Inteligencia Artificial
Práctica 2

Martín Selgas, Blanca (blanca.martins@estudiante.uam.es)
Villar Gómez, Fernando (fernando.villarg@estudiante.uam.es)
Grupo 2302 - Pareja 3

05/04/2018

Índice

1. Introducción

En el presente documento se detalla la realización de los problemas pertenecientes a la segunda práctica de Inteligencia Artificial. En ellos se requiere la resolución de determinadas tareas en el lenguaje funciónal LISP, aportando el pseudo-código, la codificación, la documentación y los comentarios de las funciónes utilizadas para tales fines.

El desarrollo de la práctica se ha realizado en la aplicación **Allegro CL 6.2 ANSI** corriendo sobre un sistema Windows 10, pero el código utilizado es compatible con otros entornos LISP como **SBCL**.

Los ejercicios de esta práctica están orientados a la resolución de problemas de búsqueda dentro de una galaxia, **Messier 35**, que tiene una serie de planetas entre los cuales se puede viajar a lo largo de agujeros blancos (unidireccionales) o agujeros de gusano (bidireccionales) con diversos costes.

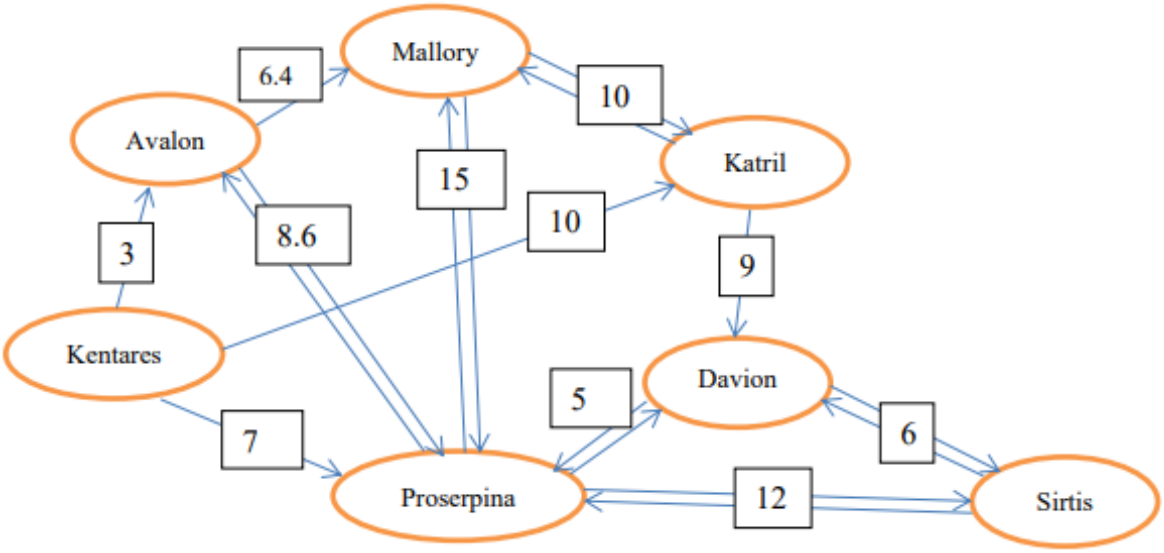

Figura 1: Agujeros blancos de la galaxia Messier 35

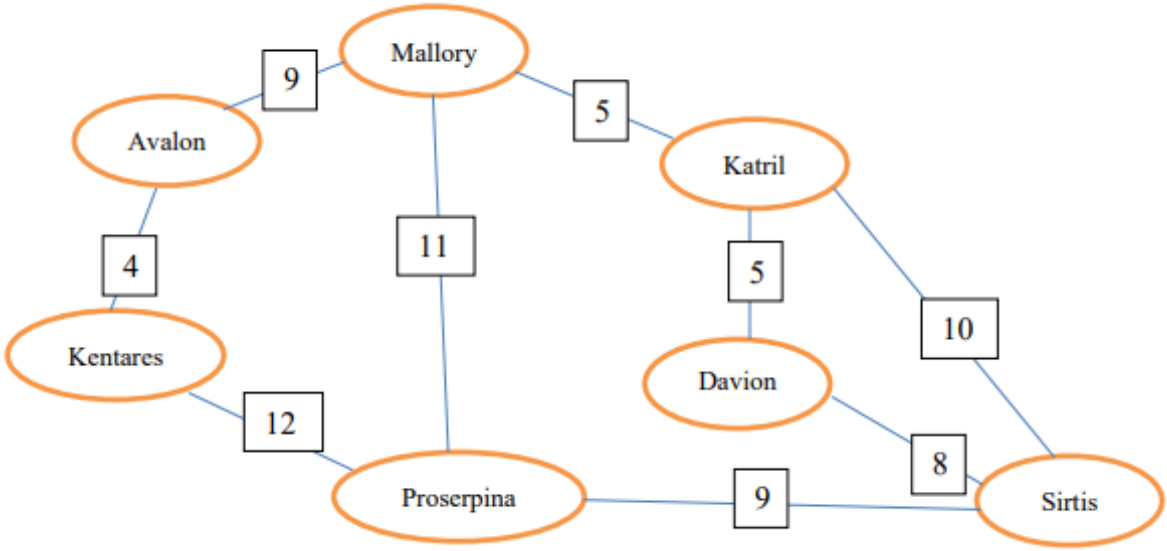

Figura 2: Agujeros de gusano de la galaxia Messier 35

2. Ejercicio 1

Para este ejercicio se utiliza la siguiente heurística determinada en el enunciado de la práctica, que estima el coste de viajar desde cualquier planeta de M-35 hasta Sirtis:

Planeta: n	**h(n)**
Avalon	15
Mallory	12
Kentares	14
Davion	5
Proserpina	7
Katril	9
Sirtis	0

Para ello, dado que en el parámetro `*sensors*` tenemos una lista de pares $(n, h(n))$, utilizamos la función `assoc` nativa de LISP, que devuelve el miembro de una lista que empieza por el elemento deseado. No se incluye pseudocódigo por la simpleza del mismo.

f-h-galaxy

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; f-h-galaxy
;;;
;;; 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)
  (second (assoc state sensors)))

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

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Esta función nunca tendrá problemas de ejecución siempre y cuando la lista `*sensors*` sea siempre de pares $(n, h(n))$ en la que no haya planetas repetidos, puesto que la función `assoc` tiene estos requisitos para funcionar de forma correcta. De todas formas, una heurística no tendría sentido si se le pudieran asignar dos valores diferentes a un nodo.
- Toda la práctica en su conjunto depende de cuál sea el planeta de destino, puesto que la heurística variará si este planeta cambia. Por tanto, si queremos cambiar el destino, hay que cambiar también la lista que contiene la información sobre la heurística.

3. Ejercicio 2

En este ejercicio se ha seguido la recomendación del enunciado, con la creación de una función que englobe el funcionamiento de las dos solicitadas, cuya codificación final se muestra bajo la de la función principal.

navigate

PSEUDOCÓDIGO

```
Entrada: name (nombre de la acción)
         state (nombre del estado (planeta) de origen)
         holes (lista de todos los posibles viajes: (origen, destino, coste))
         planets-forbidden (lista de planetas por los que no se puede pasar)
Salida:  lista de acciones posibles desde el origen

Procesamiento:
         si holes está vacío,
             evalúa a NIL
         en caso contrario,
             siguiente := navigate(name, state, holes[1:n], planets-forbidden)
             si state=holes[0][0] & holes[0][1] NOT IN planets-forbidden
                 accion = nueva-accion(name, state, holes[0][1], holes[0][2])
                 evalúa a accion + siguiente
             en caso contrario,
                 evalúa a siguiente
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; navigate
;;;
;;; Returns the list of actions that can be executed from one state using
;;; one operator.
;;;
;;;  Input: name : the name of the action (vis. the operator)
;;;         state: the current state (vis. the planet we are on)
;;;         holes: a holes list, that is, a list of three of a kind
;;;                where the first element is the name of a state,
;;;                the second the name of the final state, and the third
;;;                a number estimating the cost to reach the goal.
;;;         planets-forbidden: a list containing the forbidden planets.
;;;
;;;  Returns:
;;;    A list with the possible actions or NIL if no action can be executed.
;;;

(defun navigate (name state holes planets-forbidden)
  (unless (null holes)
    (let ((primero      (first holes))
          (navigate-sig (navigate name state
                                  (rest holes) planets-forbidden)))
    (if (and (equal (first primero) state)
             (not (some #'(lambda(x) (equal x (second primero)))
                        planets-forbidden)))
      (append (list (make-action :name name
                                 :origin state
                                 :final (second primero)
                                 :cost (third primero)))
              navigate-sig)
      navigate-sig))))

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; navigate-white-hole
;;;
;;; Returns the list of actions that can be executed from one state using
;;; white holes.
;;;
;;;  Input: state      : the current state (vis. the planet we are on)
;;;         white-holes: a holes list, that is, a list of three of a kind
;;;                      where the first element is the name of a state,
;;;                      the second the name of the final state, and the third
;;;                      a number estimating the cost to reach the goal.
;;;
;;;  Returns:
;;;    A list with the possible actions or NIL if no action can be executed.
;;;

(defun navigate-white-hole (state white-holes)
  (navigate 'navigate-white-hole state white-holes NIL))

(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
;;;
;;; Returns the list of actions that can be executed from one state using
;;; worm holes.
;;;
;;;  Input: state     : the current state (vis. the planet we are on)
;;;         worm-holes: a holes list, that is, a list of three of a kind
;;;                     where the first element is the name of a state,
;;;                     the second the name of the final state, and the third
;;;                     a number estimating the cost to reach the goal.
;;;         planets-forbidden: a list containing the forbidden planets.
;;;
;;;  Returns:
;;;    A list with the possible actions or NIL if no action can be executed.
;;;

(defun navigate-worm-hole (state worm-holes planets-forbidden)
  (navigate 'navigate-worm-hole state worm-holes planets-forbidden))

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

COMENTARIOS:

- La implementación por separado es exactamente la misma con la única diferencia en la determinación del nombre de la acción.

4. Ejercicio 3

4.1. Apartado 3A

Se ha usado una función en este apartado que devuelve la lista de planetas obligatorios por los que aún se tiene que pasar en un camino dado por un nodo:

get-pending-mandatory

PSEUDOCÓDIGO

