|  |
| --- |
| Manual Técnico |
| [Escreva o subtítulo do documento] |
|  |
| [Escreva a síntese do documento aqui. Normalmente, a síntese é um breve resumo do conteúdo do documento. Escreva a síntese do documento aqui. Normalmente, a síntese é um breve resumo do conteúdo do documento.] |
|  |

Manual Técnico

[Escreva o subtítulo do documento]

# Requisitos cumpridos

Todos os requisitos pedidos no enunciado foram cumpridos.

O Manual Técnico deverá conter o algoritmo geral e por partes devidamente

comentado; descrição dos objetos que compõem o projeto, incluindo dados e procedimentos;

identificação das limitações e opções técnicas. Deverá ser apresentada uma análise critica dos

resultados das execuções do programa onde deverá transparecer a compreensão das limitações

do projeto. Deverão usar uma análise comparativa do conjunto de execuções do programa

para cada algoritmo e cada problema, permitindo verificar o desempenho de cada algoritmo e

das heurísticas. Deverá, por fim, apresentar a lista dos requisitos do projeto (listados neste

documento) que não foram implementados.

# Complexidade de espaço

|  |  |
| --- | --- |
| Algoritmo | Complexidade pior caso |
| Breadth-First Search | O(número de nós) |
| Depth-First Search | O(número de nós) |
| A\* Search | O(N) |
| IDA\* Search | O(N2) |

\*N – número de nós

# Síntese do funcionamento dos algoritmos

## Breadth-First Search

Breadth-First search é como atravessar uma árvore onde cada nó é um estado que pode ser um potencial candidato para a solução. Ele expande nós da raiz da árvore e, em seguida, gera um nível da árvore de cada vez até encontrar uma solução. Em cada iteração, o nó na cabeça da fila é removido e, em seguida, expandido. Os nós filhos gerados são então adicionados à cauda da fila.

Vantagens:

* Se houver uma solução, BFS definitivamente vai descobrir.
* Se houver mais de uma solução, então BFS pode encontrar o mínimo que requer menos número de etapas.

Desvantagens:

* Necessita de muita memória.
* Se a solução estiver muito longe da raiz vai demorar muito tempo.

## Depth-First Search

Depth-First search é como atravessar cada ramo uma árvore onde cada nó é um estado que pode ser um potencial candidato para a solução. Em cada iteração, o nó na cabeça da fila é removido e, em seguida, expandido. Os nós filhos gerados são então adicionados à cabeça da fila.

Vantagens:

* Requisito de memória é mais linear face ao grafo de pesquisa
* Encontra a solução sem ter de explorar muito a árvore (contando com grandes valores de profundidade) precisando de pouco tempo e espaço.

Desvantagens:

* Não garante que encontra a solução de menor custo

## A\* Search

Esta consiste ter uma fila ordenada por **f(n)=g(n)+h(n)** em que g(n) é o custo do nó n e h(n) é o seu valor heurístico (custo estimado). Em cada iteração, o nó na cabeça da fila é removido e, em seguida, expandido. Os nós filhos gerados são então adicionados ordenadamente pelo custo à fila.

Vantagens:

* Dá um opimo caminho para um objetivo se a heurística for admissível
* Expande o menor número possível de nós

Desvantagens:

* Requer muito espaço
* Requer um calculo adicional para calcular o h\*

## IDA\* Search

Consiste em aplicar uma série de vezes o método de procura em profundidade com limiares de profundidade variáveis, em que o limite é dado em termos de ***f*.** Na primeira pesquisa o limiar *L* é dado por *f’(n0) = g(n0) + h’(n0) = h’(n0),* em que *n0* é o nó inicial. Só se expande nós com *f’(n) <= L*. Se a solução não for encontrada, passa-se a usar novo limiar tal que *L1 = min (f(n))* em que *F(n)* é o conjunto de nós visitados por expandir.

