Inteligência Artificial - Problema do Quatro

1. **Manual Técnico**

Inteligência Artifical - Joaquim Filipe
Quarto

Realizado por:
Ricardo Lopes 180221044
Rui Silva 180221045

2. **Acrónimos e convenções usadas**

- **base_path** : Variável global que define o caminho para o qual irão ser enviados os ficheiros criados.
- **kebab-case** : Similiar à snake case mas, ao invés de utilizar *underscores* utiliza hífens para substituir os espaços.

3. **Introdução**

Este projeto tem como objetivo principal a utilização de algoritmos de procura em espaço de estados para a resolução do problema do jogo do Quarto. O utilizador serve-se do programa, utiliza um dos algoritmos disponíveis no menu e obtém a sequência de jogadas necessárias para preencher um tabuleiro 4x4 que pode estar inicialmente vazio ou preenchido com peças do adversário e/ou do próprio utilizador.

Para cumprir os requisitos abaixo tabelados o programa foi dividido em 4 ficheiros:

- puzzle.lisp - Implementação do código relacionado com o problema.
- procura.lisp - Implementação dos algoritmos de procura.
- projeto.lisp - Interação com o utilizador e implementação da escrita e leitura de ficheiros.
- problemas.dat - Tabuleiros disponíveis para exploração.

Requisito	Descrição
Procura BFS	O programa deverá permitir que o utilizador encontre a melhor solução do problema recorrendo ao algoritmo BFS.
Procura DFS	O programa deverá permitir que o utilizador encontre a melhor solução do problema recorrendo ao algoritmo DFS após especificar a profundidade máxima.
Procura A*	O programa deverá permitir que o utilizador encontre a melhor solução do problema recorrendo ao algoritmo A* recorrendo a uma heurística predefinida.
Interface	O programa deverá de apresentar ao utilizador uma interface de lógica simplista.

4. **Instalação e Utilização**

1. Descarregar os ficheiros do programa.
2. Instalar o IDE LispWorks na versão 7.1.2.
3. Carregar os 4 ficheiros para o IDE.
4. Digitar o comando ***(jogar)***.

---

5. **Ficheiros e Funções**

```
    Estrutura de um nó
        <nó> ::= ( <jogo>::=(<tabuleiro> <reserva>) <profundidade> <nó-pai>
        <heurística>)
```

5.1 **problemas.dat**

Contém os estados de todos os jogos disponíveis ao utilizador na interface separados por espaços.

Estrutura do ficheiro (divisão por estados)

```
(
(
 ((branca quadrada alta oca) (preta quadrada baixa cheia)  0 (preta quadrada alta
oca))
 ((branca redonda alta oca) (preta redonda alta oca) (branca redonda alta cheia)
0)
 (0 (preta redonda alta cheia) (preta redonda baixa cheia) 0)
 ((branca redonda baixa oca) (branca quadrada alta cheia) (preta redonda baixa
oca) (branca quadrada baixa cheia))
)
(
 (preta quadrada alta cheia)
 (preta quadrada baixa oca)
 (branca redonda baixa cheia)
 (branca quadrada baixa oca)
)
)

...
```

5.2 **procura.lisp**

Engloba a implementação dos algoritmos de procura. Mantivemos uma abordagem recursiva ao longo de todo o projeto, e estas funções não fugiram à regra.

5.2.1 Algoritmo de Procura em Largura

Este algoritmo atribuí ao nó-corrente o valor do primeiro nó da lista de abertos e, se este nó não existir então o algoritmo dá erro, senão ir-se-á verificar se algum dos sucessores deste nó é solução do problema através da função objetivo que é recebida nos parâmetros. O algoritmo recebe também a letra do problema, a lista de abertos, de fechados e a hora atual.
A função bfs é chamada recursivamente com a concatenação dos abertos com os sucessores do nó-corrente e coloca-o nos fechados.

```
(defun bfs(no-inicial f-objetivo f-sucessores &optional letra abertos fechados
(tempo (get-universal-time)))
(let ((sucessores (funcall f-sucessores no-inicial)))
 (cond
      ((null no-inicial) nil)

     ((funcall f-objetivo no-inicial) no-inicial)

      ((lista-solucaop sucessores f-objetivo) (mostrar-solucao (lista-solucaop
sucessores f-objetivo) (1+ (length fechados)) (+ (length abertos) (length
sucessores) (1+ (length fechados))) (- (get-universal-time) tempo) letra))

      ((null abertos) (bfs (car (abertos-bfs abertos sucessores)) f-objetivo f-
sucessores letra (cdr (abertos-bfs abertos sucessores))  (cons no-inicial
fechados) tempo))

       (t (bfs (car abertos) f-objetivo f-sucessores letra (cdr (abertos-bfs
abertos sucessores)) (cons no-inicial fechados) tempo))
 )
)
)
```

