Práctica 2 Inteligencia Artificial

Pareja 09 (Lucía Asencio y Juan Riera)

Abril 2018

1 Modelización del problema

EJERCICIO 1: Evaluación del valor de la heurística

PSEUDOCÓDIGO

Entrada: state estado actual, sensors heurística
Salida: h(state)
Procesamiento:
Para cada elto-sensor:
 Si estado-elto-sensor == estado-actual:
  Devuelve h(estado-elto-sensor)

CÓDIGO

```
(defun f-h-galaxy (state sensors)
  (unless
    (null sensors)
    (if (equal (first (first sensors)) state)
      ;; Si el primer elt de la sublista es el state, devuelvo
      ;; el segundo elt de la sublista
      (second (first sensors))
      ;; Si no, sigo buscando en el resto de los sensores
      (f-h-galaxy state (rest sensors)))))

(f-h-galaxy 'Sirtis *sensors*) ;-> 0
(f-h-galaxy 'Avalon *sensors*) ;-> 15
(f-h-galaxy 'Earth *sensors*) ;-> NIL
(f-h-galaxy 'Proserpina *sensors*) ;-> 7
```

EJERCICIO 2: Operadores

PSEUDOCÓDIGO navigate-white-holes:

Entrada: el estado actual y la lista de agujeros blancos
Salida: la lista de acciones que se pueden realizar desde el estado actual a través e agujeros blancos
Procesamiento:
1) Elimino de la lista los agujeros blancos cuyo origen no sea mi estado actual
2) Recorro la lista resultado formando las acciones de recorrer todos los agujeros blancos de la lista. Es decir, para cada elemento de la lista, formo la acción resultado de recorrer ese elemento (que es un agujero blanco.

PSEUDOCÓDIGO navigate-worm-holes:

Entrada: el estado actual, la lista de agujeros de gusano y la lista de los planetas prohibidos **Salida:** la lista de acciones que se pueden realizar desde el estado actual a través e agujeros blancos **Procesamiento:**

1) Recorro toda la lista de agujeros de gusano elemento por elemento

2) Si uno de los dos planetas que une el agujero es el estado actual y el otro no pertenece a los planetas prohibidos, lo dejo en la lista y lo reordeno si es necesario de tal forma que quede como primer elemento de la estructura del agujero el estado actual, y el otro planeta quede en segundo lugar.

3) Si no se cumple esta codición elimino el elemento de la lista.

4) Vuelvo a recorrer toda la lista, por cada elemento formo una acción resultante de recorrer el agujero de gusano del primer elemento de la estructura de dicho agujero al segundo.

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN: Exercise 2 -- Navigation operators
;;

;;;;
;; Funcion auxiliar que a partir de un estado y una lista de agujeros genera acciones
;; resutantes de moverse del estado actual al segundo elemento de cada agujero
;; asumiendo que es el destinoy que se puede navegar a el. El argumento type
;; indicara el nombre que se le da a la accion (white hole o black hole)
;;;;
(defun navigate-holes (type state holes)
  (mapcar #'(lambda
              (x)
              (make-action ;; Construye la accion
                :name type ;; cuyo nombre es el tipo de accion
                       ;; (agujero blanco o de gusano)
                :origin (first x) ;; Su origen es el principio del agujero
                :final (second x) ;; Su destino es el final del agujero
                :cost (third x))) ;; Si coste es el coste del agujero
          holes))

;;;;
;; Devuelve una lista de acciones que se pueden realizar desde el estado actual hacia
;; los agujeros blancos de la lista
;;;;
(defun navigate-white-hole (state white-holes)
    (navigate-holes 'navigate-white-hole state ;; Agujero blanco
        (remove state       ;; Creamos una lista con los agujeros blancos
               white-holes  ;; que se pueden recorrer desde el nodo en el que estamos
               :test #'(lambda(x y)
               (not (eql x (first y)))))))
```

