Memoria P2 IA
Salvador Martin, Blanca Mercado
Grupo 2362, Pareja 2

En las normas de entrega pone "*Es imprescindible asegurarse de que el código entregado se ejecuta de manera automática, sin errores ni salidas de ningún tipo en las versiones y entornos recomendados e instalados en los laboratorios.*", sobre esto aclarar dos puntos, en nuestra versión de portacle ejecuta todo perfectamente, y aun que hemos realizado pruebas, las bases y todos los casos de error que se nos han ocurrido, no se han añadido en el fichero de código debido a esta indicación.

Código Ejercicio 1
Usamos una implementación lo mas simple posible para este ejercicio.

```
(defun f-h (city heuristic)
  (second (assoc city heuristic))) ;devuelve la heuristica de la ciudad
```

Código Ejercicio 2
Usamos macan para eliminar los nil que nos devuelven los que no tienen como ciudad origen city, para el buen uso de mapcan necesitamos que cada uno de ellos sea una lista, por eso la creación de la lista por cada acción.

```
(defun navigate (city lst-edges)
  (mapcan
    #'(lambda (x)
        (when (equal city (first x)) ;lo hace solo para los city
          (list ;creacion de lista para el buen uso de mapcan
            (make-action :name 'action ;crea las acciones con lst-edges
              :origin (first x)
              :final (second x)
              :cost (third x)))))
          lst-edges))
```

Código Ejercicio 3
Usamos una primera función para ver si el nodo es una de las ciudades destino, en caso afirmativo comprobamos para cada ciudad obligatoria si el nodo ha pasado por ella con una función auxiliar.

```
(defun f-goal-test (node destination mandatory)
  (cond
    ((null node) nil)
    ((null (find (node-city node) destination)) nil) ;si no es un destino nil
    ((null mandatory) t) ;no hay ciudades obligatorias
    (t
      (every #'(lambda (x) x)
        (mapcar #'(lambda (mandator) ;para cada ciudad en mandatory
          (f-goal-test-aux node mandator)) ;comprueba si ha visitado la ciudad
        mandatory)))))
```

```
(defun f-goal-test-aux (node mandator)
    (cond
      ((equal (node-city node) mandator) t)
      ((null (node-parent node)) nil)
      (t (f-goal-test-aux (node-parent node) mandator))))
```

Código Ejercicio 4

Miramos si las ciudades son iguales, en caso afirmativo pasamos la función del ejercicio 3 (para no repetir código) para ver si han pasado por las ciudades obligatorias.

```
(defun f-search-state-equal (node-1 node-2 &optional mandatory)
  (cond
    ((or (null node-1) (null node-2)) nil)
    ((and ;true si se cumplen las 3
      (equal ;comprueba si es la misma ciudad
        (node-city node-1)
        (node-city node-2))
      (f-goal-test ;comprueba ciudades obligatorias
        node-1
        (list (node-city node-1))
        mandatory)
      (f-goal-test ;comprueba ciudades obligatorias
        node-2
        (list (node-city node-2))
        mandatory))
      t)))
```

Código Ejercicio 5

Definimos el problema *travel* con las funciones que hemos ido creando y las ciudades proporcionadas en la practica.

```
(defparameter *travel* ;crea el problema del enunciado
  (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-city node)
*trains*))))
```

Código Ejercicio 6

Sacamos todas las acciones con la función “succ” del problema, y para cada una de ellas creamos un nodo con toda la información.

```
(defun expand-node (node problem)
  (if (or (null node) (null problem))
    nil
    (mapcar;creamos un nodo para cada accion
      #'(lambda (action)
          (expand-node-action ;sacamos todas las acciones
            node
            action
            problem))
          (funcall (problem-succ problem) node))))
```

```
(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)) ;aumentamos el camino
    :h      (funcall (problem-f-h problem) (action-final action)) ;calculamos
la heuristica
    :f      (+
      (+ (node-g node) (action-cost action))
      (funcall (problem-f-h problem) (action-final action))))) ;calculamos f
```

Código Ejercicio 7

Usamos tres funciones a petición del enunciado que hay dentro del fichero de código, pero debido a nuestra implementación solo serian necesarias 2.

