Pareja: 02
Alba Ramos Pedroviejo
Nicolás Serrano Salas

Práctica 2
Grupo: 2361
INTART

1. Obtenemos una lista de pares desde city mediante la llamada a assoc. Obtenemos el primer elemento de esa lista, que es la ciudad que queremos evaluar, y de ese par obtenemos el segundo elemento que es el valor de la heurística.

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

Se han ejecutado los siguientes tests:

```
(f-h '() *heuristic*) ;; NIL
(f-h 'Nantes '()) ;; NIL
(f-h 'Nantes *heuristic*) ;;  75.0
(f-h 'Marseille *heuristic*) ;; 145.0
(f-h 'Lyon *heuristic*) ;; 105.0
(f-h 'Madrid *heuristic*)  ;; NIL
(f-h 'Nancy *heuristic*) ;; 50.0
```

2. De la lista de nodos recibida nos quedamos solo con aquellos que empiecen por la ciudad que queremos estudiar. Para todos ellos, crearemos una acción que consista en navegar desde su ciudad hacia la ciudad a la que están conectados con el coste indicado.

```
(defun navigate (city lst-edges )
 (mapcar #'(lambda(y)(make-action :name 'navigate :origin city :final
(second y) :cost (third y)))
         (remove-if #'(lambda(x)(not (equal city (first x))) )
lst-edges)))
```

Se han ejecutado los siguientes tests:

```
(navigate 'Avignon '()) ;; NIL
(navigate '() *trains*) ;; NIL
(navigate 'Madrid *trains*) ;; NIL
(action-cost (car (navigate 'Avignon *trains*))) ;; 30.0
(mapcar #'action-cost (navigate 'Avignon *trains*)) ;; (30.0 16.0)
(action-final (car (navigate 'Avignon *trains*))) ;; LYON
```

```
(mapcar #'action-final (navigate 'Paris *trains*)) ;; (CALAIS NANCY
NEVERS ORLEANS ST-MALO)
```

3. Primero miraremos si alguno de los destinos es el nodo que nos pasan, en cuyo caso llamaremos a nuestra función auxiliar pasándole el padre de este nodo y la lista de nodos obligatorios.
Hemos definido una función auxiliar que, dados un nodo y una lista de nodos obligatorios, comprueba si alguno de ellos coincide con el nodo y lo elimina de la lista. Después, seguirá llamando recursivamente con el padre del nodo, con el objetivo de vaciar la lista de nodos obligatorios. Esto quiere decir que el camino hacia el nodo original contiene los nodos obligatorios, entonces nuestra función devolverá TRUE. Si, en cambio, llegamos a la raíz y no hemos vaciado la lista, entonces el camino no contiene todos los obligatorios y el camino no es correcto.

```
(defun f-goal-test-aux (node mandatory)
 (if (null mandatory)
     T
     (unless (null node)
             (f-goal-test-aux (node-parent node) (remove-if
#'(lambda(x)(equal (node-city node) x)) mandatory)))))

(defun f-goal-test (node destination mandatory)
 (when (some #'(lambda(x) (equal (node-city node) x)) destination)
       (f-goal-test-aux (node-parent node) mandatory)))
```

Se han ejecutado los siguientes tests:

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

4. Hemos comprobado que ambos nodos representan la misma ciudad y que ambos pasen por el camino obligatorio llamando a la función auxiliar creada en el ejercicio anterior. Además, hemos establecido que dos valores NIL no sean iguales. Esto es para evitar que si la función recibe como parámetros dos valores NIL, entonces no continúe porque esto no son nodos y, como tal, no van a tener un espacio de búsqueda para compararlos.

```
(defun f-search-state-equal (node-1 node-2 &optional mandatory)
 (unless (or (null node-1) (null node-2))
   (when (equal (node-city node-1) (node-city node-2))
         (and (f-goal-test-aux (node-parent node-1) mandatory)
              (f-goal-test-aux (node-parent node-2) mandatory)))))
```

Se han ejecutado los siguientes tests:

```
(f-search-state-equal '() '()) ;; NIL
(f-search-state-equal node-calais '() '()) ;; NIL
(f-search-state-equal node-calais node-calais-2 '()) ;; T
(f-search-state-equal node-calais node-calais-2 '(Reims)) ;; NIL
(f-search-state-equal node-calais node-calais-2 '(Nevers)) ;; T
(f-search-state-equal node-nancy node-paris '()) ;; NIL
(f-search-state-equal node-calais node-calais '(Paris Nancy)) ;; T
```