```
;;;;
;; Devuelve una lista con las acciones que se pueden realizar al atravesar agujeros
;; negros desde el estado actual. Descarta los panetas prohibidos.
;;;;
(defun navigate-worm-hole (state worm-holes planets-forbidden)
  (navigate-holes 'navigate-worm-hole state ;; Agujero de gusano
    (mapcan
      #'(lambda(y)       ;; Creamos una lista con los agujeros
                         ;; de gusano que se pueden recorrer desde
                         ;; el estado actual
          (cond
            ((eql (first y) state) ;; Si el primer elemento es el estado
                                   ;; actual, conservamos el orden
             (if
               (null (member (second y) ;; Si no pertenece a planets-forbidden
                             planets-forbidden))
               (list y)))
            ((eql (second y) state) ;; Si es el segundo, invertimos el orden
                                    ;; antes de mandarlo a la funcion navigate-holes
                                    ;; (esto es meramente para simplificar
                                    ;; navigate-holes)
             (if (null (member (first y) ;; Si no pertenece a planets-forbidden
                               planets-forbidden))
               (list (list state
                           (first y)
                           (third y)))))))
      worm-holes)))
```

COMENTARIOS

Utilizamos aquí una función auxiliar llamada *navigate-holes* que recibe como único argumento una lista de agujeros sin importar si son blancos o de gusano, y recorre toda la lista. Para cada elemento crea una estructura acción que tiene como origen el primer elemento del agujero, como destino el segundo y coste el tercero. Va susituyendo cada elemento por la acción que crea con él.

EJERCICIO 3A: Goal-test

PSEUDOCÓDIGO

Entrada: node (estado actual), planets-destination (posibles planetas de destino) y planets-mandatory (planetas de paso obligatorio)
Salida: T si el planeta actual es un posible planeta destino y se han pasado por todos los planetas obligatorios
Procesamiento:
Si node esta planets-destination
AND
Para cada planeta en planets-mandatory
Existe ancestro de node == planeta
Devuelve T
Si no devyelve FALSE

CÓDIGO

```
;; FUNCION AUXILIAR

(defun all-mandatory-visited (node planets-mandatory)
  (cond
    ((null planets-mandatory) T) ;; Si todos los obligatorios visitados: T
    ((null node) NIL) ;; Si no, si no quedan nodos padre donde buscar: NIL
    ((member (node-state node) planets-mandatory :test #'equal)
     (all-mandatory-visited
       (node-parent node)
       (remove (node-state node) planets-mandatory :test #'equal)))
       ;; Si el nodo esta en la lista, lo retiramos y repetimos con padre y con la nueva lista
    (T (all-mandatory-visited (node-parent node) planets-mandatory)))))
    ;; En otro caso, repetimos con el padre y con la misma lista

;; Con node-01, 02 , 03 y 04 de abajo:
;; (setq mandatory '(Avalon Katril))
;; (all-mandatory-visited node-03 mandatory ) T
;; (all-mandatory-visited node-02 mandatory ) NIL
;; (all-mandatory-visited node-04 mandatory ) T


(defun f-goal-test-galaxy (node planets-destination planets-mandatory)
  (and
    (member (node-state node) planets-destination :test #'equal)
    (all-mandatory-visited 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

Para este ejercicio hemos optado por la inclusión de una función auxiliar que, dado un nodo, investiga entre sus ancestros si han sido visitados todos los planetas obligatorios

EJERCICIO 3B: Igualdad de estados

PSEUDOCÓDIGO

Entrada: node1 y node2 (estados a comparar), planets-mandatory (planetas de visita obligada)
Salida: T si ambos estados son ek mismo, NIL en otro caso
Procesamiento:
Si node1 == node2
AND
obligatorios-por-visitar(node1) == obligatorios-por-visitar(node2)
Devuelve True
Si no: devuelve NIL

