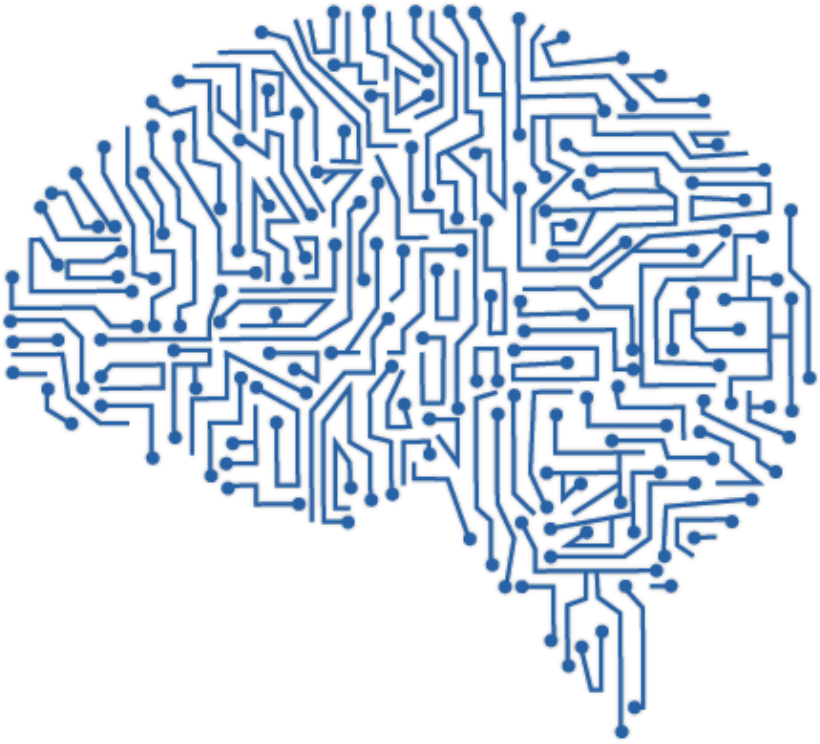
Práctica 2

Inteligencia Artificial



Alfonso Bonilla Trueba – David García Fernández
PAREJA 4 | GRUPO 2313 | EPS-UAM

Ejercicio 1 – Test para determinar si se ha alcanzado el objetivo

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    f-goal-test-galaxy (state planets-destination)
;;;    Comprueba si se ha alcanzado el objetivo.
;;;
;;;    INPUT:
;;;           state: Estado que se quiere comprobar si es objetivo
;;;           planets-destination: Lista que contiene los planetas destino
;;;    OUTPUT:
;;;           T si el estado es el objetivo o NIL en caso contrario
;;;
(defun f-goal-test-galaxy (state planets-destination)
       (when (member state planets-destination)
                     t))

;;; Casos de Prueba
;;; (f-goal-test-galaxy 'Sirtis *planets-destination*) ;-> T
;;; (f-goal-test-galaxy 'Avalon *planets-destination*) ;-> NIL
;;; (f-goal-test-galaxy 'Urano *planets-destination*) ;-> NIL
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Ejercicio 2- Evaluación del valor de la heurística

PSEUDOCODIGO:

f-h-galaxy (state sensors)
    if state = null or sensores = null:
        NIL
    else:
        Se devuelve la heurística que hay para el estado dado.

CODIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    f-h-galaxy (state sensors)
;;;    Calcula la el valor de la heuristica en el estado que recibe.
;;;
;;;    INPUT:
;;;           state: Estado del que calcular el valor heuristico
;;;           sensors: Lista de pares planeta-valor heuristico
;;;    OUTPUT:
;;;           Valor de la heuristica o NIL si no se ha podido calcular
;;;
(defun f-h-galaxy (state sensors)
  (when sensors
    (if (eq state (first (first sensors)))
      (second (first sensors))
      (f-h-galaxy state (rest sensors)))))

