PRÁCTICA 2

María Barroso Honrubia
Blanca Abella Miravet
Pareja 06

Inteligencia Artificial
Grupo 2301

1. Modelización del problema

EJERCICIO 1

Codificación de una función que calcula el valor de la heurística en el estado actual. Dicho valor es el coste de ir desde el estado actual a Sirtis.

Batería de ejemplos

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

Pseudocódigo

```
Entrada:
        state: estado actual
        sensors: lista de pares (estado coste)
Salida:
        heurística en el estado actual o nil si no tiene coste o

si sensors == null
        evalúa a nil
estado = first (first (sensors))
coste =  second (first (sensors))

si state == estado
        evalúa al coste de dicho estado
si no
        f-h-galaxy state (rest (sensors))
```

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)
          (if (null sensors) ; caso de error, devuelve nil
                nil
          (let ((estado (first (first sensors))) ; estado = first (first sensors)
                (coste (second (first sensors)))) ; coste =  second (first sensors)
          (if (equal state estado) ; si state == estado
                coste
```

(f-h-galaxy state (rest sensors)))))) ; si no, llamada recursiva a f-h-galaxy con sensor sin el primer par

---

EJERCICIO 2

Codificación de dos funciones operadores que devuelven la lista de acciones que se pueden efectuar a partir de un estado dado, utilizando en un caso agujeros blancos, y en otro agujeros de gusano.
En el caso de agujeros de gusano, no se puede acceder a los planetas pertenecientes a *planets-forbidden*.

Batería de ejemplos

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

Navigate-white-hole

```
(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))
```

Pseudocódigo

Navigate-worm-hole

Entrada:
- state: estado actual
- holes: Listas constantes de tripletes formados por planeta origen, planeta destino y gasto energético (coste)
- planets-forbidden: planetas prohibidos posibles en worm-holes, o nil

Salida:
- lista de acciones permitidas

Para cada par x (planeta-destino coste) devuelto por la función get-actions
- construimos acción tal que
  - name = NAVIGATE-WORM-HOLE
  - origin = state
  - final = (first x)
  - cost = (second x)

Navigate-white-hole

Entrada:
- state: estado actual
- holes: Listas constantes de tripletes formados por planeta origen, planeta destino y gasto energético (coste)

Salida:

lista de acciones permitidas

Para cada par x (planeta-destino coste) devuelto por la función get-actions
construimos acción tal que
name = NAVIGATE-WHITE-HOLE
origin = state
final = (first x)
cost = (second x)

Código

Funciones auxiliares:

```
;; get-actions
;; función que devuelve la lista de pares (planeta-destino coste) de acciones permitidas a partir de un estado a través de los distintos agujeros existentes
;; Input
;;              state: estado actual
;;              holes: Listas constantes de tripletes formados por planeta origen,
;;    planeta destino y gasto energético (coste)
;;              planets-forbidden: planetas prohibidos posibles en worm-holes, o nil
;; Return
;;              lista de acciones permitidas

(defun get-actions (state holes planets-forbidden)
        (unless (null holes) ; si holes == null, evalua a nil
        (let ((planeta-origen (caar holes))
                (planeta-destino (cadar holes))
                (dest-cost (cdar holes)))
        (if (and (equal state planeta-origen) ; si state == planeta origen
                (not (member planeta-destino planets-forbidden :test #'equal))) ; si el planeta destino no pertenece a planets-forbidden
                (cons dest-cost (get-actions state (rest holes) planets-forbidden)) ; formamos lista de pares (planeta-destino coste)
                (get-actions state (rest holes) planets-forbidden))))) ; otro caso, llamada recursiva con el resto de tripletas de holes
```

```
;; navigate
;; función que construye una lista de acciones que se pueden efectuar a partir de un estado a través agujeros blancos o negros
;; Input
;;              state: estado actual
;;              holes: Listas constantes de tripletes formados por planeta origen,
;;              planeta destino y gasto energético (coste)
;;              planets-forbidden: planetas prohibidos posibles en worm-holes, o nil
;;              name: nombre de la accion a construir
;; Return
;;              lista de acciones permitidas

