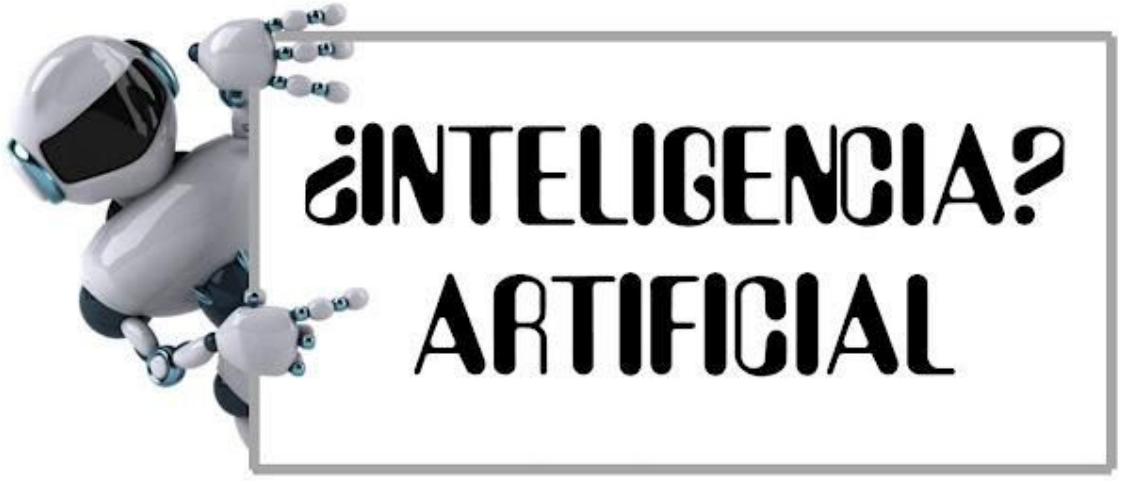
MEMORIA

Práctica 2: Búsqueda



Andrés Salas Peña
Ricardo Riol Fernández

Grupo 2301 - Pareja 08
Pr2 - Inteligencia Artificial

1. Introducción

En esta segunda práctica, nos adentramos aún más en la programación funcional de LISP. Entre los elementos nuevos que manejaremos a lo largo de la práctica, se encuentran las estructuras y todo lo relacionado con ellas (cómo crearlas, cómo acceder a ellas, …).

Por otro lado, en cuanto al sentido más funcional, el objeto de la práctica es el desarrollo de un módulo de inteligencia artificial para resolver el problema conocido como “path-finder”. Dicho problema consiste en encontrar el camino que conlleva menor “gasto” para llegar de un lugar origen a otro destino.

Concretando en nuestra práctica, el problema del path-finder se centra en la galaxia Messier 35 (M35) donde existen una serie de planetas que se “conectan” mediante agujeros blancos unidireccionales y agujeros de gusano bidireccionales.

En el siguiente subapartado, se muestra el código facilitado para la realización de la práctica. Su autoría corresponde a los profesores de la práctica y no a nosotros y además nos permite saber enfocar la práctica de una manera más particular. Por estos dos aspectos, hemos decidido incluir dicho código en este apartado y no en el siguiente que se corresponde al código pedido para la realización de la práctica y cuyos autores somos nosotros mismos (Andrés y Ricardo).

1.1 Código Previo

PLANETS

```
(defparameter *planets* '(Avalon Davion Katril Kentares Mallory Proserpina Sirtis))
```

WHITE-HOLES

```
(defparameter *white-holes*
  '((Avalon Mallory 6.4) (Avalon Proserpina 8.6)
    (Mallory Katril 10) (Mallory Proserpina 15)
    (Katril Mallory 10) (Katril Davion 9)
    (Kentares Katril 10) (Kentares Avalon 3) (Kentares Proserpina 7)
    (Proserpina Avalon 8.6) (Proserpina Davion 5) (Proserpina Mallory 15) (Proserpina Sirtis
12)
    (Davion Proserpina 5) (Davion Sirtis 6)
    (Sirtis Davion 6) (Sirtis Proserpina 12)
    ))
```

