P2 Inteligencia Artificial:

Búsqueda

Grupo 2361: Equipo 8

Guillermo Rodríguez Senís, Javier Muñoz Haro

**Introducción**

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

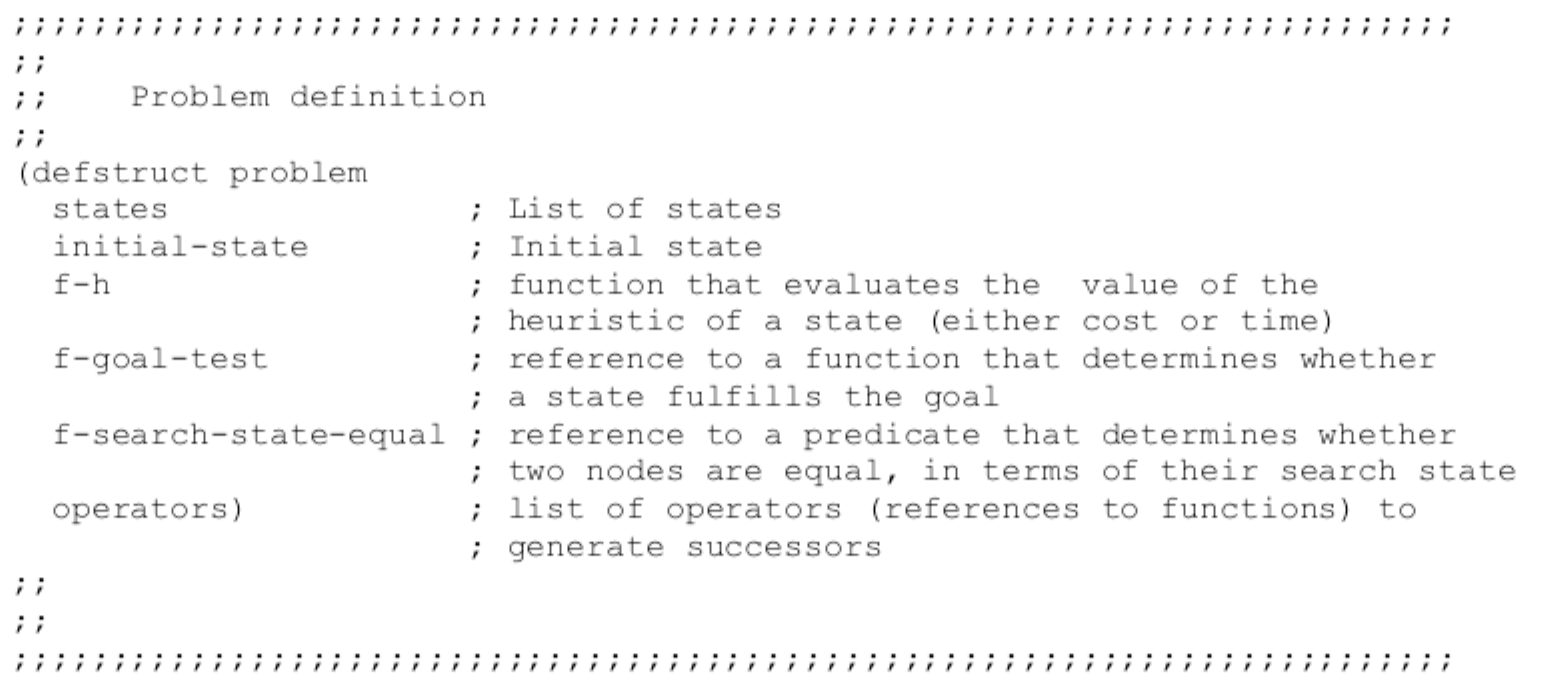
Con las diferencias mencionadas previamente entre los dos grafos, debemos aportar una mas, y posiblemente la mas significativa, es que la red de ferrocarril(que es bidireccional) tiene ciudades “prohibidas”, es decir, ciudades por las que no podemos pasar, mientras que los canales (que son unidireccionales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