5.2.2 Algoritmo de Procura em Profundidade

Este algoritmo é bastante similiar ao anterior mas, além dos parâmetros recebidos anteriormente também recebe a profundidade máxima que serve para definir até que profundidade os nós podem ser expandidos. Este algoritmo é distinguível do BFS não só pela profundidade como também pela concatenação dos sucessores com a lista de abertos, que é feita colocando os sucessores no início de abertos.

```
(defun dfs(no-inicial f-objetivo f-sucessores &optional (profundidade 9999) letra
abertos fechados (tempo (get-universal-time)))
  (let ((sucessores (funcall f-sucessores no-inicial profundidade)))
    (cond
     ((null no-inicial) nil)

     ((funcall f-objetivo no-inicial) no-inicial)

     ((lista-solucaop sucessores f-objetivo) (mostrar-solucao (lista-solucaop
sucessores f-objetivo) (1+(length fechados)) (+ (length abertos) (length
sucessores) (1+ (length fechados))) (- (get-universal-time) tempo) letra))

     ((null abertos) (dfs (car (abertos-dfs abertos (reverse sucessores))) f-
```

```
objetivo f-sucessores profundidade letra (cdr (abertos-dfs abertos (reverse
sucessores)))  (cons no-inicial fechados) tempo))

     (t (dfs (car abertos) f-objetivo f-sucessores profundidade letra (cdr
(abertos-dfs abertos (reverse sucessores))) (cons no-inicial fechados) tempo))
     )
    )
)
```

5.2.3 Algoritmo de Procura A*

A estrutura deste algoritmo é semelhante à dos anteriores. A única diferença é que todos os nós têm o seu valor heurístico e irão ser escolhidos e expandidos aqueles cujo valor heurístico for o menor dentro da lista dos nós abertos. A ordenação dos nós é feita dentro desta função ***abertos-a****

```
(defun astar(no-inicial f-objetivo f-sucessores &optional letra abertos fechados
(tempo (get-universal-time)))
  (let ((sucessores (funcall f-sucessores no-inicial)))
    (cond
     ((null no-inicial) nil)

     ((funcall f-objetivo no-inicial) no-inicial)

     ((lista-solucaop sucessores f-objetivo) (mostrar-solucao (lista-solucaop
sucessores f-objetivo) (1+ (length fechados)) (+ (length abertos) (length
sucessores) (1+ (length fechados))) (- (get-universal-time) tempo) letra))

     ((null abertos) (astar (car (abertos-a* abertos sucessores fechados)) f-
objetivo f-sucessores letra (cdr (abertos-a* abertos sucessores fechados))  (cons
no-inicial fechados) tempo))

     (t (astar (car abertos) f-objetivo f-sucessores letra (cdr (abertos-a*
abertos sucessores fechados)) (cons no-inicial fechados) tempo))
     )
    )
)
```

Heurística

Para um determinado tabuleiro x, a função heurística é tal que:
h(x) = 4 - p(x)

p(x) é o valor máximo do alinhamento de peças tomando em conta todas as possibilidade em termos de direção e características das peças, i.e., o número máximo de peças com características comuns já alinhadas na horizontal, na diagonal ou na vertical.

5.2.3 Algoritmo de Procura IDA*

A estrutura deste algoritmo é semelhante à do ***a**** . Utiliza um limiar que é atualizado consoante o valor do menor custo da lista dos abertos.Se o menor custo for maior do que o limiar, então o algoritmo explora a árvore desde o início senão continua recursivamente como o algoritmo a*.
Apesar de poupar bastante memória ao utilizar o limiar para recomeçar a exploração da árvore, vai gerar os mesmos nós várias vezes.

```
(defun idastar(no-inicial f-objetivo f-sucessores &optional (limiar (custo no-
inicial)) letra abertos fechados (tempo (get-universal-time)))
  (let ((sucessores (funcall f-sucessores no-inicial)))
    (cond
     ((null no-inicial) nil)

     ((funcall f-objetivo no-inicial) no-inicial)

     ((lista-solucaop sucessores f-objetivo) (mostrar-solucao (lista-solucaop
sucessores f-objetivo) (1+ (length fechados)) (+ (length abertos) (length
sucessores) (1+ (length fechados))) (- (get-universal-time) tempo) letra))

     ((null abertos) (idastar no-inicial f-objetivo f-sucessores (custo (car (sort
sucessores 'comparar-no))) letra (abertos-ida* abertos sucessores fechados (custo
no-inicial)) nil tempo))

     (t (idastar (car abertos) f-objetivo f-sucessores limiar letra (cdr (abertos-
ida* abertos sucessores fechados limiar)) (cons no-inicial fechados) tempo))
     )
    )
)
```

5.3 **projeto.lisp**

Neste ficheiro encontra-se implementada a interação com o utilizador. O programa é iniciado através da função ***jogar*** que apresenta o menu principal três opções disponíveis, **Jogar!** para a escolha do algoritmo, **Mostrar tabuleiros** para ver os tabuleiros disponíveis e **Sair** para terminar o jogo.

Apresentação do menu-inicial.

```
(defun menu-inicial()
  (format t "~%           Jogo do Quatro")
  (format t "~%=======================================")
  (format t "~%         Seja bem-vindo!")
  (format t "~%         1 - Jogar!")
  (format t "~%         2 - Mostrar tabuleiros")
  (format t "~%         0 - Sair~%>")
)
```

Implementação da escolha no menu-inicial.

```
(defun jogar()
(menu-inicial)
  (let ((opcao (read)))
    (cond
     ((= opcao 1) (iniciar))
     ((= opcao 2) (escolher-tabuleiro))
     ((= opcao 0) (format t "Adeus!"))
     (t (iniciar))
    )
    )
)
```

A escolha da segunça opção invoca a função ***escolher-tabuleiro*** que mostra os tabuleiros disponíveis e devolve o tabuleiro escolhido recorrendo à função ***mostrar-tabuleiro***.

```
(defun escolher-tabuleiro()
(menu-problemas)
(let* ((opcao (read))
           (problemas (escolher-ficheiro (format nil "~Aproblemas.dat"
*base_path*))))
      (cond
       ((= opcao 0) (jogar))
       ((or (< opcao 0) (> opcao 6)) (escolher-tabuleiro))
       (t (mostrar-tabuleiro opcao problemas))
      )
)
)
```

