Manual Técnico

Jogo do Cavalo

Índice

- Introdução
- Regras
- Ficheiro Projeto
  - Interface com o utilizador
  - Ler e Enviar ficheiros .dat
- Ficheiro Puzzle
- Ficheiro Procura
  - Heurística
  - Algoritmos DPS e BFS
  - Algoritmo A*
  - Calculos de Eficiência
- Resultados Finais
  - Resultados (sem regras)
  - Resultados (com regras)
- Observações Finais

Introdução

Este documento tem a finalidade de fornecer a documentação Técnica relacionada com o programa desenvolvido no âmbito da disciplina de Inteligência Artificial, escrito em *Lisp,* do qual o objetivo é resolver tabuleiros do ***Jogo do Cavalo*** com a utilização de algoritmos e árvores de procura, tal tabuleiro deve ser escolhido pelo utilizador assim como o algoritmo a ser executado.

> Este projeto foi produzido e desenvolvido no IDE ***Visual Studio Code*** com a utilização do interpretador de ***Lisp*** CLisp. **Tedo sido apenas compilado e testado neste ambiente.**

Os ficheiros de código foram divididos e organizados da seguinte forma:

- **projeto.lisp** Carrega os outros ficheiros de código, escreve, lê ficheiros e trata da interação com o utilizador.
- **puzzle.lisp** Código relacionado com o problema.
- **procura.lisp** Deve conter a implementação de:
  1. Algoritmo Breath-First-Search (BFS)
  2. Algoritmo Depth-First-Search (DFS)
  3. Algoritmo A*

Regras

As regras impostas para as jogadas são:

- Se a casa escolhida tiver um número com dois dígitos diferentes, por exemplo 57, então, em consequência, o número simétrico 75 é apagado do tabuleiro, tornando esta casa inacessível durante o resto do jogo. Ou

seja, nenhum cavalo pode terminar outra jogada nessa casa.

- Se um cavalo for colocado numa casa com um número "duplo", por exemplo 66, então qualquer outro número duplo pode ser removido e o jogador deve escolher qual em função da sua estratégia (por default remover a de maior valor). Depois de um jogador deixar a casa para se movimentar para outra, a casa onde estava fica também inacessível para o jogo, ficando o numero da casa apagado.

Para boa interpretação de todos os problemas foi criada uma variável global que permite adicionar ou retirar as regras no jogo (esta encontrada no topo do ficheiro puzzle):

```
(setq aplicar-regras nil)
```

Esta retira a aplicação das regras durante o processo dos algoritmos, pode ser desligadas com "nil" e ligadas com "t".

Aplicamos esta implementação devido a ser impossível realizar o problema "E" com recurso as regras, mesmo fazendo a mão prova-se que não é possível chegar a uma solução.

O projeto será entregue com as regras aplicadas e portanto sem solução para o "E".

Ficheiro Projeto

Neste ficheiro podemos encontrar o carregamento de outros ficheiros de código, a leitura e escrita do ficheiro ***problemas.dat,*** escrita no ficheiro ***resultados.dat,*** por fim toda a interface e gestão da comunicação com o utilizador.

Interface com o utilizador

Para a interface com o utilizador foram desenvolvidas várias funções de print e controlo de interface com a leitura do input por parte do utilizador e gestão do mesmo dependendo da opção que o mesmo seleciona.

- Todo o programa é inicializado quando o utilizador corre o programa, este dará inicio com a função ***(first-start)*** que cria o novo tabuleiro para o problema F (falado mais a frente) e chama a função ***(start)*** que realmente inicializa todo o ciclo de interações possíveis por parte do utilizador e permitirá ao mesmo aceder a todas as opções.

```
(defun start ()
  (progn
    (start-message)
    (let ((in (read)))
      (if (or (not (numberp in)) (> in 1) (< in 0)) (start))
      (cond
        ((eq in 1) (choose-problem))
        ((eq in 0) (progn (format t "Obrigado!")(quit)))
      )
  ...)
```

> Nesta função podemos verificar o uso do ***(read)*** que é utilizada em todos os métodos que pedem ao utilizador um input, este apresenta as opções de input na função ***(start-message)*** que é meramente um

print e pede ao utilizador uma opção, dependendo da opção o programa vai proceder para a opção selecionada, caso a opção dada não seja viável será pedido um novo input.