Vantagens:

* Requisito de memória é mais linear
* Dá um opimo caminho para um objetivo se a heurística for admissível

Desvantagens:

* Neste caso, sendo a solução dada por uma sequência de *N* nós o A\* explora O(N) enquanto o *IDA\** requer *1+2+...+N* ou seja *O(N2).*

# Estudo comparativo

Nesta parte do documento é apresentado um estudo comparativo entre os algoritmos utilizados na resolução dos tabuleiros do enunciado.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Tabuleiro | Heurística | Algoritmo | Tempo  em segundos | Nós gerados | Nós expandidos | Profundidade | Penetrância | Fator de ramificação (margem 0.5 erro) | Profundidade máxima |
| A | N/A | BFS | 0 | 914 | 64 | 2 | 0.0022 | 29.7340 | N/A |
| A | N/A | DFS | 0 | 70 | 5 | 5 | 0.0690 | 2.0593 | 1000 |
| A | Enunciado | A\* | 0 | 31 | 2 | 2 | 0.0645 | 5.0859 | N/A |
| A | Enunciado | IDA\* | 0 | 47 | 3 | 2 | 0.0426 | 6.3799 | N/A |
| A | Desenvolvida | A\* | 0 | 31 | 2 | 2 | 0.0645 | 5.0859 | N/A |
| A | Desenvolvida | IDA\* | 0 | 47 | 3 | 2 | 0.0426 | 6.3799 | N/A |
| B | N/A | BFS | 0 | 155 | 11 | 1 | 0.0065 | 154.6973 | N/A |
| B | N/A | DFS | 0 | 15 | 1 | 1 | 1 | 14.5312 | 1000 |
| B | Enunciado | A\* | 0 | 15 | 1 | 1 | 0.0667 | 14.5312 | N/A |
| B | Enunciado | IDA\* | 0 | 15 | 1 | 1 | 0.0667 | 14.5312 | N/A |
| B | Desenvolvida | A\* | 0 | 15 | 1 | 1 | 0.0667 | 14.5312 | N/A |
| B | Desenvolvida | IDA\* | 0 | 15 | 1 | 1 | 0.0667 | 14.5312 | N/A |
| C | N/A | BFS | Indefinido[[1]](#footnote-1) | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | N/A |
| C | N/A | DFS | 0 | 150 | 12 | 12 | 0.0769 | 1.3624 | 1000 |
| C | Enunciado | A\* | 0 | 4922 | 343 | 8 | 0.0016 | 2.7341 | N/A |
| C | Enunciado | IDA\* | 0 | 2357 | 155 | 8 | 0.0034 | 2.4744 | N/A |
| C | Desenvolvida | A\* | 0 | 116 | 8 | 8 | 0.0690 | 0.0690 | N/A |
| C | Desenvolvida | IDA\* | 0 | 144 | 10 | 9 | 0.0625 | 1.5513 | N/A |
| D | N/A | BFS | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | N/A |
| D | N/A | DFS | 0 | 1170 | 39 | 39 | 0.0328 | 1.1352 | 1000 |
| D | Enunciado | A\* | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | N/A |
| D | Enunciado | IDA\* | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | N/A |
| D | Desenvolvida | A\* | 0 | 972 | 27 | 27 | 0.0278 | 1.2093 | N/A |
| D | Desenvolvida | IDA\* | 0 | 1219 | 40 | 39 | 0.0320 | 1.1366 | N/A |
| E | N/A | BFS | Indefinido | Indefinido | Indefinido | Indefinido | Indefinido | Indefinido | N/A |
| E | N/A | DFS | 0 | 783 | 29 | 29 | 0.0364 | 1.1795 | 1000 |
| E | Enunciado | A\* | 0 | 536 | 16 | 16 | 0.0299 | 1.3643 | N/A |
| E | Enunciado | IDA\* | 0 | 602 | 18 | 17 | 0.0282 | 1.3457 | N/A |
| E | Desenvolvida | A\* | 0 | 536 | 16 | 16 | 0.0299 | 1.3643 | N/A |
| E | Desenvolvida | IDA\* | 0 | 782 | 27 | 26 | 0.0332 | 1.2078 | N/A |
| F | N/A | BFS | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | N/A |
| F | N/A | DFS | 0 | 5985 | 105 | 105 | 0.0174 | 1.0565 | 1000 |
| F | Enunciado | A\* | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | N/A |
| F | Enunciado | IDA\* | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | Indefinido1 | N/A |
| F | Desenvolvida | A\* | 1 | 5959 | 101 | 101 | 0.0169 | 1.0592 | N/A |
| F | Desenvolvida | IDA\* | 1 | 6094 | 106 | 105 | 0.0172 | 1.0567 | N/A |