Primero usamos una interfaz para sacar la función de comparación de nodos de la estrategia.

```
(defun insert-nodes-strategy (nodes lst-nodes strategy)
  (if (or (null nodes) (null lst-nodes) (null strategy))
    nil
    (insert-nodes
      nodes
      lst-nodes
      (strategy-node-compare-p strategy)))) ;sacamos la funcion de comparacion
```

En este punto podríamos unir ambas listas y ordenarlas con nuestra función de comparación, pero como hemos dicho ante, a petición del enunciado lo hacemos nodo a nodo con la función auxiliar.

```
(defun insert-nodes (nodes lst-nodes node-compare-p)
  (if (null nodes)
    lst-nodes
    (insert-nodes ;insertamos cada uno de los nodos
      (rest nodes)
      (insert-node (first nodes) lst-nodes node-compare-p)
      node-compare-p)))
```

La ultima función hace la inserción nodo a nodo.

```
(defun insert-node (node lst-nodes node-compare-p)
  (sort (cons node lst-nodes) node-compare-p)) ;insertamos y ordenamos
```

Código Ejercicio 8

Creamos la estrategia A* con su función de comparación que utiliza la f para decidir cual expandir primero.

```
(defun f-leq (node-1 node-2)
  (<= (node-f node-1) ;A* evalua el nodo con menor f
      (node-f node-2)))

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

Código Ejercicio 9

Este es el ejercicio mas largo de todos y el que más nos costo así que explicaremos nuestro método de resolución por partes.

En la primera función lo único que hacemos es inicializar la lista de nodos, para ello sacamos el nombre de la ciudad inicial y creamos un nodo con ella.

```
(defun graph-search (problem strategy)
  (if (or (null problem) (null strategy))
    nil
    (graph-search-rec
      (list (make-node ;nodo inicial
        :city   (problem-initial-city problem)))
      nil ;nodos visitados
      problem ;problema para usar las funciones necesarias
      strategy)))
```

La segunda función es la que tiene mas contenido y es la que realiza el algoritmo general.

En ella tenemos dos condiciones de salida, la que encuentra el destino y la que no lo encuentra, y dos condiciones que siguen con la recursión, la primera es la principal, esta expande el nodo que toca explorar y vuelve a llamar a la función, esto se realiza en caso de que la función auxiliar “exp-cond” nos devuelva true ya que es un nodo que tenemos que explorar, en caso contrario llamaremos otra vez a la función pero sin expandir otra vez el nodo y retirándolo de open-nodes.

```
(defun graph-search-rec (open-nodes closed-nodes problem strategy)
  (let ((node-first (first open-nodes)))
    (cond
      ((null open-nodes) nil) ;condicion de salida si no se encuentra camino.
      ((funcall ;comprobacion de si el nodo a evaluar es el destino
        (problem-f-goal-test problem)
        node-first) node-first)
      ((exp-cond node-first closed-nodes) ;caso de recursion
        (graph-search-rec
          (remove
            node-first
            (insert-nodes-strategy ;open nodes con los nuevos y sin el
explorado
              (expand-node node-first problem)
              open-nodes
              strategy))
          (if (null closed-nodes) ;closed nodes con el nodo explorado
            (list node-first)
            (cons node-first closed-nodes))
          problem
          strategy))
      (t (graph-search-rec ;en caso de que el nodo ya se haya explordo
          (remove          ;se saca de open nodes
            node-first
            open-nodes)
          closed-nodes
```

```
          problem
          strategy)))))
```

Por último, la función “exp-cond” que devuelve true en caso de que el nodo a explorar no se haya explorado ya, o en caso de haberse explorado ya, si la solución de esta exploración es menos optima que la nueva.

```
(defun exp-cond (node closed-nodes)
  (if (not ;condicion de si no esta en closed-nodes
        (find
          (node-city node)
          (mapcar
            #'(lambda (closed-node)
              (node-city closed-node))
            closed-nodes)))
  t
  (every ;si esta en closed-nodes evalua si el nuevo tiene un mejor path
    #'(lambda (x)
      (< (node-g node) (node-g x)))
    (remove-if-not
      #'(lambda (close)
        (eq (node-city node) (node-city close)))
      closed-nodes))))
```

También se nos pide realizar una función que dado un problema lo resuelva por el algoritmo A*, en ella llamamos a la función de búsqueda en grafo con esta estrategia.