```
Entrada: node (nodo que representa el camino)
         planets-mandatory (lista de planetas obligatorios)
Salida:  lista de planetas obligatorios no visitados

Procesamiento:
         si node o planets-mandatory están vacíos,
             evalúa a planets-mandatory
         en caso contrario,
             si node.state IN planets-mandatory
                 eliminarDe(node.state, planets-mandatory)
             evalúa a get-pending-mandatory(node.parent, planets-mandatory)
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; get-pending-mandatory
;;;
;;; Returns the list of mandatory states that have yet to be visited.
;;;
;;;  Input: node              : the current node (vis. the planet we are on)
;;;         planets-mandatory: a list containing the names of the mandatory
;;;                            states that have to be visited.
;;;
;;;  Returns:
;;;    A list with the mandatory states not visited or NIL if all mandatory
;;;    states have been visited.

(defun get-pending-mandatory (node planets-mandatory)
  (if (or (null node)
          (null planets-mandatory))
      planets-mandatory
    (let ((state (node-state node))
          (parent (node-parent node)))
      (if (some #'(lambda (planet) (equal planet state))
                planets-mandatory)
          (get-pending-mandatory parent
                                 (remove state planets-mandatory))
        (get-pending-mandatory parent
                               planets-mandatory)))))
```

```
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- La lista que devuelve esta función no tiene ningún orden específico. Por tanto, cuando en el futuro hagamos comparaciones entre listas que devuelva esta función, habrá que tener en cuenta que no basta con un test `equal`, sino que hay que comprobar si cada uno de los elementos de una lista están contenidos en la otra y viceversa.

Finalmente, por tanto, la función para evaluar si hemos llegado a la meta es:

f-goal-test-galaxy

PSEUDOCÓDIGO

```
Entrada: node (nodo a evaluar)
         planets-destination (lista de metas)
         planets-mandatory (lista de planetas de obligatoria visita)
Salida:  T si node es meta, NIL en caso contrario

Procesamiento:
         si node.state IN planets-destination
             restantes = get-pending-mandatory(node, planets-mandatory)
             si restantes está vacío,
                 evalúa a T
             en caso contrario,
                 evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; f-goal-test-galaxy
;;;
;;; Evaluates if the current node is the goal.
;;;
;;;  Input: node : the current node (vis. the planet we are on)
;;;         planets-destination: a list containing the names of the
;;;                              destination planets.
;;;         planets-mandatory  : a list containing the names of the
;;;                              mandatory states that has to be visited.
;;;
;;;  Returns:
;;;    T if the node is the goal, NIL if it is not.
;;;

(defun f-goal-test-galaxy (node planets-destination planets-mandatory)
  (when (member (node-state node) planets-destination)
    (null (get-pending-mandatory node planets-mandatory))))

(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:

- Aunque `planets-destination` sea una lista de planetas, evidentemente basta con que el nodo actual sea un planeta de uno solo de los de la lista.
- En este caso no importa el orden del retorno de `get-pending-mandatory` dado que sólo necesitamos comprobar si nos devuelve una lista vacía o no.
- Otra posible implementación de este problema sería que la función auxiliar determinara directamente si se ha pasado o no por todos los planetas de obligatoria visita, y lo único que habría que cambiar sería que se devolviera T si se envía una lista `planets-mandatory` vacía o NIL si se envía un nodo vacío, pero para el futuro `get-pending-mandatory` será de utilidad.

4.2. Apartado 3B

De nuevo usamos en este apartado otra función auxiliar que determina si a dos nodos (caminos) les faltan los mismos planetas de obligatoria visita por visitar:

pending-mandatory-p

PSEUDOCÓDIGO

```
Entrada: node-1 (primer nodo a comparar)
         node-2 (segundo nodo a comparar)
         planets-mandatory (lista de planetas de obligatoria visita)
Salida:  T si les faltan los mismos planetas obligatorios, NIL en caso contrario

Procesamiento:
         lista-1 = get-pending-mandatory(node-1, planets-mandatory)
         lista-2 = get-pending-mandatory(node-2, planets-mandatory)
         si lista-1 está contenida en lista-2 & lista-2 está contenida en lista-1
             evalúa a T
         en caso contrario,
             evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; pending-mandatory-p
;;;
;;; Evaluates if two paths have the same pending mandatory planets to be
;;; visited.
;;;
;;;  Input: node-1            : first node to be evaluated
;;;         node-2            : second node to be evaluated
;;;         planets-mandatory : list of planets to be visited compulsory
;;;
;;;  Returns: T if both paths have the same set of pending mandatory planets
;;;           to be visited, NIL otherwise
;;;

(defun pending-mandatory-p (node-1 node-2 planets-mandatory)
  (let ((pending-1 (get-pending-mandatory node-1 planets-mandatory))
        (pending-2 (get-pending-mandatory node-2 planets-mandatory)))
    (when (and (subsetp pending-1 pending-2 :test 'equal)
               (subsetp pending-2 pending-1 :test 'equal))
      t)))

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- En esta función es en la que había que tener en cuenta la ausencia de un orden específico en la salida de `get-pending-mandatory`, dado que el test `equal` sólo devolvería verdadero si las dos listas tuvieran los mismos elementos dispuestos en el mismo orden. Este problema se resuelve comprobando si cada lista es una sublista de la otra (dado que el retorno sí es un conjunto, esto es equivalente a una igualdad/doble contenido de conjuntos).

