PRÁCTICA 2

INTELIGENCIA ARTIFICIAL

Jorge Santisteban Rivas
Javier Martínez Rubio

Ejercicio 1

F-h-time / F-h-cost

Pseudocódigo

Input:
state: la ciudad en la que estamos
sensors: una lista con heurísticas, que es una lista con pares
(state (time-est cost-est))
donde el primer elemento es el nombre de la ciudad y el segundo
un numero que estima el coste(temporal o de precio)

Returns:
El coste (un numero) o NIL si la ciudad no esta en la lista de heurísticas

Devuelve el valor de las heurísticas de una determinada ciudad

Código

```
;;; f-h-time (state-sensors) / f-h-cost (state-sensors)
;;; Devuelven el coste de un estado, o nil si no existe ese estado

(defun f-h-time (state sensors)
  (first (second (assoc state sensors))))

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

Comentarios

Estas funciones simplemente devuelven el coste estimado predefinido en la lista de heurísticas. En el caso de la heurística de coste, siempre se va a devolver 0, ya que es la heurística por defecto para todos los estados.

Ejemplos

```
CL-USER> (f-h-time 'Nantes *estimate*)

75.0

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

CL-USER> (f-h-price 'Lyon *estimate*)
0.0

CL-USER> (f-h-price 'Madrid *estimate*)
NIL
```

Ejercicio 2

En este ejercicio vamos a utilizar funciones auxiliares. Primero definimos cuatro funciones muy simples que devuelven el punto de partida, el punto de destino, y los costes de tiempo y de precio. Estas funciones son las siguientes:

```
;;Funciones auxiliares para obtener el estado inicio, el estado final y los costes

(defun start (edge)
  (first edge))

(defun end (edge)
  (second edge))

(defun cost-time (edge)
  (first (third edge)))

(defun cost-price (edge)
  (second (third edge)))
```

Una vez definidas estas funciones, creamos una función navigate de carácter general. Al ser las cuatro funciones solicitadas muy parecidas, ésta función la podremos utilizar para todas ellas, ya que simplemente tendremos que modificar los argumentos de entrada.

Navigate

Pseudocódigo

Input:

state: la ciudad desde la que queremos realizar la acción
lst-edges: lista de las aristas del grafo, cada elemento es de la forma: (source destination (cost1 cost2))
c-fun: funcion que extrae el coste correcto (tiempo o precio) del par que aparece en la arista
name: nombre dado a las acciones que se crean (ver la estructura de action)
forbidden-cities:
lista de las ciudades que no se pueden visitar en tren

Returns

Una lista con estructuras action con el origen en la ciudad actual y el destino en las ciudades a las cuales el origen está conectado

Devuelve las acciones que se pueden realizar desde una ciudad

Código

```
;; Funcion de navegación general

(defun navigate (state lst-edges cfun  name &optional forbidden )

  (remove nil (mapcar #'(lambda(x) (if (and (eql state (start x)) ;;Comprobamos que state sea el inicio
                                            (NULL (find (end x) forbidden))) ;;Comprobamos que el destino no esté en forbidden
                                       (make-action :name name
                                                    :origin state
                                                    :final (end x)
                                                    :cost (funcall cfun x))
                                       NIL)) lst-edges)))
```

Comentarios

De esta función simplemente comentar que, como la función mapcar va a devolver una lista con todo nils menos uno (el nodo del cual queremos saber la acción), hemos tenido que poner delante un remove nil.
Una vez implementada la función general de navigate, creamos las 4 solicitadas, que simplemente llamarán a esta con diferentes parámetros.

Navigation by canal

Código

```
(defun navigate-canal-time (state canals)
  (navigate state canals #'cost-time 'NAVIGATE-CANAL-TIME))

(defun navigate-canal-price (state canals)
  (navigate state canals #'cost-price 'NAVIGATE-CANAL-PRICE))
```

Comentarios
Esto es una especialización de la funcion de navegar general:dado una ciudad y una lista de canales, devuelve una lista con acciones a las que navegar desde la ciudad origen hasta las ciudades alcanzables desde el origen a través de los canales.

Navigation by train

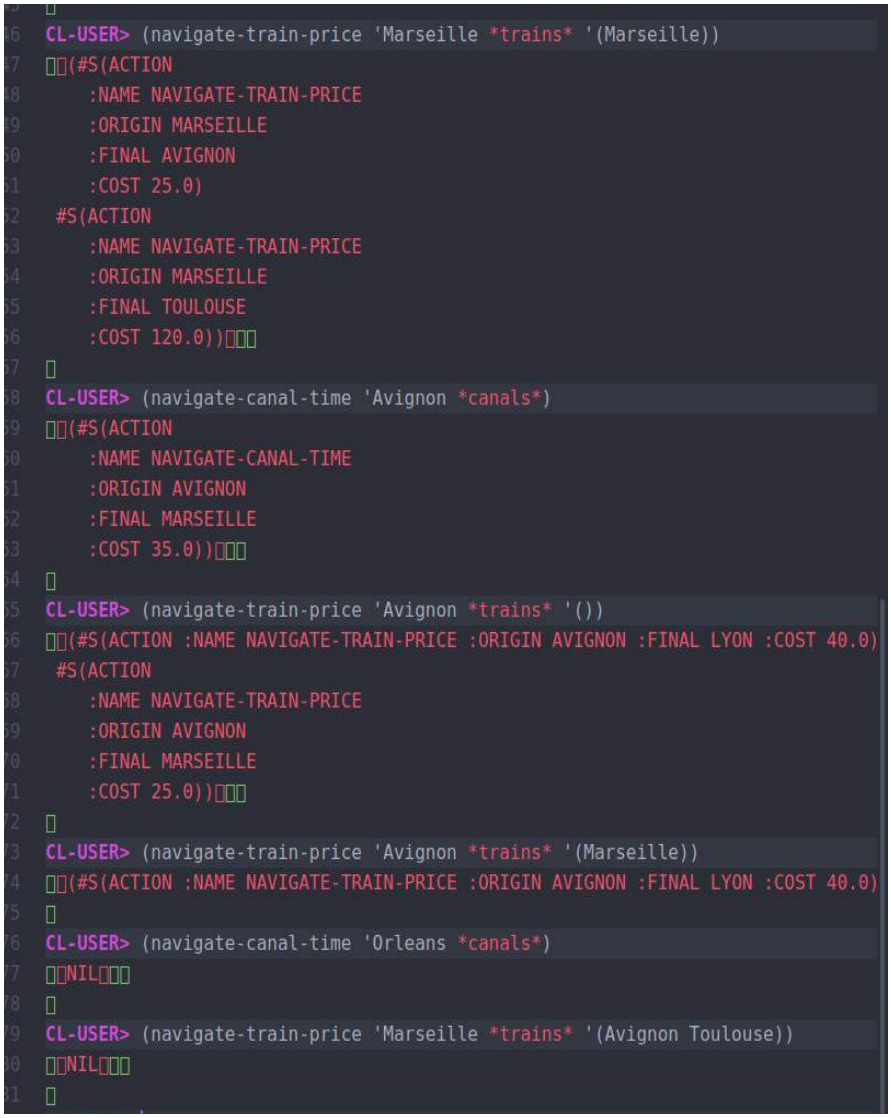
Código

```
(defun navigate-train-time (state trains forbidden)
  (navigate state trains #'cost-time 'NAVIGATE-TRAIN-TIME forbidden))

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

Comentarios

Esto es una especializacion de la funcion de navegar general: dado una ciudad y una lista de trenes, devuelve una lista con acciones a las que viajar desde la ciudad origen hasta las ciudades alcanzables desde el origen a través del tren. Tener en cuenta que esta función utiliza una lista de ciudades prohibidas.

