Javier Delgado del Cerro y Javier López Cano - Pareja 2

Práctica 2

Incluimos el código de todos los ejercicios, eliminando las cabeceras de las funciones, al final del documento, para agilizar así la lectura de los comentarios de todos los ejercicios.

Ejercicio 1

Batería de ejemplos

Comentar antes de empezar este apartado que no incluiremos en ninguna de estas baterías de ejemplos los ejemplos dados en la documentación de la práctica o en el fichero de pruebas proporcionado, pues ya hemos comprobado su correcto funcionamiento en estos casos, y consideramos que sería redundante.

En este caso específico, podemos ver que casi todos los casos conflictivos han sido pedidos en el enunciado, con lo que quedaría simplemente comprobar las expresiones en las que una o ambas de las entradas es NIL

```
(f-h-price NIL NIL) ;;; ---> NIL
(f-h-price NIL *estimate*) ;;; ---> NIL
(f-h-price 'Nantes NIL) ;;; ---> NIL
```

Y, efectivamente, vemos que funciona de la manera esperada.

Pseudo-código

A pesar de tener que elaborar dos funciones distintas, el pseudo-código es prácticamente el mismo para ambas, pues se basa en recorrer recursivamente la lista *sensors*, por tanto, mostramos únicamente el pseudo-código de una de ellas:

```
f-h-time (state, sensors):
    si vacio(sensors):
        devolver false
    si primer_elemento(sensors).state == state:
        devolver primer_elemento(sensors).time
    si no:
        f-h-time (state, resto(sensors))
```

Comentarios sobre la implementación

En este caso, el apartado no ha requerido tomar ninguna decisión especial sobre la implementación.

Ejercicio 2

Batería de ejemplos

Como en el apartado anterior, dado que las funciones pedidas tienen un funcionamiento bastante sencillo, la batería de pruebas es muy reducida. Con los ejemplos proporcionados comprobamos que el funcionamiento es correcto en los casos en que el origen existe en el grafo, tanto si tiene algún posible destino como si no (como es el caso de Orleans), y teniendo en cuenta también la lista de ciudades prohibidas, como en el

ejemplo `(navigate-train-price 'Avignon *trains* '(Marseille))`.

Queda por tanto probar únicamente los casos en los que uno o ambos de los campos de entrada sea NIL. Como las cuatro funciones se basan en una principal que es la que realiza la búsqueda (explicado en el apartado siguiente), basta probarlo en una de las cuatro funciones pedidas:

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

Comprobamos que, efectivamente, funcionan de la manera esperada.

Comentarios sobre la implementación

Para la implementación de este apartado hemos decidido definir, como se mencionaba en el enunciado, una función genérica *navigate* que, dado un estado, una lista de los vértices del grafo, una función para extraer el coste deseado del vértice y un nombre para dicha acción, junto con una posible lista de ciudades prohibidas, permite obtener una lista de estados a los que podemos llegar aplicando una acción.

De esta forma, las otras cuatro funciones son llamadas a esta primera variando la lista de vértices según sea por tren o por canal, la función para seleccionar el coste (que consiste básicamente en coger el primer o segundo elemento de una tupla), y el nombre de la acción, además de incluir o no la lista de prohibidas

Pseudo-código

Mostramos entonces únicamente el pseudo-código de la función *navigate*, pues las otras cuatro son extremadamente simples y se han mencionado en el apartado anterior.

```
navigate (state, lst-edges, cfun, name, forbidden):
    si vacia(lst-edges):
        devolver NIL
    si (primer_elemento(sensors).origen == state) y
        (primer_elemento(sensors).destino no esta en fobidden):
        devolver crear_accion(primer_elemento(sensors), state, name, cfun)
            concatenado con navigate(state, resto(lst-edges), cfun, name, forbidden)

    si no:
        navigate (state, resto(lst-edges), cfun, name, forbidden)
```

La función *crear_accion(primer_elemento(sensors))* correspondería simplemente a inicializar una acción como:

```
crear_accion(elem, state, name, cfun):
    accion.name = name
    accion.state = state
    accion.final = elem.destino
    accion.coste = cfun(elem.costes)
    devolver accion
```

Ejercicio 3

Batería de ejemplos

En este caso, la batería de ejemplos consiste en probar los casos en que:

- El nodo no está en la lista de destinos.
- El nodo está en la lista de destinos, pero el trayecto no incluye todas las ciudades obligatorias.
- El nodo está en la lista de destinos y el trayecto incluye todas las ciudades obligatorias.
- Alguno o varios de los parámetros son NIL.

Los tres primeros casos están probados en el fichero de pruebas proporcionado, por lo que solo quedaría probar las entradas conflictivas en los que uno o varios de los argumentos son NIL:

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

Comentarios sobre la implementación

Para la implementación de este ejercicio hemos decidido crear dos funciones auxiliares:

- Una primera función *navigate-path*, que devuelve una lista desde el elemento raíz hasta el nodo pasado como parámetro. Esta función se utilizará en apartados posteriores como en el 10. Para poder devolver el camino recorrido en orden, desde el nodo raíz hasta el final, usamos una función auxiliar *navigate-path-aux* que se encarga de hacer la recursión.
- Una función *check-mandatory* que, dado una lista que represente el path, generada con la función anterior, y una lista de ciudades obligatorias, comprueba si todas las ciudades obligatorias están en el path recorrido.

