Manual Técnico - Jogo do Cavalo

Inteligência Artificial

- **Docente:** Filipe Mariano
- **Realizado por:** João Jardin Nº 20180002 SW-04

Índice

- Introdução
- Arquitetura do Sistema
- Entidades e sua implementação
- Algoritmos
- Análise comparativa dos algoritmos
- Limitações e requisitos não implementados

1. Introdução

Para verificar a possibilidade de o cavalo atingir um objetivo proposto implementaram-se três algoritmos:

- Algoritmo de Busca em Largura (BFS)
- Algoritmo de Busca em Profundidade (DFS)
- Algoritmo A*

2. Arquitetura do Sistema

Este programa é composto por 4 ficheiros:

- **projeto.lisp**
  - Carrega os outros ficheiros de código, escreve e lê ficheiros, e trata da interação com o utilizador.
- **procura.lisp**
  - Contem a implementação dos algoritmos a utilizar neste projeto.
- **puzzle.lisp**
  - Contém o código relacionado com o problema, tais como operadores, heurística....
- **problemas.dat**
  - Contem os problemas do projeto

Como referido anteriormente o ficheiro puzzle.lisp contém a implementação do código relacionado com o problema, como por exemplo os operadores, que vão ser descritos mais adiante. Esta informação é essencial para o ficheiro procura.lisp, pois os algoritmos vão usar a implementação de funções como sucessores e da estrutura de dados de um nó que também consta no mesmo ficheiro.

O projeto.lisp apresenta uma maneira mais *user-friendly* do utilizador poder utilizar de forma simultânea os algoritmos implementados na procura.lisp.

Resumindo: **- puzzle.lisp -> procura.lisp -> projeto.lisp**

3. Entidades e a sua implementação

Para este problema o elemento principal é um tabuleiro.

O tabuleiro tem as dimensões 10x10 em que os valores de cada casa são entre 00 e 99, sem repetição. O cavalo é representado por 'T' no tabuleiro e as casas que o cavalo não pode visitar são marcadas como 'NIL'. A imagem seguinte representa uma ilustração do mesmo:

```
((94 25 54 89 21 8 36 14 41 96)
 (78 47 56 23 5 49 13 12 26 60)
 (0 27 17 83 34 93 74 52 45 80)
 (69 9 77 95 55 39 91 73 57 30)
 (24 15 22 86 1 11 68 79 76 72)
 (81 48 32 2 64 16 50 37 29 71)
 (99 51 6 18 53 28 7 63 10 88)
 (59 42 46 85 90 75 87 43 20 31)
 (3 61 58 44 65 82 19 4 35 62)
 (33 70 84 40 66 38 92 67 98 97))

((NIL 25 54 89 21 8 36 14 41 96)
 (78 47 56 23 5 49 13 12 26 60)
 (0 T 17 83 34 93 74 52 45 80)
 (69 9 77 95 55 39 91 73 57 30)
 (24 15 22 86 1 11 68 79 76 NIL)
 (81 48 32 2 64 16 50 37 29 71)
 (99 51 6 18 53 28 7 63 10 88)
 (59 42 46 85 90 75 87 43 20 31)
```

```
 (3 61 58 44 65 82 19 4 35 62)
 (33 70 84 40 66 38 92 67 98 97))
```

No segundo tabuleiro, o cavalo movimentou-se para a casa de valor 27. A casa inicial ficou marcada como NIL e a casa de valor 72 também ficou marcada como NIL, devido a uma das regras específicas deste jogo que vão ser abordadas de seguida.

3.1. Regras do Jogo

3.1.1. Regra do Número Simétrico

Se um número tiver dois digitos diferentes, o seu simétrico é o seu inverso. Considere-se o exemplo anterior:

- O cavalo movimentou-se para a casa de valor 27
- O seu simétrico é o inverso de 27, neste caso 72.

Se o cavalo move-se para uma casa da qual o seu simétrico ainda não tiver sido removido, então do momento em que o cavalo se desloca para essa casa, o numero simétrico é removido, ficando então a estar inacessível para o resto do jogo.

Para verificar qual o simétrico de um número e remove-lo do tabuleiro desenvolveram-se duas funções:

```
(defun numero-simetrico-p (numero)
  (and (numberp numero)
       (+ (* (mod numero 10) 10) (floor numero 10))))

 (defun remover-simetrico (simetrico tabuleiro)
  (mapcar (lambda (linha) (mapcar (lambda (celula) (if (eql celula simetrico) nil celula)) linha))
tabuleiro))
```

Em que:

- `numero-simetrico-p` devolve o simétrico de um número
- `remover-simetrico` remove o simétrico caso exista no tabuleiro.

