PRACTICA 2. BÚSQUEDA

Fecha de publicación: 2019/03/06
Fechas de entrega: [grupos jueves: miércoles 2019/03/28]
[grupos viernes: jueves 2019/03/29]
Planificación: semana 1: Ejercicios 1, 2, 3, 4, 5, 6
semana 2: Ejercicios 7, 8, 9, 10
semana 3: Ejercicios 11 + memoria Versión: 2019/03/08

IMPORTANTE:
- **Utilizad la versión Allegro CL 6.2 ANSI with IDE y COMPILAD el código.**
- **DEFINID un problema de búsqueda más simple 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 en el fichero no debe dar errores en Allegro CL 6.2 ANSI con IDE (recuerda: CTRL+A CTRL+E debe evaluar todo el código sin errores).
- El fichero con las funciones debe contener **sólo** las funciones, ni casos de prueba ni nada. Cuando se ejecuta con CTRL+A CTRL+E debe simplemente definir las funciones.
- Es importante, siempre, poner los nombres de los autores y el número del grupo de prácticas en todos los ficheros que entregáis: memoria y código.

Material recomendado:
- En cuanto a estilo de programación LISP: http://www.cs.cmu.edu/Groups/AI/html/faqs/lang/lisp/part1/faq.html
- Como referencia de funciones LISP: http://www.lispworks.com/documentation/common-lisp.html

À Calais ! Allons-y !

El problema de 'pathfinder' ('descubridor de caminos') es común situaciones en que un agente puede moverse en un espacio para alcanzar la posición de un objetivo. Por ejemplo, encontrar la salida de un laberinto. Estos problemas se modelan como búsquedas de caminos mínimos en grafos: hay lugares en que los agentes pueden estar (los nodos del grafo). 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 tiene muchas aplicaciones más allá que la búsqueda de recorridos en espacios físicos: muchos problemas en que un sistema puede estar en un número finito de estados, se puede pasar de un estado a otro a través de acciones que tienen un coste, y el problema consiste en llegar a un estado determinado con el mínimo coste, se puede modelar como problema de búsqueda.

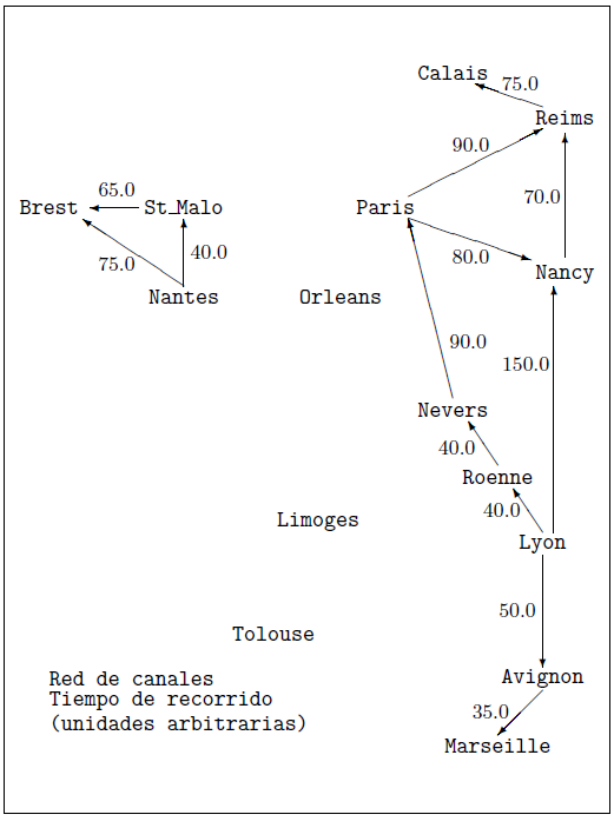
En esta práctica se propone desarrollar un módulo IA para viajar a través de Francia hasta llegar a Calais. Podemos imaginar una aplicación que se ejecuta en el móvil y que nos dará el mejor recorrido y medio de transporte para llegar a Calais optimizando el criterio que especificaremos: tiempo o coste. Hay dos maneras de moverse por Francia. La primera es el tren: el tren es muy rápido y cada línea de ferrocarril entre dos ciudades es bidireccional, es decir, se puede recorrer en los dos sentidos. El tren es también el medio de transporte más caro. Otra opción es navegar los canales que conectan varias ciudades de Francia. Los canales son más lentos que el tren y, dado que navegamos siempre en barcazas en favor de corriente, son unidireccionales: se puede ir de una ciudad A a una ciudad B pero no al revés. Los canales son también más baratos que el tren.

