Inteligencia Artificial

Práctica 2

Pareja 02

Óscar Gómez Borzdynski, José Ignacio Gómez García
23-3-2018

Ejercicio 1:

- **PSEUDOCÓDIGO**

Entrada: Un estado y una lista de heurísticas
Salida: Heurística desde el nodo estado hasta la meta.
Función general:

A = buscar-tupla-primer-elemento (estado lista-heurísticas)
Segundo-elemento (A)

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN: Exercise 1 -- Evaluation of the heuristic
;;
;; Returns the value of the heuristics for a given state
;;
;;  Input:
;;    state: the current state (vis. the planet we are on)
;;    sensors: a sensor list, that is a list of pairs
;;              (state cost)
;;           where the first element is the name of a state and the second
;;           a number estimating the cost to reach the goal
;;
;;  Returns:
;;    The cost (a number) or NIL if the state is not in the sensor list
;;
(defun f-h-galaxy (state sensors)
  (cadr (assoc state sensors)))

(f-h-galaxy 'Sirtis *sensors*) ;-> 0
(f-h-galaxy 'Avalon *sensors*) ;-> 15
(f-h-galaxy 'Earth  *sensors*) ;-> NIL
```

- **COMENTARIOS**

En este caso, para buscar la tupla usamos assoc que, en caso de que el planeta no se encuentre, devolverá NIL. Al hacer cadr obtenemos el segundo elemento de la tupla. En caso de que la tupla sea NIL significará que no existe heurística, devolviendo NIL.

Ejercicio 2:

- **PSEUDOCÓDIGO**

Entrada: Un estado, una lista de enlaces (y una lista de planetas prohibidos)
Salida: Lista con todas las acciones posibles desde el nodo actual
Función general:

```
For terna in enlaces:
    If (origen terna) == estado AND (destino terna) not in planetas prohibidos:
        Make-action (name origin destination cost)
Return all actions
```

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN: Exercise 2 -- Navigation operators
;;
;; Returns a list of all posible actions from a state
;;
;;  Input:
;;    state: the current state (vis. the planet we are on)
;;    [white/worm]-holes: Links in our galaxy of a certain type
;;    (planets-forbidden): Only if worm holes. Planets that are not allowed to be visited
;;                    allowed to be visited via worm-holes
;;
;;  Returns:
;;    A list with all possible actions from the original state
;;    avoiding the links that go to a forbidden planet.
;;

;; Function that evaluates a link
(defun check-hole (hole state forbidden name)
  (let ((origin (first hole))
        (destination (second hole))
        (cost (third hole)))
    (when (and (eql origin state) ;;If the origin of the link is the state we are on
               (null (member destination forbidden))) ;; And the destination is not forbidden
      (list (make-action :name name ;; Create an action within a list
                         :origin state
                         :final destination
                         :cost cost)))))

;; Generical function for evaluating links
(defun navigate (state holes planets-forbidden name)
  (mapcan #'(lambda (x) (check-hole x
                                    state
                                    planets-forbidden
                                    name))
    holes))

;; Function for evaluating white holes
(defun navigate-white-hole (state white-holes)
  (navigate state
            white-holes
            NIL
            'NAVIGATE-WHITE-HOLE))

;; Function for evaluating worm holes
(defun navigate-worm-hole (state worm-holes planets-forbidden)
  (navigate state
            worm-holes
            planets-forbidden
            'NAVIGATE-WORM-HOLE))
```

(navigate-worm-hole 'Mallory *worm-holes* *planets-forbidden*) **;->**
;;;(#S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN MALLORY :FINAL KATRIL :COST 5)
;;; #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN MALLORY :FINAL PROSERPINA :COST 11))

(navigate-worm-hole 'Mallory *worm-holes* NIL) **;->**
;;;(#S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN MALLORY :FINAL AVALON :COST 9)
;;; #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN MALLORY :FINAL KATRIL :COST 5)
;;; #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN MALLORY :FINAL PROSERPINA :COST 11))

