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

1. 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

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

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

**navigate (name state net):**

La función genera las acciones disponibles desde el nodo actual (state) que se pueden alcanzar dentro del conjunto de movimientos disponibles (net).

Gracias a esta función generar 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))

1. 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\*))

Asi, el ejercicio 4 se resume en la creación de la función compare-a\* e instanciar la estrategia A-star como una variable global

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

Siguiendo el ejemplo de 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 nombre: comprobar si el estado es destino(**f-goal-test**) y dar la heuristica de un determinado estado (**f-h**) y obtener la lista de los operadores disponibles, a través de los cuales se expanden los estados(**operators**).

1. EJERCICIO 6 (15 %): Expandir nodo

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

**PSEUDOCÓDIGO**

**expand-node (node problem)**

**INPUT :**node – nodo a expandir

**INPUT** **:**problem – problema que se seguirá para expandir el nodo

**OUTPUT: --** lista de nodos expandidos

**Procedimiento:**

Ops=getOperators(Problem)

Sucesores:

Suces=conseguirSucesor(nodo Problem[i]) ;;Suces es una lista

I++;

If(Problem[i]!=NULL)

Goto (Sucesores)

Sin embargo no resulta tan sencillo seguir el pseudocódigo en este caso, lo que tendremos que hacer será crear los nodos resultantes de expandir el nodo actual de acuerdo a las posibles acciones que se puedan abordar en el problema que se este tratado. Así que un pseudocódigo más cercano a la implementación final sería:

Ops=getOperators(Problem)

Sucesores:

Suces=generarSucesor(nodo Problem[i]) ;; Suces es una lista

I++;

If(Problem[i]!=NULL)

Goto (Sucesores)

1. 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, asi, 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 controlara 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 el. 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.

1. 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 vacia.

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 g inferior al del que está en closed-nodes, 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 realizado simplemente realizamos la llamada recursiva.

1. 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\*))

1. 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 q 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.

1. 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 backtranquin, es apartir 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, añadiendo una condición, ahora es necesario comprobar que la acción que genero al padre es nula

ESTO ES ASI!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! ^^^^^^

1. 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 asi que el resultado, como se ha explicado en el párrafo anterior solo nos dependerá de la profundidad y de el 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.

Asi, las estrategias de búsqueda tienen la siguiente implementacion

**Busqueda 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 TEORICAS

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 que se puede utilizar la acción “navegar por agujeros de gusano” (bidireccionales).