Manual Técnico

UC: Inteligência Artificial

Alunos:

- João Fernandes - 202100718
- Rodrigo Santos - 202100722

Docente: Joaquim Filipe

Índice

1. Introdução

No âmbito do primeiro projeto da Unidade Curricular (UC) de Inteligência Artificial (IA), do 1º semestre do 3º ano da Licenciatura em Engenharia Informática, foi solicitado aos alunos que implementassem um programa para resolução do problema Adji-boto, recorrendo a algoritmos de procura em espaços de estados e usando a linguagem de programação LISP.

O presente manual técnico tem como objetivo descrever a solução desenvolvida pelos alunos, bem como os detalhes da sua implementação.

2. Módulos

O sistema é composto pelos módulos apresentados na tabela abaixo.

Módulo	Descrição
Puzzle	Funções auxiliares à resolução do problema em questão
Procura	Algoritmos de procura em espaço de estados (BFS, DFS e A*)
Desempenho	Medidas de desempenho dos algoritmos
Projeto	Interação com o utilizador, leitura e escrita em ficheiros e carregamento dos restantes módulos

3. Entidades

Existem algumas entidades inerentes aos conceitos de procura em espaços de estados. A tabela abaixo descreve as entidades usadas no presente projeto.

Entidade	Descrição	Implementação
Estado	Estado atual do problema	Tabuleiro adji-boto, representado através de lista de listas
Nó	Estado com informação adicional	Lista contendo o estado, nível, pai e custo
Operador	Função que altera o estado de um nó	Função de distribuição de peças de uma posição para as seguintes
Sucessor	Novo estado quando aplicado um operador a um nó	Novo nó com aplicação do operador ao nó pai.
Abertos	Lista com os nós que ainda não foram expandidos	Lista
Fechados	lista com os nós que já foram expandidos	Lista

4. Algoritmos

De forma a resolver os problemas, os alunos implementaram os algoritmos Breadth-First Search, Depth-First Search e A*, usando uma abordagem funcional, recursiva e não-destrutiva.

4.1. BFS

O algoritmo BFS realiza uma procura em largura. Na solução proposta pelos alunos, a função recebe como argumento o estado inicial do problema, a lista de abertos (inicializada com o nó do estado inicial) e a lista de fechados (inicialmente vazia).

O seu funcionamento processa-se da seguinte forma:

1. Se a lista de abertos estiver vazia, retorna NIL, indicando que não existe solução;
2. Seleciona o primeiro nó da lista de abertos e gera os sucessores válidos deste nó, excluindo aqueles que já estão nas listas de abertos ou fechados;
3. Se algum dos sucessores for um nó objetivo, retorna o caminho até esse nó;
4. Se nenhum nó objetivo for encontrado, efetua a chamada recursiva com os sucessores válidos adicionadas ao final de abertos e o nó atual adicionado a fechados.

Abaixo segue a fonte do algoritmo geral:

```
(defun bfs (estadoInicial &optional (abertos (list (cria-no estadoInicial)))
(fechados '()))
  (cond
    ((null abertos) nil)  ; Se a lista de abertos estiver vazia, retorna nil
    (t
     (let ((sucessores-validos
            (remove-if-not
             (lambda (sucessor)
```

```
                (and (not (lista-tem-no sucessor abertos))
                     (not (lista-tem-no sucessor fechados))))
              (sucessores (first abertos) (gerar-operadores (estado (first
abertos)))))))
       (cond
         ((not (null (filtrar-nos-objetivos sucessores-validos)))
          (apresentar-resultado (filtrar-nos-objetivos sucessores-validos)))
         (t
          (bfs estadoInicial
               (append (rest abertos) sucessores-validos)
               (cons (first abertos) fechados))))))))
```

4.2. DFS

O algoritmo DFS realiza uma procura em profundidade. Na solução proposta pelos alunos, a função recebe como argumento o estado inicial do problema, um limite de profundidade (opcional), a lista de abertos (inicializada com o nó do estado inicial) e a lista de fechados (inicialmente vazia).

O seu funcionamento processa-se da seguinte forma:

1. Se a lista de abertos estiver vazia, retorna NIL, indicando que não há solução;
2. Seleciona o primeiro nó da lista de abertos e verifica a profundidade do nó atual;
3. Se a profundidade do nó atual for maior ou igual ao limite, ignora este nó e continua com o restante da lista de abertos;
4. Gera os sucessores válidos do nó atual, excluindo aqueles que já se encontram em abertos ou fechados;
5. Se algum dos sucessores for um nó objetivo, retorna o caminho até esse nó;
6. Se nenhum nó objetivo for encontrado, efetua a chamada recursiva com os sucessores válidos adicionados ao início de abertos e o nó atual adicionado a fechados.

