Manual Técnico do Projeto Nº1: Época de Recurso

Inteligência Artificial - Escola Superior de Tecnologia de Setúbal 2024/2025
Estudante: **Rodrigo Baptista**, número **202200217**

1. Introdução

Este Manual Técnico tem como objetivo documentar o desenvolvimento e implementação do projeto Adji-boto, uma variante do jogo Mancala, no âmbito da disciplina de Inteligência Artificial da Escola Superior de Tecnologia de Setúbal. O projeto foca-se na aplicação de algoritmos de procura para encontrar soluções eficientes dentro do espaço de estados do jogo.
O principal desafio consiste em testar diferentes estratégias de procura, nomeadamente procura em largura (BFS), procura em profundidade (DFS), A* e SMA*, analisando o desempenho de cada método.
Este documento descreve a estrutura do código, os algoritmos implementados, os operadores do jogo, as estratégias de avaliação e as decisões técnicas adotadas. Também apresenta uma análise crítica dos resultados, incluindo estatísticas sobre o desempenho dos algoritmos e possíveis limitações dos mesmos.

2. Estrutura do Código

A estrutura de ficheiros do código é a seguinte:
• **projeto.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.
• **procura.lisp**: Contém a implementação de: Procura em Largura (BFS), Procura em Profundidade (DFS), A* e SMA*, independente do domínio do problema.
• **problemas.dat**: Contém a lista de problemas a resolver, separados por linhas, em que cada problema é uma lista de duas listas com seis átomos cada.
• **log.dat**: Ficheiro que será produzido depois de ser executada uma procura. Armazena os resultados de uma procura e resultados de procuras posteriores serão anexados ao conteúdo já presente no ficheiro.

3. Algoritmo Implementado e Funções Auxiliares

Nesta secção estão os algoritmos implementados, uma breve explicação sobre os mesmos e as funções auxiliares utilizadas.

3.1 Algoritmo BFS

1. Colocamos o Nó inicial em abertos.
2. Se OPEN for uma lista vazia falha.
3. Remove o primeiro nó de OPEN e coloca-o em CLOSED.
4. Expande o nó removido. Coloca os sucessores no fim de OPEN (Procura em Largura), colocando os ponteiros para o nó removido.
5. Se algum dos sucessores é um nó objectivo sai, e dá a solução. Caso contrário vai para 2.

Código:

```
(defun bfs (initial-node objective generator operator)
  "Breadth-First-Search Algorithm. 11 - 3. Receives an INITIAL-NODE, the OBJECTIVE
state, the GENERATOR function and the game OPERATOR."
  (let
    (
      (open (list initial-node))
      (closed '()))
    (loop while open do
      (let*
        (
          (first-node (pop open)) ; Take first node and remove from open.
          (children (remove-existing (funcall generator first-node operator 'bfs)
closed 'bfs)) ; Generate children.
          (solution-node (check-solution children objective)) ; Check for a solution.
        )
        (when solution-node (return solution-node))
        (push first-node closed)
        (setf open (append open children)) ; BFS-OPEN.
      ))))
```

3.2 Algoritmo DFS

1. Colocamos o Nó inicial em abertos.
2. Se OPEN for uma lista vazia falha.
3. Remove o primeiro nó de OPEN e coloca-o em CLOSED.
4. Se o Nó excedeu a profundidade de procura vai para 2.
5. Expande o nó removido. Coloca os sucessores no início de OPEN (Procura em Profundidade), colocando os ponteiros para o nó removido.
6. Se algum dos sucessores é um nó objectivo sai, e dá a solução. Caso contrário vai para 2.

Código:

```
(defun dfs (initial-node objective generator operator max-depth)
  "Depth-First-Search Algorithm. 17 - 3. Receives an INITIAL-NODE, the OBJECTIVE
state, the GENERATOR function, the game OPERATOR and the MAX-DEPTH allowed for the
search."
  (let
    (
      (open (list initial-node))
      (closed '()))
    (loop while open do
      (let*
        (
          (first-node (pop open)) ; Take first node and remove from open.
          (solution-node (check-solution children objective)) ; Check for a solution.
          (children (remove-existing (funcall generator first-node operator 'dfs max-
depth) closed 'dfs)) ; Generate children.
        )
        (when solution-node (return solution-node))
        (push first-node closed)
        (setf open (append children open)) ; DFS-OPEN.
      ))))
```