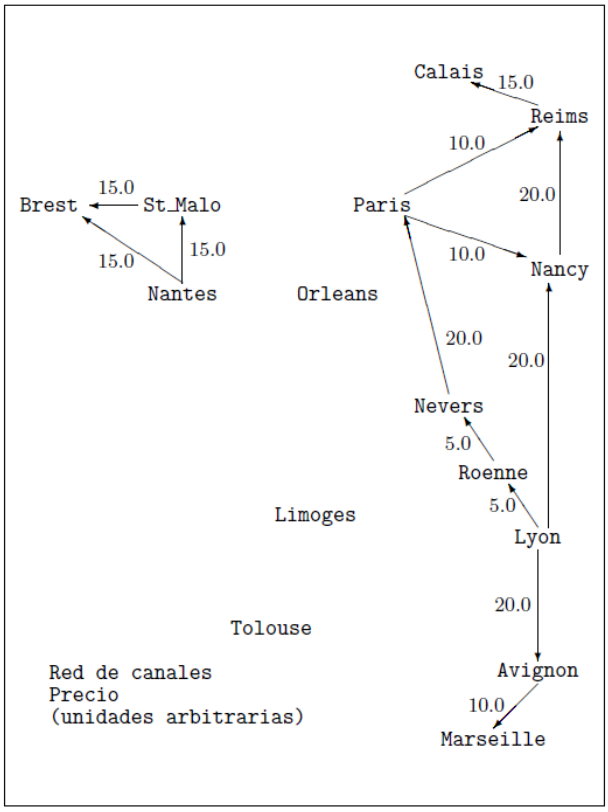
La aplicación debe, dado el lugar en que estamos, encontrar la combinación de ciudades y medios de transporte entre ellas (canal o tren) para optimizar uno de dos criterios posibles: coste o tiempo. El algoritmo recibe dos heurísticas distintas (una para el tiempo, otra para el precio) que estiman el coste de llegar de cada ciudad a Calais. En cada movimiento hay que desplazarse de una ciudad a otra conectada a ella usando uno de los dos

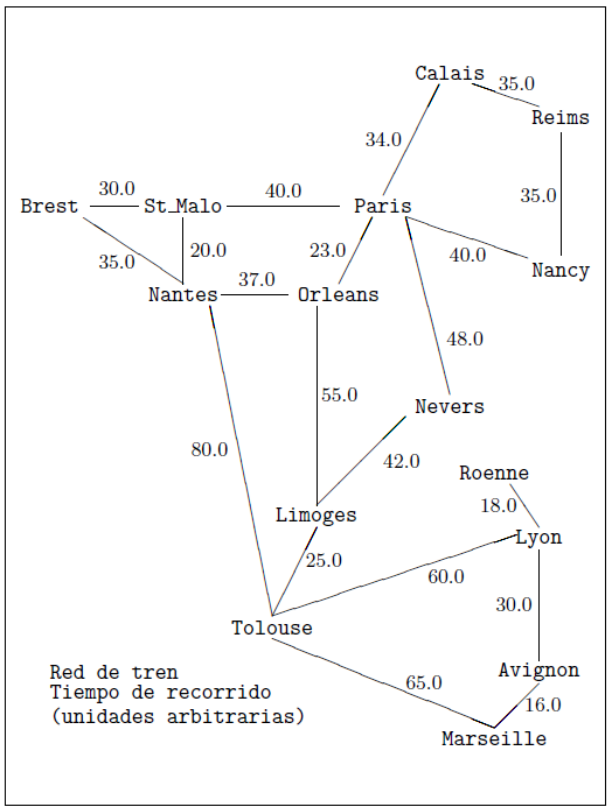
medios disponibles (canal o tren) hasta llegar a Calais

