PRÁCTICA 2: INTELIGENCIA ARTIFICIAL

1. **EJERCICIO 1 (5 %):** Test para determinar si se ha alcanzado el objetivo.

   f-goal-test-galaxy (state planets-destination):
   Esta función simplemente comprueba si el state pertenece a la lista planets-destinations.

   EJEMPLOS
   Caso correcto:
   `(f-goal-test-galaxy 'Sirtis *planets-destination*) ;-> (SIRTIS)`
   Caso erróneo 1:
   En este caso el estado no pertenece a la lista de destinos
   (f-goal-test-galaxy 'Avalon *planets-destination*) ;-> NIL
   Caso erróneo 2:
   En este caso el no es alcanzable al no pertenecer al grafo (pero si está en la lista)
   (f-goal-test-galaxy 'Avalon *planets-destination*) ;-> NIL

   COMENTARIOS
   Debido a nuestra implementación cuando un estado pertenece a la lista de destinos se devuelve una lista, no true

2. **EJERCICIO 2 (5 %):** Evaluación del valor de la heurística.

   f-h-galaxy (state sensors):
   Dada una lista de pares (estado heurística) esta función comprueba que el state pasado como argumento pertenece a la lista sensors y en tal caso devuelve la heurística de dicho estado.

   EJEMPLOS
   Caso correcto 1:
   (f-h-galaxy 'Sirtis *sensors*) ;-> 0 ; En este caso Sirtis es nodo objetivo
   Caso correcto 2:
   (f-h-galaxy 'Avalon *sensors*) ;-> 5 ; Caso nodo no objetivo
   Caso erróneo:
   (f-h-galaxy Urano *sensors*) ;-> NIL ; Caso nodo no tiene heurística

   COMENTARIOS
   Se ha añadido un caso de comprobación de error en el que se contempla la posibilidad de que el estado introducido no tenga una heurística, esto ocurre cuando el state en cuestión no pertenece al grafo.

3. **EJERCICIO 3 (20 %):** Operadores navigate-worm-hole y navigate-white-hole.

   Para la realización de estas funciones hemos implementado una función auxiliar genérica. Esta nos permite generalidad a la hora de crear diferentes formas de navegar.

   navigate (name state net):
   Dado un nodo (identificado por su state) esta función genera una lista con todas las posibles

acciones. El parámetro name hace referencia al nombre de la acción. En este caso solo existen dos (navigate-worm-hole y navigate-white-hole) pero al encapsular ambas en una sola función, se podrían considerar más.
Así la implementación de las funciones pedidas se resume de la siguiente manera:

- (defun navigate-worm-hole (state worm-holes)
  (navigate "NAVIGATE-WORM-HOLE" state worm-holes))
- (defun navigate-white-hole (state white-holes)
  (navigate "NAVIGATE-WHITE-HOLE" state white-holes))

```
EJEMPLOS
(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))
(navigate-white-hole 'Urano *white-holes*) ;-> NIL
(navigate-white-hole 'avalon *white-holes*) ;->
;;(#S(ACTION :NAME "NAVIGATE-WHITE-HOLE" :ORIGIN AVALON :FINAL
MALLORY
;;           :COST 2)
;; #S(ACTION :NAME "NAVIGATE-WHITE-HOLE" :ORIGIN AVALON :FINAL
PROSERPINA
;;           :COST 12))
```

4. **EJERCICIO 4 (5 %):** Definir estrategia para la búsqueda A*.

Siguiendo el ejemplo proporcionado de la búsqueda uniforme implementamos la búsqueda por A* mediante la comparación de los valores f de las estructuras de tipo node.

```
(defun node-compare-a* (n1 n2)
  (<= (node-f n1) (node-f n2)))

(setf *A-star*
  (make-strategy
   :name 'A-star
   :node-compare-p 'node-compare-a*))
```

Así, el ejercicio 4 consiste en la creación de la función compare-a* e instanciar la estrategia A-star como una variable global

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

Siguiendo la dinámica del enunciado (f-goal-test), hemos decidido implementar la estructura *galaxy-M35* de la siguiente manera:

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

```
 :operators (list #'(lambda(state) (navigate-worm-hole state *worm-
holes*))
                  #'(lambda(state) (navigate-white-hole state
*white-holes*)))))
```

Como puede observarse los campos del problema son rellenados con funciones lambda, que realizan las funciones indicadas por el slot: comprobar si el estado es destino (**f-goal-test**), dar la heurística de un determinado estado (**f-h**) y obtener la lista de los operadores disponibles, a través de los cuales se expanden los estados(**operators**).