Abaixo segue a fonte do algoritmo geral:

```
(defun dfs (estadoInicial &optional (limite most-positive-fixnum) (abertos (list
(cria-no estadoInicial 0))) (fechados '()))
  (if (null abertos)
      nil  ; Retorna nil se não encontrar solução
      (if (>= (nivel (first abertos)) limite)  ; Se a profundidade exceder o
limite, não continua
          (dfs nil limite (rest abertos) (cons (first abertos) fechados))
          (let ((sucessores-validos
                 (remove-if-not
                  (lambda (sucessor)
                    (and (not (lista-tem-no sucessor abertos))
                         (not (lista-tem-no sucessor fechados))))
                  (sucessores (first abertos) (gerar-operadores (estado (first
abertos)))))))
            (if (not (null (filtrar-nos-objetivos sucessores-validos)))
                (apresentar-resultado (filtrar-nos-objetivos sucessores-validos))
; Se houver objetivo, retorna o caminho
                (dfs estadoInicial limite
```

```
                    (append sucessores-validos (rest abertos))
                    (cons (first abertos) fechados)))))))
```

4.3. A*

O algoritmo A* é um algoritmo heurístico, isto é, é fornecido conhecimento do domínio de aplicação para determinar se os nós gerados têm "interesse". Na solução proposta pelos alunos, a função recebe como argumento o estado inicial do problema, uma função heurística, a lista de abertos (inicializada com o nó do estado inicial) e a lista de fechados (inicialmente vazia).

O seu funcionamento processa-se da seguinte forma:

1. Se a lista de abertos estiver vazia, retorna NIL, indicando que não há solução;
2. Seleciona o primeiro nó da lista de abertoss;
3. Se o estado do nó atual for objetivo, retorna o caminho até a esse nó;
4. Caso o nó não seja objetivo, gera os sucessores do nó atual, aplicando a função heurística;
5. Remove os nós com maior custo das listas de abertos e fechados em comparação com os sucessores gerados;
6. Remove os sucessores que já estão em abertos ou fechados;
7. Insere os sucessores válidos na lista de abertos filtrada;
8. Realiza a chamada recursvia com as novas listas de abertos e fechados.

As heurísticas implementadas são as seguintes:

- Base: Diferença entre o número total de peças e as peças já capturadas num tabuleiro;
- Personalizada: Diferença entre o número de peças e a média de peças eliminadas nas próximas jogadas que eliminam peças.

Ambas as heurísticas não são admissíveis pois o valor dado pela heurística é diferente que a distância do nó atual até ao nó objetivo.

Abaixo segue a fonte do algoritmo geral:

```
(defun a-star (estadoInicial fHeuristica &optional (abertos (list (cria-no
estadoInicial))) (fechados '()))
  "Executa o algoritmo A* usando recursão em cauda otimizada."
  (if (null abertos)
      nil ; Caso a lista de abertos esteja vazia, falha.
      (let ((no-atual (first abertos)))
        (if (tabuleiro-vaziop (estado no-atual))
            (apresentar-resultado no-atual) ; Se o tabuleiro estiver vazio,
retorna o caminho.
            (let* ((sucessores-validos
                    (filtra-sucessores
                     (sucessores no-atual (gerar-operadores (estado no-atual))
fHeuristica)
                     (rest abertos)
                     fechados)))
              (a-star
               ;; Estado
```

```
                nil
                ;; Heuristica
                fHeuristica
                ;; Abertos
                (inserir-nodes sucessores-validos (rest abertos))
                ;; Fechados
                (atualiza-fechados no-atual fechados sucessores-validos)))))))
```

5. Descrição das opções tomadas

Todas as decisões tiveram por base os conceitos aprendidos durante a UC de IA, nomeadamente a utilização do paradigma funcional, a abordagem recursiva e a não utilização de funções destrutivas.

O paradigma funcional permite que a solução tenha separação de interesses e que o código seja de mais fácil manutenção.

A abordagem recursiva permite desenvolver soluções mais concisas para problemas complexos, melhorando a legibilidade.

Ao usar o paradigma funcional, é favorecida a imutabilidade dos dados.

6. Comparação dos Algoritmos

Após a implementação dos algoritmos, os alunos procederam à análise das medidas de desempenho, nomeadamente o número de nós gerados, o comprimento do caminho, a penetrância, o fator de ramificação média e o tempo de execução.

Em traços gerais, o algoritmo A* possui um melhor desempenho em todos os critérios analisados.

O algoritmo BFS possui o pior desempenho e não encontra a maioria das soluções para os problemas, não só pelo seu funcionamento geral como também pelas limitações técnicas que serão explanadas mais adiante no presente manual.