- Após o utilizador escolher o problema que quer uma sequência de pedidos de input vão ser apresentados com a escolha de Algoritmo a utilizar, este terá 3 opções de algoritmo:

```
(defun choose-algorithm (problem)
  (progn (choose-algorithm-message)
    (let ((opt (read)))
      (cond
        ((not (numberp opt)) (progn (format t "Insira uma opcao valida!!") (choose-
algorithm problem)))
        ((or (> opt 3) (< opt 0)) (progn (format t "Insira uma opcao valida!!")
(choose-algorithm problem)))
        ((eq opt 0) (choose-problem))
        (T
          (ecase opt
            (1
              (let* ((maxDepth (choose-depth problem))
                (solution (list problem (get-internal-real-time) (dfs (node-from-
problem problem) maxDepth) (get-internal-real-time) 'DFS maxdepth)))
                (final-results solution)
              )
            )
            (2
              (let
                ((solution (list problem (get-internal-real-time) (bfs  (node-from-
problem problem)) (get-internal-real-time) 'BFS)))
                (final-results solution)
              )
            )
            (3
              (let* (
                  (heuristic (choose-heuristic problem))
                  (solution (list problem (get-internal-real-time) (a-star (node-
from-problem problem) heuristic) (get-internal-real-time) 'A* heuristic) )
                )
                (final-results solution)
              )
            )
```

Este método tem uma função um bocado mais essencial, dependendo do input do utilizador este vai executar e apresentar algoritmos diferentes, dependendo os algoritmos pode gerar novas janelas de input, assim sendo:

- Se o algoritmo selecionado for o DFS irá ser pedido ao utilizador a profundidade máxima que o algoritmo deve procurar, executar o algoritmo e logo de seguida apresentar os resultados.
- Se o utilizador selecionar BFS este irá apenas executar o algoritmo e apresentar os resultados.
- Se o utilizador selecionar A* irá pedir ao utilizador para selecionar o tipo de Heurística desejada (Heurística Base ou Heurística Personalizada), executar o algoritmo e mostrar os resultados.

- O utilizador ao ir escolher o tabuleiro que pretende resolver é lhe apresentado a lista do problemas lidos do ficheiro ***problemas.dat*** organizados e numerados para o utilizador conseguir facilmente escolher, para isso utilizamos:

```
(defun  show-problems (&optional (i 1) (problems (get-problems)))
  (cond
    ((null problems))
    (T (let ((problem (car problems)))
      (format T "~%~d - Problema ~d (~d pontos):~%" i (car problem) (cadr problem))
      (print-board (caddr problem)))
      (show-problems (+ i 1) (cdr problems))
    )
...)

(defun print-board (board &optional (stream-to-write-to t))
  (cond
    ((not (car board))
                (format stream-to-write-to "~%-------------------------------------
-----------~%"))
    (T
      (progn
        (print-row (car board) stream-to-write-to)
        (print-board (cdr board) stream-to-write-to)
      )
 ...)
```

> ***(show-problems)*** permite o print de todos os prolemas recolhidos até que a lista se encontre vazia, usamos a função ***(let)*** para guardar uma variável temporária que será cada tabuleiro, damos print do número do problema, da letra designada ao problema, número de pontos a adquirir e o tabuleiro inicial com com a ajuda da função ***(print-board),*** de seguida de forma recursiva mostramos todos os tabuleiros dando mais 1 valor ao "***i***" e o resto da lista com ***(cdr problems).***

Ler e Enviar ficheiros.dat

- Dois dos objetivos do projeto era conseguir ler os problemas do ficheiro **"problemas.dat"** e depois de achar uma solução para o problema enviar os dados para um ficheiro **"resolução.dat"**, foram então desenvolvidas duas funções para cumprir estes requisitos:

```
(defun get-problems()
  (progn
    (with-open-file (stream "problemas.dat" :if-does-not-exist :error)
      (do(
          (result nil (cons next result))
          (next (read stream nil 'eof) (read stream nil 'eof))
        )
        ((equal next 'eof) (reverse result))
      )
...)
```

