Memoria P1 IA

Grupo 2363

Antonio Solana Vera y Pablo Sánchez Redondo

27/02/2020

Índice

Entorno de compilación

Hemos utilizado Emacs (portacle) para compilar y probar nuestras funciones. Ha habido un problema, y como se puede observar, al calcular el coseno de $\frac{\pi}{2}$ el compilador no devuelve exactamente 0, lo que hace que la función no devuelva NIL.

Ejercicios

Ejercicio 1

La función `newton` es en la que hemos podido comprobar el error mencionado anteriormente. Al probar el caso de $\frac{\pi}{2}$ devuelve un número extraño.

```
(defun newton (f df-dx max-iter x0 &optional (tol-abs 0.0001))
  (if  (= max-iter 0) nil
    (let ((xn (- x0 (/ (funcall f x0) (funcall df-dx x0)))))
    (if (< (abs (- x0 xn) tol-abs))
      xn
      (newton f df-dx (- max-iter 1) xn tol-abs)))))
```

Para newton-all usamos un mapcar que aplica newton a cada una de los seeds.

```
(defun newton-all (f df-dx max-iter seeds &optional (tol-abs 0.0001))
  (mapcar #'(lambda (x0)
    (newton f df-dx max-iter x0 tol-abs)) seeds))
```

Ejercicio 2

a. Por cada elemento de lst lo metemos en una lista con elt.

```
(defun combine-elt-lst (elt lst)
  (mapcar #'(lambda (x) (list elt x))
      lst))
```

b. Por cada elemento de lst1 le aplicamos la función anterior con lst2.

```
(defun combine-lst-lst (lst1 lst2)
  (reduce #'append
    (mapcar #'(lambda(elt) (combine-elt-lst elt lst2)) lst1)))
```

c. A cada elemento de cada lista se le añaden el resto de elementos del resto de listas, hacemos esto cogiendo y llamando recursivamente a la funciión, y cuando se encuentre con una sola lista, hace una lista con cada elemento, y luego va subiendo.

```
(defun combine-list-of-lsts (lolsts)
    (if (null lolsts)
        nil
        (if (null (rest lolsts))
            (mapcar #'list (first lolsts))
            (let ((n1 (combine-list-of-lsts (rest lolsts))))
                (apply #'append
                    (mapcar #'(lambda(lst)
                            (mapcar #'(lambda(x) (cons x lst)) (first lolsts))
                        )
                        n1))))))
```

Ejercicio 3

Producto escalar:

```
(defun scalar-product (x y)
        (reduce #'+ (mapcar #'* x y)))
```

Norma euclídea:

```
(defun euclidean-norm (x)
    (sqrt (scalar-product x x)))
```

Distancia euclídea:

```
(defun euclidean-distance (x y)
        (euclidean-norm (mapcar #'- x y)))
```

Ejercicio 4

Similitud de coseno, llamamos a la función anterior para comprobar que no son de norma 0

```
(defun cosine-similarity (x y)
        (let ((xn (euclidean-norm x)) (yn (euclidean-norm y)))
            (if (or (= 0 xn) (= 0 yn)) nil
                (/ (scalar-product x y) (* xn yn)))))
```

Ejercicio 5

Sim-map coge todos los vectores y calcula la similitud, metiendolo en tuplas dentro de una lista. Select vectors quita aquellos vectores que no cumplan el threshold de la lista que devuelve sim-map.

```
(defun sim-map (lst-vectors vector fun)
    (if (null lst-vectors)
        ()
```

```
        (cons
            (cons (first lst-vectors) (funcall fun (first lst-vectors) vector))
            (sim-map (rest lst-vectors) vector fun))))

(defun select-vectors (lst-vectors test-vector similarity-fn &optional (threshold 0))
    (sort
        (remove-if #'(lambda(x) (< (rest x) threshold))
          (sim-map lst-vectors test-vector similarity-fn))
          #'(lambda(x y) (> (rest x) (rest y)))))
```

Ejercicio 6

Nearest neighbor: (Con funciones auxiliares, utiliza sim-map)

```
(defun lowest-aux (map-lst-vector lowest)
    (if (null map-lst-vector)
        lowest
        (let (
            (last-lowest (cdr lowest))
            (new-lowest (cdr (first map-lst-vector)))
            (new-lowest-cmp (first map-lst-vector)))
            (if (< last-lowest new-lowest)
                (lowest-aux (rest map-lst-vector) lowest)
                (lowest-aux (rest map-lst-vector) new-lowest-cmp)))))

(defun get-lowest (map-lst-vector)
    (lowest-aux map-lst-vector (cons '(0 0 0) 2.0)))

(defun nearest-neighbor (lst-vectors test-vector distance-fn)
    (get-lowest (sim-map lst-vectors test-vector distance-fn)))
```

Ejercicio 7

Utilizando some y every buscamos todas las reglas que tengan de segundo miembro el `goal` y si alguna (some) de ellas, todos los átomos (every) devuelven True entonces la función devuelve True. En cualquier otro caso, la función devuelve NIL, que es el equivalente a False en Lisp.

```
(defun backward-chaining-aux (goal lst-rules pending-goals)
  (if (not (null (member goal pending-goals)))
    NIL
      (if (some #'(lambda(x) (and (equal (nth 1 x) goal) (null (first x))))
        lst-rules)
          T
          (let
          ((rules (remove-if #'(lambda(x) (not (equal (nth 1 x) goal))) lst-rules)))
            (if (null rules)
              NIL
                (if (some
```

```
                #'(lambda(rule) (equal T (every #'(lambda(x)
                  (equal
                    (backward-chaining-aux x
                      (remove-if #'(lambda(x) (equal x rule)) lst-rules)
                        (append pending-goals (list goal))) 'T))
                          (first rule)
                          )))
                          rules
                    )
                    T
                    NIL))))))
```

```
(defun backward-chaining (goal lst-rules)
    (backward-chaining-aux goal lst-rules NIL))
```

Ejercicio 8

Para encontrar el camino de a 'a' 'f', que es el único que se puede realizar, pues a partir de f no se puede seguir el resultado es (a) –> (a d) –> (a d f)

En el caso de otro grafo dirigido, por ejemplo, un árbol binario completo con 9 hojas, el camino del 1 al 4 sería:

(1) –> (1 2) –> (1 3) –> (1 2 4)

```
0: (BFS D ((C)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
   1: (BFS D ((E C)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
     2: (BFS D ((B E C) (F E C)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
       3: (BFS D ((F E C) (D B E C) (F B E C))
         ((A D) (B D F) (C E) (D F) (E B F) (F)))
         4: (BFS D ((D B E C) (F B E C)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
         4: BFS returned (C E B D)
       3: BFS returned (C E B D)
     2: BFS returned (C E B D)
   1: BFS returned (C E B D)
 0: BFS returned (C E B D)
```

Busca y añade todos los caminos posibles a la lista de caminos, y en cuanto encuentra todos los caminos devuelve el más corto.

El problema con este algoritmo es que si hay algún ciclo se estancará en un bucle y sería necesario incluir una condición de salida para que esto no pase.