(navigate-white-hole 'Kentares *white-holes*) **;->**
;;;(#S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN KENTARES :FINAL AVALON :COST 3)
;;; #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN KENTARES :FINAL KATRIL :COST 10)
;;; #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN KENTARES :FINAL PROSERPINA :COST 7))

(navigate-worm-hole 'Uranus *worm-holes* *planets-forbidden*) **;-> NIL**

- **COMENTARIOS**

En este caso, hemos decidido crear una función auxiliar para analizar un enlace. Recibirá el nombre del operador que ha generado la acción, el estado en el que nos encontramos, la lista de planetas prohibidos y la terna a analizar.
Con estos datos podemos asegurarnos de que el enlace es operativo desde el estado actual.
Para crear las funciones que operan sobre agujeros de gusano y agujeros blancos, hemos creado una función que realiza ambas acciones y será llamada desde la función externa.

Ejercicio 3:

- **PSEUDOCÓDIGO**

Entrada: Un nodo, una lista de planetas objetivo y una lista de planetas de obligada visita
Salida: T si cumplimos los requerimientos, NIL en caso contrario
Función general:

If nodo in planetas-objetivo AND path includes planetas-obligados:
Return T
Else:
Return NIL

Función para comprobar el path (recursiva):

Argumentos recibidos: Nodo actual, nodos obligatorios
If nodos-obligatorios – nodo actual == NULL
Return T
Get-parent (nodo)
If parent == NULL:
Return NIL
Return check-path(parent, obligatorios-modificado)

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN: Exercise 3 -- Goal test
;;
;; Returns a list of all posible actions from a state
;;
;;  Input:
;;    node: the current node (vis. the planet we are on)
;;    planets-destination: List of planets we need to reach (one of them)
;;    planets-mandatory: List of planets we need to visit.
;;
;;  Returns:
;;    T if we reached the goal, NIL otherwise
;;

;; Checks if the path followed includes the mandatory planets
(defun check-path (node planets)
  (let ((new-planets (remove (node-state node)
                             planets)))
    (if (null new-planets) ;; No more mandatory planets to visit
        T
      (let ((parent (node-parent node)))
        (unless (null parent) ;; No more path to analyze
          (check-path parent ;; Check next step of the path
                      new-planets))))))

;; Checks if we fullfilled the requirements
(defun f-goal-test-galaxy (node planets-destination planets-mandatory)
  (and (member (node-state node)
               planets-destination) ;; Check if the node is a destination
       (check-path node planets-mandatory))) ;; Check if we visited the mandatory planets