3.3 Algoritmo A*

1. Colocamos o Nó inicial em abertos, cujo custo será 0.
2. Se OPEN for uma lista vazia falha.
3. Remove o nó de OPEN com menor custo e coloca-o em CLOSED.
4. Expande o nó removido e calcula o custo dos seus sucessores.
5. Coloca os sucessores que ainda não existem em OPEN nem CLOSED na lista de OPEN, por ordem do custo e colocando os ponteiros para o nó removido.
6. Caso o nó removido seja o nó objetivo sai, e dá a solução.
7. Associa aos sucessores já em OPEN ou CLOSED o menor custos, caso seja necessário. Apenas são colocados em OPEN os sucessores cujo custo é menor que um já existente. Se já existir um nó igual, mas com menor custo, então esquecemos o nó gerado.
8. Vai para 2.

Código:

```
(defun a-star (initial-node objective generator operator heuristic)
  "A* Algorithm. 25 - 3. Receives an INITIAL-NODE, the OBJECTIVE state, the GENERATOR
function, the game OPERATOR and an HEURISTIC function."
  (let
    (
      (open (list initial-node))
      (closed '()))
    (loop while open do
      (let*
        (
          (first-node (pop open)) ; Take first node and remove from open.
          (children (remove-existing (funcall generator first-node operator 'a-star 0
heuristic) closed 'a-star)) ; Generate children.
          (solution-node (check-solution (list first-node) objective)) ; Check for a
solution.
        )
        (when solution-node (return solution-node))
        (push first-node closed)
        (setf open (order-nodes (append open children))) ; A* PUT-SUCCESSORS-IN-OPEN.
      ))))
```

3.4 Algoritmo SMA*

O algoritmo SMA* segue o mesmo funcionamento que o A*, mas, com uma ligeira diferença:

1. Após gerarmos os sucessores, temos de ver se chegámos ao limite de memória.
2. Se sim, removemos o último nó dentro de OPEN, e definimos o custo do pai para o valor da pior heuristica.
3. Se não, fazemos o mesmo processo que o A* e voltamos para 1.

Código:

```
(defun sma-star (initial-node objective generator operator heuristic memory-limit)
  "Simplified Memory-Bounded A* (SMA*). Receives an INITIAL-NODE, the OBJECTIVE state,
the GENERATOR function, the game OPERATOR, an HEURISTIC function and a MEMORY-LIMIT."
  (let
    (
```

```
      (open (list initial-node))
      (closed '())
    )
    (loop while open do
      (let*
        (
          (first-node (pop open))
          (solution-node (check-solution (list first-node) objective))
          (children (remove-existing (funcall generator first-node operator 'sma-star
0 heuristic) closed 'sma-star))
        )
        (when solution-node (return solution-node))
        (push first-node closed)
        (setf open (order-nodes (append open children)))
        (when (> (length open) memory-limit)
          (let
            ((worst-node (car (last open))))
            (setf open (butlast open)) ; "Pop" the worst node.
            (let
              ((parent (node-previous worst-node)))
              (when parent (setf (nth 2 parent) (max (nth 2 parent) (nth 2 worst-
node)))) ; Set the heuristic of the parent node to the worst heuristic, (max).
            )))))))
```

3.5 Funções de Geração

Funções para gerar os sucessores de um dado nó.

```
(defun new-child (node operator algorithm index1 index2 &optional heuristic)
  "Creates a new child node using OPERATOR at INDEX1 (line) and INDEX2 (position),
considering ALGORITHM & HEURISTIC."
  (when node
    (let
      ((state (funcall operator index1 index2 (node-state node))))
      (when state
        (set-generated-nodes (+ (get-generated-nodes) 1))
        (create-node
          state ; Current problem state.
          (+ (node-depth node) 1) ; Depth.
          (if (eq algorithm 'a-star) ; A* is informed.
            (funcall heuristic state)
            0
          )
          node ; Parent.
        )))))

(defun generate-children (node operator algorithm &optional (max-depth 0) heuristic)
  "Generates all valid children of NODE using OPERATOR."
  (when
    (and
      (not (null node))
      (not (and (eq algorithm 'dfs) (>= (node-depth node) max-depth))) ; If DFS,
depth?
```

```
    )
    (set-expanded-nodes (+ (get-expanded-nodes) 1))
    (let
      ((children '()))
      (loop for i1 from 0 to 1 do ; Board rows.
        (loop for i2 from 0 below 6 do ; Board columns.
          (let
            ((child (new-child node operator algorithm i1 i2 heuristic)))
            (when child (push child children)))))
      (nreverse children))))
```