5. Hemos definido la siguiente estructura:

```
(defparameter *travel*
 (make-problem
   :cities                *cities*
   :initial-city          *origin*
   :f-h                   #'(lambda (city) (f-h city *heuristic*))
   :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*))
   :succ                  #'(lambda (node) (navigate node *trains*))))
```

6. Para un nodo, obtenemos todas las acciones que son posibles realizar desde él utilizando la función del problema. Luego, para cada una de las acciones, vamos a llamar a una función auxiliar que es la que va a construir los nodos sucesores utilizando la información almacenada en la acción y en el nodo que la ha generado. Concretamente, la profundidad del nuevo nodo será 1 más la profundidad del padre. El coste será el coste de la acción más el coste del padre. La heurística se obtendrá a través del problema. Finalmente, la f se obtiene sumando g y h.

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

```
(defun expand-node (node problem)
 (unless (or (null node) (null problem))
   (mapcar #'(lambda(x) (expand-node-action node x problem))
           (funcall (problem-succ problem) (node-city node)))))
```

Hemos ejecutado los siguientes tests:

```
(expand-node node-marseille-ex6 '()) ;; NIL
(expand-node '() *travel*) ;; NIL
(node-g (car (expand-node node-marseille-ex6 *travel*))) ;; 26.0
(node-city (node-parent (car (expand-node node-marseille-ex6
*travel*)))) ;; MARSEILLE
(node-parent (node-parent (car (expand-node node-marseille-ex6
*travel*)))) ;; NIL
(mapcar #'(lambda (x) (node-city (node-parent x))) (expand-node
node-nancy *travel*)) ;; (NANCY NANCY)
(node-g (car (expand-node node-marseille-ex6 *travel*))) ;; 26.0
```

7. La función realiza llamadas recursivas para ordenar el elemento a insertar en la lista. Para ello, comprueba si el elemento es menor que el primer elemento de la lista de acuerdo al criterio recibido como argumento, en cuyo caso lo inserta al principio. Si no lo es, entonces se vuelve a llamar a la función, esta vez para comparar con el resto de elementos de la lista.

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

Hemos ejecutado los siguientes tests:

```
(insert-nodes-strategy (list '())
                           '()
                           *A-star*)
;; (NIL)
(defparameter sol-ex7 (insert-nodes-strategy (list node-paris-ex7
node-nancy-ex7)
                                             (list node-calais-ex7)
```

```
                                          *A-star*))
(defparameter other-ex7 (insert-nodes-strategy sol-ex7 (list
node-marseille-ex6) *A-star*))



(mapcar #'(lambda (x) (node-city x)) sol-ex7) ;; (CALAIS PARIS NANCY)
(mapcar #'(lambda (x) (node-g x)) sol-ex7) ;; (150 0 50)
(mapcar #'(lambda (x) (node-city x)) other-ex7) ;; (CALAIS PARIS
MARSEILLE NANCY)
(mapcar #'(lambda (x) (node-g x)) other-ex7) ;; (150 0 10 50)
```

8. La búsqueda A* compara las f de los nodos, es decir, la suma de g+h. Elegirá el nodo que tenga menor f, y si la tienen igual elegirá el que primero se generó (de ahí la comparación <=).

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

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

Hemos ejecutado los siguientes tests:

```
(graph-search *travel* '()) ;; NIL
(graph-search '() *A-star*) ;; NIL
(node-g (graph-search *travel* *A-star*)) ;; 202.0
(node-city (graph-search *travel* *A-star*)) ;; CALAIS
(node-city (node-parent (graph-search *travel* *A-star*))) ;; PARIS
```

