MEMORIA PRÁCTICA 2

Autores: Victoria Pelayo e Ignacio Rabuñal

Grupo: 2301 pareja 8

Ejercicio1:

En este ejercicio nos piden implementar dos funciones para evaluar el valor de la heurística, según el tiempo o el precio, de un estado.

Pseudocódigo de las funciones:

F-H-TIME (state sensors)

```
input:
      state: nombre de la ciudad (estado) actual
      sensors: lista de los estados y sus correspondientes heurísticas
                  los elementos de la lista son (state (h-time h-price))

Output:
      Res: el valor de la heurística del tiempo

Proceso:
      i = 0
      para i < longitud de sensors:
            si state = valor-estado(sensors[i]):
                  devolver valor-heurisitica-time (sensors[i])
```

F-H-PRICE (state, sensors)

```
input:
      state: nombre de la ciudad (estado) actual
      sensors: lista de los estados y sus correspondientes heurísticas
                  los elementos de la lista son (state (h-time h-price))

Output:
      Res: el valor de la heurística del precio

Proceso:
      i = 0
```

```
para i < longitud de sensors:
        si state = valor-estado(sensors[i]):
                devolver valor-heurisitica-price (sensors[i])
```

Capturas del Código implementado:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Devuelve la heuristica del tiempo
;;
;; Input:
;; -state: estado (=nombre de la ciudad) actual
;; -sensors: lista cuyos elementos son de la siguiente forma :
;;                       (estado (h-time h-price))
;;
;; Returns:
;;      Valor de la heuristica del tiempo
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun f-h-time (state sensors)
    (cond
    ((or (null state)(null sensors)) NIL)
    ((equal state (first (first sensors))) (first (second (first sensors))))
    ( T (f-h-time state (cdr sensors)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Devuelve la heuristica del precio
;;
;; Input:
;; -state: estado (=nombre de la ciudad) actual
;; -sensors: lista cuyos elementos son de la siguiente forma :
;;                       (estado (h-time h-price))
;;
;; Returns:
;;      Valor de la heuristica del precio
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun f-h-price (state sensors)
  (cond
    ((or (null state)(null sensors)) NIL)
    ((equal state (first (first sensors))) (second (second (first sensors))))
    ( T (f-h-price state (cdr sensors)))))
;;
```

Ejemplos de ejecución ejercicio 1:

```
CG-USER(21): (f-h-time 'Nantes *estimate*)

75.0

CG-USER(22): (f-h-time 'Marseille *estimate*)

145.0

CG-USER(35): (f-h-price 'Lyon *estimate*)

0.0

CG-USER(36): (f-h-price 'Madrid *estimate*)

NIL

CG-USER(37): (f-h-time 'Madrid *estimate*)
```

```
NIL
```

Ejercicio2:

En este ejercicio tenemos que implementar 4 funciones(navigate-canal-time, navigate-canal-price, navigate-train-time, navigate-train-price) que devolverán una lista de acciones que pueden realizar dado un estado actual. Por ejemplo, navigate-canal-time nos devolverá la lista de acciones posibles, partiendo de un estado, utilizando canales y tomará como heurística el tiempo.

Nos piden crear una función auxiliar navigate más genérica que recibe una función ("cfun", una de las mencionadas anteriormente) y se encargará de devolver la lista de acciones posibles en función de la "cfun" pasada. Es importante la existencia de esta función, pues, evita repetir código.

Para este ejercicio también hemos creado otras tres funciones auxiliares: navigate-aux, que es la función recursiva de navigate; ciudad-permitida, que devuleve si una ciudad está permitida en el camino; get-time, que devuelve el coste del tiempo de un elemento determinado; get-price, que devuelve el coste del precio de un elemento determinado.

Pseudocódigos de las funciones:

Ciudad-permitida(state, forbidden)

```
Input: state (estado actual), forbidden (lista de estados prohibidos)

Output: T si está permitida y F si no es así

Proceso:

       Si estado = null:

              Devolver False

       Si prohibidas = null:

              Devolver False

       Si estado = forbidden[0]:

              Devolver True

       Si no:

              Devolver ciudad-permitida (state, resto de forbidden)


       *resto de forbidden = lista de forbidden a partir del elemento forbidden[1]
```

Get-time (lst)

```
Input: lst (lista de elementos de la forma (origin final (time price)) )

Output: Res(tiempo del primer elemento de la tabla)

Proceso:

       Si lst = null:

              Devolver null
```

Si no:

Devolver lst[0][2][0] (primer elemento del tercer elemento del primer elemento de la lista)

Get-price (lst)

Input: lst (lista de elementos de la forma (origin final (time price)))

Output: Res(precio del primer elemento de la tabla)

Proceso:

Si lst = null:

Devolver null

Si no:

Devolver lst[1][2][0] (segundo elemento del tercer elemento del primer elemento de la lista)

Navigate(state lst-edges cfun name(opcional) forbidden)

Input: state(estado actual), lst-edges(lista de los nodos del grafo), cfun(nombre de la función para saber que coste usar), nombre (es opcional, nombre que le pondremos a la acción), forbidden (lista de los estados no permitidos)

Output: Res lista de acciones originadas a partir del estado dado

Proceso:

Res = navigate-aux(state, lst-edges, name, forbidden, ())

Navigate-aux(state lst-edges cfun name(opcional) forbidden lst)

Input: state(estado actual), lst-edges(lista de nodos del grafo), cfun(función para saber que coste obtener), name(combre de la acción), forbidden(lista de estados no permitidos), lst(lista de acciones)

Output: lst(lista de acciones realizadas a partir del estado)

Proceso:

Si (estado = null OR lst-edges = null):

Devolver null

Si (estado = lst-edges[0][0] AND True = ciudad-permitida(lst-edges[0][1], forbidden)):

Acción = nueva acción{

Name = name

Origin = state

Final = lst-edges[0][1]

Cost = cfun(lst-edges)

}

Añadir-a-lst(accion)

Añadir-a-lst(navigate-aux(state lst-edges cfun name forbidden lst))

Devolver lst

Si no:

Añadir-a-lst(navigate-aux(state lst-edges cfun name forbidden lst))

Devolver lst

En esta función de manera recursiva en la lista de grafos comprobamos si el grafo corresponde su estado actual al nuestro y si a donde lleva es una ciudad permitida, si esto es así añadimos una acción que creamos con estas características a nuestra lista de acciones y el resultado de aplicar esta función al resto de nodos de la lista. Si el primero no coincide devolvemos la lista tras añadirle el resultado de aplicar esta función al resto de la lista de nodos.

Navigate-canal-time(state canals)

Input: state(estado actual), canals(lista de canales)

Output: Res(lista de acciones al usar canales a partir del estado actual indicando el coste del tiempo)

Proceso:

Deolver navigate (state, canals, función get-time, “navigate-canal-time”, null)

Navigate-canal-price(state canals)

Input: state(estado actual), canals(lista de canales)

Output: Res(lista de acciones al usar canales a partir del estado actual indicando el coste del precio)

Proceso:

Deolver navigate (state, canals, función get-price, “navigate-canal-price”, null)

Navigate-trains-price(state canals)

Input: state(estado actual), trains(lista de caminos por trenes)

Output: Res(lista de acciones al usar canales a partir del estado actual indicando el coste del precio)

Proceso:

Deolver navigate (state, canals, función get-price, “navigate-train-price”, *forbidden*)

forbidden = lista de ciudades prohibidas por tren

Navigate-trains-time(state canals)

Input: state(estado actual), trains(lista de caminos por trenes)

Output: Res(lista de acciones al usar canales a partir del estado actual indicando el coste del tiempo)

Proceso:

Devolver navigate (state, canals, función get-price, “navigate-train-price”, *forbidden*)

Código del ejercicio 2:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; General navigation function
;;
;;  Returns the actions that can be carried out from the current state
;;
;;  Input:
;;    state:       the state from which we want to perform the action
;;    lst-edges:   list of edges of the graph, each element is of the
;;                 form: (source destination (cost1 cost2))
;;    c-fun:       function that extracts the correct cost (time or price)
;;                 from the pair that appears in the edge
;;    name:        name to be given to the actions that are created (see the
;;                 action structure)
;;    forbidden-cities:
;;                 list of the cities where we can't arrive by train
;;
;;  Returns
;;    A list of action structures with the origin in the current state and
;;    the destination in the states to which the current one is connected
;;
(defun navigate (state lst-edges cfun  name &optional forbidden )
    (navigate-aux state lst-edges cfun name forbidden nil))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Funcion auxiliar de navigate
;;
;; De manera recursiva crea una lista de estructuras accion
;;
;; Input:
;; -State: Ciudad inicio
;; -lst-edges: lista de la forma (inicio fin (coste-tiempo coste-precio))
;; -cfun: funcion para "sacar" el tiempo/precio de un elemento de lst-edges
;; -name: nombre de la accion que se va a realizar
;; -forbidden: ciudades a las que NO se puede ir
;; -lst: lista de las acciones disponibles en ese state
;;
;; Returns:
;;      Una lista (lst) con todas las acciones disponibles en state
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun navigate-aux (state lst-edges cfun name &optional forbidden lst)
    (cond
    ((or (null state) (null lst-edges)) nil)
    ((and (equal state (first (first lst-edges))) (ciudad-permitida (second (first lst-edges)) forbidden))
        (append
                (list (make-action :name name :origin state :final (second (first lst-edges)) :cost (apply cfun (list lst-edges))))
                (navigate-aux state (cdr lst-edges) cfun name forbidden lst)
                lst))
    (T (append (navigate-aux state (cdr lst-edges) cfun name forbidden lst) lst))))
```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Funcion auxiliar que determina si esta permitido ir a una ciudad-permitida
;;
;; Input:
;; -State: ciudad que estamos mirarndo si esta permitida
;; -forbidden: lista de ciudades prohibidas
;;
;; Returns:
;;      T si la ciudad esta permitida, Nil si no es asi
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun ciudad-permitida(state forbidden)
    (cond
    ((null state) nil)
    ((null forbidden) T)
    ((equal state (first forbidden)) nil)
    ( T (ciudad-permitida state (cdr forbidden)))))
```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Funcion auxiliar que devuelve el coste-tiempo de un elemento de
;; la siguiente forma (origin final (time price))
;;
;; Input:
;; -lst: lista donde cada elemento es de la forma anterior
;;
;; Returns:
;;      coste-tiempo del primer elemento de la lista
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun get-time (lst)
    (if (null lst) nil
        (first (third (first lst)))))
```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Funcion auxiliar que devuelve el coste-precio de un elemento de
;; la siguiente forma (origin final (time price))
;;
;; Input:
;; -lst: lista donde cada elemento es de la forma anterior
;;
;; Returns:
;;      coste-precio del primer elemento de la lista
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun get-price (lst)
    (if (null lst) nil
        (second (third (first lst)))))
```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Especializacion de navigate
;;
;; Input:
;; -state: estado inicial
;; -canals: canales
;;
;; Output:
;;      Lista con las acciones disponibles a partir de state utilizando canals
;;          y tomando como coste el tiempo
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun navigate-canal-time (state canals)
    (navigate state canals #'get-time 'navigate-canal-time nil))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Especializacion de navigate
;;
;; Input:
;; -state: estado inicial
;; -canals: canales
;;
;; Output:
;;      Lista con las acciones disponibles a partir de state utilizando canals
;;          y tomando como coste el precio
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun navigate-canal-price (state canals)
    (navigate state canals #'get-price 'navigate-canal-price nil))
```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Especializacion de navigate
;;
;; Input:
;; -state: estado inicial
;; -trains: vias de trenes
;; -forbidden: lista de estados prohibidos
;;
;; Output:
;;      Lista con las acciones disponibles a partir de state utilizando trains
;;          y tomando como coste el tiempo
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun navigate-train-time (state trains forbidden)
  (navigate state trains #'get-time 'navigate-train-time forbidden))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Especializacion de navigate
;;
;; Input:
;; -state: estado inicial
;; -trains: vias de trenes
;; -forbidden: lista de estados prohibidos
;;
;; Output:
;;      Lista con las acciones disponibles a partir de state utilizando trains
;;          y tomando como coste el precio
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun navigate-train-price (state trains forbidden)
  (navigate state trains #'get-price 'navigate-train-price forbidden))

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

Ejemplos de ejecución ejercicio 2:

```
CG-USER(139): (navigate-canal-time 'Avignon *canals*)

(#S(ACTION :NAME NAVIGATE-CANAL-TIME :ORIGIN AVIGNON :FINAL MARSEILLE :COST 35.0))

CG-USER(140): (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))

CG-USER(141): (navigate-train-price 'Avignon *trains* '(Marseille))

(#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN AVIGNON :FINAL LYON :COST 40.0))

CG-USER(142): (navigate-canal-time 'Orleans *canals*)

NIL

CG-USER(143): (navigate-canal-time 'Lyon *canals*)

(#S(ACTION :NAME NAVIGATE-CANAL-TIME :ORIGIN LYON :FINAL NANCY :COST 150.0) #S(ACTION
:NAME NAVIGATE-CANAL-TIME :ORIGIN LYON :FINAL ROENNE :COST 40.0)

 #S(ACTION :NAME NAVIGATE-CANAL-TIME :ORIGIN LYON :FINAL AVIGNON :COST 50.0))

CG-USER(144): (navigate-train-time 'Lyon *trains* *forbidden*)

(#S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN LYON :FINAL TOULOUSE :COST 60.0)
#S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN LYON :FINAL ROENNE :COST 18.0))

CG-USER(145): (navigate-train-price 'Lyon *trains* *forbidden*)

(#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN LYON :FINAL TOULOUSE :COST 95.0)
#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN LYON :FINAL ROENNE :COST 25.0))
```

Ejercicio 3

En este ejercicio nos piden implementar una función que compruebe si un nodo ha alcanzado el objetivo. Para ello debe corresponder a una de las ciudades objetivo y haber pasado por todas las ciudades obligatorias.

Para este ejercicio hemos implementado, a parte de la función pedida en el enunciado, la función presente que determina si un elemento está presente en una lista; y camino válido que devuelve true si un camino es válido, ha pasado por todas las ciudades obligatorias.