(defparameter node-01
  (make-node :state 'Avalon) )
(defparameter node-02
  (make-node :state 'Kentares :parent node-01))
(defparameter node-03
  (make-node :state 'Katril :parent node-02))
(defparameter node-04
```

(make-node :state 'Kentares :parent node-03))
(f-goal-test-galaxy node-01 '(kentares urano) '(Avalon Katril))**; -> NIL**
(f-goal-test-galaxy node-02 '(kentares urano) '(Avalon Katril))**; -> NIL**
(f-goal-test-galaxy node-03 '(kentares urano) '(Avalon Katril))**; -> NIL**
(f-goal-test-galaxy node-04 '(kentares urano) '(Avalon Katril))**; -> T**

- **COMENTARIOS**

Para esta función hemos necesitado una función auxiliar que comprobase si el camino seguido ha pasado por todos los nodos obligatorios. Hemos decidido realizar esta función de forma recursiva, de manera que, cada vez que ascendemos un nivel, cribamos la lista de nodos obligatorios, eliminando el nodo actual de la lista. En caso de que la lista de nodos obligatorios quede vacía, significará que hemos pasado por todos ellos y devolveremos T.

Ejercicio 3b:

- **PSEUDOCÓDIGO**

Entrada: Dos nodos y una lista de nodos obligatorios
Salida: T si cumplimos los nodos son equivalentes
Función general: Si ambos nodos corresponden al mismo estado y pasan por los mismos nodos obligatorios: T
Función para conseguir la lista de nodos obligatorios por los que pasan:
Recibe: Un nodo, una lista de nodos obligatorios y el path obligatorio recogido
Si el path recorrido es igual a la lista de nodos obligatorios, devolverlo.
Si es el nodo raíz:
Si es obligatorio:
Añadir al path y devolverlo
Else:
Devolver el path
Else :
Si es obligatorio:
Comprobar path obligatorio con el nodo añadido al path.
Else:
Comprobar path obligatorio sin añadir el nodo al path.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN: Exercise  -- Equal predicate for search states
;;

;; comprueba si dos listas tienen los mismos elementos
(defun equal-lists (list1 list2)
  (and (subsetp list1 list2 :test #'equal)
      (subsetp list2 list1 :test #'equal)))

;; Devuelve los nodos obligatorios que han sido visitados para llegar al nodo
(defun get-mandatory-path (node planets-mandatory path)
  (if (equal-lists planets-mandatory path)
      path
    (let ((parent (node-parent node))
          (state (node-state node)))
      (if (null parent) ;; Nodo raiz
          (if (member state planets-mandatory) ;; Si es obligatorio
              (cons state path) ;; Poner el primero de la lista
            path);; No lo añadimos
        (if (member state planets-mandatory) ;; Si es obligatorio
            (get-mandatory-path parent ;; Resto de nodos
                                planets-mandatory
                                (cons state path)) ;; Lo añadimos
          (get-mandatory-path parent ;; Resto de nodos
                              planets-mandatory
                              path)))))) ;; No lo añadimos

;; Función para valorar si dos nodos son iguales
(defun f-search-state-equal-galaxy (node-1 node-2 &optional planets-mandatory)
  (when (equal (node-state node-1) (node-state node-2))
    (and (equal-lists (get-mandatory-path node-1 planets-mandatory NIL)
                      (get-mandatory-path node-2 planets-mandatory NIL)))))
```

(f-search-state-equal-galaxy node-01 node-01) **;-> T**
(f-search-state-equal-galaxy node-01 node-02) **;-> NIL**
(f-search-state-equal-galaxy node-02 node-04) **;-> T**

(f-search-state-equal-galaxy node-01 node-01 '(Avalon)) **;-> T**
(f-search-state-equal-galaxy node-01 node-02 '(Avalon)) **;-> NIL**
(f-search-state-equal-galaxy node-02 node-04 '(Avalon)) **;-> T**

(f-search-state-equal-galaxy node-01 node-01 '(Avalon Katril)) **;-> T**
(f-search-state-equal-galaxy node-01 node-02 '(Avalon Katril)) **;-> NIL**
(f-search-state-equal-galaxy node-02 node-04 '(Avalon Katril)) **;-> NIL**

- **COMENTARIOS**

Para esta función tenemos que comprobar dos condiciones: Si los dos nodos corresponden al mismo estado y si el path que se sigue para llegar a él pasa por los mismos nodos obligatorios. Para comprobar el path hemos creado una función auxiliar. Ésta va almacenando los nodos obligatorios y en cuanto ha recorrido todos, no continúa, ahorrando tiempo de ejecución ya que no tiene que comprobar hasta el nodo raíz.

Ejercicio 4:

- **PSEUDOCÓDIGO**

Asignar los valores necesarios para inicializar una galaxia:

States: planetas posibles
Initial-state: planeta origen
f-goal-test: función que comprueba si estamos en la meta
f-h: función heurística
f-search-state-equal: función para comprobar si dos nodos son equivalentes
operators: lista de operadores disponibles