3.1.2. Regra do Número Duplo

Um número duplo no contexto deste jogo é um número que tem dois dígitos repetidos, como por exemplo 11, 22, 33... Se um cavalo movimentar-se para uma casa que tenha valor de um número duplo, por defeito o maior duplo da casa também é removido.

```
((NIL 25 54 89 21 8 36 14 41 96)
 (78 47 T 23 5 49 13 12 26 60)
 (0 27 17 83 34 93 74 52 45 80)
 (69 9 95 77 55 39 91 73 57 30)
 (24 15 22 86 1 11 68 79 76 72)
 (81 48 32 2 64 16 50 37 29 71)
 (99 51 6 18 53 28 7 63 10 88)
 (59 42 46 85 90 75 87 43 20 31)
 (3 61 58 44 65 82 19 4 35 62)
 (33 70 84 40 66 38 92 67 98 97))

((NIL 25 54 89 21 8 36 14 41 96)
 (78 47 NIL 23 5 49 13 12 26 60)
 (0 27 17 83 34 93 74 52 45 80)
 (69 9 95 T 55 39 91 73 57 30)
 (24 15 22 86 1 11 68 79 76 72)
 (81 48 32 2 64 16 50 37 29 71)
 (NIL 51 6 18 53 28 7 63 10 88)
 (59 42 46 85 90 75 87 43 20 31)
 (3 61 58 44 65 82 19 4 35 62)
 (33 70 84 40 66 38 92 67 98 97))
```

O cavalo movimentou-se para a casa de valor 77 e então o maior duplo presente no tabuleiro, neste caso sendo o 99, também foi removido do tabuleiro. Para esta validação foram implementadas as seguintes funções:

```
(defun numero-duplo-p (numero)
(and (numberp numero)
     (not (= numero 0))
     (= (mod numero 10) (floor numero 10))))


(defun lista-duplos (tabuleiro)
(cond
  ((null tabuleiro) nil)
  ((numero-duplo-p (car tabuleiro))
   (cons (car tabuleiro) (lista-duplos (cdr tabuleiro))))
  (t (lista-duplos (cdr tabuleiro)))))

(defun remover-duplo (maior-duplo tabuleiro)
(mapcar (lambda (linha) (mapcar (lambda (celula) (if (eql celula maior-duplo) nil celula)) linha))
tabuleiro))
```

Em que:

- `numero-duplo-p` Verifica se um dado número é um número duplo
- `lista-duplos` Devolve a lista de números duplos presentes no tabuleiro.
- `remover-duplos` Remove o duplo no tabuleiro (de notar que esta função precisa da função já existente no Common Lisp `max` para corretamente remover o maior duplo presente no tabuleiro).

De referir como regras adicionais, que o cavalo não pode referir para uma casa que esteja fora dos limites do tabuleiro e não pode visitar uma casa que esteja inacessível (marcada como `NIL`). Para tal implementou-se a função `movimento-valido` seguinte função que lida com todos os casos

```
(defun movimento-valido (linha coluna tabuleiro)
"Verifica se um movimento do cavalo é válido no tabuleiro."
(cond
  ((or (null linha) (null coluna) (null tabuleiro)) nil)
  ((or (> linha 9) (< linha 0) (> coluna 9) (< coluna 0)) nil)
  ((null (celula linha coluna tabuleiro)) nil)
  (t t)))
```

3.2. Representação do Estado

Para implementação de algoritmo foi necessário usar uma estrutura de dados capaz de conter as informações necessários para o mesmo.