Pseudocódigo de la las funciones:

Presente (x lst)

```
Input: x(elemento que buscamos), lst (lista donde buscamos el elemento)
Output: T si el elemento está presente, Nil si no es así
Proceso:
       Si x = null OR lst = null:
              Devolver nil
       Si no:
              Equal(x, lst[0]) OR presente (x, rest(lst))
```

Get-camino (node lst)

```
Input: node(nodo cuyo camino queremos saber), lst(lista de estados anteriores,
camino)
```

```
Output: lst (lista de estados del camino)
Proceso:
	When node node >< node-nevers
		Parent = node-get-parent(node)
		Añadir-a-lst (get-state(node))
		Añadir-a-lst (get-camino (parent, lst))
```

Camino-valido (camino mandatory)

```
Input: camino(lista de estados), mandatory(lista de estados, las ciudades
obligatorias)
Output: T si ha camino pasa por todas, Nil si no es así
Proceso:
	Si mandatory = null:
		Devolver true
	Si no:
		Devolver presente(mandatory[0], camino) AND camino-valido(camino,
rest(mandatory))
```

f-goal-test (node destination mandatory)

```
Input: node(nodo correspondiente al estado actual), destination (ciudades destino),
mandatory(ciudades obligatorias)
Output: T si el estado es una ciudad destino y ha pasado por todas las obligatorias
Proceso:
	Si node no es un nodo OR presente(node-state, destination) = false:
		Devolver nil
	Si no:
		Devolver camino-valido(get-camino(node, nil), mandatory)
```

Código implementado:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; 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)
  (if (or (not (node-p node)) (not (presente (node-state node) destination)))
    nil
    ;;si llega aqui es porque es un nodo destino
    (camino-valido (get-camino node nil) mandatory)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Determina si un elemento esta presente en una lista
;;
;; Input:
;; -x: elemento que "buscamos"
;; -lst: lista de elementos
;;
;; Returns:
;;      T si x esta presente en lst, nil si no es asi
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun presente (x lst)
    (if (or (null x) (null lst)) nil
     (or (equal (first lst) x) (presente x (cdr lst)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Obtiene el camino de un nodo
;;
;; Input:
;; -node: nodo cuyo camino buscamos
;; -lst: lista de ciudades del camino hasta ese nodo
;;
;; Returns:
;;      lst, lista del camino de ciudades anteriores hasta ese nodo
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun get-camino (node lst)
    (when (and (node-p node) (not (equal node-nevers node)))
        (append (list (node-state node)) (get-camino (node-parent node) lst) lst)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Mira si las ciudades "obligatorias estan en el camino"
;;
;; Input:
;; -camino: lista de ciudades que forman el camino
;; -mandatory: lista de ciudades "obligatorias"
;;
;; Returns:
;;      T si si pasa por todas las ciudades obligatorias, nil
;;      si no es asi
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun camino-valido(camino mandatory)
  (if (null mandatory)
    T
  (and (presente (first mandatory) camino)
       (camino-valido camino (cdr mandatory)))))
```

```
CG-USER(104): (f-goal-test node-calais '(Calais Marseille) '(Paris Nancy))

T

CG-USER(105): (f-goal-test node-calais '(Calais Marseille) '(Paris Limoges))

NIL

CG-USER(106): (f-goal-test node-paris '(Calais Marseille) '(Paris))

NIL

CG-USER(107): (f-goal-test node-calais '(Calais Marseille) '(Paris Nancy))

T

CG-USER(108): (f-goal-test nil '(hola hola) '(ninguno))

NIL

*Como decision hemos tomado que node-nevers no puede ser un nodo de estado final,
pues representa que no ha habido ningún camino

CG-USER(109): (f-goal-test node-nevers nil nil)

NIL

**function auxiliar

CG-USER(110): (get-camino node-calais nil)

(CALAIS REIMS NANCY PARIS)

CG-USER(111): (get-camino node-nevers nil)

NIL

CG-USER(116): (get-camino node-paris '(madrid barcelona))

(PARIS MADRID BARCELONA)5
```

Ejercicio4

En este ejercicio nos piden crear una función para ver si dos estados de búsqueda son iguales (misma ciudad y mismas ciudades obligatorias visitadas).

Hemos creado dos funciones auxiliares, una que devuelve las ciudades obligatorias no visitadas y otra que devuelve si dos listas son “equivalentes”, es decir, mismos elementos aunque no tienen por que estar en el mismo orden.

Pseudocódigo de las funciones:

Equivalentes (lst1 lst2)

```
Input: lst1(lista), lst2(lista)

Output: T si son equivalentes y nil si no es así

Proceso:

        Devolver equivalentes-aux(lst1,lst2) AND equivalentes-aux(lst2, lst1)
```

Equivalentes-aux (lst1 lst2)

```
Input: lst1, lst2

Output: T si todos los elementos de lst1 están en lst2 y nil si no es así

Proceso:

        Si rest(lst1) = null:

                Devolver presente(lst1[0], lst2)

        Si no:

                Devolver presente(lst1[0], lst2) AND equivalentes-aux(rest(lst1), lst2)
```

No-visitados(camino mandatory lst)

```
Input: camino (lista ciudades del camino), mandatory(lista ciudades obligatorias),
lst(lista ciudades obligatorias visitadas)

Output: lst(lista de ciudades obligatorias no visitadas)

Proceso:

        Si presente(mandatory[0], camino) = nil:

                Añadir a lst (mandatory[0])

                Devolver no-visitados(camino rest(mandatory) lst)

        Si no:

                Devolver no-visitados(camino rest(mandatory) lst)
```

f-search-state-equal(node-1 node-2 mandatory(opcional))

```
Input: node-1(nodo), node-2(nodo), mandatory(ciudades obligatorias)

Output: T si son el mismo estado nil si no

Proceso:

        Si mandatory = null:

                Devolver node-1 == node-2
```

Si no:

Devolver (node-1 == node-2) AND equivalentes(no-visitados(node-1, nil mandatory, nil), no-visitados(node-2, nil, mandatory,nil))

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Devuelve si dos listas tienen los mismo elementos, sin importar
;  el orden
;
; Input:
; - lst1: primera lista
; -lst2: segunda lista
;
; Returns:
;    T si las listas tienen los mismos elementos, nil si no es asi
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
defun equivalentes (lst1 lst2)
   (and (equivalentes-aux lst1 lst2) (equivalentes-aux lst2 lst1)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Devuelve si una lista esta "contenida" en la otra
;
; Input:
; - lst1: primera lista
; -lst2: segunda lista
;
; Returns:
;    T si todos los elementos de lst1 estan en lst2
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
defun equivalentes-aux (lst1 lst2)
   (if (null (cdr lst1))
       (presente (first lst1) lst2)
   (and (presente (first lst1) lst2) (equivalentes-aux (cdr lst1) lst2))))
;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Devuelve la lista de los nodos "obligatorios" no visitados
;;
;; Input:
;; - camino: lista de las ciudades del camino
;; -mandatory: lista de las ciudades obligatorias
;; -lst : lista de las ciudades obligatorias no vistadas
;;
;; Returns:
;; -lst: lista de las ciudades obligatorias no vistadas
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun no-visitados (camino mandatory lst)
    (when (not (null mandatory))
    (if (not (presente (first mandatory) camino)) (append (list (first mandatory)) (no-visitados camino (cdr mandatory) lst) lst)
        (append (no-visitados camino (cdr mandatory) lst) lst))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; 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-2: 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)
    (cond
    ((not (and (node-p node-1) (node-p node-2))) nil)
    ((and (equal (node-state node-1) (node-state node-2)) (null mandatory)) T)
    (T (and (equivalentes (no-visitados (get-camino node-1 nil) mandatory nil) (no-visitados (get-camino node-2 nil) mandatory nil))
            (equal (node-state node-1) (node-state node-2))))))
```

Ejemplos de ejecución ejercicio 4:

```
CG-USER(144): (f-search-state-equal node-calais node-calais-2 '())
T
CG-USER(145): (f-search-state-equal node-calais node-calais-2 '(Reims))
NIL
CG-USER(146): (f-search-state-equal node-calais node-calais-2 '(Nevers))
T
CG-USER(147): (f-search-state-equal node-nancy node-paris '())
NIL

**Ejemplos de la funcion auxilair
CG-USER(148): (no-visitados nil '(madrid) nil)
(MADRID)
CG-USER(149): (no-visitados '(madrid barcelona) '(madrid paris lyon) nil)
(PARIS LYON)
CG-USER(150): (no-visitados '(madrid) nil nil)
NIL
```

Ejercicio5

En este ejercicio solo tenemos que definir dos estructuras de problemas, una para cuando se tenga en cuenta el coste de precio y otra para cuando se tenga en cuenta el coste del tiempo.

