Práctica 2: Búsqueda

Introducción

En esta práctica abordaremos los distintos tipos de búsqueda automática. Para ello implementaremos el algoritmo de búsqueda A* para resolver nuestro problema de concreto entre ciudades de Francia.

1. Modelización del problema

Ejercicio 1:

Código:

```
(defun f-h (city heuristic)
  (second (assoc city heuristic)))
```

En este ejercicio definiremos la función que evaluará el valor de la heuristica asociado a cada nodo. Para ello buscaremos en la tabla de heurística el valor asociado a la ciudad solicitada.

Ejercicio 2:

Código:

```
(defun navigate (city lst-edges)
  (remove-if #'null
    (mapcar #'(lambda (edge)
      (let ((origin (car edge)) (dest (second edge)) (cost (third edge)))
        (if (equal origin city)
          (make-action
            :name   (format NIL "Tren de ~A a ~A" origin dest)
            :origin origin
            :final  dest
            :cost cost))))
```

```
      lst-edges)))


(defun navigate2 (city lst-edges)
  (navigate-aux city lst-edges NIL))


(defun navigate-aux (city lst-edges res)
  (if (null lst-edges)
    res
    (let* ((edge (car lst-edges))
           (origin (car edge))
           (dest (second edge))
           (cost (third edge)))
      (if (equal origin city)
        (navigate-aux
          city
          (cdr lst-edges)
          (cons
            (make-action
              :name   (format NIL "Tren de ~A a ~A" origin dest)
              :origin origin
              :final  dest
              :cost   cost)
            res))
        (navigate-aux city (cdr lst-edges) res)))))
```

En este ejercicio se da una ciudad y la lista de caminos del grafo y se pide una lista de acciones que puedan realizar desde la ciudad dada. Para ello se mira en la lista de caminos aquellos cuya ciudad origen sea la que se recibe y crea el objeto acción con la ciudad destino y el coste. La preimera versión y uqe devuelve la salida que se espera en los tests lo hace con un mapcar sobre la lista de caminos y luego quita los NIL (caminos que no parten de la ciudad dada). El problema de esta implementación es que se recorre dos veces la lista, una para comprobar si es una acción válida, y otra para quitar las que no lo son. Por ello, hemos realizado una alternativa más eficiente, pero que la salida que genera está en orden inverso. Esta utiliza la recursión de forma que solo se tiene que recorrer una vez la lista, por lo uqe debería ser más eficiente y disminuir el tiempo de ejecución. Al comprobar los resultados, se vio que el tiempo de ejecución de ambas fucniones era prácticamente el mismo, e incluso algúnas veces la segunda implementación algo mayor, pero descubríamos que en cambio esta si que usaba la

memoria de manera más eficiente, así que sigue siendo una pequeña victoria. A continuación se muestran los resultados:

Break 1 [3]> (time (navigate 'Avignon *trains*))

Real time: 3.55E-4 sec.

Run time: 3.55E-4 sec.

Space: 4952 Bytes

(#S(ACTION :NAME "Tren de AVIGNON a LYON" :ORIGIN AVIGNON :FINAL LYON :COST 30.0)

#S(ACTION :NAME "Tren de AVIGNON a MARSEILLE" :ORIGIN AVIGNON :FINAL MARSEILLE :COST 16.0))

Break 1 [3]> (time (navigate2 'Avignon *trains*))

Real time: 3.49E-4 sec.

Run time: 3.48E-4 sec.

Space: 3856 Bytes

(#S(ACTION :NAME "Tren de AVIGNON a MARSEILLE" :ORIGIN AVIGNON :FINAL MARSEILLE :COST 16.0)

#S(ACTION :NAME "Tren de AVIGNON a LYON" :ORIGIN AVIGNON :FINAL LYON :COST 30.0))

Ejercicio 3:

Código:

```
(defun f-goal-test (node destination mandatory)
  (if (member (node-city node) destination)
    (mandatory-check node mandatory)))


(defun mandatory-check (node mandatory)
  (cond
    ((null mandatory)
      t)
    ((mandatory-node-check node (car mandatory))
      (mandatory-check node (cdr mandatory)))))
```

```
(defun mandatory-node-check (node city)
  (cond
    ((null node)
      NIL)
    ((equal (node-city node) city)
      t)
    (t
      (mandatory-node-check (node-parent node) city))))
```

En este ejercicio, implementamos la función que nos decidirá si un nodo es la solucíon del problema. Para ello se tendrá que comprobar que dicho nodo esté en las posibles ciudades destino, y que en el recorrido realizado, haya pasado por todas las ciudades obligatorias. Para ello usaremos las funciones auxiliares *mandatory-check* y *mandatory-node-check* que irán recorriendo tanto los padres del nodo como las ciudades obligatorias, y devolverá *true* si ha pasado por todas ellas o *nil* de lo contrario.

Ejercicio 4:

Código:

```
(defun f-search-state-equal (node-1 node-2 &optional mandatory)
  (if (equal (node-city node-1) (node-city node-2))
    (notany
      #'(lambda (city)
        (xor
          (mandatory-node-check node-1 city)
          (mandatory-node-check node-2 city)))
      mandatory)))
```

Para este ejercicio, se nos pedía implementar una función que evaluara dos nodos y devolviera *true* si son iguales o *nil* de lo contrario. Para decidir si dos nodos son el mismo, comprobaremos que coincida el nombre de la ciudad, y que en el camino recorrido ambos hayan pasado por las mismas ciudades obligatorias. Para ello usamos la función implementada anteriormente: *mandatory-node-check.* El funcionamiento de la función puede parecer complicado a simple vista y quizás no sea el más eficiente pero es el mejor que se nos ocurrió. La función itera con un *mapcar* sobre la lista de ciudades

obligatorias. Con cada una, comprueba con *mandatory-node-check* si los caminos que conforman los dos nodos han pasado por dicha ciudad, y se hace una xor. Esta xor, lo que hace es devolver NIL is los dos han visitado o no la ciudad, y T si la han visitado los dos. Esta lista se pasa a la función notany, que en caso de que haya algún T, es decir, que alguna ciudad difiera de ambos nodos, devolverá NIL y si no, devolverá T.

2. Formalización del Problema

Ejercicio 5:

Código:

```
(defun succ (node lst-edges)
  (navigate (node-city node) lst-edges))


(defparameter *travel*
  (make-problem
    :cities              *cities*
    :initial-city        *origin*
    :f-h                 #'(lambda(c) (f-h c *heuristic*))
    :f-goal-test         #'(lambda(n) (f-goal-test n *destination* *mandatory*))
    :f-search-state-equal #'(lambda(n1 n2) (f-search-state-equal n1 n2 *mandatory*))
    :succ                #'(lambda(n) (succ n *trains*))))
```

En este ejercicio se define una función que haya los sucesores de un nodo, al que se le pasa la lista de conexiones y el nodo que se va a explorar. Tambien definiremos el problema y lo guardaremos en la variable **travel**. Aqui guardaremos todo lo necesario para resolver el problema, las ciudades, el origen, la dunción de heuristica, la de solución, la que comprueba la igualdad de nodos, y la que obtiene los sucesores, todas ellas implementadas anteriormente.

Hemos reducido el numero de argumentos necesarios para solo tener que pasar los nodos o las ciudades. Se explica en las cuestiones la razon de esta decisión.

Ejercicio 6:

Código:

```
(defun expand-node-action (node f-h action)
  (let* ((new-city (action-final action))
```

```
        (new-depth (+ (node-depth node) 1))
        (g (+ (action-cost action) (node-g node)))
        (h (funcall f-h new-city))
        (f (+ g h))
        (new-node
          (make-node
            :city     new-city
            :parent   node
            :action   action
            :depth    new-depth
            :g        g
            :h        h
            :f        f)))
    (if (null (node-parent node))
      new-node
      (if (equal (node-city new-node) (node-city (node-parent node)))
        NIL
        new-node))))

(defun expand-node (node problem)
  (let ((lst-actions (funcall (problem-succ problem) node))
        (f-h (problem-f-h problem)))
    (remove-if #'null
      (mapcar
        #'(lambda (action) (expand-node-action node f-h action))
        lst-actions))))

(defun expand-node-action2 (node f-h lst-actions ret)
  (if (null lst-actions)
    ret
    (let* ((action (car lst-actions))
           (new-city (action-final action))
           (new-depth (+ (node-depth node) 1))
           (g (+ (action-cost action) (node-g node)))
```

```
        (h (funcall f-h new-city))
        (f (+ g h))
        (new-node
          (make-node
            :city     new-city
            :parent   node
            :action   action
            :depth    new-depth
            :g        g
            :h        h
            :f        f)))
    (if (null (node-parent node))
      (expand-node-action2 node f-h (cdr lst-actions) (cons new-node ret))
      (if (equal (node-city new-node) (node-city (node-parent node)))
        (expand-node-action2 node f-h (cdr lst-actions) ret)
        (expand-node-action2 node f-h (cdr lst-actions) (cons new-node ret)))))))

(defun expand-node2 (node problem)
  (let ((lst-actions (funcall (problem-succ problem) node))
        (f-h (problem-f-h problem)))
    (expand-node-action2 node f-h lst-actions NIL)))
```

En este ejercicio implementaremos la función para expandir un nodo. Para hacerlo, en primer lugar se hayan las acciones posibles a realizar desde el nodo a través de la función *succ* y la función heurística del problema. Con estos datos, se hace un *mapcar* que llama a la función *expand-node-action* con cada acción, lo que crea un nodo con los datos del nodo padre y la acción. Para evitar bucles, si el nodo padre del que se está explorando es el mismo que el destino de la acción, este no se añade a la lista y devuelve NIL. Para quitar estos NIL, se llama después a la función *remove-if*.

Al igual que con la función navigate, también hemos hecho una función que evita recorrer la lista dos veces de manera muy similar.

Ejercicio 7:

Código:

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

```
  (if (null lst-nodes)
    (list node)
    (let ((first-node (car lst-nodes)))
      (if (funcall node-compare-p node first-node)
        (cons node lst-nodes)
        (cons first-node (insert-node node (cdr lst-nodes) node-compare-p))))))

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

En este ejercicio consiste en una función que inserta una lista de nodos en otra en la que están ordenados. Para ello, esta función primero extrae la función de comparación y llama a *insert-nodes*, que llama a su vez a la función *insert-node*. Esta función va insertando nodo a nodo los de la lista haciendo una búsqueda lineal sobre la lista ordenada.

Ejercicio 8:

Código:

```
(defun A-star-node-compare-p(node-1 node-2)
  (if (= (node-f node-1) (node-f node-2))
    (<= (node-depth node-1) (node-depth node-2))
    (< (node-f node-1) (node-f node-2))))

(defparameter *A-star*
  (make-strategy
    :name 'A-star
    :node-compare-p #'A-star-node-compare-p))
```

En este ejercicio, se nos pide definir la estrategia de búsqueda A*. Esta estrategia usa para comparar dos nodos, aquel que tenga una f menor. En caso de coincidencia entre varios nodos se queda con el nodo que tenga menor profundidad.

Ejercicio 9:

Código:

```
(defun graph-search (problem strategy)
  (let* ((node-ini (make-node
                :city (problem-initial-city problem)
                :h (funcall(problem-f-h problem) (problem-initial-city problem))))
         (open-nodes (insert-nodes-strategy (expand-node node-ini problem) '() strategy))
         (closed-nodes '()))
    (if (null open-nodes)
      NIL
      (graph-search-aux problem strategy open-nodes (list node-ini)))))

(defun graph-search-aux (problem strategy open-nodes closed-nodes)
  (let ((node-aux (first open-nodes)))
    (if (funcall (problem-f-goal-test problem) node-aux)
      node-aux
      (if (exp-condition problem node-aux closed-nodes)
        (graph-search-aux problem strategy (insert-nodes-strategy (expand-node node-aux problem) (rest open-nodes) strategy) (cons node-aux closed-nodes))
        (graph-search-aux problem strategy (rest open-nodes) closed-nodes )))))

