças.

```
(defun init-pieces()
  (list 10 10 15)
)
```

Operations

- A função *operations* retorna uma lista com todas as operações.

```
(defun operations()
  (list 'piece-b 'piece-c-1 'piece-c-2 'piece-a)
)
```

- As seguintes funções retornam o tabuleiro com as peças colocadas ou *nil* caso contrário.

Piece-a

```
(defun piece-a (pieces-list index board player)
"
[pieces-list] list with the number of each piece left
[index] list with row and column
[player] player playing  player1 = 1 || player2 = -1
"
  (insert-piece pieces-list (first index) (second index) board 'piece-a player)
)
```

Piece-b

```
(defun piece-b (pieces-list index board player)
   (insert-piece pieces-list (first index) (second index) board 'piece-b player)
)
```

Piece-c-1

```
(defun piece-c-1 (pieces-list index board player)
  (insert-piece pieces-list (first index) (second index) board 'piece-c-1 player)
)
```

Piece-c-2

```
(defun piece-c-2 (pieces-list index board player)
  (insert-piece pieces-list (first index) (second index) board 'piece-c-2 player)
)
```

Final do Jogo

Get-h

- A função *get-h* retorna o valor da heurística.

```
(defun get-h(node color)
  (- (count-points (pieces-list node color)) (count-points (pieces-list node (-
color))))
)
```

Count-points

- A função *Count-points* recebe por parâmetro e retorna o valor da pontução.

```
(defun count-points (pieces-list)
  (+ (- 10 (first pieces-list)) (* (- 10 (second pieces-list)) 4) (* (- 15 (third
pieces-list)) 4))
)
```

Winner

- A função *winner* recebe por parâmetro o nó, define o vencedor, usa a função count-points. Retorna o vencedor.

```
(defun winner(node)
    (let ((p1-points (count-points (pieces-list node 1))) (p2-points (count-points
(pieces-list node -1))))
        (cond
        ((> p1-points p2-points) 'Jogador1)
        ((< p1-points p2-points) 'Jogador2)
        (t 'Empate)
        )
    )
)
```

Algoritmo

Memoização

A Memoização melhora o desempanho dos algoritmos, armazenando os resultados das funções na memória e retornando o resultado em cache quando as mesmas entradas ocorrerem novamente.

```
(let ((tab (make-hash-table)))
  (defun fib-memo (n)
    (or (gethash n tab) (let ((val (funcall #'FUNC n))) (setf (gethash n tab) val)
val))
  )
)
```

Tipos de Dados Abstratos

Os tipos abstratos de dados são criados e utilizados para guardar as informações necessárias de modo a facilitar o desenvolvimento de uma solução independentemente do problema enfrentado. Assim sendo, estes dados abstratos podem ser utilizados para tentar resolver qualquer problema com os algoritmos disponíveis.

Player-info

- A função *player-info* recebe por parâmetro uma peça e um jogador. Retorna uma lista com a peça e o jogador.

```
(defun move-info(piece move)
"
[piece] operation
[move] list with row and col
"
(list (list piece move))
)
```

Make-node

- A função *make-node* contrói um nó com o estado no *tabuleiro*, a *profundidade* e o *nó pai*. Usa a função *init-pieces* para definir a lista de peças e retorna uma lista com todos os dados.

```
(defun make-node(state &optional (parent nil) (p1-node (player-info)) (p2-node
(player-info -1)) (f 0))
  (list state parent f p1-node p2-node)
)
```

Node-state

- A função *node-state* retorna o estado do nó. O estado de um nó é representado pelo tabuleiro com as peças inseridas (em caso de utilização das operações disponíveis).

```
(defun node-state(node)
  (first node)
)
```

Node-parent

- A função *node-parent* retorna o nó pai do nó, através do ponteiro que este guarda.

```
(defun node-parent(node)
  (second node)
)
```

Node-value

- A função *node-value* retorna o valor do nó.

```
(defun node-value(node)
    (third node)
)
```

Node-p1

- A função *node-p1* retorna o nó referente ao **Jogador 1**.

```
(defun node-p1 (node)
    (fourth node)
)
```