No decorrer deste projeto tentamos resolver 6 tabuleiros com diferentes algoritmos de pesquisa em espaço de estado. Utilizamos algoritmos de procura não informados e algoritmos de procura informados. Para os algoritmos de procura informada utilizamos duas heurísticas, uma já fornecida no enunciado do projeto e outra desenvolvida por nós. Infelizmente nem todos os algoritmos conseguiram terminar e encontrar a solução pois o número de nós a explorar nalguns dos tabuleiros até encontrar a solução são imensos e, portanto, o algoritmo leva demasiado tempo a completar e a encontrar uma solução.

O Breadth-First Search é um bom algoritmo para encontrar a solução ótima, no entanto ele explora todos os nós das profundidades abaixo da profundidade da solução ótima. Quando o nó objetivo está numa profundidade muito alta, esta procura geralmente leva demasiado tempo para encontrar o objetivo.

O Depth-First Search é um algoritmo bom para encontrar uma solução que não é necessariamente a ótima. Este algoritmo expande a árvore até chegar à profundidade máxima (se não encontrar solução) e no caso deste problema em particular o algoritmo irá facilmente encontrar a solução pois a cada operador aplicado mais próximo se estará da solução.

O algoritmo A\* leva em conta uma heurística e consegue estimar o custo *h* utilizando a estimativa *h’*, sendo *h* o custo do caminho de um nó até à solução. Com esta estimativa é possível calcular o custo *f* de um nó *N* que será a soma do custo *g* (custo do nó inicial até ao nó *N*) e o *h’* (curto estimado desde o nó *N* até à solução). Assim para cada nó gerado é calculado o custo *f* e é expandido sempre o nó com menor valor *f*. Assim quanto mais próxima for a estimativa *h’* de *h* mais rápida e mais eficaz será a procura.

O algoritmo IDA\* utiliza um limite. Basicamente a procura é idêntica ao Depth-First Search com a particularidade de que são feitas várias procuras limitadas incrementalmente até atingir a solução. O algoritmo começa com um limite de valor 1 e executa a procura. Sempre que é encontrado um valor de *f* de um nó *N* superior ao limite não são gerados os sucessores desse nó *N*. No final da primeira procura se não for encontrada nenhuma solução, o valor mínimo *f* dos nós que ultrapassaram o limite passa a ser o novo limite e a procura recomeça. A procura termina após ser encontrada uma solução.

Tendo em conta o funcionamento dos algoritmos vamos rever os resultados obtidos em cada um dos tabuleiros.

## Tabuleiro A

Ambos as duas heurísticas tiveram os mesmos resultados. Considerando a heurística do enunciado e a desenvolvida por nós, o algoritmo A\* teve a melhor performance. Conseguiu atingir a solução ótima gerando apenas 31 nós e expandindo apenas 2 nós. Ainda assim o algoritmo Depth-First Search teve uma maior penetrância pois foi mais direto à solução. No fator de ramificação o Depth-First Search também teve menor valor que o A\*. Neste domínio de problema o algoritmo Depth-First Search tende a encontrar facilmente a solução sendo a profundidade máxima suficientemente grande.

## Tabuleiro B

Ambos as duas heurísticas tiveram os mesmos resultados. Neste problema o claro vencedor é o Depth-First Search que além de atingir a solução ótima consegue ser superior em todas as estatísticas. O A\* e o IDA\* neste caso têm o mesmo comportamento.