Por último, la función *f-goal-test* se encarga únicamente de comprobar que el nodo pasado como argumento es uno de los destinos, y de llamar a *check-mandatory* para asegurar que se han recorrido todas las ciudades obligatorias.

Pseudo-código

El pseudo-código para estas funciones sería entonces:

```
navigate-path-aux(node, path):
    si node no tiene padre:
        devolver concatenar(node-nombre, path)
    si no:
        devolver navigate-path-aux(node.padre, concatenar(node-nombre, path))

navigate-path(node):
    si node es null:
        devolver NIL
    si no:
        devolver navigate-path-aux(node, lista())

check-mandatory (path, mandatory):
    si vacio(mandatory):
        devolver true
    si primer_elemento(mandatory) no esta en path:
        devolver NIL
    si no:
```

```
        devolver check-mandatory (path, resto(mandatory))

f-goal-test (node, destination, mandatory):
    si node es null:
        devolver NIL
    si node.nombre no esta en destination:
        devolver NIL
    si no:
        devolver check-mandatory (navigate-path(node), mandatory)
```

Ejercicio 4

Batería de ejemplos

El comprobar que dos nodos corresponden a la misma ciudad es elemental, pues basta con comparar el campo *state* de ambos nodos. Lo complicado es, sin embargo, comprobar que los caminos son equivalentes, es decir, que tienen las mismas ciudades obligatorias. Sin embargo, la corrección de esta característica se comprueba simplemente con los ejemplos dados en el documento de pruebas. Además, en esta función no merece la pena controlar si uno o ambos de los nodos son NIL, pues al ser la función llamada cada vez que expandimos un nodo, sería muy ineficiente. Controlamos desde el resto de funciones que nunca haya un NIL en la lista de nodos abiertos.

Por tanto, la batería de ejemplos sería básicamente la proporcionada.

Comentarios sobre la implementación

Para la resolución del apartado únicamente cabe destacar que hemos desarrollado una función auxiliar, *f-equivalent-paths* que es la encargada de comprobar si los caminos son o no equivalentes. Estos caminos se obtienen mediante la función *navigate-path(node)* desarrollada en el ejercicio anterior.

Pseudo-código

El pseudo-código para estas funciones sería entonces:

```
f-equivalent-paths(path-1, path-2, mandatory):
    si vacio(mandatory):
        devolver True
    si primer_elemento(mandatory) no esta en path-1 pero si en path-2:
        devolver NIL
    si primer_elemento(mandatory) no esta en path-2 pero si en path-1:
        devolver NIL
    si no:
        devolver f-equivalent-paths(path-1, path-2, resto(mandatory))

f-search-state-equal (node-1, node-2, mandatory):
    si node-1.nombre != node-2.nombre:
        devolver NIL
    si no:
        devolver f-equivalent-paths(navigate-path(path-1), navigate-path(path-2),
mandatory)
```

Ejercicio 5

En este caso, como simplemente hemos tenido que definir las estructuras haciendo uso de todas las funciones desarrolladas anteriormente, por lo que no hay ninguna batería de pruebas o comentarios sobre la implementación que podamos hacer. La comprobación de las estructuras definidas se hace en el siguiente ejercicio.

Ejercicio 6

Batería de ejemplos

Con el ejemplo dado, comprobamos que, la función obtiene únicamente Toulouse mediante la acción *NAVIGATE-TRAIN-TIME*, y que no expande Avignon, pues está en la lista de ciudades prohibidas. Además, no tenemos ningún destino mediante las acciones de los canales, pues Marseille no tiene ninguna salida mediante estos.

Quedaría por tanto comprobar únicamente la estructura *travel-cheap*, que debería devolver lo mismo, variando únicamente los valores de g, f y h. Probamos por último con un nodo que tenga destinos mediante canales y mediante trenes, como puede ser *Reims*.

```
(defparameter node-marseille-ex6
   (make-node :state 'Marseille :depth 12 :g 10 :f 20) )
(expand-node node-marseille-ex6 *travel-cheap*) ;; --->
;(#S(NODE :STATE TOULOUSE
;         :PARENT #S(NODE
;                    :STATE MARSEILLE
;                    :PARENT NIL
;                    :ACTION NIL
;                    :DEPTH 12
;                    :G 10
;                    :H 0
;                    :F 20)
;         :ACTION #S(ACTION
;                    :NAME NAVIGATE-TRAIN-PRICE
;                    :ORIGIN MARSEILLE
;                    :FINAL TOULOUSE
;                    :COST 65.0)
;         :DEPTH 13
;         :G 130.0
;         :H 130.0
;         :F 260.0))

(defparameter node-reims-ex6
   (make-node :state 'Reims :depth 12 :g 10 :f 20) )
(expand-node node-reims-ex6 *travel-cheap*) ;; --->
;(#S(NODE :STATE CALAIS
;         :PARENT #S(NODE
;                    :STATE REIMS
;                    :PARENT NIL
;                    :ACTION NIL
;                    :DEPTH 12
;                    :G 10
;                    :H 0
;                    :F 20)
;         :ACTION #S(ACTION
```

```
;                    :NAME NAVIGATE-CANAL-PRICE
;                    :ORIGIN REIMS
;                    :FINAL CALAIS
;                    :COST 15.0)
;          :DEPTH 13
;          :G 25.0
;          :H 0.0
;          :F 25.0)
;#S(NODE :STATE CALAIS
;          :PARENT #S(NODE
;                    :STATE REIMS
;                    :PARENT NIL
;                    :ACTION NIL
;                    :DEPTH 12
;                    :G 10
;                    :H 0
;                    :F 20)
;          :ACTION #S(ACTION
;                    :NAME NAVIGATE-TRAIN-PRICE
;                    :ORIGIN REIMS
;                    :FINAL CALAIS
;                    :COST 70.0)
;          :DEPTH 13
;          :G 80.0
;          :H 0.0
;          :F 80.0)
;#S(NODE :STATE NANCY
;          :PARENT #S(NODE
;                    :STATE REIMS
;                    :PARENT NIL
;                    :ACTION NIL
;                    :DEPTH 12
;                    :G 10
;                    :H 0
;                    :F 20)
;          :ACTION #S(ACTION
;                    :NAME NAVIGATE-TRAIN-PRICE
;                    :ORIGIN REIMS
;                    :FINAL NANCY
;                    :COST 55.0)
;          :DEPTH 13
;          :G 65.0
;          :H 50.0
;          :F 115.0))
```