Node-p2

- A função *node-p2* retorna o nó referente ao **Jogador 2**.

```
(defun node-p2 (node)
  (nth (1- (length node)) node)
)
```

Player-moves

- A função *player-moves*, retorna todos os movimentos realizados por um jogador.

```
(defun player-moves(node color)
    (cond
        ((= 1 color) (second (node-p1 node)))
        (t (second (node-p2 node)))
    )
)
```

Last-move

- A função *last-move*, retorna um nó com a informação referente à última jogada.

```
(defun last-move(node color)
  (let ((moves (player-moves node color)))
    (nth (1-(length moves)) moves)
  )
)
```

Pieces-list

- A função *pieces-list* retorna uma lista com o número de peças que ainda se pode pôr, por tipo. Inicialmente a lista deverá ser sempre iniciada com valores predefinidos, (**10 10 15**) **peça-a**, **peça-b** e **peça-c**, respetivamente.

```
(defun pieces-list(node color)
    (cond
        ((= 1 color) (third (node-p1 node)))
        (t (third (node-p2 node)))
    )
)
```

Algoritmos

Os algoritmos são funções que executam um conjunto de operações com o objetivo de chegar a um estado final (pré-definido). Estes algoritmos exploram um espaço de possibilidades tentanto vários caminhos possíveis. Este processo consiste num espaço de estados.

Negamax

- A função *negamax* é responsável por executar todos os passos do algoritmo negamax.

```
(defun negamax(max-time
               &optional
               (node (make-node (empty-board) nil (player-info 1) (player-info -1)
most-negative-fixnum))
               (color 1)
               (player 1)
               (max-depth 6)
               (alpha most-negative-fixnum)
               (beta most-positive-fixnum)
               (start-time (get-internal-real-time))
               (visited-nodes 1)
               (cuts 0)
               )

  (let* ((children (order-nodes (funcall #'expand-node node (operations)
player))))
    (cond
     ((or (= max-depth 0) (null children) (>= (runtime start-time) max-time))
      (solution-node (final-node-f node color player) visited-nodes cuts start-
time))
     (t (negamax- node children max-time color player max-depth alpha beta start-
time visited-nodes cuts))
     )
    )
  )
```

Negamax-

- A função *negamax-* é uma função auxiliar, responsável por criar a árvore a partir dos sucessores.

```
(defun negamax-(parent children max-time color player max-depth alpha beta start-
time visited-nodes cuts)
    (cond
        ((= (length children) 1)
            (negamax max-time (-f (car children)) (- color) (- player) (1- max-
depth) (- beta) (- alpha) start-time (1+ visited-nodes) cuts))
        (t (let* (
            (solution (negamax max-time (-f (car children)) (- color) (- player)
(1- max-depth) (- beta) (- alpha) start-time (1+ visited-nodes) cuts))
            (best-value (node-value (max-f (car solution) parent)))
            (temp-alpha (max alpha best-value))
            (v-nodes (get-visited-nodes solution))
            (cuts-numb (get-cuts solution))
            )
                (cond
                    ((>= temp-alpha beta) (solution-node parent v-nodes (1+ cuts-
numb) start-time))
                    (t (negamax- parent (cdr children) max-time color player max-
depth temp-alpha beta start-time v-nodes cuts-numb))
                )
            )
        )
    )
)
```

Solution Node

Solution-node

```
(defun solution-node(node visited-nodes cuts-number start-time)
    (list node (list visited-nodes cuts-number (runtime start-time)))
)
```

Get-solution-node

```
(defun get-solution-node(sol-node) (car sol-node))
```

Get-visited-nodes

```
(defun get-visited-nodes(sol-node) (car (second sol-node)))
```

Get-cuts

```
(defun get-cuts(sol-node)(second (second sol-node)))
```

Get-runtime

- A função *get-runtime*

```
(defun get-runtime(sol-node)(third (second sol-node)))
```

Auxiliares

As funções auxiliares são utilizadas como suporte aos dados abstratos, aos algoritmos implementados ou até como suplemento a outras funções secundárias.