(defun navigate (state holes planets-forbidden name)
  (mapcar #'(lambda(x) (make-action :name name
                                    :origin state
```

```
                    :final (first x)
                    :cost (second x))) (get-actions state holes planets-forbidden)))
```

Funciones principales:

```
;; navigate-white-hole
;; Devuelve la lista de acciones que se pueden efectuar a partir de un estado
;; a traves agujeros blancos
;;
;;  Input:
;;    state: the current state (vis. the planet we are on)
;;    white-holes: Listas constantes de tripletes formados por planeta origen,
;;     planeta destino y gasto energético (coste)
;;
;;
;;  Returns: la lista de acciones que se pueden efectuar a partir de un estado
;; a traves agujeros blancos

(defun navigate-white-hole (state white-holes)
   (navigate state white-holes NIL 'NAVIGATE-WHITE-HOLE))
```

```
;; navigate-worm-hole
;; Devuelve la lista de acciones que se pueden efectuar a partir de un estado
;; a traves agujeros de gusano
;;
;;  Input:
;;    state: the current state (vis. the planet we are on)
;;    worm-holes: Listas constantes de tripletes formados por planeta origen,
;;     planeta destino y gasto energético (coste)
;;
;;
;;  Returns: la lista de acciones que se pueden efectuar a partir de un estado
;; a traves agujeros de gusano

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

EJERCICIO 3A

Codificación de una función que comprueba si se ha alcanzado el objetivo.

Batería de ejemplos

```
(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

Pseudocódigo

Entrada:
node: the current node
planets-destination: lista de planetas destino posibles
planets-mandatory: planetas obligatorios de visitar para que se cumpla el objetivo
Salida:
T si se ha cumplido el objetivo, NIL en caso contrario

si el estado de node no pertenece a planets-destination
evalua a nil
si no,
list-of-parents = parents (node)
si algun y de planets-mandatory no pertenece a list-of-parents
evalua a nil
en otro caso,
evalua a t

Código

Funciones auxiliares

```
;; parents
;; funcion que devuelve la lista de padres de un nodo dado
;; Input
;;              node: nodo
;; Returns
;;              lista de padres de un nodo, o nil si no tiene

(defun parents (node)
   (unless(or (null node) (null (node-parent node))) ; si node == null o el nodo padre de
node == null, devuelve nil
     (cons (node-state (node-parent node)) (parents (node-parent node))))) ; formamos lista
con el estado del nodo padre
```

```
;; all-visited
;; funcion que comprueba si han sido visitados todos los planetas obligatorios a partir de la
lista de padres de un nodo
;; Input
;;              parents: lista de padres de un nodo
;;              planets-mandatory: planetas obligartorios de visitar
;; Returns
;;              T si se han visitado todos los planetas obligatorios, nil en caso contrario

(defun all-visited (parents planets-mandatory)
  (not(some #'null(mapcar #'(lambda(x) (member x parents :test #'equal))
planets-mandatory))))
```

Funcion principal

```
;; f-goal-test-galaxy
```

```
;; Comprueba si se ha alcanzado el objetivo
;;
;;  Input:
;;    node: the current node
;;    planets-destination: lista de planetas destino posibles
;;          planets-mandatory: planetas obligatorios de visitar para que se cumpla el objetivo
;;
;;  Returns: T si se ha alcanzado el objetivo, NILs en caso contrario

(defun f-goal-test-galaxy (node planets-destination planets-mandatory)
  (unless (not (member (node-state node) planets-destination)) ; si el estado del nodo
actual no pertenece a la lista de planetas destino, evalua a nil
    (all-visited (parents node) planets-mandatory))) ; comprueba que los padres del nodo
pertenecen a los planetas obligatorios
```

EJERCICIO 3B

Codificación de una función que comprueba si dos para dos nodos son iguales de acuerdo con su estado de búsqueda.

Batería de ejemplos