WORM-HOLES

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

(Kentares Avalon 4) (Kentares Proserpina 12)
(Mallory Avalon 9) (Mallory Katril 5) (Mallory Proserpina 11)
(Katril Mallory 5) (Katril Davion 5) (Katril Sirtis 10)
(Proserpina Kentares 12) (Proserpina Mallory 11) (Proserpina Sirtis 9)
(Sirtis Katril 10) (Sirtis Davion 8) (Sirtis Proserpina 9)
))

SENSORS

```
(defparameter *sensors*
  '((Avalon 15) (Davion 5) (Mallory 12) (Kentares 14) (Proserpina 7) (Katril 9) (Sirtis 0)))
```

PLANET-ORIGIN

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

PLANETS-DESTINATION

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

PLANETS-FORBIDDEN

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

PLANETS-MANDATORY

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

2. Código

2.1 Modelización del Problema

Ejercicio 1

Pseudocódigo

Entrada:
State: estado actual.
Sensors: lista de sensores que son una lista de pares estado, coste.
Salida:
Nil o el coste.
Procesamiento:
No procede

Código

```
(defun f-h-galaxy (state sensors)
  (second (assoc state sensors)))
```

Pruebas

```
(f-h-galaxy 'Sirtis *sensors*)
```

(f-h-galaxy 'Avalon *sensors*)
(f-h-galaxy 'Earth *sensors*)

Comentarios

La función se trata básicamente de una llamada a la función **assoc,** que lo que hace es encontrar el primer par que tiene como primer elemento el state que se pasa como argumento.

Ejercicio 2

Pseudocódigo

NAVIGATE-WHITE-HOLE

Entrada:
State: estado actual.
White-Holes: lista de todos los agujeros blancos.
Salida:
Una lista cuyo nombre es navigate-white-whole; su estado, el estado origen; su nodo final y el coste asociado.
Procesamiento:
La codificación es sencilla, la idea que hay detrás es recorrer toda la lista de agujeros blancos buscando aquellos en los que el estado coincida con el estado actual y con los que lo cumplan, crear una acción.

NAVIGATE-WORM-HOLE

Entrada:
State: estado actual.
Worm-holes: lista de todos los agujeros de gusano.
Planets-forbidden: lista de los planetas prohibidos por vacío cuántico.
Salida:
Una lista cuyo nombre es navigate-worm-whole; su estado, el estado origen; su nodo final y el coste asociado.
Procesamiento:
Misma idea que para la función anterior pero teniendo en cuenta que el nodo destino (planeta destino) al que llega el agujero de gusano no se encuentre en la lista de prohibidos.

Código

NAVIGATE-WHITE-HOLE

```
(defun navigate-white-hole (state white-holes)
  (let ((aux (first white-holes)))
    (cond
     ((null white-holes) nil)
     ((equal (first aux) state) (cons (make-action :name 'Navigate-white-whole
```

```
                                        :origin state
                                        :final (second aux)
                                        :cost (third aux))
                                (navigate-white-hole state (rest white-holes))))
    (t (navigate-white-hole state (rest white-holes))))))
```

NAVIGATE-WORM-HOLE

```
(defun navigate-worm-hole (state worm-holes planets-forbidden)
  (let ((aux (first worm-holes)))
    (cond
     ((null worm-holes) nil)
     ((and (equal (first aux) state) (not(find (second aux) planets-forbidden :test #'equal)))
      (cons (make-action :name 'Navigate-worm-whole
                         :origin state
                         :final (second aux)
                         :cost (third aux))
            (navigate-worm-hole state (rest worm-holes) planets-forbidden)))
     (t (navigate-worm-hole state (rest worm-holes) planets-forbidden)))))
```

NAVIGATE

```
(defun navigate (state wholes planets-forbidden)
(append (navigate-white-hole state wholes) (navigate-worm-hole state wholes
        planets-forbidden)))
```

