PRACTICA 2. BÚSQUEDA

Fecha de publicación: 2018/03/07
Fechas de entrega: [grupos jueves: miércoles 2018/04/04]
[grupos viernes: jueves 2018/04/05]
Planificación: semana 1: Ejercicios 1, 2, 3, 4, 5, 6
semana 2: Ejercicios 7, 8, 9, 10
semana 3: Ejercicios 11 + memoria Versión: 2018/03/18

IMPORTANTE:

- **Utilizad la versión Allegro CL 6.2 ANSI with IDE y COMPILAD el código.**
- **DEFINID un problema de búsqueda más simple para depurar el código.**

Forma de entrega: Según lo dispuesto en las normas de entrega generales, publicadas en Moodle

- El código debe estar en un único fichero.
- El código debe estar en un único fichero. La evaluación del código en el fichero no debe dar errores en Allegro CL 6.2 ANSI with IDE (recuerda: CTRL+A CTRL+E debe evaluar todo el código sin errores).

Material recomendado:

- En cuanto a estilo de programación LISP: http://www.cs.cmu.edu/Groups/AI/html/faqs/lang/lisp/part1/faq.html
- Como referencia de funciones LISP: http://www.lispworks.com/documentation/common-lisp.html

---

Descripción general

El problema de 'pathfinder' ('descubridor-de-caminos') es común en juegos de ordenador en los cuales los personajes pueden moverse en un espacio para alcanzar la posición de un objetivo. Por ejemplo, encontrar la salida de un laberinto. Estos problemas se modelan como búsquedas de caminos mínimos en grafos: hay lugares en que los personajes pueden estar (los nodos del grafo). Se pueden desplazar de un nodo a otro recorriendo las aristas del grafo. Recorrer una arista tiene un coste asociado. El problema computacional consiste en encontrar el camino de coste mínimo desde un nodo inicial a uno de una colección de nodos destino.

En esta práctica se propone desarrollar un módulo IA de uno de estos juegos para un jugador automático. El jugador automático debe dirigir el vuelo de una nave espacial *Pathfinder* en una galaxia. El objetivo es, saliendo de un planeta origen, llegar a uno de los planetas usando la menor cantidad posible de combustible.

En cada movimiento la nave debe navegar por la galaxia de planeta en planeta a través de una red de agujeros blancos unidireccionales o de una red de agujeros de gusano bidireccionales. Utilizar los agujeros conlleva un consumo energético para la nave mayor que cero. En la galaxia se han observado fluctuaciones anómalas del vacío cuántico, por lo que algunos agujeros de gusano no pueden ser utilizados.

El módulo IA de la nave debe usar algoritmos de búsqueda para encontrar el camino a uno de los planetas destino con el menor consumo energético posible. En cada planeta se ha desplegado un sistema de sensores que proporcionan una estimación heurística del coste para llegar al planeta destino. Es necesario recoger a dos científicas en sendos planetas de la galaxia con el objeto de investigar el origen de las anomalías de las fluctuaciones del vacío cuántico.

