Projeto Nº1: Época Normal - Fase nº 1

Inteligência Artificial 19/20

Prof. Joaquim Filipe

Eng. Filipe Mariano

Jogo do Cavalo (Knight Game)

Manual Técnico

Realizado por:

João Gomes - 150221001

André Gastão - 130221037

21 de Dezembro de 2019

Indice

Introdução

Este documento é escrito recorrendo á linguagem de marcação markdown, servindo como relatório do manual técnico do projeto Jogo do Cavalo (Knight Game) que é uma variante do problema matemático conhecido como o Passeio do Cavalo (Knight's Tour)

No âmbito da unidade curricular de Inteligência Artificial, foi nos proposto o projecto do jogo "Jogo do Cavalo", onde esta versão do jogo consiste num tabuleiro com 10 linhas e 10 colunas.

O objectivo deste projeto é resolver todos os problemas descritos no anexo do enunciado de A) a F). A resolução dos problemas mencionados será implementada na linguagem de programação funcional Common Lisp, utilizando todos os conhecimentos adquiridos na unidade curricular até ao momento, a fim de dar uma solução apropriada do problema.

Neste documento serão descritas detalhadamente todas as metricas de desenvolvimento usadas e funções implementadas.

Arquitetura do Sistema

O sistema do Jogo Do Cavalo foi implementado em linguagem LISP, utilizando o IDE LispWorks. A estrutura do projeto é composta por 4 ficheiros:

- projeto.lisp - Interação com o utilizador, escrita e leitura de ficheiros.
- puzzle.lisp - Implementação da resolução do problema incluindo seletores, operadores heuristicas e outras funcõess auxiliares.
- procura.lisp - Implementação dos algoritmos de procura BFS, DFS e A*.
- problemas.dat - Funções com os problemas de A) a F).
- solucao.dat - Que é o output descrito para cada um dos problemas solucionados com os algoritmos identificados por data, algoritmo, problema etc.

Entidades e sua implementação

Tabuleiro

O tabuleiro consiste numa apresentação sob a forma de uma lista de listas em LISP, composta por atomos, em que cada atomo representa uma casa com um valor numérico,o tabuleiro é representado por 10 linhas de 10 colunas.

Temos assim ao todo 6 problemas que são os tabuleiros de A) a F) implementados no ficheiro problemas.dat.

Em que o problema F é criado com o mesmo presuposto dos anteriores e difere na alocação aleatória e sem repetição dos atómos.

Exemplo de um Estado Inicial (Problema F)

```
((07 71 87 49 12 65 79 11 51 62)
 (22 09 95 70 56 41 04 86 78 48)
 (06 00 29 74 10 16 44 57 38 03)
 (19 39 23 21 08 13 14 83 46 80)
 (42 37 24 31 69 36 64 63 92 72)
 (26 34 27 30 98 01 52 45 55 73)
 (35 85 47 53 43 91 77 81 68 89)
 (84 61 97 54 88 18 33 17 40 76)
 (58 32 94 28 02 75 59 90 66 50)
 (82 60 25 93 05 15 67 96 20 99))
```

Funções e Regras do Jogo

Nesta secção do documento, vamos descrever as funcões e regras implementadas no projecto sabendo que os movimentos apenas podem se realizar em posições disponiveis do tabuleiro.

Regra do Simétrico

A regra do Simétrico coloca o simetrico do valor da casa do movimento como *"indisponível" (Nil)*.

```
(defun regra-simetrico (numero)
  "Retorna o numero simetrico do argumento recebido"
)
```

Regra do Duplo

A regra do duplo verifica se o valor da casa do movimento realizado é um número duplo. Se o número for duplo, coloca a casa com o maior numero duplo disponivel no tabuleiro a *indisponível*.

```
(defun regra-duplo (numero)
  "Retorna T se for simetrico senão NIL"
)

(defun Duplos-Disponiveis (tabuleiro)
  "Retorna a lista de Duplos Disponiveis Ordenada pelo o maior valor"
)
```

Regra do Cavalo

Regra do cavalo como a apelidamos, permite colocar o cavalo em Jogo, normalmente a regra é usada sempre no início da procura do tabuleiro em questão.

Esta Regra tem o pressuposto que o cavalo pode ser colocado numa casa com valor numérico.

