MEMORIA PRÁCTICA 2

Alejandro Santorum - alejandro.santorum@estudiante.uam.es
Sergio Galán Martín - sergio.galanm@estudiante.uam.es
Inteligencia Artificial
Práctica 2 Pareja 9

25 de marzo de 2019

Contents

1 Introducción

Nos encontramos ante la segunda práctica de Inteliencia Artificial. Nuestro objetivo será afianzarse en el uso del lenguaje Lisp y, además, conseguir diseñar e implementar el algoritmo de búsqueda *A**.

2 Ejercicio 1

En este ejercicio definimos las funciones que nos obtendrán la heurística para precio y tiempo de un determinado estado (ciudad) extrayéndolas de *estimate*.

Código Ejercicio 1

```
(defun f-h-time (state sensors)
  (let ((val (assoc state sensors)))
    (when (not (null val))
      (caadr val))))

(defun f-h-price (state sensors)
  (let ((val (assoc state sensors)))
    (when (not (null val))
(cadadr val))))
```

3 Ejercicio 2

El segundo ejercicio consiste en implementar los operadores, que son funciones que, dado un estado (en esta práctica, el nombre de una ciudad) devuelven una lista de las acciones que se pueden efectuar a partir de ese estado. En nuestro sistema se implementan cuatro operadores:
· A qué `ciudades` se puede llegar usando el `tren` y su `tiempo` de recorrido.
· A qué `ciudades` se puede llegar usando el `tren` y su `precio` de recorrido.
· A qué `ciudades` se puede llegar usando un `canal` y su `tiempo` de recorrido.
· A qué `ciudades` se puede llegar usando un `canal` y su `precio` de recorrido.

Código Ejercicio 2

```
;; Auxiliary function to navigate
(defun nav-aux (state edge cfun name forbidden)
  (if (and (equal state (first edge))
           (null (member (second edge) forbidden)))
      (make-action :name name
                   :origin state
                   :final (second edge)
                   :cost (funcall cfun (third edge)))))

;; Auxiliary functions for the four functions below
(defun navigate (state lst-edges cfun  name &optional (forbidden ()))
  (mapcan #'(lambda (x) (if (null x) NIL (list x)))
    (mapcar #'(lambda(x) (nav-aux state x cfun name forbidden)) lst-edges)))

(defun navigate-canal-time (state canals)
  (navigate state canals #'first 'navigate-canal-time))

(defun navigate-canal-price (state canals)
  (navigate state canals #'cadr 'navigate-canal-price))

(defun navigate-train-time (state trains forbidden)
  (navigate state trains #'first 'navigate-train-time forbidden))

(defun navigate-train-price (state trains forbidden)
  (navigate state trains #'cadr 'navigate-train-price forbidden))
```

4 Ejercicio 3

Para el tercer ejercicio, se nos pide implementar una función que compruebe si un nodo es estado final correcto. Las condiciones que se tienen que cumplir son dos, que el estado del nodo sea alguna de las ciudades destino y que haya pasado por todas las ciudades obligatorias. Para ello, aparte de la función principal, hemos implementado una función auxiliar `eliminate` que nos ayudará a comprobar que se han visitado todas las ciudades obligatorias.

Código Ejercicio 3

```
(defun f-goal-test (node destination mandatory)
  (if (and (not (null (member (node-state node) destination)))
           (null (eliminate node mandatory)))
      T
    NIL))

(defun eliminate (node mandatory)
  (if (null (node-parent node))
      mandatory
(eliminate (node-parent node) (remove (node-state node) mandatory))))
```

5 Ejercicio 4

En el cuarto ejercicio tenemos que definir una función que compruebe la igualdad de dos nodos a efectos prácticos, es decir, que estén en la misma ciudad y que hayan recorrido las mismas ciudades obligatorias. Esto será útil de cara a asegurar que la solución dada por el algoritmo de búsqueda sea óptima incluso con heurísticas no muy acertadas.

Código Ejercicio 4

```
(defun f-search-state-equal (node-1 node-2 &optional mandatory)
  (and (equal (node-state node-1) (node-state node-2))
(equal (eliminate node-1 mandatory) (eliminate node-2 mandatory))))
```

6 Ejercicio 5

En este ejercicio se pedía representar el problema en el lenguaje Lisp. Se ha usado una estructura para inicializar ciertos valores ya predefinidos (como la lista de estados, el estado inicial, heurísticas, etc).

Como el objetivo de minimización es doble (minimizar el precio y minimizar el tiempo), se han definido dos estructuras de problemas.