```
(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
```

Pseudocódigo

Entrada:

node-1: nodo a comparar
node-2: nodo a comparar
planets-mandatory: planetas obligatorios de visitar para que se cumpla el objetivo

Salida:

T si los dos nodos son iguales de acuerdo con su estado de búsqueda, NIL en caso contrario

```
si el estado del node-1 == estado del node-2
        restantes-1 = restantes ((visited node-1) planets-mandatory)
        restantes-2 = restantes ((visited node-2) planets-mandatory)
        si  restantes-1 ==  restantes-2
                evalua a t
        si no,
                evalua a nil
si no,
        evalua a nil
```

Código

Funciones auxiliares

```
;; equal-lists
;; funcion que comprueba que dos listas son iguales
;; Input
;;              l1: lista a comparar
;;              l2: lista a comparar
;; Returns:
;;              t si son iguales, nil en caso contrario

(defun equal-lists (l1 l2)
        (and (null (set-difference l1 l2)) ; si la lista de elementos de l1 que no aparecen en la l2 es nil
                 (null (set-difference l2 l1)))) ; si la lista de elementos de l2 que no aparecen en la l1 tambien es nil, las dos listas son iguales
```

```
;; visited
;; funcion que construye una lista de planetas visitados a partir de un nodo
;; Input
;;              node: nodo actual
;; Returns:
;;              lista de planetas visitados

(defun visited (node)
        (unless (null node)
           (if (null (node-parent node)) ; si el nodo no tiene padre
                (list (node-state node)) ; formamos lista con el estado del nodo actual
           (cons (node-state node) (visited (node-parent node))))); cons del estado del nodo actual y sus antecesores
```

```
;; restantes
;; funcion que devuelve la lista de planetas obligatorios que aun no se han visitado
;; Input
;;              visited: planetas visitados
;;              planets-mandatory: planetas obligatorios
;; Returns
;;              planetas obligatorios que aun no se han visitado

(defun restantes (visited planets-mandatory)
        (remove-if #'(lambda(x) (member x visited :test #'equal)) planets-mandatory))
```

Función principal

```
;; f-search-state-equal-galaxy
;; Comprueba si dos para dos nodos son iguales de acuerdo con su estado de búsqueda.
;;
;;  Input:
;;    node-1: nodo a comparar
;;    node-2: nodo a comparar
;;          planets-mandatory: planetas obligatorios de visitar para que se cumpla el objetivo
;;
```

;; Returns: T si los dos nodos son iguales de acuerdo con su estado de búsqueda, nil en caso contrario

```
(defun f-search-state-equal-galaxy (node-1 node-2 &optional planets-mandatory)
  (if (equal (node-state node-1) (node-state node-2)) ; si ambos nodos tienen el mismo estado
        (equal-lists (restantes (visited node-1) planets-mandatory) ; comprueba que la lista de planetas restantes son iguales para ambos nodos
                                    (restantes (visited node-2) planets-mandatory))
        nil))
```

---

2. Formalización del problema

EJERCICIO 4

Inicialización del valor de una estructura denominada *galaxy-M35* que representa el problema bajo estudio.

```
(defparameter *galaxy-M35*
  (make-problem
   :states             *planets*
   :initial-state      *planet-origin*
   :f-h                #'(lambda (state) (f-h-galaxy state *sensors*))
   :f-goal-test        #'(lambda (node) (f-goal-test-galaxy node *planets-destination* *planets-mandatory*))
   :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*)))))
```

EJERCICIO 5

Codificación de una función de expansión de nodos. Dado un nodo, esta función crea una lista de nodos correspondientes a un estado que se puede alcanzar directamente desde el estado del nodo dado.

Batería de ejemplos

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

(expand-node node-00 *galaxy-M35*)  ;-->
        ;;;(#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 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)
;;; #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))
```