Apenas se pode colocar o cavalo na primeira linha do tabuleiro.

A solução é um conjunto de sucessões de casas disponiveis, tendo em conta uma operação especial do cavalo aplicando as mesmas regras definidas acima *(Regra do Duplo e Simêtrico)*.

```
(defun operador-inicial-cavalo (tabuleiro indice)
  "Operador para colocar o cavalo em Jogo que recebe um tabuleiro e uma coluna,
  valida a posicao do cavalo e impõe as regras"

//Regra do Simétrico
//Regra do Duplo
...
// Substituir
...
)

(defun nova-sucessao-cavalo (no valor-casa)
"função que realiza que cria um no-inicial   "
...
 //cria nó inicial
 ...
)

(defun sucessao-cavalo (no  lista-casas-disponiveis)
"Cria o conjunto de Sucessões iniciais no tabuleiro sem cavalo"
...
///nova-sucessao-cavalo
...
)
```

Representação de Estados

O Jogo do Cavalo (Knight Game) permite o tabuleiro obter varias possibilidades de jogadas e caminhos possiveis até encontrar uma solução, neste caso o problema será equacionado em termos de estados, em que são representados os estados ou uma representação sequencial de uma estrutura de dados, operadores que permite a transição dos estados desde do estado inicial até o final, podendo utilizar a exploração de arvores. para poder representar o estado do problema, utilizamos o tipo abstrato nó.

Operadores

Existem 8 operadores ao todo ou seja (8 movimentos) que o cavalo pode realizar. Os movimentos são realizados sobre os eixos de x y ou por linha e coluna.

(x, y) => (l , c) => moves (cima/baixo , direita/esquerda)

Os seguintes operadores são definidos com estes movimentos:

operador-1 (2 -1)

operador-2 (2 1)

operador-3 (1 2)

operador-4 (-1 2)

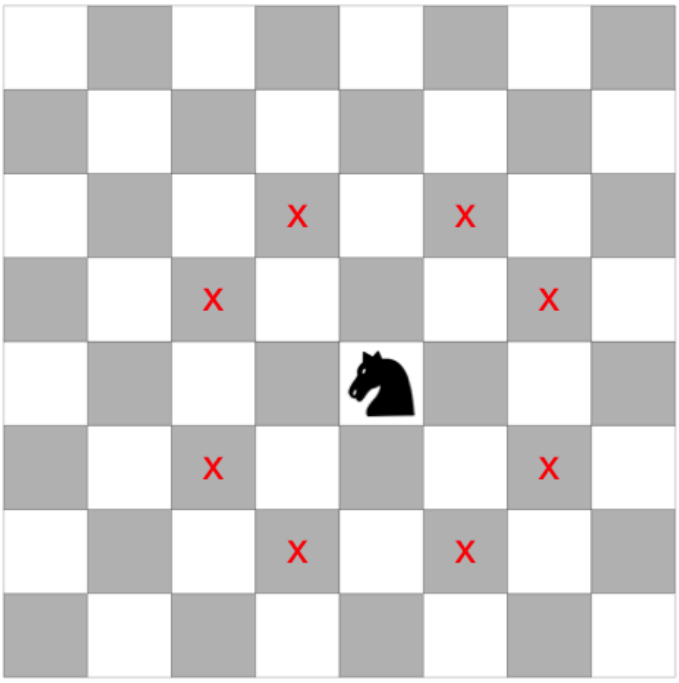
operador-5 (-2 1)

operador-6 (-2 -1)

operador-7 (-1 -2)

operador-8 (1 -2)

Na implementação dos operadores foram validados os seus movimentos bem como a aplicação de todas as Regra Referidas No Topico Acima.



Nó

Composição do Nó

A composição do nó é uma lista composta por:

(Tabuleiro | Posição | Pontos | Profundidade | Pai | Heuristica)

```
(defun cria-no (tabuleiro  posicao pontos profundidade pai &optional (h 0))
  "Cria um nó : (Tabuleiro|Posição|Pontos|Profundidade|Pai|Heuristica)"
  (list tabuleiro posicao pontos profundidade pai h)
  )
```

Seletores do Nó

Os seletores do Nó são seletores que retornam os atributos que o compõem.

