P2 Inteligencia Artificial:

Búsqueda

Grupo 2361: Equipo 8

Guillermo Rodríguez Senís, Javier Muñoz Haro

Introducción

En esta práctica se nos pide desarrollar un buscador de caminos, un buscador de caminos es sí lo que su propio nombre indica, un sistema que con distintos sensores y heurísticas encuentra un camino en un grafo entre dos nodos. En esta práctica en particular se pone a nuestra disposición dos grafos los cuales tienen como nodos distintas ciudades de Francia y que están conectadas por aristas. En lo que se refiere a las aristas los dos grafos difieren bastante, mientras que uno representa la red de ferrocarril que conecta las distintas ciudades, el otro representa la red de canales que alberga Francia y que nos permitirá, mediante ellos ir de ciudad en ciudad.

Con las diferencias mencionadas previamente entre los dos grafos, debemos aportar una mas, y posiblemente la mas significativa, es que la red de ferrocarril(que es bidireccional) tiene ciudades "prohibidas", es decir, ciudades por las que no podemos pasar, mientras que los canales (que son unidireccionales ya que siguen la corriente del agua) pueden acceder a las mismas. Por otra parte también están las ciudades obligatorias, es decir, nodos por las que el recorrido del grafo deberá pasar.

Para determinar las heurísticas nos hemos apoyado en los costes que se nos definen en la práctica para viajar de ciudad en ciudad, es decir tiempo y coste (dinero), dependiendo de los diferentes transportes que sean utilizados. El buscador de caminos deberá, teniendo en cuenta las heurísticas, buscar el camino que nos cueste menos tiempo o/y dinero, sin pasar por las ciudades prohibidas a no ser que sea por canales, y pasando por las ciudades obligatorias.

En esta práctica se nos da un fichero como material de partida en el que se formaliza el modelo previamente definido. En él se nos dan:

- **Ciudades:** Las cuales son definidas como una lista que representan sus nombres.
- **Conexiones:** Las cuales representan los dos grafos, el de canales y el ferroviario, poniendo el origen, el destino, el tiempo de recorrido y el coste del uso de dicho transporte.
- **Estimaciones:** Pares formados por ciudades y valores correspondientes a las heurísticas (tiempo y coste).

- **Ciudad origen:** Es decir, la ciudad donde se empieza el viaje.
- **Ciudad destino:** Es decir, la ciudad donde se termina el viaje
- **Ciudades obligadas:** Aquellas ciudades por las que nuestro itinerario debe pasar.
- **Ciudades prohibidas:** Aquellas ciudades por las que no se puede acceder mediante tren.

Estructuras:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;    Problem definition
;;
(defstruct problem
  states               ; List of states
  initial-state        ; Initial state
  f-h                  ; function that evaluates the  value of the
                       ; heuristic of a state (either cost or time)
  f-goal-test          ; reference to a function that determines whether
                       ; a state fulfills the goal
  f-search-state-equal ; reference to a predicate that determines whether
                       ; two nodes are equal, in terms of their search state
  operators)           ; list of operators (references to functions) to
                       ; generate successors
;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;    Node in the search algorithm
;;
(defstruct node
     state        ; state label
     parent       ; parent node (in the paths that we build)
     action       ; action that generated the current node from its parent
     (depth 0)    ; depth in the search tree
     (g 0)        ; cost of the path from the initial state to this node
     (h 0)        ; value of the heuristic
     (f 0))       ; g + h
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;    Actions
;;
(defstruct action
     name         ; Name of the operator that generated the action
     origin       ; State on which the action is applied
     final        ; State that results from the application of the action
     cost )       ; Cost of the action
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;    Search strategies
;;
(defstruct strategy
     name              ; Name of the search strategy
     node-compare-p)   ; boolean comparison
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Ejercicio 1:

En este ejercicio tenemos que implementar dos funciones las cuales saquen las dos heurísticas que tenemos en nuestro sistema, tiempo y coste.