Ejemplos



```
CL-USER> (navigate-train-price 'Marseille *trains* '(Marseille))
(#S(ACTION
    :NAME NAVIGATE-TRAIN-PRICE
    :ORIGIN MARSEILLE
    :FINAL AVIGNON
    :COST 25.0)
 #S(ACTION
    :NAME NAVIGATE-TRAIN-PRICE
    :ORIGIN MARSEILLE
    :FINAL TOULOUSE
    :COST 120.0))

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

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

CL-USER> (navigate-canal-time 'Orleans *canals*)
NIL

CL-USER> (navigate-train-price 'Marseille *trains* '(Avignon Toulouse))
NIL
```

Ejercicio 3

En este ejercicio también utilizaremos una función auxiliar, llamada f-goal-path, con la cual podremos averiguar si se llega a una ciudad final

F-goal-path

Pseudocódigo

Input:

node: nodo estructura que contiene, en la cadena de nodos padres,
un camino empezando en la ciudad inicial
mandatory: lista con los nombres de las ciudades que son obligatorias visitar

Returns

T: el camino es un camino correcto a la ciudad final
NIL: camino erroneo: o la ciudad final no es un destino o alguna de las ciudades obligatorias no están en el camino.

Procesamiento

Si el padre del nodo es null,
si el estado del nodo es el único mandatory
devuelve true
en caso contrario, devuelve nil
En caso contrario,
si el estado del nodo está en mandatory
hacemos la recursión con el padre del nodo y quitando su nombre de mandatory
en caso contrario,
hacemos la recursión con el nodo del padre

Código

```
;; Goal path
;;  Devuelve T or NIl dependiendo si el camino lleva a una ciudad final

(defun f-goal-path (node mandatory)
  (if (NULL (node-parent node))
     (if (NULL (remove (node-state node) mandatory))
        t
        NIL)

     (if (find (node-state node) mandatory)
        (f-goal-path (node-parent node) (remove (node-state node) mandatory))
        (f-goal-path (node-parent node) mandatory))))
```

F-goal-test

Pseudocódigo

Input:

node: nodo estructura que contiene, en la cadena de nodos padres,
un camino empezando en la ciudad inicial
destinations: lista con los nombres de las ciudades destino
mandatory: lista con los nombres de las ciudades que son obligatorias visitar

Returns

T: el camino es un camino correcto a la ciudad final
NIL: camino erroneo: o la ciudad final no es un destino o alguna de las ciudades obligatorias no estan en el camino.

Procesamiento

Si el estado del nodo está en destinations,
llamamos a la función auxiliar con su padre como argumento
En caso contrario,
devolvemos nil

Código

```
;; Goal test
;;  Devuelve T or NIl dependiendo si el camino lleva a una ciudad final, comprobando si
;; es una ciudad destino y si lleva a una ciudad final (a partir de la función goal-path)

(defun f-goal-test (node destinations mandatory)
  (if (find (node-state node) destinations)
      (f-goal-path (node-parent node) mandatory)
      NIL))
```

Ejemplos

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

NIL

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

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

CL-USER> (f-goal-test node-calais '(Calais Marseille) '(Paris Barcelona))
NIL

CL-USER> (f-goal-test node-calais '(Calais Madrid) '(Paris Barcelona))
NIL

CL-USER> (f-goal-test node-calais '(Calais Madrid) '())
T

CL-USER> (f-goal-test node-calais '() '())
NIL
```

Ejercicio 4

En este ejercicio codificaremos una función que indique si dos nodos son iguales de acuerdo con su estado de búsqueda. Para ello, utilizaremos también una fucnión auxiliar, cuyo objetivo será indicar si se han visitado todas las ciudades obligatorias.

F-search-path

Pseudocódigo

Input:
node: el nodo a estudiar
mandatory: lista con los nombres de las ciudades que es obligatorio visitar
Returns
mandatory: la lista una vez visitados todos los nodos

Procesamiento
Si el nodo padre es null
devuelve la lista de ciudades obligatorias
En caso contrario,
Si el estado del nodo está dentro de las ciudades obligatorias
hacemos la recursión con el nodo padre y quitando el estado de mandatory
Si no, hacemos la recusión con el nodo padre

Código

```
;; Si ha visitado todas las ciudades obligatorias devolverá NIL, sino devolverá las ciudades
;; obligatorias restantes

(defun f-search-path (node &optional mandatory)
  (if (NULL (node-parent node))
     mandatory
     (if (find (node-state node) mandatory)
        (f-search-path (node-parent node) (remove (node-state node) mandatory))
        (f-search-path (node-parent node) mandatory))))
```

F-search-state-equal

Pseudocódigo

Input:
node-1, node-2: los dos nodos que estamos comparando, cada uno
define un camino a través del enlace de nodos padre
mandatory: lista con los nombres de las ciudades que es obligatorio visitar
Returns
T: los dos nodos son equivalentes
NIL: los dos nodos no son equivalentes

Procesamiento

Si el nestado de los nodos es igual,
si el estado de búsqueda es igual
devolvemos true
si no, devolvemos NIL
En caso contrario, devolvemos NIL

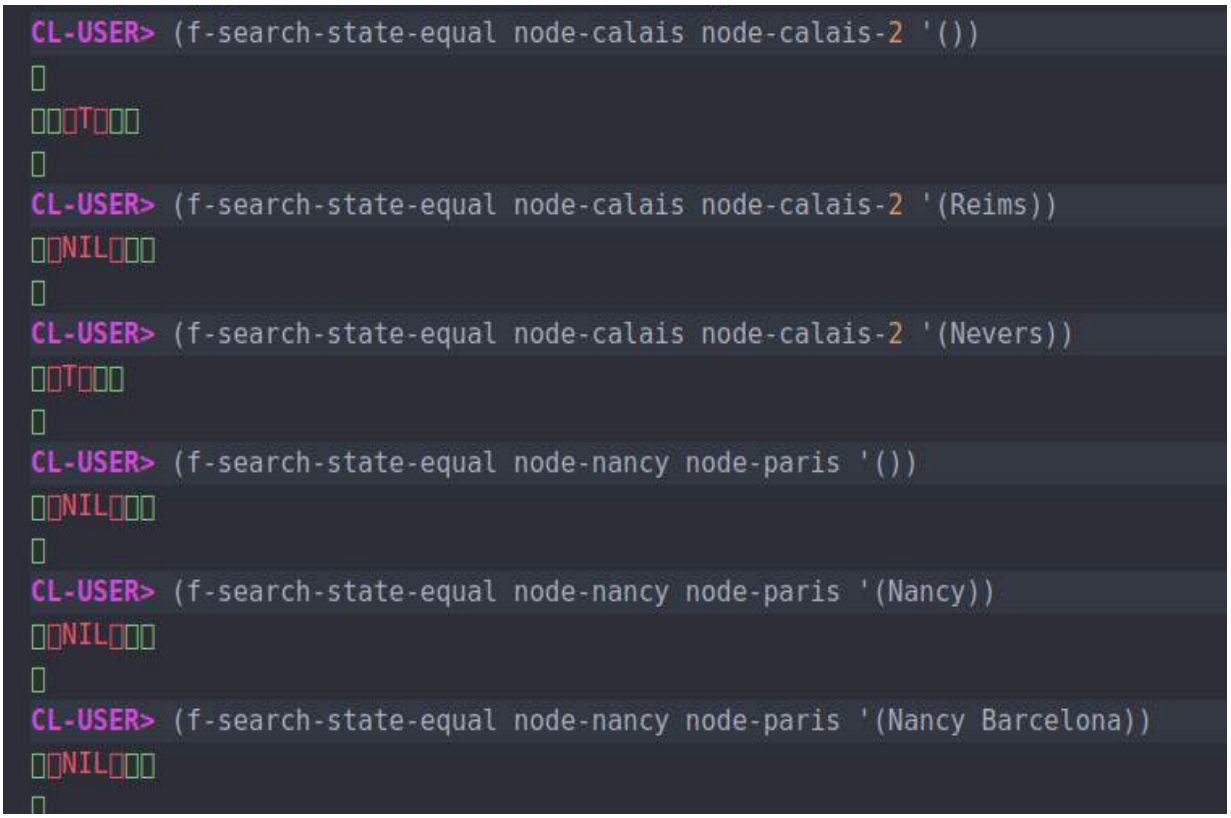
Código