;;; Casos de Prueba
;;; (f-h-galaxy 'Sirtis *sensors*) ;-> 0    ;Caso Tipico
;;; (f-h-galaxy 'Avalon *sensors*) ;-> 5    ;Caso Tipico
;;; (f-h-galaxy 'Tierra *sensors*) ;-> NIL ;Caso especial
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Ejercicio 3 - Operadores navigate-worm-hole y navigate-white-hole

PSEUDOCODIGO:

navigate-white-hole (state white-holes)
  if state = null or white- holes = null:
    NIL
  else:
    Busca en la lista de white-holes si existe una entrada para el state, si existe crea una acción en la que pone como origen state, como nodo destino el otro estado que aparezca en la entrada de la lista que hemos sacado y asignando un coste a la acción de ir de un estado a otro.

CODIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    navigate-worm-hole (state worm-holes)
;;;    Busca las acciones posibles desde el estado actual a traves
;;; de agujeros de gusano.
;;;
;;;    INPUT:
;;;           state: Estado en el que se esta
;;;           white-holes: Lista que contiene los caminos entre agujeros
;;;                de gusano y el coste de los mismos
;;;    OUTPUT:
;;;           Acciones posibles desde el estado actual o NIL si no hay
;;;   acciones posibles
;;;
(defun navigate-worm-hole (state worm-holes)
  (when worm-holes
    (let ((navigate (navigate-worm-hole state (rest worm-holes))))
      (if (eq state (first (first  worm-holes)))
        (append (list
                  (make-action
                    :name 'navigate-worm-hole
                    :origin state
                    :final (second (first worm-holes))
                    :cost (third (first worm-holes))))
                navigate)
        navigate))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    navigate-white-hole (state white-holes)
;;;    Busca las acciones posibles desde el estado actual a traves de agujeros
;;; blancos
;;;
;;;    INPUT:
;;;           state: Estado en el que se esta
;;;           white-holes: Lista que contiene los caminos entre agujeros blancos y el
;;;                coste de los mismos
;;;    OUTPUT:
;;;           Acciones posibles desde el estado actual o NIL si no hay acciones
;;;   posibles
;;;
(defun navigate-white-hole (state white-holes)
 (when white-holes
    (let ((navigate (navigate-white-hole state (rest white-holes))))
      (if (eq state (first (first white-holes)))
        (append (list
                  (make-action
                    :name 'navigate-white-hole
                    :origin state
                    :final (second (first white-holes))
```

```
                    :cost (third (first white-holes))))
                navigate)
        navigate))))


;;; Casos de Prueba
;;; (navigate-worm-hole 'Katril *worm-holes*) ;->
;(#S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN KATRIL :FINAL DAVION :COST 1)
; #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN KATRIL :FINAL MALLORY :COST 5)
; #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN KATRIL :FINAL SIRTIS :COST 10))
;;; (navigate-white-hole 'Urano *white-holes*) ;-> NIL
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Ejercicio 4 - Definir estrategia para la búsqueda A*

```
(setf *A-star*
  (make-strategy
    :name 'a-star
    :node-compare-p 'node-star))

(defun node-star (node-1 node-2)
    (<= (node-f node-1)
        (node-f node-2)))

(setf *uniform-cost*
  (make-strategy
    :name 'uniform-cost
    :node-compare-p 'node-g-<=))

(defun node-g-<= (node-1 node-2)
  (<= (node-g node-1)
      (node-g node-2)))
```

Ejercicio 5 - Representación LISP del problema

```
(setf *galaxy-M35*
 (make-problem
   :states *planets*
   :initial-state *planet-origin*
   :f-goal-test #'(lambda (state)
                    (f-goal-test-galaxy state *planets-destination*))
   :f-h #'(lambda (state)
              (f-h-galaxy state *sensors*))
   :operators (list #'(lambda (state)
                        (navigate-worm-hole state *worm-holes*))
                    #'(lambda (state)
                        (navigate-white-hole state *white-holes*))
                    )))
```

