1. Inteligência Artificial

JOGO DO BISPO - PARTE 1

Docente:

- Filipe Mariano

Alunos:

- Diogo Venâncio - 160221076
- André Gonçalves - 170221015

Indíce

Objetivo do projecto

O principal objetivo consiste em explorar o espaço de possibilidades tentando vários caminhos possíveis até encontrar a solução, recorrendo aos algoritmos de procura.

Neste sentido, o intuito do projeto passa por testar diversos tabuleiros com diversos algoritmos e analisar as soluções encontradas.

Divisão do projecto

1. **projeto.lisp**

> Carrega os outros ficheiros de código, escreve e lê ficheiros, e trata da interação com o jogador

2. **puzzle.lisp**

> Código relacionado com o problema.

3. **procura.lisp**

> Contem a implementação dos algoritmos

4. **problemas.dat**

> Contem todos os problemas do projecto

Algoritmos

1. Algoritmo de Procura BFS
2. Algoritmo de Procura DFS
3. Algoritmo de Procura A*

Projecto

Interecção com o jogador

> Função ao qual o jogador executa para iniciar o programa

```
;Mostra o menu inicial
(defun inicial-menu()
  (progn
    (format t "~% -------------------------------------")
    (format t "~%              Jogo do Bispo            ")
    (format t "~%                                       ")
    (format t "~%              1 - Começar              ")
    (format t "~%                                       ")
```

```
    (format t "~%              s - Sair                  ")
    (format t "~% ------------------------------------~%~%> "))
)

;Iniciar programa
(defun start()
  (progn
    (inicial-menu)
      (let ((opt (read)))
        (if (or (not (numberp opt)) (< opt 1) (> opt 1)) (progn (format t "Insira
uma opção válida") (start))
          (ecase opt
            ('1 (ask-algorithm))
            ('s (format t "Até á próxima!")))))
)
```

> Mensagem para o jogador selecionar o algoritmo

```
;Apresenta a mensagem para escolher o algoritmo
(defun algorithm-menu()
  (progn
    (format t "~%----------------------------------------")
    (format t "~%   Jogo do Bispo - Escolha o algoritmo    ")
    (format t "~%                                          ")
    (format t "~%           1 - Breadth First              ")
    (format t "~%           2 - Depth First                ")
    (format t "~%           3 - A*                         ")
    (format t "~%           0 - Voltar                     ")
    (format t "~%                                          ")
    (format t "~%           s - Sair                       ")
    (format t "~%                                          ")
    (format t "~% ----------------------------------------~%~%>")
  )
)
```

Puzzle

Código relacionado com o problema

> Função responsável por retornar todas as posições que o bispo pode ir numa certa diagonal

Esta função recebe um board e verifica a posição atual do bispo.

Verifica se o bispo já foi colocado no board, senão coloca o bispo numa casa random com valor na primeira linha.

Verifica se as posições estão dentro dos limites do board e se os valores não contêm 0 nas posicoes..

Se tudo for válido, adiciona a posição á lista e percorre o resto do board.

```
(defun diagonal-1-pos(board &optional (bispo-pos (posicao-bispo board)))
  (cond
   ((null bispo-pos) (diagonal-1-pos (coloca-bispo board)))
   ((or (< (car bispo-pos) 0) (< (cadr bispo-pos) 0) (> (car bispo-pos) 8) (>
(cadr bispo-pos) 8) (equal (car bispo-pos) 0) (equal (cadr bispo-pos) 0)) nil)
   (t (cons bispo-pos (diagonal-1-pos (cdr board) (list (- (car bispo-pos) 1) (+
(cadr bispo-pos) 1))))))
)
```

> Função que coloca o bispo na nova casa jogada

- Esta função recebe uma posicao jogavel pelo bispo e o board.
- Verifica se a posicao e o board têm valores.
- Guarda a posição pretendida e a posição atual do bispo.
- Se a posição do bispo não for possivel retornar, coloca o bispo no board.
- Verifica se a posição está vazia.
- De seguida, guarda o valor da casa para o qual vai corresponder á posição no board.
- Guarda o board com a posição atual do bispo como NIL.
- Guarda o board com o bispo na nova posição.
- Guarda a posição do valor simétrico da nova posição.
- Guarda o duplo maximo que existe o ultimo board.
- Se não existir o duplo desse valor nem simétrico, simplesmente retorna o board atualizado.
- Se existir duplo, este altera a casa do maior duplo para NIL.
- Se existir simétrico, altera a casa do valor simetrico para NIL.

```
(defun jogar-bispo(lista board)
  (cond
   ((and (null lista) (null board)) nil)
   (t (let* ((posicao lista)
         (bispo-pos (posicao-bispo board)))
     (cond
      ((null bispo-pos) (coloca-bispo board))
      ((null posicao) nil)
      (t (let ((valor (get-celula (car posicao) (cadr posicao) board)))
        (cond
         ((null valor) nil)
```

```
          (t (let* ((board-t (substituir (car bispo-pos) (cadr bispo-pos) board
'NIL))
                (board-t-1 (substituir (car posicao) (cadr posicao) board-t 'T))
                (simetrico (simetrico-pos valor board-t-1))
                (duplo (duplo-maximo board-t-1)))
            (cond
             ((and (null simetrico) (null duplo) board-t-1))
             ((null simetrico) (substituir (get-valor-pos duplo board-t-1) board-t-
1 'NIL))
             (t (substituir (car simetrico) (cadr simetrico) board-t-1
'NIL))))))))))))
)
```