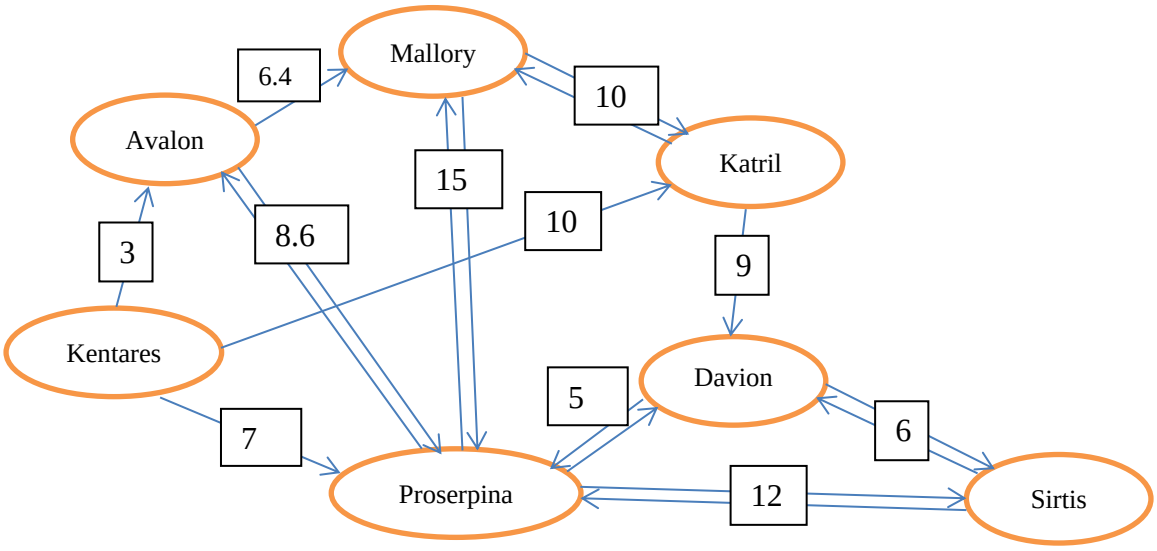
A continuación se presenta un ejemplo de galaxia: la galaxia Messier 35 (M35). Modelizaremos la galaxia como un grafo con dos tipos de arcos, los agujeros blancos (white holes) y los agujeros de gusano (wormholes), unidireccionales los primeros, bidireccionales los segundos. Trataremos los dos tipos de arcos como dos entidades independientes: en nuestro modelo habrá *operadores* que, dado un nodo, nos dirán en que nodos podemos ir. Habrá dos operadores distintos, uno para los agujeros blancos y uno para los de gusano.

Además de los dos grafos, el modelo incluye una tabla de estimación de consumo de energía, que usaremos para la heurística del algoritmo. En la tabla a la izquierda de la figura En la tabla situada a la izquierda de la figura

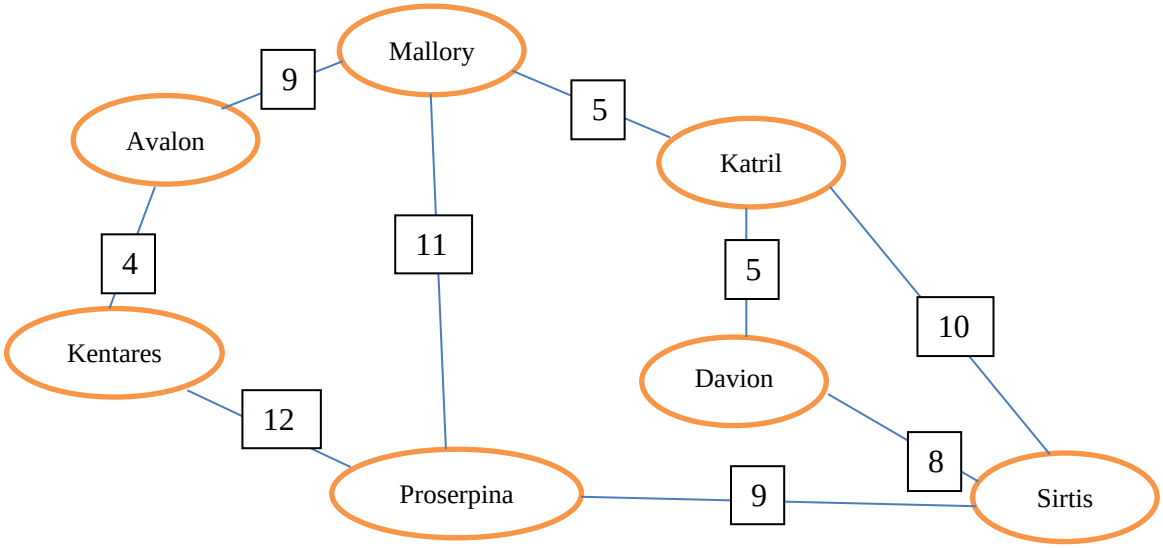
siguiente se muestra una estimación del consumo de energía para llegar a Sirtis desde cualquier planeta perteneciente a la Galaxia Messier 35.".