6. **EJERCICIO 6 (15 %):** Expandir nodo

Esta función genera los nodos sucesores a un determinado nodo siguiendo las estrategias disponibles en un problema específico.

PSEUDOCÓDIGO

```
Ops=getOperators(Problem)
Para cada o en Ops
    AñadirNodos(Suces, generarSucesor(nodo o))
```

Para implementarlo, conseguimos del problema la lista de operadores e iteramos sobre ellos con un mapcar. La acción de generar un sucesor consiste en hacer una llamada al constructor de la estructura (make-node) y rellenarla con todos los campos, sumando uno a la profundidad actual en el árbol y llamando a las funciones que obtienen la heurística del problema.

7. **EJERCICIO 7 (10 %):** Gestión de nodos. Inserta una lista de nodos en otra lista de acuerdo con una estrategia.

En este ejercicio iremos comprobando elemento a elemento de la lista ordenada si el nodo que queremos insertar debe ir antes de un elemento en concreto. Así, queda el siguiente pseudocódigo:

```
While (NO vacia(lista))
    Menor=compararStrategy(node, lista[i])          //Menor tipo boolean, node menor??
    If(Menor)
            pos++ ; continue;
    Else // caso node es mayor
            Insertar(node, pos)
            In=1
            Break
 If In=0 // node es el elemento mayor
    Insertar(Node, pos)
```

Sin embargo, para la implementación en lisp usaremos una función auxiliar que realiza la inserción de un único nodo. Esta función controlará los nodos visitados y cuando encontremos la posición correcta en la que debe ir el nodo, solamente será necesario realizar un append de la lista de nodos ya visitados, el nodo que estamos insertando y todos

los que son mayores que él (los que no se han visitado todavía). Así, la versión de lisp de este pseudocódigo queda mucho más reducida de la siguiente manera:

```
(defun insert-node-strategy (node lst-nodes strategy comp-lst)
  (cond
   ((null lst-nodes) (append comp-lst (list node))) ;;inserción al
final
   ((funcall (strategy-node-compare-p strategy) node (first lst-
nodes))
      (append comp-lst (cons node lst-nodes))) ;; inserción en la
posición correspondiente
   (T (insert-node-strategy node (rest lst-nodes) strategy
                            (append comp-lst (list (first lst-
nodes)))))))) ;; seguimos buscando la posición
```

Posteriormente, solo habrá que llamar recursivamente a esta función auxiliar para insertar todos los nodos necesarios.
Esta función, tal y como pide el enunciado, no comprueba la ordenación de la lista donde se van a insertar los nodos, de manera que, en el momento que encuentre un nodo con mayor coste al que queremos insertar, lo meterá inmediatamente delante de él en la lista.

8. **EJERCICIO 8 (15 %):** Función de búsqueda.

Para la realización de este ejercicio hemos implementado una función auxiliar con esta cabecera: **graph-search-aux (problem strategy open-nodes closed-nodes)**
Esto nos permite saber que nodos están en la lista de abiertos y que nodos ya han sido visitados.

Una vez sabemos que vamos a utilizar esta función auxiliar de manera recursiva también es fácil deducir que vamos a utilizar la función proporcionada por el enunciado para inicializar lo recursos necesarios que le pasaremos a la auxiliar. Por ello, tal y como especifica el algoritmo inicializamos la lista de nodos abiertos con el estado inicial del problema y la lista de cerrados vacía.

El resto del algoritmo ocurre en la función auxiliar, es aquí donde se comprueba que el primer nodo de la lista de abiertos cumple o no el test objetivo proporcionado por el problema. Si no se cumple y si el nodo considerado no está en la lista cerrada o, estando en dicha lista, tiene un coste inferior al del que está en la lista de cerrados, entonces expandimos el nodo actual dentro de la lista de abierto y tras borrarlo de esta lista (ya lo hemos visitado y expandido) lo añadimos a la lista de cerrados.
Con todo esto simplemente realizamos la llamada recursiva.

9. **EJERCICIO 9 (5 %):** Búsqueda A*. Implementar el algoritmo de búsqueda A*.

Simplemente llamamos a la función anterior con la estrategia A* definida en el ejercicio 4.

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

En este caso, como teníamos la variable global *A-star*, la hemos usado en la función, no

obstante, si se evitar la utilización de variables globales en el código, habría que crear de nuevo la estructura problem correspondiente a A*.

10. **EJERCICIO 10 (5 %):** Ver camino. Codifique la función que muestra el camino seguido para llegar a un nodo.

