

# Inteligencia Artificial I

## 3ra. Práctica: Algoritmos de Búsqueda

### 1 Descripción general de la práctica

El objetivo de esta práctica es estudiar el comportamiento de diferentes algoritmos de búsqueda, para lo cual primero se deberán implementar los mismos.

La práctica consta de cuatro partes:

- A. Entender las funciones codificadas en LISP dadas en el enunciado.
- B. Implementar el método de búsqueda A\* para encontrar la trayectoria óptima. El argumento de entrada debe de ser una estructura del tipo *route-finding-problem* y debe retornar el número de iteraciones utilizadas para encontrar el camino óptimo, el costo del camino encontrado y las ciudades que lo componen:  
> (defun ASTAR (problem)  
> .....  
> .....)  
> retorna → (n costo camino)
- C. Implementar el método de búsqueda por **amplitud prioritaria** utilizando el ejemplo dado (**profundidad prioritaria**).
- D. Utilizando *Dobreta* como ciudad inicial (*initial-state*) y *Fagaras* como ciudad destino (*goal*) presentar los resultados obtenidos por cada uno de los métodos; esto es: lista de ciudades que componen cada camino, costo en kilómetros y número de iteraciones. Note que el código dado como ejemplo (profundidad prioritaria) no calcula el costo en kilómetros.
- E. Explicar las *ventajas* y *desventajas* de cada uno de los métodos.

**Nota 1:** el fichero con las funciones LISP correspondientes a esta práctica se encuentran en la página Web. También en la página Web se encuentra el fichero correspondiente a la búsqueda por profundidad prioritaria.

**Nota 2:** enviar respuestas para los puntos B y C en un único fichero LISP; Las respuestas para los puntos D y E pueden ser enviadas juntas en un fichero aparte o pueden ser incluidas como comentarios en el fichero LISP anterior (**no enviar archivos zip**).

### 2 Camino entre dos ciudades

Representaremos el problema de encontrar el camino en el mapa de Rumania que une dos ciudades dadas utilizando la siguiente información.

- initial-state: ciudad origen
- goal: ciudad destino

- map: mapa en el cual se realizará la búsqueda, obviamente contiene inicial-state & goal

La definición de una estructura en LISP se realiza de la siguiente manera:

```
(defstruct route-finding-problem
  inicial-state
  goal
  map)
```

El nombre de la estructura es route-finding-problem. Una instancia o variable del tipo estructura se definirá con make y luego el nombre de la estructura. En este caso será:

```
(setq p (make-route-finding-problem))
```

A partir de ahora p representa una instancia de route-finding-problem. En el caso en cuestión los campos de la estructura son inicializados con valores por defecto (ver route-finding-problem en el fichero *my-route-finding.lsp*). Esta inicialización también se puede realizar utilizando valores definidos por el usuario de la siguiente manera:

```
(setq p (make-route-finding-problem :initial-state 'NOMBRE_DE_LA_CIUADAD_ORIGEN
:goal 'NOMBRE_DE_LA_CIUADAD_DESTINO :map *ROMANIA-MAP*))
```

De esta última manera se podrá conseguir ejecutar los métodos de búsqueda para diferentes estados iniciales y diferentes objetivos.

El acceso a los campos de la estructura es muy sencillo. Solamente hay que utilizar el nombre del campo, como si fuese una función, y pasarle como argumento la instancia de estructura que queremos leer. Por ejemplo:

```
(route-finding-problem-initial-state p)
```

### 3. Comentarios sobre LISP

A continuación se comentarán algunas funciones de LISP que pueden ser de utilidad para solucionar de forma rápida algunas situaciones:

- SUBSTITUTE: substituye un valor por otro en una lista. No hace comparaciones en sublistas

```
; substituye el 4 por el 9
> (substitute 9 4 '(1 2 3 4 5))
> (1 2 3 9 5)
```

```
;substituye por 9 todos los atomos más pequeños que 3
> (substitute 9 3 '(1 2 4 1 3 4 5) : test #'>)
> (9 9 4 9 3 4 5)
```

```
;substitute no trabaja recursivamente
> (substitute 9 4 '(1 2 3 (4) 5))
(1 2 3 (4) 5)
```