GALAXIA M35: Agujeros blancos (unidireccionales, cada arco con su consumo de energía)

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



GALAXIA M35: Agujeros de gusano (bidireccionales, cada arco con su consumo de energía)



El modelo del grafo se almacena en una serie de variables globales:

- **Planetas**: Lista constante de símbolos representando los nombres de los planetas
  `(defparameter *planets* '(Avalon Davion ...))`

- Los **agujeros blancos y agujeros de gusano**: Listas constantes de tripletes formados por **planeta origen, planeta destino y gasto energético** (coste)

```
(defparameter *white-holes*
  '((Avalon Mallory 6.4) (Avalon Proserpina 8.6)...))

 (defparameter *worm-holes*
  '((Avalon Kentares 4) (Avalon Mallory 9)
    (Davion Katril 5) (Davion Sirtis 8)
    (Katril Davion 5) (Katril Mallory 5) (Katril Sirtis 10)
```

```
(Kentares Avalon 4) (Kentares Proserpina 12)...)
```

- Los **sensores** (*heurística*): Lista constante de pares formados por el planeta y el valor de la heurística

```
(defparameter *sensors*
  '((Avalon 15) (Davion 5)…)
```

- **Planeta origen**: El estado inicial será el correspondiente a estar en el planeta origen de la galaxia.

```
(defparameter *planet-origin* 'Mallory)
```

- **Planetas destino**: Lista constante de símbolos representando los nombres de los planetas destino.

```
(defparameter *planets-destination* '(Sirtis))
```

- **Planetas prohibidos**: Lista constante de símbolos con los nombres de los planetas a los que no está permitido acceder por un agujero de gusano por las fluctuaciones anómalas del vacío cuántico que han sido observadas en la galaxia.

```
(defparameter *planets-forbidden* '(Avalon))
```

- **Planetas obligados**: Lista constante de símbolos con los nombres de los planetas en los que es necesario haber visitado para alcanzar la meta.

```
(defparameter *planets-mandatory* '(Katril Proserpina))
```

Estructuras

En esta práctica utilizaremos estructuras. Definiremos cuatro estructuras que permiten representar los elementos de nuestro programa. Una estructura “problem”, que reúne los datos relativos al problema (el estado inicial, la función para calcular el coste f, etc,); una estructura “node”, que contiene la información necesaria para trabajar con un nodo durante la búsqueda; una estructura “action”, que incluye la información que se genera cuando se cruza un arco, y una estructura “strategy” que contiene, esencialmente, una función de comparación de nodos: es la función que utilizaremos para decidir cuál será el próximo nodo que vamos a explorar.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;    Problem definition
;;
(defstruct problem
  states                ; List of states
  initial-state         ; Initial state
  f-h                   ; reference to a function that evaluates to the
                        ; value of the heuristic of a state
  f-goal-test           ; reference to a function that determines whether
                        ; a state fulfils the goal
  f-search-state-equal  ; reference to a predicate that determines whether
                        ; two nodes are equal, in terms of their search state
  operators)            ; list of operators (references to functions) to
                        ; generate successors
;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;    Node in the search algorithm
;;
(defstruct node
     state         ; state label
     parent        ; parent node
     action        ; action that generated the current node from its parent
     (depth 0)     ; depth in the search tree
     (g 0)         ; cost of the path from the initial state to this node
     (h 0)         ; value of the heuristic
     (f 0))        ; g + h
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;    Actions
;;
(defstruct action
     name          ; Name of the operator that generated the action
     origin        ; State on which the action is applied
     final         ; State that results from the application of the action
     cost )        ; Cost of the action
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;    Search strategies
;;
(defstruct strategy
     name               ; Name of the search strategy
     node-compare-p)    ; boolean comparison
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Ejercicios a realizar

1. Modelización del problema

EJERCICIO 1 (5 %.): Evaluación del valor de la heurística.

