PRÁCTICA 2. INTELIGENCIA ARTIFICIAL.

COMMON LISP
BÚSQUEDA

Gloria del Valle Cano & Leyre Canales Antón
gloria.valle@estudiante.uam.es & leyre.canales@estudiante.uam.es

Índice

Listings

1. Introducción

En esta práctica se propone desarrollar un diseño en LISP para viajar a través de Francia hasta llegar a Calais utilizando el problema de búsqueda de caminos mínimos. Para el desarrollo de la misma nos hemos ceñido a las explicaciones dadas en el enunciado de la práctica y a las estructuras y variables globales facilitadas para representar los conceptos más relevantes de nuestro programa.

Más en detalle, el desarrollo de la práctica se enmarca en las siguientes estructuras:

1. Node, representando un estado en el camino hacia el objetivo.
2. Problem, representando los parámetros asociados al problema en concreto, así como la estrategia de búsqueda a optimizar.
3. Action, representando una acción de ir de un lugar a otro ya sea por canal o por tren.
4. Strategy, que contiene una función para comparar dos nodos, favoreciendo el mejor nodo según el criterio concreto.

Los primeros ejercicios consisten fundamentalmente en la implementación de funciones auxiliares que nos ayudarán a acceder a los distintos campos de las estructuras, crear las acciones pertinentes, decidir cuándo dos nodos son iguales, etc. Forman el pilar necesario para las siguientes funciones, que recogen la mayor parte de la lógica de la aplicación, en concreto:

1. `expand-node`, la función encargada de, dado un nodo, explorar todos los nodos a los que se puede llegar desde el mismo.
2. `insert-nodes-strategy`, que se encarga de insertar los nodos siguiedno el orden dado por el criterio de búsqueda.
3. `graph-search`, la función central que se encarga de la exploración exhaustiva del grafo.

2. Ejercicio 1

En este ejercicio implementamos funciones para recuperar el coste y el tiempo de los distintos trayectos. Para ello hemos implementado una función auxiliar que nos permite unificar la sintaxis. Esta función obtiene el campo de las heurísticas y, según sea para tiempo o para coste, recuperamos `first` o el `second`.

```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; Returns the value of the heuristics for a given state
3  ;;
4  ;;  Input:
5  ;;    state: the current state (vis. the planet we are on)
6  ;;    sensors: a sensor list, that is a list of pairs
7  ;;                (state (time-est cost-est) )
8  ;;             where the first element is the name of a state and the second
9  ;;             a number estimating the costs to reach the goal
10 ;;
11 ;;  Returns:
12 ;;    The cost (a number) or NIL if the state is not in the sensor list
13 ;;
14 ;;  It is necessary to define two functions: the first which returns the
15 ;;  estimate of teh travel time, the second which returns the estimate of
16 ;;  the cost of travel
```

```
17 (defun f-h-time (state sensors)
18   (first (f-h-aux state sensors)))
19
20 (defun f-h-price (state sensors)
21   (second (f-h-aux state sensors)))
22
23 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
24 ;;
25 ;; Function f-h: Exercise 1
26 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
27 (defun f-h-aux (state sensors)
28   (cond
29     ((null sensors) nil)
30     ((equal (first (first sensors)) state)
31       (second (first sensors)))
32     (t (f-h-aux state (rest sensors)))))
```

El resultado de nuestro programa es el siguiente:

```
CL-USER 324 : 4 > (f-h-time 'Nantes *estimate*)
75.0

CL-USER 325 : 4 > (f-h-time 'Marseille *estimate*)
145.0

CL-USER 326 : 4 > (f-h-time 'Lyon *estimate*)
105.0

CL-USER 327 : 4 > (f-h-time 'Madrid *estimate*)
NIL
```

