PRÁCTICA 2. BÚSQUEDA

Fecha de Publicación:	2020/02/07	
Fecha de Entrega:	[grupos jueves:	miércoles 2020/03/18]
	[grupos viernes:	jueves 2020/03/19]
Planificación:	semana 1:	Ejercicios 1, 2, 3, 4, 5, 6
	semana 2:	Ejercicios 7, 8, 9, 10
	semana 3:	Ejercicios 11 + memoria

Versión: 2020/21/02

Importante:

- Utilizad la versión XXX de Portacle para desarrollar el código.
- Definid problemas de búsqueda sencillos (que podéis resolver con papel y lápiz) para depurar el código.

Forma de Entrega: Según lo dispuesto en las normas de entrega generales, publicadas en Moodle.

- El código debe estar en un único fichero.
- La evaluación del código no debe dar errores en Portacle.
- El fichero con las funciones debe contener **sólo** las funciones, ni casos de prueba ni nada. Cuando se compila y ejecuta el código, este debe simplemente definir las funciones, sin producir ningún resultado.
- Es importante, siempre, poner los nombres de los autores y el número del grupo de prácticas en todos los ficheros que se entragan, memoria y código. **Se penalizarán las entregas sin nombres**.

À Calais ! Allons-y !

El problema de *pathfinder* (descubridor de caminos) es una formalización de problemas en que un agente puede moverse en un espacio (físico o abstracto) para alcanzar la posición de un objetivo. *Pathfinder* puede modelar situaciones tan diferentes como buscar la salida de un laberinto o la demostración de un teorema. Estos problemas se modelan como búsquedas de caminos mínimos en grafos: hay lugares en que los agentes pueden estar (los nodos del grafo), y los agentes se pueden desplazar de un nodo a otro recorriendo las aristas del grafo. Recorrer una arista supone un coste. El problema computacional consiste en encontrar el camino de coste mínimo desde un nodo inicial a uno de una colección de nodos destino.

El problema de la búsqueda de caminos mínimos es uno de los más clásicos de la Inteligencia Artificial clásica[1], y tiene muchas aplicaciones: hay muchos problemas en que un sistema puede estar en un número finito de estados y

[1] La que Hubert Dreyfus llama GOFAI: *Good, Old-Fashioned Artificial Intelligence*.

puede pasar de un estado a otro a través de acciones que tienen un coste. En casi todos estos casos, el problema se puede modelar como un problema de búsqueda.

En esta práctica se propone desarrollar un módulo de IA para viajar a través de Francia hasta llegar a Calais. Podemos imaginar una aplicación que, estemos donde estemos, nos de el mejor recorrido para llegar a Calais en tren minimizando el tiempo de viaje (con un vínculo añadido que veremos en breve). El algoritmo recibe como entrada un mapa de la red de trenes y una heurística que, por cada ciudad de la red estime (por defecto) el tiempo necesario para llegar de esa ciudad a Calais.

Supongamos que tenemos el mapa de ferrocarril de Figura 1, donde a lado de cada línea ferroviaria se indica el tiempo de recorrido. Nuestro mapa se puede representar como un grafo ponderado: los nodos del grafo representan