Podemos ver entonces que funciona correctamente. No hemos puesto el ejemplo `(expand-node node-reims-ex6 *travel-fast*)` pues los resultados son prácticamente los mismos, cambiando únicamente los nombres de las acciones y sus costes, sin embargo, sí que lo hemos probado.

En este ejercicio, no tiene sentido probar los casos en los que el nodo o el problema son NIL, pues esto solo puede pasar por un fallo en el programa, el algoritmo no debería continuar en dicho caso.

Comentarios sobre la implementación

Hemos definido una función auxiliar, *expand-node-action*, que permite obtener el nodo resultante de aplicar una acción a un nodo padre. De esta forma, la función *expand-node* se encarga de iterar esta función auxiliar sobre las acciones generadas a partir del nodo dado y de los problem-operators de problema pasado como argumento.

Pseudo-código

El pseudo-código de estas funciones sería entonces:

```
 expand-node-action (action, parent, problem):
    crear_nodo(state=action.final,
        parent=parent,
        action=action,
        depth=parent.depth + 1,
        g=parent.gaction.cost,
        h=problem.f-h(action.final),
        f=parent.gaction.cost + problem.f-h(action.final)

expand-node(node, problem):
    action-lst = aplicar cada operador en problem.operator a node
    devolver (expand-node-action(act, node, problem) para cada act en action-lst)
```

Ejercicio 7

Batería de ejemplos

De nuevo, basta con comprobar que el algoritmo usado ordena los elementos dentro de una lista, para lo cual es suficiente el ejemplo dado en el fichero de pruebas, con el que hemos comprobado que sí que funciona correctamente.

Comentarios sobre la implementación

En este caso, hemos seguido la implementación sugerida en el enunciado, de forma que desarrollamos tres funciones. Una primera, *insert-node*, que utiliza la recursión para insertar un nodo en una lista ordenada según una función *node-compare-p*. La segunda función, *insert-nodes*, que permite insertar todos los nodos de una lista en otra lista, de forma que la lista destino está ordenada respecto a una función *node-compare-p*. Y, con estas dos funciones anteriores, la función pedida consiste en llamar a *insert-nodes* usando como función de comparación la definida en la estrategia.

Pseudo-código

El pseudo-código de las funciones desarrolladas en este apartado sería entonces:

```
insert-node (node, lst-nodes, node-compare-p):
    si vacio(lst-nodes):
        devolver lista(node)
    si node-compare-p(node, primer_elemento(lst-nodes)):
        devolver concatenar(node, lst-nodes)
    si no:
        devolver concatenar(primer_elemento(lst-nodes), insert-node(node, resto(lst-nodes),
node-compare-p))

insert-nodes(nodes, lst-nodes, node-compare-p):
```

```
    si vacio(nodes):
        devolver lst-nodes
    si no:
        devolver insert-node(primer_elemento(nodes),
                    insert-nodes(resto(nodes), lst-nodes, node-compare-p),
                    node-compare-p)

insert-nodes-strategy (nodes, lst-nodes, strategy)
    devolver insert-nodes(nodes, lst-nodes, strategy.node-compare-p)
```

Ejercicio 8

En este apartado, al igual que en el ejercicio 5, simplemente hemos tenido que definir una estructura, con lo que no es necesaria ninguna batería de pruebas ni pseudo-código especial. Lo único a comentar sería la función de comparar nodos, que se basa simplemente en una modificación de la función `node-g-<=` dada en el enunciado, de forma que comparamos el campo *f* de ambos nodos, en vez del campo *g*.

Ejercicio 9

Batería de ejemplos

Como el algoritmo se basa en la función de la heurística dada, que depende del destino, los únicos cambios que podemos hacer para comprobar la corrección del algoritmo se basan en cambiar el origen del trayecto y las listas de ciudades obligatorias y prohibidas.

Además de probar que funciona correctamente con el origen y las listas de ciudades obligatorias y prohibidas dadas, hemos probado variando estos tres parámetros, los cuatro casos especiales que pueden causar problemas. A pesar de haber probado tanto con *fast* como con *cheap*, especificamos únicamente las de *fast*, por simplicidad de la memoria.