Get-child

- A função *get-child* utiliza uma peça e aplica uma operação com um dos movimentos possíveis para criar um filho de um nó.
- Cria um nó filho e retorna-o.
- Utiliza como funções auxiliares *make-node*, *remove-used-piece*, *pieces-left* e *player-info*.

```
(defun get-child(node possible-move operation color &aux (pieces-left (pieces-list
node color)) (state (node-state node)))
    "
    [Operation] must be a function
    [color] represents the player - player1 = 1 || player2 = -1
    "
    (let* (
          (move (funcall operation pieces-left possible-move state color))
          (updated-pieces-list (remove-used-piece pieces-left operation))
              (updated-player-info (player-info color (append (player-moves node
color) (move-info operation possible-move)) updated-pieces-list))
              ;(updated-player-info (player-info color nil updated-pieces-list))
          )
      (cond
        ((null move) nil)
        ;((= color 1) (make-node move nil updated-player-info (node-p2 node)
(count-points updated-pieces-list)))
        ((= color 1) (make-node move nil updated-player-info (node-p2 node) 0))
    ;(t (make-node move nil (node-p1 node) updated-player-info (count-points
updated-pieces-list)))
        (t (make-node move nil (node-p1 node) updated-player-info 0))
      )
```

```
  )
)
```

Get-children

- A função *get-children* cria todos os sucessores possíveis a partir de um estado e um tipo de peça.
- Utiliza como função auxiliar *get-child* para criar os vários sucessores para as várias jogadas possíveis.
- Retorna uma lista com todos os sucessores de um nó aplicados a uma operação/peça.

```
(defun get-children(node possible-moves operation color)
  (cond
    ((null possible-moves) nil)
    (t (cons (get-child node (car possible-moves) operation color) (get-children
node (cdr possible-moves) operation color)))
  )
)
```

Expand-node

- A função *expand-node* cria todos os sucessores possíveis de um nó, ou seja, cria os filhos do nó ***node***, para todas as jogadas possíveis e todas as operações/peças.
- Retorna uma lista com todos os sucessores.
- Utiliza como funções auxiliares *remove-nil*, *possible-moves* e *get-children*.

```
(defun expand-node(node operations color)
  "
  [operations] must be a list with all available operations
  "
        (cond
            ((null operations) nil)
            (t (remove-nil (append (get-children node (funcall #'possible-moves
(pieces-list node color) (car operations) (node-state node) color) (car
operations) color)
                       (expand-node node (cdr operations) color))))

        )
)
```

Order-nodes

- A função *order-nodes* ordena uma lista de nós de forma decrescente.

```
(defun order-nodes(node-list)
    (sort node-list #'> :key #'node-value)
)
```

Max-f

- A função *max-f* recebe por parâmetro dois nós e devolve o nó com o valor mais alto.

```
(defun max-f(node1 node2)
    (cond
        ((>= (node-value node1) (node-value node2)) node1)
        (t node2)
    )
)
```

-f

- A função *-f* recebe por parâmetro um nó e devolve o nó com o valor invertido.

```
(defun -f(node)
    (make-node (node-state node) (node-parent node) (node-p1 node) (node-p2 node)
(- (node-value node)))
)
```

Final-node-f

- A função *final-node-f* retorna um nó com o seu valor multiplicado pela cor das peças do jogador.

```
(defun final-node-f(node color player)
    (make-node (node-state node) (node-parent node) (node-p1 node) (node-p2 node)
(* color (get-h node player)))
)
```

Performance Stats

Runtime

- A função *runtime* calcula o tempo.

```
(defun runtime(start-time)
    (/(- (get-internal-real-time) start-time) 1000)
)
```

Jogo

Game Handler

Start

- A função *start* recebe dados pelo utilizador a partir do teclado e por consequência executa a ação/operação correspondente, em relação à função *init-menu*.

```
(defun start()
    (init-menu)
    (let ((option (read)))
      (if (and (numberp option) (>= option 0) (<= option 2))
        (cond
          ((equal option 1) (get-starter))
          ((equal option 2) (get-time-limit))
          ((equal option 0) (format t "~%Adeus!"))
        )

        (progn (print "Opção inválida. Tente novamente") (start))
      )
    )
)
```