A função ***mostrar-tabuleiro*** irá apenas formatar o tabuleiro que foi extraído do ficheiro.

```
(defun mostrar-tabuleiro(opcao problemas)
 (format t "Tabuleiro ~A~%~A~%" (letra opcao) (nth (1- opcao) problemas))
 (escolher-tabuleiro)
)
```

A função ***letra*** serve para associar a cada opção que pode ser escolhida pelo utilizador uma letra do tabuleiro correspondente.

```
(defun letra (numero)
  (cond
   ((= numero 1) "A")
   ((= numero 2) "B")
   ((= numero 3) "C")
   ((= numero 4) "D")
   ((= numero 5) "E")
   ((= numero 6) "F")
```

```
;;   ((= numero 7) "G")
   (t nil)
   )
)
```

Leitura do ficheiro

Variável global do caminho base

```
(defparameter *base_path*
"C:/Users/lkrak/Desktop/RicardoLopes_180221044_RuiSilva_180221045_P1/")
```

A seguinte função ***escolher-ficheiro*** mantém o ficheiro do ***caminho*** aberto até realizar as operações da função ***ler-ficheiro***.

```
(defun escolher-ficheiro(caminho)
  (with-open-file (f caminho
                     :direction :input
                     :if-does-not-exist nil)
    (ler-ficheiro f)
    )
)
```

A função ***ler-ficheiro*** percorre o ficheiro dos tabuleiros e junta-os à variável ***tabuleiros***. Como a lista dos tabuleiros lidos termina no nil(final do ficheiro) quando o "ponteiro" chega a nil então enviamos os tabuleiros invertidos porque foram introduzidas de forma decrescente.

```
(defun ler-ficheiro(f &optional (tabuleiros '()))
  (let* ((tabuleiro (read f nil :end))
         (tabuleiros(cons tabuleiro tabuleiros)))
    (cond
     ((equal tabuleiro :end) (reverse (cdr tabuleiros)))
     (t (ler-ficheiro f tabuleiros))
     )
    )
)
```

Escrita do ficheiro

A função ***mostrar-solucao*** junta todos os requisitos necessários à solução e à eficiência do algoritmo utilizado numa lista e invoca a função ***escrever-ficheiro***.

```
(defun mostrar-solucao (no expandidos gerados tempo letra &aux (penetrancia (/
(no-profundidade no) gerados)))
  (escrever-ficheiro (list no expandidos gerados (ramificacao no gerados)
penetrancia tempo) letra)
)
```

A função ***escrever-ficheiro*** constrói um ficheiro txt com o nome ***solução-letra do tabuleiro*.txt** recorrendo aos valores da *lista*.

```
(defun escrever-ficheiro (lista letraTabuleiro)
  (if (null
       (with-open-file (str (format nil "~Asolucao~A.txt" *base_path*
letraTabuleiro)
                            :direction :output
                            :if-exists :supersede
                            :if-does-not-exist :create)
         (format str "Nó solução: ~A~%" (first lista))
         (format str "Nº de nós expandidos: ~A~%" (second lista))
         (format str "Nº de nós gerados: ~A~%" (third lista))
         (format str "Fator de ramificação: ~A~%" (fourth lista))
         (format str "Penetrância: ~A~%" (fifth lista))
         (format str "Tempo de execução: ~A s~%" (sixth lista)))
       ) (retomar-jogo letraTabuleiro) (format t "Erro a escrever ficheiro"))
  )
```

Se a escrita da solução ocorrer com sucesso, então é apresentada esta mensagem e o jogo repete-se.

```
(defun retomar-jogo (letraTabuleiro)
(format t "Sucesso a escrever para ficheiro solucao~A.txt" letraTabuleiro)
(iniciar)
)
```

Apresentação do menu dos algoritmos

```
(defun menu-algoritmo()
  (format t "~%           Jogo do Quatro")
  (format t "~%=======================================")
  (format t "~%         Escolha o algoritmo")
  (format t "~%         1 - Breadth-First Search")
  (format t "~%         2 - Depth-First Search")
  (format t "~%         3 - A* Search ")
  (format t "~%         0 - Sair~%>")
)
```

Implementação da escolha no menu-algoritmo.

A escolha da primeira opção no menu-inicial da função ***jogar*** invoca a função ***iniciar*** que apresenta o menu dos algoritmos disponíveis para a resolução do problema.

```
(defun iniciar()
  (menu-algoritmo)
  (let ((opcao (read)))
    (cond
     ((= opcao 1) (escolher-problema 'bfs))
     ((= opcao 2) (escolher-profundidade 'dfs))
     ((= opcao 3) (escolher-heuristica 'astar))
     ((= opcao 0) (jogar))
     (t (iniciar))
    )
    )
)
```

Qualquer uma das opções irá guiar o utilizador até ser invocada a função escolher-problema que utiliza a opção escolhida pelo utilizador na seguinte interface.

```
(defun menu-problemas()
  (format t "~%           Jogo do Quatro")
  (format t "~%=======================================")
  (format t "~%         Escolha o tabuleiro")
  (format t "~%         1 - Tabuleiro A")
  (format t "~%         2 - Tabuleiro B")
  (format t "~%         3 - Tabuleiro C")
  (format t "~%         4 - Tabuleiro D")
  (format t "~%         5 - Tabuleiro E")
  (format t "~%         6 - Tabuleiro F")
;;  (format t "~%         7 - Tabuleiro G")
  (format t "~%         0 - Voltar~%>")
)
```

