MEMORIA PRACTICA 2

Jose Manuel De Frutos Porras

Daniel Redondo Castilla

Pareja 9

Grupo 2301

EJERCICIO 1

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; f-goal-test-galaxy
;;; Comprueba si se ha alcanzado el objetivo.
;;;
;;; Entrada: state [estado actual]
;;;          planets-destination [lista de planetas objetivos]
;;;
;;; Salida: T [si es estado objetivo]
;;;         nil [caso contrario]
;;;
;;;

(defun f-goal-test-galaxy (state planets-destination)
  ;; Comprobamos que el estado esta en la lista de objetivos
  (not (null (member state planets-destination))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; ;Evaluacion
;;;  ;Casos Tipicos
;;;    (f-goal-test-galaxy 'Sirtis *planets-destination*) ;-> T
;;;    (f-goal-test-galaxy 'Avalon *planets-destination*) ;-> NIL
;;;  ;Caso Especiales
;;;    (f-goal-test-galaxy 'Urano *planets-destination*) ;-> NIL
;;;    (f-goal-test-galaxy nil *planets-destination*); -> NIL
;;;    (f-goal-test-galaxy 'Sirtis nil); -> NIL
```

EJERCICIO 2

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  f-h-galaxy
;;;   Devuelve la heuristica del estado actual
;;;
;;;
;;;  Entrada: state [estado actual]
;;;           sensors [diccionario de heuristica]
;;;
;;;  Salida: valor de la heuristica para el estado

(defun f-h-galaxy (state sensors)
  ;; Buscamos el estado en el diccionario de heuristica
  (second (assoc state sensors)))




;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; ;Evaluacion
;;;  ;Casos Tipicos
;;;    (f-h-galaxy 'Sirtis *sensors*) ;-> 0
;;;    (f-h-galaxy 'Avalon *sensors*) ;-> 5
```

;;; Casos Especiales
;;;   (f-h-galaxy nil *sensors*) ;-> nil

;;;   (f-h-galaxy 'Avalon nil) ;-> nil Buscamos la heuristica del estado en un diccionario vacio

EJERCICIO 3

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; navigate-white-hole (state white-holes)
;;;  Entrada: state [estado actual]
;;;          white-holes [Aristas]
;;;
;;;  Salida: Devuelve una lista de acciones posibles desde el estado actual.
;;;
;;;


(defun navigate-white-hole (state white-holes)
  (when white-holes ;Caso base ya no quedan mas agujeros blancos en la lista, luego ya no se pueden generar mas acciones
    (let ((current (first white-holes)))
     ;; Si el estado que buscamos es el nodo origen del agujero blanco creamos la accion correspondiente. Agujeros blancos bidireccionales
     (if(eql state (first current))
        (cons (make-action :name 'NAVIGATE-WHITE-HOLE
                    :origin state
                    :final (second current)
                    :cost (third current))(navigate-white-hole state (rest white-holes)))
      (navigate-white-hole state (rest white-holes)))))) ; En caso de no ser el estado el nodo origen del
                                                         ;  agujero blanco continuamos la busqueda




;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJEMPLOS
;;;
;;; ;GENERALES
;;;  (navigate-white-hole 'Katril *white-holes*) ; ->
;;; (#S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN KATRIL :FINAL DAVION :COST 2)
;;;  #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN KATRIL :FINAL MALLORY :COST 6))
;;;
;;;
;;; ;ESPECIFICOS
;;;   (navigate-white-hole 'Urano *white-holes*) ;-> NIL No hay agujero blanco con ese estado inicial
;;;   (navigate-white-hole 'Katril nil);-> nil
;;;   (navigate-white-hole nil *white-holes*);-> nil
;;;
;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  navigate-worm-hole
;;;
;;;
;;;  Entrada: state [estado actual]
;;;          worm-holes [Aristas]
;;;
;;;  Salida: Devuelve una lista de acciones posibles desde el estado actual.

(defun navigate-worm-hole (state worm-holes)
 (when worm-holes
  (let ((current (first worm-holes))) ; Analizamos el primer agujero de gusano de la lista
   ;;Tanto el destino como el origen pueden iguales a state ya que los agujeros de gusano son
   ;;bidireccionales.
   (if(or (eql state (first current)) (eql state (second current)))
      ;;Anadimos a la lista la action si no la hemos anadido antes(adjoin).
      (adjoin (make-action :name 'navigate-worm-hole
                           :origin state
                           :final (if(eql state (first current))
                                      (second current)
                                     (first current))
                           :cost (third current)) (navigate-worm-hole state (rest worm-holes)) :test 'equalp)
     ;;Equalp T si las dos estructuras tienen la misma impresion por pantalla.
     (navigate-worm-hole state (rest worm-holes))))) ;Recursion con el resto de agujeros de gusano de
                                                     ; la lista.