- **SUBST**: trabaja como el **SUBSTITUTE** pero recursivamente:  

```
> (subst 9 4 '(1 2 3 (4) 5))
(1 2 3 (9) 5)
```
- **MERGE**: une dos listas ordenadas manteniendo el orden en el resultado. Es importante que las dos listas que se van a unir estén ya ordenadas. El primer argumento de la primitiva **MERGE** corresponde al tipo que queremos obtener como resultado. A continuación se deberán especificar las dos listas y finalmente, como un último argumento, se debe indicar el operador de comparación que se utilizará para ordenar los elementos. Se debe tener en cuenta que si alguno de los elementos de alguna de las dos listas no son átomos, estos operadores no funcionan. El problema de unir de forma ordenada dos listas ordenadas se puede resolver de diferentes maneras como se puede ver en el siguiente ejemplo:

```
; ordena de menor a mayor
> (merge 'list '(1 3 5)'(2 4 6) #'<)
(1 2 3 5 6 7)
```

si los elementos no son átomos se ha de indicar como un *:key* la operación que ha de realizarse sobre cada elemento de la lista antes de poder aplicar el operador de comparación. Esta modificación de la operación básica del **MERGE** cambiará en función de la estructura de anidamiento de nuestra lista.

```
> (merge 'list '((1 a) (3 b)(5 c)) '((2 d)(4 e)(6 f)) #'< : key #'car)
((1 a)(2 d)(3 b)(4 e)(5 c)(6 f))
```

el resultado anterior también se puede obtener utilizando una función *Lambda* en lugar de un comparador. En este caso, ya no hace falta el argumento *:key*. No obstante, esta sintaxis es más complicada que la anterior.

```
> (merge 'list '((1 a)(3 b)(5 c)) '((2 d)(4 e)(6 f))
#'(lambda (x y)(< (car x)(car y)) ))
((1 a)(2 d)(3 b)(4 e)(5 c)(6 f))
```

- **SORT**: ordena una lista. En general, el formato es el siguiente:  

```
(sort lista predicado : key s)
```

que es equivalente a:  

```
(sort lista #'(lambda (x y)( predicado (s x)(s y))))
```

Por ejemplo:

```
> (sort '(3 65 1 6) #'>)
(65 6 3 1)
```

En el siguiente caso no podremos comparar dos listas con “>”, sino que necesitaremos aplicar otra operación previa sobre cada elemento de la lista antes de la comparación. Igual que con el *merge*, tenemos dos sintaxis posibles:

```
>(sort '((3 (a b))(65 (c d))(1 (e f))(6 (g h))) #'> :key #'car)
((65 (c d))(6 (g h))(3 (a b))(1 (e f)))
```

```
> (sort '((3 ( a b))(65 (c d))(1 (e f))(6 (g h)))
#' (lambda (x y)(> (car x)(car y))))
((65 (c d))(6 (g h))(3 (a b))(1 (e f)))
```

- **INTERSECTION:** calcula la intersección de dos listas. Cuando la intersección se produce con una sublista, esta función no da el resultado esperado. Esto es así dado que para comparar los elementos de una lista con el de la otra lista para obtener la intersección, *intersection* utiliza el operador de igualdad *eql*. El operador *eql* no sirve para comparar listas, solamente para comparar átomos. Lo que se necesita es modificar la primitiva básica de *intersection* para poder utilizar el *equal* en lugar del *eql*. Esto último tiene sentido solamente si sabemos que nuestras listas contienen sublistas. Por ejemplo:

```
> (intersection '(1 2 3 4) '(3 4 5 6))
(3 4)
```

;en caso de que las listas contengan sublistas el resultado que se obtendrá será el siguiente:

```
> (intersection '(1 2 (3 4)) '((3 4) 5 6))
NIL
```

;esto último se puede solucionar de la siguiente manera:

```
(intersection '(1 2 (3 4)) '((3 4) 5 6) :test 'equal)
```

- **SET-DIFFERENCE:** dada dos listas retorna el/los elementos de la primera lista que no aparecen en la segunda. El problema comentado anteriormente con el *eql* en la primitiva *intersection*, también aparece aquí. La solución pasa por utilizar el *equal* al igual que en el ejemplo anterior:

```
> (set-difference '(1 2 3 4) '(4 5 6 7))
(1 2 3)
```

;utilizando el *eql* por defecto, ningún elemento de la primera sublista

;aparece en la segunda, todos son diferentes

; incluso la sublista (3 4)

```
> (set-difference '(1 2 (3 4)) '((3 4) 5 6))
(1 2 (3 4))
```

;esto se soluciona también con el *equal*

```
(set-difference '(1 2 (3 4)) '((3 4) 5 6) :test 'equal)
(1 2)
```