Pruebas

```
(navigate-white-hole 'Kentares *white-holes*)
(navigate-worm-hole 'Mallory *worm-holes* *planets-forbidden*)
(navigate-worm-hole 'Mallory *worm-holes* NIL)
(navigate-white-hole 'Kentares *white-holes*)
(navigate-worm-hole 'Uranus *worm-holes* *planets-forbidden*)
```

Comentarios

Tal y como se puede comprobar ambas funciones de navegación son muy similares. Cabe destacar el uso de la función find que busca un elemento en una lista y la manera de construir una acción con el make-action :campo1 valorcampo1 :campo2 valorcampo2 …
De la función navigate no se comenta pseudocódigo porque es una llamada a ambas funciones de navigate y en la práctica no se usa para nada.

Ejercicio 3A

Pseudocódigo

F-GOAL-TEST-GALAXY

Entrada:
Node: nodo inicial.
Planets-destination: lista de todos los planetas de destino.

Planets-mandatory: lista de los planetas por los que no se puede pasar.

Salida:

True si se alcanzan o Nil si no.

Procesamiento:

El procesamiento es un poco más complejo que el anterior. En este caso, se requiere de una función auxiliar ya que la principal maneja solo el caso base. La auxiliar es la que contiene el código en sí que básicamente busca si el nodo padre pertenece a la lista de obligatorios. En caso de que sí, lo elimina y sigue buscando. Además contiene las condiciones de parada de que el nodo sea nulo o que la lista de planetas obligatorios haya sido encontrada entera y por tanto esté vacía.

Código

F-GOAL-TEST-GALAXY

```
(defun f-goal-test-galaxy (node planets-destination planets-mandatory)
  (and
   (find (node-state node) planets-destination :test #'equal)
   (equal nil (f-goal-test-galaxy-aux node planets-mandatory))))
```

F-GOAL-TEST-GALAXY-AUX

```
(defun f-goal-test-galaxy-aux (node planets-mandatory)
  (if (null node)
      planets-mandatory
    (f-goal-test-galaxy-aux (node-parent node)
                            (remove (find (node-state node) planets-mandatory :test #'equal)
                                    planets-mandatory :test #'equal))))
```

Pruebas

```
(defparameter node-01 (make-node :state 'Avalon) )
(node-state node-01)
(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))
(f-goal-test-galaxy node-02 '(kentares urano) '(Avalon Katril))
(f-goal-test-galaxy node-03 '(kentares urano) '(Avalon Katril))
(f-goal-test-galaxy node-04 '(kentares urano) '(Avalon Katril))
```

Comentarios

La dificultad del ejercicio radica en darse cuenta de que la función principal debe llamar a una recursiva y de cuáles son las condiciones de parada. Ambos aspectos ya se han comentado en el apartado de procesamiento del pseudocódigo.

Ejercicio 3B

Pseudocódigo

Entrada:
Node-1: primer nodo de la comparación.
Node-2: segundo nodo de la comparación.
Planets-mandatory (opcional): lista de los planetas por los que no se puede pasar.
Salida:
True si es repetido o NIL si no.
Procesamiento:
Para ver si el estado de búsqueda se repite o no, vemos que sucede con los dos nodos. Sí son el mismo, entonces el estado será repetido en caso de haber encontrado todos los planetas obligatorios. En otro caso, habría que comprobar que además de que ambos nodos sean el mismo, que su objetivo también sea el mismo, y esto se hace con la llamada a la función auxiliar del ejercicio anterior.

