Inteligência Artificial - Jogo do Cavalo

Manual Técnico

Autores:

- Henoch Mendes Vitureira Nº170221014
- Patrick Battisti Forsthofer Nº190200007

Docentes:

- Prof. Joaquim Filipe
- Eng. Filipe Mariano

Introdução

Este documento tem como o objetivo documentar tecnicamente a implentação em Common Lisp do jogo do cavalo, uma variante do problema do cavalo, cuja finalidade é processar um dado tabuleiro pontuado de modo a atinguir uma pontuação desejada e apresentar a solução do problema em questão, com recurso a algoritmos de procura. Os algoritmos implementados foram o Deapth-First Search, o Breadth-First Search e o A*.

A primeira parte do documento dá a conhecer a estrutra de ficheiros do programa assim como o conteúdo dos mesmos. Posteriormente, são demonstrados os algoritmos implementados e os objetos que compôem o projeto.

Para finalizar, são apresentadas as limitações técnicas do programa.

Ficheiros:

- **Project.lisp**:Carrega os outros ficheiros de código, escreve e lê ficheiros, e trata da interação com o utilizador.
- **puzzle.lisp**: Código relacionado com o problema.
- **search.lisp**: Código independente do problema. Contem a implementação dos algoritmos de Procura de Largura Primeiro (BFS), Procura do Profundidade Primeiro (DFS)e Procura do Melhor Primeiro (A*).

Ficheiro puzzle.lisp

Este ficheiro contém função que auxiliaram acessos e cálculo de modo a promover a modularidade e seleçao de informação no projeto.

Incialização dos Algoritmos

As funções abaixo foram as que serviram para a inicalização dos algoritmos. Nestes são pedidas todas as informações necessárias do utilizador para as funções poderem correr. Nomeadamente a posição inicial do

cavalo e a estratégia de remoção de números "duplos". Existe uma função deste tipo para cada algoritmo implementado.

```
(defun dfsearch (board)
"Initializes the DFS algorithm with a provided board"
  (let* (
         (start-pos (read-start-position))
         (start-points (car (successor-value (first start-pos) (second start-pos)
board)))
         (board-with-horse (replace-value (first start-pos) (second start-pos)
board T))
         (strategy (read-strategy))
         (open-nodes (list (make-root-node board-with-horse start-points nil
strategy)))
         (start-time (current-time))
         )
    (write-bfsdfs-statistics board (dfs open-nodes (list nil) (read-target-points)
(read-depth) strategy) start-time (current-time) 'DFS)
    )
  )

(defun bfsearch (board)
"Initializes the BFS algorithm with a provided board"
  (let* (
         (start-pos (read-start-position))
         (start-points (car (successor-value (first start-pos) (second start-pos)
board)))
         (board-with-horse (replace-value (first start-pos) (second start-pos)
board T))
         (strategy (read-strategy))
         (open-nodes (list (make-root-node board-with-horse start-points nil
strategy)))
         (start-time (current-time))
         )

    (write-bfsdfs-statistics board (bfs open-nodes (list nil) (read-target-points)
strategy) start-time (current-time) 'BFS)
    )
  )

(defun a*search (board heuristic)
"Initializes the A* algorithm with a provided board"

  (let* (
         (start-pos (read-start-position))
         (start-points (car (successor-value (first start-pos) (second start-pos)
board)))
         (board-with-horse (replace-value (first start-pos) (second start-pos)
board T))
         (strategy (read-strategy))
         (open-nodes (list (make-root-node board-with-horse start-points nil
```

```
strategy)))
         (start-time (current-time))
         )

    (write-bfsdfs-statistics board (a* open-nodes (list nil) (read-target-points)
strategy heuristic) start-time (current-time) 'A*)
    )
  )
```

Construção de Nós

A estrutura de nós implementada foi a seguinte: (node)::= ((state) (parent-node) (depth) (h) (f)). Nesta estrutura, "state" representa um estado, parent-node o nó antesessor, depth a profundidade do nó atual, "h" o valor heurístico e "f" o valor da soma entre g e h, isto é, o custo e o seu valor heurístico. Os métodos abaixo serviram para construir nós, nós solução, que seriam utilizados na representação de estatísticas, e nós raiz.

```
;;<node>::= (<state>  <parent-node> <depth> <h> <f>)
;;<state>::= (<board> <point-sum>)
(defun make-node (board points parent-node &optional (depth 0) (f nil))
"constructs a node with the structure <node>::= (<state>  <parent-node> <depth>
<h> <f>)"
  (list (list board points) parent-node depth f)
  )

(defun make-solution-node (solution-path board open-list closed-list &optional
depth target-points current-points)
"constructs a list that is used to write the statistics"
   (list solution-path board (length open-list) (length closed-list) depth target-
points current-points)
  )

(defun make-root-node (board-placed-horse current-points parent &optional
strategy)
"constructs root node"
  (make-node (remove-simmetric current-points board-placed-horse strategy)
current-points nil 0 0)
  )
```