```
;; F-search-state-equal

(defun f-search-state-equal (node-1 node-2 &optional mandatory)
  (if (eql (node-state node-1) (node-state node-2) )
      (if (eql (f-search-path node-1 mandatory) (f-search-path node-2 mandatory))
          t
          NIL)
      NIL))
```

Comentarios

Esta función determina si dos nodos son iguales respecto a la solución del problema: dos nodos son equivalentes si representan la misma ciudad y si el camino contiene las mismas ciudades obligatorias

Ejemplos



Ejercicio 5

En este ejercicio simplemente tendremos que inicializar dos estructuras que representen los problemas de tiempo mínimo y de coste mínimo.

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

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

Notar que la conectividad de la red usando canales y trenes es implícita en los operadores: hay una lista de dos operadores, cada uno coge un solo parámetro: el nombre de la ciudad, y devuelve una lista de acciones, indicando a que ciudades se puede mover y a coste. La lista de aristas son constantes en el segundo parámetro de los operadores de navegación

Ejercicio 6

Esta función es la encargada de expandir nodos. Dado un nodo, esta función crea una lista de nodos, cada nodo correspondiente a un estado que se puede alcanzar directamente desde el estado del nodo dado.
En este ejercicio utilizaremos dos funciones auxiliares, una que aplica a un nodo todos los operadores de un problema (expand-node-operator) y otra que genera el nodo con todos sus campos (expand-node-action) .

Expand-node-operator

Pseudocódigo

Input:

node: la node estructura desde donde empezamos.

problem: la problem estructura con la lista de operadores

Returns

Una lista (action_1,...,action_n) de acciones a los que se puede llegar desde el actual con los operadores

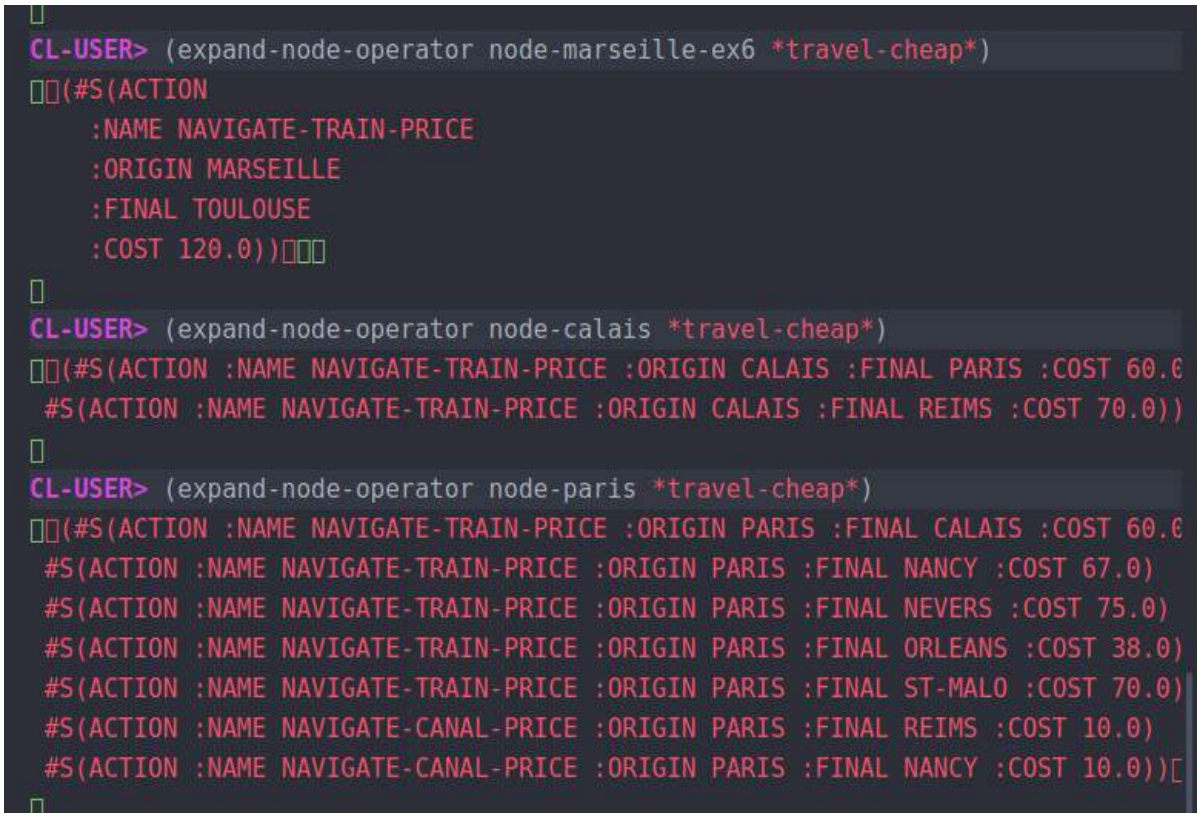
Procesamiento

Aplicamos cada uno de los operadores del problema al nodo y guardamos cada uno de las acciones en una lista.

Código

```
(defun expand-node-operator (node problem)
  (mapcan #'(lambda(x) (funcall x node)) (problem-operators problem)))
```

Ejemplos



Expand-node-action

Pseudocódigo

Input:

node: la node estructura desde donde empezamos.
action: action del nodo
f-h: funcion que nos devuelve la heuristica

Returns

Una node estructura con el nodo al que se puede llegar a partir de esa action (modificando la g y la h)

Procesamiento

Creamos una nueva estructura node, calculando la h (a partir de la funcion f-h) y el resto de datos obtenidos de la estructura action.

Código