3.6 Funções Auxiliares

Funções auxiliares para gestão dos nós nos algoritmos de procura.

```
(defun order-nodes (original)
  "Orders nodes in ORIGINAL using the built-in stable-sort function according to node
cost."
  (stable-sort original #'< :key #'node-cost))

(defun remove-existing (l closed algorithm)
  "Removes nodes from L that are already in CLOSED."
  (remove-if #'(lambda (node) (or (null (node-state node)) (node-existsp node closed
algorithm))) l))

(defun check-solution (l objective)
  "Returns the first node in L that satisfies OBJECTIVE, or NIL if none do."
  (find-if objective l))
```

3.7 Operador de Jogo

Função utilizada para efetuar uma jogada.

```
(defun game-operator (line-index position-index board)
  "Denotes the play is to be made at line INDEX1 and index INDEX2 of the BOARD."
  (if (and (valid-linep line-index) (valid-line-indexp position-index) (not (null
board)))
    (let*
      (
        (total-pieces (cell line-index position-index board))
        (holes (distribute-pieces total-pieces line-index position-index)))
      (if (= total-pieces 0)
        nil ; Invalid operation.
        (replace-value line-index position-index (change-board holes board) 0) ; Turns
the moved value to zero and returns the new board.
      ))
    nil))
```

3.8 Heurísticas

Heurística base, que se baseia no número de peças que faltam capturar do tabuleiro:

```
(defun game-heuristic-base (board)
  "Receives a BOARD and returns the heuristic (h = o - c) where o is the number of
pieces to capture and c is the number of pieces captured."
  (let*
    (
      (total-pieces (get-total-pieces))
      (current-pieces (board-piece-count board))
      (captured-pieces (- total-pieces current-pieces)))
    (- total-pieces captured-pieces)))
```

Heurística avançada, que se baseia na distribuição das peças do tabuleiro:

```
(defun game-heuristic-advanced (board)
  "Heuristic based on total pieces left and their spread. Fewer pieces and clustered
formations lead to a lower heuristic value."
  (let* ((total-pieces (board-piece-count board))
         (empty-spaces (count-empty-spaces board))
         (piece-spread (piece-spread-factor board)))
    (if (= total-pieces 0)
      0
      (+ total-pieces (* 0.5 empty-spaces) (* 0.3 piece-spread)))))
```

4. Tipos Abstratos de Dados

4.1 Estado do Jogo (Tabuleiro)

O tabuleiro representa-se por uma lista de duas listas, em que cada lista tem 6 átomos correspondentes ao número de peças nas células.

```
(defun board-test ()
  "Returns a 2x6 board that corresponds to exercise d)."
  '((1 2 3 4 5 6)
    (6 5 4 3 2 1))
)
```

4.2 Especificação do Nó

O nó utilizado nos algoritmos de procura foi uma lista com 4 elementos, o primeiro elemento representa o estado do jogo, o segundo elemento representa a profundidade do nó, o terceiro elemento representa o valor heuristico do nó e o quarto elemento representa o nó que o gerou, ou seja, o pai.

```
(defun create-node (state depth heuristic previous)
  (list state depth heuristic previous)
)
```

5. Limitações e Opções Técnicas

A única limitação que existe é que não é calculado o fator de ramificação médio. As opções técnicas tomadas poderiam ter sido melhores, ou seja, utilizar mais a recursividade e menos instruções destrutivas e sequenciais, no entanto, encontraram-se

alguns problemas com a implementação recursiva dos algoritmos (stack overflow) pelo que se optou por utilizar aquele tipo de instruções.

6. Análise Estatística

O tempo de execução máximo foi geralmente de 30 segundos, se o mesmo fosse ultrapassado cortou-se a execução do algoritmo e assumiu-se que a solução não seria encontrada, indentificado como NA na tabela.
Foram encontradas as soluções para todos os problemas, se desejar ver com mais detalhe pode encontrar os ficheiros *raw* .txt extraídos diretamente do ficheiro log.dat.