A função ***escolher-problema*** recebe um algoritmo e, com base na opção escolhida, pelo utilizador(algoritmo e tabuleiro) irá verificar qual o algoritmo que tem de invocar para saber que argumentos tem de utilizar no mesmo.

```
(defun escolher-problema(algoritmo &optional (profundidade 9999))
    (menu-problemas)
    (let* ((opcao (read))
           (problemas (escolher-ficheiro (format nil "~Aproblemas.dat"
*base_path*)))
           )
      (cond
       ((= opcao 0) (iniciar))
```

```
        ((or (< opcao 0) (> opcao 6)) (escolher-problema algoritmo profundidade))
        (t
         (if(equal algoritmo 'dfs) (funcall algoritmo (cria-no (nth (1- opcao)
problemas)) 'no-solucaop 'sucessores profundidade (concatenate 'string (string
algoritmo) "_" (letra opcao))) (funcall algoritmo (cria-no (nth (1- opcao)
problemas)) 'no-solucaop 'sucessores (concatenate 'string (string algoritmo) "_"
(letra opcao))))
         )
        )
       )
)
```

Apresentação do menu da profundidade

```
(defun menu-profundidade()
  (format t "~%           Jogo do Quatro")
  (format t "~%=======================================")
  (format t "~%    Escolha a profundidade máxima")
  (format t "~%    -1 - Voltar~%>")
)
```

Implementação da escolha no menu-profundidade.

```
(defun escolher-profundidade(algoritmo)
  (menu-profundidade)
  (let ((profundidade (read)))
    (cond
     ((< profundidade -1) (escolher-profundidade algoritmo))
     ((= profundidade -1) (iniciar))
     (t (escolher-problema algoritmo profundidade))
     )
    )
)
```

Apresentação do menu da heurística

```
(defun menu-heuristica()
  (format t "~%           Jogo do Quatro")
  (format t "~%=======================================")
  (format t "~%         Escolha a heurística")
  (format t "~%         1 - Heurística Enunciado")
  (format t "~%         0 - Voltar~%>")
)
```

Implementação da escolha no menu-heuristica.

```
(defun escolher-heuristica(algoritmo)
  (menu-heuristica)
  (let ((opcao (read)))
    (cond
     ((= opcao 1) (escolher-problema algoritmo))
     ((= opcao 0) (iniciar))
     (t (escolher-heuristica algoritmo))
     )
    )
)
```

5.4 **puzzle.lisp**

Seletores

A função ***tabuleiro*** recebe, um nó, e devolve o tabuleiro.

```
(defun tabuleiro (no)
  (caar no)
)
```

A função ***reserva*** recebe, um nó, e devolve a reserva de peças.

```
(defun reserva (no)
  (cadar no)
)
```

A função ***linha**** recebe, um índice e o tabuleiro, e retorna uma lista que representa essa linha do tabuleiro.

```
(defun linha (indice tabuleiro)
  (cond
    ((or (< indice 0) (null tabuleiro)) nil)
    ((zerop indice) (car tabuleiro))
    (t (linha (1- indice) (cdr tabuleiro)))
  )
)
```

A função ***coluna*** recebe, um índice e o tabuleiro, e retorna uma lista que representa essa coluna do tabuleiro.

```
(defun coluna (indice tabuleiro)
  (cond
    ((or (< indice 0) (null tabuleiro)) nil)
    (t (maplist #'(lambda (linhaTabuleiro &aux (cabeca (linha indice (car
linhaTabuleiro)))) cabeca) tabuleiro))
  )
)
```

A função ***celula*** recebe, dois índices (linha e coluna) e o tabuleiro, e retorna o valor presente nessa célula do tabuleiro.

```
(defun celula (linhaTabuleiro colunaTabuleiro tabuleiro)
  (linha linhaTabuleiro (coluna colunaTabuleiro tabuleiro))
)
```

A função ***diagonal-1***, recebe, um tabuleiro, e retorna uma lista que representa uma diagonal desse tabuleiro. Considera-se que seja a diagonal a começar pela célula na 1ª linha e 1ª coluna. Vai conter uma função lambda que vai recebendo uma lista das linhas, através do maplist, criando uma variável tamanho que vai fazer a subtração desta lista pelo tamanho original do tabuleiro, conseguindo assim ir buscando as celulas incrementalmente, à medida que o maplist é percorrido.

```
(defun diagonal-1 (tabuleiro)
  (maplist #'(lambda (tabuleiroParte &aux (tamanho (- (length tabuleiro) (length
tabuleiroParte)))) (celula tamanho tamanho tabuleiro)) tabuleiro)
)
```

A função ***diagonal-2*** recebe um tabuleiro e retorna uma lista que representa uma diagonal desse tabuleiro. Considera-se que seja a diagonal a começar pela célula na última linha e 1ª coluna. Vai conter uma função lambda que irá receber uma lista de linhas, consoante o que o maplist lhe envia, nessa função vai ser calculado a diferença entre esta lista e o tamanho original do tabuleiro, conseguindo assim 2 variáveis opostas, uma que vai incrementando (tamanho) e outra que vai decrementando (tabuleiroParte), sendo assim possível ir buscar a diagonal, desde a última coluna e primeira linha até à primeira linha e última coluna, à medida que o maplist é percorrido.

```
(defun diagonal-2 (tabuleiro)
  (maplist #'(lambda (tabuleiroParte &aux (tamanho (- (length tabuleiro) (length
tabuleiroParte)))) (celula (1- (length tabuleiroParte)) tamanho tabuleiro))
tabuleiro)
)
```