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  BEGIN: Exercise 4 -- Define the galaxy structure
;;
;;
(defparameter *galaxy-M35*
  (make-problem
   :states            *planets*
   :initial-state     *planet-origin*
   :f-goal-test       #'(lambda (node)
                      (f-goal-test-galaxy node *planets-destination*
                                  *planets-mandatory*))
   :f-h               #'(lambda (state)
                      (f-h-galaxy state *sensors*))
   :f-search-state-equal #'(lambda (node-1 node-2)
                      (f-search-state-equal-galaxy node-1
                                          node-2
                                          *planets-mandatory*))
   :operators         (list #'(lambda (node)
                        (navigate-worm-hole (node-state node)
                                       *worm-holes*
                                       *planets-forbidden*))
                      #'(lambda (node)
                        (navigate-white-hole (node-state node)
                                        *white-holes*)))))
```

- **COMENTARIOS**

En este apartado nos surgió una duda. f-h y operators no están abstraídos tanto como nos gustaría. Reciben estados, que no coinciden con los nodos de búsqueda, haciendo que la implementación no se vea tan clara.

Ejercicio 5:

- **PSEUDOCÓDIGO**

Entrada: Un nodo y un problema
Salida: Lista de nodos a los que podemos acceder desde el nodo actual.
Función general:

Realizar todos los operadores sobre el nodo.
Por cada acción:
Crear nodo con datos:
Estado = Accion.destino
Padre = nodo actual
Accion = Accion generada
Depth = Padre.depth + 1
g = Parent.g + accion.coste
h = f-h state
f = g + h

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN Exercise 5: Expand node
;;

;; Function to get all the posible actions from a node
(defun get-all-actions (node problem)
  (mapcan #'(lambda (x) (funcall x (node-state node)))
    (problem-operators problem)))

;; Function to expand a node
(defun expand-node (node problem)
  (mapcar #'(lambda (x) ;; Iterating through all actions
          (let* ((final (action-final x))
              (g (+ (node-g node) (action-cost x)))
              (h (funcall (problem-f-h problem) final)))
            (make-node :state final ;; Create a new node structure
                  :parent node
                  :action x
                  :depth (+ (node-depth node) 1)
                  :g g
                  :h h
                  :f (+ g h))))
     (get-all-actions node problem)))

(expand-node (make-node :state 'Kentares :depth 0 :g 0 :f 0) *galaxy-M35*)
;;;(#S(NODE :STATE AVALON
;;;       :PARENT #S(NODE :STATE KENTARES
;;;                  :PARENT NIL
;;;                  :ACTION NIL
;;;                  :DEPTH 0
;;;                  :G ...)
;;;       :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE
;;;                   :ORIGIN KENTARES
;;;                   :FINAL AVALON
;;;                   :COST 3)
;;;       :DEPTH 1
```

```
;;;       :G ...)
;;; #S(NODE :STATE KATRIL
;;;       :PARENT #S(NODE :STATE KENTARES
;;;                  :PARENT NIL
;;;                  :ACTION NIL
;;;                  :DEPTH 0
;;;                  :G ...)
;;;       :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE
;;;                   :ORIGIN KENTARES
;;;                   :FINAL KATRIL
;;;                   :COST 10)
;;;       :DEPTH 1
;;;       :G ...)
;;; #S(NODE :STATE PROSERPINA
;;;       :PARENT #S(NODE :STATE KENTARES
;;;                  :PARENT NIL
;;;                  :ACTION NIL
;;;                  :DEPTH 0
;;;                  :G ...)
;;;       :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE
;;;                   :ORIGIN KENTARES
;;;                   :FINAL PROSERPINA
;;;                   :COST 7)
;;;       :DEPTH 1
;;;       :G ...)
;;; #S(NODE :STATE PROSERPINA
;;;       :PARENT #S(NODE :STATE KENTARES
;;;                  :PARENT NIL
;;;                  :ACTION NIL
;;;                  :DEPTH 0
;;;                  :G ...)
;;;       :ACTION #S(ACTION :NAME NAVIGATE-WORM-HOLE
;;;                   :ORIGIN KENTARES
;;;                   :FINAL PROSERPINA
;;;                   :COST 12)
;;;       :DEPTH 1
;;;       :G ...))
```

Ejercicio 6:

- **PSEUDOCÓDIGO**

Entrada: Dos listas de nodos y una estrategia
Salida: La primera lista introducida en la segunda siguiendo la estrategia
Función general:

Mientras queden nodos en la primera lista:
Introducir el primer nodo en la segunda lista

Función para insertar un nodo:

Si el nodo cumple la estrategia en esa posición devolverlo al principio de la lista
Si no, insertarlo omitiendo el primer nodo de la lista

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  BEGIN Exercise 6 -- Node list management
;;;

;; Parameter for the unifirm-cost strategy
(defparameter *uniform-cost*
  (make-strategy :name 'uniform-cost
                 :node-compare-p #'(lambda (x y) (if (null y)
                                                     t
                                                     (< (node-g x)
                                                      (node-g y))))))  ;; Compare path cost to node

;; Insert a node in the corresponding position accord to the strategy
(defun insert-sort-node (node lst-nodes strategy)
  (let ((first (first lst-nodes)))
    (if (funcall (strategy-node-compare-p strategy)
                 node
                 first) ;; Check strategy
        (cons node lst-nodes) ;; Place the node
      (cons first
            (insert-sort-node node ;; Recursive call
                              (rest lst-nodes)
                              strategy)))))

;; Insert nodes in lst-nodes according to strategy
(defun insert-nodes-strategy (nodes lst-nodes strategy)
  (if (null nodes)
      lst-nodes ;; No more nodes to insert
    (insert-nodes-strategy (rest nodes) ;; Recursive call
                           (insert-sort-node (first nodes) ;; Inserting one node
                                             lst-nodes
                                             strategy)
                           strategy)))

(defparameter node-00
  (make-node :state 'Proserpina :depth 8 :g 4 :f 0) )
(defparameter node-01
  (make-node :state 'Avalon :depth 0 :g 0 :f 0) )
(defparameter node-02
  (make-node :state 'Kentares :depth 2 :g 50 :f 50) )
(defparameter node-03
  (make-node :state 'Davion :depth 0 :g 6 :f 0) )
(defparameter node-04
  (make-node :state 'Katril :depth 2 :g 0 :f 50) )
```

```
(defparameter lst-nodes-00
        (list node-04 node-03))

(print (insert-nodes-strategy (list node-00 node-01 node-02)
                    lst-nodes-00
                    *uniform-cost*))
;;;
;;;(#S(NODE :STATE AVALON :PARENT NIL :ACTION NIL :DEPTH 0 :G 0 :H 0 :F 0)
;;; #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0 :F 20)
;;; #S(NODE :STATE AVALON :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H
0 :F 20)
;;;       :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN PROSERPINA :FINAL AVALON :COST 12)
:DEPTH 13 :G 22 :H 5 :F 27)
;;; #S(NODE :STATE DAVION :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H
0 :F 20)
;;;       :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN PROSERPINA :FINAL DAVION :COST 14)
:DEPTH 13 :G 24 :H 1 :F 25)
;;; #S(NODE :STATE MALLORY :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10
:H 0 :F 20)
;;;       :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN PROSERPINA :FINAL MALLORY :COST 17)
:DEPTH 13 :G 27 :H 7 :F 34)
;;; #S(NODE :STATE SIRTIS :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0
:F 20)
;;;       :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN PROSERPINA :FINAL SIRTIS :COST 10) :DEPTH
13 :G 20 :H 0 :F 20)
;;; #S(NODE :STATE KENTARES :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10
:H 0 :F 20)
;;;       :ACTION #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN PROSERPINA :FINAL KENTARES :COST 21)
:DEPTH 13 :G 31 :H 4 :F 35)
;;; #S(NODE :STATE MALLORY :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10
:H 0 :F 20)
;;;       :ACTION #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN PROSERPINA :FINAL MALLORY :COST 16)
:DEPTH 13 :G 26 :H 7 :F 33)
;;; #S(NODE :STATE SIRTIS :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0
:F 20)
;;;       :ACTION #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN PROSERPINA :FINAL SIRTIS :COST 7) :DEPTH
13 :G 17 :H 0 :F 17)
;;; #S(NODE :STATE KENTARES :PARENT NIL :ACTION NIL :DEPTH 2 :G 50 :H 0 :F 50))