Las dos funciones son las siguientes:

```lisp
(defun f-h-time (state sensors)
  (if (null sensors)
    nil
    (if (eql state (first (first sensors)))
      (first (second (first sensors)))
      (f-h-time state (rest sensors))
    )
  )
 )
```

```lisp
(defun f-h-price (state sensors)
  (if (null sensors)
    nil
    (if (eql state (first (first sensors)))
      (second (second (first sensors)))
      (f-h-price state (rest sensors))
    )
  )
 )
```

Y se nos ponen los siguientes ejemplos:

```lisp
(f-h-time 'Nantes *estimate*) ; -> 75.0
(f-h-time 'Marseille *estimate*) ; -> 145.0
(f-h-time 'Lyon *estimate*) ; -> 105.0
(f-h-time 'Madrid *estimate*) ; -> NIL
```

Los cuales al ejecutarse en la consola de Clisp obtenemos:

```
[17]> (f-h-time 'Nantes *estimate*)
75.0
```

```
[18]> (f-h-time 'Marseille *estimate*)
145.0
```

```
[19]> (f-h-time 'Lyon *estimate*)
105.0
```

```
[20]> (f-h-time 'Madrid *estimate*)
NIL
```

Ejercicio 2:

Para este ejercicio tenemos que implementar cuatro funciones: navigate-canal-time, navigate-canal-price, navigate-train-time y navigate-train-price

Como vemos en la imagen, hemos implementado las funciones que se nos piden, y por otra parte la función auxiliar de forbidden-state, la cual chequea si la ciudad destino es una ciudad prohibida y por tanto no es accesible.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Navigation by canal
;;
;; This is a specialization of the general navigation function: given a
;; state and a list of canals, returns a list of actions to navigate
;; from the current city to the cities reachable from it by canal navigation.
;;

;;lista de ciudades a las que se piuede llegar desde state buscando por canales solo, devolviendo el tiempo que tarda
(defun navigate-canal-time (state canals)
  (navigate state canals #'first 'NAVIGATE-CANAL-TIME)
)

;;lista de ciudades a las que se piuede llegar desde state buscando por canales solo, devolviendo el precio que cuesta
(defun navigate-canal-price (state canals)
 (navigate state canals #'second 'NAVIGATE-CANAL-PRICE)
)
;;;;;;;;;;;;;;;PRIVATE FUNCTION BY GUILLERMO AND JAVI forbidden-state
(defun forbidden-state (state forbidden)
  (if (null forbidden)
    T
    (if (eql (first forbidden) state)
      NIL
      (forbidden-state state (rest forbidden))
    )
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Navigation by train
;;
;; This is a specialization of the general navigation function: given a
;; state and a list of train lines, returns a list of actions to navigate
;; from the current city to the cities reachable from it by train.
;;
;; Note that this function takes as a parameter a list of forbidden cities.
;;
(defun navigate-train-time (state trains forbidden)
  (navigate state trains #'first 'NAVIGATE-TRAIN-TIME forbidden)
)

(defun navigate-train-price (state trains forbidden)
  (navigate state trains #'second 'NAVIGATE-TRAIN-PRICE forbidden)
)
```

Todas las funciones previamente implementadas son auxiliares con respecto a la función troncal, que es navigate, esta función es la que decide, dependiendo de la situación en la que nos encontremos del recorrido que acción tomar.

```
;;
(defun navigate (state lst-edges cfun name &optional forbidden )
  (if (null lst-edges)
    NIL
    (if (and (eql (first(first lst-edges)) state) (forbidden-state (second(first lst-edges)) forbidden) )
      (cons (make-action :name name
                         :origin state
                         :final (second(first lst-edges))
                         :cost (funcall cfun (third(first lst-edges)) ) ) (navigate state (rest lst-edges) cfun name forbidden))
      (navigate state (rest lst-edges) cfun name forbidden)
    )
  )
)
```

Cuando ejecutamos la batería de ejemplos proporcionada nos salen los siguientes resultados:

Para estos ejemplos, obtenemos:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;
;; Exercise 2:
;;
;;

(navigate-canal-time 'Avignon *canals*) ;->
;(#S(ACTION :NAME NAVIGATE-CANAL-TIME
;           :ORIGIN AVIGNON
;           :FINAL MARSEILLE
;           :COST 35.0))

(navigate-train-price 'Avignon *trains* '()) ;->
;(#S(ACTION :NAME NAVIGATE-TRAIN-PRICE
;           :ORIGIN AVIGNON
;           :FINAL LYON
;           :COST 40.0)
; #S(ACTION :NAME NAVIGATE-TRAIN-PRICE
;           :ORIGIN AVIGNON
;           :FINAL MARSEILLE
;           :COST 25.0))


(navigate-train-price 'Avignon *trains* '(Marseille)) ;->
;(#S(ACTION :NAME NAVIGATE-TRAIN-PRICE
;           :ORIGIN AVIGNON
;           :FINAL LYON
;           :COST 40.0))

(navigate-canal-time 'Orleans *canals*)  ;-> NIL
```