```
(defun expand-node-action (node action f-h)
  (let ((h (funcall f-h (action-final action)))
        (g (+ (node-g node) (action-cost action))))
    (make-node :state (action-final action)
               :parent node
               :action action
               :g g
               :h h
               :f (+ h g))))
```

Expand-node

Pseudocódigo

Input:

node: la node estructura desde donde empezamos.
problem: la problem estructura con la lista de operadores

Returns

Una lista (node_1,...,node_n) de nodos a los que se puede llegar desde el actual

Procesamiento

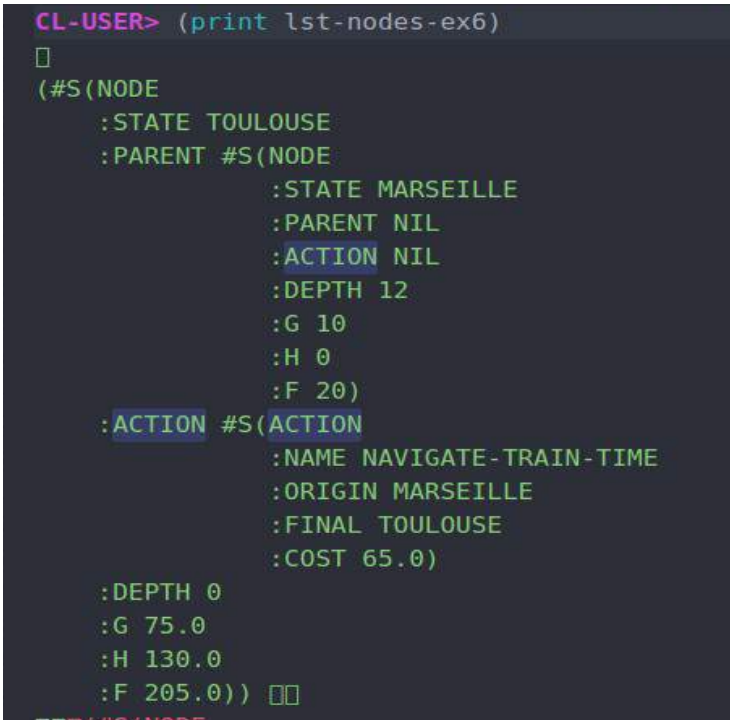
iteramos (usando mapcar) sobre todos los operadores del problema, y para cada uno de ellos, llamamos a expand-node-operator, para determinar las ciudades a las que se puede llegar usando ese operador. El operador nos devuelve una lista de actions. Iteramos de nuevo en esta lista de actions y, para cada una, llamamos a expand-node-action que crea una node estructura con el nodo al que se puede llegar a partir de esa action.

Código

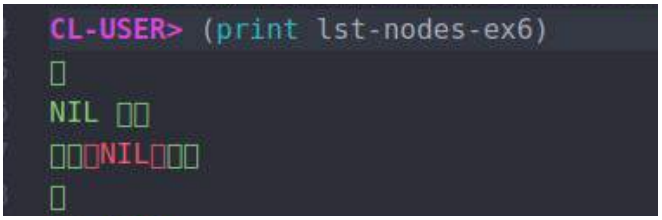
```
(defun expand-node (node problem)
```

(mapcar #'(lambda(x) (expand-node-action node x (problem-f-h problem))) (expand-node-operator node problem)))

Ejemplos



Ya que la ejecución de esta función es demasiado larga, no incluiremos más ejemplos de esta, simplemente probaremos el caso en el que el estado del nodo sea una ciudad que no esté en el mapa, por ejemplo, Madrid



Ejercicio 7

En este ejercicio implementamos una función que inserte los nodos de una lista en una lista de nodos de manera que la segunda lista esté ordenada respeto al criterio de comparación de una estrategia dada. Se supone que la lista en que se insertan los nodos ya esté ordenada respeto al criterio deseado, mientras que la lista que se inserta puede tener cualquier orden.

La funcionalidad está divida en tres funciones. La primera, insert-node,inserta un nodo en una lista manteniendo el orden. La segunda, insert-nodes, inserta los nodos de una lista no ordenada en la ordenada, uno a uno, fusionando las dos listas. La última funcion, insert-node-strategy es simplemente una interfaz recibiendo una estrategia, extrae la funcion de comparación y llama a insert-nodes.

Insert-node

Pseudocódigo

Input:

node: el nodo a insertar en la lista

lst-nodes: la lista de nodos (ordenada) en la cual los nodos dados se insertaran

node-compare-p: una funcion nodo x nodo --> 2 que devuelve T si primer nodo va antes que el segundo.

Returns

Una lista de nodos ordenados (segun node-compare-p) que incluye tanto la lista y el nodo a insertar.

Procesamiento

Si la lista de nodos esta vacía

Hemos llegado al final, por lo que devolvemos el nodo en una lista

Si no

Si el nodo satisface la funcion de comparacion con el primero

Devolvemos la lista insertando el nodo al principio

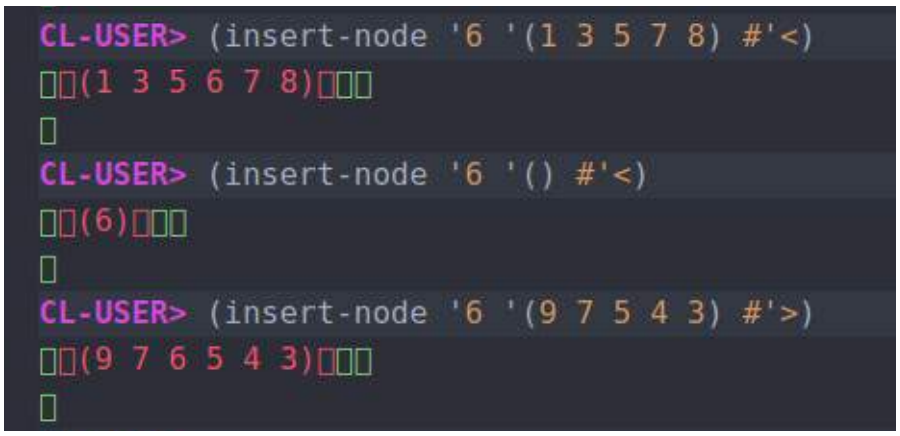
Si no:

Devolvemos una lista formada por el primer nodo de la lista y una llamada recursiva a la función pero aplicada al rest de la lista

Código

```
(defun insert-node (node lst-nodes node-compare-p)
  (if (NULL lst-nodes)
      (list node)
      (if (funcall node-compare-p node (first lst-nodes))
          (cons node lst-nodes)
          (cons (first lst-nodes) (insert-node node (rest lst-nodes) node-compare-p)))))
```

Ejemplos



Probamos algunos ejemplos con números para ver más claramente el funcionamiento

Insert-nodes

Pseudocódigo

Input:

nodes: la (possiblemente desordenada) lista de nodos a insertar en la otra lista
lst-nodes: la lista de nodos (ordenada) en la cual los nodos dados se insertaran
node-compare-p: una funcion nodo x nodo --> 2 que devuelve T si primer nodo va antes que el segundo.

Returns

Una lista de nodos ordenados (segun node-compare-p) que incluye tanto la lista y los nodos a insertar.

Procesamiento

Vamos insertando mediante recursion uno a uno cada uno de los nodos de la lista desordenada en la lista ordenada utilizando la función insert-node

Código

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

Ejemplos

```
CL-USER> (insert-nodes '(4 6 9) '(1 2 3 7 8) #'<)
(1 2 3 4 6 7 8 9)

CL-USER> (insert-nodes '(4 6 9) '(8 7 5 3 2) #'>)
(9 8 7 6 5 4 3 2)

CL-USER> (insert-nodes '() '(8 7 5 3 2) #'>)
(8 7 5 3 2)
```

Probamos algunos ejemplos con números para ver más claramente el funcionamiento

Insert-nodes-strategy

Pseudocódigo

Input:

nodes: la (possiblemente desordenada) lista de nodos a insertar en la otra lista
lst-nodes: la lista de nodos (ordenada) en la cual los nodos dados se insertaran
strategy: la estrategia para la comparación de nodos

Returns

Una lista de nodos ordenados (segun node-compare-p en strategy) que incluye tanto la lista los nodos a insertar.

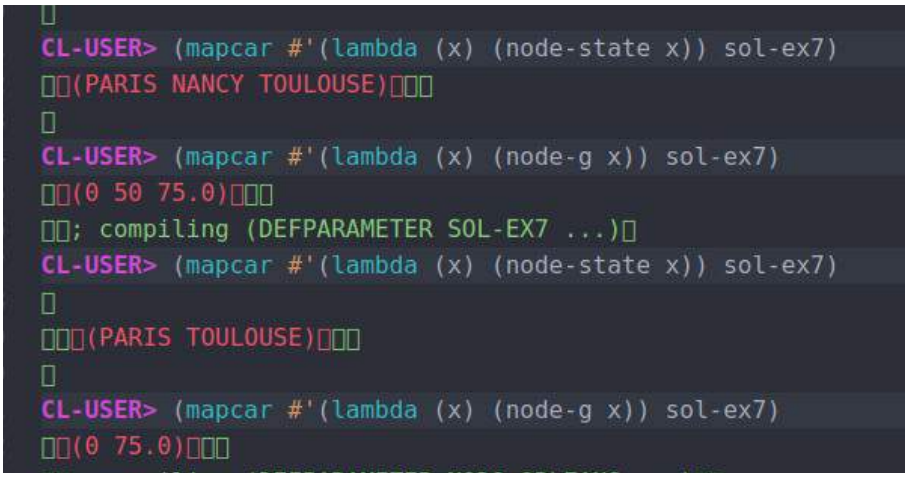
Procesamiento

El procesamiento es igual que insert-nodes pero obteniendo la función de node-compare-p de strategy.

Código

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

Ejemplos



Probamos para diferentes ejemplos, y vemos que funciona

Ejercicio 8

En este ejercicio simplemente definimos la estrategia A*. Para ello definimos una variable global, una estructura estrategia, cuya función node-compare-p también la definiremos. En A*, el primer nodo a analizar es aquel con menor f (g+h). Por ello, bastará que la función comparativa compare el valor de las f's de dos nodos.

Código

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

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

Ejercicio 9

Está función es la encargada de la búsqueda. Busca un camino que soluciona un problema dado usando una estrategia de búsqueda. Aquí tenemos tres funciones principales: la primera es una simple interfaz que extrae la informacion relevante de las problem y strategy estructura, construyendo una lista de open-nodes (que contiene unicamente) el nodo origen (identificado con el nombre de la ciudad) y una lista de closed-nodes list (una lista vacía) y llama a una función auxiliar.

La primera función auxiliar es una función recursiva que extrae nodos de la lista abierta, los expande, inserta a sus vecinos en la lista abierta y al nodo expandido . Para seguir esta versión del algoritmo necesitamos una segunda función auxilar que será la que "explore" el nodo. Para que el nodo no sea explorado se tienen que cumplir dos condiciones:

- el nodo ya esté en la lista de closed-nodes
- la estimacion de ese camino que tenemos sea mejor que la que ofrece el nodo a explorar de la lista abierta

Además utilizaremos dos funciones auxiliares más, una que a partir de una lista de nodos nos devuelve pares con el nombre de la ciudad y el valor de g. La otra, comprueba si hay algun nodo igual en una lista de nodos.

Get-state-n-g

Código