Ejercicio 6 - Expandir nodo

Busca en problem el nodo que le hemos pasado, si lo encuentra saca los elementos de las listas worm-holes, white-holes y sensors, con estos elementos se extraen los costes y las heurísticas y con ellos se asignan a una lista de estructuras nodes que se devolverá, donde los datos que se rellena son el padre (que es node), la acción (que consiste en llamar a la función del ejercicio 3), g que sera la suma del coste de desplazarse de un nodo a otro más la “g” que arrastre nodo, “h” que sera la heurística que hemos sacado de sensores y “f”

que es la suma de los datos anteriores. Este proceso se repetirá para cada elemento que hayamos extraído de worm-holes y white-holes

CODIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    (defun expand-node (node problem)
;;;    Expande el nodo segun el problema a tratar, devolviendo todos los nodos
;;; a los que se puede ir.
;;;
;;;    INPUT:
;;;           node: Nodo a expandir
;;;           problem: Problema bajo estudio segun el cual realizar la expansion
;;;    OUTPUT:
;;;           Nodos a los que se puede ir desde el nodo expandido o NIL
;;;
(defun expand-node (node problem)
  (mapcar #'(lambda (accion)
              (let ((ge     (+ (node-g node) (action-cost accion)))
                    (hache  (funcall (problem-f-h problem) (action-final accion))))
                (make-node
                 :state (action-final accion)    ; state label
                 :parent node                    ; parent node
                 :action accion                  ; action that generated the current node
from its parent
                 :depth (+ (if (node-depth node)
                               (node-depth node)
                             0)
                           1)  ; depth in the search tree
                 :g ge                           ; cost of the path from the initial state
of this node
                 :h hache                        ; value of the heuristic
                 :f (+ ge hache))                ; g + h
                )
              )
    (expand-aux node (problem-operators problem))))

(defun expand-aux (node problem-operators)
  (let ((acciones (funcall (first problem-operators) (node-state node))))
    (if (rest problem-operators)
        (append acciones (expand-aux node (rest problem-operators)))
      acciones)))

;;; Casos de Prueba
;;; (setf node-00
;;;   (make-node :state 'Proserpina :depth 12 :g 10 :f 20))
;;;
;;; (print
;;;   (setf lst-nodes-00
;;;     (expand-node node-00 *galaxy-M35*)))
```

Ejercicio 7 - Gestión de nodos

Inserta de forma ordenada la lista de nodos nodes en la lista de lst-nodes mediante la estrategia strategy. La forma de hacerlo será mediante una función (sort) que comprobará que dados dos nodos si la estrategia

devuelve NIL o T y dependiendo del resultado los ordena de una forma o de otra.

CODIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    defun insert-nodes-strategy (nodes lst-nodes strategy)
;;;    Inserta una lista de nodos en otra lista de acuerdo a una estrategia.
;;;
;;;    INPUT:
;;;           nodes: Lista de nodos a insertar
;;;           lst-nodes: Lista de nodos en la que insertar
;;;           strategy: Estrategia que seguir para la insercion
;;;    OUTPUT:
;;;           Lista resultado de la insercion
;;;
(defun insert-nodes-strategy (nodes lst-nodes strategy)
  (sort (append nodes lst-nodes) (strategy-node-compare-p strategy)))

;;; Casps de Prueba
;;;
;;; (setf node-00
;;;   (make-node :state 'Proserpina :depth 12 :g 10 :f 20))
;;;
;;; (setf node-01
;;;  (make-node :state 'Avalon :depth 0 :g 0 :f 0))
;;;
;;; (setf node-02
;;;  (make-node :state 'Kentares :depth 2 :g 50 :f 50))
;;;
;;; ------------------------------------------------------------
;;; (print(insert-nodes-strategy (list node-00 node-01 node-02)
;;;   lst-nodes-00
;;;   *uniform-cost*))
```

Ejercicio 8 - Función de búsqueda

CODIGO:

El problema que hemos resuelto en el ejercicio 8, consiste en trazar un camino del nodo origen hasta el nodo destino, si dicho camino existe, y empleando las estructuras definidas y la funcionalidad implementada anteriormente. En nuestro caso hemos descompuesto el problema en tres funciones:

- graph-search: la que se pide en el enunciado y que en nuestro caso se encarga de inicializar la lista de abiertos y cerrados.
- `graph-search-algorithm: que se encarga de implementar el algoritmo descrito en el enunciado de la práctica.`
- `esta-en-cerrado: una función auxiliar de graph-search-algorithm que se encarga de recorrer recursivamente la lista de cerrados para ver si ese nodo ya ha sido abierto con un peso menor`

CODIGO:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    graph-search (problem strategy)
;;;    Realiza busqueda en grafo
;;;
;;;    INPUT:
;;;           problem: Problema sobre el que realizar la busqueda
;;;           strategy: Estrategia que seguir para la busqueda
;;;    OUTPUT:
;;;           Resulta de la busqueda realizada por la funcion graph-search-aux
;;;

(defun graph-search (problem strategy)
  (let* ((nombre (problem-initial-state problem))
         (heu (funcall (problem-f-h problem) nombre)))
    (graph-search-algorithm problem
                            strategy
                            (list (make-node
                                   :state nombre
                                   :parent nil
                                   :action nil
                                   :depth 0
                                   :g 0
                                   :h heu
                                   :f heu))
                            (list nil))))

(defun esta-en-cerrado (nodo closed-nodes)
  (if (first closed-nodes)
    (let ((nombre-nodo (node-state nodo))
          (nombre-cerrado (node-state (first closed-nodes)))
          (nodog (node-g nodo))
          (cerradog (node-g (first closed-nodes))))
      (if (eq nombre-nodo nombre-cerrado)
        (when (< nodog cerradog)
          (and t (esta-en-cerrado nodo (rest closed-nodes))))
        (esta-en-cerrado nodo (rest closed-nodes))))
    nodo))


(defun graph-search-algorithm (problem strategy open-nodes closed-nodes)
  (let  ((nodo (first open-nodes)))
    (when nodo
      (let ((nodo-no-esta-cerrado (esta-en-cerrado nodo closed-nodes))
            (nuevos-abiertos (rest open-nodes)))
        (if (funcall (problem-f-goal-test problem) (node-state nodo))
            nodo
          (if nodo-no-esta-cerrado
              (graph-search-algorithm problem
                                      strategy
                                      (insert-nodes-strategy (expand-node nodo problem)
                                                             nuevos-abiertos
                                                             strategy)
                                      (cons nodo closed-nodes))
            (graph-search-algorithm problem
                                    strategy
                                    nuevos-abiertos
                                    closed-nodes)))))))
```

```
;;; Casos de Prueba
;;; (graph-search *galaxy-M35* *A-star*)
;;;
;;; #S(NODE :STATE SIRTIS
;;; :PARENT
;;; #S(NODE :STATE ...
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

Ejercicio 9 - Búsqueda A*

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    a-star-search (problem)
;;;    Realiza la busqueda A* para el problema dado
;;;
;;;    INPUT:
;;;           problem: Problema sobre el que realizar la busqueda
;;;    OUTPUT:
;;;           Resultado de la busqueda o NIL si no se ha podido realizar
;;;
(defun a-star-search (problem)
  (graph-search problem *a-star* ))

;;; Casos de Prueba
;;; (a-star-search *galaxy-M35*);->
;
; #S(NODE :STATE SIRTIS
; :PARENT
; #S(NODE :STATE ...
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Ejercicio 10 - Ver Camino

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    defun tree-path (node)
;;;    Llama a la funcion que muestra el camino seguido para llegar a un nodo.
;;;
;;;    INPUT:
;;;           node: Nodo al que se ha llegado
;;;    OUTPUT:
;;;           Lista con los nombres de los planetas por los que se ha pasado para
;;;         llegar al nodo o NIL.
;;;
(defun tree-path (node)
  (reverse (tree-path-aux node)))