Já o algoritmo DFS tem também um bom desempenho, com a salvaguarda que se deve fornecer o limite de profundidade muito próximo ao do nó objetivo ou, não fornecer limite de profundidade de forma a deixar o algoritmo explorar os estados tanto quanto possível.

A penetrância é apresentada como 0 em alguns casos por ser um valor muito baixo, ficando nesta forma depois do arredondamento.

6.1. Execução BFS

Problema	Nº Nós Gerados	Comprimento Caminho	Penetrância	Fator Ramificação Média	Tempo Execução
A	25	4	0,16	1,90	0,003s
B	Não encontrado	-	-	-	-
C	3756	6	0,00	3,76	3,1490s (*)

Problema	Nº Nós Gerados	Comprimento Caminho	Penetrância	Fator Ramificação Média	Tempo Execução
D	Não encontrado	-	-	-	-
E	Não encontrado	-	-	-	-
F	Não encontrado	-	-	-	-
G	Não encontrado	-	-	-	-

(*) Neste resultado foi necessário aumentar o tamanho da stack e está a ser contabilizado o tempo gasto na execução desse comando.

6.2. Execução DFS

Problema	Nº Nós Gerados	Comprimento Caminho	Penetrância	Fator Ramificação Média	Tempo Execução
A	11	6	0,55	1,22	0,003s
B	99	18	0,18	1,22	0,015s
C	39	10	0,26	1,22	0,003s
D	362	53	0,15	1,03	0,014s
E	809	108	0,13	1,03	0,052s
F	631	97	0,15	1,03	0,05s
G	741	101	0.14	1,03	0.04s

6.3. Execução A* - Heurística Base

Problema	Nº Nós Gerados	Comprimento Caminho	Penetrância	Fator Ramificação Média	Tempo Execução
A	22	6	0,27	1,42	0,005s
B	156	14	0,09	1,32	0,009s
C	78	10	0,13	1,32	0,004s
D	554	39	0,07	1,12	0,04s
E	534	35	0,07	1,12	0,045s
F	540	39	0,07	1,12	0,049s
G	792	52	0,07	1,12	0,065s

6.4. Execução A*- Heurística Personalizada

Esta heurística encontra resultados com caminhos mais curtos, no entanto utiliza mais memória.

Problema	Nº Nós Gerados	Comprimento Caminho	Penetrância	Fator Ramificação Média	Tempo Execução
A	16	4	0,25	1,61	0,003s
B	248	11	0,04	1,51	0,037s
C	48	6	0,12	1,61	0,007s
D	294	18	0.06	1,22	0,064s
E	610	29	0,05	1,12	0,148s
F	644	31	0,05	1,12	0,009s
G	10940	35	0,00	1,22	17,326s (*)

(*) Neste resultado foi necessário aumentar o tamanho da stack e está a ser contabilizado o tempo gasto na execução desse comando.

7. Limitações Técnicas

Existem algumas limitações técnias no funcionamento do presente projeto, relacionadas com as limitações de memória do IDE LispWorks.

Desta forma, não é possível encontrar solução para os problemas B, C, D, E, F e G descritos no enunciado usando o algoritmo BFS (não aumentando o tamanho da memória stack). Isto deve-se ao facto de os problemas, na sua representação em grafo, possuirem o nó objetivo numa profundidade elevada, fazendo com que o método de procura em largura não tenha memória disponível no IDE suficiente para chegar à solução.

8. Requisitos Não Cumpridos

Após o desenvolvimento do presente projeto, elencam-se os seguintes requisitos não cumpridos:

- Implementação dos algoritmos mais eficientes em memória, como SMA*, IDA* e RBFS.

9. Conclusão

Ao longo deste projeto, foi possível aplicar na prática os conhecimentos teóricos adquiridos na UC de IA, no que diz respeito aos algoritmos de procura em espaços de estados.

A implementação da solução para o problema Adji-boto em LISP proporcionou uma valiosa experiência de programação e um aprofundamento da compreensão dos conceitos fundamentais de IA. O desenvolvimento deste programa não só permitiu consolidar os conhecimentos sobre os algoritmos de procura, mas também possibilitou a melhoria das competências de resolução de problemas e de pensamento lógico.

Além disso, este projeto evidenciou a importância da eficiência computacional na implementação de soluções para problemas complexos. Os alunos foram confrontados com a necessidade de otimizar os seus algoritmos

para lidar com a explosão combinatória dos estados do problema Adji-boto, o que resultou numa compreensão mais profunda das implicações práticas das escolhas algorítmicas.

Em suma, os alunos implementaram com sucesso uma solução eficaz para o problema proposto, aplicando os algoritmos em procura de espaços de estados lecionados na UC de IA.