Tabuleiros

Um tabuleiro representa-se por uma lista de listas, em que as listas contidas pela primeira representam as linhas do tabuleiro. Para obter os tabuleiros que o projeto requisitava que fossem resolvidos e documentados, foram criadas funções do seguinte tipo:

```
(defun board-a ()
  "Returns board A, GOAL: 70"
```

```
  '(
    (02 20 44 nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil nil)
    (nil 03 30 nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil nil)
    (nil nil nil 22 nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil nil)
    )
  )
```

Seletores

Foram implementados diversos seletores para a estrutura do nó, de modo a facilitar a seleção de dados do mesmo.

```
;; parent-node
(defun parent-node (node)
"gets the parent of a node"
  (cond
   ((null node) nil)
   (t (second node))
  )
)
;; test: (parent-node (test-node))
;; resultado: NIL

;; stade-node
(defun state-node (node)
"gets the depth of a node"
  (first node)
)

;; node-state-board
(defun node-state-board (node)
"gets the state of a node"
  (first (state-node node))
)

;; node-state-point-sum
(defun node-state-point-sum (node)
"gets the summed points of a node"
        (cond
        ((null node) 0)
        (t (second (state-node node)))
        )
)
```

```
;; node-f
(defun node-f (node)
"gets the value of a node"
        (cond
                ((null node) nil)
                (t (fourth node))
        )
)
```

Além de outros métodos seletores, foi implementado um método para converter uma posição do tabuleiro numa posição de xadrez correspondente. esta serviu para mostrar a solução de um problema.

```
(defun position-to-chess (horse-pos)
"returns a posititon converted into a chess position"
        (list (string (code-char (+ 65 (second horse-pos)))) (+ 1 (first horse-
pos)))
)
```

Ficheiro search.lisp

Este ficheiro, tal como descrito anteriormente, contém a implementação dos algoritmos.

Algoritmos

Abaixo encontra-se a implementação do algoritmo **Deapth-First Search**, isto é, procura por profundidade. Inicialmente o algoritmo atribui ao nó atual o primeiro nó da lista de abertos, que é fornecido no chamamento da função, e uma lista de nós fechados, que começa vazia. O critério de paragem do algoritmo é dado pela confirmação da verificação de se a lista de nós abertos está vazia ou se chegou ao número de pontos desejado. Caso não haja solução encontada num iteração, o algoritmo é executado recursivamente, adicionando os sucessores do nó atual ao início da lista de nós abertos e o nó atual à lista de nós fechados (expandidos).

```
(defun dfs (open-list closed-list target-points max-depth strategy)
  (cond ((null open-list) (format t "No solution found"))
        (t
         (let*  (
                 (node (car open-list))
                 (horse-pos (horsep node))
                 (current-board (horsep node))
                 (current-successors (successors (first horse-pos) (second horse-
pos) node max-depth strategy 0 (append closed-list open-list)))
                 )

           (cond
            ((<= target-points (node-state-point-sum node))
                  (format-output node "DFS")
```

```
                (make-solution-node (reverse (get-solution-path node)) (node-
state-board node) open-list closed-list max-depth target-points (node-state-point-
sum node))
                )
            (t
             (dfs
              (append current-successors (cdr open-list))
              (append (cdr closed-list) (list node))
              target-points
              max-depth
              strategy
              )
             ))))))
```

O algoritmo **Breadth-First Search** foi implementado de forma idêntica à do Deapth-First Search, tirando na forma de como concatenar os os sucessores, que são colocados no final da lista de abertos.

A algoritmo A* foi implementado segundo a sua definição, tirando a parte onde há a atualização do valor de f de todos os nós, abertos ou fechados e ocorre a migração de fechados para abertos, caso o valor de f tenha sido alterado para menor.

Operadores

Os operadores são executados numa função apenas, onde é enviada a posição atual do cavalo e são sumados e subtraidos a estes valores as 8 posições possíveis do cavalo no estado seguinte.

```
(defun successors (line-index column-index node max-depth strategy &optional
(target-points 0) closed-open-list heuristic)
  "generates successors"
  (cond  (
          (>= (depth-node node) max-depth) nil)
         (t (append
             (successor-avaliable (- line-index 2) (- column-index 1) node
strategy target-points closed-open-list heuristic)
             (successor-avaliable (- line-index 2) (+ column-index 1) node
strategy target-points closed-open-list heuristic)
             (successor-avaliable (+ line-index 2) (- column-index 1) node
strategy target-points closed-open-list heuristic)
             (successor-avaliable (+ line-index 2) (+ column-index 1) node
strategy target-points closed-open-list heuristic)
             (successor-avaliable (- line-index 1) (- column-index 2) node
strategy target-points closed-open-list heuristic)
             (successor-avaliable (- line-index 1) (+ column-index 2) node
strategy target-points closed-open-list heuristic)
             (successor-avaliable (+ line-index 1) (- column-index 2) node
strategy target-points closed-open-list heuristic)
             (successor-avaliable (+ line-index 1) (+ column-index 2) node
strategy target-points closed-open-list heuristic)
             )
            )
```

```
          )
   )
```