Para obtermos os problemas usamos a função ***(with-open-file)***, guardamos o resultado de cada problema no result e vamos lendo o ficheiro com o next e juntando até ao ***éof (end of file)***.

```
(defun final-results (solution)
  (with-open-file (stream "./resultados.dat" :direction :output :if-exists :append
:if-does-not-exist :create)
    (write-final-results solution stream)
    (close stream)
  )
  (write-final-results solution t)
)
```

Na exportação dos resultados finais de um problema recebemos a solução completa com todos os dados necessários e de novo com ***(with-open-file)*** criamos o ficheiro ou caso já esteja criado adicionamos a nova informação ao mesmo. Por fim chamamos a função ***(write-final-results).***

```
(defun write-final-results (solution type)
  (let* (
      (problem (+ 1 (first solution)))
      (start-time (second solution))
      (solution-node (third solution))
      (end-time (fourth solution))
      (search (fifth solution))
      (elapsed-time (* (/ (float (- end-time start-time)) internal-time-units-per-
second) 1000))
      (seconds (truncate (/ elapsed-time 1000)))
      (miliseconds (mod (round elapsed-time) 1000))
    )
    (progn
      (format type "~t RESULTADOS FINAIS DA RESOLUCAO DO TABULEIRO:~%")
      (format type "~%~t -- Problema: ~a" problem)
      (format type "~%~t -- Algoritmo: ~a " search)
      (if (equal search 'A*)
        (format type "~%~t -- Heuristica: ~a " (car (last solution)))
      )
      (if (equal search 'DFS)
        (format type "~%~t -- Profundidade Maxima: ~a " (car (last solution)))
      )
      (format type "~%~t -- Tempo Demorado: ~w.~w segundos" seconds miliseconds)
      (format type "~%~t -- Numero de nós gerados: ~a" (generated-nodes solution-
node))
      (format type "~%~t -- Numero de nós expandidos: ~a" (third solution-Node))
      (format type "~%~t -- Penetrancia: ~F" (penetrance solution-Node))
      (format type "~%~t -- Fator de ramificacao media: ~F" (average-branching-fator
solution-Node))
      (format type "~%~t -- Profundidade da solucao: ~a" (solution-lenght solution-
node))
      (format type "~%~t -- Caminho solução: ~a" (first solution-node))
      (format type "~%~%~t -- Tabuleiro Inicial: ~%")
      (format type "-------------------------------------------------")
      (print-node-individual (first-node solution-node) type)
      (format type "~%~t -- Tabuleiro Final: ~%")
```

```
      (format type "-------------------------------------------------")
      (print-node-individual (final-node solution-node) type)
      (format type "~%~%")
      (if (equal type t)
        (format t "Obrigado por jogar!~%~%Escreva a instrucao (start) para comecar de
novo.")
   ...)
```

A função ***(write-final-results)*** recebe a *stream* e toda a informação da solução e organiza em variáveis para facilitar o seu acesso, desta maneira a função irá depois dar print da informação tanto no terminal para o utilizador ver, mas também para o ficheiro dessa maneira guardando os dados.

```
(defun f-to-file ()
  (let* (
      (problems (get-problems))
      (problem-f (find-if (lambda (problem) (string= "F" (car problem))) problems))
    )
    (cond
      (problem-f
        (let* (
            (updated-problems
              (mapcar
                (lambda (problem)
                  (if (string= (car problem) "F")
                    (list (car problem) (cadr problem) (random-board))
                    problem
                  )
                )
                problems
              )
            )
          )
          (with-open-file (output-stream "problemas.dat" :direction :output)
            (progn
              (mapcar (lambda (problem) (write-problem problem output-stream))
updated-problems)
              (finish-output output-stream)
              (close output-stream)
            )
          )
          updated-problems
        )
      )
      (t "Didn't find problem f")
 ...)

(defun write-problem (problem stream)
  (progn
    (format stream "~%(~w ~w~%~t(~%" (car problem) (cadr problem))
    (mapcar
      (lambda (line)
        (progn
          (format stream "~2t(")
```

```
          (mapcar
            (lambda (cel)
              (format stream "~5a" cel)
            )
            line)
          (format stream ")~%")
        )
      )
      (car (last problem)))
    (format stream "~t)~%)~%~%")
  ...)
```

> Estes 2 métodos tem como objetivo colocar um novo tabuleiro "F" para cada inicio do programa feito pelo utilizador, a função ***(f-to-file)*** vai buscar todo os problemas através da função ***(get-problems)*** que a partir dai procura pelo problema "F" na lista de problemas, quando encontrar vai atualizar os problemas com a nova tabela, depois rescrever todo o ficheiro ***resultados.dat*** com os problemas atualizados de forma bem formatada com a função ***(write-problem)*** que recebe cada problema e a stream e escreve no ficheiro de maneira que fique bem identada.