Código:

```
(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-state node) *canals*))
                     #'(lambda (node) (navigate-train-price (node-state 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-state node) *canals*))
                     #'(lambda (node) (navigate-train-time (node-state node) *trains* *forbidden*)))))
```

Ejercicio 6

En este ejercicio nos piden que creemos una función que expanda un nodo, como función auxiliar hemos creado una que se llema lista-definitiva-nodos, que devuelve una lista de nodos resultantes de expandir el primero aplicándole las posibles acciones.

Para expandir un nodo, primero sacamos las acciones que se pueden realizar en ese nodo, utilizando los operadores de la estructura problema. Y después para cada una de las acciones creamos un nodo con los atributos correspondientes.

Pseudocódigo:

```
Expand-node-operator (node cfun)
Input: node(nodo), cfun(función operador)
Output: Res (lista de estados finales al aplicar ese operador a ese nodo)
Proceso:
        Lista-acciones = cfun(node)
        Res = null
        Para cada i <= longitud( lista-acciones ):
                Añadir a Res estado-final-de(lista-acciones[i])
```

Devolver Res

Expand-node-action(parent action problem)
Input: parent(nodo padre), action (accion), problem(problema)
Output: Node resultante de aplicar esa acción a ese nodo en ese problema
Proceso:
Nodo node
Node.estado = estado-final-de(action)
Node.parent = parent
Node.depth = 1 + Depth-de(parent)
Node.g = g-de(parent) + coste-accion(action)
Node.h= aplicar función problema-f-h(problema) a estado-final(action)
Node.f = node.h + node.g

Devolver node

Expand-node(node parent)
Input: node(nodo a expandir), problem
Output: Res(una lista de nodos resultantes de explorar node)
Proceso:
Acciones = aplicar-operadores-deproblem(problem) a node

Devolver lista-nodos-def(node, problema acciones, nil)

Lista-nodos-def(node problema lst-acciones lst-nodes)
Input: node(nodo que expandimos), problema(problema con la información necesaria), lst-acciones(lista de acciones(y/o listas de acciones)), lst-nodes(lista de nodos resultantes de expandir node)
Output: lst-nodes(lista de los nodos resultantes de expandir node)
Proceso:
Cuando no sea null lst-acciones:
Acción = lst-acciones[0]
Si acción es estructura action:
Devolver Concatenar(en una lista):
Lista-nodos-def(node problema rest(lst-acciones) lst-nodes)
Expand-node-action(node action problem)
Lst-nodes
Si no(accion es lista de acciones):
Devolver Concatenar(un una lista):
Lista-nodos-def(node problema acción nil)
Lista-nodos-def(node problema rest(lst-acciones) lst-nodes)

Código de las funciones

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Creates a list with all the nodes that can be reached from the
;;  current one using all the operators in a given problem
;;
;;  Input:
;;    node:   the node structure from which we start.
;;    problem: the problem structure with the list of operators
;;
;;  Returns:
;;    A list (node_1,...,node_n) of nodes that can be reached from the
;;    given one
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun expand-node (node problem)
  (if (or (not (node-p node)) (not (problem-p problem)))
      nil
    (let ((acciones (mapcar #'(lambda(x) (funcall x node)) (problem-operators problem))))
      (lista-nodos-def node problem acciones nil))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  devuelve la lista de nodos resultantes de expandir un nodo aplicandole
;;  distintas acciones
;;
;;  Input:
;;    node:   nodo que expandimos
;;    problem: estructura problema que contiene la informacion necesaria
;;    lst-acciones: lista de acciones(o de listas de acciones) para expandir
;;    lst-nodes: lista de nodos resultantes de expandir node
;;
;;  Returns:
;;   lista de nodos que resultan de expandir node
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun lista-nodos-def (node problem lst-acciones lst-nodes)
  (when (not (null lst-acciones))
    (let
        ((accion (first lst-acciones)))
      (if (action-p accion)
          (append (lista-nodos-def node problem (cdr lst-acciones) lst-nodes)
                  (cons (expand-node-action node accion problem) lst-nodes))
        ;sacamos todos los nodos de esa lista de acciones y los concatenamos a la final
        (append (lista-nodos-def node problem accion nil)
                (lista-nodos-def node problem (cdr lst-acciones) lst-nodes))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Devueve un nodo segun la accion que se haga al padre (parent)
;;
;; Input:
;; -parent: nodo padre en el que aplicamos la accion
;; -action: accion que realizamos
;; -problem: es una estructura problema que tiene la lista de operadores
;;
;; Output:
;;      Nodo(estructura) hijo obtenido al realizar la accion al padre
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun expand-node-action (parent action problem)
    (if (or (null action) (null parent) (null problem)) nil
        (make-node
            :state (action-final action)
            :parent parent
            :action action
            :depth (+ 1 (node-depth parent))
            :g (+ (action-cost action) (node-g parent))
            :h (funcall (problem-f-h problem) (action-final action))
            :f (+ (funcall (problem-f-h problem) (action-final action)) (+ (action-cost action) (node-g parent))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Devuelve los estados a los que podemos llegar usando un operador
;; en un nodo
;;
;; Input:
;; -node: Nodo del estado actual
;; -cfun: funcion que sera el operador para obtener nuevos nodos
;;
;; Output:
;;      lista de los estados finales tras aplicar cfun en ese nodo
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun expand-node-operator (node cfun)
    (mapcar #'(lambda(x)(action-final x)) (funcall cfun node)))
```

Ejemplos de ejecución ejercicio 6

**Ejemplo de la función auxiliar expand-node-action

(Con este ejemplo comprobamos que esta función funcione bien)

Hemos ido pidiendo cada uno de los atributos de nodo porque en la terminal no salía el nodo al completo

```
CG-USER(95): (node-f (expand-node-action node-marseille-ex6 #S(ACTION :NAME NAVIGATE-
TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) *travel-fast*))

205.0

CG-USER(96): (node-g (expand-node-action node-marseille-ex6 #S(ACTION :NAME NAVIGATE-
TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) *travel-fast*))

75.0

CG-USER(97): (node-h (expand-node-action node-marseille-ex6 #S(ACTION :NAME NAVIGATE-
TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) *travel-fast*))

130.0

CG-USER(98): (node-depth (expand-node-action node-marseille-ex6 #S(ACTION :NAME
NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) *travel-fast*))

13

CG-USER(99): (node-action (expand-node-action node-marseille-ex6 #S(ACTION :NAME
NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) *travel-fast*))

#S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0)

CG-USER(100): (node-parent (expand-node-action node-marseille-ex6 #S(ACTION :NAME
NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) *travel-fast*))

#S(NODE :STATE MARSEILLE :PARENT NIL :ACTION NIL :DEPTH 12 :G ...)

CG-USER(101): (node-state (expand-node-action node-marseille-ex6 #S(ACTION :NAME
NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) *travel-fast*))

TOULOUSE
```

Ejemplos de ejecución de la función expand-node:

***ejemplo del enunciado**

```
CL-USER(53): (defparameter lst-nodes-ex6

  (expand-node node-marseille-ex6 *travel-fast*))

LST-NODES-EX6


CL-USER(54): (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 ...))
```

***introduciendo un problema no valido**

```
CL-USER(77): (expand-node node-paris nil)

NIL
```

***introduciendo un nodo no valido**

```
CL-USER(78): (expand-node nil *travel-cheap*)

NIL
```

***Paris con la heuristica sobre el precio.**

Hemos elegido parís porque tiene muchas conexiones con otras ciudades.