Procura

Implementação dos algoritmos

> Algoritmo BFS

1. Nó inicial -> Abertos
2. Se Abertos vazio, falha
3. Remove o primeiro no de Abertos (n) e coloca-o em fechados
4. Expande o no n. Coloca os sucessores no fim de Abertos, colocando os ponteiros para n.
5. Se algum dos sucessores é um no objetivo sai, e dá a solução. Caso contrário vai para chamar novamente a função

```
(defun bfs(sucessor abertos &optional moves solucao fechados)
  (let* ((atual (car abertos)) ;No atual
         (lista (posicoes-jogadas-possiveis (jogadas-possiveis (car atual)) (car
atual))) ;Lista das jogadas possiveis
         (sucessores (funcall sucessor atual 'bfs lista)) ;Sucessores do no atual
         (abertos-new (lista-abertos-bfs (cdr abertos) sucessores)) ;Adiciona á
lista de nos abertos
         (fechados-new (cons atual fechados))) ;Adiciona o no atual á lista de
fechados
    (cond
     ((null abertos) nil)
     ((equal moves (profun-no atual)) (list atual (length abertos-new) (length
fechados-new))) ;Se os movimentos forem iguais á profundidade do no atual
     ((null abertos-new) (list atual (length abertos-new) (length fechados-new)))
;Se a atual lista de abertos estiver vazia
     ((and (not (null solucao)) (<= solucao (g-no atual))) (list atual (length
abertos-new) (length fechados-new))) ;Se a solucao não for null e for menor ou
igual ao valor g do no atual
     (t (bfs sucessor abertos-new moves solucao fechados-new))))
)
```

Algoritmo DFS

1. Nó inicial -> Abertos
2. Se Abertos vazio, falha
3. Remove o primeiro no de Abertos (n) e coloca-o em fechados
4. Se a profundidade de n é maior que d vai chamar novamente a função
5. Expande o no n. Coloca os sucessores no inicio de Abertos, colocando os ponteiros para n.
6. Se algum dos sucessores é um no objetivo sai, e dá a solução. Caso contrário vai para chamar novamente a função

```
(defun dfs(sucessor abertos &optional moves solucao fechados)
  (let* ((atual (car abertos)) ;No atual
         (lista (posicoes-jogadas-possiveis (jogadas-possiveis (car atual)) (car
atual))) ;Lista das jogadas possiveis
         (sucessores (funcall sucessor atual 'dfs lista)) ;Sucessores do no atual
         (abertos-new (lista-abertos-dfs sucessores (cdr abertos))) ;Adiciona á
lista de nos abertos
         (fechados-new (cons atual fechados))) ;Adiciona o no atual á lista de
fechados
    (cond
     ((null abertos) nil)

     ;Se os movimentos forem menores que a profundidade do no atual (Se a
profundidade de atual é maior que o movimentos atribuidos vai para 2)
     ((> (profun-no atual) moves) (dfs sucessor abertos-new moves solucao fechados-
new))
     ((null abertos-new) (list atual (length abertos-new) (length fechados-new)))
;Se a atual lista de abertos estiver vazia
     ((and (not (null solucao)) (<= solucao (g-no atual))) (list atual (length
abertos-new) (length fechados-new))) ;Se a solucao não for null e for menor ou
igual ao valor g do no atual
     (t (dfs sucessor abertos-new moves solucao fechados-new))))
)
```