Starter-view

- A função *starter-view* mostra o menu referente à opção de quem irá começar primeiro *Humano* ou *Computador*.

```
(defun starter-view()
    (format t "~%_____________________________________________________________")
    (format t "~%\\                      BLOKUS DUO                           /")
    (format t "~%/                  Quem começa primeiro?                    \\")
    (format t "~%\\                    (tipo das peças)                       /")
    (format t "~%/     1 - Humano (1)                                        \\")
    (format t "~%\\     2 - Computador (2)                                    /")
    (format t "~%/     0 - Voltar                                            \\")
    (format t "~%\\_____________________________________________________________/~%~%>
")
)
```

Get-starter

- A função *get-starter* recebe dados pelo utilizador a partir do teclado e por consequência executa a ação/operação correspondente, em relação à função *starter-view*.

```
(defun get-starter()
  (starter-view)
  (let ((option (read)))
```

```
    (cond
      ((or (< option 0) (> option 2)) (progn (format t "Insira uma opcao valida")
(get-starter)))
      ((= option 0) (start))
      (t (get-time-limit option))
    )
  )
)
```

Time-limit-view

- A função *time-limit-view* mostra o menu referente à opção escolha do tempo limite para uma jogada do *computador*.

```
(defun time-limit-view()
  (format t "~%_____________________________________________________________")
  (format t "~%\\                        BLOKUS DUO                          /")
  (format t "~%/          Defina o tempo limite para uma jogada            \\")
  (format t "~%\\             do computador (1 - 20) segundos.               /")
  (format t "~%\\                                                           \\")
  (format t "~%/     0 - Voltar                                            /")
  (format t "~%\\                                                         \\")
  (format t "~%/_____________________________________________________________/~%~%> ")
)
```

Get-time-limit

- A função *get-time-limit* recebe dados pelo utilizador a partir do teclado, neste caso recebe um número que corresponde ao tempo limite para uma jogada do computador *time-limit-view*.

```
(defun get-time-limit(&optional (starter nil))
  (progn (time-limit-view)
    (let ((option (read)))
      (cond
        ((and (= option 0) (null starter)) (start))
        ((= option 0) (get-starter))
        ((or (not (numberp option)) (< option 1) (> option 20)) (progn (format t
"Insira uma opção válida") (get-time-limit)))
        (t (list starter option))
      )
    )
  )
)
```

Files Handler

Get-log-path

- A função *get-log-path* retorna o caminho para o arquivo de log.

```
(defun get-log-path()
  (make-pathname :host "D" :directory '(:absolute "GitHub\\Blokus2") :name "log"
:type "dat")
)
```

Human-vs-Computer

- A função *human-vs-computer* é responsável por executar o algoritmo no jogo entre **Humano** (Utilizador) contra **Computador**.

```
(defun human-vs-computer(max-time player &optional (node (make-node (empty-
board))))
  (let* (
          (p-moves    (expand-node node (operations) 1))
          (p-pieces   (pieces-list node 1))
          (can-p-play (or (null p-moves) (= 0 (apply '+ p-pieces))))
; t = can't play
          (c-moves    (expand-node node (operations) -1))
          (c-pieces   (pieces-list node -1))
          (can-c-play (or (null c-moves) (= 0 (apply '+ c-pieces))))
; t = can't play
          (can-current-play (cond ((= player 1) can-c-play) (t can-c-play)))
; t = can't play
        )
    (cond
      ((and can-p-play can-c-play) (log-footer node))
      ((= 1 player)
; Man
        (cond
          ((eval can-current-play)
            (progn
              (format t "~%~%________Sem Jogadas________~%~%")
              (human-vs-computer max-time (- player) node))
          )
          (t(let* (
                    (piece (piece-input node player))
                    (indexes (move-input node piece player))
                    (move (get-child node indexes piece player))
                  )
              (progn
                  (log-file (solution-node move 0 0 0) player)
                  (format t "Jogou a peca ~a na posicao (~a , ~a)~%" piece (first
indexes) (second indexes))
                  (human-vs-computer max-time (- player) move)
              )
            )
          )
        )
```

```
      )
      (t
; Computer
        (cond
          ((eval can-current-play)
            (progn
              (format t "~%~%________Computador Sem Jogadas________~%~%")
              (human-vs-computer max-time (- player) node)
            )
          )
          (t(let* (
                    (solution-nd (negamax max-time node 1 player))
                    (sol-node (get-solution-node solution-nd))
                  )
              (progn
                (log-file solution-nd player)
                (human-vs-computer max-time (- player) (make-node (node-state sol-
node) nil (node-p1 sol-node) (node-p2 sol-node)))
              )
            )
          )
        )
      )
    )
  )
)
```