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; ;GENERALES
;;;  (navigate-worm-hole 'Katril *worm-holes*) ; ->
;;;   ;(#S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN KATRIL :FINAL DAVION :COST 1)
;;;   ; #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN KATRIL :FINAL MALLORY :COST 5)
;;;   ; #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN KATRIL :FINAL SIRTIS :COST 10))
;;;
;;; ;ESPECIFICOS
;;;   (navigate-worm-hole 'Urano *worm-holes*) ;-> NIL No hay agujero de gusano con ese
;;;                                                                    estado inicial
;;;   (navigate-worm-hole 'Katril nil);-> nil
;;;   (navigate-worm-hole nil *worm-holes*);-> nil
;;;
```

EJERCICIO 4

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Estrategia A*
;;;

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

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

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Estategia coste uniforme
;;; (para los ejemplos)

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

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

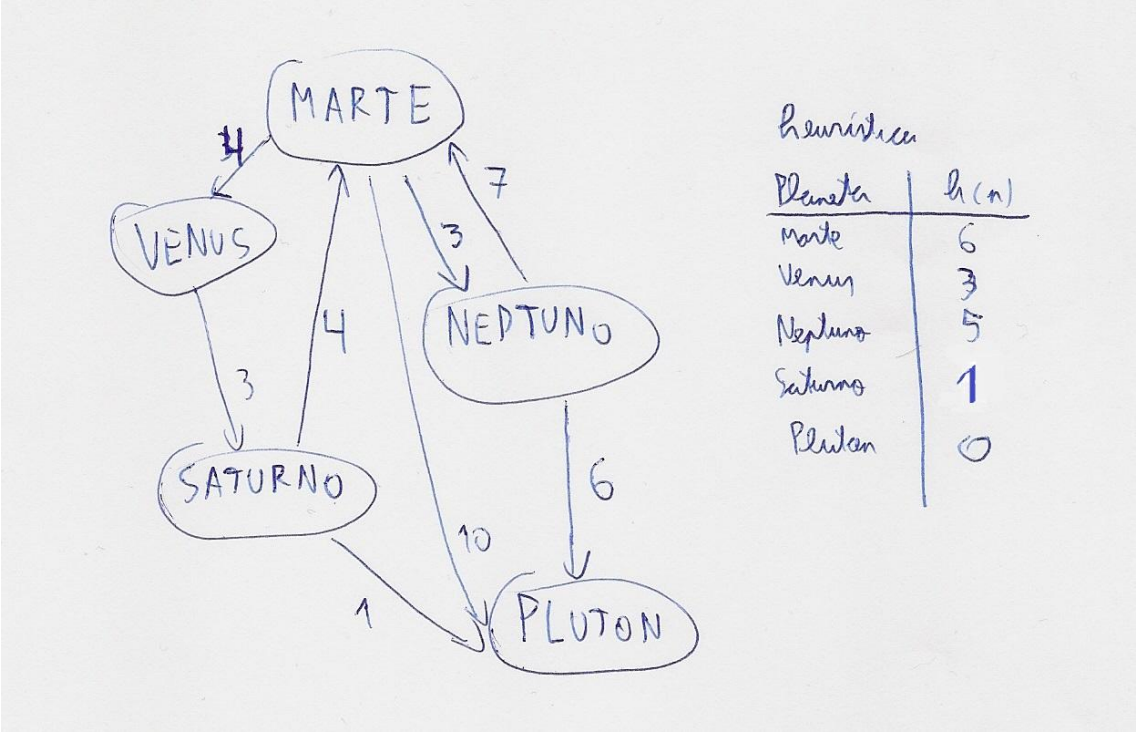
EJERCICIO 5

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Estructura definicion del problema
;;;

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

COMENTARIO: Hemos creado una estructura problema adicional, más pequeña, para debuguear más fácilmente las funciones más complejas.