A função ***no-jogo*** recebe um nó e retorna o jogo (tabuleiro e as reservas).

```
(defun no-jogo (no)
  (car no)
)
```

A função ***no-profundidade*** recebe um nó e retorna a profundidade do nó.

```
(defun no-profundidade (no)
  (cadr no)
)
```

A função ***no-pai*** recebe um nó e retorna o pai do nó.

```
(defun no-pai (no)
  (caddr no)
)
```

A função ***no-heuristica*** recebe um nó e retorna a heuristica do nó.

```
(defun no-heuristica (no)
  (cadddr no)
)
```

Construtor

A função ***cria-no*** recebe um jogo (tabuleiro e reservas) e devolve um nó que é uma lista do jogo, da profundidade e do pai do nó inicial.

```
(defun cria-no (jogo &optional (g 0) (pai nil) (heuristica (heuristica (car
jogo))))
  (list jogo g pai heuristica)
)
```

****Funções auxiliares para o cálculo da heurística**

A função ***heurística*** recebe o tabuleiro e percorre cada peça de cada linha para calcular a função ***h(x) = 4 - p(x)*** onde ***p(x)*** representa o valor máximo das propriedades em comum na linha, diagonal ou coluna onde se encontra a peça. Como ***p(x)*** obtém o valor máximo então quer-se o ***h(x)*** menor logo aplica-se a função ***min*** à lista alisada dos máximos de cada peça.

```
(defun heuristica (tabuleiro)
  (cond
```

```
   ((null tabuleiro) nil)
   (t (apply 'min (alisa (mapcar #'(lambda (linha) (mapcar #'(lambda (peca) (- 4
(calcular-max tabuleiro peca))) linha)) tabuleiro))))
   )
)
```

A função ***calcular-max*** recebe um tabuleiro e uma peça. O tabuleiro serve para calcular o índice da coluna e linha da peça. Se os índices forem inválidos então devolve nil, se a casa estiver vazia peça devolve 0, caso contrário, esta função irá devolve o máximo entre o máximo das propriedades comuns da coluna, linha e:

1. Se os índices da coluna e linha forem iguais, então a peça encontra-se na ***diagonal-1*** e tem de se calcular também.
2. Se a soma dos índices da coluna e linha forem iguais à quantidade de linhas que existem então tem de se calcular a ***diagonal-2***.
3. 0, caso contrário.

```
(defun calcular-max (tabuleiro peca &aux (colunaPeca (encontrar-peca tabuleiro
peca 'coluna)) (linhaPeca (encontrar-peca tabuleiro peca 'linha)))
  (cond ((or (null colunaPeca) (null linhaPeca)) nil)
        ((atom peca) 0)
        (t
         (max (propriedade-comum peca (coluna colunaPeca tabuleiro)) (propriedade-
comum peca (linha linhaPeca tabuleiro))
              (cond
               ((= colunaPeca linhaPeca) (propriedade-comum peca (diagonal-1
tabuleiro)))
               ((= (+ colunaPeca linhaPeca) (length tabuleiro)) (propriedade-comum
peca (diagonal-2 tabuleiro)))
               (t 0)
               )
              )
         )
        )
)
```

A função ***propriedade-comum*** recebe uma peça e uma lista de peças, e verifica qual das características da peça se repete mais vezes na lista que contém essa peça, é de notar que a lista pode ser uma diagonal, coluna ou linha.

```
(defun propriedade-comum (peca lista &optional (x 0))
(cond
       ((null peca) x)
       (t (propriedade-comum (cdr peca) lista (max x (propriedadep (car peca)
(alisa lista)))))
   )
```

A função ***encontrar-peca*** encontra o índice da coluna ou linha (funções de pesquisa) onde se encontra a peça passada nos argumentos dentro do tabuleiro se a mesma existir.

```
(defun encontrar-peca (tabuleiro peca f-pesquisa &optional (indice (1- (length
tabuleiro))))
  (cond
   ((< indice 0) nil)
   ((peca-existep peca (funcall f-pesquisa indice tabuleiro)) indice)
   (t (encontrar-peca tabuleiro peca f-pesquisa (1- indice)))
   )
)
```

A função ***peca-existep*** verifica se uma peça existe num tabuleiro recursivamente comparando cada peca do tabuleiro à peça passada por argumentos.

```
(defun peca-existep (peca lista)
  (cond
       ((null lista) nil)
       ((equal peca (car lista)) t)
       (t (peca-existep peca (cdr lista)))
       )
)
```

Funções auxiliares de verificação da solução e de geração de sucessores

A função ***casa-vaziap*** recebe dois índices (linha e coluna) e o tabuleiro e devolve T se a casa estiver vazia e NIL caso contrário. O valor de uma casa vazia no Problema do Quatro é o valor 0. Sendo para isso necessário verificar apartir do valor da celúla resultante se a mesma é um átomo, resultando no T, ou não, resultando no NIL.

```
(defun casa-vaziap (linhaTabuleiro colunaTabuleiro tabuleiro)
  (if (or (< linhaTabuleiro 0) (< colunaTabuleiro 0)) nil (atom (celula
linhaTabuleiro colunaTabuleiro tabuleiro)))
)
```