9. Hemos codificado el pseudocódigo que nos enseñan en el enunciado de la siguiente manera:
Lo que está dentro de repeat y fin hemos creado una función auxiliar graph-search-aux que hace la lógica del algoritmo, y lo de fuera que inicializa las listas abiertas y cerradas como graph-search.

```
(defun graph-search-aux (problem strategy open-nodes closed-nodes)
   (unless (null open-nodes)
     (if (funcall (problem-f-goal-test problem) (car open-nodes))
         (car open-nodes)
         (if (or (not (member (car open-nodes) closed-nodes))
```

```
                (funcall (strategy-node-compare-p strategy) (car
open-nodes) (car (member (car open-nodes) closed-nodes))))
            (graph-search-aux problem
                              strategy
                              (insert-nodes-strategy (expand-node (car
open-nodes) problem)
                                                     (cdr open-nodes)
                                                     strategy)
                              (insert-nodes-strategy (list (car
open-nodes))
                                                     closed-nodes
                                                     strategy))
            (graph-search-aux problem strategy (cdr open-nodes)
closed-nodes)))))

(defun graph-search (problem strategy)
 (unless (or (null problem) (null strategy))
   (let ((open-nodes (list (make-node :city (problem-initial-city
problem)
                                      :parent '()
                                      :action '())))
         (closed-nodes '()))
     (graph-search-aux problem strategy open-nodes closed-nodes))))
```

Por último para codificar (defun a-star-search (problem) ...) simplemente hemos hecho lo siguiente:

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

Los test de este ejercicio son los mismos que los del ejercicio 8, los cuales necesitan estas funciones.

10. Primero se nos pide codificar 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 a un nodo dado. Para ellos la dividimos en una función principal y una función auxiliar la cual llamaremos recursivamente.

```
(defun solution-path-aux (path node)
 (if (null node)
     path
```

```
     (solution-path-aux (cons (node-city node) path) (node-parent
node))))

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

Después se nos pide codificar una función que muestra la secuencia de acciones para llegar a un nodo. Para ellos nos basamos un poco en la implementación de la función anterior y queda:

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

(defun action-sequence (node)
 (if (or (null node) (null (node-action node)))
     '()
     (action-sequence-aux (list (node-action node)) (node-parent
node))))
```

En este caso en la condición de recursión comprobamos también aparte de que si el nodo que nos pasan sea NIL, si la acción del nodo que nos pasan es null, lo que indicaría que ha sido el nodo de partida y no ha habido ninguna acción que ha generado dicho nodo.

Hemos ejecutado los siguientes tests:

```
(solution-path NIL) ;; NIL
(solution-path (a-star-search *travel*)) ;; (MARSEILLE TOULOUSE LIMOGES
ORLEANS PARIS CALAIS)
(action-sequence NIL) ;; NIL
(action-sequence (a-star-search *travel*))
;; (#S(ACTION :NAME NAVIGATE :ORIGIN MARSEILLE :FINAL TOULOUSE :COST
65.0)
;;  #S(ACTION :NAME NAVIGATE :ORIGIN TOULOUSE :FINAL LIMOGES :COST
25.0)
;;  #S(ACTION :NAME NAVIGATE :ORIGIN LIMOGES :FINAL ORLEANS :COST 55.0)
;;  #S(ACTION :NAME NAVIGATE :ORIGIN ORLEANS :FINAL PARIS :COST 23.0)
;;  #S(ACTION :NAME NAVIGATE :ORIGIN PARIS :FINAL CALAIS :COST 34.0))
```

11. Se nos pide diseñar las estrategias para la búsqueda en anchura y en profundidad, para ello seguimos la misma lógica que cuando hemos creado la búsqueda en A*. Para ello primero creamos una función de comparación entre dos nodos y luego creamos las estrategias correspondientes.

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

(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)
 (<  (node-depth node-1)(node-depth node-2)))

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

En estas comparaciones en vez de comparar f queremos comparar la profundidad, en la búsqueda en anchura primero queremos ir evaluando nodos por niveles así que tendrán preferencia los nodos de menor profundidad. En la búsqueda por profundidad es justamente al revés, como queremos explorar primero el nodo que se acaba de expandir exploramos en nodo de mayor profundidad.