3. Ejercicio 2

En este segundo ejercicio tenemos que codificar funciones que devuelvan las posibles acciones desde un estado dado. Para ello recorremos las listas de `train` o `canals`, seleccionando aquellas entradas en las que el estado inicial coincida. Hay que tener en cuenta que en el caso de los trenes no podemos acceder a las ciudades en la lista `*forbidden*`. En este ejercicio hemos implementado una función auxiliar común `navigate` que nos sirve para no duplicar código, puesto que la lógica es muy similar en todos los casos.

```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;;
3  ;; General navigation function
4  ;;
5  ;;  Returns the actions that can be carried out from the current state
6  ;;
7  ;;  Input:
8  ;;    state:       the state from which we want to perform the action
9  ;;    lst-edges:   list of edges of the graph, each element is of the
10 ;;                 form: (source destination (cost1 cost2))
11 ;;    c-fun:       function that extracts the correct cost (time or price)
12 ;;                 from the pair that appears in the edge
13 ;;    name:        name to be given to the actions that are created (see the
14 ;;                 action structure)
15 ;;    forbidden-cities:
16 ;;                 list of the cities where we can't arrive by train
17 ;;
18 ;;  Returns
19 ;;    A list of action structures with the origin in the current state and
```

```
20 ;;    the destination in the states to which the current one is connected
21 ;;
22 (defun navigate (state lst-edges cfun name &optional forbidden)
23   (if (or (null state) (null lst-edges))
24       nil
25     (mapcar #' (lambda (x)
26                                  (make-action
27                                   :name name
28                                   :origin state
29                                   :final (second x)
30                                   :cost (funcall cfun (third x))))
31
32             (remove-if
33              #'(lambda (x)
34                  (or (equal (second x) (first forbidden))
35                      (not (equal state (first x))))) lst-edges))))
36
37
38 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
39 ;;
40 ;; Navigation by canal
41 ;;
42 ;; This is a specialization of the general navigation function: given a
43 ;; state and a list of canals, returns a list of actions to navigate
44 ;; from the current city to the cities reachable from it by canal navigation.
45 ;;
46 (defun navigate-canal-time (state canals)
47   (navigate state canals #'(lambda (cost) (first cost)) 'navigate-canal-time nil))
48
49 (defun navigate-canal-price (state canals)
50   (navigate state canals #'(lambda (cost) (first (rest cost))) 'navigate-canal-price nil))
51
52 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
53 ;;
54 ;; Navigation by train
55 ;;
56 ;; This is a specialization of the general navigation function: given a
57 ;; state and a list of train lines, returns a list of actions to navigate
58 ;; from the current city to the cities reachable from it by train.
59 ;;
60 ;; Note that this function takes as a parameter a list of forbidden cities.
61 ;;
62 (defun navigate-train-time (state trains forbidden)
63  (navigate state trains #'(lambda (cost) (first cost)) 'navigate-train-time forbidden))
64
65 (defun navigate-train-price (state trains forbidden)
66   (navigate state trains #'(lambda (cost) (first (rest cost))) 'navigate-train-price forbidden))
```

La salida de nuestro programa es la siguiente:

```
CL-USER 328 : 4 > (navigate-canal-time 'Avignon *canals*)
(#S(ACTION :NAME NAVIGATE-CANAL-TIME :ORIGIN AVIGNON :FINAL MARSEILLE :COST 35.0))

CL-USER 329 : 4 > (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))

CL-USER 330 : 4 > (navigate-train-price 'Avignon *trains* '(Marseille))
(#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN AVIGNON :FINAL LYON :COST 40.0))

CL-USER 331 : 4 > (navigate-canal-time 'Orleans *canals*)
NIL
```

4. Ejercicio 3

La función `f-goal-test` debe comprobar si el estado actual cumple las condiciones de ser el objetivo. Para ello:

1. El estado del nodo actual debe coincidir con el estado objetivo.
2. El camino que ha llevado al nodo actual debe contener las ciudades obligatorias.

La lógica de la función consiste por tanto en recuperar el camino hasta el nodo actual, para lo que hemos escrito una función `collect-states`, que, dado un nodo, recorre sus ancestros recursivamente. La otra parte de la lógica es llevada a cabo por la función `cities-in-path-p`, que, dada una lista de ciudades obligatorias y un camino, comprueba que todas están contenidas.

```
1
2  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
3  ;;
4  ;; Goal test
5  ;;
6  ;;  Returns T or NIl depending on whether a path leads to a final state
7  ;;
8  ;;  Input:
9  ;;    node:        node structure that contains, in the chain of parent-nodes,
10 ;;                 a path starting at the initial state
11 ;;    destinations: list with the names of the destination cities
12 ;;    mandatory:  list with the names of the cities that is mandatoryu to visit
13 ;;
14 ;;  Returns
15 ;;    T: the path is a valid path to the final state
16 ;;    NIL: invalid path: either the final city is not a destination or some
17 ;;         of the mandatory cities are missing from the path.
18 ;;
19 (defun f-goal-test (node destination-cities mandatory-cities)
20   (and (member (node-state node) destination-cities)
21        (cities-in-path-p (collect-states (node-parent node))
22                          mandatory-cities)))
23
24 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
25 ;; Aux-rec functions -- Exercise 3
26 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
27 (defun cities-in-path-p (path cities)
28   (equal (length cities)
29          (length (remove-if #'not
30                             (mapcar #'(lambda (city)
31                                         (member city path))
32                                     cities)))))
33
34 (defun collect-states (parent &optional states)
35   (if (null parent)
36       states
37       (collect-states (node-parent parent)
38                       (cons (node-state parent)
39                             states))))
```

Creamos los siguientes nodos para probar nuestro código:

```
(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))
```

Obteniendo como resultado la salida a continuación:

```
CL-USER 337 : 4 > (f-goal-test node-calais '(Calais Marseille) '(Paris Limoges))
NIL

CL-USER 338 : 4 > (f-goal-test node-paris '(Calais Marseille) '(Paris))
NIL

CL-USER 339 : 4 > (f-goal-test node-calais '(Calais Marseille) '(Paris Nancy))
T
```

5. Ejercicio 4

En este ejercicio se codifica una función que comprueba si dos nodos son iguales de acuerdo con su estado de búsqueda. Dos nodos son iguales si:

1. Su estado es igual.
2. Su camino coincide en las mismas ciudades obligatorias.

Para su implementación primero comprobamos que ambos estados coincidan, y posteriormente si sus caminos contienen las mismas ciudades obligatorias. Para esto último hemos implementado una función `path-contains-same-cities-p`, que recibe dos caminos y la lista de ciudades obligatorias. También hemos escrito una función auxiliar `member-p`, que devuelve `T` si `elt` está en `lst`, y `NIL` en caso contrario.

```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;;
3  ;; Determines if two nodes are equivalent with respect to the solution
4  ;; of the problem: two nodes are equivalent if they represent the same city
5  ;, and if the path they contain includes the same mandatory cities.
6  ;;  Input:
7  ;;    node-1, node-1: the two nodes that we are comparing, each one
8  ;;                    defining a path through the parent links
9  ;;    mandatory:  list with the names of the cities that is mandatory to visit
10 ;;
11 ;;  Returns
12 ;;    T: the two ndoes are equivalent
13 ;;    NIL: The nodes are not equivalent
14 ;;
15 (defun f-search-state-equal (node1 node2 &optional mandatory)
16   (when (equal (node-state node1)
17                (node-state node2))
18     (let ((path1 (collect-states node1))
19           (path2 (collect-states node2)))
20           (or (null mandatory)
21               (path-contains-same-cities-p path1 path2 mandatory)))))
22
23 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
24 ;; Aux-rec functions -- Exercise 4
25 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
26 (defun path-contains-same-cities-p (path1 path2 mandatory)
```

```
27    (every #'(lambda (x) (equal x t))
28           (mapcar #'(lambda (mandatory-city)
29                               (let ((is-in-path1 (member-p mandatory-city path1))
30                                     (is-in-path2 (member-p mandatory-city path2)))
31                                    (or (and is-in-path1 is-in-path2)
32                                        (and (not is-in-path1)
33                                             (not is-in-path2)))))
34                       mandatory)))
35
36  (defun member-p (elt lst)
37    (not (null (member elt lst))))
```

Hemos definido este nodo para probar el ejercicio:

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

Habiendo obtenido esta salida:

```
CL-USER 341 : 4 > (f-search-state-equal node-calais node-calais-2 '())
T

CL-USER 342 : 4 > (f-search-state-equal node-calais node-calais-2 '(Reims))
NIL

CL-USER 343 : 4 > (f-search-state-equal node-calais node-calais-2 '(Nevers))
T

CL-USER 344 : 4 > (f-search-state-equal node-nancy node-paris '())
NIL
```

6. Ejercicio 5

Iniciamos el valor de las estructuras `*travel-cheap*` y `*travel-fast*` de tal manera que representen los problemas de interés. Nótese que las funciones se escriben como *lambdas*, de tal manera que el algoritmo sea agnóstico a la parametrización del mismo.

```
1   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2   ;;  Note that the connectivity of the netowrk using canals and trains
3   ;;  holes is implicit in the operators: there is a list of two
4   ;;  operators, each one takes a single parameter: a state name, and
5   ;;  returns a list of actions, indicating to which states one can move
6   ;;  and at which cost. The lists of edges are placed as constants as
7   ;;  the second parameter of the navigate operators.
8   ;;
9   ;;  There are two problems defined: one minimizes the travel time,
10  ;;  the other minimizes the cost
11
12  (defparameter *travel-cheap*
13    (make-problem
14     :states                *cities*
15     :initial-state         *origin*
16     :f-h                   #'(lambda (state) (f-h-time state *estimate*))
17     :f-goal-test           #'(lambda (node) (f-goal-test node *destination* *mandatory*))
18     :f-search-state-equal  #'(lambda (node-1 node-2)
19                                (f-search-state-equal node-1 node-2 *mandatory*))
20     :operators             (list #'(lambda (node)
21                                      (navigate-canal-price
22                                        (node-state node)
```

```
23                                          *canals*))
24                                    #'(lambda (node)
25                                        (navigate-train-price
26                                          (node-state node)
27                                          *trains*
28                                          *forbidden*))))
29     )
30
31   (defparameter *travel-fast*
32     (make-problem
33       :states                  *cities*
34       :initial-state           *origin*
35       :f-h                     #'(lambda (state) (f-h-price state *estimate*))
36       :f-goal-test             #'(lambda (node) (f-goal-test node *destination* *mandatory*))
37       :f-search-state-equal    #'(lambda (node-1 node-2)
38                                    (f-search-state-equal node-1 node-2 *mandatory*))
39       :operators               (list #'(lambda (node)
40                                          (navigate-canal-time
41                                            (node-state node)
42                                            *canals*))
43                                    #'(lambda (node)
44                                        (navigate-train-time
45                                          (node-state node)
46                                          *trains*
47                                          *forbidden*))))
48     )
```

7. Ejercicio 6

En este ejercicio codificamos una de las funciones más importantes del algoritmo, `expand-node`, que se encarga de expandir todos los posibles nodos objetivo correspondientes a un estado alcanzable desde el nodo dado. Esta función recibirá también el problema a considerar, puesto que el coste de la exploración depende de ello.