Ejemplo 1:

```
[30]> (navigate-canal-time 'Avignon *canals*)
(#S(ACTION :NAME NAVIGATE-CANAL-TIME :ORIGIN AVIGNON :FINAL MARSEILLE :COST 35.0))
```

Ejemplo2:

```
[31]> (navigate-train-price 'Avignon *trains* '())
(#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN AVIGNON :FINAL LYON :COST 40.0)
 #S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN AVIGNON :FINAL MARSEILLE :COST 25.0))
```

Ejemplo 3:

```
[32]> (navigate-train-price 'Avignon *trains* '(Marseille))
(#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN AVIGNON :FINAL LYON :COST 40.0))
```

Ejemplo 4:

```
[33]> (navigate-canal-time 'Orleans *canals*)
NIL
```

Ejercicio 3:

Para este ejercicio implementamos la función f-goal-test. Esta función determinará si la ciudad a la que hemos llegado es en realidad la ciudad objetivo, es decir, el final de nuestro trayecto.

En nuestra implementación, pese a pedir solo realizar una función, hemos tenido que realizar 3 funciones auxiliares para modularizar la funcionalidad:

```
;;FUNCION PRIVADA. Indica si el state de nodo se encuentra en la lista destination (T o NIL)
(defun node-in-destination (node destination)
  (if (null destination)
    NIL
    (if (eql (node-state node) (first destination))
      T
      (node-in-destination node (rest destination))
    )
  )
)

;;FUNCION PRIVADA (recursiva para path-check-mandatory). Indica si en el path formado por node y sus padres esta el state mandatory
(defun mandatory-in-path(node mandatory)
  (if (null node)
    NIL
    (if (eql (node-state node) mandatory)
      T
      (mandatory-in-path (node-parent node) mandatory)
    )
  )
)

;;FUNCION PRIVADA.  Indica si el path formado por node y sus padres pasa por todos los states de la lista mandatory (T o NIL)
(defun path-check-mandatory(node mandatory)
  (if (null mandatory)
    T
    (and
      (mandatory-in-path node (first mandatory))
      (path-check-mandatory node (rest mandatory))
    )
  )
)
```

Y luego nuestra función troncal, que llama a las demás:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Goal test
;
;  Returns T or NIl depending on whether a path leads to a final state
;
;  Input:
;    node:       node structure that contains, in the chain of parent-nodes,
;                a path starting at the initial state
;    destinations: list with the names of the destination cities
;    mandatory:  list with the names of the cities that is mandatoryu to visit
;
;  Returns
;    T: the path is a valid path to the final state
;    NIL: invalid path: either the final city is not a destination or some
;         of the mandatory cities are missing from the path.
;
defun f-goal-test (node destination mandatory)
 (and
   (node-in-destination node destination)
   (path-check-mandatory node mandatory)
 )
```

La batería de pruebas que se nos proporciona en este ejercicio es la siguiente:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;
; Exercise 3:
;
;



defparameter node-nevers
  (make-node :state 'Nevers) )
defparameter node-paris
  (make-node :state 'Paris :parent node-nevers))
defparameter node-nancy
  (make-node :state 'Nancy :parent node-paris))
defparameter node-reims
  (make-node :state 'Reims :parent node-nancy))
defparameter node-calais
  (make-node :state 'Calais :parent node-reims))


f-goal-test node-calais '(Calais Marseille) '(Paris Limoges)); -> NIL
f-goal-test node-paris '(Calais Marseille) '(Paris)); -> NIL
f-goal-test node-calais '(Calais Marseille) '(Paris Nancy)); -> T
```