Código

```
(defun f-search-state-equal-galaxy (node-1 node-2 &optional planets-mandatory)
  (cond
   ((null planets-mandatory) (equal (node-state node-1) (node-state node-2)))
   (t (and (equal (node-state node-1) (node-state node-2))
           (equal nil (set-exclusive-or (f-goal-test-galaxy-aux node-1 planets-mandatory)
(f-goal-test-galaxy-aux node-2 planets-mandatory)))))))
```

Pruebas

```
(f-search-state-equal-galaxy node-01 node-01)
(f-search-state-equal-galaxy node-01 node-02)
(f-search-state-equal-galaxy node-02 node-04)

(f-search-state-equal-galaxy node-01 node-01 '(Avalon))
(f-search-state-equal-galaxy node-01 node-02 '(Avalon))
(f-search-state-equal-galaxy node-02 node-04 '(Avalon))

(f-search-state-equal-galaxy node-01 node-01 '(Avalon Katril))
(f-search-state-equal-galaxy node-01 node-02 '(Avalon Katril))
(f-search-state-equal-galaxy node-02 node-04 '(Avalon Katril))
```

Comentarios

El problema que podía surgir en el apartado anterior es que se repitiera indefinidamente el mismo resultado de búsqueda y para ello nace esta función.

2.2 Formalización del Problema

Ejercicio 4

Pseudocódigo

Campos:
States: todos los planetas de la galaxia.
Initial-state: planeta de partida.
F-h: heurística del estado.
F-goal-test: referencia a la función que determina si se alcanza un estado.
F-search-state-equal: referencia al predicado que determina la igualdad de estados.
Operators: lista de operadores que generan sucesores.

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

Comentarios

Este ejercicio consiste simplemente en saber manejar y llamar bien a los datos proporcionados en las estructuras y parámetros dados así como a funciones creadas por nosotros en los ejercicios anteriores.

Ejercicio 5

Pseudocódigo

EXPAND-NODE
Entrada:
Node: nodo expandido.
Problem: problema que queremos resolver, en este caso Galaxy-M35.
Salida:
Lista de nodos expandidos.

Procesamiento:
Si el nodo que pasamos a expandir es el final entonces ya está. Si no, debemos llamar a una función auxiliar. Esta irá haciendo cada una de las acciones listadas para dicho problema y guardará los datos en los distintos nodos a los que se va accediendo hasta que finalmente no haya ninguna acción que se pueda realizar.

BUILD-ACTIONS
Entrada:
Node: nodo.
Act: lista con las acciones.
Salida:
Acciones construidas de acuerdo a la estructura definida en el ejercicio 4.
Procesamiento:
Esta función auxiliar simplemente construye acciones a partir de un nodo y un problema.

Código

EXPAND-NODE
```
(defun expand-node (node problem)
  (if (funcall (problem-f-goal-test problem) node)
      'final
    (expand-node-aux node (build-actions node (problem-operators problem)) problem )))
```

BUILD-ACTIONS
```
(defun build-actions (node act)
  (if (null act)
      nil
    (append (funcall (first act) (node-state node)) (build-actions node (rest act)))))
```

EXPAND-NODE-AUX
```
(defun expand-node-aux (node list_actions problem)
  (let ((action (first list_actions)))
    (if (null list_actions)
        nil
      (let ((aux1 (action-final action))
            (aux2 (action-cost action)))
        (cons (make-node : state aux1
                         : parent node
                         : action action
                         : depth (+ 1 (node-depth node))
                         : g (+ (node-g node) aux2)
                         : h (funcall (problem-f-h problem) aux1)
                         : f (+ (+ (node-g node) aux2) (funcall (problem-f-h problem) aux1)))
              (expand-node-aux node (rest list_actions) problem))))))
```

Pruebas

(build-actions (make-node :state 'Kentares :depth 0 :g 0 :f 0) (problem-operators *galaxy-M35*))

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

(defparameter lst-nodes-00 (expand-node node-03 *galaxy-M35*))
(print lst-nodes-00)

(expand-node (make-node :state 'Kentares :depth 0 :g 0 :f 0) *galaxy-M35*)

Comentarios

La novedad de este ejercicio es el uso de funcall para poder llamar a funciones que se encuentran dentro de estructuras en lugar de la llamada usual a una función que estamos acostumbrados a usar. Por otro lado, es un problema difícil de abordar en una misma función y se clarifica y se codifica mucho más rápido haciendo una subdivisión del problema, de ahí la creación de la función build-action.