```
CL-USER(79): (expand-node node-paris *travel-cheap*)
(#S(NODE :STATE REIMS
         :PARENT #S(NODE :STATE PARIS
                         :PARENT #S(NODE
                                    :STATE
                                    NEVERS
                                    :PARENT
                                    NIL
                                    :ACTION
                                    NIL
                                    :DEPTH
                                    0
                                    :G
                                    ...)
                         :ACTION NIL
                         :DEPTH 0
                         :G ...)
         :ACTION #S(ACTION :NAME NAVIGATE-CANAL-PRICE
                           :ORIGIN PARIS
                           :FINAL REIMS
                           :COST 10.0)
         :DEPTH 1
         :G ...)
 #S(NODE :STATE NANCY
         :PARENT #S(NODE :STATE PARIS
                         :PARENT #S(NODE
                                    :STATE
                                    NEVERS
                                    :PARENT
                                    NIL
                                    :ACTION
                                    NIL
                                    :DEPTH
```

```
                                    0
                                    :G
                                    ...)
                          :ACTION NIL
                          :DEPTH 0
                          :G ...)
        :ACTION #S(ACTION :NAME NAVIGATE-CANAL-PRICE
                          :ORIGIN PARIS
                          :FINAL NANCY
                          :COST 10.0)
        :DEPTH 1
        :G ...)
#S(NODE :STATE CALAIS
        :PARENT #S(NODE :STATE PARIS
                          :PARENT #S(NODE
                                    :STATE
                                    NEVERS
                                    :PARENT
                                    NIL
                                    :ACTION
                                    NIL
                                    :DEPTH
                                    0
                                    :G
                                    ...)
                          :ACTION NIL
                          :DEPTH 0
                          :G ...)
        :ACTION #S(ACTION :NAME NAVIGATE-TRAIN-PRICE
                          :ORIGIN PARIS
                          :FINAL CALAIS
                          :COST 60.0)
        :DEPTH 1
        :G ...)
#S(NODE :STATE NANCY
        :PARENT #S(NODE :STATE PARIS
                          :PARENT #S(NODE
                                    :STATE
                                    NEVERS
                                    :PARENT
                                    NIL
                                    :ACTION
                                    NIL
                                    :DEPTH
                                    0
                                    :G
                                    ...)
                          :ACTION NIL
                          :DEPTH 0
                          :G ...)
        :ACTION #S(ACTION :NAME NAVIGATE-TRAIN-PRICE
                          :ORIGIN PARIS
                          :FINAL NANCY
                          :COST 67.0)
        :DEPTH 1
        :G ...)
#S(NODE :STATE NEVERS
        :PARENT #S(NODE :STATE PARIS
                          :PARENT #S(NODE
                                    :STATE
                                    NEVERS
```

```
                                    :PARENT
                                    NIL
                                    :ACTION
                                    NIL
                                    :DEPTH
                                    0
                                    :G
                                    ...)
                           :ACTION NIL
                           :DEPTH 0
                           :G ...)
          :ACTION #S(ACTION :NAME NAVIGATE-TRAIN-PRICE
                            :ORIGIN PARIS
                            :FINAL NEVERS
                            :COST 75.0)
          :DEPTH 1
          :G ...)
  #S(NODE :STATE ORLEANS
          :PARENT #S(NODE :STATE PARIS
                          :PARENT #S(NODE
                                    :STATE
                                    NEVERS
                                    :PARENT
                                    NIL
                                    :ACTION
                                    NIL
                                    :DEPTH
                                    0
                                    :G
                                    ...)
                           :ACTION NIL
                           :DEPTH 0
                           :G ...)
          :ACTION #S(ACTION :NAME NAVIGATE-TRAIN-PRICE
                            :ORIGIN PARIS
                            :FINAL ORLEANS
                            :COST 38.0)
          :DEPTH 1
          :G ...)
  #S(NODE :STATE ST-MALO
          :PARENT #S(NODE :STATE PARIS
                          :PARENT #S(NODE
                                    :STATE
                                    NEVERS
                                    :PARENT
                                    NIL
                                    :ACTION
                                    NIL
                                    :DEPTH
                                    0
                                    :G
                                    ...)
                           :ACTION NIL
                           :DEPTH 0
                           :G ...)
          :ACTION #S(ACTION :NAME NAVIGATE-TRAIN-PRICE
                            :ORIGIN PARIS
                            :FINAL ST-MALO
                            :COST 70.0)
          :DEPTH 1
          :G ...))
```

Ejercicio7

En este ejercicio nos piden implementar un función que inserte una lista de nodos en otra(ya ordenada) de manera que quede la lista ordenada en función de un estrategia, pasada como parámetro de entrada.

En este ejercicio nos proporcionan el modelo de dos funciones que necesitaremos crear, aún así, nosotros hemos creado una tercera que inserta un único nodo en una lista ya ordenanda.

Pseudocódigo de la funciones:

```
Insert-node (node lst-nodes node-compare-p)
Input: node(nodo), lst-nodes(lista de nodos ya ordenada), node-compare-p (función que
compara dos nodos)
Output: lst-nodes (lista de nodos ordenada)
Proceso:
	Si lst-nodes = null:
		Devolver Lst-nodes
	// Como esto es una función interna si pasa eso habría habido un error
	// al llamar a esa función
	Si no es un nodo node:
		Devolver nil
	si node-compare-p (node, lst-nodes[i]) = True:
		añadir-delante(node, lst-nodes)
	si no:
		añadir delante(lst-nodes[0], insert-node(node, rest(lst-nodes)))

insert-nodes (nodes lst-nodes node-compare-p)
Input: nodes(lista de nodos), lst-nodes(lista de nodos ya ordenada), node-compare-p
(función que compara dos nodos)
Output: lista ya ordenada con todos los nodos de ambas listas
Proceso:
	If nodes = null:
		Devolver lst-nodes
	Si no:
		Lst-nodes-2 = insert-node(nodes[i], lst-nodes, node-compare-p)
		Devolver insert-nodes (rest(nodes), lst-nodes-2, node-compare-p)

Insert-nodes-strategy(nodes lst-nodes strategy)
Input: nodes(lista de nodos), lst-nodes(lista de nodos ya ordenada),
strategy(estrategia para saber como ordenar los nodos)
Output: lista con todos los nodos ya ordenada
Proceso:
	Si nodes = null
		Devovler lst-nodes
	Si lst-nodes = null
		Devolver insert-nodes-strategy(rest(nodes), lista-de-nodes[0],
strategy)
	Si no:
		Devolver insert-nodes(nodes, lst-nodes, funcion-comparar-de(strategy))
```

Código de las funciones:

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; 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.
;;
(defun insert-nodes (nodes lst-nodes node-compare-p)
    (if (null nodes) lst-nodes
        (let ((lst-nodes-2 (insert-node (first nodes) lst-nodes node-compare-p)))
        (insert-nodes (cdr nodes) lst-nodes-2 node-compare-p))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Inserts a list of nodes in an ordered list keeping the result list
;; ordered with respect the given strategy
;;
;; 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
;;    strategy: the strategy that gives the criterion for node
;;              comparison
;;
;; 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 defined in te strategy.
;;
;; Note:
;;   You will note that this function is just an interface to
;;   insert-nodes: it allows to call using teh strategy as a
;;   parameter; all it does is to "extract" the compare function and
;;   use it to call insert-nodes.
;;
(defun insert-nodes-strategy (nodes lst-nodes strategy)
    (cond ((not (strategy-p strategy)) nil)
        ((null nodes) lst-nodes)
        ((null lst-nodes) (insert-nodes-strategy (cdr nodes) (list (first nodes)) strategy))
        (T (insert-nodes nodes lst-nodes (strategy-node-compare-p strategy)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Inserta un nodo en una lista ordenanda
;;
;; Input:
;; -node: nodo que queremos insertar
;; -lst-nodes: lista de nodos ordenados
;; -node-compare-p: funcion para comparar dos nodos
;;      (node-compare-p nodo-1 nodo-2) devuelve T si el primer nodo va antes
;;
;; Output:
;;      lst-nodes: lista de nodos ordenados con node incluido
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun insert-node (node lst-nodes node-compare-p)
    (cond ((not (node-p node)) nil)
        ((null lst-nodes) (list node))
        ;; node va antes que el primero de lst-nodes
        (T (if (funcall node-compare-p node (first lst-nodes)) (append (list node) lst-nodes)
        ;;node va despues que el primero
            (append (list (first lst-nodes)) (insert-node node (cdr lst-nodes) node-compare-p))))))
```