3.2.1. Nó

A estrutura de dados escolhida foi um nó em que:

- O **primeiro** elemento corresponde ao **estado** do nó, neste caso especifíco representa o tabuleiro;
- O **segundo** elemento representa o **nó pai**;
- O **terceiro** elemento a **heurística** do nó;
- O **quarto** elemento a **profundidade** do nó;
- O **quinto** elemento corresponde aos **pontos acumulados**;
- O **sexto** elemento corresponde ao **objetivo de pontos** (constante).

```
(defun criar-no (tabuleiro pai h profundidade pontos objetivo)
  "Cria um nó com o estado do tabuleiro, o nó pai, a heurística, a profundidade, os pontos e o
```

```
objetivo."
  (list tabuleiro pai h profundidade pontos objetivo))

(defun estado-do-no (no)
"Retorna o estado atual (tabuleiro) do nó"
  (first no))

(defun pai-do-no (no)
"Retorna o nó-pai do nó"
  (second no))

(defun h-do-no (no)
"Retorna o valor heurístico do nó"
  (third no))

(defun profundidade-do-no (no)
"Retorna a profundidade do nó"
  (fourth no))


(defun pontos-do-no (no)
"Retorna os pontos acumulados do nó."
  (fifth no))


(defun objetivo-do-no (no)
"Retorna o objetivo de pontos (constante)."
  (sixth no))
```

3.2.2. Operadores

Os operadores representam os movimentos possíveis num determinado estado. Para o Problema do Cavalo o máximo de movimentos possíveis serão 8, desde que essas casas não tenham sido ainda visitadas ou removidas pela regra dos simétricos ou duplos.

De forma a evitar código duplicado e atendendo que os operadores tinham o mesmo comportamento e apenas diferiam no movimento, criou-se uma função `movimentar-cavalo` que recebe o tabuleiro, a linha e a coluna para o qual o cavalo se move.

```
(defun movimentar-cavalo (tabuleiro deslocamento-linha deslocamento-coluna)
  (let* ((tabuleiro-com-cavalo (posicionar-cavalo tabuleiro))
         (cavalo-posicao (posicao-cavalo tabuleiro-com-cavalo))
         (nova-linha (+ (first cavalo-posicao) deslocamento-linha))
         (nova-coluna (+ (second cavalo-posicao) deslocamento-coluna)))
    (if (movimento-valido nova-linha nova-coluna tabuleiro-com-cavalo)
        (let ((casa-destino (celula nova-linha nova-coluna tabuleiro-com-cavalo)))
          (cond
            ((and casa-destino (numero-duplo-p casa-destino))
             (let* ((duplos (lista-duplos (concatena tabuleiro-com-cavalo)))
                    (maior-duplo (if duplos (apply #'max duplos))))
               (values (remover-duplo maior-duplo (substituir nova-linha nova-coluna (substituir
(first cavalo-posicao) (second cavalo-posicao) tabuleiro-com-cavalo nil) 'T))
                       (if (numberp casa-destino) casa-destino 0))))
            ((and casa-destino (numero-simetrico-p casa-destino))
             (let ((simetrico (numero-simetrico-p casa-destino)))
               (values (remover-simetrico simetrico (substituir nova-linha nova-coluna (substituir
(first cavalo-posicao) (second cavalo-posicao) tabuleiro-com-cavalo nil) 'T))
                       (if (numberp casa-destino) casa-destino 0))))
            (t (values (substituir nova-linha nova-coluna (substituir (first cavalo-posicao) (second
cavalo-posicao) tabuleiro-com-cavalo nil) 'T)
                       (if (numberp casa-destino) casa-destino 0)))))
        nil)))
```

De forma que:

```
(defun operador-1 (tabuleiro)
  "Move o cavalo 2 linhas para baixo e 1 coluna para a direita."
  (movimentar-cavalo tabuleiro 2 1))

(defun operador-2 (tabuleiro)
  "Move o cavalo 1 linha para baixo e 2 colunas para a direita."
  (movimentar-cavalo tabuleiro 1 2))

(defun operador-3 (tabuleiro)
  "Move o cavalo 1 linha para cima e 2 colunas para a direita"
  (movimentar-cavalo tabuleiro -1 2))

(defun operador-4 (tabuleiro)
  "Move o cavalo 2 linhas para cima e 1 coluna para a direita"
  (movimentar-cavalo tabuleiro -2 1))

(defun operador-5 (tabuleiro)
  "Move o cavalo 2 linhas para cima e 1 coluna para a esquerda"
  (movimentar-cavalo tabuleiro -2 -1))

(defun operador-6 (tabuleiro)
  "Move o cavalo 1 linha para cima e 2 colunas para a esquerda"
  (movimentar-cavalo tabuleiro -1 -2))

(defun operador-7 (tabuleiro)
  "Move o cavalo 1 linha para baixo e 2 colunas para a esquerda"
  (movimentar-cavalo tabuleiro 1 -2))

(defun operador-8 (tabuleiro)
  "Move o cavalo 2 linhas para baixo e 1 coluna para a esquerda"
  (movimentar-cavalo tabuleiro 2 -1))


CL-USER 2 > (pprint(operador-1(tabuleiro-teste)))

((NIL 25 54 89 21 8 36 14 41 96)
 (78 47 56 23 5 49 13 12 26 60)
 (0 T 17 83 34 93 74 52 45 80)
 (69 9 77 95 55 39 91 73 57 30)
 (24 15 22 86 1 11 68 79 76 NIL)
 (81 48 32 2 64 16 50 37 29 71)
 (99 51 6 18 53 28 7 63 10 88)
 (59 42 46 85 90 75 87 43 20 31)
 (3 61 58 44 65 82 19 4 35 62)
 (33 70 84 40 66 38 92 67 98 97))
```