- Prohibir una ciudad obligatoria a la que se pueda pasar únicamente por tren, por ejemplo, *Orleans*:

  ```
  NIL
  ```

- Empezar directamente en la ciudad destino, con una ciudad obligatoria:

  Hemos dejado *París* como ciudad obligatoria. El resultado es:

  ```
  #S(NODE :STATE CALAIS
     :PARENT
     #S(NODE :STATE PARIS
        :PARENT
        #S(NODE :STATE CALAIS :PARENT NIL :ACTION NIL :DEPTH 0 :G 0 :H 0.0
           :F 0.0)
        :ACTION
        #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN CALAIS :FINAL PARIS
           :COST 34.0)
        :DEPTH 1 :G 34.0 :H 30.0 :F 64.0)
     :ACTION
     #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN PARIS :FINAL CALAIS :COST 34.0)
     :DEPTH 2 :G 68.0 :H 0.0 :F 68.0)
  ```

- Empezar directamente en la ciudad destino, sin ciudades obligatorias:

  ```
  #S(NODE :STATE CALAIS :PARENT NIL :ACTION NIL :DEPTH 0 :G 0 :H 0.0 :F 0.0)
  ```

- Prohibir ciudades de forma que el destino sea inalcanzable. Por ejemplo, empezando en *St-Malo*, podemos prohibir *Paris* y *Nantes*, de forma que el resultado es:

  ```
  NIL
  ```

Todos los casos funcionan de forma esperada, comprobando así que no se producen fallos como bucles infinitos, por ejemplo.

Comentarios sobre la implementación

A la hora de implementar el funcionamiento pedido en este apartado, nos hemos basado en el pseudocódigo que se nos proporciona en el enunciado. Sin embargo, hemos decidido crear algunas funciones auxiliares para facilitarnos la implementación de este pseudocódigo y para evitar repetir código.

Las funciones auxiliares que se han implementado son `root-node` que simplemente es una función que genera el nodo raíz de un problema , haciendo que su padre y su acción sea `nil`, su coste sea 0 y su estado el estado origen del problema; y la función `node-in-lst` que comprueba si un nodo está en una lista, esta segunda función va recorriendo la lista comparando el nodo con cada uno de los nodos de la lista mediante la función para comparar nodos en la estructura `problem`.

Estas dos funciones se emplean en las funciones `graph-search` y `graph-search-aux` que son las funciones pedidas y que siguen el pseudocódigo proporcionado.

Pseudocódigo

Según lo comentado anteriormente, el pseudocódigo de las funciones implementadas en este apartado es:

```
a-star-search (problem):
    devolver graph-search(problem, *A-star*)


graph-search (problem, strategy):
    devolver graph-search-aux(problem, list(root-node(problem)), NIL, strategy)


root-node (problem):
    devolver crear_nodo(state = problem.initial-state,
        parent = NIL,
        action = NIL,
        depth = 0,
        g = 0,
        h = problem.f-h(problem.initial-state))
        f = problem.f-h(problem.initial-state))


graph-search-aux (problem, open-nodes, closed-nodes, strategy):
    si open-nodes está vacía:
        devolver NIL
```

```
    si no:
        current-node := first(open-nodes)
        si current-node es el nodo destino:
            devolver current-node
        si no:
            rest-nodes := rest(open-nodes)
            repeated-node := node-in-lst(current-node, closed-nodes, problem)
            si repeated-node es nil (el nodo no esta en la lista de cerrados) o g(current-
node) <= g(repeated-node):
                devolver graph-search-aux(problem, insert-nodes-strategy(expand-
node(current-node, problem), rest-nodes, strategy), cons(current-node, closed-nodes))
            si no:
                devolver graph-search-aux(problem, rest-nodes, closed-nodes, strategy)


node-in-lst(node, lst, problem):
    si lst está vacía:
        devolver NIL
    si no:
        si node es igual (en el problema) a first(lst):
            devolver first(list)
        si no:
            devolver node-in-lst(node, rest(lst), problem)
```

Ejercicio 10

Batería de ejemplos

Del mismo modo que en varios apartados anteriores consideramos que, sabiendo que las funciones de los apartados anteriores funcionan de manera correcta, para comprobar el correcto funcionamiento de las funciones de este apartado bastaría con comprobar que si no se les pasa ningún nodo devuelven `nil`, y si se les pasa uno devuelven el camino o la secuencia de acciones de forma correcta. Por tanto, consideramos que tan solo es necesario añadir en esta batería de ejemplos el caso en el que a `action-sequence` se le pasa un nodo que es `nil`, puesto que los casos de prueba que se nos proporcionan son suficientes para comprobar el correcto funcionamiento de la implementación en todos los demás casos.

Este ejemplo sería el siguiente:

```
(action-sequence NIL) ; --> NIL
```

Realizando esta simple comprobación, así como los casos de prueba proporcionados, comprobamos que nuestra implementación funciona de manera correcta.

Comentarios sobre la implementación

Para implementar la función `solution-path` tenemos una función llamada `navigate-path` que implementados en un apartado anterior que realiza el funcionamiento pedido. Por tanto, `solution-path` simplemente devuelve el resultado de llamar a la función `navigate-path` con el nodo pasado por argumento.

En el caso de `action-sequence` no disponemos de una función que tenga la misma funcionalidad, por lo que esta sí debe diseñarse de cero. Para ello en primer lugar comprobamos si el nodo es `nil`, y si lo es devolvemos `nil`. En caso contrario llamamos a una función auxiliar que hemos llamado `action-sequence-aux` que emplea recursión para ir recorriendo las acciones que han generado los distintos nodos del camino seguido hasta el nodo que se pasa como argumento. Para ello establecemos que esta función auxiliar se llame recursivamente con el padre del nodo actual añadiendo a una lista de acciones (que comienza vacía) la acción que lo ha generado (que se obtiene del campo `action` de la estructura `node`), y establecemos un caso base (cuando el nodo no tiene padre) en el que se devuelve la lista de acciones que se tiene.