(defun tree-path-aux (node)
  (unless (null node)
    (cons (node-state node)
          (tree-path-aux (node-parent node)))))
;;; Casos de Prueba
;;;
;;; (tree-path nil) ; -> NIL
;;; (tree-path #S(NODE :STATE MALLORY ...)) ; -> (KENTARES PROSERPINA MALLORY
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Ejercicio 11 - Secuencia de acciones de un camino

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    defun action-sequence (node)
;;;    Llama a la funcion que muestra la secuencia de acciones para llefar a un nodo.
;;;
;;;    INPUT:
```

```
;;;           node: Nodo al que se ha llegado
;;;    OUTPUT:
;;;           Lista con las acciones que se han realizado para llegar al  nodo o NIL.
;;;
(defun action-sequence (node)
  (reverse (action-sequence-aux node)))

(defun action-sequence-aux (node)
  (unless (null (node-action node))
    (cons (node-action node)
          (action-sequence-aux (node-parent node)))))

;;; Casos de Prueba
;;;
;;; (action-sequence (a-star-search *galaxy-M35*))
;;; (#S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN KENTARES :FINAL AVALON :COST 3)...)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Ejercicio 12 - Otras estrategias de búsqueda

```
(setf *depth-first*
 (make-strategy
    :name 'depth-first
    :node-compare-p 'depth-first-node-compare-p))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    depth-first-node-compare-p (node-1 node-2)
;;;    Realiza una busqueda en profundidad
;;;
;;;    INPUT:
;;;           node-1: Nodo a comparar
;;;           node-2: Nodo a comparar
;;;    OUTPUT:
;;;           Lista con los nombres de los planetas de los nodos expandidos hasta
;;;         encontrar la solucion.
;;;
(defun depth-first-node-compare-p (node-1 node-2)
  (when node-1
    (when node-2
      (>
        (node-depth node-1)
        (node-depth node-2)))))

;;; Casos de Prueba
;;; (tree-path (graph-search *galaxy-M35* *depth-first*))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(setf *breadth-first*
  (make-strategy
   :name 'breadth-first
   :node-compare-p 'breadth-first-node-compare-p))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    breadth-first-node-compare-p (node-1 node-2)
;;;    Realiza una busqueda en anchura
;;;
;;;    INPUT:
;;;           node-1: Nodo a comparar
;;;           node-2: Nodo a comparar
;;;    OUTPUT:
```

```
;;;           Lista con los nombres de los planetas de los nodos que recorre hasta
;;;         encontrar la solucion.
;;;
(defun breadth-first-node-compare-p (node-1 node-2)
  (when node-1
    (when node-2
      (<
        (node-depth node-1)
        (node-depth node-2)))))

;;; Casos de Prueba
;;; (tree-path (graph-search *galaxy-M35* *breadth-first*))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Cuestiones:

1. Porque añade una capa de abstracción que permite cambiar el programa sin alterar su estructura básica.

1.1. Se pueden probar diferentes conjuntos de datos y algoritmos sin necesidad de cambiar la mayor parte del código ya escrito (se cambian valores definidos en estructuras).

1.2 Porque permiten alterar la funcionalidad de dichas funciones sin necesidad de alterar el resto del programa. Esto es posible gracias a que se ocultan los procedimientos en estructuras y funciones que cuyo input y output está definido y es independiente del resto del proceso.

2. La eficiencia de un programa depende de para que se quiera emplear dicho programa. Si el objetivo es solo trazar la ruta del origen al destino se debería liberar toda la memoria ocupada en otras rutas innecesarias, aunque se debería mantener hasta que se sepa que esas rutas no son válidas, en nuestro caso, hasta que se encuentra el nodo destino.

3. 27,252 cons cells, 706,528 other bytes, 5,424 static bytes
   25,862 cons cells, 693,432 other bytes, 0 static bytes

4. cpu time (total) 16 msec user
   cpu time (total) 47 msec user

5. La navegación de cualquier forma, se realiza desde la función expandir nodo (ejercicio 6), ahí se podría pasar como parámetro una variable que permita limitar el número de expansiones dentro de la función. Otra opción es limitar el número de expansiones para evitar la navegación, esto se podría realizar en la función search-graph (ejercicio 8)