```
(defun get-state-n-g (lst-nodes)
  (mapcar #'(lambda(x) (list (node-state x) (node-g x))) lst-nodes))
```

Comentarios

A partir de mapcar y una función lambda vamos obteniendo el nombre de la ciudad y el valor de g de cada uno de los nodos de la lista de nodos.

Check-node-equal

Código

```
(defun check-node-equal (node closed-nodes problem)
  (if (NULL closed-nodes)
      nil
      (if (funcall (problem-f-search-state-equal problem) node (first closed-nodes))
          t
          (check-node-equal node (rest closed-nodes) problem))))
```

Comentarios

Funcion auxiliar que comprueba si el nodo es igual que alguno que esté en la lista de cerrados. Para ello irá comprobando uno a uno hasta que encuentre alguno (devolviendo t) o si no encuentra ninguno dara NIL.

Graph-search-aux2

Código

```
(defun graph-search-aux2 (problem open-nodes closed-nodes strategy)
  (let ((new-open-nodes (insert-nodes-strategy (expand-node (first open-nodes) problem) (rest open-nodes) strategy))
        (new-closed-nodes (cons (first open-nodes) closed-nodes)))
  (graph-search-aux problem new-open-nodes new-closed-nodes strategy)))
```

Comentarios

Funcion que explora un nodo, inserta en la lista de abiertos los nodos expandidos y añade el nodo explorado a la lista de cerrados, llamando después de nuevo a graph-search-aux.

Graph-search-aux

Pseudocódigo

Input:

problem: la problem estructura de la cual obtenemos la informacion
(goal testing function, action operatos, etc.
open-nodes: la lista de open nodes, nodos que esperan ser explorados
closed-nodes: la lista de closed nodes: nodos ya explorados
strategy: la estrategia que decide que nodo es el siguiente en ser
extraido de la lista de abiertos

Returns

NIL: no hay camino hasta los nodos destino
Si hay un camino devuelve el nodo con la ciudad final

Procesamiento

Inicializar la lista de nodos open-nodes con el estado inicial inicializar la lista de nodos closed-nodes con la lista vacía recursión:
si la lista open-nodes está vacía, terminar[no se han encontrado solución]
extraer el primer nodo de la lista open-nodes
si dicho nodo cumple el test objetivo evaluar a la solución y terminar.
en caso contrario
si el nodo considerado no está en closed-nodes o, estando en dicha lista, tiene un coste g inferior al del que está en closed-nodes
- expandir el nodo e insertar los nodos generados en la lista open-nodes de acuerdo con la estrategia strategy.
- incluir el nodo recién expandido al comienzo de la lista closed-nodes.
Continuar la búsqueda eliminando el nodo considerado de la lista open-nodes.

Código

;;Graph-search-aux

```
(defun graph-search-aux (problem open-nodes closed-nodes strategy)
  (let ((state  (node-state (first open-nodes)))
        (state-n-g (get-state-n-g closed-nodes))
        (fst-node (first open-nodes)))
  (if (NULL open-nodes)
      NIL
      (if (funcall (problem-f-goal-test problem) fst-node)
          fst-node
          (if (NULL (assoc state state-n-g))
              (graph-search-aux2 problem open-nodes closed-nodes strategy)
              (if (check-node-equal fst-node closed-nodes problem)
                  (if (< (node-g fst-node) (second (assoc state state-n-g)))
                      (graph-search-aux2 problem open-nodes closed-nodes strategy)
                      (graph-search-aux problem (rest open-nodes) closed-nodes strategy))
                  (graph-search-aux2 problem open-nodes closed-nodes strategy)))))))
```

Comentarios

Lo que devolvemos es una estructura compleja: el nodo contiene en "parent" el nodo anterior en el camino, que contiene otro en "parent", y así hasta el nodo inicial.Por lo que devolvemos una compleja estructura anidada que contiene no solo el nodo final sino el camino completo.

Graph-search

Pseudocódigo

Input:

problem: la problem estructura de la cual obtenemos la informacion
(goal testing function, action operatos, etc.
strategy: la estrategia que decide que nodo es el siguiente en ser
extraido de la lista de abiertos

Returns

NIL: no hay camino hasta los nodos destino
Si hay un camino devuelve el nodo con la ciudad final

Procesamiento

Llama a graph-search-aux creando un nodo con la ciudad origen como estado y con los argumentos pasados como parámetros

Código

```
;;Graph-search

(defun graph-search (problem strategy)
  (let ((h (funcall (problem-f-h problem) (problem-initial-state problem))))
  (graph-search-aux problem (list (make-node :state (problem-initial-state problem)
                                  :parent NIL
                                  :action NIL
```

```
                    :g 0
                    :h h
                    :f h))
        '() strategy)))
```

Comentarios

Esta función simplemente crea una open-nodes con un unico nodo (el source) y closed-nodes como una lista vacía.

A-star-search

El código de esta sencilla función es el siguiente

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

Esta función simplemente resuelve un problema utilzando el algoritmo A-estrella

Ejemplos

Probamos primero el ejemplo por defecto, es decir, de Marseille a Calais, sin pasar por Avignon y pasando por Paris

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 145.0
                                :F 145.0)
                          :ACTION #S(ACTION
                                :NAME NAVIGATE-TRAIN-TIME
                                :ORIGIN MARSEILLE
                                :FINAL TOULOUSE
                                :COST 65.0)
                          :DEPTH 0
                          :G 65.0
                          :H 130.0
                          :F 195.0)
                    :ACTION #S(ACTION
                          :NAME NAVIGATE-TRAIN-TIME
                          :ORIGIN TOULOUSE
                          :FINAL LIMOGES
                          :COST 25.0)
                    :DEPTH 0
                    :G 90.0
                    :H 100.0
                    :F 190.0)
              :ACTION #S(ACTION
                    :NAME NAVIGATE-TRAIN-TIME
```

```
                    :ORIGIN LIMOGES
                    :FINAL ORLEANS
                    :COST 55.0)
                :DEPTH 0
                :G 145.0
                :H 55.0
                :F 200.0)
        :ACTION #S(ACTION
                :NAME NAVIGATE-TRAIN-TIME
                :ORIGIN ORLEANS
                :FINAL PARIS
                :COST 23.0)
        :DEPTH 0
        :G 168.0
        :H 30.0
        :F 198.0)
:ACTION #S(ACTION
        :NAME NAVIGATE-TRAIN-TIME
        :ORIGIN PARIS
        :FINAL CALAIS
        :COST 34.0)
:DEPTH 0
:G 202.0
:H 0.0
:F 202.0)###
```

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 0.0
                                                :F 0.0)
                                        :ACTION #S(ACTION
                                                :NAME NAVIGATE-TRAIN-PRICE
                                                :ORIGIN MARSEILLE
                                                :FINAL TOULOUSE
                                                :COST 120.0)
                                        :DEPTH 0
                                        :G 120.0
                                        :H 0.0
                                        :F 120.0)
                                :ACTION #S(ACTION
                                        :NAME NAVIGATE-TRAIN-PRICE
                                        :ORIGIN TOULOUSE
                                        :FINAL LIMOGES
                                        :COST 35.0)
                                :DEPTH 0
                                :G 155.0
                                :H 0.0
                                :F 155.0)
                        :ACTION #S(ACTION
                                :NAME NAVIGATE-TRAIN-PRICE
                                :ORIGIN LIMOGES
                                :FINAL NEVERS
                                :COST 60.0)
                        :DEPTH 0
                        :G 215.0
                        :H 0.0
                        :F 215.0)
                :ACTION #S(ACTION
                        :NAME NAVIGATE-CANAL-PRICE
```