Geração de Sucessores

A função abaixo foi a utilizada para a geração de sucessores no jogo do cavalo. Recebe as coordenadas de uma posição, e verifica se o cavalo pode visitar a mesma. Caso possa, devolve um nó sucessor.

```
(defun successor-avaliable (line-index column-index node strategy target-points
closed-open-list heuristic)
   (cond
   ((or (< line-index 0) (> line-index 9)) nil)
   ((or (< column-index 0) (> column-index 9)) nil)
   ((null (car (successor-value line-index column-index (node-state-board node))))
nil)
   (t
    (let* (
           (horse-pos (horsep node))
           (node-board (node-state-board node))
           (points-to-sum (car (successor-value line-index column-index node-
board)))
           (board-no-horse (replace-value (first horse-pos) (second horse-pos)
(remove-node points-to-sum node-board)))
           (board-no-simmetric (remove-simmetric points-to-sum board-no-horse
strategy))
           (board-to-be (replace-value line-index column-index board-no-simmetric
T))
           (points (+ (node-state-point-sum node) points-to-sum))
           (h (get-heuristic-fn board-no-horse (- target-points points) points-to-
sum strategy heuristic (depth-node node)))
           (f (+ (depth-node node) h))
           )

      (cond ((node-in-open-closed board-to-be closed-open-list) nil)
            (t (list (make-node board-to-be points node (1+ (depth-node node))
f)))
            )
      )
    )
   )
  )
```

Remoção do Simétrico

Esta é a função que se ocupa da remoção de valor simétrico ou "duplo" num tabuleiro, em função da escolha da estratégia do utilizador como máximo ou mínimo.

```
(defun simmetric-value (value board &optional (strategy 'max))
  (let* (
         (simmetric (reverse (write-to-string value)))
         )

    (cond
     ((< value 10) (* value 10))
     ((equal (parse-integer simmetric) value) (min-max-asymmetric-node board
strategy))
     (t (parse-integer simmetric))
     )
    )
  )
```

Ficheiro project.lisp

Começar Jogo

A função seguinte ocupa-se de iniciar o jogo do cavalo, pedindo os inputs necessários ao utilizador.

```
(defun start-game()
  (let (
        (mode (read-mode))
        (algorithm (read-algorithm))
        )

    (cond
     ((equal mode 'problems)
      (cond ((equal algorithm 'DFS)
             (dfsearch (read-problem))
             )
            ((equal algorithm 'BFS)
             (bfsearch (read-problem))
             )
            ((equal algorithm 'A*)
             (a*search (read-problem) (read-heuristic))
             )
            )
      )
     ((equal mode 'exercises)
      (cond ((equal algorithm 'DFS)
             (dfsearch (read-exercise))
             )
            ((equal algorithm 'BFS)
             (bfsearch (read-exercise))
             )
            ((equal algorithm 'A*)
             (a*search (read-exercise) (read-heuristic))
             )
            )
```

```
        )

       (T (format t "Thankyou for Playing!")
          t))
     ))
```

Leitura de Inputs

Foram criadas várias funções de navegação, de modo a tornar possível a leitura de inputs do utilizador. O género de função abaixo foi implementada para todo o tipo de leitura de input.

```
(defun read-algorithm()
"Allows to make the reading of the algorithm to use"
  (progn
    (show-algorithm)
    (let ((answer (read)))
      (cond ((= answer 1) 'DFS)
            ((= answer 2) 'BFS)
            ((= answer 3) 'A*)
            ((= answer 4) nil)
            (T (format t "Insert a valid option please!")
              (read-algorithm))
    )))
  )

(defun show-algorithm()
  "Reads the algorithm choice"
  (progn
    (format t "    ~%---------------------HORSE GAME---------------------------")
    (terpri)
    (format t "   ~%|                Choose an algorithm:
|")
    (terpri)
    (format t "   ~%|                1 - Depth-First                          |")
    (format t "   ~%|                2 - Breadth-First                        |")
    (format t "   ~%|                3 - A*                                   |")
    (terpri)
    (format t "   ~% ------------------------------------------------------------
~%~%> ")
    )
  )
```

Estatísticas

Abaixo encontram-se o estudo estatístico do Problema A.

Board	Algorithm	Starting time	Ending Time	Number of Generated Nodes	Number of Expanded Nodes	Penetrance Level	Median Branching Factor

Board	Algorithm	Starting time	Ending Time	Number of Generated Nodes	Number of Expanded Nodes	Penetrance Level	Median Branching Factor
A	BFS	22:13:34	22:13:35	4	3	0.75	1.1421085
A	DFS	22:14:7	22:14:13	4	3	0.75	1.1421085
A	A*	22:14:34	22:14:37	4	3	0.75	1.1421085
A	A*	22:14:44	22:14:46	4	3	0.75	1.1421085

Limitações

Em termos de limitações, o programa poderia estar mais modular, com menos código repetido, e melhorar refactorado no geral. O algoritmo A* não executa o passo da atualização dos valores de F nos nós abertos e fechados, que não foi implementado.