CÓDIGO

```
;; FUNCION AUXILIAR

;; Devuelve, dado un nodo y una lista de node-states obligatorios, los
;; node-states que quedan por visitar

(defun mandatory-to-visit (node mandatory)
  (cond
    ((null node) mandatory) ;; Si ya no quedan antecesores, mandatory contiene
    ;; los que faltan por visitar
    ((member (node-state node) mandatory :test #'equal) ;; Si nodo esta en mandatory,
     (mandatory-to-visit                          ;; lo retiramos de la lista
       (node-parent node)                         ;; y llamamos al padre con
       (remove (node-state node) mandatory)))     ;; con la nueva lista
    (T (mandatory-to-visit (node-parent node) mandatory))))
    ;; Si no, repetimos con el padre y con la misma lista
```

```
;;;;;;; EJEMPLOS ;;;;;;;;;

;; (mandatory-to-visit node-02 '(Avalon)) NIL
;; (mandatory-to-visit node-02 '(Avalon Katril)) (KATRIL)
;; (mandatory-to-visit node-03 '(Avalon Katril)) NIL
;; (mandatory-to-visit node-04 '(Avalon Katril Tetera)) (TETERA)
;; (mandatory-to-visit node-04 '(Taza Avalon Katril Tetera)) (TAZA TETERA)

;; FUNCION AUXILIAR

;; Devuelve true si los planetas obligatorios que quedan por visitar a node-1
;; son los mismos que quedan por visitar a node-2
;; null-XOR nos permite comprobar si 2 conjuntos son iguales

(defun same-mandatory-to-visit (node-1 node-2 planets-mandatory)
  (null
    (set-exclusive-or (mandatory-to-visit node-1 planets-mandatory)
                      (mandatory-to-visit node-2 planets-mandatory))))

;;;;;;; EJEMPLOS ;;;;;;;;;

(same-mandatory-to-visit node-02 node-02 '(Avalon Katril)) ;; T
(same-mandatory-to-visit node-02 node-02 '(Avalon)) ;; T
(same-mandatory-to-visit node-02 node-04 '(Avalon Katril)) ;; NIL
(same-mandatory-to-visit node-03 node-04 '(Avalon Katril)) ;; T
(same-mandatory-to-visit node-03 node-04 '()) ;; T

;; FUNCION PRINCIPAL

;; Mismo estado si mismo planeta y mismos planetas por visitar
(defun f-search-state-equal-galaxy (node-1 node-2 &optional planets-mandatory)
  (and
    (equal (node-state node-1) (node-state node-2))
    (same-mandatory-to-visit 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

En este apartado hemos hecho uso de 2 funciones auxiliares:

- La primera de ellas, dado un nodo y una lista de mandatory-planets, devuelve qué planetas de la lista quedan por visitar entre el nodo y sus ancestros
- La segunda permite comprobar si, dados dos nodos y una lista de planetas obligatorios, a ambos nodos les quedan los mismos planetas obligatorios por visitar. Esta funcion, en vez de comprobar elemento a elemento si $S \subset T$ y $T \subset S$, hemos usado la funcion XOR para conjuntos, que devuelve aquellos elementos que aparecen exactamente en 1 de los 2 conjuntos. Por tanto, si esta función devuelve NIL, todos los elementos aparecen en ambos conjuntos y por tanto son iguales

2 Formalización del problema

EJERCICIO 4: Representación LISP del problema

CÓDIGO

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

Ejercicio 5: Expansión de nodo

PSEUDOCÓDIGO

Entrada: El nodo a expandir y el problema completo
Salida: Una lista que contiene todos los nodos a los que se puede acceder desde el nodo actual en el problema.
Procesamiento:
1) Recorremos toda la lista de operadores del problema y se los aplicamos al nodo actual
2) Cada operador devuelve la lista de acciones que se pueden realizar desde el nodo actual con el operador en el que estamos
3) Para cada una de estas acciones construimos el nodo hijo resultante de llevar a cabo esa acción
4) Unimos todas las listas de nodos hijo posibles a traves de cada uno de los operadores.

CÓDIGO

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Funcion auxiliar que expande un nodo recibido como primer argumento
;; con el operador recibido como segundo, en el problema recibido
;; como tercero
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun expand-operator (node operator problem)
    (mapcar #'(lambda (actn)
         ;; Creamos un nodo con los datos correspondientes
            (make-node
                :state (action-final actn) :parent node
                :action actn :depth (+ (node-depth node) 1)
                :g (+ (node-g node) (action-cost actn))
                :h (funcall (problem-f-h problem)
                            (action-final actn))
                :f (+ (+ (node-g node) (action-cost actn))
                    (funcall (problem-f-h problem)
                            (action-final actn)))))
        (funcall operator node)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Funcion principal
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defun expand-node (node problem)
  (mapcan ;; Llama a expand operator para todos los operadores
    #'(lambda (operator) (expand-operator node operator problem))
    (problem-operators problem)))
```

COMENTARIOS

La función utiliza una función auxiliar llamada $expand - operator$, que recibe el nodo actual, un operador y un problema, y ejecuta el operador sobre el nodo. Con la lista de acciones resultante construye la lista de nodos hijo a los que se puede llegar a traves de todas esas acciones.