Pseudocódigo

```
Entrada:
        node: nodo a expandir
        problem: problema bajo estudio
Salida:
        lista de nodos accesibles por node

actions = union-actions ((node-state node) problem)
lista = ()
Para cada action de actions
        gval = (node-g node) + (action-cost action)
        hval = f-h-galaxy (action-final action)
        new-node = make-node tal que
                state = (action-final action)
                parent = node
                action = action
                depth = (node-depth node) + 1
                g = gval
                h = hval
```

f = gval + hval
lista += new-node

Código

Función auxiliar

```
;; union-actions
;; funcion que crea una lista con las acciones permitidas utilizando agujeros blancos y de gusano
;; Input
;;              state: estado actual
;;              problem: problema bajo estudio
;; Returns
;;              lista de acciones permitidas por cada operador

(defun union-actions (state problem)
        (append (funcall (first (problem-operators problem)) state)
                        (funcall (second (problem-operators problem)) state)))
```

Función principal

```
;; expand-node
;;  función de expansión de nodos
;;
;;  Input:
;;    node: nodo a expandir
;;    problem: problema bajo estudio
;;
;;  Returns: lista de nodos correspondientes a un estado que se puede alcanzar directamente desde el estado del nodo dado

(defun expand-node (node problem)
        (mapcar #'(lambda(action)
                        (let ((gval (+ (node-g node) (action-cost action)))
                                (hval (funcall (problem-f-h problem) (action-final action))))
                                        (make-node :state (action-final action)
                                                        :parent node
                                                        :action action
                                                        :depth (+ (node-depth node) 1)
                                                        :g gval
                                                        :h hval
                                                        :f (+ gval hval))))
        (union-actions (node-state node) problem)))
```

EJERCICIO 6

Gestión de nodos.

Escribe una función que inserte los nodos de una lista en una lista de nodo de manera que la lista esté ordenada

respeto al criterio de comparación de una estrategia dada. Se supone que la lista en que se insertan los nodos ya
esté ordenada respeto al criterio deseado, mientras que la lista que se inserta puede tener cualquier orden.

Batería de ejemplos

```
(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) )

(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-<=))

(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 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 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)
;;; #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 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 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 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 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)
;;; #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-WHITE-HOLE :ORIGIN PROSERPINA :FINAL SIRTIS
:COST 12)
;;;         :DEPTH 13   :G 22        :H 0   :F 22)
;;; #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 KENTARES
;;;         :PARENT NIL
;;;         :ACTION NIL
;;;         :DEPTH 2         :G 50    :H 0   :F 50))
```

```
(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)
```

Pseudocódigo

Entrada:

nodes: lista de nodos a insertar

lst-nodes: lista de nodos en la que insertar los nodos

strategy: estrategia que usaremos para insertar los nodos en la lista

Salida:

lista de nodos ordenada según la estrategia

ordenamos la lista lst-nodes original

si first nodes < first lst

lo insertamos en lst y volvemos a realizar con el resto de nodes y la nueva lst

si no:

el primer elemento sera el first de lst. colocamos en su posicion first de nodes y con la nueva lst, volvemos a hacer lo mismo con todos los elementos de nodes