Ejercicio 6

Pseudocódigo

NODE-G-<=

Entrada:
Node-1: primer nodo a comparar.
Node-2: segundo nodo a comparar.
Salida:
True si el primer nodo es menor o igual que el segundo, nil en otro caso.
Procesamiento:
En la salida queda explicado.

UNIFORM-COST

Campos:
Name: nombre del parámetro.
Node-compare-p: función de comparación usada.

INSERT-NODES-STRATEGY

Entrada:
Node: lista de nodos.
Lst-Nodes: Lista ordenada de nodos.
Strategy: estrategia usada para poner las listas en orden.
Salida:
Una lista ordenada con todos los nodos.
Procesamiento:

De nuevo manejamos un caso base muy sencillo en la función principal y todo el trabajo propio de la inserción de nodos se delega a la función auxiliar. Para saber el orden en el que se deben ir insertando los nodos se recurre a la función de igualdad anterior. Y se va haciendo una lista ordenada, sabiendo que una de las listas pasadas ya está ordenada y por tanto solo hay que introducir en su sitio correspondiente cada elemento de la otra lista.

Código

NODE-G-<=

```
(defun node-g-<= (node-1 node-2)
  (<= (node-g node-1)
      (node-g node-2)))
```

UNIFORM-COST

```
(defparameter *uniform-cost*
  (make-strategy
   :name 'uniform-cost
   :node-compare-p #'node-g-<=))
```

INSERT-NODES-STRATEGY-AUX

```
(defun insert-nodes-strategy-aux (node lst-nodes strategy)
  (cond
   ((null lst-nodes) (list node))
   ((funcall (strategy-node-compare-p strategy) node (first lst-nodes))
    (cons node lst-nodes))
   (t
    (cons (first lst-nodes) (insert-nodes-strategy-aux node (rest lst-nodes) strategy))))
```

INSERT-NODES-STRATEGY

```
(defun insert-nodes-strategy (nodes lst-nodes strategy)
  (if (null nodes)
      lst-nodes
    (insert-nodes-strategy (rest nodes) (insert-nodes-strategy-aux (first nodes) lst-nodes
strategy) strategy)))
```

Pruebas

```
(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*))
(print
 (insert-nodes-strategy (list node-00 node-01 node-02)
                    (sort (copy-list lst-nodes-00) #'<= :key #'node-g)
                    *uniform-cost*))
```

Comentarios

De nuevo el problema es bastante complejo si se trata de realizar en una sola función, nosotros recurrimos a dos funciones más, la auxiliar y la de menor o igual para poder insertar de manera más clara y visual. También por esta razón decidimos implementar una estructura uniform-cost.

2.3 Búsquedas

Ejercicio 7

Pseudocódigo

NODE-F-<=

Entrada:
Node-1: primer nodo a comparar.
Node-2: segundo nodo a comparar.
Salida:
True si la suma de g+h del primero es menor o igual que la del segundo, nil en otro caso.
Procesamiento:
Explicado en la salida.

A-STAR

Campos:
Name: nombre de la estrategia.
Node-compare-p: función de comparación usada.

Código

NODE-F-<=

```
(defun node-f-<= (node-1 node-2)
  (< (+ (node-g node-1) (node-h node-1))
     (+ (node-g node-2) (node-h node-2))))
```

A-STAR

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

Comentarios

Este ejercicio es una ejemplificación del anterior, haber realizado el anterior de manera tan clara, particionada y profunda hace que este apartado sea muy sencillo de realizar.

Ejercicio 8

Pseudocódigo

GRAPH-SEARCH

Entrada:

Problem: problema a resolver.

Strategy: estrategia que se usará para resolver el problema.

Salida:

Si no hay solución Nil. Si sí, un nodo que satisfaga el test objetivo.

Procesamiento:

El procesamiento de esta función y su auxiliar ha consistido en seguir el pseudocódigo facilitado.

FIND-DUPLICATES

Entrada:

Node: nodo del que buscamos repeticiones.

List: lista en la que buscaremos las posibles repeticiones del nodo.

Problem: problema que se está resolviendo.

Salida:

Devuelve la lista sin duplicados.