Nomeadamente:

- *no-estado-tabuleiro*
- *no-Posicao*
- *no-pontos*
- *no-profundidade*
- *no-pai*
- *no-H*

- *no-custo*

Sucessões

A Sucessão de um determinado nó, é um conjunto de movimentos permitidos ao cavalo numa certa posição.

```
(defun novo-sucessor (no op &optional (heuristica 0))
  "Cria um novo sucessor a partir do no que recebe e a operação"
  //cria o nó a partir da operação recebida
)


(defun sucessores (no ops algoritmo f-heuristica &optional (prof nil))
//
)
```

Algoritmos e sua implementação

No âmbito deste projeto considera-se o problema do cavalo, cuja o objetivo principal consiste em atingir a pontuação definida para o problema, no menor numero possivel de jogadas, onde surge a necessidade de utilizarmos os algoritmos lecionadas nas aulas,para solucionar os problemas na deslocação do cavalo ao longo do tabuleiro, em jogadas possiveis até não ser possivel atingir os objetivos.

Solução

A solução é uma função de paragem aos algorimos implementados que tem duas condições, o cavalo já não tem mais movimentos disponiveis, ou o objetivo dos pontos foi cumprido.

Problema	Objetivo
A	70
B	60
C	270
D	600
E	300
F	2000

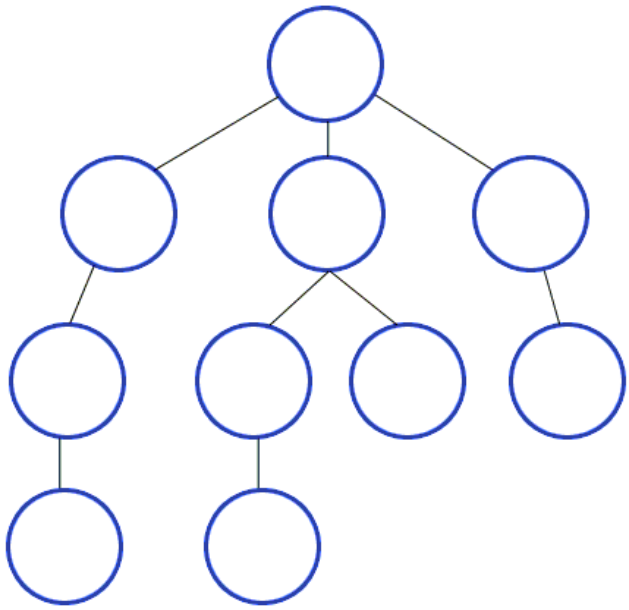
```
(defun no-solucaop (no alg)
 " Retorna T/NIL consoante se chegamos ao objetivo ou não houver mais movimentos"

)
```

Neste projecto, iremos detalhar os algoritmos implementdos abaixo.

BFS (Breadth First Search)

Este algorimo consiste na pesquisa por largura dos estados do nó da raiz até ao nó solução *(GOAL)*, explorando os nós vizinhos na profundiade atual do tabuleiro, garantindo que não volta abrir o nó pai que já foi aberto antes. Os nós explorados são os nós abertos são colocados atrás da fila dos nós sucessores.

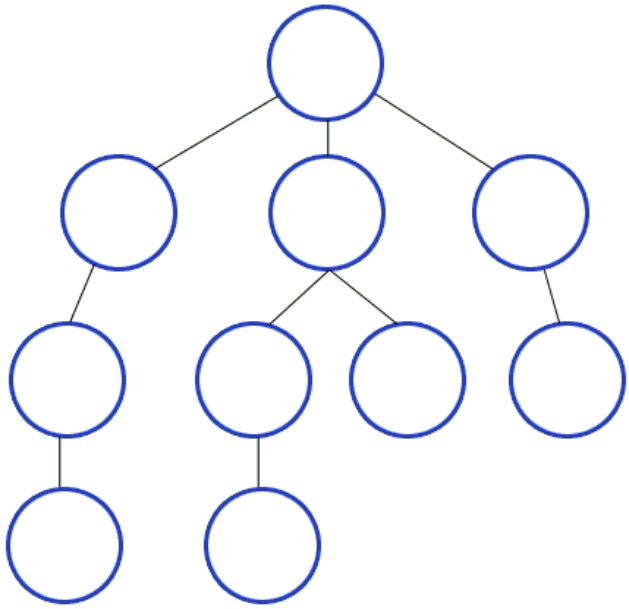
```
(defun abertos-bfs (abertos sucessores)
  "Devolve a lista de abertos do BFS"
  (append abertos sucessores)
  )
```