A função ***remover-peca*** recebe uma peça e uma lista com as peças de reserva e devolve uma nova lista sem essa peça de reserva. Verificando essa mesma peça recursivamente, percorrendo assim toda a lista, igualando a peça à cabeça da lista, caso seja diferente adicionando a uma nova lista essa mesma cabeça recursivamente, e caso seja igual a peça vai ser ignorada, enviando novamente a função.

```
(defun remover-peca (peca tabuleiroReserva)
  (cond ((null tabuleiroReserva) nil)
        ((equal peca (car tabuleiroReserva)) (remover-peca peca (cdr
tabuleiroReserva)))
```

```
        (t (cons (car tabuleiroReserva) (remover-peca peca (cdr
tabuleiroReserva))))
   )
)
```

A função ***substituir-posicao*** recebe um índice, uma peça e uma lista que representará uma linha do tabuleiro e substitui pelo valor pretendido nessa posição. Verificando essa mesma peça recursivamente, percorrendo assim toda a lista, igualando o indíce a zero, caso seja diferente adiciona a uma nova lista essa mesma cabeça recursivamente, e caso seja igual a peça vai ser adicionada juntamente com o resto da lista.

```
(defun substituir-posicao (indice peca linhaTabuleiro)
  (cond ((null linhaTabuleiro) nil)
        ((zerop indice) (cons peca (cdr linhaTabuleiro)))
        (t (cons (car linhaTabuleiro) (substituir-posicao (1- indice) peca (cdr
linhaTabuleiro))))
  )
)
```

A função ***substituir*** recebe dois índices (linha e coluna), uma peça e o tabuleiro. A função deverá retornar o tabuleiro com a célula substituída pelo valor pretendido. Vai utilizar a função substituir-posicao definida anteriormente. Percorrendo assim, recursivamente as linhas do tabuleiro, criando uma nova lista que vai adicionando a cabeça da lista do tabuleiro, até que a variável definida, linhaTabuleiro, seja igual a zero, significando que é a linha pretendida pelo utilizador e utilizando agora a função ***substituir-posicao*** para alterar a linha, sendo assim adicionada à lista a linha alterada, juntamente com o resto das linhas do tabuleiro não percorridas.

```
(defun substituir (linhaTabuleiro colunaTabuleiro peca tabuleiro)
  (cond ((null tabuleiro) nil)
        ((zerop linhaTabuleiro) (cons (substituir-posicao colunaTabuleiro peca
(linha linhaTabuleiro tabuleiro)) (cdr tabuleiro)))
        (t (cons (car tabuleiro) (substituir (1- linhaTabuleiro) colunaTabuleiro
peca (cdr tabuleiro)))))
)
```

A função ***no-solucaop*** recebe um no e retorna T caso o no contenha uma solução e NIL caso contrário. Neste nó vão ser testadas todas as hipóteses onde podem existir soluções, ou seja as diagonais, as linhas, percorrida com a ajuda de um mapcar que devolve uma lista de T ou NIL, onde se existir pelo menos um T é válida, ou as colunas, percorridas com um maplist de modo a conseguir, incrementalmente, aceder a cada coluna existente, devolvolvendo tambem uma lista de T ou NIL, onde se existir pelo menos um T é válida, se alguma destas conter uma solução a função irá retornar um T e NIL caso contrário.

```
(defun no-solucaop(no)
  (let ((tabuleiro (tabuleiro no)))
    (cond
       ((or (solucaop (diagonal-1 tabuleiro))
```

```
            (solucaop (diagonal-2 tabuleiro))
            (eval(cons 'or (mapcar #'solucaop tabuleiro)))
            (eval (cons 'or (maplist #'(lambda (tabuleiroParte &aux (tamanho (-
(length tabuleiro) (length tabuleiroParte)))) (solucaop (coluna tamanho
tabuleiro))) tabuleiro))))
        t)
       (t nil)
       )
    )
)
```

A função ***solucaop*** recebe uma lista e opcionalmente a lista das propriedades e retorna T caso a lista seja uma solução e NIL caso contrário. Vai ser percorrida recursivamente, até que o tamanho da lista, seja igual ao valor do retorno da função *propriedadep*, sendo testada com todas as propriedades existentes recursivamente. Sendo assim só vai encontrar uma solução quando a quantidade de propriedades iguais na lista seja igual a quatro.

```
(defun solucaop (lista &optional (props (propriedades)))
 (cond ((null props) nil)
       ((= (length lista) (propriedadep (car props) (alisa lista))) t)
       (t (solucaop lista (cdr props)))
       )
)
```

A função ***propriedades*** retorna todas as propriedades existentes.

```
(defun propriedades ()
  '(BRANCA PRETA REDONDA QUADRADA ALTA BAIXA OCA CHEIA)
)
```

A função ***propriedadep*** recebe uma peca e uma propriedade e retorna a quantidade de vezes que a propriedade se repete na lista. Percorre a lista da peça recursivamente e incrementando, sempre que a propriedade definida se repetir na lista.

```
(defun propriedadep(propriedade lista)
  (cond ((null lista) 0)
        ((equal propriedade (car lista)) (1+ (propriedadep propriedade (cdr
lista))))
        (t (propriedadep propriedade (cdr lista)))
)
)
```

