|  |
| --- |
| Inteligencia Artificial |
| Práctica 2 |
| Pareja 02 |

|  |
| --- |
| Óscar Gómez Borzdynski, José Ignacio Gómez García  1-1-2018 |

**Ejercicio 1:**

* **PSEUDOCÓDIGO**

*Entrada:* Un estado y una lista de heurísticas

*Salida:* Heurística desde el nodo estado hasta la meta.

*Función general:*

A = buscar-tupla-primer-elemento (estado lista-heurísticas)

Segundo-elemento (A)

* **CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;**

**;; BEGIN: Exercise 1 -- Evaluation of the heuristic**

**;;**

**;; Returns the value of the heuristics for a given state**

**;;**

**;; Input:**

**;; state: the current state (vis. the planet we are on)**

**;; sensors: a sensor list, that is a list of pairs**

**;; (state cost)**

**;; where the first element is the name of a state and the second**

**;; a number estimating the cost to reach the goal**

**;;**

**;; Returns:**

**;; The cost (a number) or NIL if the state is not in the sensor list**

**;;**

(defun f-h-galaxy (state sensors)

(cadr (assoc state sensors)))

(f-h-galaxy 'Sirtis \*sensors\*) **;-> 0**

(f-h-galaxy 'Avalon \*sensors\*) **;-> 15**

(f-h-galaxy 'Earth \*sensors\*) **;-> NIL**

* **COMENTARIOS**

En este caso, para buscar la tupla usamos assoc que, en caso de que el planeta no se encuentre, devolverá NIL. Al hacer cadr obtenemos el segundo elemento de la tupla. En caso de que la tupla sea NIL significará que no existe heurística, devolviendo NIL.

**Ejercicio 2:**

* **PSEUDOCÓDIGO**

*Entrada:* Un estado, una lista de enlaces (y una lista de planetas prohibidos)

*Salida:* Lista con todas las acciones posibles desde el nodo actual

*Función general:*

For terna in enlaces:

If (origen terna) == estado AND (destino terna) not in planetas prohibidos:

Make-action (name origin destination cost)

Return all actions

* **CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;**

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

**;;**

**;; Returns a list of all posible actions from a state**

**;;**

**;; Input:**

**;; state: the current state (vis. the planet we are on)**

**;; [white/worm]-holes: Links in our galaxy of a certain type**

**;; (planets-forbidden): Only if worm holes. Planets that are not allowed to be visited**

**;; allowed to be visited via worm-holes**

**;;**

**;; Returns:**

**;; A list with all possible actions from the original state**

**;; avoiding the links that go to a forbidden planet.**

**;;**

**;; Function that evaluates a link**

(defun check-hole (hole state forbidden name)

(let ((origin (first hole))

(destination (second hole))

(cost (third hole)))

(when (and (eql origin state**) ;;If the origin of the link is the state we are on**

(null (member destination forbidden))) **;; And the destination is not forbidden**

(list (make-action :name name **;; Create an action within a list**

:origin state

:final destination

:cost cost)))))

**;; Generical function for evaluating links**

(defun navigate (state holes planets-forbidden name)

(mapcan #'(lambda (x) (check-hole x

state

planets-forbidden

name))

holes))

**;; Function for evaluating white holes**

(defun navigate-white-hole (state white-holes)

(navigate state

white-holes

NIL

'NAVIGATE-WHITE-HOLE))

**;; Function for evaluating worm holes**

(defun navigate-worm-hole (state worm-holes planets-forbidden)

(navigate state

worm-holes

planets-forbidden

'NAVIGATE-WORM-HOLE))

(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 'Uranus \*worm-holes\* \*planets-forbidden\*) **;-> NIL**

* **COMENTARIOS**

En este caso, hemos decidido crear una función auxiliar para analizar un enlace. Recibirá el nombre del operador que ha generado la acción, el estado en el que nos encontramos, la lista de planetas prohibidos y la terna a analizar.

Con estos datos podemos asegurarnos de que el enlace es operativo desde el estado actual.

Para crear las funciones que operan sobre agujeros de gusano y agujeros blancos, hemos creado una función que realiza ambas acciones y será llamada desde la función externa.

**Ejercicio 3:**

* **PSEUDOCÓDIGO**

*Entrada:* Un nodo, una lista de planetas objetivo y una lista de planetas de obligada visita

*Salida:* T si cumplimos los requerimientos, NIL en caso contrario

*Función general:*

If nodo in planetas-objetivo AND path includes planetas-obligados:

Return T

Else:

Return NIL

*Función para comprobar el path (recursiva):*

Argumentos recibidos: Nodo actual, nodos obligatorios

If nodos-obligatorios – nodo actual == NULL

Return T

Get-parent (nodo)

If parent == NULL:

Return NIL

Return check-path(parent, obligatorios-modificado)

* **CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;**

**;; BEGIN: Exercise 3 -- Goal test**

**;;**

**;; Returns a list of all posible actions from a state**

**;;**

**;; Input:**

**;; node: the current node (vis. the planet we are on)**

**;; planets-destination: List of planets we need to reach (one of them)**

**;; planets-mandatory: List of planets we need to visit.**

**;;**

**;; Returns:**

**;; T if we reached the goal, NIL otherwise**

**;;**

**;; Checks if the path followed includes the mandatory planets**

(defun check-path (node planets)

(let ((new-planets (remove (node-state node)

planets)))

(if (null new-planets) **;; No more mandatory planets to visit**

T

(let ((parent (node-parent node)))

(unless (null parent) **;; No more path to analyze**

(check-path parent **;; Check next step of the path**

new-planets))))))

**;; Checks if we fullfilled the requirements**

(defun f-goal-test-galaxy (node planets-destination planets-mandatory)

(and (member (node-state node)

planets-destination) **;; Check if the node is a destination**

(check-path node planets-mandatory))) **;; Check if we visited the mandatory planets**

(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 esta función hemos necesitado una función auxiliar que comprobase si el camino seguido ha pasado por todos los nodos obligatorios. Hemos decidido realizar esta función de forma recursiva, de manera que, cada vez que ascendemos un nivel, cribamos la lista de nodos obligatorios, eliminando el nodo actual de la lista. En caso de que la lista de nodos obligatorios quede vacía, significará que hemos pasado por todos ellos y devolveremos T.