DFS (Depth-first search)

O algoritmo DFS permite procurar os nós em profundidade no tabuleiro assim contrário ao BFS que procura na largura, sendo assim Este explora o nó quando cavalo se desloca de uma casa para outra ao longo do tabuleiro. Estes nós explorados são os sucessores e colocados a frente da fila dos nós abertos.

```
(defun abertos-dfs (abertos sucessores)
"Devolve a lista de abertos do dfs"
  (append sucessores abertos)

)
```



A* (A* Search Algorithm)

O algoritmo A* tal como os anteriores servem para procurar os nós numa arvora, isto é
os algoritmos BFS e DFS consistem na procura dos estados num espaço de problema menos complexo, e o contrario do A* que permite a procura dos estados num espeço com grande problema complexo, gerando uma arvor de sucessores atraves da função heuristica que calcula o custo, sendo este custo é utilizado para ordenar as listas de nós das possiveis jogadas no tabuleiro para atingir o nó da possivel solução.

Ordenação

Algoritmo de ordenação usado para ordenar a lista de nós abertos no algoritmo A*:

1. ordenar-nos

É uma função robusta de pouca eficiencia/performance visto que procura todos os nós dentro da lista de Abertos.

Função

```
(defun ordenar-nos (lista)
  "Ordena o so consoante o custo para ser usado na lista de abertos ordenada no algoritmo A* "
  (sort (copy-seq lista) #'< :key #'no-custo)
)
```

2. Quicksort

O Quicksort adota a estratégia de divisão e conquista. A estratégia consiste em rearranjar as chaves de modo que as chaves "menores" precedam as chaves "maiores". Em seguida o quicksort ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada.

Função

```
(defun quicksort (no)
  "Algoritmo de ordenação usado para ordenar a lista de nós abertos no algoritmo A*"
  (cond
   ((null no) nil)
   (t
    (let* ((x (car no))
           (resto (cdr no))
           (fn (lambda (a) (< (no-h a) (no-h x)))))
      (append (quicksort (remove-if-not fn resto))
              (list x)
              (quicksort (remove-if fn resto)))
      )
    )
   )
  )
```

Heurística

A procurar por heurística permite realizar a pesquisa por meio da quantificação de proximidade de um determinado objetivo, neste projeto iremos utlizar heurística para encontrarmos as possiveis soluções consoante os objetivos de cada tabuleiro.

```
    (LET ((heuristica))
      (defun definir-heuristica(x) (setf heuristica-def x))
      (defun heuristica-usada()
        (cond
        ((equal heuristica-def  'base ) 'h   )
        ((equal heuristica-def  'implementada ) 'h   )
        (t  nil)
        )))
```

Heuristica Base

Tal como descrito no enunciado, utilizamos Heuristica de base que previlegia visitar as casas com maior numero pontos, para determinar o tabuleiro x.

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

m(x) é a média por casa dos pontos que constam no tabuleiro x no momento

o(x) é o número de pontos que faltam para atingir o valor definido como objetivo no momento

Heuristica Desenvolvida

A nossa Heuristica segundo o problema deverá refletir uma forma de os movimentos não querem optar por um Path com casas Duplas e que previlegia visitar as casas com maior numero pontos.

Ao longo do desenvolvimento do projeto estudamos várias teorias como (Hamiltonian Path) e também Warnsdorff mas sem sucesso. Apenas temos umas ideias a tirar.

Always move the knight to an adjacent, unvisited square with minimal degree

SMA* (Simplified Memory Bounded A* Search Algorithm)

Simplified Memory Bounded A* - SMA*:

- Se tiver memória suficiente para guardar toda a árvore tem a mesma funcionalidade de A*;
- Esquece o nó com *f(x)* mais elevado, em caso de empate, remove o de de mais baixo;
- Antes de remover uma sub-arvore, guarda no nó antecessor info sobre o melhor caminho dessa sub arvore;
- Um nó cuja profundidade for igual ao limite de memória de numero de nós, é imediatamente valorizado com f=inf.