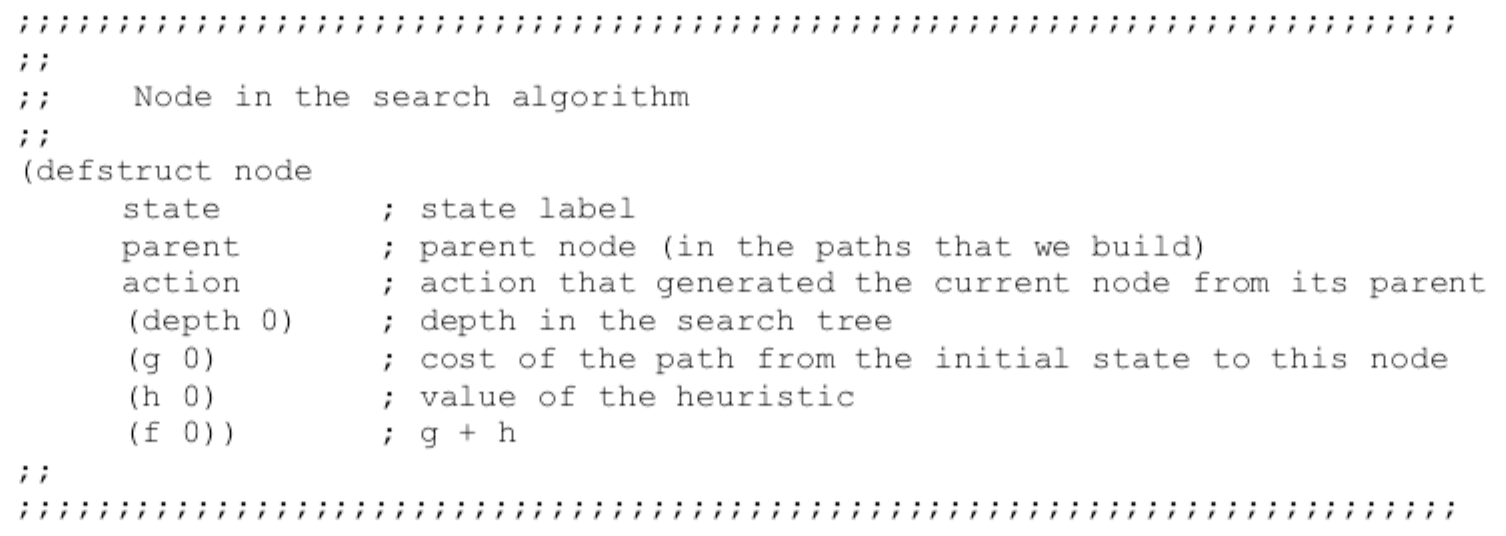
Para determinar las heurísticas nos hemos apoyado en los costes que se nos definen en la práctica para viajar de ciudad en ciudad, es decir tiempo y coste (dinero), dependiendo de los diferentes transportes que sean utilizados. El buscador de caminos deberá, teniendo en cuenta las heurísticas, buscar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

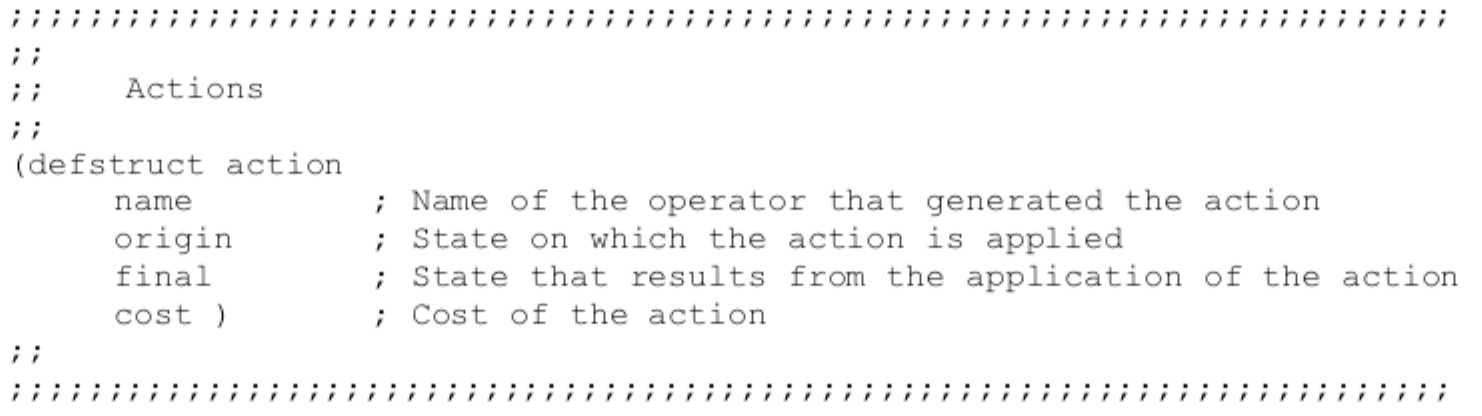
En esta práctica se nos da un fichero como material de partida en el que se formaliza el modelo previamente definido. En él se nos dan:

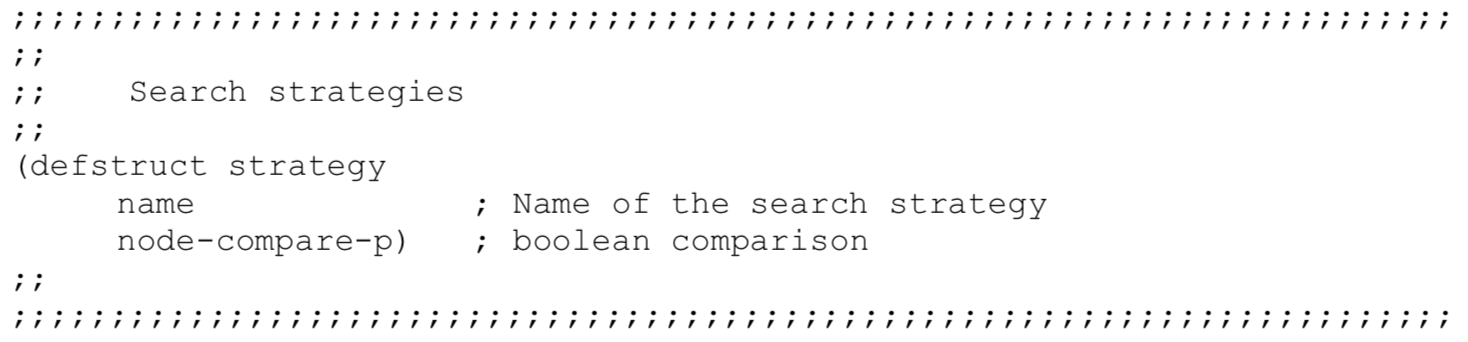
* **Ciudades:** Las cuales son definidas como una lista que representan sus nombres.
* **Conexiones:** Las cuales representan los dos grafos, el de canales y el ferroviario, poniendo el origen, el destino, el tiempo de recorrido y el coste del uso de dicho transporte.
* **Estimaciones:** Pares formados por ciudades y valores correspondientes a las heurísticas (tiempo y coste).
* **Ciudad origen:** Es decir, la ciudad donde se empieza el viaje.
* **Ciudad destino:** Es decir, la ciudad donde se termina el viaje
* **Ciudades obligadas:** Aquellas ciudades por las que nuestro itinerario debe pasar.
* **Ciudades prohibidas:** Aquellas ciudades por las que no se puede acceder mediante tren.