Procesamiento:

Consiste esencialmente en una llamada al predicado de igualdad del problema parseado de tal forma que nos devuelva la lista sin duplicados.

A-STAR-SEARCH

Entrada:

Problem: problema a resolver.

Salida:

Si no hay solución nil. Si sí, un nodo que satisfaga el test objetivo.

Procesamiento:

Llamada con los parámetros adecuados a la función de graph-search.

Código

GRAPH-SEARCH

```
(defun graph-search (problem strategy)
  (let ((state (problem-initial-state problem)))
    (graph-search-aux problem strategy (list (make-node :state state
                                                        :parent nil
                                                        :action nil))
          nil)))
```

FIND-DUPLICATES

```
(defun find-duplicates (node list problem)
  (let ((aux (first list)))
    (cond
     ((null list) nil)
     ((funcall (problem-f-search-state-equal problem) node aux) aux)
     (t (find-duplicates node (rest list) problem)))))
```

GRAPH-SEARCH-AUX

```
(defun graph-search-aux (problem strategy open-nodes closed-nodes)
  (unless (null open-nodes)
  (let ((actual (first open-nodes))
        (rep (find-duplicates (first open-nodes) closed-nodes problem)))
    (cond
     ((funcall (problem-f-goal-test problem) actual)
      actual)
     ((or (equal rep nil) (<= (node-g actual) (node-g rep)))
      (graph-search-aux problem
                        strategy
                        (insert-nodes-strategy (expand-node actual problem) (rest open-nodes)
strategy)
                        (cons actual closed-nodes)))
     (t (graph-search-aux problem
                          strategy
                          (rest open-nodes)
                          closed-nodes))))))
```

A-STAR-SEARCH

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

Pruebas

```
(defparameter node-03 (make-node :state 'Mallory :depth 0 :g 0 :f 0) )

(find-duplicates node-03 (list node-03) *galaxy-M35*)
(graph-search *galaxy-M35* *A-star*)
(print (a-star-search *galaxy-M35*))
```

Comentarios

De nuevo se aplica la regla de divide y vencerás, el pseudocódigo dado era muy largo y lo hemos subdividido el problema en varias funciones, la auxiliar es la que recoge toda la riqueza del pseudocódigo proporcionado en el enunciado.

Ejercicio 9

Pseudocódigo

SOLUTION-PATH

Entrada:
Node: nodo destino.
Salida:
Lista de los distintos planetas que han sido visitados.
Procesamiento:
Simplemente se va incluyendo en la lista cada nodo visitado.

ACTION-SEQUENCE

Entrada:
Node: nodo destino.
Salida:
Lista de acciones que se han realizado.
Procesamiento:
Misma idea que el pseudocódigo de justo arriba pero con acciones realizadas y no nodos.

Código

SOLUTION-PATH

```
(defun solution-path (node)
  (if (null node)
      nil
    (append (solution-path (node-parent node)) (list (node-state node)))))
```

ACTION-SEQUENCE

```
(defun action-sequence (node)
  (if (null node)
      nil
    (append (action-sequence (node-parent node)) (list (node-action node)))))
```

Pruebas

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

Comentarios

Ambas funciones son muy sencillas de implementar pero clarifican mucho al usuario acerca del funcionamiento del algoritmo usado y de los nodos encontrados y acciones realizadas.

Ejercicio 10

Pseudocódigo

DEPTH-FIRST-NODE-COMPARE-P

Entrada:
Node-1: primer nodo a comparar.
Node-2: segundo nodo a comparar.
Salida:
True si la profundidad del primero es mayor o igual que la del segundo, nil en otro caso.
Procesamiento:
El algoritmo de búsqueda en profundidad se basa en analizar primero los sucesores y cuando se acabe con estos, se continua con el otro nodo hijo. Por tanto la función de comparación entre nodos a elegir usa simplemente que el de mayor profundidad será el elegido primero para expandir.