Para ello lo que hacemos es primero generar la lista de acciones permitidas utilizando una función auxiliar. `create-actions`. Esta lista es recorrida posteriormente iterativamente, generando como resultado un nodo objetivo para cada una.

```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; The main function of this section is "expand-node", which receives
3  ;; a node structure (the node to be expanded) and a problem structure.
4  ;; The problem structure has a list of navigation operators, and we
5  ;; are interested in the states that can be reached using anuy one of
6  ;; them.
7  ;;
8  ;; So, in the expand-node function, we iterate (using mapcar) on all
9  ;; the operators of the problem and, for each one of them, we call
10 ;; expand-node-operator, to determine the states that can be reached
11 ;; using that operator.
12 ;;
13 ;; The operator gives us back a list of actions. We iterate again on
14 ;; this list of action and, for each one, we call expand-node-action
15 ;; that creates a node structure with the node that can be reached
16 ;; using that action.
17 ;;
18
19 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
20 ;;
21 ;;  Creates a list with all the nodes that can be reached from the
22 ;;  current one using all the operators in a given problem
23 ;;
24 ;;  Input:
25 ;;    node:    the node structure from which we start.
26 ;;    problem: the problem structure with the list of operators
```

```
27 ;;
28 ;;  Returns:
29 ;;    A list (node_1,...,node_n) of nodes that can be reached from the
30 ;;    given one
31 ;;
32 (defun expand-node (node problem)
33   (mapcar
34    #'(lambda (action)
35              (let ((gvalue (+ (node-g node)
36                               (action-cost action)))
37                    (hvalue (funcall (problem-f-h problem)
38                                     (action-final action))))
39                   (make-node
40                    :state (action-final action)
41                    :parent node
42                    :action action
43                    :depth (1+ (node-depth node))
44                    :g gvalue
45                    :h hvalue
46                    :f (+ gvalue hvalue))))
47    (create-actions node problem)))
48
49 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
50 ;; Aux-rec functions -- Exercise 6
51 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
52 (defun create-actions (node problem)
53   (append (funcall (first (problem-operators problem)) node)
54           (funcall (second (problem-operators problem)) node)))
```

Para probar su funcionamiento hemos definido la lista de nodos a partir de Marseille con el problema de `*travel-fast*`:

```
(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*))
```

Al imprimir la lista obtenemos la siguiente salida:

```
CL-USER 347 : 4 > (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 #) :DEPTH 13 :G 75.0 :H 0.0 :F 75.0))
(#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 #) :DEPTH 13 :G 75.0 :H 0.0 :F 75.0))
```

8. Ejercicio 7

En este ejercicio, en definitiva, tenemos que implementar el algoritmo `insert-sort`, para insertar en orden en una lista dada, respecto al criterio de comparación de una estrategia. En nuestro caso debemos insertar una lista de nodos, por lo que recursivamente llamamos a `insert-node`.

La lógica de nuestro `insert-node` es la siguiente:

1. Si la lista en la que insertamos está vacía, devolver una lista con el nodo a insertar.
2. Si la comparación entre el nodo a insertar y el primero de la lista, según el criterio dado, es `T`, entonces insertamos el nodo en el primer lugar de la lista.
3. En caso contrario, entramos recursivamente en la función tratando de insertar el nodo en el resto de la lista.

La función matriz, `insert-nodes-strategy`, únicamente controla el caso base de que ya no queden más nodos a insertar.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  Merges two lists of nodes, one of them ordered with respect to a
;;;  given strategy, keeping the result ordered with respect to the
;;;  same strategy.
;;;
;;;  This is the idea: suppose that the ordering is simply the
;;;  ordering of natural numbers. We have a "base" list that is
;;;  already ordered, for example:
;;;      lst1 --> '(3 6 8 10 13 15)
;;;
;;;  and a list that is not necessarily ordered:
;;;
;;;      nord --> '(11 5 9 16)
;;;
;;;  the call (insert-nodes nord lst1 #'<) would produce
;;;
;;;    (3 5 6 8 9 10 11 13 15 16)
;;;
;;;  The functionality is divided in two functions. The first,
;;;  insert-node, inserts a node in a list keeping it ordered.
;;;  The last function, insert-node-strategy is a simple interface that
;;;  receives a strategy, extracts from it the comparison function,
;;;  and calls insert-node recursively.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Aux-rec function -- Exercise 7
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun insert-node (node lst-nodes node-compare-p)
  (cond
    ((null lst-nodes) (list node))
    ((funcall node-compare-p node (first lst-nodes))
      (cons node lst-nodes))
    (t (cons (first lst-nodes)
             (insert-node node
                 (rest lst-nodes)
                  node-compare-p)))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Inserts a list of nodes in an ordered list keeping the result list
;; ordered with respect to the given comparison function
;;
;; Input:
;;    nodes: the (possibly unordered) node list to be inserted in the
;;           other list
;;    lst-nodes: the (ordered) list of nodes in which the given nodes
;;               are to be inserted
;;    node-compare-p: a function node x node --> 2 that returns T if the
;;                    first node comes first than the second.
;;
;; Returns:
;;    An ordered list of nodes which includes the nodes of lst-nodes and
;;    those of the list "nodes@. The list is ordered with respect to the
;;    criterion node-compare-p.
```