Ejemplos de ejecución ejercicio7:

```
**función auxiliar insert-node:
CG-USER(91): (defparameter aux (insert-node node-paris-ex7 (list node-nancy-ex7)
(strategy-node-compare-p *uniform-cost*)))
AUX

CG-USER(92): (mapcar #'(lambda (x) (node-state x)) aux)
(PARIS NANCY)
```

(a esta función se le llama desde otras el control de errores de los otros dos parámetros ya se habría realizado previamente)

```
CG-USER(93): (insert-node nil (list node-nancy-ex7) (strategy-node-compare-p
*uniform-cost*))
NIL
```

Insert-nodes-strategy básicamente llama a insert-nodes, también realiza los primeros controles de errores, pasándole la función comparación, por lo que directamente vamos a mostrar los resultados de ejecutar esta función:

```
CG-USER(97): sol-ex7
(#S(NODE :STATE PARIS :PARENT NIL :ACTION NIL :DEPTH 0 :G ...) #S(NODE :STATE NANCY
:PARENT NIL :ACTION NIL :DEPTH 2 :G ...)
 #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 ...))

CG-USER(98): (mapcar #'(lambda (x) (node-state x)) sol-ex7)
(PARIS NANCY TOULOUSE)

CG-USER(99): (mapcar #'(lambda (x) (node-g x)) sol-ex7)
(0 50 75.0)
```

Si no introducimos ninguna estrategia válida:

```
CG-USER(132): (insert-nodes-strategy (list node-paris-ex7 node-nancy-ex7)
                                             lst-nodes-ex6
                                             nil)
NIL
```

Si no introducimos nodos a ordenar:

```
CG-USER(133): (insert-nodes-strategy nil
                                             lst-nodes-ex6
                                             *uniform-cost*)
(#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 ...))

CG-USER(134): (equal (insert-nodes-strategy nil
                                             lst-nodes-ex6
                                             *uniform-cost*) lst-nodes-ex6)
T
```

Si no introducimos una lista de nodos ordenados:

```
CG-USER(136): (insert-nodes-strategy (list node-paris-ex7 node-nancy-ex7) nil
                                          *uniform-cost*)
(#S(NODE :STATE PARIS :PARENT NIL :ACTION NIL :DEPTH 0 :G ...) #S(NODE :STATE NANCY
:PARENT NIL :ACTION NIL :DEPTH 2 :G ...))

CG-USER(137): (mapcar #'(lambda (x) (node-state x)) (insert-nodes-strategy (list
node-paris-ex7 node-nancy-ex7) nil
                                          *uniform-cost*)
)
(PARIS NANCY)

CG-USER(138): (mapcar #'(lambda (x) (node-g x)) (insert-nodes-strategy (list node-
paris-ex7 node-nancy-ex7) nil
                                          *uniform-cost*))
(0 50)
```

Ejercicio 8

En este ejercicio solo nos piden inicializar una variable global cuyo valor sea la estrategia para realizar la búsqueda A*.

Pseudocódigo de la función que compara dos nodos:
Node-f-<= (node-1 node-2)
Input: node-1(nodo), node-2(nodo)
Output: T si el node-1 va antes que el 2(tiene menor f) o nil si no es así
Proceso:
Devolver obtener-f(node-1) <= obtener-f(node-2)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; funcion que compara dos nodos segun su valor en f
;;
;; Input:
;; -node-1 primer nodo que se compara
;; node-2 segundo nodo que se compara
;;
;; Output:
;;      T si la f del primero es menor que la del 2
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun node-f-<= (node-1 node-2)
  (<= (node-f node-1)
      (node-f node-2)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Estructura estrategia para A*
;;
;; De nombre le hemos puesto f-cost
;; Utilizamos para comparar la f pues esta es la estrategia que
;; se utiliza en A*
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defparameter *A-star*
  (make-strategy
    :name 'f-cost
    :node-compare-p #'node-f-<=))
```

Ejercicio9

En este ejercicio nos piden codificar una función de búsqueda en un grafo, y otra que sea concretamente búsqueda utilizando la estrategia A*.

Para este ejercicio hemos seguido el pseudocódigo proporcionado en el enunciado.

Pseudocódigo de las funciones:

Devolver-nodo(node lst)

```
Input: node(nodo que buscamos), lst(lista donde lo buscamos)

Output: Res(nodo que sea igual(según la función de mismo estado) al node y pertenezca
a lst)

Proceso:
        Si lst = null:
                Devolver nil
        Si node = lst[0]:
                Devolver lst[0]
        Si no:
                Devolver devolver-nodo(node, rest(lst))
```

Graph-search-aux(problem open-nodes closed-nodes strategy)

```
Input: problem(problema, que contiene la informacion general), open-nodes(lista de
nodos descubiertos pero no explorados), closed-nodes(lista de nodos ya explorados),
strategy(estrategia que vamos a seguir para resolver la búsqueda).

Output: Res, nodo que corresponde al estado objetivo o nil en caso de no encontrarse
solución

Proceso:
                Si open-nodes = null:
                      Devolver nil
                Si no:
                      Primer-nodo = open-nodes[0]
                      Si primer-nodo = OBJETIVO:
                             Devolver primer-nodo

                      Copia = devolver-node(primer-nodo, closed-nodes)
                      Si copia = null:
                             Open-nodes = añadir-a-cdr-open-nodes(expand-node-
                             strategy(primer-nodo))

                             Devolver graph-search-aux(problema, open-nodes,
                                    Añadir-aprimer-node, closed-nodes), strategy)

                      Si obtener-g(primer-nodo) <= obtener-g(copia):
                                                  Open-nodes = añadir-a-cdr-open-
                             nodes(expand-node-strategy(primer-nodo))

                             Devolver graph-search-aux(problema, open-nodes,
                                    Añadir-aprimer-node, closed-nodes), strategy)

                      Si no:
                             Devolver graph-search-aux(problem, rest(open-nodes),
                      añadir-a(primer-nodo, closed-nodes), strategy)
```

Graph-search(problem strategy)

Input: problem(estructura problema, contiene la información general), strategy(estrategia para seguir en la búsqueda)

Output: Nodo final objetivo o nil si no se ha encontrado

Proceso:
Devolver graph-search-aux(problema, crear-lista(make-node(problema-inicio)),nil, strategy)

a-star-search (problem)
Input: problem(problema, contiene la información general que necesitaremos para resolver la búsqueda con A*)

Output: Output: Nodo final objetivo o nil si no se ha encontrado

Proceso:
Devolver: graph-search(problema ***A-star***)

Código de las funciones

**en graph-search-aux hemos utilizado if's anidados, en vez de cond's, porque así seguimos de manera lógica el problema, esto ya ha quedado reflejado en el pseudcódigo