Aquí observamos una prueba de ejecución de los siguientes algoritmos, viendo que ambos llegan a la solución y que la profundidad alcanzada en el algoritmo de profundidad es >= que la alcanzada en el algoritmo en anchura.

```
(graph-search *travel* *depth-first*)
;; #S(NODE
;;    :CITY CALAIS
;;    :PARENT #S(NODE
;;               :CITY PARIS
;;               :PARENT #S(NODE
;;                          :CITY ST-MALO
;;                          :PARENT #S(NODE
;;                                     :CITY NANTES
;;                                     :PARENT #S(NODE
;;                                                :CITY TOULOUSE
;;                                                :PARENT #S(NODE
;;                                                           :CITY LYON
```

```
;;                                                  :PARENT #S(NODE
;;                                                            :CITY AVIGNON
;;                                                            :PARENT #S(NODE
;;                                                                      :CITY MARSEILLE
;;                                                                      :PARENT NIL
;;                                                                      :ACTION NIL
;;                                                                      :DEPTH 0
;;                                                                      :G 0
;;                                                                      :H 0
;;                                                                      :F 0)
;;                                                            :ACTION #S(ACTION
;;                                                                      :NAME NAVIGATE
;;                                                                      :ORIGIN MARSEILLE
;;                                                                      :FINAL AVIGNON
;;                                                                      :COST 16.0)
;;                                                            :DEPTH 1
;;                                                            :G 16.0
;;                                                            :H 135.0
;;                                                            :F 151.0)
;;                                                  :ACTION #S(ACTION
;;                                                            :NAME NAVIGATE
;;                                                            :ORIGIN AVIGNON
;;                                                            :FINAL LYON
;;                                                            :COST 30.0)
;;                                                  :DEPTH 2
;;                                                  :G 46.0
;;                                                  :H 105.0
;;                                                  :F 151.0)
;;                                        :ACTION #S(ACTION
;;                                                  :NAME NAVIGATE
;;                                                  :ORIGIN LYON
;;                                                  :FINAL TOULOUSE
;;                                                  :COST 60.0)
;;                                        :DEPTH 3
;;                                        :G 106.0
;;                                        :H 130.0
;;                                        :F 236.0)
;;                              :ACTION #S(ACTION
;;                                        :NAME NAVIGATE
;;                                        :ORIGIN TOULOUSE
;;                                        :FINAL NANTES
;;                                        :COST 80.0)
;;                              :DEPTH 4
;;                              :G 186.0
;;                              :H 75.0
;;                              :F 261.0)
;;                    :ACTION #S(ACTION
;;                              :NAME NAVIGATE
;;                              :ORIGIN NANTES
;;                              :FINAL ST-MALO
;;                              :COST 20.0)
;;                    :DEPTH 5
;;                    :G 206.0
;;                    :H 65.0
;;                    :F 271.0)
;;          :ACTION #S(ACTION
;;                    :NAME NAVIGATE
;;                    :ORIGIN ST-MALO
```

```
;;                          :FINAL PARIS
;;                          :COST 40.0)
;;               :DEPTH 6
;;               :G 246.0
;;               :H 30.0
;;               :F 276.0)
;;    :ACTION #S(ACTION :NAME NAVIGATE :ORIGIN PARIS :FINAL CALAIS :COST 34.0)
;;    :DEPTH 7
;;    :G 280.0
;;    :H 0.0
;;    :F 280.0)

(graph-search *travel* *breadth-first*)
;; #S(NODE
;;    :CITY CALAIS
;;    :PARENT #S(NODE
;;               :CITY PARIS
;;               :PARENT #S(NODE
;;                          :CITY ST-MALO
;;                          :PARENT #S(NODE
;;                                     :CITY NANTES
;;                                     :PARENT #S(NODE
;;                                                :CITY TOULOUSE
;;                                                :PARENT #S(NODE
;;                                                           :CITY MARSEILLE
;;                                                           :PARENT NIL
;;                                                           :ACTION NIL
;;                                                           :DEPTH 0
;;                                                           :G 0
;;                                                           :H 0
;;                                                           :F 0)
;;                                                :ACTION #S(ACTION
;;                                                           :NAME NAVIGATE
;;                                                           :ORIGIN MARSEILLE
;;                                                           :FINAL TOULOUSE
;;                                                           :COST 65.0)
;;                                                :DEPTH 1
;;                                                :G 65.0
;;                                                :H 130.0
;;                                                :F 195.0)
;;                                     :ACTION #S(ACTION
;;                                                :NAME NAVIGATE
;;                                                :ORIGIN TOULOUSE
;;                                                :FINAL NANTES
;;                                                :COST 80.0)
;;                                     :DEPTH 2
;;                                     :G 145.0
;;                                     :H 75.0
;;                                     :F 220.0)
;;                          :ACTION #S(ACTION
;;                                     :NAME NAVIGATE
;;                                     :ORIGIN NANTES
;;                                     :FINAL ST-MALO
;;                                     :COST 20.0)
;;                          :DEPTH 3
;;                          :G 165.0
;;                          :H 65.0
```

```
;;                          :F 230.0)
;;              :ACTION #S(ACTION
;;                          :NAME NAVIGATE
;;                          :ORIGIN ST-MALO
;;                          :FINAL PARIS
;;                          :COST 40.0)
;;              :DEPTH 4
;;              :G 205.0
;;              :H 30.0
;;              :F 235.0)
;;    :ACTION #S(ACTION :NAME NAVIGATE :ORIGIN PARIS :FINAL CALAIS :COST 34.0)
;;    :DEPTH 5
;;    :G 239.0
;;    :H 0.0
;;    :F 239.0)
```