A função ***alisa*** recebe uma lista com sub-listas e devolve a mesma sem sub-listas. Vai ser percorrida recursivamente, criando assim uma nova lista, sendo apenas adicionada, caso a cabeça da lista seja um átomo,

se não acontecer é chamada novamente a função mas com essa mesma cabeça, que neste caso não é um átomo, sendo assim essa cabeça percorrida.

```
(defun alisa(lista)
   (cond
       ((null lista) nil)
       ((atom (car lista)) (cons (car lista) (alisa (cdr lista))))
       (t (append (alisa (car lista)) (alisa (cdr lista))))
    )
)
```

A função ***lista-solucaop*** recebe uma lista de nós e devolve o nó que seja uma solução e NIL caso contrário. Vai funcionar recursivamente, percorrendo a lista de nós, até que, a cabeça dessa lista contenha uma solução, retornando a mesma e caso contrário NIL.

```
(defun lista-solucaop (listaJogos f-objetivo)
  (cond ((null listaJogos) nil)
        ((funcall f-objetivo (car listaJogos)) (car listaJogos))
        (t (lista-solucaop (cdr listaJogos) f-objetivo))
        )
)
```

A função ***no-existep*** recebe um nó, uma lista e um algoritmo e devolve T se o nó existe dentro da lista e NIL caso contrário. Vai funcionar recursivamente, percorrendo a lista, e dependendo do algoritmo vai ter condições que diferem, sendo que no *dfs* vai ter de comparar a profundidade do nó recebi com o da cabeça da lista, sendo que se a do nó seja a menor vai devolver T, no *bfs* vai ser comparado os tabuleiros do nó e o da cabeça da lista, sendo que esta vai condição vai ser repetida para os outros algoritmos, e devolve T se forem iguais, e por fim o ***a**** que vai comparar os custos do nó com o da cabeça da lista e caso o do nó seja menor devolve T. Sendo que se a lista for percorrida e não seja encontrado nenhum nó que entre em nenhuma destas condições é devolvido NIL.

```
(defun no-existep(no lista algoritmo)
  (cond
     ((null lista) nil)
     ((and (equal algoritmo 'dfs) (equal (tabuleiro no) (tabuleiro (car lista)))
(< (no-profundidade no) (no-profundidade (car lista))))  t)
```

Funções auxiliares dos algoritmos

A função ***abertos-bfs*** serve para construir uma nova lista de abertos que consistirá na junção da antiga lista de nós abertos com os sucessores. Se algum dos nós na lista de abertos existir nos sucessores então não irá ser introduzido. Os sucessores são postos no final da lista de nós abertos.

```
(defun abertos-bfs(nos-abertos sucessores)
  (cond ((null sucessores) nos-abertos)
        ((no-existep (car sucessores) nos-abertos 'bfs) (abertos-bfs nos-abertos
(cdr sucessores)))
        (t (abertos-bfs (append nos-abertos (cons (car sucessores) nil)) (cdr
sucessores)))
        )
)
```

A função ***abertos-dfs*** é semelhante a anterior só que os sucessores são postos no início da lista de nós abertos.

```
(defun abertos-bfs(nos-abertos sucessores)
  (cond ((null sucessores) nos-abertos)
        ((no-existep (car sucessores) nos-abertos 'bfs) (abertos-bfs nos-abertos
(cdr sucessores)))
        (t (abertos-bfs (append nos-abertos (cons (car sucessores) nil)) (cdr
sucessores)))
        )
)
```

A função *_abertos-a_** também é similar às anteriores só que insere os sucessores por ordem crescente de custo, ou seja, a lista resultante será sempre a do nó de menor custo até ao de maior custo. Para tal, é utilizada a função ***inserir-ordenado***.

```
(defun abertos-a* (nos-abertos sucessores fechados)
  (cond ((null sucessores)  nos-abertos)
        ((no-existep (car sucessores) fechados 'a*) (abertos-a* nos-abertos (cdr
sucessores) (troca-no (car sucessores) fechados)))
        ((no-existep (car sucessores) nos-abertos 'a*) (abertos-a* (inserir-
ordenado (car sucessores) (remover-no (car sucessores) nos-abertos)) (cdr
sucessores) fechados))
;;        (t (abertos-a* (sort (append nos-abertos (cons (car sucessores) nil))
'comparar-no) (cdr sucessores) fechados))
        (t (abertos-a* (inserir-ordenado (car sucessores) nos-abertos) (cdr
sucessores) fechados))
        )
)
```

A função ***abertos-ida**** é similar à anterior ***abertos-a**** só que assim que deteta que nenhum dos sucessores tem um custo menor que o limiar, retorna nil para se recomeçar a construção da árvore.

```
(defun abertos-ida* (nos-abertos sucessores fechados limiar &optional (first 0))
  (cond ((null sucessores)  nos-abertos)
        ((and (< first 1) (analisar-limiar sucessores limiar)) nil)
        ((no-existep (car sucessores) fechados 'a*) (abertos-ida* nos-abertos (cdr
```

```
sucessores) (troca-no (car sucessores) fechados) limiar (1+ first)))
        ((no-existep (car sucessores) nos-abertos 'a*) (abertos-ida* (inserir-
ordenado (car sucessores) (remover-no (car sucessores) nos-abertos)) (cdr
sucessores) fechados limiar (1+ first)))
        (t (abertos-ida* (inserir-ordenado (car sucessores) nos-abertos) (cdr
sucessores) fechados limiar (1+ first)))
        )
)
```