```
                    :ORIGIN NEVERS
                    :FINAL PARIS
                    :COST 10.0)
                :DEPTH 0
                :G 225.0
                :H 0.0
                :F 225.0)
        :ACTION #S(ACTION
                :NAME NAVIGATE-CANAL-PRICE
                :ORIGIN PARIS
                :FINAL REIMS
                :COST 10.0)
        :DEPTH 0
        :G 235.0
        :H 0.0
        :F 235.0)
  :ACTION #S(ACTION
        :NAME NAVIGATE-CANAL-PRICE
        :ORIGIN REIMS
        :FINAL CALAIS
        :COST 15.0)
  :DEPTH 0
  :G 250.0
  :H 0.0
  :F 250.0)###
```

Añadimos un ejemplo más, que será de Brest a Avignon, sin pasar por Nantes y pasando por Limoges.

travel-fast

```
#S(NODE
  :STATE AVIGNON
  :PARENT #S(NODE
        :STATE MARSEILLE
        :PARENT #S(NODE
                :STATE TOULOUSE
                :PARENT #S(NODE
                        :STATE LIMOGES
                        :PARENT #S(NODE
                                :STATE ORLEANS
                                :PARENT #S(NODE
                                        :STATE PARIS
                                        :PARENT #S(NODE
                                                :STATE ST-MALO
                                                :PARENT #S(NODE
                                                        :STATE BREST
                                                        :PARENT NIL
                                                        :ACTION NIL
                                                        :DEPTH 0
                                                        :G 0
                                                        :H 90.0
                                                        :F 90.0)
                                                :ACTION #S(ACTION
                                                        :NAME NAVIGATE-TRAIN-TIME
                                                        :ORIGIN BREST
                                                        :FINAL ST-MALO
                                                        :COST 30.0)
                                                :DEPTH 0
                                                :G 30.0
                                                :H 65.0
                                                :F 95.0)
                                        :ACTION #S(ACTION
                                                :NAME NAVIGATE-TRAIN-TIME
                                                :ORIGIN ST-MALO
                                                :FINAL PARIS
                                                :COST 40.0)
                                        :DEPTH 0
                                        :G 70.0
                                        :H 30.0
                                        :F 100.0)
                                :ACTION #S(ACTION
                                        :NAME NAVIGATE-TRAIN-TIME
                                        :ORIGIN PARIS
                                        :FINAL ORLEANS
                                        :COST 23.0)
```

```
                    :DEPTH 0
                    :G 93.0
                    :H 55.0
                    :F 148.0)
                :ACTION #S(ACTION
                    :NAME NAVIGATE-TRAIN-TIME
                    :ORIGIN ORLEANS
                    :FINAL LIMOGES
                    :COST 55.0)
                :DEPTH 0
                :G 148.0
                :H 100.0
                :F 248.0)
            :ACTION #S(ACTION
                :NAME NAVIGATE-TRAIN-TIME
                :ORIGIN LIMOGES
                :FINAL TOULOUSE
                :COST 25.0)
            :DEPTH 0
            :G 173.0
            :H 130.0
            :F 303.0)
        :ACTION #S(ACTION
            :NAME NAVIGATE-TRAIN-TIME
            :ORIGIN TOULOUSE
            :FINAL MARSEILLE
            :COST 65.0)
        :DEPTH 0
        :G 238.0
        :H 145.0
        :F 383.0)
  :ACTION #S(ACTION
        :NAME NAVIGATE-TRAIN-TIME
        :ORIGIN MARSEILLE
        :FINAL AVIGNON
        :COST 16.0)
  :DEPTH 0
  :G 254.0
  :H 135.0
  :F 389.0)###
```

travel-cheap

```
###S(NODE
  :STATE AVIGNON
  :PARENT #S(NODE
        :STATE LYON
        :PARENT #S(NODE
            :STATE TOULOUSE
            :PARENT #S(NODE
                :STATE LIMOGES
                :PARENT #S(NODE
                    :STATE ORLEANS
                    :PARENT #S(NODE
                        :STATE PARIS
                        :PARENT #S(NODE
                            :STATE ST-MALO
                            :PARENT #S(NODE
                                :STATE BREST
                                :PARENT NIL
                                :ACTION NIL
                                :DEPTH 0
                                :G 0
                                :H 0.0
                                :F 0.0)
                            :ACTION #S(ACTION
                                :NAME NAVIGATE-TRAIN-PRICE
                                :ORIGIN BREST
                                :FINAL ST-MALO
                                :COST 40.0)
                            :DEPTH 0
                            :G 40.0
                            :H 0.0
                            :F 40.0)
                        :ACTION #S(ACTION
                            :NAME NAVIGATE-TRAIN-PRICE
                            :ORIGIN ST-MALO
                            :FINAL PARIS
```

```
                                        :COST 70.0)
                                    :DEPTH 0
                                    :G 110.0
                                    :H 0.0
                                    :F 110.0)
                                :ACTION #S(ACTION
                                    :NAME NAVIGATE-TRAIN-PRICE
                                    :ORIGIN PARIS
                                    :FINAL ORLEANS
                                    :COST 38.0)
                                :DEPTH 0
                                :G 148.0
                                :H 0.0
                                :F 148.0)
                            :ACTION #S(ACTION
                                :NAME NAVIGATE-TRAIN-PRICE
                                :ORIGIN ORLEANS
                                :FINAL LIMOGES
                                :COST 85.0)
                            :DEPTH 0
                            :G 233.0
                            :H 0.0
                            :F 233.0)
                        :ACTION #S(ACTION
                            :NAME NAVIGATE-TRAIN-PRICE
                            :ORIGIN LIMOGES
                            :FINAL TOULOUSE
                            :COST 35.0)
                        :DEPTH 0
                        :G 268.0
                        :H 0.0
                        :F 268.0)
                    :ACTION #S(ACTION
                        :NAME NAVIGATE-TRAIN-PRICE
                        :ORIGIN TOULOUSE
                        :FINAL LYON
                        :COST 95.0)
                    :DEPTH 0
                    :G 363.0
                    :H 0.0
                    :F 363.0)
:ACTION #S(ACTION
                    :NAME NAVIGATE-CANAL-PRICE
                    :ORIGIN LYON
                    :FINAL AVIGNON
                    :COST 20.0)
:DEPTH 0
:G 383.0
:H 0.0
:F 383.0)###
```

Ejercicio 10

Este ejercicio es simplemente una continuación del anterior, ya que se pide que se muestren solamente los nombres de los nodos que se han visitado para llegar a un destino.

Solution-path

Pseudocódigo

Input:

node: nodo del cual obtener el camino

Returns

El camino explorado

Procesamiento

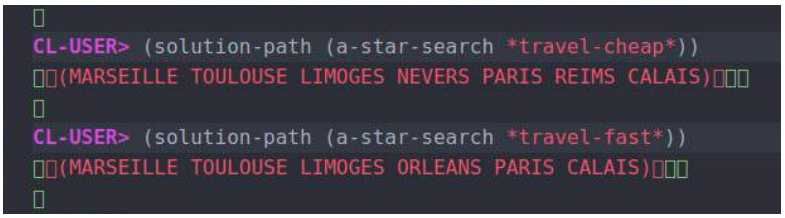
Si el nodo es null, devolvemos nil
En caso contrario,
devolvemos en una lista el nombre del nodo con el resultado de evaluar la función en el nodo padre

Código

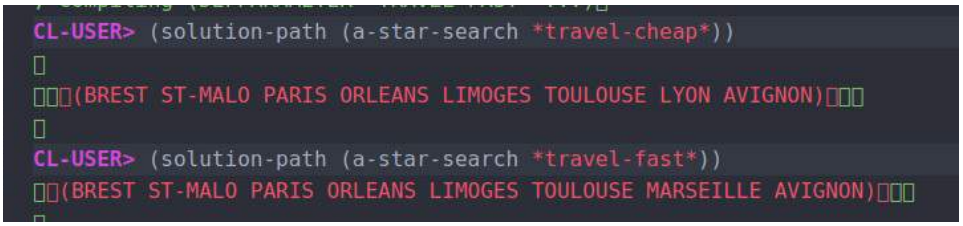
```
;;Solution-path

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

Ejemplos

Probamos los mismos ejemplos que en el apartado anterior. Primero el camino por defecto (Marseille-Calais)



Probamos el siguiente ejemplo (Brest-Avignon)



Action-sequence

Pseudocódigo

Input:

node: nodo del cual obtener las acciones del camino

Returns

Las acciones del nodo explorado

Procesamiento

Si el nodo es null, devolvemos nil
En caso contrario,
devolvemos en una lista el nombre del nodo con el resultado de evaluar la función en el nodo padre