```
53 ;;
54 (defun insert-nodes-strategy (nodes lst-nodes strategy)
55   (let ((node-compare-p (strategy-node-compare-p strategy)))
56        (cond ((null nodes) lst-nodes)
57         ((null lst-nodes)
58           (insert-nodes-strategy (rest nodes)
59                       (list (first nodes))
60                       strategy))
61         ;; insertar N nodos
62         (t  (insert-nodes-strategy (rest nodes)
63                                    (insert-node (first nodes)
64                                                 lst-nodes
65                                                 node-compare-p)
66             strategy)))))
```

Finalmente, definimos la función de comparación de nodos para los casos de prueba, la estrategia de coste uniforme, dos nodos más y, por último, los insertamos según la estrategia:

```
(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*))
```

La salida que obtenemos en relación a este ejercicio es la siguiente:

```
CL-USER 353 : 4 > (mapcar #'(lambda (x) (node-state x)) sol-ex7)
(PARIS NANCY TOULOUSE)

CL-USER 354 : 4 > (mapcar #'(lambda (x) (node-g x)) sol-ex7)
(0 50 75.0)
```

9. Ejercicio 8

En este ejercicio implementamos la búsqueda A–estrella, que básicamente consiste en comparar los valores `node-f` de ambos nodos.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; A strategy is, basically, a comparison function between nodes to tell
;; us which nodes should be analyzed first. In the A* strategy, the first
;; node to be analyzed is the one with the smallest value of g+h
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun node-compare-a-star-p (node1 node2)
  (when (and (not (null node1))
             (not (null node2)))
    (<= (node-f node1)
        (node-f node2))))


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

10. Ejercicio 9

En este ejercicio implementamos `graph-search`. Esta función busca un camino que resuelve el problema dado según un criterio. En este caso tenemos que implementar una función auxiliar, que será la encargada de hacer todo el trabajo. La función `graph-search` es simplemente una interfaz para inicializar las listas de nodos abiertos y cerrados y llama a la función auxiliar.

La función auxiliar `graph-search-aux`, por el contrario, es una función recursiva que expande por orden todos los nodos a los que es posible llegar desde cada nodo actual. Por cada exploración se comprueba si hemos llegado al objetivo, en cuyo caso devolvemos el nodo. Además es necesario controlar la inserción de nodos en la lista de cerrados, ya que es relevante escoger siempre los nodos con menor coste, aunque sean iguales. Para ello utilizamos la función `expand-it`. Este procedimiento continúa hasta acabar con los nodos a explorar y devolver el camino que obtenga mejor coste según la estrategia proporcionada.

Por último, escribimos la función `a-star-search`, que es únicamente un *wrapper* de `graph-search` con la estrategia `A-star`.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;     Searches a path that solves a given problem using a given search
;;;     strategy. Here too we have two functions: one is a simple
;;;     interface that extracts the relevant information from the
;;;     problem and strategy structure, builds an initial open-nodes
;;;     list (which contains only the starting node defined by the
;;;     state), and initial closed node list (the empty list), and calls
;;;     the auxiliary function.
;;;
;;;     The auxiliary is a recursive function that extracts nodes from
;;;     the open list, expands them, inserts the neighbors in the
;;;     open-list, and the expanded node in the closed list. There is a
;;;     caveat: with this version of the algorithm, a node can be
;;;     inserted in the open list more than once. In this case, if we
;;;     extract a node in the open list and the following two condition old:
;;;
;;;      the node we extract is already in the closed list (it has
;;;      already been expanded)
;;;        and
;;;      the path estimation that we have is better than the one we
;;;      obtain from the node in the open list
;;;
;;;      then we ignore the node.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
28 ;;
29 ;;  Auxiliary search function (the one that actually does all the work
30 ;;
31 ;;  Input:
32 ;;    problem: the problem structure from which we get the general
33 ;;             information (goal testing function, action operatos, etc.
34 ;;    open-nodes: the list of open nodes, nodes that are waiting to be
35 ;;                visited
36 ;;    closed-nodes: the list of closed nodes: nodes that have already
37 ;;                  been visited
38 ;;    strategy: the strategy that decide which node is the next extracted
39 ;;              from the open-nodes list
40 ;;
41 ;;    Returns:
42 ;;     NIL: no path to the destination nodes
43 ;;     If these is a path, returns the node containing the final state.
44 ;;
45 ;;     Note that what is returned is quite a complex structure: the
46 ;;     node contains in "parent" the node that comes before in the
47 ;;     path, that contains another one in "parents" and so on until
48 ;;     the initial one. So, what we have here is a rather complex
49 ;;     nested structure that contains not only the final node but the
50 ;;     whole path from the starting node to the final.
51 ;;
52 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
53 ;; Aux-- Exercise 9
54 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
55 (defun expand-it (node closed-lst)
56     (if (null closed-lst)
57       T
58       (let ((fst (first closed-lst)))
59         (if  (and (f-search-state-equal node fst *mandatory*)
60                   (>= (node-g node) (node-g fst)))
61              NIL
62              (expand-it node (rest closed-lst))))))
63
64 (defun graph-search-aux (problem strategy open-nodes closed-nodes)
65   (let ((first-open (first open-nodes)))
66     (cond ((null open-nodes))
67           ;; si el primer nodo de la lista de abiertos es el objetivo devolver y terminar
68           ((f-goal-test first-open
69                         *destination*
70                         *mandatory*)
71            first-open)
72           ;; si el primer nodo de la lista de abiertos no esta en la lista de cerrados
73           ((expand-it first-open closed-nodes)
74            (graph-search-aux problem
75                              strategy
76                              (insert-nodes-strategy (expand-node first-open problem)
77                                                     (rest open-nodes)
78                                                     strategy)
79                              (cons first-open closed-nodes)))
80           (t
81            (graph-search-aux problem
82                              strategy
83                              (rest open-nodes)
84                              closed-nodes)))))
85
86 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
87 ;;
88 ;;  Interface function for the graph search.
89 ;;
90 ;;  Input:
91 ;;    problem: the problem structure from which we get the general
92 ;;             information (goal testing function, action operatos,
93 ;;             starting node, heuristic, etc.
94 ;;    strategy: the strategy that decide which node is the next extracted
95 ;;              from the open-nodes list
```

```
96  ;;
97  ;;    Returns:
98  ;;     NIL: no path to the destination nodes
99  ;;     If these is a path, returns the node containing the final state.
100 ;;
101 ;;    See the graph-search-aux for the complete structure of the
102 ;;    returned node.
103 ;;    This function simply prepares the data for the auxiliary
104 ;;    function: creates an open list with a single node (the source)
105 ;;    and an empty closed list.
106 ;;
107 (defun graph-search (problem strategy)
108   (let  ;; llamo open-nodes a la lista con el estado inicial
109                 ((open-nodes (list
110                     (make-node
111                      :state (problem-initial-state problem)))))
112                 ;; inicializo closed-nodes a nil porque no hay nodos explorados inicialmente
113                 (graph-search-aux problem strategy open-nodes nil)))
114
115 ;
116 ;  A* search is simply a function that solves a problem using the A* strategy
117 ;
118 (defun a-star-search (problem)
119   (graph-search problem *A-star*))
```