Algoritmo A*

1. Nó inicial -> Abertos.
2. Se Abertos vazio, falha
3. Remove o nó de Abertos (n) com menor custo de f e coloca-o em Fechados
4. Expande o nó n. Calcula o f de cada um dos sucessores
5. Coloca os sucessores que não existem em Abertos nem Fechados na lista de Abertos, por ordem de f colocando os ponteiros para n
6. Se algum sucessor for um nó objetivo termina e dá a solução
7. Vai chamar novamente a função

```
(defun a*(sucessor heuristica abertos solucao &optional (moves 10) fechados)
  (let* ((atual (f-best abertos solucao)) ;No atual com melhor valor de f segundo
```

```
a solucao pretendida
        (lista (posicoes-jogadas-possiveis (jogadas-possiveis (car atual)) (car
atual))) ;Lista das jogadas possiveis
        (sucessores (funcall sucessor atual 'a* lista moves solucao heuristica))
;Sucessores do no atual
        (abertos-new (append (cdr abertos) sucessores)) ;Adiciona á lista de nos
abertos
        (fechados-new (cons atual fechados)) ;Adiciona o no atual á lista de
fechados
        (best-fechado (f-best fechados solucao))) ;Verifica qual é o no fechado
com melhor valor de f
    (cond
     ((null abertos) nil) ;Se a lista de abertos estiver vazia
     ((null abertos-new) (list atual (length abertos-new) (length fechados-new)))
;Se a nova lista de abertos estiver vazia
     ((equal moves (profun-no atual)) (list atual (length abertos-new) (length
fechados-new))) ;Se o nº de movimentos for igual á profundidade do no atual
     ((<= solucao (g-no atual)) (list atual (length abertos-new) (length fechados-
new))) ;Se a solucao atribuida for menor ou igual ao valor g do no atual
     ((and (not (null best-fechado)) (> (f-calcula best-fechado) (f-calcula
atual))) ;Se o melhor no fechado nao estiver vazio, se o melhor no fechado com o
valor f for maior que o valor f do atual no, chama novamente o algoritmo a* mas na
nova lista de abertos, é adicionado o melhor no fechado com valor f e em fechados
o mesmo é removido
      (a* sucessor heuristica (cons best-fechado abertos-new) solucao moves
(remover-se #'(lambda (x) (if (equal x best-fechado) x)) fechados-new)))
     (t (a* sucessor heuristica abertos-new solucao moves fechados-new))))
)
```

Análise Comparativa

Como forma de compararmos os valores de output de cada algoritmo, usámos o board D com movimentos máximo igual a 10 e pontos objetivo 400. O A* fizemos para ambas as heuristicas.

> Algoritmo BFS

```
(NIL NIL NIL NIL NIL NIL 78 NIL)
(NIL NIL NIL NIL NIL NIL NIL NIL)
(NIL NIL 56 NIL NIL NIL NIL NIL)
(NIL NIL NIL NIL NIL NIL T NIL)
(34 NIL NIL NIL NIL NIL NIL NIL)
(NIL 54 NIL NIL NIL 17 NIL NIL)
(45 NIL NIL NIL NIL 31 NIL NIL)
(NIL NIL NIL 71 NIL NIL 43 NIL)

Inicio: 18:18:20
Fim: 18:18:20
Nos gerados: 32
```

```
Nos expandidos: 32
Profundidade max: 10
Objetivo: 400
Penetrância: 7/32
Comprimento: 7
Pontos totais: 200
```

Algoritmo DFS

```
(NIL NIL NIL NIL NIL NIL 78 NIL)
(NIL NIL T NIL NIL NIL NIL NIL)
(NIL NIL 56 NIL NIL NIL NIL NIL)
(NIL NIL NIL NIL NIL NIL NIL NIL)
(34 NIL NIL NIL NIL NIL NIL NIL)
(NIL 54 11 NIL NIL 17 NIL NIL)
(45 NIL NIL 42 NIL NIL NIL NIL)
(NIL NIL NIL 71 NIL NIL NIL NIL)

Inicio: 18:20:48
Fim: 18:20:48
Nos gerados: 72
Nos expandidos: 72
Profundidade max: 10
Objetivo: 400
Penetrância: 1/12
Comprimento: 6
Pontos totais: 252
```

Algoritmo A* - Heuristica Base

```
(NIL NIL NIL NIL NIL NIL NIL NIL)
(NIL NIL 23 NIL NIL NIL NIL NIL)
(NIL NIL 56 NIL NIL NIL NIL NIL)
(NIL 32 NIL NIL NIL NIL NIL NIL)
(NIL NIL NIL NIL NIL NIL NIL NIL)
(NIL NIL 11 NIL NIL NIL NIL NIL)
(NIL NIL NIL 42 NIL NIL NIL NIL)
(NIL NIL NIL NIL NIL NIL T NIL)

Inicio: 18:21:23
Fim: 18:21:23
Nos gerados: 18
Nos expandidos: 9
Profundidade max: 10
Objetivo: 400
Penetrância: 4/9
Comprimento: 8
```

```
Pontos totais: 432
Heuristica: HEURISTICA
```

> Algoritmo A* - Heuristica Criada

```
(NIL NIL NIL NIL NIL NIL 78 NIL)
(NIL NIL 23 NIL NIL NIL NIL NIL)
(NIL NIL 56 NIL NIL NIL NIL NIL)
(NIL 32 NIL NIL NIL NIL T NIL)
(34 NIL NIL NIL NIL NIL NIL NIL)
(NIL 54 11 NIL NIL 17 NIL NIL)
(45 NIL NIL 42 NIL 31 NIL NIL)
(NIL NIL NIL 71 NIL NIL 43 NIL)

Inicio: 18:22:19
Fim: 18:22:19
Nos gerados: 9
Nos expandidos: 9
Profundidade max: 10
Objetivo: 400
Penetrância: 4/9
Comprimento: 4
Pontos totais: 168
Heuristica: HEURISTICA-CRIADA
```

Requisitos não implementados

1. Algoritmo de limitação de memória (SMA*, IDA* RBFS)