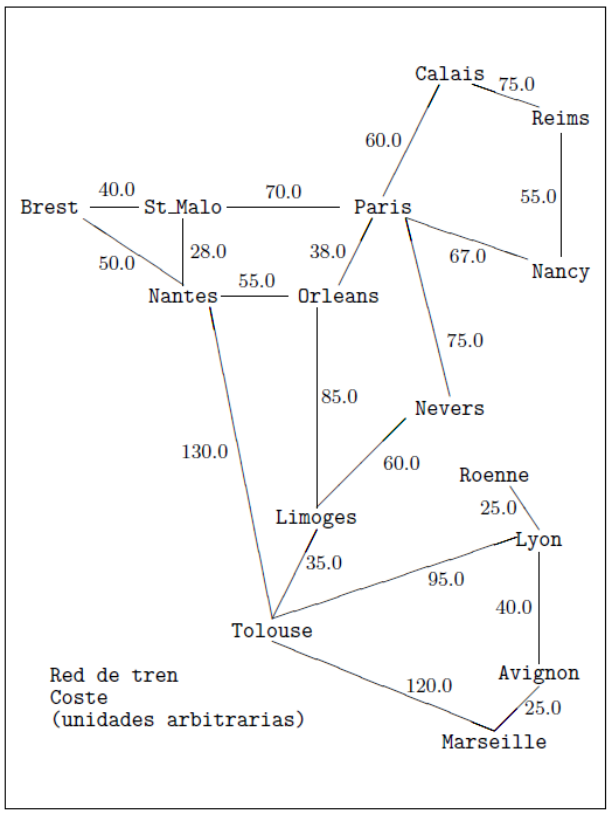
En algunas ciudades, la estación de trenes está cerrada por obras, por tanto no se puede llegar allí en tren (si la ciudad posee un canal, sí se puede llegar a través de un canal): llamaremos a estas las "ciudades prohibidas"; además, vuestro amigo Edward (que vais a visitar en Londres--por esto hay 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 las "ciudades obligatorias". El algoritmo de búsquedas recibirá por tanto dos listas (que pueden ser vacías): una de ciudades prohibidas, la otra de ciudades obligatorias.

A continuación presentamos parte de la red ferroviaria y de canales de Francia. Cada red se modela como un grafo ponderado, donde cada arco tiene dos pesos distintos (aquí lo mostramos en dos figuras distintas), correspondientes al tiempo de recorrido y al coste de cada tramo. La red ferroviaria es un grafo no dirigido, mientras la red de canales es un grafo dirigido.









Trataremos los dos tipos de arcos (tren y canal) como dos entidades independiente: en nuestro modelo habrá *operadores* que, dado un nodo, nos dirán en que nodos podemos ir. Habrá dos operadores distintos, uno para trenes y uno para canales.

Además de los grafos, el modelo incluye dos tablas de heurísticas: la primera con una estimación del tiempo necesario para llegar a Calais desde cualquier ciudad de la red, la segunda con la estimación de cuanto nos costará. Las dos tablas se representan a continuación:

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

El Modelo

El modelo del grafo se almacena en una serie de variables globales:

- **Ciudades**: Lista constante de símbolos representando los nombres de las ciudades
  ```
  (defparameter *cities* '(Brest St-Malo Nantes ...))
  ```

- Las conexiones ferroviarias y por canales. Cada una es una lista de triplas: el primer elemento de la tripla es la ciudad origen, el segundo es la ciudad destino, y el tercero es una lista con dos elementos: el tiempo de recorrido y el coste. Los arcos no dirigidos de la red de trenes se modelan como pares de arcos con el mismo coste, uno por cada dirección:

  ```
  (defparameter *canals*
    '((Paris Calais (50 20)) (Paris Nevers (70 30))…))

   (defparameter *train*
    '((Paris Calais (34 60))   (Calais Paris (34 60))
       ((Paris Nevers (48 75)) (Nevers Paris (48 75))...)
  ```

- Las **estimaciones** (*heurística*): Lista constante de pares formados por una ciudad y dos correspondientes valores de las heurísticas (tiempo de recorrido y coste)

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

- **Ciudad origen**: El estado inicial será el correspondiente a estar en la ciudad donde empieza el viaje.

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

- **Ciudad destino**: Lista constante de símbolos representando los nombres de las ciudades destino.

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

- **Ciudades prohibidas**: Lista constante de símbolos con los nombres de las ciudades a que no está permitido llegar en tren debido a obra en la estación.

  ```
  (defparameter *forbidden-cities* '(Nevers))
  ```

- **Ciudades obligadas**: Lista constante de símbolos con los nombres de las ciudades por donde es obligatorio pasar para alcanzar la meta