Codifica una función que calcula el valor de la heurística en el estado actual:

```
(defun f-h-galaxy (state sensors)...)

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

EJERCICIO 2 (20 %): Operadores `navigate-worm-hole` y `navigate-white-hole`.

Los operadores son funciones que, dado un estado (es decir, la etiqueta de un planeta), devuelven una lista de las acciones que se pueden efectuar a partir de ese estado.

En nuestra galaxia hay dos operadores: el primero nos dice a qué planetas se puede llegar usando agujeros blancos (*white holes*); el segundo a que se puede llegar usando agujeros de gusano (*wormholes*). La función devuelve una lista de acciones que nos llevan desde el primer estado al segundo (y, claramente, NIL si no hay aristas entre los dos estados). Recuerda que, debido a fluctuaciones del vacío cuántico, en este momento no está permitido acceder por un agujero de gusano a algunos planetas (los de la lista `*planets-forbidden*)`.

```
(defun navigate-white-hole (state white-holes ) ...)
(defun navigate-worm-hole (state worm-holes planets-forbidden)...)

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

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

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

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

Sugerencia: las dos funciones son muy parecidas. Es conveniente definir una función `navigate,` de carácter general y dos interfaces que le pasan los parámetros que necesita.

EJERCICIO 3A (10 %): Test para determinar si se ha alcanzado el objetivo.

Codifica una función que compruebe si se ha alcanzado el objetivo según el siguiente prototipo:

```
(defun f-goal-test-galaxy (node planets-destination 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 Uranus) '(Avalon Katril)); -> NIL
(f-goal-test-galaxy node-02 '(Kentares Uranus) '(Avalon Katril)); -> NIL
(f-goal-test-galaxy node-03 '(Kentares Uranus) '(Avalon Katril)); -> NIL
(f-goal-test-galaxy node-04 '(Kentares Uranus) '(Avalon Katril)); -> T
```

EJERCICIO 3B (10% extra): Predicado para determinar la igualdad entre estados de búsqueda.

Codifica una función que compruebe si dos para dos nodos son iguales de acuerdo con su estado de búsqueda. Es decir si se cumple que los dos nodos

(i) corresponden al mismo planeta

(ii) la lista de planetas de entre los obligados que aún han de visitar coincide.

Esta función es necesaria para determinar si un estado de búsqueda es un estado repetido, con el fin de descartarlo en caso de que se demuestre que no puede conducir a la solución óptima.

```
(defun f-search-state-equal-galaxy (node-1 node-2 &optional 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
```

2. Formalización del problema

EJERCICIO 4 (5%): Representación LISP del problema.

Inicializa el 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-goal-test          #'(lambda (node) ...)
   :f-search-state-equal #'(lambda (node-1 node-2)...)
   :operators            (list
                          #'(lambda (node)
                              ...)
                          ...)))
```

EJERCICIO 5 (15 %): Expandir nodo.

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

Nota: el problema contiene una lista de operadores, y hay que considerar los estados que se pueden alcanzar usando todos estos operadores. Cada operador devuelve una lista de acciones. Hay que construir los nodos que se pueden generar usando estas acciones.