Código Ejercicio 5

```
(defparameter *travel-cheap*
  (make-problem
   :states *cities*
   :initial-state *origin*
   :f-h #'(lambda (state) (f-h-price state *estimate*))
   :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*))
   :operators (list #'(lambda (node) (navigate-canal-price (node-state node) *canals*))
                    #'(lambda (node) (navigate-train-price (node-state node) *trains* *
                        forbidden*)))
   )
  )

(defparameter *travel-fast*
  (make-problem
   :states *cities*
   :initial-state *origin*
   :f-h #'(lambda (state) (f-h-time state *estimate*))
   :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*))
   :operators (list #'(lambda (node) (navigate-canal-time (node-state node) *canals*))
                    #'(lambda (node) (navigate-train-time (node-state node) *trains* *
                        forbidden*)))
   )
  )
```

7 Ejercicio 6

Para este ejercicio se nos pide implementar una función que expanda un nodo, esto es, genere todos los posibles nodos desde el nodo en el que estamos. En nuestro problema en concreto esto se traduce en "ir" a todas las ciudades a las que podamos ir, tanto en tren como en canal, desde la ciudad en la que estamos. Devolverá una lista con todos los posibles siguientes pasos a dar.

Código Ejercicio 6

```
(defun expand-node (node problem)
  (mapcan #'(lambda (x) (if (null x) NIL x))
    (mapcar #'(lambda (x) (expand-node-operator node x (problem-f-h problem)))
      (problem-operators problem))))

(defun expand-node-operator (node operator heuristic)
  (mapcar #'(lambda (act)
              (make-node
               :state (action-final act)
               :parent node
               :action act
               :depth (+ (node-depth node) 1)
               :g (+ (node-g node) (action-cost act))
               :h (funcall heuristic (action-final act))
               :f (+
                   (+ (node-g node) (action-cost act))
                   (funcall heuristic (action-final act)))))
(funcall operator node)))
```

8 Ejercicio 7

El séptimo ejercicio es parecido a un viejo amigo. Es muy similiar a alguno de los ejercicios de la primera práctica, en el que había que insertar elementos en una lista en orden. En aquella vez utilizamos como operador el comparador desigualdades de números números reales, esta vez se utiliza como comparador el predicado (o la estrategia) pasado como argumento de entrada a la función.

Código Ejercicio 7

```
(defun insert-nodes (nodes lst-nodes node-compare-p)
  (if (null  nodes)
      lst-nodes
    (insert-nodes (rest nodes) (insert-node-aux
                                (first nodes)
                                lst-nodes
                                node-compare-p) node-compare-p)))

(defun insert-node-aux (node lst-nodes node-compare-p)
  (cond
   ((null lst-nodes) (list node))
   ((funcall node-compare-p node (first lst-nodes))
    (cons node lst-nodes))
   (T (cons
       (first lst-nodes)
       (insert-node-aux node (rest lst-nodes) node-compare-p)))
   ))
```

9 Ejercicio 8

Para el octavo ejercicio se nos pide definir una estrategia para la búsqueda A*, esto es, una función que le diga al algoritmo que nodo expandir primero de la lista de abiertos. Para A*, el criterio para elegir el siguiente a expandir es el que menor f tenga, siendo f = g + h, con g coste real para llegar hasta dicho nodo y h heurística de dicho estado.

Código Ejercicio 8

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

10 Ejercicio 9

Aquí viene la parte esencial de la práctica, donde implementaremos el algoritmo de búsqueda general con la ayuda de las funciones hechas en los apartados anteriores, y comprobaremos el funcionamiento con una estrategia y problema concretos. Siguiendo el pseudocódigo del enunciado de la práctica, codificamos las siguientes funciones:
graph-search, que se encargará de crear el nodo inicial y llamar a la función auxiliar graph-search-aux
graph-search-aux, que se encargará de realizar la recursión (expandir los nodos, ordenarlos en la lista de abiertos, comprobar si estamos expandiendo un nodo que ya esté en la lista de cerrados y en ese caso comprobar cuál tiene menor coste, etc...)
Por último, también se nos pide definir otra función que ejecute el algoritmo usando la estrategia A*, creemos que no hay mucho que comentar de su implementación.