BREADTH-FIRST-NODE-COMPARE-P

Entrada:
Node-1: primer nodo a comparar.
Node-2: segundo nodo a comparar.
Salida:
True si la profundidad del primero es menor o igual que la del segundo, nil en otro caso.
Procesamiento:
El algoritmo de búsqueda en anchura se basa en analizar primero los nodos con menor profundidad del árbol. Por tanto, la función de comparación entre nodos a elegir usa que el de menor profundidad será el elegido primero para expandir.

Código

DEPTH-FIRST STRATEGY

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

(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))

(defun depth-first-search (problem)
  (graph-search problem *depth-first*))
```

BREADTH-FIRST STRATEGY

```
(defun breadth-first-node-compare-p (node-1 node-2)
  NIL)

(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))

(defun breadth-first-search (problem)
  (graph-search problem *breadth-first*))
```

Pruebas

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

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

Comentarios

Solo se comenta el pseudocódigo de las funciones de comparación porque son las que tuvimos que implementar, el resto del código del ejercicio es similar al realizado para A* en el ejercicio 7. Se crea una estrategia depth-first y breadth-first y se enmascara la llamada a graph-search con las funciones depth-first-search y breadth-first-search.
Se hacen dos definiciones de las funciones de comparación tanto de profundidad como de anchura para demostrar que gracias a la manera en la que se ha realizado la función de graph-search las funciones de comparación de profundidad y anchura son tan simples como escribir T y nil respectivamente. Aunque la lógica que siguen ambas funciones de comparación son las expresadas en el pseudocódigo.

2.4 Preguntas - Ejercicio 11

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

Para resolver el problema del path-finder, se debe pensar en pequeño para después poder pensar en grande. Por ello, la práctica se ha dividido en varias secciones, una primera que corresponde a la modelización del problema, una segunda relacionada con la formalización del mismo y una tercera que se corresponde con la resolución del problema de búsqueda que es sobre la que versa esta pregunta.
La resolución principal del problema de búsqueda, que se consigue solucionar ya en el ejercicio 8, se corresponde con A*. Se usa A* porque manejamos una heurística gracias a

los sensores que se poseen en cada planeta y por tanto tenemos una cierta estimación del coste que puede ocasionarnos la búsqueda.
En el ejercicio 10, se plantean otras opciones mucho más genéricas como pueden ser primero en profundidad o primero en anchura, estos algoritmos también resuelven el problema aunque sin tener en cuenta la heurística y por tanto sin aprovecharnos de ciertas formalizaciones concretas de este problema como son los sensores.

2.4.1.1 ¿Qué ventajas aporta?

Básicamente esta subpregunta se debe resolver hablando de qué ventajas aporta la búsqueda informada con respecto a la búsqueda ciega ya que el algoritmo A* es un algoritmo de búsqueda informada y tanto primero en anchura como primero en profundidad son algoritmos de búsqueda ciega.
Pues bien, la búsqueda informada se basa en un conocimiento específico que se posee sobre el problema. Dicho conocimiento nos permite guiar la búsqueda para mejorar la eficiencia del algoritmo. En definitiva, la búsqueda informada es más eficiente pero más específica que la búsqueda ciega.
Eso sí, para poder aplicar A* se debe definir una serie de reglas que ayuden a dirigir la búsqueda, estas reglas son las que conocemos como heurística y que en este caso específico se posee gracias a la existencia de sensores en cada nodo o planeta que nos permiten guiar la búsqueda de manera adecuada.

2.4.1.2 ¿Por qué se han utilizado funciones lambda para especificar el test objetivo, la heurística y los operadores del problema?

Como hemos mencionado anteriormente, la búsqueda informada necesita de una serie de reglas (heurística) para resolver el problema. En este caso, se usan funciones lambda para especificar la heurística para que si queremos modificar el código con un nuevo método para calcular los costes simplemente se cambie la llamada a la función lambda.