Ejercicio 6: Manejo de listas de nodos

PSEUDOCÓDIGO

Entrada: Una lista de nodos (no necesariamente ordenada), una lista de nodos ordenada y su estrategia de ordenación.
Salida: Una lista ordenada que contiene todos los nodos de ambas listas
Procesamiento:
1) Tomamos el primer elemento de la primera lista, si no existe, devolvemos la segunda lista
2) Si existe continuamos la recursion, que nos devuelve una lista ordenada que contenga todos los elementos de la segunda lista y los restantes de la primera.
3) Comparamos este primer nodo de la primera lista nodo de la lista con todos los elementos de la lista ordenada devuelta por la recursión hasta encontrar su lugar.
4) Insertamos el nodo en el resultado de la recursión de tal forma que la lista sigue ordenada y devolvemos esa lista.

CÓDIGO

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Funcion auxiliar que inserta en la lista de nodos recibida como
;; segundo argumento el nodo recibido omo primero segun la estrategia
;; de ordenacion recibida como tercero.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun insert-node-strategy (node lst-nodes strategy)
   (cond ((null lst-nodes) ;; Si hemos llegado al final de la recursion
       (list node))  ;; (no hay mas nodos) devolvemos una lista
                  ;; que contenga el nodo

       ((funcall (strategy-node-compare-p strategy) ;; Llamamos a la
                 node                ;; estrategia de comparacion
```

```
                (first lst-nodes))     ;; entre el nodo y el primer
                                       ;; nodo de la lista
           (cons node lst-nodes))      ;; Si este es su ligar lo metemos
       (t (cons (first lst-nodes)      ;; Si no, continuamos la recursion
                (insert-node-strategy node (rest lst-nodes) strategy)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Funcion principal
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun insert-nodes-strategy (nodes lst-nodes strategy)
   (if (null nodes) ;; Si no quedan mas nodos
       lst-nodes ;; devolvemos la lista de nodos
       (insert-node-strategy (first nodes) ;; Si quedan mas lo insertamos
                             ;; en la lista resultante de
                             ;; continuar la recursion con el
                             ;; resto de nodos
          (insert-nodes-strategy (rest nodes) lst-nodes strategy)
          strategy)))


(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-<=))
```

COMENTARIOS

Hemos utilizado una función auxiliar llamada $insert - node - strategy(nodelst - nodesstrategy)$ que devuelve la lista recibida como segundo argumento con el nodo recibido como primer argumento según la estrategia de ordenación recibida como tercero. Se encarga por tanto de los pasos 3 y parte del 4 del pseudocódigo

3 Búsquedas

EJERCICIO 7: Definir estrategia para la búsqueda A*

PSEUDOCÓDIGO node-f-<=

```
Entrada:  nodo1 y nodo2, a comparar
Salida:  T si f(nodo1) <= f(nodo2).  NIL en otro caso
Procesamiento:
 Si coste-f-nodo1 <= coste-f-nodo2:
  devuelve T
 En otro caso:
  devuelve NIL
```

CÓDIGO

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

;; A* explora nodos en funcion de su coste f = g+h
(defun node-f-<= (node-1 node-2)
  (<= (node-f node-1)
      (node-f node-2)))

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

COMENTARIOS

Como la estrategia A* escoge el próximo nodo a explorar en función de el coste f de un nodo, la función auxiliar se encarga de, dados 2 nodos, devolver si el coste f del nodo-1 es menor o igual que el coste f del nodo-2

EJERCICIO 8: Función de búsqueda

PSEUDOCÓDIGO graph-search

```
 Entrada:  problem (estructura de problema) strategy (estrategia del problema)
 Salida:  Estructura de nodos con el camino que resuelve el problema, si existe, o
NIL
 Procesamiento:
  Si no hay nodos que explorar en la lista de abiertos:
    No hay solucion
  Si el primer nodo de la lista de abiertos es solucion:
    Devuelve la solucion
  Si el primer nodo de abiertos no está en la lista de cerrados
  OR
  Si el primer nodo de abiertos está en cerrados pero tiene menor coste g:
    Expande el nodo
    Inserta el resultado en la lista de abiertos según estrategia
    Retira de abiertos el nodo que ha sido expandido
    Añade a cerrados el nodo que ha sido expandido
    Repite la llamada con los nuevos parámetros
  En otro caso:
    Retira de abiertos el primer nodo
    Repite la llamada con los nuevos parámetros
```

PSEUDOCÓDIGO a-star-search

```
 Entrada problem (estructura del problema)
 Salida:  estructura de nodos con el camino, soluciónsi existe, NIL en caso
contrario
 Procesamiento:
  Hacer graph-search con la estrategia propia de A*
```

CÓDIGO