Código Ejercicio 9

```
(defun graph-search-aux (problem open-nodes closed-nodes strategy)
  (let
      ;; Nodo a expandir
      ((exp-node (first open-nodes))
       ;; Nodo de la lista de cerrados equivalente a exp-node
       ;; Puede ser NIL
       (cl-node (first (member (first open-nodes)
                               closed-nodes
                               :test (problem-f-search-state-equal problem)))))
    (cond ((null exp-node)
           NIL)
          ((funcall (problem-f-goal-test problem) exp-node)
           exp-node)
          ((or (null cl-node) (< (node-g exp-node) (node-g cl-node)))
           (graph-search-aux problem (insert-nodes (expand-node exp-node problem) (rest
               open-nodes)
                                                   (strategy-node-compare-p strategy))
                             (insert-nodes (list exp-node) closed-nodes
                                           (strategy-node-compare-p strategy)) strategy))
(T (graph-search-aux problem (rest open-nodes) closed-nodes strategy)))))

(defun graph-search (problem strategy)
  (let ((start-node (make-node
                     :state (problem-initial-state problem)
                     :parent NIL
                     :action NIL
                     :depth 0
                     :g 0
                     :h (funcall (problem-f-h problem) (problem-initial-state problem))
```

```
                    :f (funcall (problem-f-h problem) (problem-initial-state problem)))))
(graph-search-aux problem (list start-node) '() strategy)))

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

11 Ejercicio 10

Se pide implementar, a partir de un nodo que es el resultado de una búsqueda, una función que muestre el camino seguido para llegar al resultado y, adicionalmente, la función que muestre la secuencia de acciones para llegar a dicho nodo.

Código Ejercicio 10

```
;*** solution-path ***

(defun solution-path (node)
  (if (null node)
      NIL
    (append  (solution-path (node-parent node)) (list (node-state node)))))

;*** action-sequence ***
; Visualize sequence of actions

(defun action-sequence (node)
  (if (null (node-parent node))
      NIL
    (append  (action-sequence (node-parent node)) (list (node-action node)))))
```

12 Ejercicio 11

Tras haber implementado ya el algoritmo principal y haberlo probado con la estrategia A*, se nos pide definir otras dos estrategias conocidas en el mundo de los algoritmos de búsqueda, como son la búsqueda en anchura y la búsqueda en profundidad. En la primera, priman los nodos con menor profundidad, mientras que en la segunda se expanden antes los más profundos.

Código Ejercicio 11

```
(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))
```

13 Ejercicio 12

Por último, se nos pide definir una heurística de precio nueva, ya que la que se nos daba es la heurística 0, la cuál es admisible y monótona pero no muy útil si queremos optimizar el tiempo de ejecución. Para definir una heurística más o menos buena hemos ido desde el estado final (Calais) hacia atrás, planteando la ecuación que se tiene que cumplir para que la heurística sea monótona a cada paso. La condición es:

$$h(n) \leq coste(n, n') + h(n')$$

Siendo `h` la función que nos da la heurística de un nodo, y `coste` la función que nos da el coste real de ir de n a n'. Esta desigualdad se tiene que cumplir para cada nodo n y para todos los vecinos n' de cada nodo.

Usando este método obtenemos los siguientes valores de la heurística:

Ciudad	Precio (est.)
Calais	0
Reims	15
Paris	20
Nancy	30
Orleans	55
St.Malo	80
Nantes	95
Brest	100
Nevers	35
Limoges	80
Roenne	35
Lyon	40
Toulouse	10
Avignon	70
Marseille	95

Código Ejercicio 12

```
(defparameter *estimate-new*
  '((Calais (0.0 0.0)) (Reims (25.0 15.0)) (Paris (30.0 20.0))
    (Nancy (50.0 30.0)) (Orleans (55.0 55.0)) (St-Malo (65.0 80.0))
    (Nantes (75.0 95.0)) (Brest (90.0 100.0)) (Nevers (70.0 35.0))
    (Limoges (100.0 80.0)) (Roenne (85.0 35.0)) (Lyon (105.0 40.0))
    (Toulouse (130.0 100.0)) (Avignon (135.0 70.0)) (Marseille (145.0 95.0))))



(defparameter *travel-cost-new*
  (make-problem
   :states *cities*
   :initial-state *origin*
   :f-h #'(lambda (state) (f-h-price state *estimate-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*))
   :operators (list #'(lambda (node) (navigate-canal-price (node-state node) *canals*))
                    #'(lambda (node) (navigate-train-price (node-state node) *trains* *
                        forbidden*)))
   )
  )
```

Comparación de tiempos usando la heurística trivial y una heurística monótona