```
(setf *planets-test* '(Marte Venus Neptuno Saturno Pluton))

(setf *jumps-test* '((Marte Venus 4) (Neptuno Marte 7) (Marte Pluton 10)
                     (Venus Saturno 3) (Saturno Pluton 1) (Saturno Marte 4)
                     (Marte Neptuno 3) (Neptuno Pluton 6)))

(setf *sensors-test*
  '((Marte 6) (Venus 3) (Neptuno 5) (Saturno 1) (Pluton 0)))

(setf *origin-test* 'Marte)

(setf *destinations-test* '(Pluton))
```

```
(setf *problem-test*
  (make-problem
   :states *planets-test*
   :initial-state *origin-test*
   :fn-goal-test (make-fn :name 'f-goal-test-galaxy
                          :lst-args *destinations-test*)
   :fn-h (make-fn :name 'f-h-galaxy
                  :lst-args (list *sensors-test*))
   :operators (list (make-fn :name 'navigate-white-hole
                             :lst-args (list *jumps-test*)))))
```



EJERCICIO 6

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; expand-node
;;; expande un nodo dado un problema. (El problema especifica la funcion la estrategia para expandir)
;;;
;;; Entrada: state [estado actual]
;;;          problem [problema de busqueda. Ver estructura problem]
;;;
;;; Salida: lista con los nodos generados al expandir el nodo state
;;;
```

```
(defun make-new-node (state lst-actions problem)
  (when lst-actions ; Caso base de la recursion. ya no quedan mas acciones en la lista. Por tanto ya no
                    ;podemos crear mas nodos.
    (let* ((action (first lst-actions))
           (node (action-final action))
           (heuristica (problem-fn-h problem))
           (valuenode-g (+  (node-g state) (action-cost action)))
           (valuenode-h (apply (fn-name heuristica) node (fn-lst-args heuristica))))

      ;;Anadimos a lista el nuevo nodo correspondiente a la accion procesada(la primera de la lista).
      (cons (make-node :state node ; nodo destino de la accion
                       :parent state ; nodo padre de la accion.
                       :action action ; accion que genero al nodo.
                       :depth (+ (node-depth state) 1) ;La profundidad es la del nodo padre +1
                       :g valuenode-g ; el g nuevo sera el g del nodo padre + el coste de la accion.
                       :h valuenode-h ; la heuristica del nuevo nodo;
                       :f (+ valuenode-g valuenode-h)) ; f + g del nuevo nodo.
            (make-new-node state (rest lst-actions) problem))))) ; Recursion con el resto de las acciones de la
                                                                 ; lista.

(defun generate-all-nodes (state operator problem)
  (when operator  ;Caso base de la recursion. Ya no quedan mas operadores devolvemos nil.
    (let (( current-op (first operator)))
      ;; generamos todos los nodos a partir de los operadores del problemas. Consideramos operador por
      ;; operador y hacemos recursion sobre ellos.
      (append (make-new-node state (apply (fn-name current-op) (node-state state) (fn-lst-args current-
op)) problem)
              (generate-all-nodes state (rest operator) problem))))) ; hacemos recursion sobre la lista de
                                                                     ; operadores.


(defun expand-node(state problem)
  (generate-all-nodes state (problem-operators problem) problem))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; ;Casos Tipicos
;;;   (setf node-00 (make-node :state 'Proserpina :depth 12 :g 10 :f 20))
;;;   (length(setf lst-node-00 (expand-node node-00 *galaxy-M35*))) ; -> se generan 7 nodos hijos.
;;;
;;; ;Casos Especiales
;;;
;;; ;Expandir un nodo que no tiene sucesores. Proserpina no tiene sucesor
;;;
;;;
;;; (setf *white-holes-expand* '((Avalon Mallory 2) (Avalon Proserpina 2)
;;;   (Davion Proserpina 4) (Davion Sirtis 1)
;;;   (Katril Davion 2) (Katril Mallory 6)
;;;   (Kentares Avalon 3) (Kentares Proserpina 2) (Kentares Katril 2)
;;;   (Mallory Katril 6) (Mallory Proserpina 7)))
;;;
```

```
;;; (setf *galaxy-M35-expand*
;;;     (make-problem
;;;   :states *planets*
;;;   :initial-state *planet-origin*
;;;   :fn-goal-test (make-fn :name 'f-goal-test-galaxy
;;;           :lst-args *planets-destination*)
;;;   :fn-h (make-fn :name 'f-h-galaxy
;;;         :lst-args (list *sensors*))
;;;   :operators (list (make-fn :name 'navigate-white-hole
;;;                :lst-args (list *white-holes-expand*)))))
;;;
;;; (expand-node node-00 *galaxy-M35*) ; -> nil
```

EJERCICIO 7

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Insert a list of nodes into another list of nodes
;;;
;;; (defun insert-nodes-strategy (nodes lst-nodes strategy)...)