## Tabuleiro C

O tabuleiro C foi demasiado difícil para o algoritmo Breadth-First Search e, portanto, levaria tempo indefinido até atingir a solução, visto que o tempo é limitado não foi possível esperar até que a solução fosse encontrada. A heurística desenvolvida supera claramente a do enunciado conseguindo também gerar menos nós que o Depth-First Search.

## Tabuleiro D

Apenas o algoritmo Depth-First Search e os algoritmos de procura informada com a Heurística desenvolvida conseguiram atingir uma solução a tempo aceitável. Os restantes algoritmos não conseguiram chegar a uma solução mesmo tentando uma abordagem iterativa. O IDA\* teve um comportamento praticamente idêntico ao do Depth-First Search utilizando a Heuristica desenvolvida por nós. O A\* conseguiu encontrar uma solução numa profundidade muito inferior à encontrada tanto pelo IDA\* tanto como pelo Depth-First Search.

## Tabuleiro E

O tabuleiro C foi demasiado difícil para o algoritmo Breadth-First Search e, portanto, levaria tempo indefinido até atingir a solução, visto que o tempo é limitado não foi possível esperar até que a solução fosse encontrada. Neste tabuleiro ambas as heuristicas tiveram o mesmo resultado usando o A\*. Quanto ao IDA\*, a heurística do enunciado supera a heurística desenvolvida por nós pois encontra um solução melhor apesar de gerar mais ou menos o mesmo número de nós. O Depth-First search tem a melhor penetrância e fator de ramificação, no entanto perde na solução encontrada e no número de nós gerados e expandidos.

## Tabuleiro F

Apenas o algoritmo Depth-First Search e os algoritmos de procura informada com a Heurística desenvolvida conseguiram atingir uma solução a tempo aceitável. Os restantes algoritmos não conseguiram chegar a uma solução mesmo tentando uma abordagem iterativa. Considerando apenas a heurística desenvolvida, o algoritmo IDA\* teve praticamente o mesmo comportamento que o Depth-First Search. O A\* consegui atingir uma solução melhor com uma profundidade de 101. O A\* também consegui obter um fator de ramificação superior aos demais.

# Funções

## Projeto.lisp

* iniciar ()
  + "Função que inicializa o programa, chamando a função que apresenta o menu inicial."
* diretoria-atual ()
  + "Função que define um caminho para leitura dos ficheiros."
* pedir-directoria ()
  + "Pede a directoria dos ficheiros ao utilizador"
* menu-principal ()
  + "Apresenta o menu principal com as opcões do programa"
* menu-jogar()
  + "função responsavel por fazer uma simulação onde se escolhe o tabuleiro, objetivo, algoritmo e se necessario profundidade e heuristica"
* regras ()
  + "função que devolve as regras"
* imprime-tabuleiro ()
  + "função que imprime um tabuleiro"
* criar-linha-horizontal (lista)
  + "Imprime uma linha de valores booleanos como um linha de traços horizontais"
* criar-linha-vertical (lista)
  + "Imprime uma linha de valores booleanos como um linha de traços verticais"
* desenhar-tabuleiro-aux (matriz1 matriz2 stream)
  + "Ajuda a desenhar o tabuleiro recebendo duas listas"
* desenhar-tabuleiro (tabuleiro stream)
  + "Desenha o tabuleiro"
* imprimir-resultado (stream resultado)
  + "função que imprime o resultado de uma simulação"
* escolher-tabuleiro()
  + "função que pede ao jogador o tabuleiro a usar"
* opcao-existe (elemento lista)
  + "função que verifica se a opção que o utilizador inseriu está dentro das hipóteses"
* resultado-simulacao(resultado)
  + " função que mostra e grava o resultado da simulação"
* obter-objectivo(tabuleiro)
  + "Lê do utilizador o número objetivo de caixas a fechar"
* escolher-algoritmo()
  + "função que pede o algoritmo a usar ao utilizador"
* obter-profundidade()
  + "função que pede a profundidade ao utilizador"
* escolher-heuristica ()
  + "Recebe do utilizador a decisão de qual heurística usar"
* imprime-pai(no stream)
  + "função que imprime recursivamente todas as etapas do tabuleiro desde a raiz até ao estado atual"