Pseudo-código

El pseudocódigo de las funciones pedidas será por tanto el siguiente:

```
solution-path (node):
    return navigate-path node


action-sequence-aux (node, actions):
    si el nodo tiene padre:
        return action-sequence-aux(padre(nodo), cons(accion(nodo), actions))
    si no:
        return actions


action-sequence (node):
    si el nodo es nil:
        return NIL
    si no:
        return action-sequence-aux (node, NIL)
```

Ejercicio 11

Batería de ejemplos

Sabiendo que las funciones implementadas en apartados anteriores funcionan de forma correcta, para probar el correcto funcionamiento de este apartado basta con probar que las nuevas estrategias definidas (y sus respectivas funciones de comparación) hacen que las funciones de búsqueda encuentren el camino correcto con dicha estrategia. Para probar esto es suficiente con probar para las 2 nuevas estrategias que en la resolución de los problemas `*travel-fast*` y `*tavel-cheap*` se encuentra el camino correcto.

Por lo tanto la batería de ejemplos propuesta para este apartado será:

```
(solution-path (graph-search *travel-cheap* *depth-first*)) ; -> (MARSEILLE TOULOUSE NANTES
ST-MALO PARIS REIMS CALAIS)

(solution-path (graph-search *travel-fast* *depth-first*)) ; -> (MARSEILLE TOULOUSE NANTES
ST-MALO PARIS REIMS CALAIS)

(solution-path (graph-search *travel-cheap* *breadth-first*)) ; -> (MARSEILLE TOULOUSE
NANTES ST-MALO PARIS REIMS CALAIS)

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

Al probar estos cuatro ejemplos vemos que las nuevas estrategias están implementadas de forma correcta.

Comentarios sobre la implementación

Como hemos estructurado el programa de forma modular, basta con definir una nueva función de comparar nodos para recorrer el árbol de forma distinta, aplicando así otros algoritmos de búsqueda.

En el caso de búsqueda en profundidad, los nodos se almacenan en una pila a medida que se descubren, de forma que basta con que la función `depth-first-node-compare-p` devuelva siempre un *True*, con la lista de abiertos actúa como una pila.

Para la búsqueda en anchura, la lista de abiertos se tiene que comportar como una cola, añadiendo los nodos siempre al final. Para esto, basta con que la función `breadth-first-node-compare-p` devuelva siempre NIL, de forma que al insertar un nodo, este acaba siempre en el final de la lista.

Pseudo-código

Como hemos mencionado en el apartado anterior, el pseudo-código de las dos funciones sería:

```
depth-first-node-compare-p (node-1, node-2):
    devolver True

breadth-first-node-compare-p (node-1, node-2):
    devolver NIL
```

Ejercicio 12

La heurística escogida es el coste del enlace más barato para salir de la ciudad.

Sabemos que si una heurística es monótona, entonces es también admisible. Así, para comprobar si nuestra heurística es válida para solucionar el problema podemos comprobar si es monótona, y, si lo es, entonces será admisible para nuestro problema.

Para probar que la heurística escogida es monótona, por definición, debe cumplir:

$$h(x) \leq g(x \to x') + h(x')$$

Donde x' es el nodo sucesor de x al que se quiere ir, y $g(x \to x')$ es el coste de ir de x a x'.

Viendo los grafos que se nos proporcionan y analizando la heurística proporcionada vemos que la heurística de un nodo (el coste del enlace más barato para salir de él) será siempre mayor o igual al coste de ir desde ese nodo a cualquiera de los adyacentes, pues el coste para ir a alguno de los adyacentes será el coste de alguna de los enlaces que salen del nodo, y como heurística hemos cogido el menor de estos costes. Por lo tanto, sabemos que:

$$h(x) \leq g(x \to x')$$

Además de esto, al observar los grafos vemos que no existe ningún enlace cuyo coste sea negativo por lo que ningún nodo puede tener una heurística negativa, de esto deducimos que:

$$h(x) \leq g(x \to x') \leq g(x \to x') + h(x')$$

Por lo tanto, la heurística elegida cumple los requisitos para ser monótona, luego esta es también admisible.

Aplicando esta heurística, el parámetro `*estimate-new*` queda de la siguiente manera:

```
(defparameter *estimate-new*
  '((Calais (0.0 0.0)) (Reims (25.0 15.0)) (Paris (30.0 10.0))
    (Nancy (50.0 20.0)) (Orleans (55.0 38.0)) (St-Malo (65.0 15.0))
    (Nantes (75.0 15.0)) (Brest (90.0 40.0)) (Nevers (70.0 20.0))
    (Limoges (100.0 35.0)) (Roenne (85.0 5.0)) (Lyon (105.0 5.0))
    (Toulouse (130.0 35.0)) (Avignon (135.0 10.0)) (Marseille (145.0 25.0))))
```

Para generar la nueva estructura `*travel-cost-new*` basta con tomar la estructura `*travel-cheap*` de los apartados anteriores y cambiar su parámetro `f-h` (función que determina la heurística de un nodo) para que emplee la nueva estructura `*estuimate-new*` en lugar de la antigua `*estimate*`. De este modo la nueva estructura será:

```
(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*)))))
```

Por último, para comprobar que el algoritmo implementado es más eficiente con esta nueva heurística que con la heurística previa (siempre 0), ejecutamos el algoritmo con ambas heurísticas empleando la función `time` de `lisp`, que nos muestra el tiempo de ejecución. Al hacerlo obtenemos el siguiente resultado:

```
(time (solution-path (a-star-search *travel-cheap*)))
;;; ->
;;; cpu time (non-gc) 0.015625 sec user, 0.000000 sec system
;;; cpu time (gc)     0.000000 sec user, 0.000000 sec system
;;; cpu time (total)  0.015625 sec user, 0.000000 sec system
```

```
;;; real time  0.011000 sec (142.0%)
;;; space allocation:
;;;  43,116 cons cells, 888,808 other bytes, 0 static bytes
;;; Page Faults: major: 0 (gc: 0), minor: 0 (gc: 0)
;;;(MARSEILLE TOULOUSE LIMOGES NEVERS PARIS REIMS CALAIS)

(time (solution-path (a-star-search *travel-cost-new*)))
;;; ->
;;; cpu time (non-gc) 0.015625 sec user, 0.000000 sec system
;;; cpu time (gc)     0.000000 sec user, 0.000000 sec system
;;; cpu time (total)  0.015625 sec user, 0.000000 sec system
;;; real time  0.008000 sec (195.3%)
;;; space allocation:
;;;  43,116 cons cells, 888,680 other bytes, 0 static bytes
;;; Page Faults: major: 0 (gc: 0), minor: 0 (gc: 0)
;;;(MARSEILLE TOULOUSE LIMOGES NEVERS PARIS REIMS CALAIS)
```

Como se observa en los resultados obtenidos, el tiempo de ejecución con esta nueva heurística se reduce considerablemente, pasando de 0,011 segundos a 0,008 segundos, lo que nos demuestra que este algoritmo es notablemente más eficiente con la nueva heurística propuesta.

Preguntas

1. ¿Por qué se ha realizado este diseño para resolver el problema de búsqueda?

Hemos implementado el problema de forma modular, dividido en funciones sencillas y cada una de ellas con un fin específico.

De esta forma, es mucho más sencillo de programar y debuggear, y permite ser aplicado a cualquier grafo, con múltiples estrategias de búsqueda. Basta con definir las nuevas heurísticas, un test que permita comprobar si hemos alcanzado el estado final, y las acciones a aplicar a un estado para obtener sus sucesores.

2.A ¿Qué ventajas aporta?

Como hemos mencionado en el test anterior, la principal ventaja, además de la posibilidad de aplicar el algoritmo A^* a otros problemas que necesitamos resolver, como se explica en el apartado siguiente, es la limpieza y claridad del código, que permite ser entendido fácil y rápidamente, de forma que se agiliza la solución de posibles errores.

2.B ¿Por qué se han utilizado funciones lambda para especificar el test objetivo, la heurística y los operadores del problema?

El uso de funciones lambda para estos tests permite que estas sean intercambiadas fácilmente, pudiendo aprovechar así los algoritmos principales (que son los más difíciles de programar) para obtener el camino óptimo en cualquier otro problema, sustituyendo simplemente estas funciones lambda y algunas estructuras como los nuevos problemas a usar.

3. Sabiendo que en cada nodo de búsqueda hay un campo "parent", que proporciona una referencia al nodo a partir del cual se ha generado el actual ¿Es eficiente el uso de memoria?

Como el algoritmo se basa en la construcción de un árbol, bien los hijos tienen una referencia al padre, o bien el padre a los hijos, con lo que es inevitable usar estas referencias a nodos.

La única forma que conocemos y que permite mejorar la eficiencia en el uso memoria del algoritmo sería el uso de IDA^* en vez de A^*.

4. ¿Cuál es la complejidad espacial del algoritmo implementado?

Por ser A^* el algoritmo, sabemos que la fórmula para calcular la complejidad espacial es $O(b^{\lceil C^*/\varepsilon \rceil})$, donde:

- b es el factor de ramificación.
- C^* es el coste del camino de la solución óptima.
- ε es el coste mínimo de una acción.

5. ¿Cuál es la complejidad temporal del algoritmo?

En este caso, por ser A^* el algoritmo, a complejidad espacial y la temporal coinciden, por lo que la fórmula sería exactamente la misma que en el apartado anterior.

6. Indicad qué partes del código se modificarían para limitar el número de veces que se puede utilizar la acción "navegar por agujeros de gusano"

Si lo que queremos es limitar el número de veces que podemos pasar por un enlace bidireccional, es decir, un enlace mediante tren entre dos ciudades, lo más sencillo es modificar la tabla de trenes, almacenado en nuestro programa en la estructura $*trains*$, añadiendo un parámetro que indicase el número de veces que podemos pasar por dicho enlace.

De esta forma, bastaría con modificar ligeramente la función *f-goal-test* de forma que se comprobase, además de no recorrer las ciudades prohibidas, que no se sobrepasen el número de veces que se recorre cada enlace por tren.

Código

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;    Node in search tree
;;
(defstruct node
```

```
  state           ; state label
  parent          ; parent node
  action          ; action that generated the current node from its parent
  (depth 0)       ; depth in the search tree
  (g 0)           ; cost of the path from the initial state to this node
  (h 0)           ; value of the heurstic
  (f 0))          ; g + h