Ficheiro Puzzle

No ficheiro Puzzle teremos todos os métodos relacionados com as operações e gestão do tabuleiro, este ficheiro é essencial para todo o processo dos algoritmos.

```
(defun print-board (board &optional (stream-to-write-to t))
  (cond
    ((not (car board)) (format stream-to-write-to "----------------------------------
-----------------~%"))
    (T
      (progn
        (print-row (car board) stream-to-write-to)
        (print-board (cdr board) stream-to-write-to)
      )
...)

(defun print-row (row &optional (stream-to-write-to t) (column 9) )

  (cond
    ((> 0 column) (format stream-to-write-to "~%"))
    (t
      (progn
        (format stream-to-write-to "~5a" (car row))
        (print-row (cdr row) stream-to-write-to (1- column) )
      )
  ...)
```

> Estas função tem como objetivo dar print de um tabuleiro de jogo de maneira legível, estas funções podem ser utilizadas tanto para meio do algoritmo ver como está o estado do tabuleiro tanto para ver os tabuleiros iniciais para selecionar na interface do utilizador.

```
(defun current-position (board &optional (value-to-find t) (row 0) (column 0) )
  (cond
    ((null board) nil)
    ((eq value-to-find (cell board row column)) (list row column))
    ((< 9 row) nil)
    ((< 9 column) (current-position board  value-to-find (1+ row)))
    (t (current-position board value-to-find row (1+ column)))
...)
```

> Esta função tem como objetivo procurar o cavalo no tabuleiro de maneira iterativa, vai procurar por todas as colunas todas as posições pelo átomo "t" representativo do cavalo, este é útil para perceber qual a movimentação que o mesmo pode fazer e para gerir os nós nos algoritmos.

- Esta função tem como o objetivo colocar o cavalo em toda a primeira linha do tabuleiro para a geração dos primeiros nós dos algoritmos.

```
(defun all-horse-start-positions (board &optional (column 0))
  (cond
    ((current-position board) nil)
    (T (let ((first-row (car board)))
      (cond
        ((> column 9) '())
        ((nth column first-row)
          (append (list (set-horse-start-position board column)) (all-horse-start-
positions board (1+ column)))
        )
        (t (all-horse-start-positions board (1+ column)))
   ...)
```

> Funciona da seguinte maneira, caso o cavalo não esteja colocado vai percorrer toda a primeira linha e colocar o cavalo em todas as posições possíveis e retornar todos os tabuleiros com os cavalos colocados.

- O operador principal do tabuleiro é a movimentação do cavalo ao longo das várias jogadas nos algoritmos.

```
(defun move (board amount-row amount-column)
  (let
    ((horse-position (current-position board)))
    (cond
      ((null horse-position)
        nil
      )
      ((or
        (> (+ (car horse-position) amount-row) 9)
        (< (+ (car horse-position) amount-row) 0)
        (> (+ (cadr horse-position) amount-column) 9)
        (< (+ (cadr horse-position) amount-column) 0)
        ) nil)
      (T
        (let
          ((value-at (cell board (+ amount-row (car horse-position)) (+ amount-column
```

```
(cadr horse-position)) )))
          (cond
            ((not (numberp value-at)) nil)
            (T
              (if aplicar-regras
                (apply-number-rules (place
                  (+ amount-row (car horse-position))
                  (+ amount-column (cadr horse-position))
                  (place (car horse-position) (cadr horse-position) board)
                  T
                ) value-at)
                (place
                  (+ amount-row (car horse-position))
                  (+ amount-column (cadr horse-position))
                  (place (car horse-position) (cadr horse-position) board)
                  T
                )
            ...)
```

> Primeiramente encontra a posição do cavalo no tabuleiro, caso o mesmo não seja encontrado retorna NIL, caso seja, verifica se ele está dentro dos limites, caso passe a todos os requisitos avança. A partir dai vai atribuir á variável "value-at" a posição que pretende ir, se o programa for executado com as regras vai aplicar as regras para a posição a mover, mover o cavalo e colocar NIL na posição anterior com a função ***(place)***, caso não sejam aplicadas move apenas o cavalo e coloca NIL na posição anterior.