IDA

A pesquisa de profundidade em iteração é aplicada a uma pesquisa de A*, onde este o algoritmo permite realizar a pesquisa repetida e profunda com a profundidade limitada, pois ao inves de limitar o numero de ligações entre os caminhos, logo limita-se o valor da função f(n) ou o custo do caminho. este limite começa com valor do nó inicial ou valor minimo. e de seguida realiza a pesquisa limitada por profundiade, mas nunca expande um nó com maior valor ou final. isto é: se a pesquisa limida de profundidade falhar de
maneira não natural, logo o proximo limite será o minímo dos valores que excedem o limite anteriormente pesquisados, entre tanto esta pesquisa não foi aplicada ao nosso projeto por falta do tempo.

Limitações técnicas e ideias para desenvolvimento futuro

Ao longo do desenvolvimento do projeto, tivemos algumas dificuldades em implementar os algoritimos de procura, tendo encontrado alguns problemas com lispworks ao ter a memoria demasiado pequena no espaço da procura, sempre que utiliza-se o algoritmo BFS. Além disso também tivemos alguma dificuldade na implementação dos operadores, visto que têm vária validações devido ás regras do jogo.

Resultados

Para poder comparar a eficácia dos 4 algoritmos funcionais foi feito uma tabela com as estatisticas de cada algoritmo na resolução de cada problema.

**ABF(Average Branching Factor)* - Fator Médio de Ramificação

BFS (Breadth First Search)

Problema	Nós Gerados	Nós Expandidos	*g(x)* Profundidade	Penetrância	Points	ABF

Problema	Nós Gerados	Nós Expandidos	*g(x)* Profundidade	Penetrância	Points	ABF
A	12	7	2	0.16666667	49	1.198952
B	30	13	1	0.033333335	21	1.4831691
C	12	2	0	0.0	12	1.198952
D	55	15	1	0.018181818	97	1.7105427
E	37	12	2	0.054054056	56	1.5400125
F	16938	2191	4	0.00023615539	282	4.893774

DFS (Depth-first search)

(Usando uma profundidade de 35)

Problema	Nós Gerados	Nós Expandidos	Profundidade *g(x)*	Penetrância	Points	ABF	Runtime
A	6	4	2	0.33333334	49	1.0284217	0.001
B	15	11	9	0.6	65	1.2557953	0.001
C	11	5	3	0.27272728	154	1.198952	0.001
D	16	4	1	0.125	186	1.3126388	0.003
E	27	14	12	0.44444445	186	1.4263256	0.015
F	124	37	35	0.28225806	1975	1.9947598	0.022

A* (A* Search Algorithm)

Problema	Nós Gerados	Nós Expandidos	Profundidade *g(x)*	Heuristíca *h(x)*	Penetrância	Points	ABF
A	8	6	2	*h(x)=o(x)/m(x)* 1.5681818	0.25	49	1.0852652
B	15	7	1	*h(x)=o(x)/m(x)* 8.181818	0.06666667	21	1.2557953
C	8	3	0	*h(x)=o(x)/m(x)* 0	0.0	12	1.0852652
D	34	14	3	*h(x)=o(x)/m(x)* 8.988086	0.0882353	197	1.5400125
E	28	16	12	*h(x)=o(x)/m(x)* 9.593023	0.42857143	226	1.4831691
F	150	45	34	*h(x)=o(x)/m(x)* 5.263158	0.22666668	1796	2.0516033

SMA* (Simplified Memory Bounded A* Search Algorithm)

(Usando como maximo de Memoria 4 Nós)

Problema	Nós Gerados	Nós Expandidos	Profundidade *g(x)*	Heuristíca *h(x)*	Penetrância	Points	ABF
A	8	6	2	1.5681818	0.25	49	1.0852652
B	15	7	1	*h(x)=o(x)/m(x)* 8.181818	0.06666667	21	1.2557953
C	8	3	0	*h(x)=o(x)/m(x)* 0	0.0	12	1.0852652
D	34	14	3	*h(x)=o(x)/m(x)* 8.988086	0.0882353	197	1.5400125

Problema	Nós Gerados	Nós Expandidos	Profundidade *g(x)*	Heuristíca *h(x)*	Penetrância	Points	ABF
E	28	16	12	*h(x)=o(x)/m(x)* 9.593023	0.42857143	226	1.4831691
F	150	43	32	9.51555	0.21333334	1630	2.0516033