**Estructuras:**



****

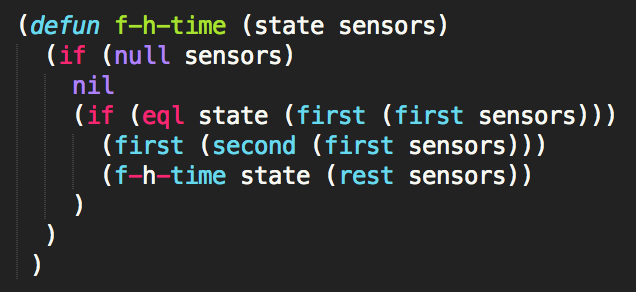
****

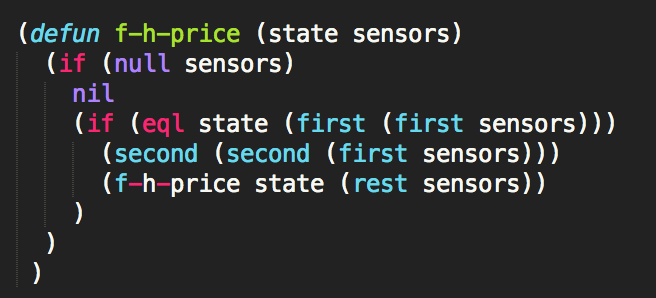
****

**Ejercicio 1:**

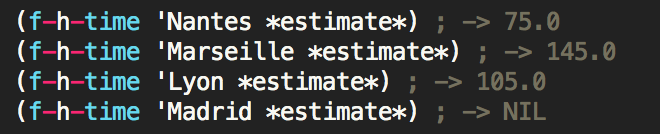
En este ejercicio tenemos que implementar dos funciones las cuales saquen las dos heurísticas que tenemos en nuestro sistema, tiempo y coste.

Las dos funciones son las siguientes:

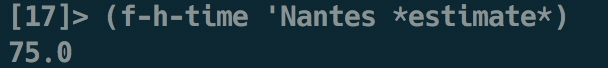




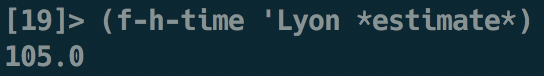
Y se nos ponen los siguientes ejemplos:

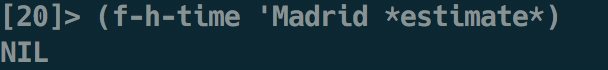


Los cuales al ejecutarse en la consola de Clisp obtenemos:





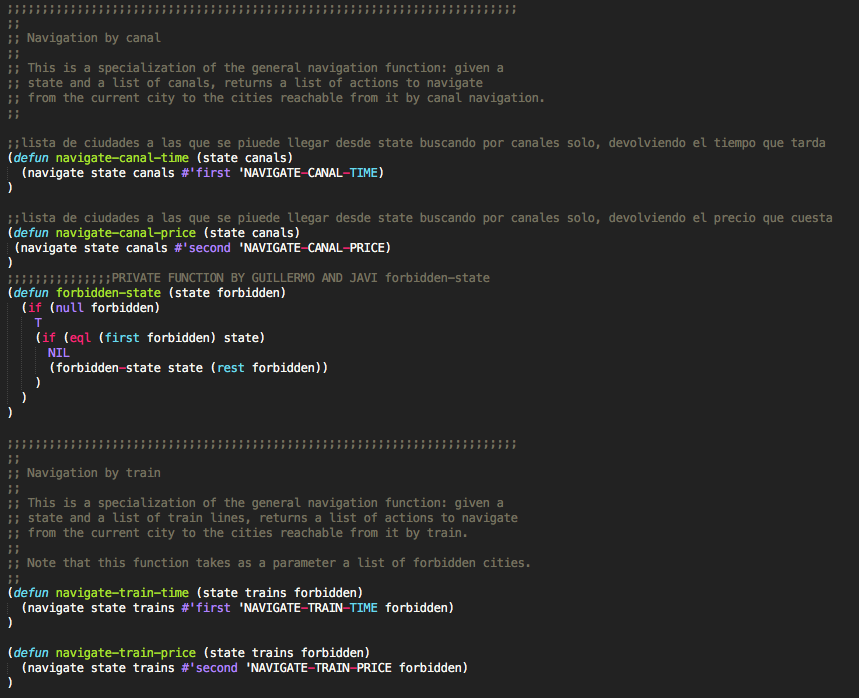




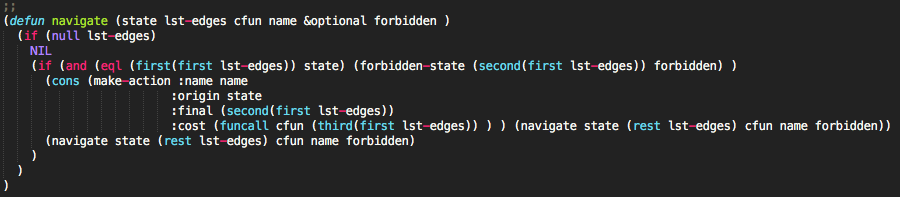
**Ejercicio 2:**

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

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

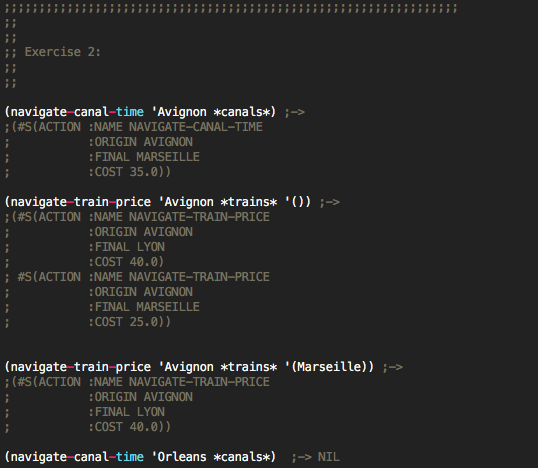


Todas las funciones previamente implementadas son auxiliares con respecto a la función troncal, que es navigate, esta función es la que decide, dependiendo de la situación en la que nos encontremos del recorrido que acción tomar.



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

Para estos ejemplos, obtenemos:



**Ejemplo 1:**

****

**Ejemplo2:**

****

**Ejemplo 3:**

****

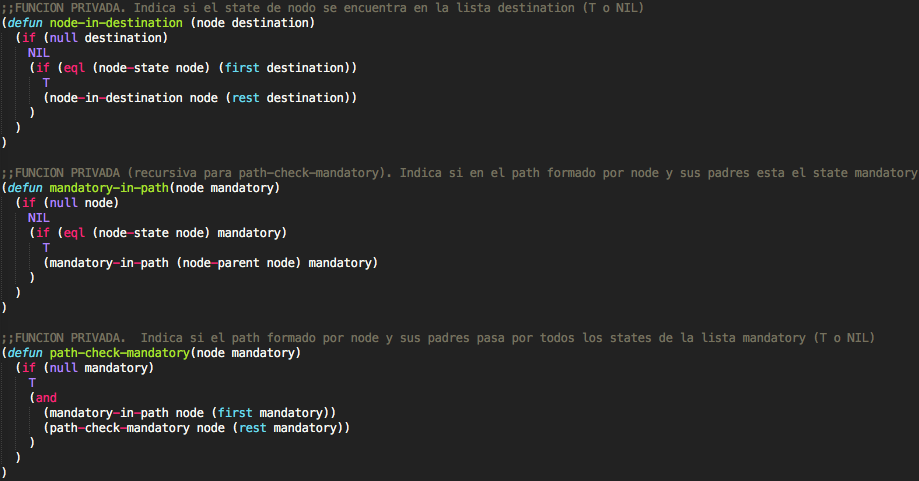
**Ejemplo 4:**

****

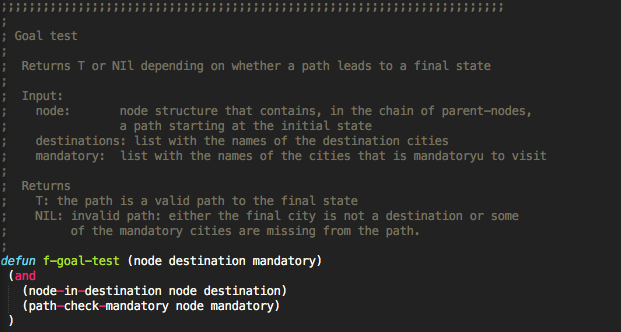
**Ejercicio 3:**

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

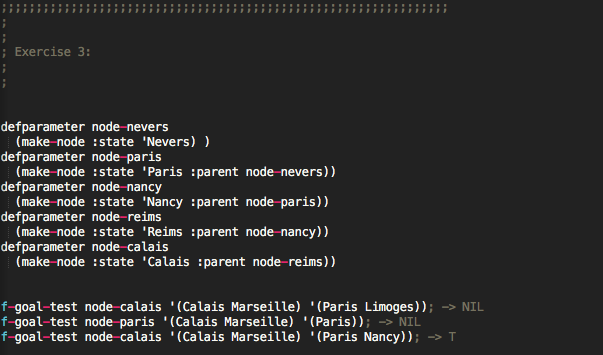
En nuestra implementación, pese a pedir solo realizar una función, hemos tenido que realizar 3 funciones auxiliares para modularizar la funcionalidad:



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



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



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

**Ejemplo 1:**



**Ejemplo 2:**



**Ejemplo 3:**



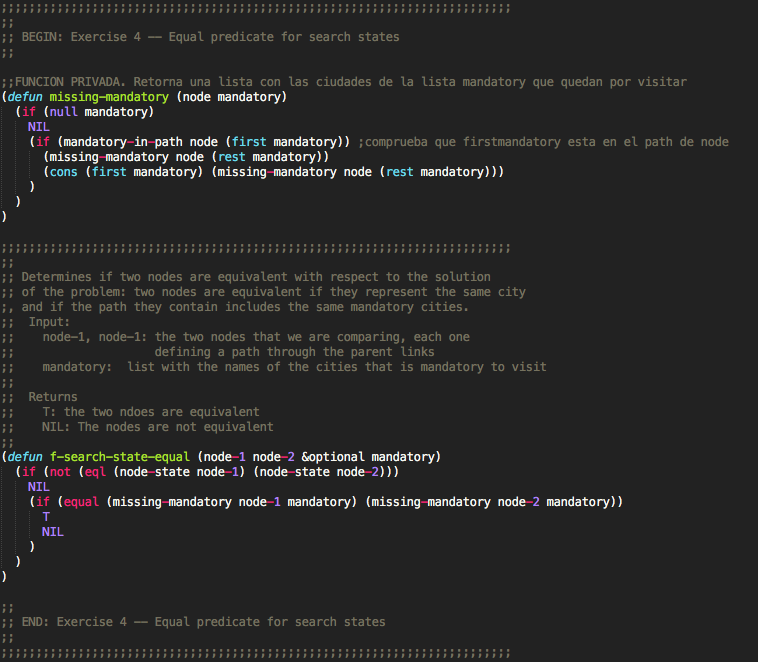
Como vemos, los resultados esperados coinciden con los obtenidos.

**Ejercicio 4**

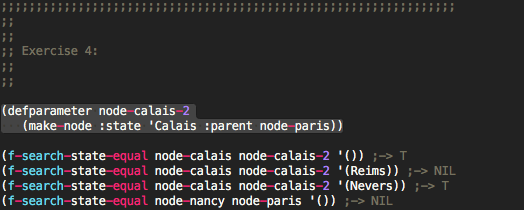
En este ejercicio, se nos pide implementar una función que básicamente compare los nodos (los cuales representan las ciudades) y los estados de los mismos, de acuerdo a la funcionalidad de búsqueda.

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

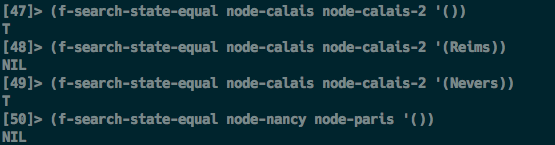
Nuestra implementación del código se divide en dos funciones, como hemos hecho previamente, para dividir la funcionalidad.



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



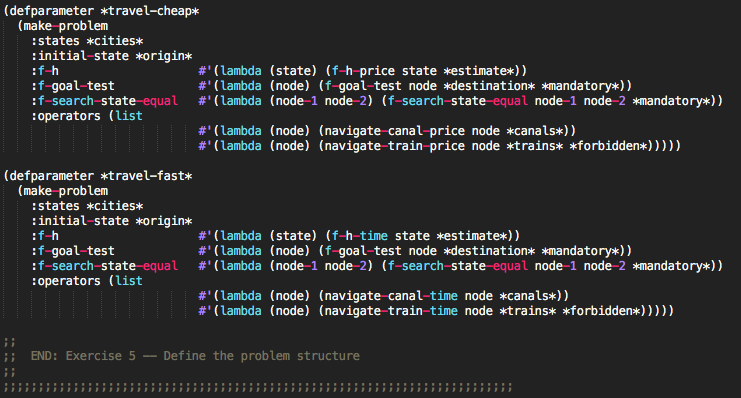
Y los resultados que obtenemos al ejecutar esos ejemplos son los siguientes:



**Ejercicio 5**

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

Las estructuras que son proporcionadas son las siguientes:



Como podemos observar y se explica en el enunciado de la práctica, cada estructura tiene una función heurística y dos operadores (tiempo y coste).

(Para este ejercicio no hay batería de ejemplos).

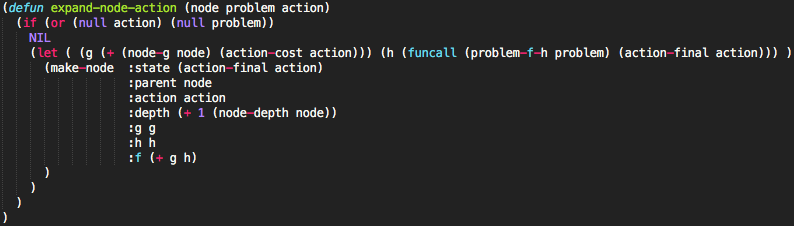
**Ejercicio 6**

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

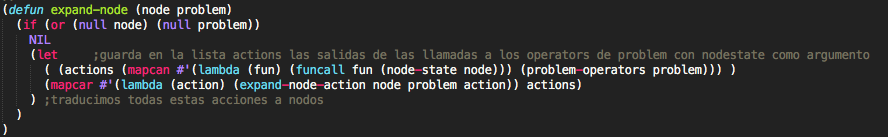
Para hacer esto en el problema se nos pide implementar la función expand-node.

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

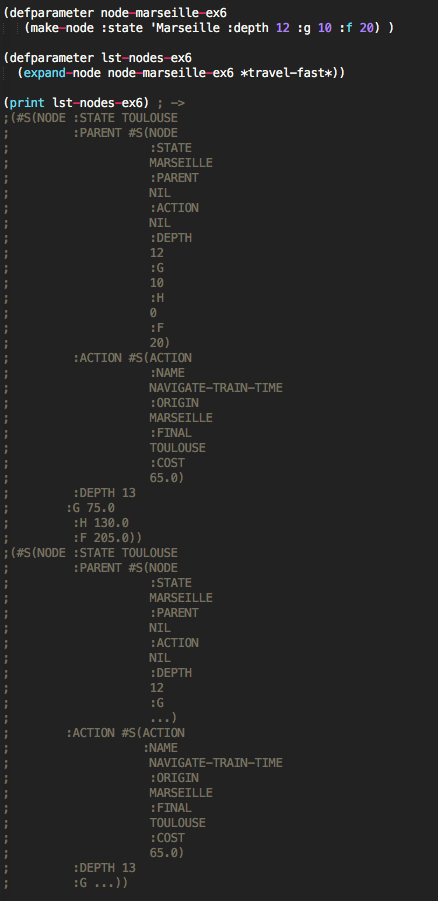
Para crear los nodos que se pueden alcanzar desde el nodo actual iteramos con expand-node, pero es esta función, expand-node-action la que se encarga de crear la estructura node que representa el nodo al que se puede llegar desde el nodo actual



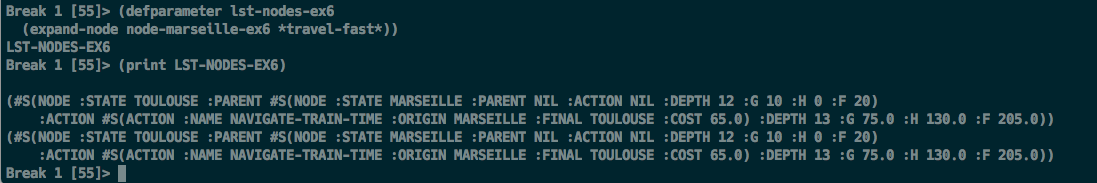
Nuestra función principal es, como hemos dicho antes, expand-node, la cual devuelve la lista de nodos de donde se puede acceder dado el nodo actual.