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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

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

(defparameter *cities* '(Calais Reims Paris Nancy Orleans
                         St-Malo Brest Nevers Limoges
                         Roenne Lyon Toulouse Avignon Marseille))
(defparameter *trains*
  '((Paris Calais (34.0 60.0))      (Calais Paris (34.0 60.0))
    (Reims Calais (35.0 70.0))      (Calais Reims (35.0 70.0))
    (Nancy Reims (35.0 55.0))       (Reims Nancy (35.0 55.0))
    (Paris Nancy (40.0 67.0))       (Nancy Paris (40.0 67.0))
    (Paris Nevers (48.0 75.0))      (Nevers Paris (48.0 75.0))
    (Paris Orleans (23.0 38.0))     (Orleans Paris (23.0 38.0))
    (Paris St-Malo (40.0 70.0))     (St-Malo Paris (40.0 70.0))
    (St-Malo Nantes (20.0 28.0))    (Nantes St-Malo (20.0 28.0))
    (St-Malo Brest (30.0 40.0))     (Brest St-Malo (30.0 40.0))
    (Nantes Brest (35.0 50.0))      (Brest Nantes (35.0 50.0))
    (Nantes Orleans (37.0 55.0))    (Orleans Nantes (37.0 55.0))
    (Nantes Toulouse (80.0 130.0))  (Toulouse Nantes (80.0 130.0))
    (Orleans Limoges (55.0 85.0))   (Limoges Orleans (55.0 85.0))
    (Limoges Nevers (42.0 60.0))    (Nevers Limoges (42.0 60.0))
    (Limoges Toulouse (25.0 35.0))  (Toulouse Limoges (25.0 35.0))
    (Toulouse Lyon (60.0 95.0))     (Lyon Toulouse (60.0 95.0))
    (Lyon Roenne (18.0 25.0))       (Roenne Lyon  (18.0 25.0))
```

```
    (Lyon Avignon (30.0 40.0))      (Avignon Lyon (30.0 40.0))
    (Avignon Marseille (16.0 25.0)) (Marseille Avignon (16.0 25.0))
    (Marseille Toulouse (65.0 120.0)) (Toulouse Marseille (65.0 120.0))))

(defparameter *canals*
  '((Reims Calais (75.0 15.0)) (Paris Reims (90.0 10.0))
    (Paris Nancy (80.0 10.0)) (Nancy reims (70.0 20.0))
    (Lyon Nancy (150.0 20.0)) (Nevers Paris (90.0 10.0))
    (Roenne Nevers (40.0 5.0)) (Lyon Roenne (40.0 5.0))
    (Lyon Avignon (50.0 20.0)) (Avignon Marseille (35.0 10.0))
    (Nantes St-Malo (40.0 15.0)) (St-Malo Brest (65.0 15.0))
    (Nantes Brest (75.0 15.0))))

(defparameter *estimate*
  '((Calais (0.0 0.0)) (Reims (25.0 0.0)) (Paris (30.0 0.0))
    (Nancy (50.0 0.0)) (Orleans (55.0 0.0)) (St-Malo (65.0 0.0))
    (Nantes (75.0 0.0)) (Brest (90.0 0.0)) (Nevers (70.0 0.0))
    (Limoges (100.0 0.0)) (Roenne (85.0 0.0)) (Lyon (105.0 0.0))
    (Toulouse (130.0 0.0)) (Avignon (135.0 0.0)) (Marseille (145.0 0.0))))

(defparameter *origin* 'Marseille)

(defparameter *destination* '(Calais))

(defparameter *forbidden*  '(Avignon))

(defparameter *mandatory* '(Paris))

(defun f-h-time (state sensors)
  (cond ((null sensors) NIL)
    ((eql (first (first sensors)) state) (first (second (first sensors))))
    (t (f-h-time state (rest sensors)))))

(defun f-h-price (state sensors)
  (cond ((null sensors) NIL)
    ((eql (first (first sensors)) state) (second (second (first sensors))))
    (t (f-h-time state (rest sensors)))))
;;
;; END: Exercise 1 -- Evaluation of the heuristic
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN: Exercise 2 -- Navigation operators
;;

(defun navigate (state lst-edges cfun name &optional forbidden)
  (cond ((null lst-edges) NIL)
    ((and (eql (first (first lst-edges)) state) (not (find (second (first lst-edges))
forbidden)))
      (cons
        (make-action
```

```
            :name name
            :origin state
            :final (second (first lst-edges))
            :cost (funcall cfun (third (first lst-edges))))
        (navigate state (rest lst-edges) cfun name forbidden )))
    (t (navigate state (rest lst-edges) cfun name forbidden ))))

(defun navigate-canal-time (state canals)
  (navigate state canals #'first 'navigate-canal-time))

(defun navigate-canal-price (state canals)
  (navigate state canals #'second 'navigate-canal-price))

(defun navigate-train-time (state trains forbidden)
  (navigate state trains #'first 'navigate-train-time forbidden))

(defun navigate-train-price (state trains forbidden)
  (navigate state trains #'second 'navigate-train-price forbidden))

;;
;; END: Exercise 2 -- Navigation operators
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN: Exercise 3 -- Goal test
;;

(defun navigate-path-aux (node path)
    (if (null (node-parent node))
        (cons (node-state node) path)
        (navigate-path-aux (node-parent node) (cons (node-state node) path))))

(defun navigate-path (node)
    (if (null node)
        NIL
        (navigate-path-aux node '())))

(defun check-mandatory (path mandatory)
  (cond ((null mandatory) t)
    ((not (find (first mandatory) path)) NIL)
    (t (check-mandatory path (rest mandatory)))))

(defun f-goal-test (node destination mandatory)
  (if (null node)
    NIL
    (if (not (find (node-state node) destination))
      NIL
      (check-mandatory (navigate-path node) mandatory))))

;;
;; END: Exercise 3 -- Goal test
```