12. Hemos definido la heurística *heuristic-new* tomando como heurística para todos los nodos la misma distancia, y esta igual a la menor de todas (16.0), salvo en la meta que es 0. Hemos creado un nuevo problema que use esta heurística.

```
(defparameter *heuristic-new*
 '((Calais 0.0) (Reims 16.0) (Paris 16.0)
   (Nancy 16.0) (Orleans 16.0) (St-Malo 16.0)
   (Nantes 16.0) (Brest 16.0) (Nevers 16.0)
   (Limoges 16.0) (Roenne 16.0) (Lyon 16.0)
   (Toulouse 16.0) (Avignon 16.0) (Marseille 16.0)))

(defparameter *travel-new*
 (make-problem
   :cities                 *cities*
   :initial-city           *origin*
   :f-h                    #'(lambda (city) (f-h city
*heuristic-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*))
   :succ                   #'(lambda (node) (navigate node
*trains*))))
```

Hemos definido la heurística *heuristic-new* tomando como heurística 0.0 para todos los nodos, la opción más optimista posible. Hemos definido otro problema para usar esta heurística.

```
(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 (city) (f-h city
*heuristic-cero*))
   :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*))
   :succ                  #'(lambda (node) (navigate node
*trains*))))
```

Hemos realizado el mismo test para probar las tres heurísticas y comparar los tiempos y ciclos de CPU que tarda cada una. Como es lógico, *heuristic-cero* es la que más tarda con diferencia, ya que está siendo excesivamente optimista y no es realista. Vemos que *heuristic-new* tarda más que la que teníamos originalmente, porque también está siendo bastante optimista y poco precisa: está asumiendo que todos los nodos son igual de buenos. Podemos observar que la mejor es la que ya teníamos, porque esta sí que establece unos nodos mejores que otros y afina más la búsqueda.

```
(time (graph-search *travel* *A-star*))
;; Evaluation took:
;;   0.000 seconds of real time
;;   0.000027 seconds of total run time (0.000027 user, 0.000000
system)
;;   100.00% CPU
;;   100,258 processor cycles
;;   0 bytes consed
```

```
(time (graph-search *travel-new* *A-star*))
;; Evaluation took:
;;   0.001 seconds of real time
;;   0.001622 seconds of total run time (0.001622 user, 0.000000
system)
;;   200.00% CPU
;;   6,140,122 processor cycles
;;   786,368 bytes consed

(time (graph-search *travel-cero* *A-star*))
;; Evaluation took:
;;   0.004 seconds of real time
;;   0.003367 seconds of total run time (0.003367 user, 0.000000
system)
;;   75.00% CPU
;;   12,743,535 processor cycles
;;   1,834,960 bytes consed
```

Cuestiones

1a. La ventaja es que usando búsqueda en grafo con A* y una heurística monótona entonces la búsqueda será óptima

1b. Las funciones lambda nos permiten tener un problema fijo al que ir pasándole las distintas ciudades y los distintos nodos sin variar el resto de parámetros de las funciones que se llaman.

2. El uso de memoria es eficiente porque lo que almacenan todos los nodos son *referencias*, no bloques de memoria en sí. Son punteros, de forma que en memoria existe una única instancia de cada nodo, y todo lo demás son referencias a esa zona.

3. La complejidad espacial del algoritmo es exponencial porque necesita mantener todos los nodos generados en memoria.

4. La complejidad temporal del algoritmo es también exponencial.