Una vez que cargamos los parámetros, ejecutamos en el interprete de Clisp y obtenemos:

Ejemplo 1:

```
[41]> (f-goal-test node-calais '(Calais Marseille) '(Paris Limoges))
NIL
```

Ejemplo 2:

```
[42]> (f-goal-test node-paris '(Calais Marseille) '(Paris))
NIL
```

Ejemplo 3:

```
[43]> (f-goal-test node-calais '(Calais Marseille) '(Paris Nancy))
T
```

Como vemos, los resultados esperados coinciden con los obtenidos.

Ejercicio 4

En este ejercicio, se nos pide implementar una función que básicamente compare los nodos (los cuales representan las ciudades) y los estados de los mismos, de acuerdo a la funcionalidad de búsqueda.

Dos nodos son iguales bajo las condiciones de que ambos “representen” la misma ciudad y que la lista de ciudades pendientes por visitar coincida.

Nuestra implementación del código se divide en dos funciones, como hemos hecho previamente, para dividir la funcionalidad.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN: Exercise 4 -- Equal predicate for search states
;;

;;FUNCION PRIVADA. Retorna una lista con las ciudades de la lista mandatory que quedan por visitar
(defun missing-mandatory (node mandatory)
  (if (null mandatory)
    NIL
    (if (mandatory-in-path node (first mandatory)) ;comprueba que firstmandatory esta en el path de node
      (missing-mandatory node (rest mandatory))
      (cons (first mandatory) (missing-mandatory node (rest mandatory)))
    )
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Determines if two nodes are equivalent with respect to the solution
;; of the problem: two nodes are equivalent if they represent the same city
;, and if the path they contain includes the same mandatory cities.
;;  Input:
;;    node-1, node-1: the two nodes that we are comparing, each one
;;                    defining a path through the parent links
;;    mandatory:  list with the names of the cities that is mandatory to visit
;;
;;  Returns
;;    T: the two ndoes are equivalent
;;    NIL: The nodes are not equivalent
;;
(defun f-search-state-equal (node-1 node-2 &optional mandatory)
  (if (not (eql (node-state node-1) (node-state node-2)))
    NIL
    (if (equal (missing-mandatory node-1 mandatory) (missing-mandatory node-2 mandatory))
      T
      NIL
    )
  )
)

;;
;; END: Exercise 4 -- Equal predicate for search states
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

La batería de datos proporcionada para este ejercicio es la siguiente:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;
;; Exercise 4:
;;
;;

(defparameter node-calais-2
   (make-node :state 'Calais :parent node-paris))

(f-search-state-equal node-calais node-calais-2 '()) ;-> T
(f-search-state-equal node-calais node-calais-2 '(Reims)) ;-> NIL
(f-search-state-equal node-calais node-calais-2 '(Nevers)) ;-> T
(f-search-state-equal node-nancy node-paris '()) ;-> NIL
```

Y los resultados que obtenemos al ejecutar esos ejemplos son los siguientes:

```
[47]> (f-search-state-equal node-calais node-calais-2 '())
T
[48]> (f-search-state-equal node-calais node-calais-2 '(Reims))
NIL
[49]> (f-search-state-equal node-calais node-calais-2 '(Nevers))
T
[50]> (f-search-state-equal node-nancy node-paris '())
NIL
```

Ejercicio 5

Para este problema, tenemos que crear dos estructuras para formalizar el problema de tiempo mínimo y precio mínimo (que son básicamente las dos heurísticas que hemos definido, y que cuanto menor valor tengan, mejor será el camino que vayamos encontrando).

Las estructuras que son proporcionadas son las siguientes:

```
(defparameter *travel-cheap*
  (make-problem
    :states *cities*
    :initial-state *origin*
    :f-h                    #'(lambda (state) (f-h-price state *estimate*))
    :f-goal-test            #'(lambda (node) (f-goal-test node *destination* *mandatory*))
    :f-search-state-equal   #'(lambda (node-1 node-2) (f-search-state-equal node-1 node-2 *mandatory*))
    :operators (list
                            #'(lambda (node) (navigate-canal-price node *canals*))
                            #'(lambda (node) (navigate-train-price node *trains* *forbidden*)))))

(defparameter *travel-fast*
  (make-problem
    :states *cities*
    :initial-state *origin*
    :f-h                    #'(lambda (state) (f-h-time state *estimate*))
    :f-goal-test            #'(lambda (node) (f-goal-test node *destination* *mandatory*))
    :f-search-state-equal   #'(lambda (node-1 node-2) (f-search-state-equal node-1 node-2 *mandatory*))
    :operators (list
                            #'(lambda (node) (navigate-canal-time node *canals*))
                            #'(lambda (node) (navigate-train-time node *trains* *forbidden*)))))

;;
;;  END: Exercise 5 -- Define the problem structure
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Como podemos observar y se explica en el enunciado de la práctica, cada estructura tiene una función heurística y dos operadores (tiempo y coste). (Para este ejercicio no hay batería de ejemplos).

Ejercicio 6

La realización de este ejercicio consiste en expandir un nodo. Expandir un nodo es realizar una operación que consiste en, dado un nodo, realizar una lista, correspondiente a un estado en el que se pueden alcanzar los demás, partiendo del nodo actual.

Para hacer esto en el problema se nos pide implementar la función expand-node.

A esta función se le pasa una estructura nodo y una estructura problema. La estructura tiene una lista de operadores, que básicamente nos da información sobre los estados que se pueden alcanzar si llegásemos a ellos.

Para crear los nodos que se pueden alcanzar desde el nodo actual iteramos con expand-node, pero es esta función, expand-node-action la que se encarga de crear la estructura node que representa el nodo al que se puede llegar desde el nodo actual

```
(defun expand-node-action (node problem action)
  (if (or (null action) (null problem))
    NIL
    (let ( (g (+ (node-g node) (action-cost action))) (h (funcall (problem-f-h problem) (action-final action))) )
      (make-node  :state (action-final action)
                  :parent node
                  :action action
                  :depth (+ 1 (node-depth node))
                  :g g
                  :h h
                  :f (+ g h)
      )
    )
  )
)
```

Nuestra función principal es, como hemos dicho antes, expand-node, la cual devuelve la lista de nodos de donde se puede acceder dado el nodo actual.

```
(defun expand-node (node problem)
  (if (or (null node) (null problem))
    NIL
    (let     ;guarda en la lista actions las salidas de las llamadas a los operators de problem con nodestate como argumento
      ( (actions (mapcan #'(lambda (fun) (funcall fun (node-state node))) (problem-operators problem))) )
      (mapcar #'(lambda (action) (expand-node-action node problem action)) actions)
    ) ;traducimos todas estas acciones a nodos
  )
)
```

La batería de pruebas que proporciona este ejercicio es la siguiente:

```
(defparameter node-marseille-ex6
   (make-node :state 'Marseille :depth 12 :g 10 :f 20) )

(defparameter lst-nodes-ex6
  (expand-node node-marseille-ex6 *travel-fast*))

(print lst-nodes-ex6) ; ->
;(#S(NODE :STATE TOULOUSE
;         :PARENT #S(NODE
;                    :STATE
;                    MARSEILLE
;                    :PARENT
;                    NIL
;                    :ACTION
;                    NIL
;                    :DEPTH
;                    12
;                    :G
;                    10
;                    :H
;                    0
;                    :F
;                    20)
;         :ACTION #S(ACTION
;                    :NAME
;                    NAVIGATE-TRAIN-TIME
;                    :ORIGIN
;                    MARSEILLE
;                    :FINAL
;                    TOULOUSE
;                    :COST
;                    65.0)
;         :DEPTH 13
;        :G 75.0
;         :H 130.0
;         :F 205.0))
;(#S(NODE :STATE TOULOUSE
;         :PARENT #S(NODE
;                    :STATE
;                    MARSEILLE
;                    :PARENT
;                    NIL
;                    :ACTION
;                    NIL
;                    :DEPTH
;                    12
;                    :G
;                    ...)
;        :ACTION #S(ACTION
;                   :NAME
;                    NAVIGATE-TRAIN-TIME
;                    :ORIGIN
;                    MARSEILLE
;                    :FINAL
;                    TOULOUSE
;                    :COST
;                    65.0)
;         :DEPTH 13
;         :G ...))
```