Con la misma idea, para poder modificar los tests objetivos de la búsqueda se usa otra función lambda.
Y con un poco más de rigor, se usan funciones lambda para los operadores. Los operadores no son ni más ni menos que los enlaces entre nodos o planetas que se pueden usar. Gracias a una función lambda podremos añadir más enlaces o cambiar ciertos enlaces específicos y que siga funcionando adecuadamente nuestro código.
Por tanto, la respuesta genérica a este subapartado es que se usan funciones lambda para que el código se encuentre bien dividido y sea reciclable y reutilizable fácilmente.

2.4.2 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?

La idea de usar en cada nodo un campo parent es fundamental para poder conocer el camino que sigue nuestro algoritmo porque no hay otra forma de hacerlo. La otra manera de realizarlo que se nos ocurre, es guardando el camino entero pero el gasto de memoria sería mucho mayor ya que según avanzamos y expandimos nodos, el camino se hace mucho más grande.

Podemos decir que es la solución más eficiente aunque su gasto de memoria sea elevado, es la solución menos mala.

2.4.3 ¿Cuál es la complejidad espacial del algoritmo implementado?

Los algoritmos de búsqueda implementados son tres: A* , búsqueda en profundidad y búsqueda en anchura. Para todos, b se corresponde con el factor de ramificación (número máximo de sucesores de cualquier nodo).

Búsqueda A*

$O(b^{đ})\ con\ đ = C^*/\varepsilon$ siendo C* el coste óptimo y ε el mínimo coste por acción (para una heurística arbitraria).

Búsqueda Primero en Profundidad

$b * m + 1$ siendo m la profundidad máxima del árbol.

Búsqueda Primero en Anchura

$O(b^{d+1})$ siendo d la profundidad del nodo objetivo más superficial.

A partir de estos datos y dado un ejemplo específico se podrían sustituir los valores y obtener sus distintas complejidades espaciales.

2.4.4 ¿Cuál es la complejidad temporal del algoritmo?

Los algoritmos de búsqueda implementados son tres: A* , búsqueda en profundidad y búsqueda en anchura. Para todos, b se corresponde con el factor de ramificación (número máximo de sucesores de cualquier nodo).

Búsqueda A*

$O(b^{đ})\ con\ đ = C^*/\varepsilon$ siendo C* el coste óptimo y ε el mínimo coste por acción (para una heurística arbitraria).

Búsqueda en Profundidad

$O(b^m)$ siendo m la profundidad máxima del árbol.

Búsqueda en Anchura

$O(b^d)$ siendo d la profundidad del nodo objetivo más superficial.

A partir de estos datos y dado un ejemplo específico se podrían sustituir los valores y obtener sus distintas complejidades temporales.

2.4.5 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).

La solución más eficiente que se nos ocurre es la creación de una estructura para los agujeros de gusano que contenga un campo con el número de veces que se puede acceder al propio agujero. Debido a este pequeño cambio, se modifican todas las funciones que realicen acciones ya que para hacer un navigate de agujeros gusano, primero se deberá comprobar si se puede (si el campo contador es mayor que 0) y en caso de que se pueda y se realice la acción, decrementar dicho contador del agujero. Con estos cambios, se habría añadido la funcionalidad necesaria.

3. Conclusiones

A lo largo de la práctica nos hemos ido haciendo cada vez más expertos en el uso y en la creación de estructuras hasta el punto de parecernos natural la creación de las mismas para clarificar y simplificar el código. Asimismo, en los últimos ejercicios sobre todo, hemos recurrido a la división de la tarea principal pedida en pequeñas subtareas haciéndonos la vida mucho más fácil. También queremos destacar nuestro aprendizaje acerca de las funciones de LISP recurriendo a algunas como assoc o distinguiendo perfectamente lo que es un funcall de una llamada a una función más al uso.
Por otro lado, la práctica nos ha facilitado la compresión de los algoritmos de búsqueda dados en clase de teoría como el de A* o la búsqueda en profundidad y/o anchura. Gracias a los mismos, somos capaces de dar una solución automática y eficiente a los problemas de tipo path-finder que era en lo que consistía la práctica. También gracias a estos algoritmos de búsqueda hemos sido capaces de distinguir con claridad entre los conceptos genéricos de búsqueda ciega y búsqueda informada.