

Manual Técnico

Dots and Boxes Game

Inteligência Artificial

Índice

- [Introdução](#)
- [Ficheiro Projeto](#)
 - [Interface com o utilizador](#)
 - [Ler e Enviar ficheiros .dat](#)
- [Ficheiro Puzzle](#)
- [Ficheiro Procura](#)
 - [Gerar nós filhos](#)
 - [Algoritmos](#)
 - [DFS](#)
 - [BFS](#)
 - [Medidas de Desempenho](#)
- [Resultados 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 **Dots and Boxes** com a utilização de algoritmos e árvores de procura, tal tabuleiro deve ser fornecido pelo utilizador e indicando a quantidade de caixas pretendentes fechar a aplicação deverá concluir o tabuleiro com o número de caixas fechadas pedidas.

Este projeto foi produzido produzido e desenvolvido na aplicação **Visual Studio Code** com a utilização do interpretador de **Lisp CLisp**.

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

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

Ficheiro Projeto

Neste ficheiro podemos encontrar o carregamento dos outros ficheiros de código, a leitura do ficheiro **problemas.dat** e toda a interface com o utilizador.

Interface com o utilizador

Para a interface com o utilizador foram desenvolvidas várias funções de print e controlo da interface com a leitura de input por parte do utilizador.

- Todo o programa é inicializado com o input do utilizador (**start**) este dará início a todo o ciclo de opções que o utilizador tem e permitirá ao utilizador aceder a todas as opções do programa

```
(defun start())
```

```

(progn
  (startMessage)
  (let ((opt (read)))
    (if (or (not (numberp opt)) (> opt 2) (< opt 0)) (progn (format t "Por favor escolha
uma opcao possivel!~%" (start))
    (ecase opt
      ('1 (progn (printProblems) (start)))
      ('2 (chooseProblem))
      ('0 (progn (format t " Obrigado!")(quit)))
    )
  ...))

```

Nesta função podemos verificar o uso do **(read)** que é utilizada em todos métodos que pedem ao utilizador um input, este apresenta as opções de input na função **(startMessage)** 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 de novo o input.

- Após o utilizador escolher o problema que quer numa sequência de pedidos de input vai ser apresentado com a escolha de Algoritmo a utilizar, este terá as 3 opções de algoritmos pedido:

```

(defun chooseAlgorithm(board)
  (progn (chooseAlgorithmMessage)
    (let ((opt (read)))
      (cond ((not (numberp opt)) (progn (format t "Insira uma opcao valida!!")
(chooseAlgorithm)))
        ((or (> opt 3) (< opt 0)) (progn (format t "Insira uma opcao valida!!")
(chooseAlgorithm)))
        ((eq opt 0) (chooseProblem))
        (t (let* (
          (boxes (second board))
          (board (third board)))
          (ecase opt
            (1
              (let* ((maxDepth (chooseDepth board))
                (solution (list (getTime) (dfs (list (createNode board NIL boxes)) maxDepth)
                  (getTime) 'DFS maxDepth)))
                (progn (writeFinalResults solution) solution)
              )
            )
            (2
              (let
                ((solution (list (getTime) (bfs (list (createNode board NIL boxes)))
                  (getTime) 'BFS)))
                (progn (writeFinalResults solution) solution)
              )
            )
            (3
              (let* ((heuristic (chooseHeuristic board))
                (solution (list (getTime) (dfs (list (createNode board NIL boxes)) maxDepth)
                  (getTime) 'A* heuristic)))
                (progn (writeFinalResults solution) solution)
              )
            )
          ...))

```

- Este Método tem uma função diferente pois dependendo do input do utilizador vai apresentar e executar janelas diferentes, começa com a apresentação da mensagem como normal, porém:
 - Se o utilizador selecionar DFS irá pedir ao utilizador a profundidade máxima para executar o algoritmo, executar o algoritmo e logo de seguida apresentar os resultados.
 - Se o utilizador selecionar BFS 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.