```
(defun graph-search-aux (problem open-nodes closed-nodes strategy)
  (if (null open-nodes)
      nil ; no hay solucion
    (let ((primer-nodo (first open-nodes)))
      (if (funcall (problem-f-goal-test problem) primer-nodo)
          primer-nodo ; evaluo a solucion
        (let ((simil (devolver-nodo primer-nodo closed-nodes)))
          (if (or (null simil) (< (node-g primer-nodo) (node-g simil)))
              (graph-search-aux
               problem
               (insert-nodes-strategy
                (expand-node primer-nodo problem)
                (cdr open-nodes)
                strategy)
               (cons primer-nodo closed-nodes) strategy )
            (graph-search-aux problem (cdr open-nodes)
                              (cons primer-nodo closed-nodes)
                              strategy)))))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Devuelve el primer elemento que encuentre de la lista que sea igual a nodo
;;
;; Input:
;;  -nodo: nodo que buscamos que este en la lista (teniendo en cuenta el estado)
;;  -lst: lista de nodos
;;
;; Output:
;;      Un nodo igual si lo encuentra o nil, si no
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun devolver-nodo (nodo lst)
  (cond
   ((null lst)
    nil)
   ((equal (node-state nodo)
    (node-state (first lst))) (first lst))
   (T (devolver-nodo nodo (cdr lst)))))
```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  Interface function for the graph search.
;;
;;  Input:
;;    problem: the problem structure from which we get the general
;;             information (goal testing function, action operatos,
;;             starting node, heuristic, etc.
;;    strategy: the strategy that decide which node is the next extracted
;;              from the open-nodes list
;;
;;    Returns:
;;     NIL: no path to the destination nodes
;;     If these is a path, returns the node containing the final state.
;;
;;    See the graph-search-aux for the complete structure of the
;;    returned node.
;;    This function simply prepares the data for the auxiliary
;;    function: creates an open list with a single node (the source)
;;    and an empty closed list.
;;
(defun graph-search (problem strategy)
  (when (and (problem-p problem) (strategy-p strategy))
    (graph-search-aux problem (list (make-node :state (problem-initial-state problem))) '() strategy )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  A* search is simply a function that solves a problem using the A* strategy
;;
;; Input: problem (problema a resolver con busqueda A*)
;; Output: nodo solucion
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun a-star-search (problem)
  (graph-search problem *A-star*))
```

Ejemplos de ejecución ejercicio 9:

```
CG-USER(13): (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 # :ACTION # :DEPTH 2 :G ...)
                                                        :ACTION #S(ACTION :NAME
NAVIGATE-TRAIN-PRICE :ORIGIN LIMOGES :FINAL NEVERS :COST 60.0)
                                                        :DEPTH 3
                                                        :G ...)
                                        :ACTION #S(ACTION :NAME NAVIGATE-CANAL-PRICE
:ORIGIN NEVERS :FINAL PARIS :COST 10.0)
                                        :DEPTH 4
                                        :G ...)
                        :ACTION #S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN PARIS
:FINAL REIMS :COST 10.0)
                        :DEPTH 5
                        :G ...)
        :ACTION #S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN REIMS :FINAL CALAIS
:COST 15.0)
        :DEPTH 6
        :G ...)

**Si introducimos parámetros de entrada no válidos devolverá nil
CG-USER(14): (graph-search nil *A-star*)
NIL

CG-USER(15): (graph-search *travel-cheap* nil)
NIL

CG-USER(16): (graph-search *travel-cheap* 'jola)
NIL

CG-USER(17): (graph-search 'novalido *A-star*)
```

```
NIL

CG-USER(23): (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 # :ACTION # :DEPTH 1 :G ...)
                                                        :ACTION #S(ACTION :NAME
NAVIGATE-TRAIN-TIME :ORIGIN TOULOUSE :FINAL LIMOGES :COST 25.0)
                                                        :DEPTH 2
                                                        :G ...)
                                        :ACTION #S(ACTION :NAME NAVIGATE-TRAIN-TIME
:ORIGIN LIMOGES :FINAL ORLEANS :COST 55.0)
                                        :DEPTH 3
                                        :G ...)
                        :ACTION #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN ORLEANS
:FINAL PARIS :COST 23.0)
                        :DEPTH 4
                        :G ...)
        :ACTION #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN PARIS :FINAL CALAIS :COST
34.0)
        :DEPTH 5
        :G ...)
```

Ejercicio10

En esta función nos piden calcular el camino(nombre de las ciudades) hasta llegar a un nodo (resultado de un búsqueda) y la secuencia de acciones para llegar a un nodo.

Pseudocódigo de las funciones:

```
Solution-path-aux(node lst)
Input: node(nodo del que buscamos el camino), lst(lista de estados acumulados)
Output: lista de estados
Proceso:
                Si node-parent = null:
                      Devolver añadir-alista(node-state, lst)
                añadir-a-lista (node-state, lst)
                devolver añadir-a-lista(node-path-aux(node-parent), lst)

solution-path(node)
Input: node(nodo del que buscamos el camino)
Output: lista de estados hasta llegar al nodo
Proceso:
                Devolver solution-path-aux(node, nil)

Action-sequence-aux(node, lst)
Input: node(nodo del que buscamos secuencias de acciones), lst(acciones acumuladas
hasta ahora)
Output: lista de secuencia de acciones
Proceso:
                Si node-parent = null
                      Devolver: añadir-a-lista(node-accion, lst)
                Si no:
                      Añadir-a-lista(node-accion, lst)
                      Devolver añadir-a-lista(action-sequence-aux(node, lst), lst)

Action-sequence(node)
```

```
Input: node(nodo del cual queremos saber la secuencia de acciones hasta él)
Output: lista de acciones
Proceso:
                Si node-parent = null
                      Devolver: node-accion
                Si no:
                      Devolver action-sequence-aux(node-parent,lista-de(node-action))
```

Código de las funciones:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Devuelve el camino hasta un nodo solucion
;;
;; Input:
;;  -node: nodo solucion del cual buscamos el camino
;;
;; Output: lista de estados hasta el nodo
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun solution-path (node)
  (if (not (node-p node)) nil
    (solution-path-aux (node-parent node) (list (node-state node)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Auxiliar de solution-path
;;
;; Input:
;; -node: nodo del cual buscamos el camino
;; -lst: lista d estados acumulados hasta ese nodo
;;
;; Output:
;;       lst, lista de los estados del camino
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun solution-path-aux (node lst)
  (if (null (node-parent node)) (cons (node-state node) lst)
    (solution-path-aux (node-parent node) (cons (node-state node) lst))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Devuelve el camino de acciones hasta un nodo solucion
;;
;; Input:
;;  -node: nodo solucion del cual buscamos el camino
;;
;; Output: lista de acciones hasta el nodo
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun action-sequence (node)
  (if (not (node-p node)) nil
    (eliminar-nil (action-sequence-aux (node-parent node) (list (node-action node))))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Auxiliar de action-sequence-aux
;;
;; Input:
;; -node: nodo del cual buscamos el camino
;; -lst: lista de acciones acumuladas hasta ese nodo
;;
;; Output:
;;       lst, lista de las acciones del camino
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun action-sequence-aux (node lst)
  (if (null (node-parent node)) (cons (node-action node) lst)
    (action-sequence-aux (node-parent node) (cons (node-action node) lst))))

;;;
```

Ejemplos de ejecución del ejercicio 10:

```
CG-USER(30): (solution-path nil)
NIL

CG-USER(31): (solution-path (a-star-search *travel-fast*))
(MARSEILLE TOULOUSE LIMOGES ORLEANS PARIS CALAIS)

CG-USER(32): (solution-path (a-star-search *travel-cheap*))
(MARSEILLE TOULOUSE LIMOGES NEVERS PARIS REIMS CALAIS)

CG-USER(58): (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))