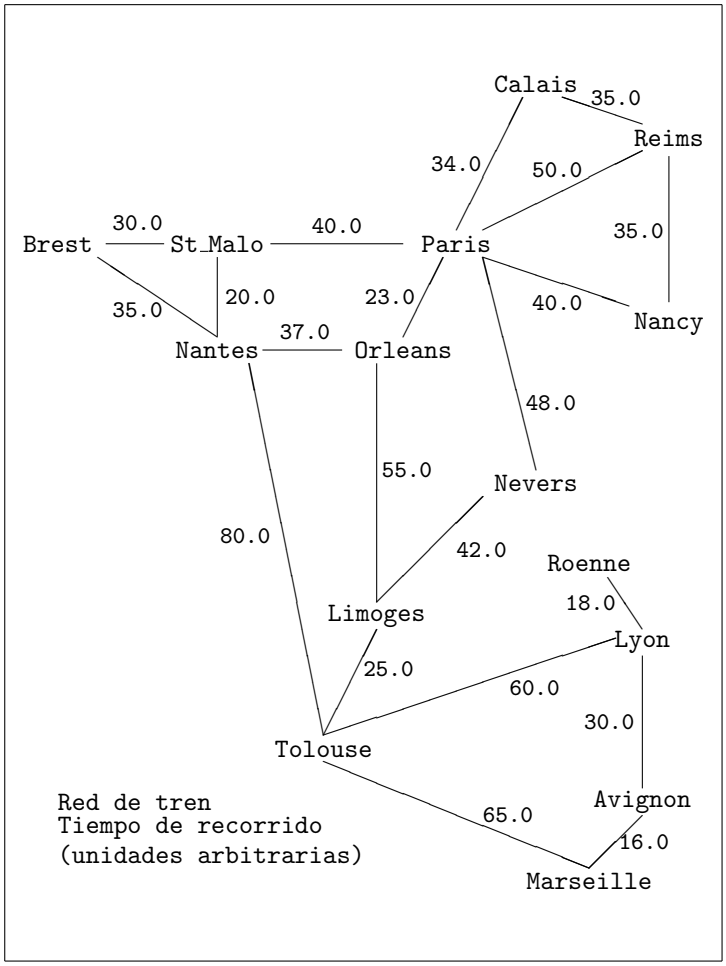

Figure 1: La red de trenes que usaremos para llegar a Calais. Cada nodo representa una ciudad, y las aristas representan conexiones directas entre las ciudades. El número a lado de cada arista representa el tiempo (en unidades arbitraria) de recorrido de ese tramo ferroviario.

las ciudades, una *acción* consiste en recorrer una arista (es decir: ir de una ciudad a otra conectada directamente con la primera sin estaciones intermedias), y el coste de esta acción es el tiempo de recorrido marcado en la línea correspondiente. A primera vista, puede parecer obvio modelar un *estado* del problema simplemente como un nodo

Ciudad	**Tiempo (est.)**
Calais	0
Reims	25
Paris	30
Nancy	50
Orleans	55
St. Malo	65
Nantes	75
Brest	90
Nevers	70
Limoges	100
Roenne	85
Lyon	105
Toulouse	130
Avignon	135
Marseille	145

Figure 2: Tabla con la heurística para el problema. Por cada ciudad se estima (por defecto) el tiempo necesario para llegar a Calais.

del grafo (es decir: la ciudad) donde nos encontramos. Las cosas, en realidad, son un poco más complicadas a causa de una condición añadida que vamos a introducir ahora mismo.

Pues resulta que vuestro amigo Edward (que váis a visitar en Londres—por esto hay que llegar a Calais) os ha pedido que le compréis algunas cosas en ciertas ciudades, por tanto el recorrido para ir a Calais debe obligatoriamente pasar por estas ciudades. Llamaremos a estas ciudades las *ciudades obligatorias*.

La presencia de ciudades obligatoria cambia el problema en dos maneras. Primero, cambia la averiguación de la solución. Para declarar el problema resuelto, no es suficiente averiguar que hemos llegado a un nodo del grafo que representa Calais: hay que averiguar, además, que el camino que hemos recorrido para llegar allí contenga todas las ciudades obligatorias.

El segundo cambio es más sutil, y tiene que ver con la comparación de dos estados del problema (esta comparación es necesaria durante la búsqueda para evitar bucles infinitos en que seguimos analizando estados que ya hemos analizado). Si no tuvieramos las ciudades obligatorias, dos estados serían iguales si, y sólo si, representan la misma ciudad. Con las ciudades obligatorias, dos estados son iguales si cumplen las condiciones siguientes:

1. representan la misma ciudad y
2. en el camino hacia esa ciudad hemos pasado por las mismas ciudades obligatorias.

(¿Por qué? Este es un punto muy importante y es esencial entenderlo para llevar a cabo esta práctica.)

En el modelo de la red de trenes de Figura 1 hay que recordar que los trenes circulan en ambas direcciones: el grafo que usaremos como modelo será, por tanto, un grafo no dirigido. En nuestro diseño el grafo entrará de manera implícita: definiremos un *operador* que, dado un nodo, nos dirá a que nodos nos podemos mover y cuanto nos cuesta (esta información constituirá una *acción*). El operador, claramente, contendrá de alguna manera la representación explícita del grafo, pero es importante notar que el resto del programa navega el grafo sólo a través del operador. De esta manera, si queremos cambiar el grafo en que navegamos será suficiente cambiar la definición del operador, sin tocar el resto del programa.