- Para a execução das regras impostas foram criadas uma sequência de funções para gerir essas regras estas sendo:

```
(defun apply-number-rules (board val)
  (remove-reverse-number (remove-double-number board val) val)
)

(defun remove-double-number (board val)
  (cond
    ((not (numberp val)))
    ((= val (reverse-digits val))
      (let* (
          (double-digit-numbers (find-double-digit-numbers board))
          (position-to-remove (car double-digit-numbers))
        )
        (if position-to-remove
          (place (car position-to-remove) (cadr position-to-remove) board)
          board
        )
      )
    )
    (t board)
  ...)

(defun remove-reverse-number (board val)
  (let ((pos (current-position board (reverse-digits val))))
    (cond
      ((null pos) board)
```

```
      (t (place (car pos) (cadr pos) board))
   ...)

(defun find-double-digit-numbers (board &optional (value 1))
  (cond
    ((> value 9) '())
    ((current-position board (* value 11))
      (append
        (list (current-position board (* value 11)))
        (find-double-digit-numbers board (1+ value))
      )
    )
    (t  (find-double-digit-numbers board (1+ value)))
 ...)

(defun reverse-digits (n)
  (cond
    ((not (numberp n)) nil)
    ((> n 99) nil)
    ((< n 10) (* n 10))
    (t
      (let (
          (first-digit (floor n 10))
          (second-digit (mod n 10))
        )
        (+ (* 10 second-digit) first-digit)
      )
  ...)
```

A função ***(apply-number-rules)*** vai chamar a função ***(remove-reverse-number)*** e ***(remove-double-number)*** assim chamando caso o número seja duplo ou não, caso não seja será removido o seu simétrico, caso seja será removido o número duplo mais pequeno encontrado no tabuleiro. A função ***(reverse-digits)*** recebe um número e retorna o seu simétrico como função auxiliar para ***(remove-reverse-number).***

- Para o ultimo problema, o problema "F" teremos de gerar um tabuleiro completamente aleatório para isso utilizamos:

```
(defun shuffle (board)
  (cond
    ((car board) (let (
          (n (nth (random (length board)) board))
        )
        (append (cons n nil) (shuffle (remove n board)))
      )
    )
    (T '())
...)

(defun list-numbers (&optional (n 100))
  (cond
    ((= n 1) (list 0))
    (T (append (list (1- n)) (list-numbers (1- n))))
 ...)
```

```
(defun random-board (&optional (board (shuffle (list-numbers))) (n 10))
 (cond
  ((null board) nil)
  (t (cons (subseq board 0 n) (random-board (subseq board n) n))
...)
```

> Primeiramente criamos uma lista de números de 0 até 100 na função ***(list-numbers)*** seguidamente misturamos os números da lista com a função ***(shuffle)*** com a lista misturada é criada um novo tabuleiro na função ***(random-board)*** com esses dados que seguidamente iram ser colocados no ficheiro ***problemas.dat.***

Ficheiro Procura

O ficheiro de procura contém todas as funções para o correto funcionamento dos algoritmos:

- Esta é a função responsável por criar um nó, sendo a base do ficheiro esta recebe o tabuleiro atual, a pontuação atual, a pontuação objetivo nó pai, consequentemente terá a profundidade do nó, a sua heurística e a função de avaliação.

```
(defun create-node (board score final-score parent &optional (d 0) (h 0) (f 0))
  (list board score final-score parent d h f)
)
```

- Temos também algumas funções seletoras para facilitar a gestão de dados

```
(defun node-board (node)
  (nth 0 node)
)

(defun node-score (node)
  (nth 1 node)
)

(defun node-final-score (node)
  (nth 2 node)
)

(defun node-parent (node)
  (nth 3 node)
)

(defun node-depth (node)
  (nth 4 node)
)

(defun node-heuristic (node)
  (nth 5 node)
)

(defun node-f (node)
```

```
  (nth 6 node)
)

(defun final-node (node)
  (car(last node))
)
```