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


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

 (defun f-equivalent-paths(path-1 path-2 &optional mandatory)
    (cond ((null mandatory)
        t)
      ((and (not (find (first mandatory) path-1)) (find (first mandatory) path-2))
        NIL)
      ((and (not (find (first mandatory) path-2)) (find (first mandatory) path-1))
        NIL)
      (t (f-equivalent-paths path-1 path-2 (rest mandatory)))))

(defun f-search-state-equal (node-1 node-2 &optional mandatory)
  (if (not (eql (node-state node-1) (node-state node-2)))
    NIL
    (f-equivalent-paths (navigate-path node-1) (navigate-path node-2) mandatory)))

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  BEGIN: Exercise 5 -- Define the problem structure
;;

(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*)))))

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


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN Exercise 6: Expand node
;;

(defun expand-node-action (action parent problem)
    (let ((g-value (+ (node-g parent) (action-cost action)))
            (h-value (funcall (problem-f-h problem) (action-final action))))
        (make-node
          :state (action-final action)
          :parent parent
          :action action
          :depth (+ 1 (node-depth parent))
          :g g-value
          :h h-value
          :f (+ g-value h-value))))

(defun expand-node (node problem)
  (let ((action-list (mapcan (lambda (c) (funcall c node)) (problem-operators problem))))
    (mapcar (lambda (action) (expand-node-action action node problem)) action-list)))
;;
;;  END: Exercise 6 -- Expand node
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  BEGIN Exercise 7 -- Node list management
;;;

(defun insert-node (node lst-nodes node-compare-p)
    (cond ((null lst-nodes)
            (list node))
        ((funcall node-compare-p node (first lst-nodes))
            (cons node lst-nodes))
        (t (cons (first lst-nodes) (insert-node node (rest lst-nodes) node-compare-p)))))

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

```lisp
        (insert-node (first nodes) (insert-nodes (rest nodes) lst-nodes node-compare-p)
node-compare-p)))

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

;;
;;    END: Exercize 7 -- Node list management
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; BEGIN: Exercise 8 -- Definition of the A* strategy
;;

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

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

;;
;; END: Exercise 8 -- Definition of the A* strategy
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    BEGIN Exercise 9: Search algorithm
;;;

(defun node-in-lst(node lst problem)
    (cond ((null lst) NIL)
        ((funcall (problem-f-search-state-equal problem) node (first lst)) (first lst))
        (t (node-in-lst node (rest lst) problem))))

(defun graph-search-aux (problem open-nodes closed-nodes strategy)
    (if (null open-nodes)
            NIL ; No se encuentra la solución
        (let ((current-node (first open-nodes)))
            (if (funcall (problem-f-goal-test problem) current-node)
                    current-node ; Devuelve la solución
                (let ((rest-nodes (rest open-nodes)) (repeated-node  (node-in-lst current-
node closed-nodes problem)))
                    (if (or (null repeated-node) (<= (node-g current-node) (node-g
repeated-node)))
                            (graph-search-aux problem
                                (insert-nodes-strategy
```

```
                                    (expand-node current-node problem)
                                    rest-nodes
                                    strategy)
                                (cons current-node closed-nodes)
                                strategy)
                        (graph-search-aux problem rest-nodes closed-nodes strategy)))))))

(defun root-node (problem)
    (make-node
        :state (problem-initial-state problem)
        :parent NIL
        :action NIL
        :depth 0
        :g 0
        :h (funcall (problem-f-h problem) (problem-initial-state problem))
        :f (funcall (problem-f-h problem) (problem-initial-state problem))))

(defun graph-search (problem strategy)
    (graph-search-aux problem (list (root-node problem)) '() strategy))

;
;  A* search is simply a function that solves a problem using the A* strategy
;
(defun a-star-search (problem)
    (graph-search problem *A-star*))


;;
;; END: Exercise 9 -- Search algorithm
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    BEGIN Exercise 10: Solution path
;;;
;*** solution-path ***

(defun solution-path (node)
    (navigate-path node))

;*** action-sequence ***
; Visualize sequence of actions

(defun action-sequence-aux (node actions)
    (if (null (node-parent node))
            actions
        (action-sequence-aux (node-parent node) (cons (node-action node) actions))))

(defun action-sequence (node)
    (if (null node)
            NIL
```

```
        (action-sequence-aux node NIL)))

;;;
;;;    END Exercise 10: Solution path / action sequence
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    BEGIN Exercise 11
;;;


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

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



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

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



;;;
;;;    END Exercise 11
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    BEGIN Exercise 12
;;;

;;; La heurística escogida es el coste del enlace más barato para salir de la ciudad

(defparameter *estimate-new*
  '((Calais (0.0 0.0)) (Reims (25.0 15.0)) (Paris (30.0 10.0))
    (Nancy (50.0 20.0)) (Orleans (55.0 38.0)) (St-Malo (65.0 15.0))
    (Nantes (75.0 15.0)) (Brest (90.0 40.0)) (Nevers (70.0 20.0))
    (Limoges (100.0 35.0)) (Roenne (85.0 5.0)) (Lyon (105.0 5.0))
    (Toulouse (130.0 35.0)) (Avignon (135.0 10.0)) (Marseille (145.0 25.0))))
```

```
(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*)))))
;;;
;;;    END Exercise 12
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```