(defun node-in-list(problem node closed-nodes)
  (if (null closed-nodes)
    NIL
    (if (funcall(problem-f-search-state-equal problem) node (first closed-nodes))
      (first closed-nodes)
      (node-in-list problem node (rest closed-nodes)))))

(defun exp-condition(problem node closed-nodes)
```

```
 (let ((node-copy (node-in-list problem node closed-nodes)))
   (or (null node-copy) (< (node-g node) (node-g node-copy)))))


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

En este ejercicio, implementaremos la función de búsqueda en el grafo *graph-search* . Ésta recibirá como argumentos el problema y la estrategia a seguir para resolver el problema. Para ello se definirá node-ini como el nodo en el que se comienza la búsqueda obtenido de la ciudad inicial de problem. Se definirá la lista de open-nodes como los nodos resultantes al expandir node-ini, y closed-nodes como una lista vacia.

A continuación, llamaremos a *graph-search-aux* pasandole la lista de open-nodes y closed-nodes. Esta función extraera el primer nodo de open-nodes, y si es la solución (lo comprueba con *f-goal-test*) acabará la busqueda devolviendo este último nodo, si no lo es llamará a la función auxiliar exp-condition, y si está devuelve true expandirá el nodo y llamará a *graph-search-aux* (función recursiva) insertando en open-nodes la expansión del nodo y sacando este de la lista y añadiendolo a closed-nodes.

La función *exp-condition* devuelve true si el nodo no está en la lista de closed-nodes o si está en la lista pero su g es más pequeña que la de la copia del nodo de la lista. Si no se cumple esta condición, se llamara a *graph-search-aux* con la lista de open-nodes quitando el primero que lo acabamos de explorar.

También implementamos la función *a-star-search* que lo unico que hará sera llamar a la función *graph-search* con la estrategia A*.

Ejercicio 10:

Código:

```
(defun solution-path (node)
  (if (null (node-parent node))
    (node-city node)
    (solution-path-aux node (list (node-city node)))))


(defun solution-path-aux(node path)
  (if (null (node-parent node))
    path
    (solution-path-aux (node-parent node) (cons (node-city (node-parent node)) path))))
```

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

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

En este ejercicio se nos pide una función que dado un nodo, devuelva el camino que ha recorrido (*solution-path*) o las acciones que ha ido realizando (*action-sequence*). La idea esla misma para ambas funciones y es llamar a una función auxiliar recursiva con el nodo y una lista donde ira metiendo el recorrido. Esta función auxiliar se ira llamando a si misma pasandole como argumento el nodo-parent del nodo y la lista del recorrido, y finalizará cuando llegue a un nodo que no tenga padre (*solution-path*) o que no tenga una accion para llegar a él (*action-sequence*).

Ejercicio 11:

Código:

```
(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))
```

En este apartado hemos implementado las estrategias de busqueda en anchura y profundidad. Son implementaciones sencillas, ya que la busqueda en anchura explora siempre los nodos que menor profundidad tengan por lo que al comparar cuando insertamos los nodos recien explorados, estos tendran siempre anchura mayor o igual a los de la lista, por lo que se insertaran al final, y basta con que la funcion devuelva *nil* siempre.

Para profundidad ocurre lo contrario, al expandir un nodo, hay que colocar sus sucesores los primeros de la lista para seguir expandiendolos. Por esta razón su función de comparación devuelve siempre *true*.