Además del grafo, los datos del problema incluyen una tabla de heurística con una estimación (por defecto) del tiempo necesario para llegar a Calais desde cualquier ciudad (Figura 2). A diferencia del resto del programa, la tabla de heurísticas es específica para el problema con destino Calais: si el destino cambia, es necesario cambiar la tabla.

Los datos

Los datos del problema se almacenan en una serie de variables globales. Todas estas variables están definidas en el fichero **`p2_IA_2020_test.cl`**.

Ciudades: una lista constante con el nombre de las ciudades de la red:

```
(defparameter *cities*
    '(Brest St-Malo Nantes ...))
```

La red: una lista con las conexiones ferroviarias. Se trata de una lista de triplas: el primer elemento de la tripla es la ciudad origen, el segundo la ciudad destino y el tercero es el tiempo de recorrido del arco. Los arcos no dirigidos se modelan como pares de arcos dirigidos, con el mismo coste en cada dirección.

```
(defparameter *trains*
    '((Paris Calais 34) (Calais Paris 34)
      (Paris Nevers 48) (Nevers Paris 48) ... ))
```

Heurística: una lista constante de pares formados por una ciudad y el valor correspondiente de la heurística

```
(defparameter *heuristic*
    '((Calais 0) (Paris 30) ... ))
```

Ciudad origen: El nodo del grafo donde empieza el recorrido:

```
(defparameter *origin* 'Marseille)
```

Ciudades destino: Una lista constante con los nombres de las ciudades destino. Utilizamos una lista para permitir una fácil generalización del problema: llegar desde la ciudad oringen a cualquiera de las ciudades destinos (todas las funciones que se implementan deben funcionar con una lista de destinos).

```
(defparameter *destination* '(Calais))
```

Ciudades obligadas: lista constante con los nombres de las ciudades por donde es obligatorio pasar para alcanzar la meta

```
(defparameter *mandatory-cities* '(Nantes Paris))
```

El modelo

En esta práctica utilizaremos *estructuras* para crear un modelo del problema. Definiremos cuatro estructuras. Una estructura *problem*, que reúne los datos relativo al problema (el estado inicial, la función para calcular el coste, la función que averigua si hemos encontrado una solución, etc.):

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;    Problem definition
;;
(defstruct problem
  cities               ; List of cities
  initial-city         ; Initial city
  f-h                  ; reference to a function that evaluates the  value
                       ; of the heuristic of a city
  f-goal-test          ; reference to a function that determines whether
                       ; a state fulfills the goal
  f-search-state-equal ; reference to a predicate that determines whether
                       ; two nodes are equal, in terms of their search state
  succ)                ; operator that, given a node, generates the actions
                       ; that can be executed in it
;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

una estructura *node*, que contiene la información necesaria para trabajar con un nodo durante la búsqueda:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;    Node in the search algorithm
;;
(defstruct node
  city         ; city in which this node places us
  parent       ; parent node (in the path that we build)
  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 heuristic
  (f 0))       ; g + h
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

una estructura *action*, que incluye la información que se genera cuando se cruza un arco;

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;    Actions
;;
(defstruct action
```

```
  name         ; Name of the operator that generated the action
  origin       ; City to which the action is applied
  final        ; City in which we are as a result of the application of the action
  cost )       ; Cost of the action
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

y una estructura *strategy*, que contiene, esencialmente, una función de comparación de nodos: es la función que utilizaremos para decidir cuál será el próximo nodo que vamos a explorar:

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

Ejercicios a realizar

1. Modelización del problema

Ejercicio 1 (5%). Evaluación del valor de la heurística.

Codifique una función que calcule el valor de la heurística en la ciudad actual:

```
(defun f-h (city heuristic) ...)
```

Ejemplos de ejecución:

```
(f-h 'Nantes *heuristic*) ;-> 75.0
(f-h 'Marseille *heuristic*) ;-> 145.0
(f-h 'Madrid *heuristic*) ;-> NIL
```

Ejercicio 2 (15%). Definición del operador *navigate*.

El operador recibe dos parámetros: el nombre de una ciudad y la lista de los arcos (ponderados) del grafo. El operador devuelve una lista de acciones: las acciones que se pueden efectuar desde la ciudad especificada.

```
(defun navigate (city edges) ... )
```

Ejemplo de ejecución: véase el fichero **p2_IA_2020_tests.cl**

Ejercicio 3 (7%). Test para determinar si se ha alcanzado el objetivo.