- Como opção o utilizador pode ver todos os problemas lidos do ficheiro **problemas.dat** organizados e numerados, para isso usamos:

```
(defun printBoard(board &optional (stream t))
  (not (null (mapcar #'(lambda(l) (format stream "~%~t~t ~a" l)) board)))
  (format t "~%~%------~%"
    -----~%)
  )

(defun printProblems (&optional (i 1) (problems (getProblems)))
  (cond
    ((null problems))
    (T (let ((problem (car problems)))
        (format T "~% ~d- Tabuleiro ~d (~d caixas):~%" i (car problem) (cadr problem))
        (printBoard (caddr problem)))
        (printproblems (+ i 1) (cdr problems))
      )
    )
  )
```

(printProblems) permite dar print de todos os problemas recolhidos até que a lista esteja vazia, usamos a função **(let)** para guardar uma variável temporária que será um tabuleiro, damos print do número do problema, da letra designada ao problema, número de caixas designadas a fechar e o tabuleiro inicial com a ajuda da função **(printBoard)**, de seguida de forma recursiva dando mais 1 valor para o número do board e o resto da lista com **(cdr problems)**

Ler e Enviar ficheiros .dat

- Um dos objetivos do projeto era conseguir ler os problemas do ficheiro **problemas.dat** e no final da resolução dos problemas enviar a mesma para um ficheiro **resolucao.dat** as duas funções principais com esse objetivo são:

```
(defun getProblems ()
  (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 writeFinalResultsFile (solution)
  (let* ((startTime (first solution))
        (solutionNode (second solution))
        (endTime (third solution))
        (search (fourth solution))
        (with-open-file (str "../lisp/resultados.dat" :direction :output :if-exists :append
                          :if-does-not-exist :create)
          (writeFinalResults str solutionNode startTime endTime search (last solution))
        )
    ...)
```

Na exportação dos resultados finais de um problema recebemos a solução completa com todos os dados necessários e dividimos em várias variáveis para ser mais fácil de organizar no print final, de novo com o **(with-open-file)** criamos o ficheiro ou caso já esteja criado adicionamos a nova informação ao mesmo, de seguida executamos a função **(writeFinalResults)** que recebe todas as informações da solução assim como a stream para por a informação no ficheiro, desta maneira a função irá depois dar print da informação tanto no terminal para o utilizador ver como no ficheiro dessa maneira guardando os dados todos como visto abaixo.

```
(defun writeFinalResults (stream solutionNode startTime endTime search &optional depth)
  (progn
    (format stream "%~tRESULTADOS FINAIS DA RESOLUCAO DO TABULEIRO::~~%")
    (format stream "%~t -- Objetivo caixas fechadas: ~a " (countClosedBoxes(getSolutionNode
solutionNode)))
    (format stream "%~t -- Algoritmo: ~a " search)
    (format stream "%~t -- Inicio: ~a::~~a" (first startTime) (second startTime) (third
startTime))
    (format stream "%~t -- Fim: ~a::~~a" (first endTime) (second endTime) (third endTime))
    (format stream "%~t -- Numero de nos gerados: ~a" (+ (second solutionNode)(third
solutionNode)))
    (format stream "%~t -- Numero de nos expandidos: ~a" (second solutionNode))
    (format stream "%~t -- Penetrancia: ~F" (penetrance solutionNode))
    (format stream "%~t -- Fator de ramificacao media: ~F" (averageBranchingFator
solutionNode))
    (if (eq search 'DFS)
      (format stream "%~t -- Profundidade maxima: ~a" (car depth)))
    (format stream "%~t -- Comprimento da solucao: ~a" (length (car solutionNode)))
    (format stream "%~t -- Estado Inicial")
    (printFinalBoard (first (first solutionNode)) stream)
    (format stream "%~t -- Estado Final")
    (printFinalBoard (getSolutionNode solutionNode) stream)
    (format stream "%~::~~%"))

    (format T "%~tRESULTADOS FINAIS DA RESOLUCAO DO TABULEIRO::~~%")
    (format t "%~t -- Objetivo caixas fechadas: ~a " (countClosedBoxes(getSolutionNode
solutionNode)))
    (format t "%~t -- Algoritmo: ~a " search)
    (format T "%~t -- Inicio: ~a::~~a" (first startTime) (second startTime) (third startTime))
    (format t "%~t -- Fim: ~a::~~a" (first endTime) (second endTime) (third endTime))
    (format t "%~t -- Numero de nos gerados: ~a" (+ (second solutionNode)(third solutionNode)))
    (format t "%~t -- Numero de nos expandidos: ~a" (second solutionNode))
    (format t "%~t -- Penetrancia: ~F" (penetrance solutionNode))
    (format T "%~t -- Fator de ramificacao media: ~F" (averageBranchingFator solutionNode))
    (if (eq search 'DFS)
      (format t "%~t -- Profundidade maxima: ~a" (car depth)))
    (format t "%~t -- Comprimento da solucao: ~a" (length (car solutionNode)))
    (format t "%~::~~t -- Estado Inicial~%")
    (printFinalBoard (first (first solutionNode)))
    (format t "%~t -- Estado Final~%")
    (printFinalBoard (getSolutionNode solutionNode))
    (format t "%~::~~%")
    (format t "Obrigado por jogar!~%")
    (format t "Com os melhores cumprimentos, Luis Rocha e Samuel Ribeiro ~%::~~%")
    (quit)
  )
)
```