Ao termos os operadores pode-se então implementar as funções de gerar um novo sucessor e os sucessores respetivamente:

```
(defun novo-sucessor (no operador)
  (multiple-value-bind (novo-tabuleiro pontos-ganhos)
      (funcall operador (estado-do-no no))
    (when novo-tabuleiro
      (let ((nova-profundidade (1+ (profundidade-do-no no)))
            (pontos-acumulados (+ (pontos-do-no no) pontos-ganhos))
            (objetivo (objetivo-do-no no)))
        (let ((nova-heuristica (heuristica novo-tabuleiro pontos-acumulados objetivo)))
          (criar-no novo-tabuleiro no nova-heuristica nova-profundidade pontos-acumulados
objetivo))))))
```

```
(defun sucessores (no lista-op &optional algoritmo maxprofundidade)
  (cond
    ((and (equal algoritmo 'dfs) maxprofundidade (= (profundidade-do-no no) maxprofundidade))
     nil)
    (t (let ((sucessores (remove-if #'null (mapcar #'(lambda (operador) (novo-sucessor no operador))
lista-op))))
    sucessores))))
```

4. Algoritmos

4.1. BFS

```
(defun bfs (no-inicial objetivo-func sucessores-func operadores)

  (setf *abertos* (list no-inicial))
  (setf *fechados* nil)

  (cond ((funcall objetivo-func no-inicial) no-inicial)
        (t
         (let ((ultimo-no-expandido nil))
           (loop
             (when (null *abertos*)
               (return ultimo-no-expandido))

             (let ((no-atual (pop *abertos*)))
               (setf ultimo-no-expandido no-atual)

               (when (funcall objetivo-func no-atual)
                 (return no-atual))

               (let ((sucessores (funcall sucessores-func no-atual operadores nil nil)))
                 (dolist (sucessor sucessores)
                   (unless (no-existep sucessor *abertos*)
                     (unless (no-existep sucessor *fechados*)
                       (push sucessor *abertos*)))))

               (push no-atual *fechados*)))))))
```

4.2. DFS

```
(defun dfs (no-inicial objetivo-func sucessores-func operadores &optional (profundidade-maxima 8))
  (setf *abertos* (list no-inicial))
  (setf *fechados* nil)

  (cond ((funcall objetivo-func no-inicial) no-inicial)
        (t
  (let ((ultimo-no-expandido nil))
    (loop while *abertos* do
      (let ((no (pop *abertos*)))
        (setf ultimo-no-expandido no)

        (when (or (funcall objetivo-func no)
                  (>= (profundidade-do-no no) profundidade-maxima))
          (return-from dfs no))
```

```
        (push no *fechados*)

        (let ((sucessores (funcall sucessores-func no operadores)))
          (dolist (sucessor sucessores)
            (unless (or (no-existep sucessor *abertos*)
                        (no-existep sucessor *fechados*))
              (setf *abertos* (cons sucessor *abertos*))))))
    ultimo-no-expandido))))
```

4.3. A*

Para o algoritmo A* a função heuristica foi a heuristica base do projeto:

h(x) = o(x)/m(x)