- Esta é a função responsável por criar um nó solução, este vai receber o nó, o tamanho da lista de abertos e o tamanho da lista de fechados

```
(defun create-solution-node (node size-of-queue size-of-visited)
  (list (node-to-movements node) size-of-queue size-of-visited node)
)
```

- Foram criadas funções seletoras para facilitar a gestão de dados

```
(defun solution-node-path (solution-node)
  (nth 0 solution-node)
)

(defun solution-node-size-queue (solution-node)
  (nth 1 solution-node)
)

(defun solution-node-size-visited (solution-node)
  (nth 2 solution-node)
)

(defun solution-node-node (solution-node)
  (nth 3 solution-node)
)

(defun generated-nodes (solution)
  (+ (solution-node-size-queue solution) (solution-node-size-visited solution))
)
```

- Esta função faz a verificação para saber quando um nó é uma solução do problema, esta recebe um nó e verifica se a pontuação atual é superior ou igual à pontuação objetivo.

```
(defun is-solution (node)
  (<= (node-final-score node) (node-score node))
)
```

- Esta função é a responsável por fazer os cálculos da pontuação atual, esta recebe a pontuação atual, o tabuleiro do nó pai e o tabuleiro do nó filho e adiciona a pontuação que o cavalo obteve na nova jogada á pontuação que já tinha.

```
(defun calculate-score (current-score previous-board new-board)
  (let* (
      (pos (current-position new-board))
      (value (cell previous-board (car pos) (cadr pos)))
    )
    (+ current-score value)
  )
)
```

Heurística

- Esta é a função responsável por acrescentar o valor heurístico e o valor da função de avaliação a um nó. A solução que encontramos foi uma função que recebe o nó e o valor heurístico e cria um nó igual mas com o valor heurístico e a função de avaliação no nó.

```
(defun apply-heuristic (node heuristic-value)
  (create-node
    (node-board node)
    (node-score node)
    (node-final-score node)
    (node-parent node)
    (node-depth node)
    heuristic-value
    (+ (node-depth node) heuristic-value)
  )
)
```

- Esta é a função responsável por calcular o valor heurístico para cada nó onde este é recebido e aplicando a formula heurística fornecida ele devolve o resultado.

```
(defun calculate-heuristic-default (node)
  (cond
    ((= 0 (total-available-squares (node-board node))) 0)
    ((= 0 (/ (float (total-points (node-board node))) (total-available-squares (node-
board node)))) 0)
    (t
      (/
        (- (node-final-score node) (node-score node))
        (/
          (float (total-points (node-board node)))
          (total-available-squares (node-board node))
        )
...)
```

Algoritmos DFS e BFS

- Esta função representa uma implementação recursiva dos algoritmos ***BFS*** e ***DFS*** visto que a sua implementação é bastante semelhante só diferenciando na linha onde se adiciona os nós filhos no inicio ou

fim da lista de nós abertos decidimos otimizar o código de forma a que a função recebe um "b" onde o valor pode ser "*True*" ou "*nil*" dependendo do algoritmo que se quer executar, mas devido a uma elevada quantidade de *Stack Overflows* não estamos a usar esta solução.

```
(defun fs-recursive (node b &optional (queue nil) (visited nil))
  (cond
    ((is-solution node) (prog2 (print-node node) visited))
    ((null node) visited)
    ((and (null queue) (null visited)) (fs node b (list node) '()))
    ((car queue)
      (if b
        (fs (car queue) b (append (cdr queue) (generate-children node)) (cons node
visited))
        (fs (car queue) b (append (generate-children node) (cdr queue)) (cons node
visited))
      )
    )
    (t visited)
  )
)
```