Ficheiro Puzzle

No ficheiro Puzzle teremos todos os métodos relacionados com as operações e gestão do tabuleiro, este que é essencial para todo o processo dos algoritmos para colocar novos arcos, verificar caixas fechadas etc...

- Nas funções mais básicas temos algo como retornar a lista dos arcos horizontais e arcos verticais, que apesar de básicos facilitam bastante nas operações avançadas.

```
(defun getHorizontalArcs (board)
  (car board)
)

(defun getVerticalArcs (board)
  (car (cdr board))
)
```

- Para conseguirmos determinar que valor temos numa certa posição do tabuleiro criamos a função **getArcOnPosition** que recebe as coordenadas e a lista horizontal/vertical dos arcos e devolve apenas o valor da posição (0 ou 1)

```
(defun getArcOnPosition (x y board)
  (if (and (/= x 0) (/= y 0))
      (nth (- y 1) (nth (- x 1) board))
  )
)
```

- Para colocar um arco numa certa posição foi criada a função **arcOnPosition** e a função auxiliar **replaceElem**

```
(defun replaceElem(list x &optional (y 1))
  "Função que recebe um índice, uma lista e valor y e deverá substituir o elemento nessa
  posição pelo valor y, que deve ser definido com o valor de default a 1"
  (cond
    ((= (- x 1) 0) (cons y (cdr list)))
    (T (cons (car list) (replaceElem (cdr list) (- x 1) y)))
  )
)

(defun arcOnPosition (x y list)
  "Insere um arco (representado pelo valor 1) numa lista que representa o conjunto de
  arcos horizontais ou verticais de um tabuleiro."
  (cond
    ((= x 1)
     (cons (replaceElem (nth (- x 1) list) y) (cdr list))
    )
    (T (cons (car list) (arcOnPosition (- x 1) y (cdr list))))
  )
)
```

Na função **arcOnPosition** verificamos vamos avançando na lista até que $x=1$ (a lista para trocar o y seja encontrada) de forma recursiva, caso cheguemos a essa lista chamamos a função **replaceElem** que de uma forma parecida vai encontrar a posição a alterar e vai lá colocar o arco (valor 1) no sítio pretendido.