- `m(x)` é a média por casa dos pontos que constam no tabuleiro x,
- `o(x)` é o número de pontos que faltam para atingir o valor definido como objetivo.

```
(defun a* (no-inicial objetivo-func sucessores-func operadores h-func)
  "Executa a busca pelo algoritmo A*."
  (setf *abertos* (list no-inicial))
  (setf *fechados* nil)

  (cond ((funcall objetivo-func no-inicial) no-inicial)
        (t
  (let ((ultimo-no-expandido nil))
    (loop
      (when (null *abertos*)
        (return ultimo-no-expandido))

      (let ((no-atual (reduce (lambda (n1 n2)
                                (if (< (no-custo n1) (no-custo n2))
                                    n1
                                    n2))
                              *abertos*)))
        (setf ultimo-no-expandido no-atual)
        (setf *abertos* (remove no-atual *abertos*))

        (when (funcall objetivo-func no-atual)
          (return no-atual))

        (push no-atual *fechados*)

        (let ((sucessores (funcall sucessores-func no-atual operadores 'a*)))
          (dolist (sucessor sucessores)
            (unless (or (member sucessor *abertos*)
                        (member sucessor *fechados*))
              (push sucessor *abertos*)))))))))
```

5. Análise comparativa dos algoritmos

Após a implementação dos algoritmos verificou-se a eficiência dos mesmos para cada um dos problemas do projeto:

De relembrar:

Problema	Objetivo
A	70 pontos

Problema	Objetivo
B	60 pontos
C	270 pontos
D	600 pontos
E	300 pontos
F	2000 pontos

5.1. BFS

Problema	Nós Gerados	Nós Expandidos	Profundidade	Pontos Acumulados	Penetrância	Fator de Ramificação	Objetivo Atingido?
A	3	3	2	49	0.6666667	1.3027556	Não
B	9	9	9	65	1.0	0.9999896	Sim
C	16	13	5	272	0.3125	1.4166187	Sim
D	24	12	11	662	0.45833334	1.1258245	Sim
F	121	43	39	2017	0.32231405	1.0589038	Sim

5.2. DFS

nota: Para o DFS considerou-se uma profundidade máxima de 40.

Problema	Nós Gerados	Nós Expandidos	Profundidade	Pontos Acumulados	Penetrância	Fator de Ramificação	Objetivo Atingido?
A	3	3	2	49	0.6666667	1.3027556	Não
B	9	9	9	65	1.0	0.9999896	Sim
C	16	13	5	272	0.3125	1.4166187	Sim
D	24	12	11	662	0.45833334	1.1258245	Sim
F	134	49	39	2005	0.29104477	1.0545231	Sim

5.3. A*

Problema	Nós Gerados	Nós Expandidos	Profundidade	Pontos Acumulados	Heurística	Penetrância	Fator de Ramificação	Objetivo Atingido?
A	3	3	2	49	1.5	0.6666667	1.3027556	Não
B	9	9	9	65	0	1.0	0.9999896	Sim
C	18	15	5	272	0.06567164	0.2777778	2.5766993	Sim
D	43	24	9	616	0.41558442	0.20930232	1.306419	Sim
F	123	30	27	2002	0.048683554	0.2195122	1.0954608	Sim

De notar:

- Não foi possível realizar o problema-E;
- O problema F pressupõe um tabuleiro aleatório gerado, pelo que para cada um dos algoritmos, o tabuleiro aleatório gerado foi diferente.

De uma maneira geral pode-se verificar que os algoritmos para os problemas apresentados comportam-se de maneira semelhante. Outro ponto a realçar é que quando o problema envolve mais nós, o algoritmo A* é mais eficaz do que os algoritmos BFS e DFS,

tomando por exemplo, o problema F em que o A* encontra o nó-solução a uma profundidade de 27, uma menor profundidade face aos outros dois algoritmos.

6. Limitações e Requisitos não implementados

- Nenhum dos algoritmos consegue resolver o problema E, ficando qualquer um dos algoritmos em loop infinito;
- Não foi implementada uma segunda heuristica;
- Não foram implementados os algoritmos SMA*, IDA* e RBFS apesar destes serem opcionais;
- Apesar de terem sido implementadas funções de leitura para o ficheiro `problemas.dat` sempre que é adicionado um problema tem que se criar manualmente um nó para o problema.
- Para executar qualquer um dos algoritmos é necessário aumentar o tamanho da *stack*;
- Atendendo que não se conseguiu de uma maneira arbirtrária modificar o caminho do ficheiro `problemas.dat` para o utilizador ler o ficheiro, terá de ir ao ficheiro `projeto.lisp` e na função:

  ```
  (defun obter-problemas ()
    (make-pathname :host "c"
                   :directory '(:absolute "Users" "PC" "Desktop" "Universidade" "IA2023" "IA"
  "Projeto" "lisp")
                   :name "problemas"
                   :type "dat"))
  ```

  Modificar o caminho conforme necessário.