Finalmente, probamos que la salida sea la correcta:

```
CL-USER 355 : 4 > (graph-search *travel-cheap* *A-star*)
#S(NODE :STATE CALAIS :PARENT #S(NODE :STATE REIMS :PARENT
#S(NODE :STATE PARIS :PARENT #S(NODE :STATE NEVERS :PARENT
#S(NODE :STATE LIMOGES :PARENT #S(NODE :STATE TOULOUSE :PARENT
#S(NODE :STATE MARSEILLE :PARENT NIL
    :ACTION NIL :DEPTH 0 :G 0 :H 0 :F 0) :ACTION
    #S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN
        MARSEILLE :FINAL TOULOUSE :COST 120.0)
        :DEPTH 1 :G 120.0 :H 130.0 :F 250.0) :ACTION
    #S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN TOULOUSE :FINAL LIMOGES
        :COST 35.0) :DEPTH 2 :G 155.0 :H 100.0 :F 255.0) :ACTION
    #S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN LIMOGES :FINAL NEVERS
        :COST 60.0) :DEPTH 3 :G 215.0 :H 70.0 :F 285.0) :ACTION
    #S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN NEVERS :FINAL PARIS
        :COST 10.0) :DEPTH 4 :G 225.0 :H 30.0 :F 255.0) :ACTION
    #S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN PARIS :FINAL REIMS
        :COST 10.0) :DEPTH 5 :G 235.0 :H 25.0 :F 260.0) :ACTION
    #S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN REIMS :FINAL CALAIS :COST 15.0)

CL-USER 356 : 4 > (a-star-search *travel-fast*)
#S(NODE :STATE CALAIS :PARENT
#S(NODE :STATE PARIS :PARENT
#S(NODE :STATE ORLEANS :PARENT
#S(NODE :STATE LIMOGES :PARENT
#S(NODE :STATE TOULOUSE :PARENT
#S(NODE :STATE MARSEILLE :PARENT NIL
    :ACTION NIL :DEPTH 0 :G 0 :H 0 :F 0) :ACTION
    #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE
        :COST 65.0) :DEPTH 1 :G 65.0 :H 0.0 :F 65.0) :ACTION
```

```
    #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN TOULOUSE :FINAL LIMOGES
        :COST 25.0) :DEPTH 2 :G 90.0 :H 0.0 :F 90.0) :ACTION
    #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN LIMOGES :FINAL ORLEANS
        :COST 55.0) :DEPTH 3 :G 145.0 :H 0.0 :F 145.0) :ACTION
    #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN ORLEANS :FINAL PARIS
        :COST 23.0) :DEPTH 4 :G 168.0 :H 0.0 :F 168.0) :ACTION
    #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN PARIS :FINAL CALAIS
        :COST 34.0) :DEPTH 5 :G 202.0 :H 0.0 :F 202.0)
```