Piece-view

- A função *piece-view* mostra o menu de escolha do tipo de peça a colocar.

```
(defun piece-view(pieces)
  (progn
    (format t "~%_____________________________________________________________")
    (format t "~%\\           Escolha o tipo de peca a colocar              /")
    (cond
      ((> (first pieces) 0)
        (format t "~%/                    1 - peca A
\\"))
      ((> (second pieces) 0)
        (format t "~%\\                    2 - peca B
/"))
      ((> (third pieces) 0)
        (progn
          (format t "~%\\                      3 - peca C1
\\")
          (format t "~%/                      4 - peca C2
/")
        )
      )
    )
```

```
    (format t
"~%/_____________________________________________________\\~%~%>")
  )
)
```

Run-h-vs-c

- A função *run-h-vs-c* inicia o jogo Humano contra Computador, utilizando como função auxiliar *human-vs-computer*.

```
(defun run-h-vs-c()
  (let* ((starter (get-starter)) (starter-val (cond ((= 1 starter) 1) (t -1) ))
(max-time (get-time-limit)))
    (progn
      (log-header max-time)
      (human-vs-computer max-time starter-val)
    )
  )
)
```

Piece-input

- A função *piece-input* recebe a peça a utilizar pelo utilizador.

```
(defun piece-input(node player &optional (pieces (pieces-list node player)))
    (prog
        (piece-view pieces)
        (let (option (read))
          (cond
            ((or (not (numberp option)) (< 1 option) (> 4 option))
              (progn
                (format t "~%
_____________________________________________________")
                (format t "~%/                Escolha uma opcao valida
/")
                (format t
"~%_____________________________________________________~%")
                (piece-input node player pieces)
              )
            )
            (t
              (cond
                ((and (= 1 option) (> (first pieces) 0)) 'piece-a)
                ((and (= 2 option) (> (second pieces) 0)) 'piece-b)
                ((and (= 3 option) (> (third pieces) 0)) 'piece-c-1)
                ((and (= 4 option) (> (third pieces) 0)) 'piece-c-2)
                (t (progn
                  (format t "~%
```

```
_____________________________________________________________")
                    (format t "~%/                  Escolha uma opcao valida
/")
                    (format t
"~%_____________________________________________________________~%")
                    (piece-input node player pieces)
                  ))
               )
             )
           )
         )
    )
)
```

Move-view

- A função *move-view* mostra uma mensagem, que pede ao tulizador para inserir a posição do tabuleiro que deseja inserir a peça.

```
(defun move-view(moves &optional (first-time t) (piece-numb 1) &aux (move (car
moves)))
  (cond
    ((null moves) (format t
"~%_____________________________________________________________~%"))
    (t (progn
      (cond
        ((eval first-time)
          (progn
          (format t
"~%_____________________________________________________________")
          (format t "~%\\               Escolha a posicao da peca
/")
          (format t "~%/
\\")
          )
        )
      )
          (format t "~%\\                    ~a - (~a , ~a)                    /"
piece-numb (first move) (second move))
      (move-view (cdr moves) nil )
    ))
  )
)
```

Move-input

- A função *move-input* recebe a posição que o utilizador deseja inserir a peça no tabuleiro.

```
(defun move-input(node piece player)
  (let* (
          (pieces (pieces-list node player))
          (state (node-state node))
          (moves (possible-moves pieces piece state player))
        )
    (progn
      (move-view moves)
      (let (option (read))
        (cond
          ((or (not (numberp option)) (< option 1) (> option (length moves)))
          (progn
                  (format t "~%
_____________________________________________________________")
                  (format t "~%/                Escolha uma opcao valida
/")
                  (format t
"~%_____________________________________________________________~%")
                  (move-input node piece player)
          )
        )
        (t (nth (1- option) moves))
      )
    )
  )
)
```