    Esta función deberá guardar en una lista el camino seguido hasta llegar al nodo pasado por argumento en la siguiente cabecera: **tree-path (node)**. Debido a que hay que llevar una lista con los nodos, hemos decidido implementar una función auxiliar con el siguiente prototipo: **tree-path-aux (node lst-states).**

    Ahora solo queda ir metiendo en la lista, de manera recursiva, los padres de los nodos tratados. Una vez lleguemos a un nodo sin padre se habrá acabado la recursión y tendremos el camino buscado.

11. **EJERCICIO 11 (5 %):** Secuencia de acciones de un camino. Codifique la función que muestra la secuencia de acciones para llegar a un nodo.

    Siguiendo un esquema muy similar al anterior debemos crear una función auxiliar en la que llevaremos la lista de las acciones. Estas acciones al igual que en el ejercicio anterior, se van consiguiendo siguiendo un algoritmo de tipo backtracking: es a partir del nodo final que, preguntando por el padre del nodo y su acción llegamos, mediante una recursión a obtener la lista deseada.

    La diferencia con el caso anterior es que el caso de parada cambia debido a que el nodo raíz no fue generado por ninguna acción. Así la recursión se para cuando se llega a un nodo que no fue generado por ninguna acción.

12. **EJERCICIO 12 (5 %):** Otras estrategias de búsqueda. Diseñe una estrategia para realizar búsqueda en profundidad y otra para realizar la búsqueda en anchura:

    Cabe destacar que tanto la búsqueda en profundidad como la búsqueda en anchura son algoritmos que solo tienen en cuenta (si no se añaden otros criterios) la profundidad a la que se han encontrado los nodos.

    En este caso, no se sigue ningún otro criterio de ordenación así que el resultado, como se ha explicado en el párrafo anterior solo nos dependerá de la profundidad y del orden de generación de los nodos.

    Además, sabemos que el algoritmo de búsqueda en profundidad da prioridad de exploración a los nodos de profundidad mayor, mientras que la búsqueda en anchura da más prioridad cuanto menor sea la profundidad.

    Así, las estrategias de búsqueda tienen la siguiente implementación

    Búsqueda en Profundidad

```
(setf *depth-first*
```

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

Comparador de la búsqueda en Profundidad

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

Busqueda en Anchura

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

Comparador de la búsqueda en Anchura

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

No es muy ilustrativo poner ejemplos dado que estos dependen en todo momento de cómo se hayan ordenado los nodos cuando se crearon las variables globales

PREGUNTAS TEÓRICAS

1. **¿Por qué se ha realizado este diseño para resolver el problema de búsqueda?**
   El diseño nos ha parecido muy bueno ya que además de ser válido para cualquier problema, la implementación con estructuras hace más sencilla la codificación.

   En concreto,

   1.1 ¿qué ventajas aporta?
   El diseño mediante estructuras aporta la importante ventaja de la generalidad. Podemos definir infinitos problemas y estrategias de búsqueda y para todos ellos la función graph-search sería válida.

   1.2 ¿Por qué se han utilizado funciones lambda para especificar el test objetivo, la heurística y los operadores del problema?
   De nuevo el motivo es la generalidad. Al tener funciones lambda el método graph-search simplemente realizaría un funcall sobre el slot de la estructura que corresponda independientemente 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?**
   Sí, ya que cuando se expande un nodo, todos sus hijos en el campo parent tienen la referencia a dicho nodo, por tanto, independientemente de cuantos hijos pueda tener el nodo, la instancia del padre será única para todos ellos.

3. **¿Cuál es la complejidad espacial del algoritmo implementado?**
   La complejidad es exponencial debido a que el algoritmo debe mantener en memoria todos los nodos cerrados y abiertos, por tanto, el espacio necesario crecerá exponencialmente con el árbol.

4. **¿Cuál es la complejidad temporal del algoritmo?**
   Si la heurística es monótona decreciente y está bien definida, el algoritmo debería converger rápidamente.

5. **Indicad qué partes del código se modificarían para limitar el número de veces que se puede utilizar la acción "navegar por agujeros de gusano" (bidireccionales).**
   Para evitar estropear la generalidad conseguida con esta implementación, una posible solución es añadir dentro de la estructura de action un contador del número máximo de veces que se permite usar cada operador. Así, cada vez que se expanda un nodo (ejercicio 6) habría que comprobar que el nuevo slot, limit, no es igual a 0. En este caso, no se generarían nodos con esta acción.
   Además, si no se pretende establecer un límite para el número de ocasiones en las que se puede usar una acción, entonces simplemente habría que poner un numero negativo a la hora de crear el método de navegación.