;;;
;;; ENTRADA
;;;    nodes: lista de nodos a insertar
;;;    lst-nodes: lista de nodos donde se va a insertar
;;;    strategy: estrategia segun la cual estan ordenados los nodos
;;;
;;; SALIDA
;;;    lista de todos los nodos ordenada segun la estrategia

;;;
;;;PSEUDOCODIGO
;;; si nodes vacia -> devolver lst-nodes
;;; en otro caso
;;;    coger el primer nodo de nodes
;;;      new-lst-nodes <- recorrer la lista hasta encontrar una posicion donde insertar nuevo nodo de acuerdo a estrategia
;;;    insert-nodes-strategy (resto nodes, new-lst-nodes, strategy)

(defun insert-nodes-strategy (nodes lst-nodes strategy)
  ;;caso base: si no hay mas nodos a insertar devolvemos la lista ya ordenada
  (if (null nodes) lst-nodes
    ;;insertamos el nodo en la lista segun la estrategia y llamamos recursivamente con el resto de nodos
    (insert-nodes-strategy (rest nodes) (insert-a-node (first nodes) lst-nodes strategy) strategy)))

(defun insert-a-node (node lst-nodes strategy)
  (if (not(null lst-nodes))
      (if (funcall (strategy-node-compare-p strategy) node (first lst-nodes))
          ;;si el nodo va en esta posicion lo colocamos
          (cons node lst-nodes)
        ;;en otro caso, intentamos en la siguiente posicion
        (cons (first lst-nodes) (insert-a-node node (rest lst-nodes) strategy)))
    ;;si llegamos al final de la lista lo colocamos ahi
    (list node)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJEMPLOS
;;;
;;; CASOS TIPICOS
;;;(setf node-00
;;;  (make-node :state 'Proserpina :depth 12 :g 10 :f 20) )
;;;(setf node-01
;;;  (make-node :state 'Avalon :depth 0 :g 0 :f 0) )
;;;(setf node-02
;;;  (make-node :state 'Kentares :depth 2 :g 50 :f 50) )
;;;(setf lst-nodes-00
;;;  (expand-node node-00 *galaxy-M35*))

;;;(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 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 1) :depth 13
;;;        :g 11 :h 4 :f 15)
;;; #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 2) :depth 13
;;;        :g 12 :h 5 :f 17)
;;; #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 4) :depth 13
;;;        :g 14 :h 1 :f 15)
;;; #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 6) :depth 13
;;;        :g 16 :h 7 :f 23)
;;; #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 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 7) :depth 13
;;;        :g 17 :h 7 :f 24)
;;; #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 nil :action nil :depth 2
;;;        :g 50 :h 0 :f 50)) ; -> Ejemplo del enunciado
;;;
;;;
```