```
;;; EJECUCION A* USANDO LA HEURISTICA TRIVIAL
CG-USER(45): (time (a-star-search *travel-cheap*))
; cpu time (non-gc) 0.000000 sec user, 0.000000 sec system
; cpu time (gc)     0.015625 sec user, 0.000000 sec system
; cpu time (total)  0.015625 sec user, 0.000000 sec system
; real time  0.021000 sec ( 74.4%)

;;; EJECUCION A* USANDO UNA HEURISTICA MONOTONA
CG-USER(46): (time (a-star-search *travel-cost-new*))
; cpu time (non-gc) 0.000000 sec user, 0.000000 sec system
; cpu time (gc)     0.000000 sec user, 0.000000 sec system
; cpu time (total)  0.000000 sec user, 0.000000 sec system
; real time  0.004000 sec (  0.0%)
```

Vemos que con la nueva heurística la búsqueda tarda la quinta parte del tiempo que tarda con la heurística 0, lo cuál era de esperar. Quizá para una ejecución no se note mucho, ya que son décimas o centésimas de segundo, pero en el caso de que se quiera ejecutar numerosas veces, sin duda una reducción de tiempo del orden de 5 es de agradecer.

1. ¿Por qué se ha realizado este diseño para resolver el problema de búsqueda?
Porque es un diseño muy modularizado y abstracto, con una descomposición funcional muy lograda, lo cuál hace que sea fácil de implementar y muy adaptable a otros problemas similares, no solo al planteado en esta práctica de encontrar el camino más corto entre Marsella y Calais.

2.a. ¿Qué ventajas aporta?
Principalmente permite poder utilizar el mismo software implementado en la resolución de otros problemas, teniendo que cambiar muy pocas líneas (sobre todo las estructuras iniciales de estados y heurísticas).

2.b. ¿Por qué se han utilizado funciones lambda para especificar el test objetivo, la heurística y los operadores del problema?
Las funciones lambda tienen varias ventajas: son de las herramientas más usuales en los lenguajes de programación funcional (Python y Java las implementan, pero no son su "plato fuerte") algo que sí caracteriza

el lenguage Lisp. Por otro lado, favorece la generalización de las estructuras de problema y estrategia, resultando mucho más sencilla una posible migración a otro problema. Por último, las funciones lambda son muy útiles a la hora de utilizar técnicas recursivas (algo que se usa de forma frecuente), y que se ve aún más potenciado con funciones como `mapcar` o `mapcan`

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?
Sí, porque con esta técnica no necesitamos mantener todos los nodos del camino en memoria (con su nombre, acciones y resto de información de la estructura nodo), sino que se guarda una simple referencia, por la cual podemos obtener con un simple *backtracking* el camino por el que se ha alcanzado el nodo objetivo o final (suponiendo un problema finito).

4. ¿Cuál es la complejidad espacial del algoritmo implementado?
La complejidad espacial de A* es:

$$O(b^d)\ con:$$

$$b := factor\ de\ ramificación$$

$$d := \frac{C^*}{\epsilon}\ con:$$

$$C^* := coste\ óptimo$$

$$\epsilon := mínimo\ coste\ por\ acción$$

5. ¿Cuál es la complejidad temporal del algoritmo?
Idéntico al apartado anterior (misma complejidad que la espacial).

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).
Entendemos que estamos hablando de limitar el número de trenes que se pueden coger, ya que son los caminos bidireccionales. La mejor manera que se nos ocurre de implementar esa funcionalidad sería modificar las funciones navigate-train-time y navigate-train-price (y consecuentemente navigate y navigate-aux), haciendo que reciban el número máximo de trenes que se pueden coger, y dentro de cada nodo llevar un contador del número de veces que se ha usado la acción navigate-train. Al aplicar la acción navigate-train, se miraría si el número de trenes que se han tomado para llegar a ese nodo es el máximo, en ese caso no se generaría un nuevo nodo por esa acción. En caso de que el número de trenes usados sea menor que el máximo, se generan los nuevos nodos incrementando el contador de trenes usados.

14 Conclusión

En esta práctica hemos implementado el algoritmo de búsqueda A*, pero con la lección que nos quedamos de todo esto es que una buena descomposición funcional puede convertir días de programación en horas de planificación.
El claro ejemplo de ello es que no hemos tenido que hacer pseudocódigo (por esa razón no se adjunta, no tendría sentido hacerlo una vez ya programadas las funciones). Podríamos haber necesitado realizar un esquema del funcionamiento del algoritmo A*, pero dicho pseudocódigo estaba incluido en la documentación de la práctica.