- ASSOC: busca que sublista del segundo argumento (que se espera que sea una lista asociada, es decir, una lista de sublista) comienza con el valor dado como primer argumento del assoc.  
 ;Solamente retorna la primera sublista que comienza con 1  
 ;si hay mas casos, como en el ejemplo que se presenta debajo, se perderán.  
 ;Por ejemplo, se perderá la sublista (1 20 28) la cual también comienza por un 1  
 (assoc 1 '((2 3 4) (1 c d) (5 6 6) (1 20 28)))  
 (1 c d)  
 ;el problema de la sublista y el eql por defecto  
 >(assoc '(1) '(((2) 3 4) ((1) 5 6) ((3) 7 8)))  
 NIL  
 ;se soluciona con el equal  
 >(assoc '(1) '(((2) 3 4) ((1) 5 6) ((3) 7 8)) :test 'equal)  
 ((1) 5 6)

#### **4. Código en LISP para el problema de la búsqueda del camino entre dos ciudades**

```

;;; -*- Mode: Lisp; Syntax: Common-LISP; -*- File: search/domains/route-finding
;;; Find a Route Between Cities on a Map
;;; Defining Problems
.....
(defstruct problem
  "A problem is defined by the initial state, and the type of problem it is.
  We will be defining subtypes of PROBLEM later on."
  (initial-state (required)) ; A state in the domain
  (goal nil) ; Optionally store the desired state here.
  (num-expanded 0) ; Number of nodes expanded in search for solution.
  (iterative? nil) ; Are we using an iterative algorithm?
)

;;; Defining the route finding Problems
.....

(defstruct (route-finding-problem (:include problem
  (initial-state 'Arad)
  (goal 'Bucharest)))
  "The problem of finding a route from one city to another on a map.
  A state in a route-finding problem is just the name of the current
  city. Note that a more complicated version of this problem would
  augment the state with considerations of time, gas used, wear on
  car, tolls to pay, etc."
  (map *Romania-map*))

.....

(defun successors (problem city-name)
  "Return a list of (action . new-state) pairs.
  In this case, the action and the new state are both the name of the city."
  (let ((result nil))

```

```

(setq interm (city-neighbors (find-city city-name problem)))
(do () ((equal (car interm) NIL) (mapcar 'first result) )
      (push (cons (first (first interm)) (first (first interm))) result)
      (setq interm (rest interm)))
)
)
)

```

```

.....

```

```

(defun edge-cost (problem current-city next-city)
  "The edge-cost is the road distance to the next city."
  (road-distance (find-city current-city problem) next-city)
)

```

```

.....

```

```

(defun h-cost (problem city-name)
  "The heuristic cost is the straight-line distance to the goal."
  (straight-distance (find-city city-name problem)
                    (find-city (problem-goal problem) problem)))

```

```

.....

```

```

;;; Inserta los sucesores en la lista de caminos

```

```

(defun inserta_sucesores (sucesores camino)
  (cond ((null sucesores) nil)
        (T (cons (cons (car sucesores) camino) (inserta_sucesores (cdr sucesores) camino))))
)
)

```

```

;----- loop?

```

```

;recibe una lista de la forma:

```

```

; (ORADEA ZERIND SIBIU ARAD)

```

```

;y return T if the first state (new added state already exists,

```

```

;y NIL otherwise

```

```

(defun loop? (one-expanded-state)
  (let ((st (car one-expanded-state)))
    (dolist (element (cdr one-expanded-state))
      (when (member st (list element)) (return t))))))

```

```

;----- eliminate-loops

```

```

;recibe una lista de la forma:

```

```

; ((FAGARAS SIBIU ARAD ZERIND) (RIMNICU SIBIU ARAD ZERIND))

```

```

;que representa todas las posibles expansiones del ultimo estado estudiado

```

```

;devuelve la misma lista habiendo eliminado aquellas sublistas que llevan a un ciclo

```

```

(defun eliminate-loops (expanded-states &aux expanded-states-without-loops (h nil))
  (dolist (cs expanded-states expanded-states-without-loops)
    (when (not (loop? cs)) (setq expanded-states-without-loops
                                (cons cs expanded-states-without-loops))))))

```

```

;;; The City and Map data structures

```

```

.....

```

```

(defstruct city (:type list))

```

```

"A city's loc (location) is an (x y) pair. The neighbors slot holds
a list of (city-name . distance-along-road) pairs. Be careful to

```