Computer-only

```
(defun computer-only(max-time player &optional (node (make-node (empty-board))))
  (let* (
          (c1-moves    (expand-node node (operations) 1))
          (c1-pieces   (pieces-list node 1))
          (can-c1-play (or (null c1-moves) (= 0 (apply '+ c1-pieces))))        ; t
= can't play
          (c2-moves    (expand-node node (operations) -1))
          (c2-pieces   (pieces-list node -1))
          (can-c2-play (or (null c2-moves) (= 0 (apply '+ c2-pieces))))        ; t
= can't play
          (can-current-play (cond ((= player 1) can-c1-play) (t can-c2-play)))  ;
t = can't play
          ;(player-numb (cond ((= player 1) 1) (t 2)))
        )
    (cond
      ((and can-c1-play can-c2-play) (log-footer node))
; endgame (+ log.dat)
      (t
        (cond
          ((eval can-current-play)
            (progn
```

```
              (format t "~%~%________Sem Jogadas________~%~%")
              (computer-only max-time (- player) node)
            )
          )
          (t(let* (
                    (solution-nd (negamax max-time node 1 player))
                    (sol-node (get-solution-node solution-nd))
                  )
              (progn
                (log-file solution-nd player)
                (computer-only max-time (- player) (make-node (node-state sol-
node) nil (node-p1 sol-node) (node-p2 sol-node)))
              )
            )
          )
        )
      )
    )
  )
)
```

Run-computer-only

```
(defun run-computer-only()
  (let ((max-time (get-time-limit)))
    (progn
      (log-header max-time)
      (computer-only max-time 1)
    )
  )
)
```

Menus

Init-menu

- A função *init-menu* mostra o menu inicial do jogo.

```
(defun init-menu()
    (format t "~%_____________________________________________________________")
    (format t "~%\\                      BLOKUS DUO                            /")
    (format t "~%/                                                            \\")
    (format t "~%\\     1 - Humano vs Computador                              /")
    (format t "~%/       2 - Computador vs Computador                         \\")
    (format t "~%\\     0 - Sair                                              /")
    (format t
"~%/_____________________________________________________________\\~%~%>")
)
```

Start

- A função *start* dá inicio ao jogo, mostrando o menu inicial.

```
(defun start()
    (init-menu)
    (let ((option (read)))
      (if (and (numberp option) (>= option 0) (<= option 2))
        (cond
          ((equal option 1) (get-starter))
          ((equal option 2) (run-computer-only))
          ((equal option 0) (format t "~%Adeus!"))
        )

        (progn (print "Opção inválida. Tente novamente") (start))
      )
    )
)
```

Start-view

- A função *start-view* mostra o menu de escolha de quem inicia o jogo, humano (utilizador) ou computador.

```
(defun starter-view()
    (format t "~%_____________________________________________________________")
    (format t "~%\\                      BLOKUS DUO                            /")
    (format t "~%/                  Quem começa primeiro?                    \\")
    (format t "~%\\                    (tipo das peças)                        /")
    (format t "~%/     1 - Humano (1)                                         \\")
    (format t "~%\\      2 - Computador (2)                                    /")
    (format t "~%/     0 - Voltar                                             \\")
    (format t "~%\\_____________________________________________________________/~%~%>
")
)
```

Get-starter

- A função *get-starter*

```
(defun get-starter()
  (starter-view)
  (let ((option (read)))
    (cond
      ((or (< option 0) (> option 2)) (progn (format t "Insira uma opcao valida")
(get-starter)))
```

```
      ((= option 0) (start))
      (t option)
    )
  )
)
```