6.1 Problema A

Estado Inicial: `((0 0 0 0 0 2) (0 0 0 0 4 0))`

Indicadores de Desempenho	BFS	DFS	A* Base	A* Advanced	SMA* Base	SMA* Advanced
Tempo de Execução	6 ms	1 ms	4 ms	6 ms	5 ms	7ms
Nós Gerados	25	11	12	14	12	14
Nós Expandidos	11	6	7	8	7	8
Penetrância	16%	~54,55%	~33,33%	~28,57%	~33,33%	~28,57%
Profundidade da Solução	4	6	4	4	4	4

6.2 Problema B

Estado Inicial: `((2 2 2 2 2 2) (2 2 2 2 2 2))`

Indicadores de Desempenho	BFS	DFS	A* Base	A* Advanced	SMA* Base	SMA* Advanced
Tempo de Execução	NA	14 ms	NA	35 ms	NA	35 ms
Nós Gerados	NA	99	NA	82	NA	82
Nós Expandidos	NA	18	NA	18	NA	18
Penetrância	NA	~18,18%	NA	~19.51%	NA	~19,51%
Profundidade da Solução	NA	18	NA	16	NA	16

6.3 Problema C

Estado Inicial: `((0 3 0 3 0 3) (3 0 3 0 3 0))`

Indicadores de Desempenho	BFS	DFS	A* Base	A* Advanced	SMA* Base	SMA* Advanced
Tempo de Execução	27826 ms	6 ms	9 ms	13 ms	7 ms	12 ms

Nós Gerados	9136	39	29	31	29	31
Nós Expandidos	1564	10	8	11	8	11
Penetrância	~0,07%	~25,64%	20,69%	~32,26%	20,69%	~32,26%
Profundidade da Solução	6	10	6	10	6	10

6.4 Problema D

Estado Inicial: `((1 2 3 4 5 6) (6 5 4 3 2 1))`

Indicadores de Desempenho	BFS	DFS	A* Base	A* Advanced	SMA* Base	SMA* Advanced
Tempo de Execução	NA	59 ms	28476 ms	349 ms	12399 ms	592 ms
Nós Gerados	NA	365	2316	380	2316	543
Nós Expandidos	NA	54	815	65	815	112
Penetrância	NA	~14,8%	~1,04%	~9,74%	~1,04%	~6,81%
Profundidade da Solução	NA	54	24	37	24	37

6.5 Problema E

Estado Inicial: `((2 4 6 8 10 12) (12 10 8 6 4 2))`

Indicadores de Desempenho	BFS	DFS	A* Base	A* Advanced	SMA* Base	SMA* Advanced
Tempo de Execução	NA	195 ms	1760 ms	1883 ms	1141 ms	2039 ms
Nós Gerados	NA	733	694	1084	694	1376
Nós Expandidos	NA	104	290	265	290	348
Penetrância	NA	~14,19%	~4,76%	~3,41%	~4,76%	~2,69%
Profundidade da Solução	NA	104	33	37	33	37

6.6 Problema F

Estado Inicial: `((48 0 0 0 0 0) (0 0 0 0 0 48))`

Indicadores de Desempenho	BFS	DFS	A* Base	A* Advanced	SMA* Base	SMA* Advanced
Tempo de Execução	NA	250 ms	320 ms	350 ms	282 ms	337 ms
Nós Gerados	NA	796	381	397	381	397

Nós Expandidos	NA	118	74	66	74	66
Penetrância	NA	~14,82%	~8,92%	~10,08%	~8,92%	~10,08%
Profundidade da Solução	NA	118	34	40	34	40

6.7 Problema G

Estado Inicial: `((8 8 8 8 8 8) (8 8 8 8 8 8))`

Indicadores de Desempenho	BFS	DFS	A* Base	A* Advanced	SMA* Base	SMA* Advanced
Tempo de Execução	NA	201 ms	1976 ms	454 ms	1591 ms	420 ms
Nós Gerados	NA	663	862	456	862	456
Nós Expandidos	NA	98	202	81	202	81
Penetrância	NA	~14,78%	~4,41%	~9,65%	~4,41%	~9,65%
Profundidade da Solução	NA	98	38	44	38	44

7. Conclusões

Podemos determinar que no que toca a requisitos pedidos, apenas faltou o cálculo do fator médio de ramificação.
Foi possível encontrar a solução óptima de todos os problemas e tendo em conta a análise de desempenho efetuada, os algoritmos informados encontram a solução óptima (para a heurística base) mas demoram mais tempo a lá chegar, enquanto que o algoritmo BFS, embora encontre a solução óptima, demora muito tempo a fazê-lo. O DFS não encontra a solução óptima no entanto é o mais rápido.
A penetrância determina quais os nós expandidos relevantes para a resolução do problema (percentagem maior é melhor) e, segundo a análise, a heurística avançada, embora não encontre a solução óptima (não é admissível) é a mais eficiente.
Com isto dito, considera-se que os problemas foram resolvidos com sucesso.