```
;;; CASOS ESPECIALES
;;; (insert-nodes-strategy (list node-00 node-01 node-02) nil  *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 KENTARES :PARENT NIL :ACTION NIL :DEPTH 2 :G 50 :H 0 :F 50)); solo ordena los nodos
;;;
;;; (insert-nodes-strategy nil lst-nodes-00 *uniform-cost*); -> devuelve lst-nodes-00
```

EJERCICIO 8

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Realiza la búsqueda para el problema dado utilizando una estrategia
;;;
;;; open: lista de nodos generados, pero no explorados
;;; closed: lista de nodos generados y explorados
;;; strategy: estrategia de búsqueda implementada como una ordenación de la lista open-nodes
;;; goal-test: test objetivo (predicado que evalúa a T si un nodo cumple la condición de ser meta)
;;;
;;; Entrada : problema [estructura definición del problema]
;;;               strategy [estrategia de acuerdo a la cual se ordenan los nodos y se decide si explorar un
;;;                         nodo ya explorado anteriormente]
;;; Evalúa:
;;; Si no hay solución: NIL
;;; Si hay solución: un nodo que cumple el test objetivo
;;;
;;; PSEUDOCODIGO
;;; (defun graph-search (problem strategy)
;;;   inicializar la lista de nodos open con el estado inicial
;;;   inicializar la lista de nodos closed con la lista vacía
;;;   recursión:
;;;     o si la lista open-nodes está vacía, terminar [no se han encontrado solución]
;;;     o extraer el primer nodo de la lista open-nodes
;;;     o 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 es inferior
;;;       al que está en closed-nodes de acuerdo con la estrategia strategy
;;;             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.
;;;     o Continuar la búsqueda eliminando el nodo considerado de la lista open-nodes.

(defun graph-search (problem strategy)
  (let ((heuristica (apply (fn-name (problem-fn-h problem)) (problem-initial-state problem) (fn-lst-args (problem-fn-h problem)))))
    (graph-search-with-lists problem strategy  (list(make-node :state (problem-initial-state problem)
                                                               :parent nil
                                                               :action nil
                                                               :depth 0
                                                               :g 0
                                                               :h heuristica
                                                               :f heuristica))
                             nil)))
```

```
(defun graph-search-with-lists (problem strategy open closed)
  (when open ;caso base de la recursion
    (let ((current (first open)) (goal-test (problem-fn-goal-test problem)))
      ;Comprobamos si el nodo es meta
      (if (funcall (fn-name goal-test) (node-state current) (fn-lst-args goal-test))
          ;Si es meta lo devolvemos
          current
        ;Si no hay cerrados no buscamos en cerrados
        (if (null closed)
            (graph-search-with-lists
              problem
              strategy
              (insert-nodes-strategy (expand-node current problem) (rest open) strategy)
              (append (list current) closed))
          ;Buscamos si el nodo esta en la lista de cerrados
          (let ((search (member-if (lambda (x) (eql (node-state current) (node-state x))) closed)))
            (if (or (null search) (funcall (strategy-node-compare-p strategy) current (first search)))
                ;si no esta en cerrados o es inferior segun estrategia, expandimos nodo y lo aniadimos a cerrados
                (graph-search-with-lists
                  problem
                  strategy
                  (insert-nodes-strategy (expand-node current problem) (rest open) strategy)
                  (append (list current) closed))
              ;si esta lo eliminamos de abiertos
              (graph-search-with-lists problem strategy (rest open) closed)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJEMPLOS
;;;
;;; CASOS TIPICOS
;;; (graph-search *problem-test* *A-star*) ;Ejemplo mas corto ;->
;;; #S(NODE :STATE PLUTON
;;;     :PARENT
;;;     #S(NODE :STATE SATURNO :PARENT #S(NODE # # # # # # #) :ACTION #S(ACTION # # # #) :DEPTH 2 :G 7 :H 1 :F 8)
;;;      :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN SATURNO :FINAL PLUTON :COST 1) :DEPTH 3 :G 8 :H 0 :F 8)
;;;
;;; (graph-search *galaxy-M35* *A-star*) ;Ejemplo del enunciado ;->
;;; #S(NODE :STATE SIRTIS
;;;     :PARENT
;;;     #S(NODE :STATE DAVION :PARENT #S(NODE # # # # # # #) :ACTION #S(ACTION # # # #) :DEPTH 2 :G 3 :H 1 :F 4)
;;;     :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN DAVION :FINAL SIRTIS :COST 1) :DEPTH 3 :G 4 :H 0 :F 4)
;;;
```

EJERCICIO 9

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; a-star-search
;;; Realiza la búsqueda A* para el problema dado
;;;
;;; Entrada: problema [definición del problema]
;;; Evalúa:
;;; Si no hay solución: NIL
;;; Si hay solución: un nodo que cumple el test objetivo
;;;
;;; Basta con implementar correctamente la estrategia (para expandir los nodos)
;;; caracteristica de A*. Con esta estrategia llamamos a la busqueda en grafo.
;;;