Y estos son los resultados obtenidos:

```
Break 1 [55]> (defparameter lst-nodes-ex6
  (expand-node node-marseille-ex6 *travel-fast*))
LST-NODES-EX6
Break 1 [55]> (print LST-NODES-EX6)

(#S(NODE :STATE TOULOUSE :PARENT #S(NODE :STATE MARSEILLE :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0 :F 20)
    :ACTION #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) :DEPTH 13 :G 75.0 :H 130.0 :F 205.0))
(#S(NODE :STATE TOULOUSE :PARENT #S(NODE :STATE MARSEILLE :PARENT NIL :ACTION NIL :DEPTH 12 :G 10 :H 0 :F 20)
    :ACTION #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) :DEPTH 13 :G 75.0 :H 130.0 :F 205.0))
Break 1 [55]>
```

Ejercicio 7

La función por implementar, insert-nodes-strategy, se encarga de insertar los nodos siguiendo un criterio específico (estrategia).

Hemos creado una función privada, llamada insert-node, la cual inserta el nodo en la lista ordenada sin desordenar esta.

```
(defun insert-node (node lst-nodes node-compare-p)
  (if (null lst-nodes)  ;lista vacia retorna lista con node
    (cons node lst-nodes)
    (if (funcall node-compare-p node (first lst-nodes)); si cumple la comparacion d
      (cons node lst-nodes)
      (cons (first lst-nodes) (insert-node node (rest lst-nodes) node-compare-p)) ;
    )                                                                              ;
  )
)
```

La función privada le sirve a la función insert-nodes para mantener el orden de los nodos insertados.

```
(defun insert-nodes (nodes lst-nodes node-compare-p)
  (if (null nodes)
    lst-nodes
    (insert-nodes (rest nodes) (insert-node (first nodes) lst-nodes node-compare-p) node-compare-p)
  )
)
```

Po último, tenemos la función troncal, la cual tiene como entrada una lista de nodos ordenada, otra lista de nodos (potencialmente desordenados) y una estrategia. Esta función devuelve la lista de nodos ordenada, tal que:

```
(defun insert-nodes-strategy (nodes lst-nodes strategy)
  (if (null strategy)
    NIL
    (insert-nodes nodes lst-nodes (strategy-node-compare-p strategy)))
)
```

Ejemplo de prueba:

Entrada:

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

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

(defparameter node-paris-ex7
  (make-node :state 'Paris :depth 0 :g 0 :f 0) )

(defparameter node-nancy-ex7
  (make-node :state 'Nancy :depth 2 :g 50 :f 50) )


(defparameter sol-ex7 (insert-nodes-strategy (list node-paris-ex7 node-nancy-ex7)
                                             lst-nodes-ex6
                                             *uniform-cost*))

(mapcar #'(lambda (x) (node-state x)) sol-ex6) ; -> (PARIS NANCY TOULOUSE)
(mapcar #'(lambda (x) (node-g x)) sol-ex6) ; -> (0 50 75)
```

Salida 1:

```
Break 3 [57]> (mapcar #'(lambda (x) (node-state x)) sol-ex7)
(PARIS NANCY TOULOUSE)
```

Salida 2:

```
Break 3 [57]> (mapcar #'(lambda (x) (node-g x)) sol-ex7)
(0 50 75.0)
```

Ejercicio 8

Este ejercicio consiste básicamente en definir un algoritmo para la búsqueda A*, el cual definimos en nuestro fichero:

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

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