- Criamos outra solução com base nas otimizações previamente referidas mas onde integramos iteração com o propósito de mitigar o problema. Na função *fs* é recebido o *"node"*, *"b"* e opcionalmente a *"maximum-depth"*. A função começa por pôr o nó atual dentro da lista de abertos e cria a lista de fechados vazia, depois enquanto a lista de abertos tem nós retiramos o primeiro nó da lista de abertos e guardamos na variável *current-node* e verificamos se este é um nó solução, se sim devolvemos o nó com o formato de solução, caso contrário ele mete o *current-node* na lista de fechados, o valor do B é depois verificado para saber se foi escolhido o ***BFS*** ou o ***DFS***, para consequentemente meter os filhos no fim ou no início da lista dos nós abertos.

```
(defun fs (node b &optional (maximum-depth most-positive-fixnum))
  (let (
      (queue (list node))
      (visited '())
    )
    (loop while queue do
      (let* ((current-node (car queue)))
        (if (is-solution current-node)
          (return (create-solution-node current-node (length queue) (length
visited)))
          (progn
            (setq visited (cons current-node visited))
            (if b
              (setq queue (append (cdr queue) (generate-children current-node)))
              (if (> (1+ (node-depth current-node)) maximum-depth)
                nil
                (setq queue (append (generate-children current-node) (cdr queue)))
              )
  ...)

(defun bfs (node)
```

```
  (fs node T)
)

(defun dfs (node &optional (max-depth most-positive-fixnum))
  (fs node NIL max-depth)
)
```

Algoritmo A*

- Para o algoritmo A* definimos função seguinte. Comecamos por inserir o nó inicial dentro da lista de abertos, de seguinda começa o loop principal da função que acaba quando a lista de abertos está vazia, neste são implementadas as instruções seguintes: removemos o primeiro nó de dentro da lista e verificamos se é solução, se for é criado o nó de solução e retornado. Caso contrário para cada nó filho desse faremos o seguinte: aplicamos a heuristica, verificamos se este se encontra dentro da lista de abertos, se sim e a heuristica for melhor do que a anterior atualizamos esse valor e reordenamos a lista com base no valor da função de avaliação.

```
(defun a-star (node &optional (heuristic-function 'calculate-heuristic-default))
  (let ((queue (list node))
      (visited '()))
    (loop while queue do
      (let ((current (pop queue)))
        (if (is-solution current)
          (return (create-solution-node current (length queue) (length visited)))
          (progn
            (do* ((children (generate-children current) (cdr children))) ((null
children))
              (let* (
                  (child (apply-heuristic (car children) (funcall heuristic-function
(car children))))
                  (found-in-queue (find child queue :key #'node-board))
                  (found-in-visited (find child visited :key #'node-board))
                )
                (cond
                  (found-in-queue
                    (if (> (node-f found-in-queue) (node-f child))
                      (setq queue (order-queue (cons child (remove found-in-queue
queue))))
                    )
                  )
                  (found-in-visited
                    (if (> (node-f found-in-visited) (node-f child))
                      (setq visited (remove found-in-visited visited) queue (order-
queue (cons child queue)))
                    )
                  )
                  (t (setq queue (order-queue (cons child queue))))
              ...)
          (push current visited)
      )
...)
```

Calculos de Eficiência

- Para os cálculos de eficiência de um algoritmo foram criadas 2 funções principais e uma auxiliar estas sendo:

```
(defun penetrance (solution-node)
    (coerce (/ (node-depth (solution-node-node solution-node)) (generated-nodes
solution-node)) 'float)
)


(defun average-branching-fator (solution &optional (depth (node-depth (solution-node-
node solution))) (generatedNodes (generated-nodes solution))
  (tolerance 0.1) (min 0) (max (generated-nodes solution)))
  (let ((average (/ ( + min max) 2)))
       (cond
        ((< (- max min) tolerance) (/ (+ max min) 2))
        ((< (auxiliar-branching average depth generatednodes) 0)
                (average-branching-fator solution depth generatednodes tolerance
average max))
          (t (average-branching-fator solution depth generatednodes tolerance min
average))
       )
  )
)

(defun auxiliar-branching (average depth generatedNodes)
 (cond
   ((= 1 depth) (- average generatednodes))
   (T (+  (expt average depth)
          (auxiliar-branching average (- depth 1) generatednodes)))
  )
)
```

> A primeira função apenas calcula e retorna a penetrância da solução dividindo a profundidade da solução pelo número de nós gerados. A segunda função calcula o fator médio de ramificação que com a ajuda da função ***(auxiliar-branching)*** calcula o valor para mostrar ao utilizador no final, no resultado.

Resultados Finais

Nas tabelas a baixo temos todos os resultados obtidos dos problemas, nos resultados apresentados temos as estatísticas e informações assim como a indicação de qual foi o problema e o algoritmo aplicado.

A primeira tabela representa a aplicação dos algoritmos sem as regras de corte de casas de jogo.