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


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJEMPLOS
;;;
;;; ;CASOS TIPICOS
;;;    (a-star-search *galaxy-M35*); ->
;;;       #S(NODE :STATE SIRTIS :PARENT #S(NODE :STATE DAVION
;;;       :PARENT #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 2)
;;;        :DEPTH 1
;;;        :G ...)
;;;        :ACTION #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN KATRIL :FINAL DAVION :COST 1)
;;;        :DEPTH 2
;;;        :G ...)
;;;       :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN DAVION :FINAL SIRTIS :COST 1)
;;;       :DEPTH 3
;;;       :G ...)
;;;
;;; ;CASOS ESPECIALES
;;;
;;; Definimos una lista de agujeros blancos pero donde no hay camino posible hasta Sirtis
;;;(setf *white-holes-a-star* '((Avalon Mallory 2) (Avalon Proserpina 2);;;  (Davion Proserpina 4)
;;;  (Katril Davion 2) (Katril Mallory 6)
;;;  (Kentares Avalon 3) (Kentares Proserpina 2) (Kentares Katril 2)
;;;  (Mallory Katril 6) (Mallory Proserpina 7)
;;;  (Proserpina Avalon 2) (Proserpina Mallory 7) (Proserpina Davion 4)
;;;  (Sirtis Proserpina 10) (Sirtis Davion 1)))
;;;
```

```
;;; (setf *galaxy-M35-a-star*
;;;    (make-problem
;;;  :states *planets*
;;;  :initial-state *planet-origin*
;;;  :fn-goal-test (make-fn :name 'f-goal-test-galaxy
;;;           :lst-args *planets-destination*)
;;;  :fn-h (make-fn :name 'f-h-galaxy
;;;        :lst-args (list *sensors*))
;;;  :operators (list (make-fn :name 'navigate-white-hole
;;;        :lst-args (list *white-holes-a-star*)))))
;;;
;;;    (a-star-search *galaxy-M35-a-star* ) ;->nil No hay camino hasta Sirtis usando solo los agujeros blancos definidos anteriormente.
;;;
```

EJERCICIO 10

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; graph-path
;;; función que muestra el camino seguido para llegar a un nodo
;;;
;;; Entrada: node[nodo para el que queremos encontrar el camino seguido]
;;; Evalúa:
;;;  Devuelve una lista con el camino para llegar al nodo.
;;;  nil si el nodo pasado no existe.
;;;
;;; PSEUDOCODIGO
;;; (defun graph-path(node)
;;;     inicializar lista de nodos que conforman el camino
;;;     Recursion:
;;;      if node es nil
;;;        devolver nil
;;;      anadir el nodo al inicio de la lista.
;;;      graph-path(padre de node)
;;;

(defun make-list-nodes(node)
 (when node
  ;; Vamos anadiendo los nodos padres del nodo original a la lista y asi obtenemos el camino hasta el
  ;; nodo objetivo
  (cons (node-state node) (make-list-nodes (node-parent node)))))

(defun graph-path(node)
 (reverse (make-list-nodes node))) ;Damos la vuelta a la lista por motivos de legibilidad.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJEMPLOS
;;;
;;; CASOS TIPICOS
;;;
;;;  (graph-path  (a-star-search *galaxy-M35*)); -> (KENTARES KATRIL DAVION SIRTIS) ;
;;;  Buscamos el camino mas corto desde el estado inicial del problema hasta el estado final que es
;;;;  alcanzable.
;;;
;;; CASOS ESPECIALES
;;;  (graph-path nil) ;-> NIL
;;;
```

```
;;;   ;Caso de nodo sin padre
;;;   (setf node-01
;;;     (make-node :state 'Avalon :parent nil :depth 0 :g 0 :f 0))
;;;
;;;   (graph-path node-01) ;-> (Avalon)
;;;
;;;
```

EJERCICIO 11

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; action-sequence
;;; función que muestra la secuencia de acciones para llegar a un nodo.
;;;
;;; Entrada: node[nodo para el que queremos encontrar el camino seguido]
;;;
;;; Evalúa:
;;;   Devuelve una lista de las acciones generadas para llegar al nodo pasado como argumento.
;;;   nil si el nodo pasado no existe.
;;;
;;; PSEUDOCODIGO
;;; (defun action-sequence(node)
;;;      inicializar lista de acciones que conforman el camino al nodo
;;;      Recursion:
;;;        if node es nil
;;;          devolver nil
;;;        anadir la accion correspondiente al nodo al inicio de la lista.
;;;        action-sequence(padre de node)
;;;
```

```
(defun make-list-actions(node)
  ;; Caso base de la recursion devolvemos nil, que es cuando ya hemos alcanzado el nodo inicial, que
  ;; genero el nodo objetivo pasado como argumento
  (when (node-parent node)
    ;; Vamos anadiendo las acciones correspondientes a cada nodo a la lista del camino hasta el nodo
    ;; objetivo.
    (cons (node-action node) (make-list-actions (node-parent node)))))