- O aspeto principal para o final do jogo é a quantidade de caixas que estão fechadas, dessa maneira temos a função **checkClosedBox** e a função **countClosedBoxes** que conta o número total de caixas fechadas no tabuleiro.

```
(defun checkClosedBox (x y board)
  (if (or (or (< x 1) (< y 1)) (>= x (length (getHorizontalArcs board))))
      NIL
      (and
        "A" (= (getArcOnPosition x y (getHorizontalArcs board)) 1)
        "B" (= (getArcOnPosition y x (getVerticalArcs board)) 1)
      )
  )
)
```

```

        "C" (= (getArcOnPosition (+ y 1) x (getVerticalArcs board)) 1)
        "D" (= (getArcOnPosition (+ x 1) y (getHorizontalArcs board)) 1)
    )
)
)

(defun countClosedBoxes (board &optional (row 1) (col 1))
(cond
  ((>= col (length (getHorizontalArcs board))) (countClosedBoxes board (1+ row)))
  ((>= row (length (getVerticalArcs board))) 0)
  (T
    (+
      (if (checkClosedBox row col board) 1 0)

      (countClosedBoxes board row (1+ col))
    )
  )
)
)
)

```

A função **checkChosedBox** irá apenas receber as coordenadas de um arco horizontal que vai funcionar de referencia, se pretencer ao tabuleiro e não estiver na ultima fila vai avaliar o arco horizontal diretamente em baixo, o arco vertical a sua esquerda e a sua direita, caso todos esses tiverem o valor de 1 (tiverem arco preenchido) a função vai retornar **T(Verdade)** caso contrario **NIL(Falso)**.

A função **countClosedBoxes** verifica o tabuleiro todo utilizando a **length** das listas dos arcos horizontais e verticais e para cada arco horizontal vai executar a função **checkChosedBox** e somar todos os casos em que há uma caixa fechada.

- Para colocar arcos Horizontais e Verticais para a criação de filhos para os algoritmos de forma mais eficiente foram criadas as funções **horizontalArc** e **verticalArc**

```

(defun horizontalArc (x y board)
(if
  (and
    (<= x (length (getHorizontalArcs board)))
    (<= y (length (car (getHorizontalArcs board)))))
  )
  (if (/= (getArcOnPosition x y (getHorizontalArcs board)) 1)
    (list (arcOnPosition x y (getHorizontalArcs board))
      (getverticalarcs board))
    0
  )
)
)

(defun verticalArc (x y board)
(if
  (and
    (<= y (length (getVerticalArcs board)))
    (<= x (length (car (getVerticalArcs board)))))
  )
  (if (/= (getArcOnPosition y x (getVerticalArcs board)) 1)
    (list (gethorizontalarcs board)
      (arcOnPosition y x (getVerticalArcs board)))
    0
  )
)
)

```

As duas funções são bastante parecidas apenas com a diferença que uma coloca um arco vertical e a outra um arco horizontal.

A função que recebe dois índices e o tabuleiro e coloca um arco nessa posição. A função deverá retornar NIL caso já exista um arco colocado nessa posição ou caso a posição indicada seja fora dos limites do tabuleiro."

Ficheiro Procura

O ficheiro procura contém todos os métodos para a execução correta dos algoritmos. Este tem como formato inicial o nó que é a lista de:

- tabuleiro atual, pai do nó, objetivo de caixas a fechar, profundidade do nó e Heurística (quando necessário)

```
(defun createNode (board parent boxes &optional (d 0) (h 0))
  (list board parent boxes d h)
)
```

Gerar Nós Filhos

Para corretamente gerar todas as opções dos nós filhos separamos as duas opções (arcos verticais e horizontais) em duas funções bastante parecidas apenas com a diferença das funções de colocação de arcos diferentes.

```
(defun generateChildrenHorizontal (node &optional (x 1) (y 1))
  (cond
    ( (> y (length (car (getHorizontalArcs(nodeGetBoard node)))))
      (generateChildrenHorizontal node (1+ x))
    )
    ( (> x (length (getHorizontalArcs(nodeGetBoard node)))) '())
    ( (/= 0 (getarconposition x y (gethorizontalarcs(nodegetboard node))))
      (generateChildrenHorizontal node x (1+ y)))
  )
  (T
    (cons
      (createnode
        (horizontalArc x y (nodeGetBoard node))
        node
        (nodegetboxes node)
        (1+ (nodeGetDepth node))
      )
      (generateChildrenHorizontal node x (1+ y))
    )
  )
)

(defun generateChildrenVertical (node &optional (x 1) (y 1))
  (cond
    ( (> y (length (getVerticalArcs(nodeGetBoard node)))))
      (generateChildrenVertical node (1+ x))
    )
    ( (> x (length (car (getVerticalArcs(nodeGetBoard node))))) '())
    ( (/= 0 (getarconposition y x (getverticalarcs(nodegetboard node))))
      (generateChildrenVertical
node x (1+ y)))
  )
  (T
    (cons
      (createnode
        (VerticalArc x y (nodeGetBoard node))
        node