Resultados (sem regras)

Tabuleiro	Algoritmo	Objetivo	Tempo de execução	Nós Expandidos	Penetrância	Fator de ramificação	Profundidade da solução
A	DFS	70	0.007 seg	6	0.375	1.59375	3
A	BFS	70	0.009 seg	8	0.3	1.7578125	3

Tabuleiro	Algoritmo	Objetivo	Tempo de execução	Nós Expandidos	Penetrância	Fator de ramificação	Profundidade da solução
A	A* (base)	70	0.007 seg	5	0.42857143	1.5039062	3
A	A* (nova)	70					
B	DFS	60	0.8 seg	10	0.666666	1.0839844	10
B	BFS	60	0.015 seg	50	0.19607842	1.2451172	10
B	A* (base)	60	0.021 seg	50	0.2	1.3183594	10
B	A* (nova)	60					
C	DFS	270	0.008 seg	8	0.3529412	1.2949219	6
C	BFS	270	0.012 seg	37	0.13636364	1.5898437	6
C	A* (base)	270	0.014 seg	27	0.20588236	1.4277344	7
C	A* (nova)	270					
D	DFS	600	0.018 seg	39	0.26	1.2207031	13
D	BFS	600	0.091 seg	280	0.03257329	1.6114502	10
D	A* (base)	600	0.096 seg	111	0.09160305	1.3112793	12
D	A* (nova)	600					
E	DFS	300	1.906 seg	2485	0.00841346	1.3330078	21
E	BFS	300	31.243 seg	28491	0.00063615	1.6235533	19
E	A* (base)	300	0.034 seg	20	0.4871795	1.1044922	19
E	A* (nova)	300					
F	DFS	2000	0.046 seg	39	0.2708334	1.0898437	39
F	BFS	2000	31.089 seg	28491	0.00063615	1.6235533	19
F	A* (base)	2000	0.369 seg	25	0.18518518	1.0876465	25
F	A* (nova)	2000					

Resultados (com regras)

Tabuleiro	Algoritmo	Objetivo	Tempo de execução	Nós Expandidos	Penetrância	Fator de ramificação	Profundidade da solução
A	DFS	70	0.009 seg	6	0.375	1.59375	3
A	BFS	70	0.009 seg	8	0.3	1.7578125	3

Tabuleiro	Algoritmo	Objetivo	Tempo de execução	Nós Expandidos	Penetrância	Fator de ramificação	Profundidade da solução
A	A* (base)	70	0.007 seg	4	0.5	1.359375	3
A	A* (nova)	70					
B	DFS	60	0.010 seg	10	0.66666	1.0839844	10
B	BFS	60	0.020 seg	50	0.19607843	1.2451172	10
B	A* (base)	60	0.024 seg	50	0.2	1.3183594	10
B	A* (nova)	60					
C	DFS	270	0.009 seg	8	0.3529412	1.2949219	6
C	BFS	270	0.021 seg	37	0.13953489	1.6376953	6
C	A* (base)	270	0.029 seg	31	0.15789473	1.5957031	6
C	A* (nova)	270					
D	DFS	600	0.038 seg	31	0.30952382	1.1894531	13
D	BFS	600	0.141 seg	229	0.04166666	1.5527344	10
D	A* (base)	600	0.218 seg	169	0.05649717	1.5124512	10
D	A* (nova)	600					
E	DFS	300	——	——	——	——	——
E	BFS	300	——	——	——	——	——
E	A* (base)	300	——	——	——	——	——
E	A* (nova)	300	——	——	——	——	——
F	DFS	2000	0.064 seg	42	0.32283464	1.0231934	41
F	BFS	2000	——	——	——	——	——
F	A* (base)	2000	0.231 seg	27	0.23636363	1.1010742	26
F	A* (nova)	2000					

Observações Finais

Como pode ser visto nos resultados finais o algoritmo A* é muito mais eficiente para problemas mais complicados tendo melhores tempos de execução e por norma penetrância melhor, o BFS é de longe o menos eficiente e que ocupa mais espaço a executar visto que nos problemas mais complexos as opções são muitas e não consegue gerir essa explosão combinatória.

Por outro lado nos problemas mais fácies e diretos o DFS pode por vezes ser mais eficiente que o A* devido a ter um caminho mais uniforme e mais direto.