CG-USER(59): (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))
```

Ejercicio11

En este ejercicio nos piden definir estrategias y funciones de comparación de nodos para búsqueda en anchura y búsqueda en profundidad.

Pseudocódigo de las funciones:

Depth-first-node-compare-p(node-1 node-2)

```
Input: node-1, node-2 (nodos que queremos comparar)
Output: T si node-1 tiene una profundidad mayor o igual a node-2 y nil en caso
contrario.
Proceso:
        Si profundidad(node-1) >= profundidad(node-2):
                Devolver T
        Si no:
                Devolver nil
```

Breadth-first-node-compare-p(node-1 node-2)

```
Input: node-1, node-2 (nodos que queremos comparar)
Output: T si node-1 tiene una profundidad menor o igual a node-2 y nil en caso
contrario.
Proceso:
        Si profundidad(node-1) <= profundidad(node-2):
                Devolver T
        Si no:
                Devolver nil
```

Código de las funciones:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Funcion de comparacion de nodos para busqueda en profundidad
;;
;; Input:
;; -node-1: nodo que queremos comparar con node-2
;; -node-2: nodo que queremos comparar con node-1
;;
;; Output:
;;      true si la profundidad del node-1 es mayor o igual que la de node-2
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun depth-first-node-compare-p (node-1 node-2)
  (>= (node-depth node-1) (node-depth node-2)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Estructura estrategia para busqueda en profundidad
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defparameter *depth-first*
  (make-strategy
   :name 'depth-first
   :node-compare-p #'depth-first-node-compare-p))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Funcion de comparacion de nodos para busqueda en anchura
;;
;; Input:
;; -node-1: nodo que queremos comparar con node-2
;; -node-2: nodo que queremos comparar con node-1
;;
;; Output:
;;      true si la profundidad del node-1 es menor o igual que la de node-2
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun breadth-first-node-compare-p (node-1 node-2)
  (<= (node-depth node-1) (node-depth node-2)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Estructura estrategia para busqueda en anchura
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defparameter *breadth-first*
  (make-strategy
   :name 'breadth-first
   :node-compare-p #'breadth-first-node-compare-p))
```

Ejemplos de ejecución ejercicio11:

```
Break 4 [50]> (solution-path (graph-search *travel-fast* *depth-first*))
(MARSEILLE TOULOUSE LYON ROENNE NEVERS LIMOGES ORLEANS NANTES BREST ST-MALO
 PARIS NANCY REIMS CALAIS)
```

```
Break 4 [50]> (solution-path (graph-search *travel-cheap* *breadth-first*))
(MARSEILLE TOULOUSE NANTES ORLEANS PARIS CALAIS)
```

Ejercicio12

En este último ejercicio tenemos que crear una heurística admisible para la opción de viajar barato. Definimos la heurística de manera que a cada nodo le corresponde la mitad del coste del camino hasta el nodo objetivo. Esto, teniendo en cuenta el camino más corto y, si se puede viajar en canal, teniendo en cuenta el precio del canal dado que es más barato. Esta heurística es admisible ya que por definición es menor que el coste real en cualquier caso.

Después de realizar los cálculos obtenemos la siguiente lista de estimaciones y la siguiente definición del problema de coste mínimo con la nueva heurística:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Definicion de nuevas estimaciones de coste para la heuristica creada
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defparameter *estimate-new*
  '((Calais (0.0 0.0)) (Reims (25.0 7.5)) (Paris (30.0 12.5))
    (Nancy (50.0 17.5)) (Orleans (55.0 31.5)) (St-Malo (65.0 47.5))
    (Nantes (75.0 55.0)) (Brest (90.0 62.5)) (Nevers (70.0 22.5))
    (Limoges (100.0 74.0)) (Roenne (85.0 25.0)) (Lyon (105.0 27.5))
    (Toulouse (130.0 75.0)) (Avignon (135.0 47.5)) (Marseille (145.0 60.0))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Definicion del problema de coste minimo con la nueva heuristica de coste
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defparameter *travel-cost-new*
  (make-problem
   :states *cities*
   :initial-state *origin*
   :f-h #'(lambda(state) (f-h-price 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) *canals*))
                    #'(lambda (node) (navigate-train-price (node-state node) *trains* *forbidden*)))))
```

A la hora de medir los tiempos de ejecución de la búsqueda con la heurística de coste cero y la recién definida por nosotros obtenemos los siguientes resultados en los que podemos notar una notable mejora en el tiempo de búsqueda y una buena reducción del espacio ocupado:

```
Break 10 [56]> (time (graph-search *travel-cheap* *A-star*))
Real time: 0.0079895 sec.
Run time: 0.0 sec.
Space: 34428 Bytes
```

```
Break 10 [56]> (time (graph-search *travel-cost-new* *A-star*))
Real time: 0.0020172 sec.
Run time: 0.0 sec.
Space: 22996 Bytes
```

Respuesta a las preguntas finales:

1.

Para el diseño utilizado nos hemos guiado de las indicaciones de la práctica. Tenemos una estructura para cada “objeto” que utilizamos y que creemos que necesita distinción, nodos,acciones,problema…

2.

(a)

La principal ventaja es que podemos desglosar el problema, pues si queremos obtener descendientes de un nodo, por ejemplo, no necesitamos toda la información del problema, con el nodo y la acción que le aplicamos sería suficiente; es decir, podemos desglosar el problema en partes, para solucionarlo.

Otra ventaja es que facilita el desarrollo de la práctica, pues, nosotros a la hora de pensarlo también vamos dividiendo el problema en partes.

(b)

La razón de utilizar funciones lambda es porque así podríamos crear varias estructuras de problema y nosotros ir especificando distintas heurísticas, operadores… Esto nos da libertad, pues podemos adaptar un problema que tengamos (que no sea necesariamente el ejemplo de la práctica) a nuestra estructura.

3.

Si, pues estamos usando punteros y para ciertas funciones, por ejemplo hallar el camino de estados hasta un nodo, deberíamos de guardar toda la estructura, o el estado y luego buscarlo en una lista de nodos… De estas otras maneras tendríamos que utilizar mucha mas memoria, pues se por ejemplo un nodo tiene 100 hijos habría que guardar 100 veces la estructura nodo; o si lo hiciésemos guardando solo el estado y tuviésemos un gran número de nodos habría que buscar siempre en una lista de muchos elementos y esto puede resultar ineficiente.

4.

La complejidad espacial de A* es exponencial con respecto al tamaño del problema debido a que se tienen que almacenar todos los posibles siguientes nodos de cada estado. Normalmente es el factor limitante del algoritmo.

5.

La complejidad temporal de A* está estrechamente relacionada con la heurística utilizada. La ejecución tomará un tiempo exponencial cuando la heurística no sea de buena calidad, mientras que en el mejor caso el algoritmo resolverá el problema en un tiempo lineal. La condición necesaria para que esto ocurra es que la diferencia entre el valor heurístico y el coste real sea menor o igual que O(log h*(n)) (Siendo h*(n) el coste real).

6.

La función de búsqueda implementada graph-search-aux es similar a la función búsqueda-en-grafo de las transparencias de teoría. Sí la heurística utilizada es monótona, A* usando búsqueda-en-grafo es completa y óptima, por lo tanto, no habría problemas con pasar demasiadas veces por un camino bidireccional. Aún así, en el caso de que la heurística utilizada no fuera monótona podríamos solucionar el problema añadiendo a cada elemento del parámetro *trains* otro número en la lista de costes que indique el máximo número de veces que se puede pasar por ese enlace. Contando con este dato, podríamos modificar la función f-goal-test para que compruebe, además de que has pasado por las ciudades obligatorias, que no sobrepases el número de veces que se puede usar el enlace.