## Puzzle.lisp

* criar-tabuleiro-vazio (n m)
  + "Gera um tabuleiro vazio com n linhas e m colunas"
* criar-tabuleiro-cheio (n m)
  + "Gera um tabuleiro cheio com n linhas e m colunas"
* get-arcos-horizontais (tabuleiro)
  + "Retorna a lista dos arcos horizontais de um tabuleiro"
* get-arcos-verticais (tabuleiro)
  + "Retorna a lista dos arcos verticiais de um tabuleiro"
* arco-na-posicao (i lista)
  + "Recebe uma lista de arcos e tenta inserir um arco na posição i"
* arco-aux (x y matriz)
  + "Recebe uma matriz de arcos e tenta inserir um arco na posição x y"
* arco-horizontal (x y tabuleiro)
  + "Recebe um tabuleiro e tenta inserir um arco na posição x y dos arcos horizontais"
* arco-vertical (x y tabuleiro)
  + "Recebe um tabuleiro e tenta inserir um arco na posição x y dos arcos verticais"
* numero-caixas-horizontal (tabuleiro)
  + "Dá número de caixas na horizontal"
* numero-caixas-vertical (tabuleiro)
  + "Dá número de caixas na vertical"
* criar-operacao (x y funcao)
  + "Cria uma função lambda que representa uma operação através de uma operação (arco-horizontal/arco-vertical) e a posição x e y"
* criar-operacoes-decrementarY (x y funcao)
  + "Decrementa o valor de y recursivamente e vai criando operações com o valor de x e y e a função"
* criar-operacoes-decrementarX (x y funcao)
  + "Decrementa o valor de x recursivamente e vai chamando a função 'criar-operacoes-decrementarY' com o valor de x e y e a funcao"
* criar-operacoes (n m)
  + "Gera todos os operadores possíveis para um tabuleiro de n por m"
* aplicar-consecutivamente (tabuleiro operacoes)
  + "Aplica um conjunto de operações consecutivas a um tabuleiro"
* teste-preecher (n m)
  + "Realiza um teste que gera todos os operadores possiveis e os aplica num tabuleiro n por m consecutivo, com objetivo a preecher todo o tabuleiro com arcos"
* mapear-bool-binario (lista)
  + "Mapeia uma lista de valores booleanos (t e nil) para uma lista de valores binarios (1 0)"
* criar-candidatos-aux (matriz)
  + "Remove a última coluna da matriz"
* alisa (lista)
  + "Retorna a lista com todos os elementos contidos na lista principal"
* criar-candidatos (matriz)
  + "Cria uma matriz com os candidatos tendo em conta que arcos paralelos na mesma caixa são considerados candidatos"
* numero-caixas-fechadas (tabuleiro)
  + "Devolve o número fechadas num tabuleiro"
* bfs (abertos sucessores)
  + "Função de ordenação e junção da lista de abertos com a lista de sucessores no algoritmo breadth-first"
* dfs (abertos sucessores)
  + "Função de ordenação e junção da lista de abertos com a lista de sucessores no algoritmo depth-first"
* a-asterisco (abertos sucessores)
  + "Função de ordenação e junção da lista de abertos com a lista de sucessores no algoritmo a\*"
* ida-asterisco (abertos sucessores)
  + "Função de ordenação e junção da lista de abertos com a lista de sucessores no algoritmo ida\*"
* sucessores (no lista-operadores f-algoritmo prof-max &optional (heuristica nil))
  + "Gera os sucessores"
* criar-solucao (o)
  + "cria a solução"
* heuristica (o)
  + "heuristica do professor"
* verificar-n-arcos-faltam (caixa n)
  + "Verifica o numero de arcos que faltam para completar a caixa é igual ao numero recebido"
* n-caixas-a-faltar-x-arcos(caixas n)
  + "Recebe as caixas e o numero de arcos a faltar e verifica quantas caixas existem com esse numero de arestar por completar"
* h-numero-partilhas-horizonta-duas-linhas-quadrados(linha1 linha2 n1 n2)
  + "função que calcula o numero de partilhas na horizontal recebendo duas linhas e o numero de arcos que deve faltar em cada linha"
* convert-top-bottom(linha)
  + "Função que junta as arestar de baixo e cima de cada caixa conforme a linha"
* converter-tabuleiro(tabuleiro)
  + "função que converte o tabuleiro em caixas"
* converter-aux(tops-bottoms lefts-rights)
  + "função que junta os tops-bottoms e os left-rights em toda uma caixa"
* mapear-para-binario (matriz)
  + "Função que recebe uma matriz3D e a tranforma para binario nil-0 e t-1"
* calcular-heuristica2 ( n-caixas-objetivo ;;caixas a fazer