Nota: *(graph-search *travel-cheap* *A-star*)* y *(a-star-search *travel-cheap*)* son equivalentes.

```
CL-USER 357 : 5 > (a-star-search *travel-cheap*)
#S(NODE :STATE CALAIS :PARENT #S(NODE :STATE REIMS :PARENT
#S(NODE :STATE PARIS :PARENT #S(NODE :STATE NEVERS :PARENT
#S(NODE :STATE LIMOGES :PARENT #S(NODE :STATE TOULOUSE :PARENT
#S(NODE :STATE MARSEILLE :PARENT NIL
    :ACTION NIL :DEPTH 0 :G 0 :H 0 :F 0) :ACTION
    #S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN
        MARSEILLE :FINAL TOULOUSE :COST 120.0)
        :DEPTH 1 :G 120.0 :H 130.0 :F 250.0) :ACTION
    #S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN TOULOUSE :FINAL LIMOGES
        :COST 35.0) :DEPTH 2 :G 155.0 :H 100.0 :F 255.0) :ACTION
    #S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN LIMOGES :FINAL NEVERS
        :COST 60.0) :DEPTH 3 :G 215.0 :H 70.0 :F 285.0) :ACTION
    #S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN NEVERS :FINAL PARIS
        :COST 10.0) :DEPTH 4 :G 225.0 :H 30.0 :F 255.0) :ACTION
    #S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN PARIS :FINAL REIMS
        :COST 10.0) :DEPTH 5 :G 235.0 :H 25.0 :F 260.0) :ACTION
    #S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN REIMS :FINAL CALAIS :COST 15.0)
```

11. Ejercicio 10

Llegados a este punto, lo único que nos queda es, por un lado, extraer el camino que nos ha llevado hasta el objetivo, y por el otro, recuperar la secuencia de acciones hasta el mismo. De esto precisamente se encargan las dos funciones protagonistas de este ejercicio, `solution-path` y `action-sequence`. Nótese que en ambos casos hemos usado `reverse`, puesto que hemos ido concatenando cada nodo padre o acción al final de la lista.

Hemos escrito una función auxiliar, `visited-node`, que construye una lista de nodos visitados que posteriormente usamos para extraer el nodo padre.

```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;;;
3  ;;;    BEGIN Exercise 10: Solution path
4  ;;;
5  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
6  ;*** solution-path ***
7  (defun visited-node (node)
8         (unless (null node)
9                 ;; si el nodo no tiene padre
10                (if (null (node-parent node))
11                        ;; formo la lista con el nodo actual
12                        (list (node-state node))
```

```
13                          ;; meto en la primera posicion el nodo actual con sus anteriores
14                          (cons (node-state node)
15                                (visited-node (node-parent node))))))
16
17
18  (defun solution-path (node)
19    (reverse (visited-node node)))
20
21  ;*** action-sequence ***
22  ; Visualize sequence of actions
23  (defun action-node (node)
24    (unless (or (null node) (null (node-action node)))
25      (cons (node-action node)
26            (action-node (node-parent node)))))
27  (defun action-sequence (node)
28    (reverse (action-node node)))
```

Probamos el código obteniendo la siguiente salida:

```
CL-USER 362 : 6 > (solution-path nil)
NIL

CL-USER 363 : 6 > (solution-path (a-star-search *travel-fast*))
(MARSEILLE TOULOUSE LIMOGES ORLEANS PARIS CALAIS)

CL-USER 364 : 6 > (solution-path (a-star-search *travel-cheap*))
(MARSEILLE TOULOUSE LIMOGES NEVERS PARIS REIMS CALAIS)

CL-USER 365 : 6 > (action-sequence (a-star-search *travel-fast*))
(#S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE
                                                        :COST 65.0)
#S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN TOULOUSE :FINAL LIMOGES :COST 25.0)
#S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN LIMOGES :FINAL ORLEANS :COST 55.0)
#S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN ORLEANS :FINAL PARIS :COST 23.0)
#S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN PARIS :FINAL CALAIS :COST 34.0))

CL-USER 366 : 6 > (action-sequence (a-star-search *travel-cheap*))
(#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN MARSEILLE :FINAL TOULOUSE
                                                         :COST 120.0)
#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN TOULOUSE :FINAL LIMOGES :COST 35.0)
#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN LIMOGES :FINAL NEVERS :COST 60.0)
#S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN NEVERS :FINAL PARIS :COST 10.0)
#S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN PARIS :FINAL REIMS :COST 10.0)
#S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN REIMS :FINAL CALAIS :COST 15.0))
```

12. Ejercicio 11

En esta sección implementamos dos estrategias de búsqueda adicionales, como son la búsqueda en profundidad y en anchura.

La primera de ellas debe explorar un camino hasta el final antes de explorar cualquier otro nodo. Por ello, devolvemos `T` si la profundidad del `nodo1` es menor que la del `nodo2`. En el caso de la búsqueda en anchura hacemos exactamente lo contrario.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;     BEGIN Exercise 11: Depth & Bredth
;;;
(defun depth-first-node-compare-p (node1 node2)
  (when (and (not (null node1))
             (not (null node2)))
    T))

(defparameter *depth-first*
  (make-strategy
   :name 'depth-first
   :node-compare-p #'depth-first-node-compare-p))

(defun breadth-first-node-compare-p (node1 node2)
  NIL)

(defparameter *breadth-first*
  (make-strategy
   :name 'breadth-first
   :node-compare-p #'breadth-first-node-compare-p))
```

En este ejercicio hemos obtenido la siguiente salida:

```
CL-USER 179 > (solution-path (graph-search *travel-fast* *depth-first*))
(MARSEILLE TOULOUSE LYON ROENNE NEVERS LIMOGES ORLEANS NANTES BREST
ST-MALO PARIS ST-MALO BREST NANTES TOULOUSE LYON NANCY REIMS CALAIS)

CL-USER 180 > (solution-path (graph-search *travel-fast* *breadth-first*))
(MARSEILLE TOULOUSE NANTES ST-MALO PARIS CALAIS)
```

13. Ejercicio 12

1. Definir una nueva lista llamada *estimate-new* que tenga la misma heurística para los tiempos de recorrido y la nueva heurística para los costes
2. Definir una nueva estructura de problema *travel-cost-new* que defina el problema de coste mínimo con la nueva heurística

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;     BEGIN Exercise 12: Cost heuristic
;;;
(defparameter *estimate-new*
  '((Calais (0.0 0.0)) (Reims (25.0 10.0)) (Paris (30.0 20.0))
    (Nancy (50.0 30.0)) (Orleans (55.0 45.0)) (St-Malo (65.0 55.0))
    (Nantes (75.0 65.0)) (Brest (90.0 70.0)) (Nevers (70.0 50.0))
    (Limoges (100.0 80.0)) (Roenne (85.0 75.0)) (Lyon (105.0 85.0))
    (Toulouse (130.0 100.0)) (Avignon (135.0 105.0)) (Marseille (145.0 130.0))))

(defparameter *travel-cost-new*
  (make-problem
   :states                *cities*
   :initial-state         *origin*
   :f-h                   #'(lambda (state) (f-h-time state *estimate-new*))
   :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-state node)
```

```
23                                        *canals*))
24                                    #'(lambda (node)
25                                        (navigate-train-price
26                                          (node-state node)
27                                          *trains*
28                                          *forbidden*)))))
```

3 Usando las funciones del LISP, medir el tiempo de ejecución del problema de coste mínimo con la heurística original y con la nueva.

Hemos probado esta heurística de coste en los siguientes casos, obteniendo las siguientes salidas por terminal:

```
CL-USER 91 > (defparameter lst-nodes-ex6
  (expand-node node-marseille-ex6 *travel-cost-new*))
LST-NODES-EX6

CL-USER 92 >  (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-PRICE :ORIGIN MARSEILLE
    :FINAL TOULOUSE :COST 120.0) :DEPTH 13 :G 130.0 :H 130.0 :F 260.0))
(#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-PRICE :ORIGIN MARSEILLE
    :FINAL TOULOUSE :COST 120.0) :DEPTH 13 :G 130.0 :H 130.0 :F 260.0))

CL-USER 100 : 3 >  (solution-path (a-star-search *travel-cost-new*))
(MARSEILLE TOULOUSE LIMOGES NEVERS PARIS REIMS CALAIS)

CL-USER 96 > (action-sequence (a-star-search *travel-cost-new*))
(#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN MARSEILLE :FINAL TOULOUSE
                                                         :COST 120.0)
#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN TOULOUSE :FINAL LIMOGES :COST 35.0)
#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN LIMOGES :FINAL NEVERS :COST 60.0)
#S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN NEVERS :FINAL PARIS :COST 10.0)
#S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN PARIS :FINAL REIMS :COST 10.0)
#S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN REIMS :FINAL CALAIS :COST 15.0))
```

¿Por qué se ha realizado este diseño para resolver el problema de búsqueda? La razón por la cual se ha implementado este diseño es porque se trata de una solución genérica, ya que podemos definir el tipo de estrategia a utilizar para cada búsqueda, lo cual puede resultar muy beneficioso.

En concreto, ¿Qué ventajas aporta? ¿Por qué se han utilizado funciones lambda para especificar el test objetivo, la heurística y los operadores del problema? Por un lado se trata de una

solución flexible, ya que no existe la necesidad de implementar diferentes funciones para resolver cada problema, nada más que una estructura `problem`. En concreto, usamos *lambdas* porque nos permiten abstraer los parámetros del dominio del problema.

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? Se trata de un uso eficiente en memoria ya que se le proporciona la referencia al nodo padre evitando generar un nodo nuevo cada vez que se crea un hijo.

¿Cuál es la complejidad espacial del algoritmo implementado? La complejidad espacial es exponencial para el número de conexiones de tren, mientras que la complejidad de exploración de los canales escala de manera lineal, puesto que son unidireccionales.

¿Cuál es la complejidad temporal del algoritmo? Es la misma, ya que se mantienen todos los nodos en memoria.

Indicad qué partes del código se modificarían para limitar el número de veces que se puede utilizar la acción "navegar por trenes" (bidireccionales). Para ello se tendría que modificar la función `f-goal-test`, añadiendo un contador para registrar el número de veces que se ha utilizado el tren en la lista de acciones. Además, se debería comprobar que ese número sea menor que el límite, el cual se debe especificar *a priori* en un nuevo campo en la estructura `problem`. En el caso de que se supere dicho límite, el camino no contaría como objetivo.