(defun action-sequence (node)
  (reverse (make-list-actions node))) ;Damos la vuelta a la lista por motivos de legibilidad.
```

COMENTARIO

Otra implementación seria usando la función definida en el apartado anterior. Creamos la lista de nodos que conforman el camino hasta node (nodo pasado como argumento) y para cada uno de estos nodos de la lista consideramos la acción que lo genera (estructura de nodo) y la añadimos a una lista de acciones inicialmente vacía.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJEMPLOS
;;;
;;; CASOS TIPICOS
;;;
;;;   (action-sequence (a-star-search *galaxy-M35*));->
;;;(#S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN KENTARES :FINAL KATRIL :COST 2)
;;;  #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN KATRIL :FINAL DAVION :COST 1)
;;;  #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN DAVION :FINAL SIRTIS :COST 1))
;;;
;;; CASOS ESPECIALES
;;;
;;;   ;Caso de nodo sin accion que lo genere
;;;   (setf node-01
;;;     (make-node :state 'Avalon :depth 0 :g 0 :f 0))
;;;
;;;   (action-sequence node-01) ;-> nil
;;;
;;;   ;Caso de nodo sin padre
;;;   (setf node-01
;;;     (make-node :state 'Avalon :parent nil :action  :depth 0 :g 0 :f 0))
;;;
;;;   (action-sequence node-01) ;->  nil
```

EJERCICIO 12

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Diseno de una estrategia de busqueda en profundidad.
;;;
;;; En busqueda en profundidad considerado dos nodos de la lista de abiertos y primero
;;; expandimos el que tiene mayor profundidad.

(defun depth-first-node-compare-p(node-1 node-2)
  (> (node-depth node-1) (node-depth node-2))) ;Comparamos la profundidad de los dos nodos


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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJEMPLOS
;;;
;;; CASOS TIPICOS
;;; (graph-path (graph-search *problem-test* *depth-first*)) ; -> (MARTE VENUS SATURNO PLUTON)
;;; (action-sequence (graph-search *problem-test* *depth-first*)) ; ->
;;;  (#S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN MARTE :FINAL VENUS :COST 4)
;;;; #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN VENUS :FINAL SATURNO :COST 3)
;;;  #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN SATURNO :FINAL PLUTON :COST 1))
;;;
```

```
;;; CASOS ESPECIALES
;;;
;;; (graph-path (graph-search *galaxy-M35* *depth-first*)) ; -> CUIDADO STACKOVERFLOW. Se genera
un blucle y nunca llegamos al estado objetivo.
;;;
;;; (setf *jumps-loop* '((Marte Venus 1) (Venus Neptuno 1) (Neptuno Saturno 1) (Saturno Marte 1)))
;;;
;;; (setf *problem-loop*
;;;   (make-problem
;;;    :states *planets-test*
;;;    :initial-state *origin-test*
;;;    :fn-goal-test (make-fn :name 'f-goal-test-galaxy
;;;                           :lst-args *destinations-test*)
;;;    :fn-h (make-fn :name 'f-h-galaxy
;;;                   :lst-args (list *sensors-test*))
;;;    :operators (list (make-fn :name 'navigate-white-hole
;;;                              :lst-args (list *jumps-loop*)))))
;;;
;;; (graph-path (graph-search *problem-loop* *depth-first*)) ;Comprobamos si para cuando hay un
;;; bucle
;;; No para, ya que por la definicion de la estrategia explora los nodos cada vez mas profundos

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Diseno de una estrategia de busqueda en anchura
;;;
;;; En busqueda en anchura expandimos primero los nodos que estan a menor profundidad

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

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJEMPLOS
;;;
;;; CASOS TIPICOS
;;;
;;; (graph-path (graph-search *galaxy-M35* *breadth-first*)) ; -> (KENTARES PROSERPINA SIRTIS)
;;; (action-sequence (graph-search *galaxy-M35* *breadth-first*)) ; ->
;;;    ;(#S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN KENTARES :FINAL PROSERPINA :COST 2)
;;;    ;#S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN PROSERPINA :FINAL SIRTIS :COST 7))
;;;
;;; (graph-path (graph-search *problem-test* *breadth-first*)) ;-> (MARTE PLUTON)
;;; (action-sequence (graph-search *problem-test* *breadth-first*)); ->
;;;       (#S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN MARTE :FINAL PLUTON :COST 10))
```