A função **analisar-limiar**_ recebe uma lista e um limiar e verifica se todos os nós ultrapassam ou não o limiar com o objetivo de atualizar ou não o limiar.

```
(defun analisar-limiar (lista limiar)
  (eval (cons 'and (mapcar #'(lambda(peca) (if (> (custo peca) limiar) T NIL))
lista)))
)
```

A função ***troca-no*** recebe um nó e troca-o no lugar do antigo. Esta função serve expecialmente para o algoritmo A* em que podemos ter de trocar os nós na lista de fechados caso o custo do nó corrente (a ser introduzido) for menor do que o correspondente.

```
(defun troca-no(no lista)
  (cond
      ((null lista) nil)
      ((equal (tabuleiro no) (tabuleiro (car lista))) (cons no (cdr lista)))
      (t (cons (car lista) (cons (cdr lista) nil)))
   )
)
```

A função ***comparar-no*** recebe dois nós e comparar os custos. Se o custo do primeiro for menor do que o custo de segundo.

```
(defun comparar-no(no-a no-b)
  (< (custo no-a) (custo no-b))
)
```

A função ***custo*** recebe um nó e calcula o custo do nó (profundidade + heurística).

```
(defun custo(no)
  (+ (no-profundidade no) (no-heuristica no))
)
```

A função ***remover-no*** recebe um nó e uma lista e devolve uma nova lista recursivamente sem o nó se o mesmo existir na lista.

```
(defun remover-no(no lista)
  (cond
   ((null lista) nil)
   ((equal no (car lista)) (cdr lista))
   (t (cons (car lista) (remover-no no (cdr lista))))
   )
)
```

A função ***inserir-ordenado*** certifica-se de que um nó é sempre introduzido na lista mas, se este nó tiver um custo menor do que algum dos que já estão na lista então é inserido nessa posição recursivamente através da função ***insere***.

```
(defun inserir-ordenado(no lista &optional (indice 0))
  (cond
   ((null lista) (list no))
   ((equal indice (1- (length lista))) (append lista (cons no nil)))
   ((comparar-no no (nth indice lista)) (insere no indice lista))
   (t (inserir-ordenado no lista (1+ indice)))
)
)
```

A função ***insere*** insere um valor numa posição da lista, mantendon a ordem dos restantes elementos. O primeiro elemento tem o número de ordem 0 (zero).

```
(defun insere (e n l)
  (labels ((insere-aux (lista p)
             (cond ((null lista) (list e))
                   ((zerop p) (cons e lista))
                   (t (cons (car lista) (insere-aux (cdr lista) (1- p)))))))
    (insere-aux l n)
    )
)
```

Operadores e sucessores

A função ***operador*** substitui a casa do tabuleiro indicada pelas coordenadas linhaTabuleiro e colunaTabuleiro e devolve o tabuleiro com a peça adicional retirada das reservas se a casa estiver vazia e as coordenadas forem válidas.

```
(defun operador (linhaTabuleiro colunaTabuleiro peca no)
    (cond ((or (null peca) (null no) (null (casa-vaziap linhaTabuleiro
colunaTabuleiro (tabuleiro no)))) nil)
```

```
        (t (cria-no (cons (substituir linhaTabuleiro colunaTabuleiro peca
(tabuleiro no)) (cons (remover-peca peca (reserva no)) nil)) (1+ (no-profundidade
no)) no))
    )
)
```

A função ***sucessores*** constrói a lista de nós sucessores do nó aplicando a cada célula o operador com os índices correspondentes à célula.

```
(defun sucessores (no &optional (profundidade 9999) (pecas (reserva no))
(linhaTabuleiro 0) (colunaTabuleiro 0))
  (remove nil
          (cond
           ((or (null no) (null pecas) (>= (no-profundidade no) profundidade))
nil)
           ((= linhaTabuleiro 4) (sucessores no profundidade (cdr pecas) 0 0))
           ((= colunaTabuleiro 4) (sucessores no profundidade pecas (1+
linhaTabuleiro) 0))
           (t (cons (operador linhaTabuleiro colunaTabuleiro (car pecas) no)
(sucessores no profundidade pecas linhaTabuleiro (1+ colunaTabuleiro))))
           )
          )
)
```

Funções auxiliares de avaliação da eficiência

A função ***f-polinomial*** soma do polinómio criado recorrendo à equação ***B + B^2 + ... + B^L = T***

```
(defun f-polinomial (B L valor-T)
  (cond
   ((= 1 L) (- B valor-T))
   (T (+ (expt B L) (f-polinomial B (- L 1) valor-T)))
  )
)
```

A função ***ramificacao*** recorre ao método da bisseção para calcular o fator médio de ramificação do nó, recorrendo ao número de nós gerados ***T*** e à profundidade, ***L***.

```
(defun ramificacao (no valor-T &optional (valor-L (no-profundidade no)) (erro 0.1)
(bmin 1) (bmax 10e11))
  (let ((bmedio (/ (+ bmin bmax) 2)))
    (cond
     ((< (- bmax bmin) erro) (/ (+ bmax bmin) 2))
     ((< (f-polinomial bmedio valor-L valor-T) 0) (ramificacao no valor-T valor-L
erro bmedio bmax))
     (t (ramificacao no valor-T valor-L erro bmin bmedio))
```

```
        )
      )
  )
```

---

6. **Melhorias na aplicação**

- Utilização de closures para melhorar o desempenho dos algoritmos.
- Implementação de uma nova heurística melhorada.