Time-limit-view

- A função *time-limit-view* mostra a mensagem para o utilizador escolher o tempo limite da jogada do computador.

```
(defun time-limit-view()
  (format t "~%_____________________________________________________________")
  (format t "~%\\                        BLOKUS DUO                          /")
  (format t "~%/          Defina o tempo limite para uma jogada            \\")
  (format t "~%\\             do computador (1 - 20) segundos.               /")
  (format t "~%\\                                                           \\")
  (format t "~%/     0 - Voltar                                            /")
  (format t "~%\\                                                         \\")
  (format t "~%/_____________________________________________________________/~%~%> ")
)
```

Get-time-limit

- A função *get-time-limit*

```
(defun get-time-limit()
  (progn (time-limit-view)
    (let ((option (read)))
      (cond
        ((= option 0) (start))
        ((or (not (numberp option)) (< option 1) (> option 20)) (progn (format t
"Insira uma opção válida") (get-time-limit)))
        (t option)
      )
    )
  )
)
```

Header

- A função *header*

```
(defun header(stream max-time)
  (format stream "~%---------------- INICIO -----------------~%max-time: ~a~%~%"
```

```
max-time)
)
```

Log-header

- A função *log-header*

```
(defun log-header(max-time)
  (progn
    (with-open-file (file (get-log-path) :direction :output :if-exists :append
:if-does-not-exist :create) (header file max-time))
    (header t max-time)
  )
)
```

Log-file

- A função *log-file*

```
(defun log-file(solution-nd color)
  (let* (
        (player (cond ((= color 1) 1) (t 2)))
        (node (get-solution-node solution-nd))
        (move (last-move node color))
        (visited-nodes (get-visited-nodes solution-nd))
        (cuts (get-cuts solution-nd))
        (alg-runtime (get-runtime solution-nd))
      )
    (progn
      (with-open-file (file (get-log-path) :direction :output :if-exists :append
:if-does-not-exist :create)
        (log-stream file (node-state node) player (first move) (second move)
visited-nodes cuts alg-runtime))
      (log-stream t (node-state node) player (first move) (second move) visited-
nodes cuts alg-runtime)
    )
  )
)
```

Log-stream

- A função *log-stream*

```
(defun log-stream (stream state player piece indexes nodes-visited cuts runtime)
  (progn
    (format-board state stream)
```

```
    (format stream "~%Jogador ~a ~%" player)
    (format stream "Jogou a peca ~a na posicao (~a , ~a)~%" piece (first indexes)
(second indexes))
    (format stream "Nos Analisados: ~a ~%" nodes-visited)
    (format stream "Numero de Cortes: ~a ~%" cuts)
    (format stream "Tempo de Execucao: ~a ~%" runtime)
  )
)
```

Log-footer

- A função *log-footer*

```
(defun log-footer(node)
  (progn
    (with-open-file (file (get-log-path) :direction :output :if-exists :append
:if-does-not-exist :create) (log-winner node file))
    (log-winner node t)
  )
)
```

Log-winner

- A função *log-winner*

```
(defun log-winner(node stream)
  (let ((pieces-p1 (pieces-list node 1)) (pieces-p2 (pieces-list node -1)))
    (format stream
"~%_____________________________________________________________")
    (format stream "~%\\                        BLOKUS DUO
/")
    (format stream "~%/
\\")
    (format stream "~%\\                         Vencedor
/")
    (format stream "~%/                            ~a                      \\"
(winner node))
    (format stream "~%\\
/")
    (format stream "~%/    Jogador 1: ~a pontos  vs  Jogador 2: ~a pontos      \\"
(count-points pieces-p1) (count-points pieces-p2))
    (format stream "~%\\      pecas: ~a            pecas: ~a         /" pieces-p1
pieces-p2)
    (format stream "~%/
\\")
    (format stream
"~%\\_____________________________________________________________/~%~%> ")
  )
)
```

Formatters

Format-board-line

- A função *format-board-line*

```
(defun format-board-line (board &optional (stream t))
  (cond
   ((null (first board)) "")
   (t (format stream "~a~%" (first board)))
   )
)
```

Format-board

- A função *format-board*

```
(defun format-board (board &optional (stream t))
  (cond
   ((null board) "")
   (t (format stream "~&" (append(format-board-line board stream) (format-board
(cdr board) stream))))
   )
)
```