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

Estructuras

En esta práctica utilizaremos estructuras. Definiremos cuatro estructuras que permiten representar los elementos de nuestro programa. Una estructura “problem”, que reúne los datos relativos al problema (el estado inicial, la función para calcular el coste f, etc,); una estructura “node”, que contiene la información necesaria para trabajar con un nodo durante la búsqueda; una estructura “action”, que incluye la información que se genera cuando se cruza un arco, 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.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;    Problem definition
;;
(defstruct problem
  states               ; List of states
  initial-state        ; Initial state
  f-h                  ; function that evaluates the  value of the
                       ; heuristic of a state (either cost or time)
  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
  operators)           ; list of operators (references to functions) to
                       ; generate successors
;;
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;    Node in the search algorithm
;;
(defstruct node
     state         ; state label
     parent        ; parent node (in the paths 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
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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

Ejercicios a realizar

1. Modelización del problema

EJERCICIO 1 (5 %.): Evaluación del valor de la heurística. Codifique unas funciones que calculen el valor de las heurísticas de precio o tiempo (una función por cada heurística) en el estado actual:

```
(defun f-h-time (state heuristic)...)

(defun f-h-price (state heuristic)...)
```

Ejemplos de ejecución

```
(f-h-time 'Nantes *estimate*) ;-> 75.0
(f-h-time 'Marseille *estimate*) ;-> 145.0
(f-h-price 'Lyon *estimate*) ;-> 0.0
(f-h-price 'Madrid *estimate) ;-> NIL
```

EJERCICIO 2 (15 %): Operadores `navigate-canal-time, navigate-canal-price, navigate-train-time` y `navigate-train-price.`

Los operadores son funciones que, dado un estado (es decir, el nombre de una ciudad), devuelven una lista de las acciones que se pueden efectuar a partir de ese estado.

En nuestro sistema hay cuatro operadores:
1. a que ciudades se puede llegar usando el tren; devuelve las acciones con el correspondiente *tiempo de recorrido*
2. a que ciudades se puede llegar usando el tren; devuelve las acciones con el correspondiente *precio de recorrido*
3. a que ciudades se puede llegar usando un canal; devuelve las acciones con el correspondiente *tiempo de recorrido*
4. a que ciudades se puede llegar usando un canal; devuelve las acciones con el correspondiente *precio del recorrido*

Hay que recordar que en algunas ciudades no se puede llegar en tren, por tanto los operadores correspondientes reciben una lista de ciudades prohibidas.

```
(defun navigate-canal-time (state canals ) ...)
(defun navigate-canal-price (state canals ) ...)
(defun navigate-train-time (state trains forbidden-cities)...)
(defun navigate-train-price (state trains forbidden-cities)...)
```

Ejemplos de ejecución: véase el fichero **`p2_IA-2019_tests.cl`**.

Sugerencia: las funciones son muy parecidas. Es conveniente definir una función `navigate,` de carácter general y cuatro interfaces que le pasan los parámetros que necesita.

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

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

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

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 las ciudades objetivo y si el camino contiene *todas* las ciudades obligatorias.

Ejemplos de ejecución: véase el fichero **`p2_IA-2019_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 su estado de búsqueda.
Es decir si se cumple que los dos nodos:

(i) corresponden a la misma ciudad
(ii) la lista de ciudades de entre las obligadas que aún han de visitar coincide.

Esta función es necesaria para determinar si un estado de búsqueda es un estado repetido, con el fin de descartarlo en caso de que se demuestre que no puede conducir a la solución óptima.

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

Ejemplos de ejecución: véase el fichero **`p2_IA-2019_tests.cl`**.

2. Formalización del problema

EJERCICIO 5 (5%)**:** Representación LISP del problema.

Inicialice el valor de dos estructuras, llamadas *travel-fast* y *travel-cheap* que representan los problemas de tiempo mínimo y precio mínimo, respectivamente. Cada estructura problema contiene sólo una función de heurística (relativa a precio o tiempo) y dos operadores (el operador "canal" y el operador "tren" relativos al criterios que se quiere minimizar).

```
(defparameter *travel-cheap*
  (make-problem
   :states               *cities*
   :initial-state        *origin*
   :f-h                  #'(lambda (state) ...)
   :f-goal-test          #'(lambda (node) ...)
   :f-search-state-equal #'(lambda (node-1 node-2)...)
   :operators            (list
                          #'(lambda (node)
                              ...)
                             ...)))

(defparameter *travel-fast*
  (make-problem
   :states               *cities*
   :initial-state        *origin*
   :f-h                  #'(lambda (state) ...)
   :f-goal-test          #'(lambda (node) ...)
   :f-search-state-equal #'(lambda (node-1 node-2)...)
   :operators            (list
                          #'(lambda (node)
                              ...)
                             ...)))
```

EJERCICIO 6 (10%): Expandir nodo.

