Memoria P2 IA - Búsqueda

Grupo 2363

Antonio Solana Vera y Pablo Sánchez Redondo

27/02/2020

Contents

Entorno de compilación

Hemos utilizado Emacs (portacle) para compilar y probar nuestras funciones.

Ejercicios

Ejercicio 1

Esta fución simplemente devuelve el valor asignado a cada una de las ciudades, para esto usamos el assoc que busca la tupla coincidente.

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

Ejercicio 2

Primero elimina todos los elementos que no tengan la ciudad y luego crea todas las acciones posibles desde ella.

```
(defun navigate (city lst-edges)
    (mapcar #'(lambda (x)
    (make-action
      :name 'UselessName
      :origin (nth 0 x)
      :final (nth 1 x)
      :cost (nth 2 x)))
            (remove-if-not #'(lambda (x) (eq (car x) city)) lst-edges)))
```

Ejercicio 3

En la función principal comprobamos que la ciudad es una de las ciudades de destino, y luego, si es así, llamamos a la función recursiva auxiliar. Ésta comprueba que no es el principio (que no tiene padre), elimina la ciudad de la lista de mandatory si estuviera y vuelve a llamar a la función.

La condición de salida (no hay padre), comprueba después que mandatory está vacío y si lo esta devuelve T y si no NIL.

```
(defun f-goal-test (node destination mandatory)
  (if (not (null (member (node-city node) destination)))
      (f-goal-test-aux node mandatory)
      NIL))

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

```
    (f-goal-test-aux (node-parent node) (remove-if
      #'(lambda (x) (equal (node-city node) x)) mandatory))))
```

Ejercicio 4

Primero comprobamos que las ciudades de los nodos son iguales y luego, usando f-goal-test, comprobamos que visitan todas las ciudades *mandatory*.

```
(defun f-search-state-equal (node-1 node-2 &optional mandatory)
  (and
    (equal (node-city node-1) (node-city node-2))
  ; Check both cities are the same
      (if mandatory ; If there is a mandatory list
        (and (f-goal-test node-1 (list (node-city node-1)) mandatory)
      (f-goal-test node-2 (list (node-city node-2)) mandatory))
      ; Check both nodes
          T))) ; No mandatory list means we are good
```

Ejercicio 5

Definimos el parámetro travel con las funciones definidas anteriormente.

```
(defparameter *travel*
    (make-problem
        :cities *cities*
        :initial-city *origin*
        :f-h #'f-h
        :f-goal-test #'f-goal-test
        :f-search-state-equal #'f-search-state-equal
        :succ #'navigate))
```

Ejercicio 6

Con un mapcar, creamos cada uno de los nodos usando la función de navigate.

```
(defun expand-node (node problem)
    (mapcar
        #'(lambda (act) (make-node
            :city (action-final act)
            :parent node
            :action act
            :depth (+ 1 (node-depth node))
            :g (+ (action-cost act) (node-g node))
            :h (f-h (action-final act) *heuristic*)
            :f (+ (action-cost act) (f-h (action-final act) *heuristic*))))
        (navigate (node-city node) *trains*)))
```

Ejercicio 7

Usando sort, con la estrategia que se pasa por argumento, ordenamos la lista generada por ambas listas.

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

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

Ejercicio 8

Conociendo el algoritmo A* creamos el node-compare-p.

Elige el camino según sea el menor actualmente, por lo tanto:

```
(defparameter *A-star*
    (make-strategy
        :name 'A-star
        :node-compare-p #'(lambda (x y) (< (node-f x) (node-f y)))))
```

Ejercicio 9

Este algoritmo junta todo lo realizado anteriormente para encontrar un camino. Depende del destino y las ciudades obligatorias marcadas como parámetros.

```
(defun exp-cond (node closed)
    (let ((found-node (find node closed :test #'f-search-state-equal)))
        (if (NULL found-node)
            T
            (< (node-g node) (node-g found-node)))))

(defun graph-search-aux (problem strategy open closed goal-test)
    (if (or (NULL open) (NULL (first open)))
        NIL
        (if (funcall goal-test (first open) *destination* *mandatory*)
            (first open) ; Are we there yet?
            (if (exp-cond (first open) closed)
                (graph-search-aux
                    problem
                    strategy
                    (insert-nodes-strategy
            (expand-node (first open) problem) (rest open) strategy)
                    (cons (first open) closed)
                    goal-test)
                (graph-search-aux
          problem
          strategy
          (rest open) closed goal-test)))))
```

```
(defun graph-search (problem strategy)
    (graph-search-aux
        problem
        strategy
        (list (make-node
    :city (problem-initial-city problem)
    :parent NIL
    :action NIL))
        NIL
        #'f-goal-test))
```

Ejercicio 10

Imprime el camino recorrido subiendo por los nodos padre.

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

Ejercicio 11

Estrategias de profundidad y anchura.

```
(defparameter *depth-first*
    (make-strategy
        :name 'depth-first
        :node-compare-p #'(lambda (x y) (> (node-depth x) (node-depth y)))))

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

Ejercicio 12

Heuristicas nueva y cero.

```
(defparameter *heuristic-new*
    '((Calais 5.0) (Reims 5.0) (Paris 4.0)
    (Nancy 5.0) (Orleans 3.0) (St-Malo 3.0)
    (Nantes 2.0) (Brest 3.0) (Nevers 3.0)
    (Limoges 2.0) (Roenne 0.0) (Lyon 2.0)
    (Toulouse 1.0) (Avignon 1.0) (Marseille 0.0)))

; Heuristic defined in exercise 12
(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)))
```

Cuestiones

1. a. Es fácil actualizar la heurística y el sistema de nodos es sencillo y no ocupa mucha memoria. A parte puedes cambiar la estrategia de camino y estudiar otras posibilidades fácilmente.

   b. Para luego poder fácilmente cambiar esos datos y que no haya que cambiar todo el código entero a la hora de probar nuevos caminos.

2. Dado que son referencias al nodo padre y no guarda el nodo padre en sí

3. Un nodo por cada ciudad y una accion por cada camino entre ciudades. Y cada búsqueda tiene un único problema.

4. O(log h^*(x)) Donde h^* es la heurística perfecta desde x a la meta.