Código

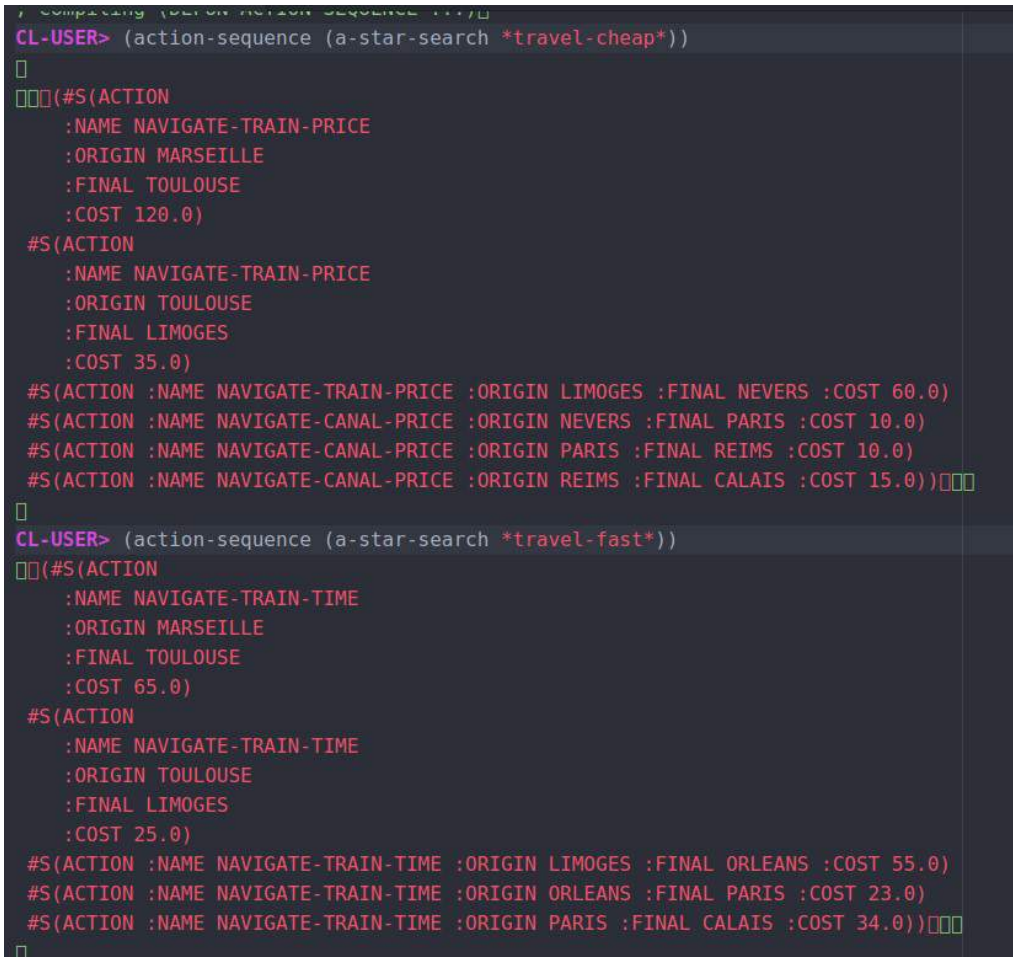
```
;;Action-sequence

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

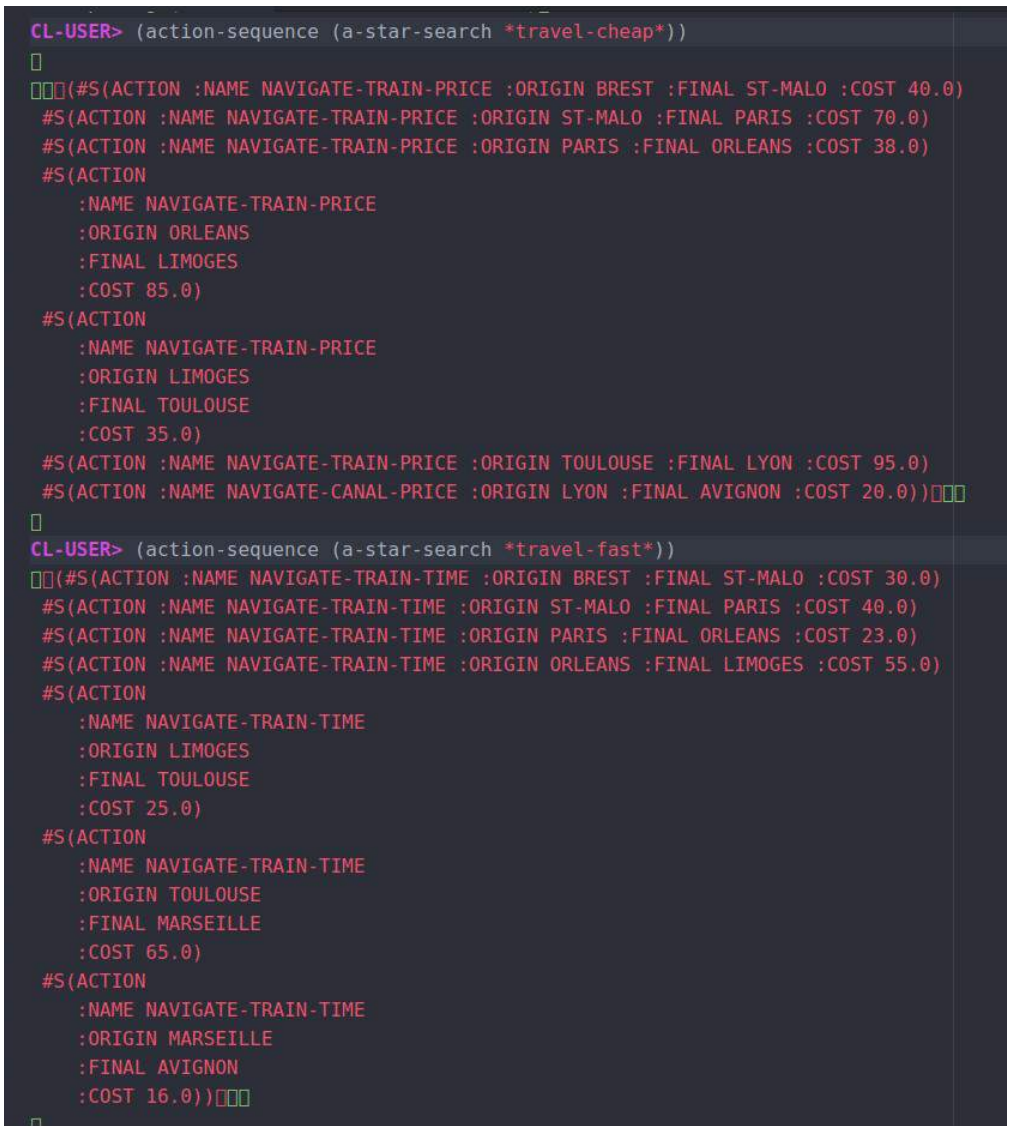
Comentarios

Esta función es igual que la anterior, solo que en vez de devolver el nombre de las ciudades, devuelve todas las acciones realizadas.

Ejemplos

Probamos los mismos ejemplos que en los casos anteriores





Ejercicio 11

En este ejercicio se nos pide diseñar una estrategia para la búsqueda en profundidad y otra para la búsqueda en anchura.

Depth-first-search

Para realizar la búsqueda en profundidad los nodos recien explorados tienen mayor prioridad, por los que serán añadidos los primeros a la lista de abiertos, por lo que con poner un TRUE valdrá y añadirá el nodo el primero a la lista. De esta manera no expandiremos el árbol de izquierda a derecha sino de derecha a izquierda (siendo una manera igualmente válida)

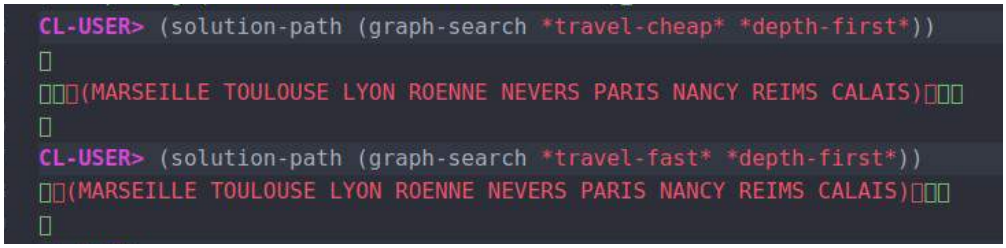
```
(defun depth-first-node-compare-p (node-1 node-2)
  t)
```

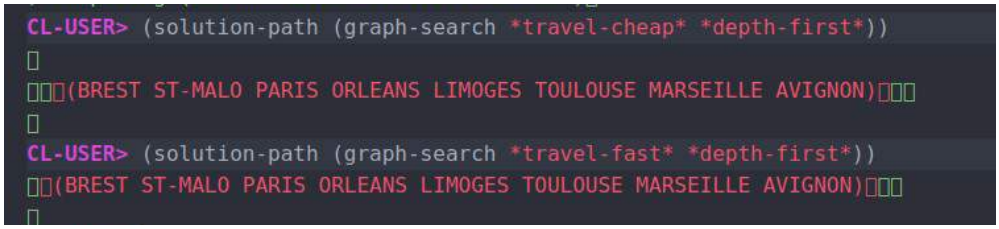
Definimos después la estrategia de búsqueda en profundidad

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

Ejemplos

Probamos los mismo ejemplos que anteriormente, aunque haremos uso de la función solution-path para ver directamente el camino





Observamos como los caminos cambian, ya que la estrategia de búsqueda es distinta

<u>Breath-first-search</u>

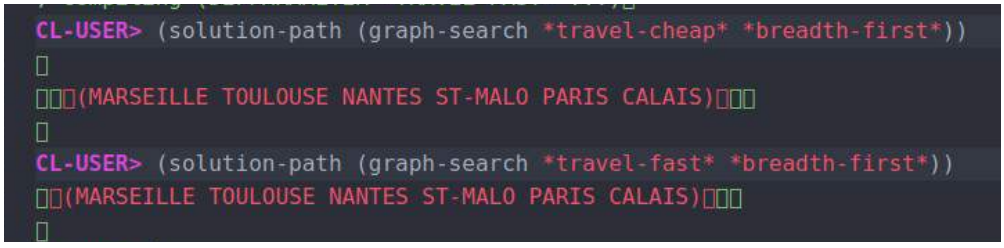
Para realizar la búsqueda en anchura los nodos recien explorados serán los últimos añadidos a la lista de abiertos,por lo que con poner un NIL valdrá y añadirá el nodo el último de la lista.

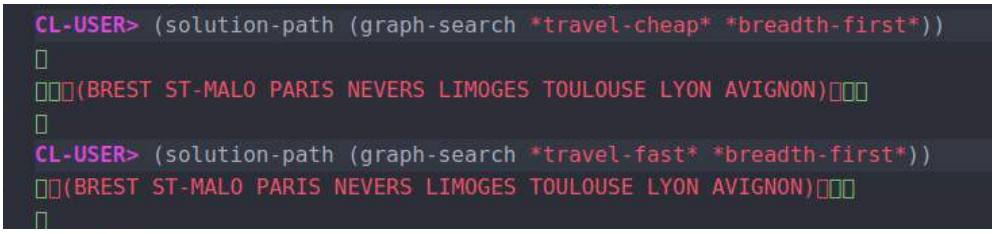
```
(defun breadth-first-node-compare-p (node-1 node-2)
  NIL)
```

Definimos la estrategia de búsqueda en anchura

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

Ejemplos
Hacemos lo mismo que en el caso anterior





Al igual que en el caso anterior observamos como los caminos cambian, ya que la estrategia de búsqueda es distinta

Ejercicio 12

En este ejercicio tenemos que implementar una heurística para el coste, algo menos conservadora que el coste sea 0 para todas las ciudades. Para ello hemos procedido de la siguiente manera: empezando por las ciudades con enlace directo a Calais (Reims en este caso), hemos obtenido el menor coste desde ella y lo hemos dividido entre 3 (para obtener una heurística que subestime el coste real en cualquier caso). Una vez hecho para estas ciudades, hemos hecho lo mismo para las respectivas ciudades con enlace directo a estas, sumándole el resultado anteriormente calculado. Ejecutando este sencillo algoritmo de manera recursiva hasta llegar a Marseille hemos llegado a la siguiente tabla de heurísticas:

```
(defparameter *estimate-new*
  '((Calais (0.0 0.0)) (Reims (25.0 5.0)) (Paris (30.0 8.333))
    (Nancy (50.0 11.666)) (Orleans (55.0 21.0)) (St-Malo (65.0 31.666))
    (Nantes (75.0 39.333)) (Brest (90.0 45.0)) (Nevers (70.0 15.0))
    (Limoges (100.0 35.0)) (Roenne (85.0 16.666)) (Lyon (105.0 18.333))
    (Toulouse (130.0 46.666)) (Avignon (135.0 31.666)) (Marseille (145.0 40.0))))
```

Una vez obtenida esta nueva heurística, tenemos que implementarla en un nuevo problema, al que llamaremos *travel-cheap-new*:

```
(defparameter *travel-cheap-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-train-price (node-state node) *trains* *forbidden*))
            #'(lambda (node) (navigate-canal-price (node-state node) *canals*)))))
```

Una vez implementada la nueva heurística y el nuevo problema lo único que tendremos que hacer será comparar la antigua heurística (la de coste 0) con la nueva:

Heurística antigua:

Heurística nueva:

Como podemos observar la nueva heúristica tarda un orden de magnitud menos que la anterior, mientras que en bytes utilizados son dos órdenes de magnitud, por lo que podemos concluir que la nueva heurística es mas eficiente temporal y espacialmente.

Preguntas

1.2.

Para resolver el problema de búsqueda el diseño empleado ha sido el algoritmo A estrella. Hemos utilizado este algoritmo ya que es completo, es decir, en caso de existir la solución, siempre dará con ella. Su optimización depende de la heurística empleada, siendo óptimo si ésta es monótona. En nuestro caso concreto, la heurística por defecto para estimar el coste era 0, la cual es admisible, por lo que el camino encontrado siempre es el óptimo, pero al ser esta la heurística más conservadora, la búsqueda no era lo más eficiente posible.

La ventaja que nos ofrece este diseño es la posibilidad de realizar la búsqueda de forma más eficiente, lo cual hemos realizado cambiando la heurística, a otra también admisible (como es obvio) pero algo menos conservadora. Hemos observado en el ejercicio 12 que en efecto los tiempos de ejecución y el uso de memoria mejoraban significativamente.

A nivel de diseño,la razón de la utilización de funciones lambda para especificar el test objetivo, la heurística y los operadores del problema es debido a que, con esta implementación, si modificamos los ficheros sobre los que se ejecuta el programa (es decir, el número de trenes, canales, sus nombres, las heurísticas...) el programa seguirá funcionando correctamente, sin necesidad de agregar más cambios.

Además para que sea más óptima hemos añadido la eliminación de estados repetidos, mediante la implentación de la función f-search-state-equal, la cual determina que dos nodos son iguales si representan la misma ciudad y el camino contiene las mismas ciudades obligatorias.

3. Es obvio que el uso de la memoria no es el más eficiente posible, ya que se gasta mucho espacio en almacenar todo el nodo padre. Pero también es cierto que es necesario tener una referencia a dicho nodo, aunque a lo mejor, para no gastar tanto espacio, en vez de guardar todo el nodo se podrían guardar solo algunos campos importantes de este, como el nombre y las acciones, ya que el resto de valores del nodo no son importantes para el hijo.

4. El mayor problema de A estrella es la cantidad de espacio que necesita para ser ejecutado. Ya que necesita almacenar todos los posibles siguientes nodos desde cada estado, la cantidad de memoria que requerirá será exponencial con respecto a la magnitud del problema.

5. La complejidad temporal del algoritmo A estrella depende directamente de la calidad de la heurística empleada. En el peor de los casos, la complejidad será exponencial, mientras que con una heurística buena, en el mejor de los casos, el algoritmo se ejecutará en tiempo lineal.