Por tanto, la función para evaluar si dos estados de búsqueda son iguales es la siguiente (teniendo en cuenta que ser un estado de búsqueda igual implica estar en el mismo planeta y tener la misma lista de planetas obligatorios sin visitar):

f-search-state-equal-galaxy

PSEUDOCÓDIGO

```
 Entrada: node-1 (primer nodo a comparar)
          node-2 (segundo nodo a comparar)
          planets-mandatory (opcional, si no se incluye cuenta como lista vacía)
 Salida:  T si node-1 y node-2 corresponden al mismo estado, NIL en caso contrario

 Procesamiento:
          si node-1.state = node-2.state,
              si planets-mandatory está vacío,
                  evalúa a T
              en caso contrario,
                  evalúa a pending-mandatory-p(node-1, node-2, planets-mandatory)
          en caso contrario,
              evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; f-search-state-equal-galaxy
;;;
;;; Evaluates if two nodes are the same search state. Two nodes are the
;;; same search state when they have the same state and they have visited
;;; the same mandatory planets.
;;;
;;;  Input: node-1               : first node to be evaluated
;;;         node-2               : second node to be evaluated
;;;         [planets-mandatory]  : list of planets to be visited compulsory
;;;
;;;  Returns:
;;;    T if both nodes are the same search state, NIL if they are not.
;;;
(defun f-search-state-equal-galaxy (node-1 node-2 &optional planets-mandatory)
  (when (equal (node-state node-1) (node-state node-2))
    (or (null planets-mandatory)
        (pending-mandatory-p node-1 node-2 planets-mandatory))))

(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:

- La lógica más entendible para esta función es la del pseudocódigo, pero la codificación es diferente y quizá menos comprensible; sin embargo, es equivalente y, a nuestro juicio, más elegante.

5. Ejercicio 4

La definición de la galaxia M-35 es la siguiente:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;

(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 (state)
                                    (navigate-white-hole state
                                                         *white-holes*))
                                #'(lambda (state)
                                    (navigate-worm-hole state
                                                        *worm-holes*
                                                        *planets-forbidden*)))))

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

6. Ejercicio 5

expand

PSEUDOCÓDIGO

```
Entrada: node (nodo a expandir)
         heuristic (heurística del problema)
         operatos (lista de operadores)
Salida:  lista de nodos resultado de la expansión.

Procesamiento:
 Si operators != null:
     Para cada acción posible aplicando operators[0] a state(node):
         new child-nodo tal que: state := estado final al aplicar la acción
                                 parent := node
                                 action := acción
                                 depth := depth(node) + 1
                                 g := g(node) + coste(acción)
                                 h := heuristic(child-nodo)
                                 f := g + h
         childs[].append(child-nodo)
     evalúa a childs[] + expand(node, heuristic, resto de operators)
```

en caso contrario:
evalúa a NIL

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; expand
;;;
;;; Returns the expansion of a given node with a heuristic and a list
;;; of operators.
;;;
;;;  Input: node     : the current node (vis. the planet we are on)
;;;         heuristic: the heuristic of the problem.
;;;         operators: list of operators that can be executed from the node.
;;;
;;;  Returns:
;;;    A list of nodes accessible from the given node by navigating using
;;;    the given operators.
;;;

(defun expand (node heuristic operators)
  (unless (null operators)
    (append (mapcar #'(lambda(x) (let* ((state (action-final x))
                                        (g     (+ (node-g node) (action-cost x)))
                                        (h     (funcall heuristic state)))
                                   (make-node :state state
                                              :parent node
                                              :action x
                                              :depth (+ (node-depth node) 1)
                                              :g g
                                              :h h
                                              :f (+ g h))))
              (funcall (first operators) (node-state node)))
            (expand node heuristic (rest operators)))))

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; expand-node
;;;
;;; Returns the expansion of a node
;;;
;;;  Input: node   : the current node (vis. the planet we are on)
;;;         problem: the problem we are solving.
;;;
;;;  Returns:
;;;    A list of nodes accessible from the given node by navigating using
;;;    the operators of the problem.
;;;

(defun expand-node (node problem)
  (expand node (problem-f-h problem) (problem-operators problem)))


(defparameter node-00
  (make-node :state 'Proserpina :depth 12 :g 10 :f 20))

(defparameter lst-nodes-00
  (expand-node node-00 *galaxy-M35*))

(print lst-nodes-00) ;->
;;; (#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 8.6)
;;;           :DEPTH 13 :G 18.6 :H 15 :F 33.6)
;;;  #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 5)
;;;           :DEPTH 13 :G 15 :H 5 :F 20)
;;;  #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 15)
;;;           :DEPTH 13 :G 25 :H 12 :F 37)
;;;  #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 12)
;;;           :DEPTH 13 :G 22 :H 0 :F 22)
;;;  #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 12)
;;;           :DEPTH 13 :G 22 :H 14 :F 36)
;;;  #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 9)
;;;           :DEPTH 13 :G 19 :H 0 :F 19)
;;;  #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 11)
;;;           :DEPTH 13 :G 21 :H 12 :F 33))

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- En este ejercicio se ha decidido concentrar la funcionalidad principal en la función auxiliar *expand* para facilitar la recursividad al iterar sobre los operadores del problema y evitar complejidad innecesaria. De esta forma, además, se aumenta la modularización de la implementación admitiendo futuros cambios en las estructuras del problema que no afecten al pseudocódigo descrito.

7. Ejercicio 6

insert-sort

PSEUDOCÓDIGO

Entrada: *node* (nodo a insertar)
lst − nodes (lista de nodos)
comparison (criterio de inserción)
Salida: lst-nodes con node insertado siguiendo el criterio de ordenación comparison.

Procesamiento:

```
Si lst-nodes == null:
    evalúa a (node)
en caso contrario
    Si comparison de node y el primero de lst-nodes == true:
        evalúa a node + lst-nodes
    en caso contrario
        evalúa a insert-sort(node, resto de lst-nodes, comparison)
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; insert-sort
;;;
;;; Inserts a node in a list of nodes according to a given comparison
;;; between nodes.
;;;
;;;  Input: node      : node to be inserted.
;;;         lst-nodes : list of ordered nodes.
;;;         comparison: comparison criteria for the insertion.
;;;
;;;  Returns:
;;;    A list of ordered nodes result of the insertion of the node in
;;;    the list of nodes.
;;;

(defun insert-sort (node lst-nodes comparison)
  (if (null lst-nodes)
      (list node)
    (if (funcall comparison node (first lst-nodes))
        (cons node
              lst-nodes)
      (cons (first lst-nodes)
            (insert-sort node (rest lst-nodes) comparison)))))

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- La idea tras este algoritmo es recorrer la lista de nodos *list-nodes* hasta encontrar un nodo que no cumpla la condición *comparison*, y entonces insertar el nodo *node* delante de él, por lo que la lista que se obtiene al final sigue estando ordenada.
- Dado que la complejidad de este algoritmo en el peor caso es solamente $O(n)$, se ha optado por esta versión sobre otras (como la correspondiente a las búsquedas binarias) ligeramente más rápidas por su simple y entendible implementación.

insert-nodes-strategy

PSEUDOCÓDIGO

Entrada: *nodes* (lista de nodos a insertar)
lst − nodes (lista de nodos ordenada)
strategy (estrategia a seguir)
Salida: lst-nodes con nodes insertados según strategy.

Procesamiento:

```
Si nodes == null:
    lst-nodes
en caso contrario:
    lista-ordenada = insert-sort(primero de nodes, lst-nodes, comparación(strategy))
    insert-nodes-strategy(resto de nodes, lista-ordenada, strategy)
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; insert-nodes-strategy
;;;
;;; Inserts a list of nodes in an ordered list of nodes following a
;;; given strategy.
;;;
;;;  Input: nodes    : list of nodes to be inserted.
;;;         lst-nodes: list of ordered nodes.
;;;         strategy : strategy to be followed in the insertion. lst-nodes
;;;                    is also supposed to be ordered following this strategy.
;;;
;;;  Returns:
;;;    A list of ordered nodes result of the insertion.
;;;

(defun insert-nodes-strategy (nodes lst-nodes strategy)
  (if (null nodes)
      lst-nodes
    (insert-nodes-strategy (rest nodes)
                           (insert-sort (first nodes)
                                        lst-nodes
                                        (strategy-node-compare-p strategy))
                           strategy)))

(defparameter node-000
  (make-node :state 'Proserpina :depth 1 :g 20 :f 20) )
(defparameter node-001
  (make-node :state 'Avalon :depth 0 :g 0 :f 0) )
(defparameter node-002
  (make-node :state 'Kentares :depth 2 :g 50 :f 50) )
(defparameter lst-nodes-00
  (list
    (make-node :state 'Davion :depth 3 :g 25 :f 30)
    (make-node :state 'Mallory :depth 3 :g 10 :f 40)
    (make-node :state 'Katril :depth 3 :g 60 :f 60)))

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

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

(insert-nodes-strategy '(4 8 6 2)
                       '(1 3 5 7)
                       (make-strategy :name 'simple :node-compare-p #'<)) ;->
;;; (1 2 3 4 5 6 7 8)

(print (insert-nodes-strategy (list node-000 node-001 node-002)
                              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 1 :G 20 :H 0 :F 20)
;;; #S(NODE :STATE MALLORY :PARENT NIL :ACTION NIL :DEPTH 3 :G 25 :H 0 :F 25)
;;; #S(NODE :STATE DAVION :PARENT NIL :ACTION NIL :DEPTH 3 :G 10 :H 0 :F 10)
;;; #S(NODE :STATE KENTARES :PARENT NIL :ACTION NIL :DEPTH 2 :G 50 :H 0 :F 50)
;;; #S(NODE :STATE KATRIL :PARENT NIL :ACTION NIL :DEPTH 3 :G 60 :H 0 :F 60))

(print
 (insert-nodes-strategy (list node-000 node-001 node-002)
```

```
                        (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 MALLORY :PARENT NIL :ACTION NIL :DEPTH 3 :G 10 :H 0 :F 25)
;;; #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 1 :G 20 :H 0 :F 20)
;;; #S(NODE :STATE DAVION :PARENT NIL :ACTION NIL :DEPTH 3 :G 25 :H 0 :F 10)
;;; #S(NODE :STATE KENTARES :PARENT NIL :ACTION NIL :DEPTH 2 :G 50 :H 0 :F 50)
;;; #S(NODE :STATE KATRIL :PARENT NIL :ACTION NIL :DEPTH 3 :G 60 :H 0 :F 60))

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- *insert-nodes-strategy* va recorriendo la lista a ordenar y hace uso de la función *insert-sort* para insertar uno a uno los nodos en la lista ordenada *list-nodes* según la estrategia dada *strategy*.

8. Ejercicio 7

La estrategia correspondiente a la búsqueda A* es la siguiente:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;

(defparameter *A-star*
  (make-strategy
   :name 'A-star-strategy
   :node-compare-p #'(lambda (node1 node2) (< (node-f node1)
                                              (node-f node2)))))

(print (insert-nodes-strategy (list node-000 node-001 node-002)
                              (copy-list lst-nodes-00)
                              *A-star*)) ;->
;;; (#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 1 :G 20 :H 0 :F 20)
;;; #S(NODE :STATE DAVION :PARENT NIL :ACTION NIL :DEPTH 3 :G 25 :H 0 :F 30)
;;; #S(NODE :STATE MALLORY :PARENT NIL :ACTION NIL :DEPTH 3 :G 10 :H 0 :F 40)
;;; #S(NODE :STATE KENTARES :PARENT NIL :ACTION NIL :DEPTH 2 :G 50 :H 0 :F 50)
;;; #S(NODE :STATE KATRIL :PARENT NIL :ACTION NIL :DEPTH 3 :G 60 :H 0 :F 60))

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

9. Ejercicio 8

graph-search-aux

PSEUDOCÓDIGO