```

```

        (nodegetboxes node)
        (1+ (nodeGetDepth node))
    )
    (generateChildrenVertical node x (1+ y))
  )
)
)

```

Nestes métodos recebemos como parâmetros o nó original e as coordenadas que vão ser iniciadas em 1, verificamos se y está dentro dos valores aceitáveis (1 a tamanho da sublista dos arcos), caso contrário adicionamos +1 ao x para passar para o próximo, verificamos se já existe um arco na posição se sim passamos para o próximo, caso passe a todas as verificações criamos um novo nó com as alterações e inserindo como pai o nó original, +1 de profundidade e chamamos recursivamente a função com o novo nó.

Algoritmos

DFS

Para o algoritmo Depth-First-Search definimos a função dfs. Se não existirem elementos no open o algoritmo acaba e retorna **NIL**.

De seguida verificamos se já ultrapassamos a profundidade desejada, se for o caso removemos esse nó da lista de abertos e inserimos no final da lista de fechados. Caso nenhuma das condições sejam verificadas declaramos as variáveis utilizando o let* para permitir declarar variáveis sequencialmente, *currNode* que é o nó atual, *children* que são os filhos do nó atual e *solution* que só é preenchida caso algum dos nós sucessores seja uma solução, prosseguimos a analisar se existe solução, se sim retornamos a solução no formato desejado, caso contrário verificamos se o nó tem sucessores se tiver chamamos a função recursivamente adicionamos os filhos no início da lista de abertos ao contrário do *BFS*.

```

(defun dfs(opened maxDepth &optional (closed '()))
  (cond
    ((not (car opened)) nil)
    ((> (nodegetdepth (car opened)) maxDepth)
      (dfs (cdr opened) maxDepth (append closed (list (car opened)))))
    (t
      (let*
        (
          (chosenNode (car opened))
          (children (generateChildrenlist chosenNode))
          (solution (firstsolution children))
        )
        (if (null solution)
          (if (car children)
            (dfs (append children (list (cdr opened))) maxdepth (append closed (list
chosenNode))) )
            (list (pathtoroot chosenNode) (length opened) (length closed))
          )
          (list (pathtoroot solution) (length opened) (length closed))
        )
      )
    )
  )
)

```

BFS

Para o algoritmo Breadth-First-Search definimos a função bfs. Se não existirem elementos no open o algoritmo acaba e retorna **NIL** caso contrário verifica se o primeiro elemento da lista de open for uma solução retorna a lista do caminho, o número de nós fechados

e abertos, se não for o caso declaramos as variáveis utilizando o `let*` para permitir declarar variáveis sequencialmente, *currNode* que é o nó atual, *children* que são os filhos do nó atual e *solution* que só é preenchida caso algum dos nós sucessores seja uma solução, prosseguimos a analisar se existe solução, se sim retornamos a solução no formato desejado, caso contrário verificamos se o nó tem sucessores se tiver chamamos a função recursivamente adicionando o nó visitado ao fim da lista de nós fechados e adicionando os filhos no final da lista de abertos, caso o nó não tenha sucessores devolvemos esse nó como solução.

```
(defun bfs(opened &optional (closed '()))
  (if(car opened)
    (if (< (countclosedboxes (nodegetboard (car opened))) (nodegetboxes (car opened)))
      (let*
        (
          (currNode (car opened))
          (children (removenil (generatechildrenlist currnode)))
          (solution (firstsolution children))
        )
      (if (null solution)
        (if (car children)
          (bfs (append (cdr opened) children) (append closed (list currnode)))
          (list (pathtoroot currnode) (length opened) (length closed))
        )
        (list (pathtoroot solution) (length opened) (length closed))
      )
    )
    (list (pathtoroot (car opened)) (length opened) (length closed))
  )
)
```