n-caixas-fechadas ;; caixas já feitas

n-caixas-faltar-1-arcos ;;caixas com 1 arco por acabar

n-caixas-faltar-2-arcos ;;caixas com 2 arco por acabar

n-caixas-faltar-3-arcos;;caixas com 3 arco por acabar

n-caixas-faltar-4-arcos;;caixas com 4 arco por acabar

)

* + "função que calcula a segunda heuristica"
* heuristica-2 (o)
  + "função que devolve o calculo da heuristica para um no"

## Procura.lisp

* existep (no lista f-algoritmo)
  + "Definir o predicado existep que permite verificar se um nó existe numa lista .
* existe-solucao (lista f-solucao f-algoritmo)
  + "Verifica se existe uma solucao ao problema numa lista de sucessores para o algoritmo dfs"
* calcular-numero-nos-gerados (fator-ramificacao profundidade)
  + "Calcula o número de nos gerados a partir do fator de ramificação e a profundidade"
* bisecao (profundidade numero-nos-gerados margem &optional (minimo 0) (maximo numero-nos-gerados
  + "faz a bissecção "
* procura-generica (no-inicial ; nó inicial
  + - f-solucao ; função que verifica se um nó é uma solucao
    - f-sucessores ; função que gera os sucessores
    - f-algoritmo ; algoritmo
    - lista-operadores ; lista dos operadores
    - &optional
    - (prof-max nil) ; profundidade máxima
    - (heuristica nil) ; heuristica
    - (abertos (list no-inicial)) ; lista de abertos
    - (fechados nil) ; lista de fechados
    - (tempo-inicial (get-universal-time)) ; timestamp de inicio da procura
    - (nos-gerados 0) ; numero de nos gerados
    - (nos-expandidos 0) ; numero de nos expandidos
    - (margem-bisecao 0.5)
    - )
  + "Permite procurar a solucao de um problema usando a procura no espaço de estados. A partir de um estado inicial, de uma funcao que gera os sucessores e de um dado algoritmo. De acordo com o algoritmo pode ser usada um limite de profundidade, uma heuristica e um algoritmo de ordenacao"
* procura-generica-ida-asterisco-aux (no-inicial ; nó inicial

f-solucao ; função que verifica se um nó é uma solucao

f-sucessores ; função que gera os sucessores

f-algoritmo ; algoritmo

lista-operadores ; lista dos operadores

limite ; limite de custo f da procura

heuristica ; heuristica

&optional

(nos-gerados 0) ; numero de nos gerados

(nos-expandidos 0) ; numero de nos expandidos

(abertos (list no-inicial)) ; lista de abertos

(fechados nil) ; lista de fechados

(margem-bisecao 0.5) ; margem de erro utilizada no metodo de bisecao)