```
Entrada: problem (problema a resolver)
         strategy (estrategia de resolución del problema)
         open-nodes (lista de abiertos)
         closed-nodes (lista de cerrados)
Salida:  camino (dado por un nodo) tras aplicar la estrategia strategy a
         las condiciones dadas

Procesamiento:
         si open-nodes está vacía,
             evalúa a NIL
         en caso contrario,
```

```
                node := open-nodes[0]
                si node es meta,
                    evalúa a node
                en caso contrario,
                    equal-closed-node := buscarMismoEstadoEn(node, closed-nodes)
                    si equal-closed-node no existe OR node.g < equal-closed-node.g
                        expanded-nodes := expand-node(node, problem)
                        new-open-nodes := insert-nodes-strategy(expanded-nodes,
                                                                open-nodes[1:n],
                                                                strategy)
                        new-closed-nodes := concatenar(node, closed-nodes)
                        evalúa a graph-search-aux(problem, strategy,
                                                  new-open-nodes, new-closed-nodes)
                    en caso contrario,
                        evalúa a graph-search-aux(problem, strategy,
                                                  open-nodes[1:n], closed-nodes)
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; graph-search-aux
;;;
;;; Returns the path (node) after applying a graph search of a given problem
;;; with a given strategy, using a list of open nodes and a list of closed
;;; nodes prearranged.
;;;
;;;  Input: problem:       problem to be solved
;;;         strategy:      strategy to be followed
;;;         open-nodes:    list of prearranged open nodes
;;;         closed-nodes:  list of prearranged closed nodes
;;;
;;;  Returns:
;;;         Node that contains the path of the result after applying graph
;;;         search with a certain strategy and initial list of open and
;;;         closed nodes.

(defun graph-search-aux (problem strategy open-nodes closed-nodes)
  (unless (null open-nodes)
    (let ((node (first open-nodes)))
      (if (funcall (problem-f-goal-test problem) node)
          node
        (let ((equal-closed-node
               (find node
                     closed-nodes
                     :test #'(lambda (x y)
                               (funcall (problem-f-search-state-equal problem)
                                        x y)))))
          (if (or (null equal-closed-node)
                  (< (node-g node) (node-g equal-closed-node)))
              (graph-search-aux problem
                                strategy
                                (insert-nodes-strategy (expand-node node problem)
                                                       (rest open-nodes)
                                                       strategy)
                                (cons node closed-nodes))
            (graph-search-aux problem
                              strategy
                              (rest open-nodes)
                              closed-nodes)))))))

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Con la función auxiliar que acabamos de detallar, la función final que toma un problema y una estrategia y realiza la búsqueda es la siguiente, donde se inicializan las estructuras a utilizar y se llama a `graph-search-aux`:

```
(defun graph-search (problem strategy)
  (let* ((root (problem-initial-state problem))
         (root-h (funcall (problem-f-h problem) root))
         (open-nodes (list (make-node :state root :h root-h :f root-h)))
         (closed-nodes '()))
    (graph-search-aux problem strategy open-nodes closed-nodes)))
```

- Por tanto, la función que encapsula la galaxia que nos ocupa, Messier 35, es la siguiente:

```
(defun a-star-search (problem)
  (graph-search problem *A-star*))
```

- Es importante puntualizar algo sobre la línea de código siguiente:

```
(if (or (null equal-closed-node)
        (< (node-g node) (node-g equal-closed-node)))
```

  Como podemos ver, el resultado de esta comparación determina cómo se llama recursivamente a la función `graph-search-aux`; si el nodo que estamos evaluando representa el mismo estado que uno de la lista de cerrados y, además, el coste de este nodo es menor que el de la lista de cerrados, tenemos que el nodo evaluado ha de tomar preferencia.