(print
 (insert-nodes-strategy (list node-00 node-01 node-02)
                (sort (copy-list lst-nodes-00) #'<= :key #'node-g)
                *uniform-cost*))
;;;
;;;(#S(NODE :STATE AVALON :PARENT NIL :ACTION NIL :DEPTH 0 :G 0 :H 0 :F 0)
;;; #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0 :F 20)
;;; #S(NODE :STATE SIRTIS :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0
:F 20)
;;;       :ACTION #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN PROSERPINA :FINAL SIRTIS :COST 7) :DEPTH
13 :G 17 :H 0 :F 17)
;;; #S(NODE :STATE SIRTIS :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0
:F 20)
;;;       :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN PROSERPINA :FINAL SIRTIS :COST 10) :DEPTH
13 :G 20 :H 0 :F 20)
;;; #S(NODE :STATE AVALON :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H
0 :F 20)
;;;       :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN PROSERPINA :FINAL AVALON :COST 12)
:DEPTH 13 :G 22 :H 5 :F 27)
```

;;; #S(NODE :STATE DAVION :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0 :F 20)
;;; :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN PROSERPINA :FINAL DAVION :COST 14) :DEPTH 13 :G 24 :H 1 :F 25)
;;; #S(NODE :STATE MALLORY :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0 :F 20)
;;; :ACTION #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN PROSERPINA :FINAL MALLORY :COST 16) :DEPTH 13 :G 26 :H 7 :F 33)
;;; #S(NODE :STATE MALLORY :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0 :F 20)
;;; :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN PROSERPINA :FINAL MALLORY :COST 17) :DEPTH 13 :G 27 :H 7 :F 34)
;;; #S(NODE :STATE KENTARES :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0 :F 20)
;;; :ACTION #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN PROSERPINA :FINAL KENTARES :COST 21) :DEPTH 13 :G 31 :H 4 :F 35)
;;; #S(NODE :STATE KENTARES :PARENT NIL :ACTION NIL :DEPTH 2 :G 50 :H 0 :F 50))

- **COMENTARIOS**

Esta función nos permite insertar los nodos en la lista de abiertos siguiendo una estrategia. Debido a que podemos crear una estrategia para cada caso, nos permite una gran versatilidad para el desarrollo del algoritmo.

Ejercicio 7:

- **PSEUDOCÓDIGO**

Definir la estrategia A*:

Un nodo x va delante de otro y si:

f(x) < f(y)

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN: Exercise 7 -- Definition of the A* strategy
;;
;; A strategy is, basically, a comparison function between nodes to tell
;; us which nodes should be analyzed first. In the A* strategy, the first
;; node to be analyzed is the one with the smallest value of g+h
;;

(defparameter *A-star*
  (make-strategy :name 'A-star
          :node-compare-p #'(lambda (x y) (if (null y)
                                      t
                                     (< (node-f x)
                                      (node-f y)))))) ;; Compare cost + heuristic
```

- **COMENTARIOS**

Para el caso de A*, tenemos que fijarnos en la función f = g + h. Es decir, lo que comprobamos es el coste estimado total para llegar desde el inicio del problema hasta la meta. G es el coste actual hasta el nodo y F la estimación heurística para llegar del nodo a la meta.

Ejercicio 8:

- **PSEUDOCÓDIGO**

Entrada: Un problema y una estrategia
Salida: NIL si no hay solución o el nodo solución (con su camino a base de padres)
Función:
Inicializar la lista de nodos open-nodes con el estado inicial inicializar la lista de nodos closed-nodes con la lista vacía.
Recursión:

1. Si la lista open-nodes está vacía, terminar [no se han encontrado solución]
2. Extraer el primer nodo de la lista open-nodes
3. Si dicho nodo cumple el test objetivo evaluar a la solución y terminar. en caso contrario si el nodo considerado no está en closed-nodes o, estando en dicha lista, tiene un coste g inferior al del que está en closed-nodes:
    a. Expandir el nodo e insertar los nodos generados en la lista open-nodes de acuerdo con la estrategia strategy.
    b. Incluir el nodo recién expandido al comienzo de la lista closed-nodes.
4. Continuar la búsqueda eliminando el nodo considerado de la lista open-nodes.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   BEGIN Exercise 8: Search algorithm
;;;