Codifique una función que compruebe si se ha alcanzado el objetivo:

```
(defun f-goal-test (node destination-cities mandatory-cities) ... )
```

La función recibe un *nodo* (¡ojo! un nodo y no simplemente el nombre de una ciudad). El nodo contiene, a través de la cadena de elementos `parent`, la indicación del recorrido que se ha hecho para llegar a él. Un nodo cumple el objetivo si representa *una* de la ciudades de la lista de objetivos y si el camino contiene *todas* las ciudades de la lista de ciudades obligatorias.

Ejemplo de ejecución: véase el fichero **p2_IA_2020_tests.cl**

Ejercicio 4 (8%). Predicado para determinar la igualdad entre estados de búsqueda.

Codifique una función que compruebe si dos nodos son iguales de acuerdo con el estado de la búsqueda. La igualdad si se cumple si dos nodos

- representan la misma ciudad y
- para llegar a esta ciudad se han visitado las mismas ciudades obligatoria.

Para la igualdad no es necesario que las ciudades obligatorias se hayan visitado en el mismo órden. Esta función es necesaria para determinar si un estado de búsqueda es un estado repetido, con el fin de descartarlo en caso en que ya se haya visitado.

```
(defun f-equal-state (node-1 node-2 &optional mandatory-cities)
  ...)
```

Ejemplo de ejecución: véase el fichero **p2_IA_2020_tests.cl**

2. Formalización del Problema

Ejercicio 5 (5%). Representación LISP del problema.

Inicialice una estructura `*travel*` que representa el problema y contiene la información necesaria para su solución:

```
(defparameter *travel*
  (make-problem
    :cities              *cities*
    :initial-city        *origin*
    :f-h                 #'(lambda (city) ... )
```

```
:f-goal-test          #'(lambda (node) ... )
:f-search-state-equal #'(lambda (node-1 node-2) ... )
:succ                 #'(lambda (node) ... )))
```

Ejercicio 6 (10%). Expasión de un nodo.

Codifique la función de expansión de un nodo. Dado un nodo, esta función crea una lista de nodos. Cada elemento de la lista corresponde a un estado que se puede alcanzar directamente desde el estado del nodo dado.

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

El resultado tiene que ser una lista de **nodos**.

(Nota: los nodos de la lista que la función devuelve tienen el nodo `node` como `parent`.)

Ejemplo de ejecución: véase el fichero **`p2_IA_2020_tests.cl`**

Ejercicio 7 (10%). Gestión de nodos

Escriba una función que inserte los nodos de una lista en otra lista de manera que la segunda lista esté ordenada respeto al criterio de comparación de la estrategia dada. Se supone que la lista en que se insertan los nodos (lista `lst-nodes` en la declaración abajo) ya esté ordenada respeto al criterio deseado, mientras la lista que se inserta (lista `nodes`) puede tener cualquier orden.

```
(defun insert-nodes-strategy (node lst-nodes strategy) ...)
```

Ejemplo de ejecución: véase el fichero **`p2_IA_2020_tests.cl`**

3. Búsquedas

Ejercicio 8 (5%). Definir la estrategia para la búsqueda A*.

Inicialice una variable global cuyo valor sea la estrategia para realizar la búsqueda A*.

```
(defparameter *A-star*
  (make-strategy ...))
```

Ejemplo: estrategia para la bśqueda de coste uniforme.

```
(defparameter *uniform-cost*
  (make-strategy
    :name            'uniform-cost
    :node-compare-p  #'g-leq))
```

```
(defun g-leq (node-1 node-2)
  (<=  (node-g node-1)
       (node-g node-2)))
```

Ejercicio 9 (20%). Función de búsqueda.