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



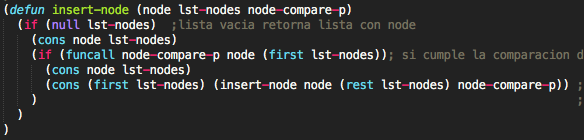
Y estos son los resultados obtenidos:



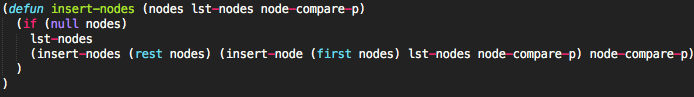
**Ejercicio 7**

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

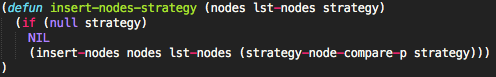
Hemos creado una función privada, llamada insert-node, la cual inserta el nodo en la lista ordenada sin desordenar esta.



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

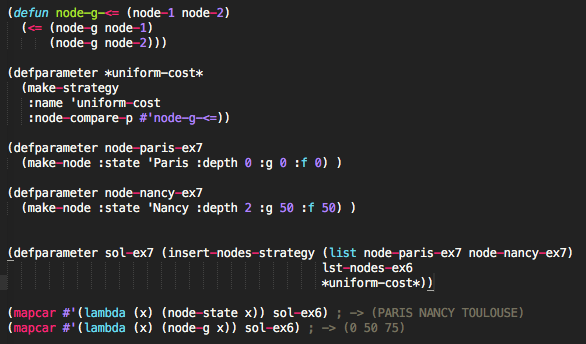


Po último, tenemos la función troncal, la cual tiene como entrada una lista de nodos ordenada, otra lista de nodos (potencialmente desordenados) y una estrategia. Esta función devuelve la lista de nodos ordenada, tal que:



**Ejemplo de prueba:**

Entrada:

****

Salida 1:

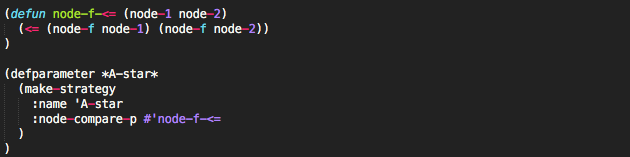


Salida 2:



**Ejercicio 8**

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



**CUESTIONES**

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

Porque es un diseño eficiente. Por una parte, este diseño evalúa los costes y permite conocer los caminos que forman los nodos. Además, dado que es un problema que claramente utiliza muchas listas (ciudades prohibidas, ciudades obligatorias, ciudades por visitar), Lisp es un lenguaje idóneo para optimizar las búsquedas. Por otro lado tiene en cuenta ambas heurísticas: tiempo y coste. Y también es eficiente el trabajar con estructuras que nos son muy pesadas pero que tienen mucha potencia como pueden ser los caminos que crean los nodos con sus padres, o todos los tipos de operadores.

2.En concreto,

a. ¿Qué ventajas aporta?

El uso de todas estas estructuras, hacen que el diseño sea muy versátil. Aun siendo estructuras tan simples, podemos trabajar sobre diferentes heurísticas y sobre diferentes transportes (tren o barco).

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

Porque hacen que el diseño sea menos pesado y mas genérico, de manera que independientemente del transporte o la heurística podemos resolver el problema con un único diseño. Además es muy cómodo a la hora de trabajar con listas.

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?

Si es eficiente. Aunque podrían existir algún otro tipo de algoritmo en la que no fuese necesario guardar las referencias en estructuras, siempre habrá que de alguna forma poder acceder a ellas. En el caso de nuestro diseño, simplemente se guarda la referencia al nodo parent para poder comprobar en las búsquedas los caminos. Esto siempre será mas eficiente que hacer copias de nodos, o tener el camino entero en la estructura

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

La complejidad espacial para el algoritmo A\* seria:

5.¿Cuál es la complejidad temporal del algoritmo?

La complejidad temporal para el algoritmo A\* coincide con la espacial y seria: 