* + "Permite procurar a solucao de um problema usando a procura no espaço de estados. A partir de um estado inicial, de uma funcao que gera os sucessores e de um dado algoritmo. De acordo com o algoritmo pode ser usada um limite de profundidade, uma heuristica e um algoritmo de ordenacao"
* substituir (i valor l)
  + "Substitui um elemento de uma lista correpondente ao índice i pelo valor"
* elemento-por-indice (i l)
  + "Devolve o elemento de uma lista correspondente ao índice i"
* matriz2d-transposta (m)
  + "Faz a transposta da matriz m"
* limpar-nils (lista)
  + "Remove os nils de uma lista"
* no-criar (estado &optional (pai nil) (profundidade 0) (controlo nil))
  + "Cria um nó"
* no-estado (no)
  + "Devolve o estado do nó"
* no-pai (no)
  + "Devolve o pai do nó"
* no-profundidade (no)
  + "Devolve a profundidade do nó"
* set-no-estado (no estado)
  + "Altera o estado de um nó"
* set-no-pai (no pai)
  + "Altera o pai do nó"
* set-no-profundidade (no profundidade)
  + "Altera o pai do nó"
* set-no-controlo (no controlo)
  + "Altera o elemeto de controlo do nó"
* no-controlo-g (no)
  + "Devolve o g do nó"
* no-controlo-h (no)
  + "Devolve o h do nó"
* no-controlo-f (no)
  + "Devolve o f do nó"
* set-no-g (no g)
  + "Altera o valor g do nó"
* set-no-h (no h)
  + "Altera o valor h do nó"
* set-no-f (no f)
  + "Altera o valor f do nó"

# Ficheiros

Estatísticas.dat

Ficheiro onde vai ser guardado cada resultado de uma simulação, sempre que o utilizador realizar uma simulação, este ficheiro irá ser editado, sedo acrescentado as estatísticas da simulação ao mesmo.

Problemas.dat

Os tabuleiros deverão constar neste ficheiro, até um máximo de 7 tabuleiros.

# Heurísticas

## Heurística dada

A heurística dada consiste em h(x) = o(x) − c(x) − 1 (1) em que:

• o(x) é o objetivo para esse tabuleiro: o número de caixas a fechar no tabuleiro x,

• c(x) é o número de caixas já fechadas do tabuleiro x.

Esta heurística acaba por ser admissível pois h\* <= h, no entanto, esta toma valores negativos afetando assim o algoritmo A\* de forma não desejável podendo mesmo vir a enganá-lo.

## Heurística desenvolvida

A heurística desenvolvida tomou vários caminhos, começou por ser uma heurística que se pensava admissível, mas, após testes do grupo encontrou-se um caso onde não era admissível. Devido a não trazer grande beneficio decidiu-se criar uma nova com base na antiga, esta consiste em:

(Nºcaixas a usar com 1 arco por faltar +

(Nºcaixas a usar com 2 arco por faltar \* 2) +

(Nºcaixas a usar com 3 arco por faltar \* 3) +

(Nºcaixas a usar com 4 arco por faltar \* 4) ) - 1

Esta heurística não é admissível, no entanto esta permite verificar o conteúdo aprendido em aula. Em aula falou-se que uma heurística não admissível tende a encontrar uma solução mais rapidamente, o que se pode comprovar nas estatísticas recolhidas. Em certos casos ela ainda toma admissibilidade devido ao de-incremento de 1 que surge devido a duas caixas com 1 arco por faltar que é partilhável. Assim apesar de não admissível esta heurística foi um bom objeto de estudo tendo resolvido todos os tabuleiros do enunciado com A\* e IDA\*.

# Limitações do Programa

O programa necessita de um compilador lisp para compilar. Neste caso foi usado LispWorks para compilar o programa. Este apresenta limitações de stack e heap, assim o programa fica limitado apenas a resolver tabuleiros que não ultrapassem os recursos disponíveis no lispworks. Sendo assim, impossível de determinar algumas estatísticas com alguns algoritmos para certos tabuleiros.

1. Devido à Heap [↑](#footnote-ref-1)