Codifique la función de expansión de 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.

Nota: el problema contiene una lista de operadores, y hay que considerar los estados que se pueden alcanzar usando todos estos operadores. Cada operador devuelve una lista de acciones. Hay que construir los nodos que se pueden generar usando estas acciones.

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

El resultado debe ser una **lista de nodos.**

Ejemplos de ejecución: véase el fichero **`p2_IA-2019_tests.cl`**.

EJERCICIO 7 (10%): Gestión de nodos.

Escriba 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.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  Insert a list of nodes into another list of nodes
;;;
;;;
(defun insert-nodes-strategy (nodes lst-nodes strategy)...)
```

Se supone que la lista `lst-nodes` está ordenada de acuerdo a dicha estrategia. La lista de nodos `nodes` no tiene por qué tener una ordenación especial.

Ejemplos de ejecución: véase el fichero **`p2_IA-2019_tests.cl`**.

3. Búsquedas

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

Inicializa una variable global cuyo valor sea la estrategia para realizar la búsqueda A*:

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

Ejemplo: Estrategia para búsqueda de coste uniforme:

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

(defun node-g-<= (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 pseudocódigo:

```
open: lista de nodos generados, pero no explorados
closed: lista de nodos generados y explorados
strategy: estrategia de búsqueda implementada como una ordenación de la
lista open-nodes
goal-test: test objetivo (predicado que evalúa a T si un nodo cumple la
condición de ser meta)

; Realiza la búsqueda para el problema dado utilizando una estrategia
; Evalúa:
;    Si no hay solución: NIL
;    Si hay solución: un nodo que cumple el test objetivo  ;

(defun graph-search (problem strategy)
     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.

Ejemplo:

(graph-search *travel-cheap* *A-star*);-> (véase el fichero de pruebas)
```

A partir de esta función, codifique

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

Ejemplos de ejecución: véase el fichero **p2_IA-2019_tests.cl**.

EJERCICIO 10 (5%): Ver el camino seguido y la secuencia 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 mostrará los estados (es decir, el nombre de las ciudades) que se han visitado para llegar al nodo dado.

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

Ejemplos:

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

(solution-path (a-star-search *travel-cheap*)) ; ->
; (MARSEILLE TOULOUSE LIMOGES NEVERS PARIS REIMS CALAIS)
```

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

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

Ejemplos de ejecución: véase el fichero **`p2_IA-2019_tests.cl`**.

EJERCICIO 11 (5%): Otras estrategias de búsqueda.

Diseñe una estrategia para realizar búsqueda en profundidad:

```
(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)
     <codigo LISP para depth-first>)

(solution-path (graph-search *travel-fast* *depth-first*))
```

Diseñe una estrategia para realizar búsqueda en anchura:

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

(defun breadth-first-node-compare-p (node-1 node-2)
     <codigo LISP para breadth-first>)

(solution-path (graph-search *travel-cheap* *breadth-first*))
```

EJERCICIO 12 (5%**): Heurística de coste**

En todos los problemas hasta ahora hemos usado una heurística de coste igual a cero para todas las ciudades. Se trata de la heurística más conservadora posible: el algoritmo funciona, pero hace una exploración sistemática del grafo que no es especialmente eficiente. En este apartado se pide:

1. crear una heurística válida de coste (la heurística debe en cualquier caso subestimar el coste real)
2. usar las funciones de tiempo de ejecución del LISP para determinar si la nueva heurística es más eficiente que la original.

Más específicamente:

1. definir una nueva lista llamada *estimate-new* que tenga la misma heurística para los tiempos de recorrido y la nueva heurística para los costes;
2. definir una nueva estructura de problema *travel-cost-new* que defina el problema de coste mínimo con la nueva heurística;
3. usando las funciones del LISP, medir el tiempo de ejecución del problema de coste mínimo con la heurística original y con la nueva.

La nueva heurística y el nuevo problema **no deben estar en el fichero de código que vais a entregar**.

En la memoria hay que poner la heurística, la estructura del problema y los tiempos de ejecución con la vieja y la nueva heurística.

En la memoria se debe incluir respuestas a las siguientes preguntas

1. ¿Por qué se ha realizado este diseño para resolver el problema de búsqueda?
2. 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?
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?
4. ¿Cuál es la complejidad espacial del algoritmo implementado?
5. ¿Cuál es la complejidad temporal del algoritmo?
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” (bidireccionales).

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 las respuestas a las preguntas 1-6 aquí arriba)