```
(defun a-star-search (problem)
  (if (null problem)
    nil
    (graph-search problem *A-star*))) ;llama a la busqueda con el algoritmo A*
```

Código Ejercicio 10

Para ambas funciones miramos desde el último nodo padre y devolvemos la ciudad o la acción, la diferencia es que en la acción no añadimos la del último ya que al ser el nodo raíz no tiene acción.

```
(defun solution-path (node)
  (cond
    ((null node) nil)
    ((null (node-parent node))
      (list (node-city node)))
    (t
      (append ;creacion de lista con los nombres de todos los nodos del path
        (solution-path (node-parent node))
        (list (node-city node))))))

(defun action-sequence (node)
  (cond
    ((null node) nil)
    ((null (node-parent (node-parent node))) ;salimos uno antes porque
      (list (node-action node)))             ;el primero no tiene action
    (t
      (append ;creacion de lista con todas las acciones llevadas
        (action-sequence (node-parent node)) ;a cabo para encontrar el nodo
        (list (node-action node))))))
```

Código Ejercicio 11

Hacemos lo mismo que en la búsqueda A* pero para la de profundidad, mayor profundidad se explora primero, y para el de anchura, menor profundidad se explora primero.

```
(defun depth-first-node-compare-p (node-1 node-2)
  (>= (node-depth node-1) ;evalua el que es mas profundo
        (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) ;evalua el que es menos profundo
        (node-depth node-2)))

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

Código Ejercicio 12

Heuristica-new creada con el coste real menos uno para que sea la mejor heurística por debajo del valor real.

```
(defparameter *heuristic-new*
  '((Calais 0.0) (Reims 34.0) (Paris 33.0)
    (Nancy 69.0) (Orleans 56.0) (St-Malo 73.0)
    (Nantes 93.0) (Brest 103.0) (Nevers 81.0)
    (Limoges 111.0) (Roenne 214.0) (Lyon 196.0)
    (Toulouse 136.0) (Avignon 217.0) (Marseille 201.0)))
```

Heuristica-cero

Todas las heurísticas a 0.

```
(defparameter *heuristic-cero*
  '((Calais 0.0) (Reims 0.0) (Paris 0.0)
    (Nancy 0.0) (Orleans 55.0) (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)))
```

Usando la función tiempo de lisp no salen estos resultados.

```
CL-USER> (time (graph-search *travel* *A-star*)); with *heuristic*
Evaluation took:
  0.000 seconds of real time
  0.000035 seconds of total run time (0.000035 user, 0.000000 system)
  100.00% CPU
  77,890 processor cycles
  0 bytes consed
```

```
CL-USER> (time (graph-search *travel* *A-star*)); with *heuristic-new*
Evaluation took:
  0.000 seconds of real time
  0.000024 seconds of total run time (0.000024 user, 0.000000 system)
  100.00% CPU
  54,644 processor cycles
  0 bytes consed
```

```
CL-USER> (time (graph-search *travel* *A-star*)); with *heuristic-cero*
Evaluation took:
  0.000 seconds of real time
  0.000077 seconds of total run time (0.000077 user, 0.000000 system)
  100.00% CPU
  175,238 processor cycles
  0 bytes consed
```

Fijándonos en los ciclos de CPU para que sea una comparación realista sin tener en cuenta la carga del propio ordenador, podemos ver que la heurística-cero es la peor de todas, duplicando los ciclos de la heurística del enunciado y triplicando la más ajustada, esto nos indica que con una heurística muy bien ajustada tardaremos menos en encontrar el camino optimo, por eso es necesario elegir una buena heurística.

Preguntas

1. a) Es una solución muy modular y que es muy útil para cambiar cada función y utilizar la que mas nos convenga en cada problema.

   b) Las funciones lambda nos permiten usar funciones con unos parámetros iguales en todo el problema e ir cambiando solo la información que nos interesa como por ejemplo los nodos.

2. Es eficiente en el sentido de que un nodo padre es guardado solo por sus hijos y no se multiplica la información para generaciones posteriores de hijos.

3. La complejidad espacial es en número de nodos sin contar el de la raíz por 2 ya que cada nodo guarda el nodo padre y a si mismo.

4. La complejidad temporal depende de la heurística utilizada como hemos visto en el ejercicio 12.