Código

Función auxiliar

```
;;insert-nodes-strategy-aux (nodes lst-nodes strategy)
;;
;; Devuelve una lista de nodos ordenados segun una estrategia dada.
;;
;;  Input:
;;         nodes: nodos a insertar en la lista
;;         lst-nodes: Lista ordenada donde insertar los nodos.
;;         strategy: estrategia que utilizar para insertar los nodos
```

```
;;
;;  Returns: la lista de nodos ordenados según la estrategia

(defun insert-nodes-strategy-aux (nodes lst-nodes strategy)
  (cond ((null nodes)
         lst-nodes)
        ((null lst-nodes)
        (insert-nodes-strategy-aux (rest nodes) (list (first nodes)) strategy)) ;; si la lista
inicial esta vacia, ordenamos los nodos
        ((null (rest nodes)) ;; solo hay un nodo que insertar
        (if (funcall (strategy-node-compare-p strategy) (first nodes) (first lst-nodes))
        (cons (first nodes) lst-nodes) ;; si el primer nodo es menor que el primero de la lista
        (cons (first lst-nodes) (insert-nodes-strategy-aux (list (first nodes)) (rest lst-nodes)
strategy)))) ;; si no es menor que el primero, colocamos el primer elemento de la lista al
principio y seguimos con el resto para ver donde insertarlo
        ;;si hay mas de un nodo que insertar
        ((if (funcall (strategy-node-compare-p strategy) (first nodes) (first lst-nodes))
        ;; si el primer nodo de nodes es menor que el de la lista
        (insert-nodes-strategy-aux (rest nodes) (cons (first nodes) lst-nodes) strategy);;
repetimos el procedimiento con el resto de nodos y la lista con el primer nodo insertado al
principio
        (insert-nodes-strategy-aux (rest nodes) (cons (first lst-nodes)
(insert-nodes-strategy-aux (list (first nodes)) (rest lst-nodes) strategy))
strategy))))) ;;repetimos el procedimiento con el resto de nodos y con la lista ordenada con
el primer nodo insertado segun la estrategia.
```

Función principal

```
;; insert-nodes-strategy (nodes lst-nodes strategy)
;;
;; Devuelve una lista de nodos ordenados según una estrategia dada.
;;
;;  Input:
;;        nodes: nodos a insertar en la lista
;;        lst-nodes: Lista donde insertar los nodos.
;;        strategy: estrategia que utilizar para insertar los nodos
;;
;;  Returns: la lista de nodos ordenados según la estrategia

(defun insert-nodes-strategy (nodes lst-nodes strategy)
  ;; llama a la auxiliar habiendo ordenado previamente lst-nodes con ayuda de la auxiliar
tambien
    (insert-nodes-strategy-aux nodes (insert-nodes-strategy-aux (rest lst-nodes) (list (first
lst-nodes)) strategy) strategy))
```

---

2. Búsquedas

EJERCICIO 7

Inicializa una variable global cuyo valor sea la estrategia para realizar la búsqueda A*:

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

*;;definimos la estrategia *A-star* con nombre A-star y con funcion de comparacion node-f-<=*

```
(defparameter *A-star*
  (make-strategy
         :name 'A-star
         :node-compare-p #'node-f-<=))
```

EJERCICIO 8

Codifica la función de búsqueda

Batería de ejemplos

```
(graph-search *galaxy-M35* *A-star*);->
;#S(NODE
;   :STATE SIRTIS
;   :PARENT #S(NODE
;         :STATE PROSERPINA
;         :PARENT #S(NODE
;               :STATE DAVION
;               :PARENT #S(NODE
;                     :STATE KATRIL
;                     :PARENT #S(NODE
;                           :STATE MALLORY
;                           :PARENT NIL
;                           :ACTION NIL
;                           :DEPTH 0
;                           :G 0
;                           :H 0
;                           :F 0)
;                     :ACTION #S(ACTION
;                           :NAME NAVIGATE-WHITE-HOLE
;                           :ORIGIN MALLORY
;                           :FINAL KATRIL
;                           :COST 10)
;                     :DEPTH 1
;                     :G 10
;                     :H 9
;                     :F 19)
;               :ACTION #S(ACTION
;                     :NAME NAVIGATE-WORM-HOLE
;                     :ORIGIN KATRIL
;                     :FINAL DAVION
;                     :COST 5)
;               :DEPTH 2
;               :G 15
;               :H 5
;               :F 20)
;         :ACTION #S(ACTION
```

```
;                   :NAME NAVIGATE-WHITE-HOLE
;                   :ORIGIN DAVION
;                   :FINAL PROSERPINA
;                   :COST 5)
;           :DEPTH 3
;           :G 20
;           :H 7
;           :F 27)
;   :ACTION #S(ACTION
;           :NAME NAVIGATE-WORM-HOLE
;           :ORIGIN PROSERPINA
;           :FINAL SIRTIS
;           :COST 9)
;   :DEPTH 4
;   :G 29
;   :H 0
;   :F 29)

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

Pseudocódigo

Entrada:

problem: problema a resolver
strategy: estrategia utilizada para resolverlo

Salida:

nodo solución del problema

Inicializar la lista de nodos open-nodes con el estado inicial
inicializar la lista de nodos closed-nodes con la lista vacía
recursión:

- si la lista open-nodes está vacía, terminar[no se han encontrado solución]
- extraer el primer nodo de la lista open-nodes
- 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
  * expandir el nodo e insertar los nodos generados en la lista open-nodes de acuerdo con la estrategia strategy.
  * incluir el nodo recién expandido al comienzo de la lista closed-nodes.
- Continuar la búsqueda eliminando el nodo considerado de la lista open-nodes.

Código

Funciones auxiliares