  Se podría pensar en un principio que esto hace necesario borrar el estado antiguo de la lista de cerrados; sin embargo, no es necesario, puesto que estamos colocando el nuevo estado al principio de la lista. Cuando esta circunstancia se repita, al realizarse la búsqueda sobre la lista linealmente, nos encontraremos primero con el estado más moderno, y por tanto los antiguos no molestan.

10. Ejercicio 9

Para la implementación de las dos funciones de este ejercicio, sencillamente se va recorriendo un camino a partir de un nodo hoja de forma inversa e introduciendo en una lista el estado o la acción de cada nodo recorrido en el proceso (ordenado de raíz a hoja).

solution-path

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; solution-path
;;;
;;; Returns the list of planets (states) of a path given by a node
;;;
;;;  Input: node: node to be evaluated
;;;
;;;  Returns: list of planets from root to leaf of the path given by the node
;;;

(defun solution-path (node)
  (unless (null node)
    (append (solution-path (node-parent node)) (list (node-state node)))))

(solution-path nil) ;->
;;; NIL
(solution-path (a-star-search *galaxy-M35*)) ;->
;;; (MALLORY KATRIL DAVION PROSERPINA SIRTIS)

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

action-sequence

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; action-sequence
;;;
;;; Returns the list of actions taken in a path given by a node
;;;
;;;  Input: node: node to be evaluated
;;;
;;;  Returns: list of actions from root to leaf in the path given by the node
;;;

(defun action-sequence (node)
  (unless (null (node-parent node))
    (append (action-sequence (node-parent node)) (list (node-action node)))))

(action-sequence (a-star-search *galaxy-M35*)) ;->
;;; (#S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN MALLORY :FINAL KATRIL :COST 5)
;;; #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN KATRIL :FINAL DAVION :COST 5)
;;; #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN DAVION :FINAL PROSERPINA :COST 5)
;;; #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN PROSERPINA :FINAL SIRTIS :COST 9))

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

11. Ejercicio 10

La única diferencia entre la estrategia A* y las de búsqueda en profundidad o en anchura es la manera en la que se comparan los nodos a la hora de ordenarlos en la lista de abiertos que se usa en la función del ejercicio 8. En concreto:

- Para la **búsqueda en profundidad** necesitamos que se explore siempre hacia abajo en el árbol, es decir, que se explore el nodo de mayor profundidad en todo momento, por lo que la función para comparar dos nodos será:

  ```
  (defun depth-first-node-compare-p (node-1 node-2)
    (> (node-depth node-1) (node-depth node-2)))
  ```

- Al contrario, para la **búsqueda en anchura** necesitamos que se explore hacia la derecha en el árbol, lo cual equivale a explorar el nodo con menor profundidad, y por tanto la función resultante será:

  ```
  (defun breadth-first-node-compare-p (node-1 node-2)
    (< (node-depth node-1) (node-depth node-2)))
  ```

Por tanto, las estrategias serán las siguientes:

depth-first

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;

(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 KATRIL DAVION PROSERPINA AVALON MALLORY KATRIL DAVION SIRTIS)

(action-sequence (graph-search *galaxy-M35* *depth-first*)) ;->
;;; (#S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN MALLORY :FINAL KATRIL :COST 10)
;;; #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN KATRIL :FINAL DAVION :COST 9)
;;; #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN DAVION :FINAL PROSERPINA :COST 5)
;;; #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN PROSERPINA :FINAL AVALON :COST 8.6)
;;; #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN AVALON :FINAL MALLORY :COST 6.4)
;;; #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN MALLORY :FINAL KATRIL :COST 10)
;;; #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN KATRIL :FINAL DAVION :COST 9)
;;; #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN DAVION :FINAL SIRTIS :COST 6))

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

breadth-first

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;

(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 KATRIL DAVION PROSERPINA SIRTIS)

(action-sequence (graph-search *galaxy-M35* *breadth-first*)) ;->
;;; (#S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN MALLORY :FINAL KATRIL :COST 10)
;;; #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN KATRIL :FINAL DAVION :COST 9)
;;; #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN DAVION :FINAL PROSERPINA :COST 5)
;;; #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN PROSERPINA :FINAL SIRTIS :COST 12))

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

12. Ejercicio 11

1. **¿Por qué se ha realizado este diseño para resolver el problema de búsqueda?:**

   Se ha realizado este diseño porque aporta una generalización cuantiosa sobre los problemas de búsqueda en general, no sólo aplicado a nuestro caso de la galaxia. La estructura `problem` permite la defición de un grafo en el que los nodos pueden ser definidos de manera arbitraria siempre y cuando tengan una heurística definida, y además ofrece herramientas para saber: si un nodo cumple con los requisitos de ser una meta; si dos nodos (caminos) corresponden a un estado repetido dentro del grafo; y cuáles son los procesos que nos llevan de unos nodos a otros.