```
(defun graph-search-rec (problem strategy open closed)
  (let*
    ((frst (first open))
     (test (problem-f-goal-test problem))
     (found (find
             frst
             closed
             :test #'(lambda (x y)
                       (funcall
                         (problem-f-search-state-equal problem)
                         x
                         y)))))
    (cond
      ; No quedan nodos donde probar: terminamos
```

```
      ((null open) NIL)
      ; Hemos alcanzado objetivo: terminamos
      ((funcall test frst) frst)
      ; Si el nodo no esta en cerrados o si que
      ; esta pero 'mejora' el valor g del cerrado:
      ; exploracion + llamada recursiva
      ((or
         (null found)
         (< (node-g found) (node-g frst)))
       ;Llamada recursiva
       (graph-search-rec
         ;El problema y la estrategia no cambian
         problem
         strategy
         ;A los abiertos hay que retirar el nodo explorado
         ; y aniadir todos los que resultan de explorarlo
         (insert-nodes-strategy
           (expand-node frst problem)
           (rest open)
           strategy)
         ;Y a los cerrados hay que aniadir el explorado
         (cons frst closed)))
      ; En otro caso: llamada recursiva sin explorar
      (T
        (graph-search-rec
          problem
          strategy
          (rest open)
          closed)))))

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


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


(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-WORM-HOLE :ORIGIN MALLORY :FINAL KATRIL
;;               :COST 5)
;;            :DEPTH 1 :G 5 :H 9 :F 14)
;;         :ACTION
;;         #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN KATRIL :FINAL DAVION
;;            :COST 5)
;;         :DEPTH 2 :G 10 :H 5 :F 15)
;;      :ACTION
;;      #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN DAVION :FINAL PROSERPINA
;;         :COST 5)
;;      :DEPTH 3 :G 15 :H 7 :F 22)
;;   :ACTION
;;   #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN PROSERPINA :FINAL SIRTIS :COST 9)
;;   :DEPTH 4 :G 24 :H 0 :F 24)

(a-star-search *galaxy-M35*);->
;;#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-WORM-HOLE :ORIGIN MALLORY :FINAL KATRIL
;;               :COST 5)
;;            :DEPTH 1 :G 5 :H 9 :F 14)
;;         :ACTION
;;         #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN KATRIL :FINAL DAVION
;;            :COST 5)
;;         :DEPTH 2 :G 10 :H 5 :F 15)
;;      :ACTION
;;      #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN DAVION :FINAL PROSERPINA
;;         :COST 5)
;;      :DEPTH 3 :G 15 :H 7 :F 22)
;;   :ACTION
;;   #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN PROSERPINA :FINAL SIRTIS :COST 9)
```

```
;;   :DEPTH 4 :G 24 :H 0 :F 24)
```

Ejercicio 9: Mostrado del listado de los nodos visitados y de las acciones realizadas

PSEUDOCÓDIGO

Entrada: Un nodo
Salida: Dependiendo de si llamamos a $solution-path(node)$ o a $action-sequence(node)$, la lista de los planetas recorridos para llegar a ese nodo, o de las acciones realizadas, respectivamente.
Procesamiento:
En el caso de solution-path:
1) Compruebo si el nodo es huérfano, si lo es así devuelvo una lista que contenga el nombre del planeta del nodo.
2) Si no es huérfano devuelvo una lista que contenga como segundo elemento el nombre del planeta del nodo y como primero el resultado de continuar la recursión con el padre del nodo.
En el caso de action-sequence el funcionamiento es exactamente el mismo, solo que en vez de almacenar los nombres de los nodos almacenamos las acciones realizadas para llegar a ellos, es decir, el contenido del campo

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   BEGIN Exercise 9: Solution path / action sequence
;;;
(defun solution-path (node)
   (unless (null node) ;; A menos que el nodo sea nil
      (if (null (node-parent node)) ;; Si es huerfano
         (list (node-state node)) ;; devolvemos una lista que
                         ;; contenga su nombre unicamente
         ;; Si no, devolvemos una lista que contenga primero el
         ;; resultado de continuar la recursion y luego el
         ;; nombre del nodo
         (append (solution-path (node-parent node))
                 (list (node-state node))))))

(solution-path nil)
(solution-path (a-star-search *galaxy-M35*))

(defun action-sequence (node)
   (unless (null node) ;; Si el nodo es nil devolvemos nil
        (if (null (node-parent node)) ;; Si es huerfano
        (list (node-action node))  ;; devolvemos una lista que
                          ;; contenga la accion del nodo
        ;; Si no es huerfano, devolvemos una lista que contenga primero
```

```
      ;; el resultado de continuar la recursion y luego la accion
      ;; asociada al nodo actual
      (append (action-sequence (node-parent node))
           (list (node-action node))))))
```