```
;;auxgs (node closed-nodes)
;;
```

```
;; input: node nodo
;;        closed-nodes lista de nodos explorados
;;
;; returns: nil si el nodo no esta en closed nodes o el nodo en closed-nodes es mayor
;;        T si node esta en closed-nodes y es menor

(defun auxgs (node closed-nodes)
   (if (not(member node closed-nodes)) ;; si node no esta en closed-nodes nil
         (node-g-<= node (first (member node closed-nodes))))) ;;compara el valor g de los
nodos
;;graph-search-aux (problem strategy opened closed)
;;
;; input: problem problema a resolver
;;        strategy estrategia a utilizar para resolverlo
;;        opened lista de nodos encontrados no explorados
;;        closed lista de nodos explorados
;;
;; returns nodo que cumple el test objetivo

(defun graph-search-aux (problem strategy opened closed)
  (cond ((null opened) nil)
        ;; si el primer nodo de opened es el nodo objetivo, lo devuelve y termina
        ((not(null (f-goal-test-galaxy (first opened) *planets-destination*
        *planets-mandatory*)))
                (first opened))
        ;;si el primer nodo de opened no esta en closed, o es menor que el de closed
         ((or(not (member (first opened) closed :test #'equal))(auxgs (first opened) closed))
          ;; expande el nodo e inserta sus hijos en opened
        (let ((open-nodes (insert-nodes-strategy (expand-node (first opened) problem)
        opened strategy))
          ;; mete el nodo en closed ya que ya ha sido explorado
        (closed-nodes (cons (first opened) closed)))
        ;; vuelve a realizar la busqueda
        (graph-search-aux problem strategy (rest open-nodes) closed-nodes)))
   ;; si el nodo es mayor que el que esta en closed, realiza la busqueda con los siguientes
nodos
   (t(graph-search-aux problem strategy (rest opened) closed))))
```

Funciones principales

```
;;graph-search (problem strategy)
;;
;; input: problem problema a resolver
;;        strategy estrategia utilizada para resolverlo
;;
;; returns: nodo que cumple el test objetivo


(defun graph-search (problem strategy)
  ;; llamamos open-nodes a la lista con el estado inicial del problema
```

```
   (let ((opened (list (make-node :state (problem-initial-state *galaxy-m35*)))))
        ;;iniciamos la busqueda con closed-nodes a nil ya que aun no habra ningun nodo explorado
        (graph-search-aux problem strategy open-nodes nil)))

;; a-star-search (problem)
;;
;; input: problem problema a resolver
;;
;; returns: solucion al problema aplicando la estrategia A-star
(defun a-star-search (problem)
        (graph-search problem *A-star*))
```

EJERCICIO 9

Ver el camino seguido y la secuencia de acciones.

Batería de ejemplos

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

(action-sequence (a-star-search *galaxy-M35*))
;;; ->
;;; (#S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN MALLORY :FINAL KATRIL :COST 10)
;;; #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))
```

Pseudocódigo

Entrada:
    node: nodo solución de un problema
Salida:
    camino seguido hasta llegar

lista con los estados de los nodos visitados. Aplicamos reverse para que la lista empiece desde el nodo inicial.