   Con este mecanismo se puede replicar cualquier tipo de grafo imaginable siempre y cuando se componga de estados y de enlaces entre los mismos, como es el caso de nuestra galaxia, en la que, incluso, existen dos tipos de enlaces entre estados.

   1.1. **¿Qué ventajas aporta?**

   Todas las mencionadas en los párrafos anteriores, además de una flexibilidad enorme dada la gran encapsulación y división de las funciones. Restricciones como las de los planetas por los que se ha de pasar obligatoriamente o los planetas por los que está prohibido viajar a través de agujeros de gusano son posibles gracias a estas características, dado que sólo han de realizarse pequeños cambios a funciones muy concretas para hacer posible su implementación (por ejemplo, la restricción sobre los planetas de prohibido acceso se realiza simplemente a través de una condición en la función `navigate`).

1.2. **¿Por qué se han utilizado funciones lambda para especificar el test objetivo, la heurística y los operadores del problema?**

Principalmente por dos motivos:

- En primer lugar, para que dichas utilidades sean lo más flexibles posible, permitiendo la creación de funciones específicas para la resolución de diferentes problemas o la fácil modificación de las mismas en caso de que se quisiera modificar ligeramente su funcionalidad. De esta manera no tenemos que crear funciones específicas para cada problema, sino que podemos utilizarlas de forma flexible en cualquier situación.
- En segundo lugar, para hacer uso de parámetros como `*planets-forbidden*`, cuyo uso en las funciones base es poco recomendable dado que disminuye mucho la abstracción y generalidad que en un principio queremos de ellas.

2. **Sabiendo que en cada nodo de búsqueda hay un campo "parent", que proporciona una referencia al nodo a partir del cual se ha generado el actual ¿es eficiente el uso de memoria?**

Dado un camino expresado a través de un nodo (por ejemplo, el camino Mallory - Katril - Davion estaría representado a través de un nodo con hoja Davion, padre Katril y raíz Mallory), todos los posibles caminos que surgen de su expansión contendrán una referencia al mismo, por lo tanto en ningún momento se está generando más memoria que la estrictamente necesaria.

Puede ocurrir que haya dos nodos con el mismo estado y pensemos que se está gastando memoria por esta repetición; pero es una sensación falsa dado que dos caminos distintos contienen información sobre los costes distinta, y por tanto a efectos prácticos constituyen dos estados diferentes que no suponen ninguna repetición de información en la memoria.

Lo único que puede inducir a un uso poco eficiente de la memoria es el uso de la lista de nodos cerrados en el algoritmo para la implementación de las búsquedas del ejercicio 8, dado que estamos añadiendo continuamente caminos que provocan que los correspondientes a estados iguales se queden obsoletos en la lista sin ser eliminados; pero, realmente, las molestias que habría que tomarse para eliminar estos estados obsoletos no compensan en absoluto lo que se gana en memoria.

3. **¿Cuál es la complejidad espacial del algoritmo implementado?**

En el peor de los casos, cuando hay que mantener todos los nodos que se crean en la lista de cerrados, hay que tener en cuenta que en cada iteración (expansión) vamos a generar un número determinado de nodos, al cual vamos a denominar como **factor de ramificación** y vamos a identificar con b. ¿Cuántas veces vamos a realizar expansiones? En el peor de los casos, cuando se llegue a la solución (que será óptima dado que estamos usando el algoritmo A* sin estados repetidos con una heurística monótona) habrá sido a través del mayor número de pasos posible, es decir, los costes de cada paso habrán sido lo más pequeños posible. Denominando al coste óptimo como C^*, al mínimo coste ϵ y al número de pasos máximo como $d = \frac{C^*}{\epsilon}$, la complejidad espacial del algoritmo pasa a ser $O(b^d)$; en definitiva, se gasta mucha memoria para almacenar la información de todos los nodos.

4. **¿Cuál es la complejidad temporal del algoritmo?**

Como se ha mencionado en el apartado anterior, el peor de los casos se da cuando hay que realizar expansiones a cada paso, por lo que se puede hacer la identificación de que por cada nodo generado se realiza una iteración de la operación básica, que en nuestro caso es la generación de un nuevo nodo expandido a través de otro. Por tanto, la complejidad temporal es la misma que la espacial, $O(b^d)$.

5. **Indicad qué partes del código se modificarían para limitar el número de veces que se puede utilizar la acción "navegar por agujeros de gusano" (bidireccionales).**

Para realizar esta tarea hay que tener en cuenta dos necesidades:

- **Conocimiento del número de veces que se han tomado agujeros de gusano.** Para ello, habría que añadir un parámetro a la estructura `node` inicializado a 0 y que se fuera incrementando cada vez que se utiliza un agujero de gusano.
- **Comprobación para no rebasar el límite.** Para ello, habría que añadir un parámetro extra a la función `navigate`, un valor numérico que se comparara con el campo añadido a la estructura `node`. Si dicho parámetro fuera `NIL` no se realizaría la comparación, de manera que no le pusiéramos límite a las acciones a través de agujeros blancos.