```
(defun expand-node (node problem)...)
```

El resultado debe ser una **lista de nodos.** Ejemplo:

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

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


(print lst-nodes-00)

;;;(#S(NODE :STATE AVALON
;;;         :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0 :F 20)
;;;         :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN PROSERPINA :FINAL AVALON :COST 8.6)
;;;         :DEPTH 13   :G 18.6  :H 15  :F 33.6)
;;; #S(NODE :STATE DAVION
;;;         :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0 :F 20)
;;;         :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN PROSERPINA :FINAL DAVION :COST 5)
;;;         :DEPTH 13   :G 15    :H 5   :F 20)
;;; #S(NODE :STATE MALLORY
;;;         :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0 :F 20)
;;;         :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN PROSERPINA :FINAL MALLORY :COST 15)
;;;         :DEPTH 13   :G 25    :H 12  :F 37)
;;; #S(NODE :STATE SIRTIS
;;;         :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0 :F 20)
;;;         :ACTION #S(ACTION :NAME NAVIGATE-WHITE-HOLE :ORIGIN PROSERPINA :FINAL SIRTIS :COST 12)
;;;         :DEPTH 13   :G 22    :H 0   :F 22)
;;; #S(NODE :STATE KENTARES
;;;         :PARENT #S(NODE :STATE PROSERPINA :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0 :F 20)
;;;         :ACTION #S(ACTION :NAME NAVIGATE-WORM-HOLE :ORIGIN PROSERPINA :FINAL KENTARES :COST 12)
;;;         :DEPTH 13   :G 22    :H 14  :F 36)
;;; #S(NODE :STATE 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))
```

EJERCICIO 6 (10 %): 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.

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

Se supone que la lista `lst-nodes` está ordenada de acuerdo a dicha estrategia. La lista de nodos `nodes` no tiene por qué tener una ordenación especial.

Ejemplo:

```
(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) )
```

```
(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))
```

Una curiosidad: LISP no es un lenguaje “strongly typed”, por tanto el siguiente ejemplo (quizás un poco más fácil a entender) también funciona:

```
(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)
```

3. Búsquedas

EJERCICIO 7 (5 %): Definir estrategia para la búsqueda A*.

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

```
(defparameter *A-star*
  (make-strategy ...))
```

Ejemplo: Estrategia para búsqueda de coste uniforme:

```
(defparameter *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 8 (20 %): Función de búsqueda.

Codifica la función de búsqueda, según el siguiente pseudocódigo:

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)

```
; Realiza la búsqueda para el problema dado utilizando una estrategia
; Evalúa:
;    Si no hay solución: NIL
;    Si hay solución: un nodo que cumple el test objetivo
;
(defun graph-search (problem strategy)
      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.

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

A partir de esta función, codifica

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

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

EJERCICIO 9 (5 %): Ver el camino seguido y la secuencia de acciones.

A partir de un nodo que es el resultado de una búsqueda, codifique

- Función que muestra el camino seguido para llegar a un nodo. Muestra los estados (es decir, el nombre de los planetas) que se han visitado para llegar al nodo dado.
  `(defun solution-path (node)...)`

  Ejemplos:

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

- Función que muestra la secuencia de acciones para llegar a un nodo:
  `(defun action-sequence (node)...)`
  Ejemplo:

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

EJERCICIO 10 (5 %): Otras estrategias de búsqueda.

Diseñe una estrategia para realizar búsqueda en profundidad:

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

(defun depth-first-node-compare-p (node-1 node-2)
     <codigo LISP para depth-first>)

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

Diseñe una estrategia para realizar búsqueda en anchura:

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

(defun breadth-first-node-compare-p (node-1 node-2)
     <codigo LISP para breadth-first>)

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

EJERCICIO 11: En la memoria se debe incluir respuestas a las siguientes preguntas

1. ¿Por qué se ha realizado este diseño para resolver el problema de búsqueda?
   En concreto,
   1.1 ¿Qué ventajas aporta?
   1.2 ¿Por qué se han utilizado funciones lambda para especificar el test objetivo, la heurística y los operadores del problema?
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?
3 ¿Cuál es la complejidad espacial del algoritmo implementado?
4 ¿Cuál es la complejidad temporal del algoritmo?
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).

Protocolo de corrección

[40%] Corrección automática (ejercicios 1-10)

La corrección de la práctica se realizará utilizando distintos problemas de búsqueda y distintas "galaxias" diferentes de la del ejemplo del enunciado, por lo que se recomienda definir una batería de pruebas con distintos problemas.

[30%] Estilo

[30%] Memoria (incluyendo las respuestas al ejercicio 11)