;;; Function to check if a node is closed
;;; It will return T if the node is already explored and
;;; has a bigger g
(defun check-closed (node closed problem)
 (unless (null closed)
  (let ((test (find node    ;; Check if the node is in the close list
                closed  ;; If the node is in it, it returns the cons of closed.
                :test #'(lambda (x y)
                      (funcall (problem-f-search-state-equal problem)
                           x
                           y)))))
   (and test (> (node-g node)
           (node-g gtest)))))) ;; If the test checks and the g is bigger,
                               ;; we dont want to explore it


(defun recursive-graph-search (open closed problem strategy)
 (unless (null open)
  (let ((first (first open)))
   (if (funcall (problem-f-goal-test problem) ;; Si es la meta
            first)
     first
    (if (check-closed first closed problem)
      (recursive-graph-search (rest open)
                  closed
                  problem
                  strategy) ;; No expandimos ese nodo
     (recursive-graph-search (insert-nodes-strategy (expand-node first
                                          problem)
                                (rest open)
```

```
                    strategy) ;; Expandimos y añadmos a abiertos
                (cons first closed) ;; Lo añadimos a cerrados
                problem
                strategy))))))

(defun graph-search (problem strategy)
  (recursive-graph-search (list (make-node :state (problem-initial-state problem)))
                NIL
                problem
                strategy))

;
;  Solve a problem using the A* strategy
;
(defun a-star-search (problem)
  (graph-search problem *A-star*))


(print (graph-search *galaxy-M35* *A-star*));->
;;;#S(NODE :STATE ...
;;;      :PARENT #S(NODE :STATE ...
;;;                :PARENT #S(NODE :STATE ...))


(print (a-star-search *galaxy-M35*));->
;;;#S(NODE :STATE ...
;;;      :PARENT #S(NODE :STATE ...
;;;                :PARENT #S(NODE :STATE ...))
```

- **COMENTARIOS**

En este ejercicio hemos unido todo el trabajo anterior para resolver el problema de nuestra Galaxia. Dentro de este ejercicio hemos decidido separar la funcionalidad de comprobar si un nodo está en la lista de cerrados para hacer más claro el código de la función principal.
La interfaz externa 'graph-search' inicializa los valores correctos para que funcione nuestra función recursiva.

Ejercicio 9:

- **PSEUDOCÓDIGO**

PATH

Entrada: Un nodo
Salida: Lista de nodos por los que hemos pasado para llegar a dicho nodo
Función:
Si el padre es NIL -> Nodo raíz -> (nodo, path)
En caso contrario -> Añadir nodo al path y seguir escalando

ACTIONS

Entrada: Un nodo
Salida: Lista de acciones que hemos realizado para llegar a dicho nodo
Función:
Si el padre es NIL -> Nodo raíz -> devolver acciones
En caso contrario -> Añadir la acción realizada y seguir escalando

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   BEGIN Exercise 9: Solution path / action sequence
;;;

;; Función recursiva para obtener el path
(defun rec-solution-path (node path)
  (let ((parent (node-parent node))
        (state (node-state node)))
    (if (null parent) ;; Nodo raiz
        (cons state path) ;; Poner el primero de la lista
      (rec-solution-path parent ;; Resto de nodos
                         (cons state path))))) ;; Ponerlos al principio de la lista y seguir

;; Función que devuelve el path hasta node
(defun solution-path (node)
  (unless (null node) ;; Comprobar si es NIL
    (rec-solution-path node NIL)))

(solution-path nil) ;;; -> NIL
(solution-path (a-star-search *galaxy-M35*))  ;;;-> (MALLORY ...)

;; Función recursiva que devuelve la lista de acciones para llegar al nodo
(defun rec-action-sequence (node actions)
  (let ((parent (node-parent node))
        (action (node-action node)))
    (if (null parent) ;; Nodo raiz
        actions ;; Devolver la lista
      (rec-action-sequence parent ;; Resto de nodos
                           (cons action actions))))) ;; Ponerlos al principio de la lista y seguir

;; Función que devuelve la lista de acciones necesarias para llegar al nodo
(defun action-sequence (node)
  (unless (null node) ;; Comprobar si es NIL
    (rec-action-sequence node NIL)))

(action-sequence (a-star-search *galaxy-M35*))
;;; ->
;;;(#S(ACTION :NAME ...))
```

- **COMENTARIOS**

Estas funciones hacen mucho más claro el camino a seguir para resolver el problema. Además especifican qué acciones hay que tomar para seguir dicho camino.