Codifique la función de búsqueda según el siguiente pseudo-código. La función utiliza los siguientes parámetros:

open	lista de nodos generados pero no explorados;
closed	lista de nodos generados y explorados;
strategy	extrategia de búsqueda implementada (de la estrategia depende la ordenación de los nodos en la lista **open-nodes**;
goal-test	test de objetivo alcanzado (predicado que evalúa a `T` si un nodo cumple la condición de ser meta

```
; Solves the given search problem using a given strategy
; Evaluates to:
;   No solution --> NIL
;   There is a solution --> a node that satisfies the objective
;
(defun graph-search (problem strategy) ... )
```

Pseudo-código:

```
1.   Inicializa la lista open-nodes con el estado inicial
2.   Inicializa la lista closed-nodes como NIL
3.   repeat
4.     if open-nodes es '()
5.       terminar [No hay solucion]
6.     extrae el primer nodo de open-nodes (nodo n)
7.     if goal-test es T para este nodo
8.       evalua la solucion y termina
9.     else
10.      if exp-cond(n)
11.        expande el nodo n
12.        inserta todos los nodos generados en open-nodes
13.        inserta n en la lista closed-nodes
14.        elimina n de la lista open-nodes
15.      fi
16.    fi
17.  end
```

La condición de le línea 10 determina si hay que expandir el nodo que se está visitando. La condición es la siguiente:

```
exp-cond(n)
```

1. `n not in` **`closed-nodes`**
2. `or`
3. `n tiene un coste inferior a la copia de n que esta en` **`closed-nodes`**

Ejemplo:

```
(graph-search *travel* *A-star*) ;-> (vease el fichero p2_IA_2020_tests.cl)
```

A partir de esta función, codifique

```
;
; Solve a problem using the A* strategy
;
(defun a-star-search (problem) ...)
```

Ejemplo de ejecución: véase el fichero **`p2_IA_2020_tests.cl`**

Ejercicio 10 (5%). Ver el camino seguido y la sequencia de acciones.

A partir de un nodo que es el resultado de una búsqueda, codifique:

- una función que muestre el camino seguido para llegar a un nodo. La función muestrará los estados (es decir, el nombre de las ciudades) que se han visitado para llegar a un nodo dado.

  ```
  (defun solution-path (node) ... )
  ```

  Ejemplos:

  ```
  (solution-path NIL)  ;-> NIL
  (solution-path (a-star-search *travel*)) ;->
  ;  (MARSEILLE TOULOUSE LIMOGES ORLEANS PARIS CALAIS)
  ```

- una función que muestra la secuencia de acciones para llegar a un nodo

  ```
  (defun action-sequence (node) ... )
  ```

Ejemplo de ejecución: véase el fichero **`p2_IA_2020_tests.cl`**

Ejercicio 11 (5%). Otras estrategias de búsqueda.

Diseñe estrategias para realizar búsqueda en profundidad y en anchura:

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

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

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

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

Ejercicio 12 (5%). Heurística de coste

En todos los problemas hasta ahora hemos usado la heurística de coste dada al principio de este enunciado. No se trata, claramente, de la única posible. En este apartado se pide:

1. Crear una heurística válida de coste (la heurística debe en cualquier caso subestimar el coste real). Definir el parametro *heuristic-new* con esta heurística.
2. Definir la "heurística cero", que da una estimación de cero por cada ciudad. Se trata de la heurística más conservadora posible: el algoritmo funciona y da el resultado correcto, pero hace una exploración sistemática del grafo que no es muy eficiente. Definir el parametro *heuristic-cero* para esta heurística.
3. Usar la función tiempo de LISP para medir el tiempo de ejecución del algoritmo con las tres heurísticas (las dos definidas aquí y la heurística dada).

Reportar la heurística creada y los tiempos de ejecución de las tres en la memoria y comentar los resultados. **Las heurísticas que se han creado no se deben entregar en el fichero de código.**

Memoria

En la memoria, incluir la respuesta a las siguientes preguntas.

1. ¿Por qué se realizado este diseño para resolver el problema de búsqueda? En concreto:
   a. ¿Qué ventajas aporta?
   b. ¿Por qué se han utilizado funciones lambda para especificar el test objetivo, la heurística y los operadores del problema?
2. 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 la memoria?
3. ¿Cuál es la complejidad espacial del algoritmo implementado?
4. ¿Cuál es la complejidad temporal del algoritmo implementado?

Protocolo de Corrección

[40%] Corrección automática (ejercicios 1–10)

La corrección de la práctica se realizará utilizando distintos problemas de búsqueda y distintas redes, diferentes de la del ejemplo del enunciado, por lo que se recomienda definir una batería de pruebas con distintos problemas.

[30%] Estilo

[30%] Memoria (incluyendo la respuesta a las preguntas de arriba.