Medidas de desempenho

Como medidas para calcular o desempenho dos algoritmos temos a Penetrância e o fator de ramificação média, estas estatísticas estarão disponíveis sempre que um algoritmo acaba e é apresentado ao utilizador os resultados.

```
(defun penetrance (solution)
  (coerce (/ (getsolutionlength solution) (+ (second solution)(third solution))) 'float)
)
```

A penetrância de um algoritmo é calculada pela divisão entre a profundidade do objetivo e o número total de nós criados(soma entre nós aberto e nós fechados), no código acima está representado esse calculo, recebemos a solução final e calculamos a penetrância e no final usamos a função **coerce** para transformar o resultado em fração.

```
(defun averageBranchingFator (solution &optional (depth (getsolutionlength solution))
  (generatedNodes (numbergeneratednodes solution))
  (tolerance 0.1) (min 0) (max (numbergeneratednodes solution)))
  (let ((average (/ (+ min max) 2)))
    (cond
      ((< (- max min) tolerance) (/ (+ max min) 2))
      ((< (auxiliarbranching average depth generatednodes) 0)
        (averagebranchingfator solution depth generatednodes tolerance average max))
      (t (averagebranchingfator solution depth generatednodes tolerance min average))
    ...))

  (defun auxiliarBranching (average depth generatedNodes)
    (cond
      ((= 1 depth) (- average generatednodes))
      (t (+ (expt average depth)
```

```

    )
    )
    (auxiliarBranching average (- depth 1) generatednodes)))
  )
)

```

Para o calculo do fator de ramificação médio foi necessário a utilização de duas funções, uma principal e uma auxiliar.

A função principal vai ter como argumentos:

- Profundidade
- Número de nós gerados
- Margem de Tolerância
- Mínimo (valor inicial 0)
- Máximo (valor inicial número de nós gerados)

Primeiramente fazer a média entre o mínimo e o máximo, calcular o número de nós gerados utilizando a média como fator (na função auxiliar), calcular a diferença entre o número de nós gerados através da média e o número de nós gerado original, caso a diferença seja menor que a margem de tolerância devolve a média atual caso contrário utilizamos a média como valor Máximo e executamos recursivamente a função.

Resultados Finais

Na tabela abaixo 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.

Tabuleiro	Algoritmo	Objetivo Caixas	Início	Fim	Nós Gerados	Nós Expandidos	Penetrância	Fator de ramificação	Profundidade da solução
A	DFS	3	2:40:35	2:40:35	18	13	0.3888889	1.2304687	7
A	BFS	3	2:41:57	2:41:57	77	72	0.038961038	3.8725586	3
A	A*	3	-	-	-	-	-	-	-
B	DFS	7	2:45:9	2:45:9	1	1	2.0	0.59375	2
B	BFS	7	2:46:6	2:46:6	1	1	2.0	0.59375	2
B	A*	7	-	-	-	-	-	-	-
C	DFS	10	2:47:40	2:47:40	20	8	0.7	1.0546875	14
C	BFS	10	-	-	-	-	-	-	-
C	A*	10	-	-	-	-	-	-	-
D	DFS	10	2:49:26	2:49:26	52	15	0.75	0.9902344	39
D	BFS	10	-	-	-	-	-	-	-
D	A*	10	-	-	-	-	-	-	-
E	DFS	20	2:51:46	2:51:46	43	14	0.7209302	1.0498047	31
E	BFS	20	-	-	-	-	-	-	-
E	A*	20	-	-	-	-	-	-	-
F	DFS	35	2:52:50	2:52:51	111	19	0.8468468	1.0026855	94
F	BFS	35	-	-	-	-	-	-	-
F	A*	35	-	-	-	-	-	-	-