Ejercicio 10:

- **PSEUDOCÓDIGO**

1. BFS:
   En BFS, introducimos los nuevos nodos explorados en ultima posición, de manera que se exploren primero aquellos nodos generados antes.
2. DFS:
   En DFS, introducimos los nuevos nodos explorados en primera posición, de manera que se exploren primero aquellos nodos recién generados.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   BEGIN Exercise 10: depth-first / breadth-first
;;;

(defun depth-first-node-compare-p (node-1 node-2)
  T) ;; Last explored node gets in the first position of open-nodes

(defparameter *depth-first*
  (make-strategy
   :name 'depth-first
   :node-compare-p #'depth-first-node-compare-p))

(solution-path (graph-search *galaxy-M35* *depth-first*))
;;; -> (MALLORY ... )

(defun breadth-first-node-compare-p (node-1 node-2)
  (null node-2)) ;; Last explored node gets in the last position of open-nodes

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

(solution-path (graph-search *galaxy-M35* *breadth-first*))
;; -> (MALLORY ... )
```

- **COMENTARIOS**

Estas funciones parecen triviales ya que DFS solamente devuelve T. Esto sucede debido a que nuestra función de inserción de nodos según una estrategia se aprovecha de eso para introducirlo en la primera posición.
En BFS sucede algo parecido, pero tenemos que tener en cuenta que la lista se acaba. En caso de que el nodo a comparar sea NIL, significa que ya estamos al final de la lista y tendremos que devolver T para que se introduzca el nodo en dicha posición.

Ejercicio 11:

1. Se ha decidido realizar este esquema debido a que abstrae el algoritmo de búsqueda del problema en sí. De esta manera podemos reutilizar esta misma función para resolver cualquier problema de búsqueda que podamos definir en la estructura "problema".
   Definir las funciones mediante Lambdas, nos facilita su llamada a lo largo del código. De esta forma, evitamos modificar el código cada vez que queramos utilizar funciones diferentes para definir el problema.
   Por otra parte, la estructura de nodos anidados nos permite obtener el path recorrido hasta el nodo de una manera más sencilla.
2. El uso de memoria con el mecanismo de nodos anidados es bastante eficiente ya que no creamos copias de cada nodo, sino que tendremos referencias a ellos. Necesitamos almacenar todos aquellos nodos que hayamos visitado en una lista de "cerrados", por lo que añadir referencias entre ellos no es costoso en memoria y la ventaja que tiene es muy significativa para la claridad del código.
3. Cada vez que expandimos un nodo, tenemos que guardarlo en memoria, por lo que estamos guardando todos los nodos explorados. Considerando 'b' el número de nodos expandidos (factor de ramificación) y 'd' el número de acciones necesarias en el camino óptimo:
   El coste espacial es O(b^d)
4. El coste temporal es muy similar, pero debemos tener en cuenta el coste de cada acción. Llamémosle 'a'. Si suponemos a = 1, el coste temporal es exactamente el mismo que el espacial. Pero si es distinto, debemos tenerlo en cuenta al expandir cada nodo. Por ello, pasaría a ser O(a*(b^d)) = O(b^d)
5. En cada nodo podríamos añadir un campo que cuente el número de "saltos de agujero de gusano realizados hasta llegar a dicho nodo". De esta manera, en la función para expandir un nodo deberíamos tener en cuenta si se ha llegado al máximo o no. Devolviendo solo las acciones realizadas mediante agujeros blancos.