Entrada:
    node: nodo solución de un problema
Salida:
    secuencias de acciones realizadas para llegar

si node nil o action-node nil
    nil
sino
    lista con la accion del nodo y aplicamos sobre el nodo padre de manera recursiva.

Código

Funciones principales

```
;; solution-path (node)
;;
;; input: node nodo solucion de un problema
;;
;; returns: lista con los estados por los que ha pasado hasta llegar a el.


(defun solution-path (node)
  (reverse (visited node))) ;; visited va devolviendo los estados de los nodos visitados y reverse les da la vuelta


;; actions (node)
;;
;; input: node nodo
;;
;; returns: lista con las acciones inversas que se han realizado hasta llegar al nodo dado

(defun actions (node)
        (unless(or (null node) (null (node-action node)))
        (cons (node-action node) (actions (node-parent node)))))

;; action-sequence(node)
;;
;; input: node nodo
;;
;; returns: lista con las acciones que se han realizado hasta llegar al nodo dado

(defun action-sequence (node)
  (reverse (actions node))) ;; actions da las acciones de fin a principio y reverse les da la vuelta
```

EJERCICIO 10

Otras estrategias de búsqueda.

Batería de ejemplos

```
(solution-path (graph-search *galaxy-M35* *depth-first*))
;;; -> No encuentra solucion por bucle infinito

(solution-path (graph-search *galaxy-M35* *breadth-first*))
;; -> (MALLORY KATRIL DAVION PROSERPINA SIRTIS)
```

Funciones principales

```
;; definimos la estrategia depth-first

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

```
;;depth-first-node-compare-p (node-1 node-2)
;;
;; input: node-1 nodo
;;        node-2 nodo
;;
;;returns: T si depth de node-1 es mayor que depth de node-2, nil en caso contrario


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

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

;;breadth-first-node-compare-p (node-1 node-2)
;;
;; input: node-1 nodo
;;        node-2 nodo
;;
;;returns: T si depth de node-1 es menor que depth de node-2, nil en caso contrario

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

EJERCICIO 11

1. ¿Por qué se ha realizado este diseño para resolver el problema de búsqueda? En concreto,
1.1 ¿Qué ventajas aporta?

Con este diseño, podemos elegir el tipo de estrategia a utilizar para cada búsqueda, lo que puede ser muy beneficioso. Además, eliminamos estados repetidos a no ser que el nuevo estado sea mejor, lo que hace la búsqueda más eficiente.

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

Se han utilizado funciones lambda para poder acceder a las funciones que realizan la acción correspondiente como un atributo del problema, existiendo de esta forma una independencia entre dichos atributos y las funciones que habían sido implementadas con anterioridad.

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?

No es eficiente, pero es lo más cómodo a la hora de implementar las funciones y poder observar de forma sencilla su recorrido.

3 ¿Cuál es la complejidad espacial del algoritmo implementado?

Igual a la complejidad temporal (se mantienen todos los nodos en memoria).

4 ¿Cuál es la complejidad temporal del algoritmo?

Para el algoritmo de búsqueda con estrategia A*, la complejidad temporal es exponencial para un h arbitrario.

Su coste es O (b^d), tal que d = $\frac{C}{\varepsilon}$ **.**

C* = Coste óptimo;

ε = mínimo coste por acción

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).

Se modificaría la función de f-goal-test-galaxy, añadiéndole un parámetro que limite el número de veces que se ha utilizado un agujero de gusano. Se comprobaría que el número de acciones que utilicen agujeros de gusano sea menor que el límite establecido. En caso de que superarse dicho límite, no se admitiría como solución y el algoritmo buscaría un camino alternativo.