Ejercicio 12:

Código:

```
(defparameter *heuristic-new*
  '((Calais 0.0) (Reims 20.0) (Paris 27.0)
    (Nancy 43.0) (Orleans 51.0) (St-Malo 59.0)
    (Nantes 71.0) (Brest 85.0) (Nevers 65.0)
    (Limoges 95.0) (Roenne 83.0) (Lyon 100.0)
    (Toulouse 129.0) (Avignon 115.0) (Marseille 135.0)))


(defparameter *travel-new*
  (make-problem
    :cities              *cities*
    :initial-city        *origin*
    :f-h                 #'(lambda(c) (f-h c *heuristic-new*))
    :f-goal-test         #'(lambda(n) (f-goal-test n *destination* *mandatory*))
    :f-search-state-equal #'(lambda(n1 n2) (f-search-state-equal n1 n2 *mandatory*))
    :succ                #'(lambda(n) (succ n *trains*))))


(defparameter *heuristic-cero*
  '((Calais 0.0) (Reims 0.0) (Paris 0.0)
    (Nancy 0.0) (Orleans 0.0) (St-Malo 0.0)
    (Nantes 0.0) (Brest 0.0) (Nevers 0.0)
    (Limoges 0.0) (Roenne 0.0) (Lyon 0.0)
```

```
   (Toulouse 0.0) (Avignon 0.0) (Marseille 0.0)))


(defparameter *travel-cero*
  (make-problem
    :cities              *cities*
    :initial-city        *origin*
    :f-h                 #'(lambda(c) (f-h c *heuristic-cero*))
    :f-goal-test         #'(lambda(n) (f-goal-test n *destination* *mandatory*))
    :f-search-state-equal #'(lambda(n1 n2) (f-search-state-equal n1 n2 *mandatory*))
    :succ                #'(lambda(n) (succ n *trains*))))
```

La nueva heurística consiste en la misma dada pero siendo algo más optimistas, por lo que deberían tener un peor desempeño, aunque mejor que la heurística trivial como vemos a continuación:

(time (solution-path (graph-search ***travel*** *A-star*)))

Real time: 0.002448 sec.

Run time: **0.002441 sec.**

Space: 79360 Bytes

(MARSEILLE TOULOUSE LIMOGES ORLEANS PARIS CALAIS)

(time (solution-path (graph-search ***travel-new*** *A-star*)))

Real time: 0.005665 sec.

Run time: **0.005629 sec.**

Space: 79344 Bytes

(MARSEILLE TOULOUSE LIMOGES ORLEANS PARIS CALAIS)

(time (solution-path (graph-search ***travel-cero*** *A-star*)))

Real time: 0.008141 sec.

Run time: **0.008081 sec.**

Space: 115344 Bytes

(MARSEILLE TOULOUSE LIMOGES ORLEANS PARIS CALAIS)

Los resultados son los esperados, ya que cuanto menos se parece la heurística al valor real, más tarda A* en encontrar la solución óptima

Cuestiones:

1) ¿Por qué se realizado este diseño para resolver el problema de búsqueda? En concreto:

1.A) ¿Qué ventajas aporta?

Una de las ventajas de esta resolución es la modularidad con la que se ha trabajado. Practicamente todas las funciones principales se pueden realizar de forma independiente sin conocer las demás. Esto además nos permite no tener que cambiar la estructura para abordar diferentes tipos de busqueda o diferentes problemas. Esto nos permite realizar por ejemplo otras estrategias de busqueda y unicamente tendriamos que cambiar la viariable *strategy* del problema que hemos definido.

Ademas esto nos permite poder trabajar en equipo de forma óptima, ya que el reparto de trabajo no tiene complicaciones ni son necesarias las funciones de los otros modulos.

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

Se ha realizado así con el objetivo de que si se quiere cambiar la heuristica a seguir, el objetivo o algún otro dato del problema, unicamente habrá que modificar la estructura de problem que hayamos definido sin tener que modificar ninguna función.

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?

Es cierto que puede parecer que no es una implementación eficiente, ya que todos los nodos recorridos tendrán que permanecer en memoria. Pero a la hora de implementar determinadas funciones como *solution-path*, *action-sequence* o para comporbar si dos nodos son iguales con *f-search-state-equal* la estructura de padres es muy eficiente y simplifica mucho la implementación. Además ya contamos con que el uso de memoria en A* suele ser muy elevado.

3 y 4) ¿Cuál es la complejidad espacial del algoritmo implementado? ¿Cuál es la complejidad temporal del algoritmo implementado?

Como hemos visto en clase de teoria la complejidad del algoritmo, así como su complejidad temporal es exponencial