COMENTARIOS

Ejercicio 10: Otras estrategias de búsqueda

Explicación

Sencillamente, para depth-first queremos sacar primero de la cola aquellos nodos que tengan la profundidad más alta, por tanto, la comparación dará positiva si el primer nodo es mas profundo que el segundo, es decir, tiene valor de prioridad mayor el nodo con mayor profundidad. Por otro lado ocurre lo contrario en breadth-first: la comparación es positiva si y solo si el primer nodo es menos profundo que el segundo. De esta manera tiene mayor valor de prioridad el nodo que menor profundidad tenga y quedará antes en la cola y se le sacará (se explorará) antes.

CÓDIGO

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

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

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



(solution-path (graph-search *galaxy-M35* *depth-first*))

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

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



(solution-path (graph-search *galaxy-M35* *breadth-first*))
```

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

Ejercicio 11: Preguntas

¿Por qué se ha realizado este diseño para resolver el problema de búsqueda?

Porque teniendo una heurística monótona como es el caso se demuestra que A* es óptimo y completo, además de ser muy eficiente. Por otra parte, nuestro diseño en concreto cuenta con una gran felixibilad y generalidad, ya que permite utilizar otros algoritmos de búsqueda definiendo una estrategia nueva como se ha hecho en el último ejercicio, y además la estructura problema sirve para problemas muy diversos, no solo para la búsqueda dentro de una galaxia.

Por otro lado, la razón por la que se utilizan funciones lambda es por su versatilidad. Un ejemplo de esta se aprecia en la estructura de la galaxia, en la que las funciones lambda permiten hacer referencias dentro de la propia funcion a otros campos de la estructura, como ocurre con los planetas prohibidos. Esto nunca habría sido posible con referencias a funciones, ya que tendrían que haber recibido esos datos como argumentos.

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?

En este aspecto, hay 2 puntos de vista:

- Por un lado, si tenemos en cuenta que por cada nodo que expandimos nos vemos obligados a guardar un atributo con la referencia al padre, el uso de memoria no es el mejor: si tenemos muchos nodos, el número de referencias que guardamos será alto y en absoluto despreciable. Conocemos algoritmos más eficientes en este sentido, como A* con profundidad iterativa. Por tanto, si no ineficiente, al menos no sería óptimo
- Por otro lado, si no estamos apurados de memoria, esta implementación de A* no supone un problema: aunque es cierto que por cada nodo necesitemos este atributo, al fin y al cabo es únicamente una referencia, no es el propio nodo. Sería problemático guardar copias de todos los ancestros de un nodo, pero no una única referencia.

¿Cuál es la complejidad espacial del algoritmo implementado? ¿Cuál es la complejidad temporal del algoritmo?

Será $O(b^a)a = (C * /epsilon)$ donde b es el factor de ramificación, $C*$ es el coste óptimo y *epsilon* es el coste mínimo por acción. Estos costes en el caso de la complejidad espacial serán espacio ocupado, y tiempo transcurrido en el caso de la complejidad temporal.

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).

Por un lado, podríamos añadir en cada nodo un atributo que nos indicara cuántos agujeros de gusano se han recorrido para llegar hasta él.

Paralelamente, el control debería llevarse desde las funciones de más alto nivel, hacia las de más bajo: tanto `a-star-search` como `graph-search` deberían transportar ese n (limite de veces) y hacerlos llegar a la función `graph-search-rec`.

Dentro de esta función, lo que debería variar con este nuevo límite es qué nodos se añaden a la lista open cuando exploramos un nodo (ya que, en vez de todos los alcanzables, se añaden todos los alcanzables que no superen el límite de viajes). Por tanto, dentro de `graph-search-rec` el valor n del límite debería hacerse llegar a la función de `expand-node`.

A su vez, ésta lo pasaría a `expand-operator` que (en el caso de que el operador fuera viajar por agujeros de gusano) sería la encargada de que el `make-node` dentro de ella sólo se ejecutara cuando el nodo del argumento de entrada no superara el límite, y además incrementaría el atributo agujeros-gusano de los nodos que sí creara