MEMORIA PRÁCTICA 1

Alejandro Santorum - alejandro.santorum@estudiante.uam.es
Sergio Galán Martín - sergio.galanm@estudiante.uam.es
Inteligencia Artificial
Práctica 1 Pareja 9

26 de febrero de 2019

Contents

1 Introducción

Nos encontramos ante la primera práctica de Inteliencia Artificial. Nuestro objetivo será comenzar en el mundo de la programación funcional utilizando el lenguaje Common Lisp.

Los tres primeros ejercicios propuestos para la práctica son básicamente una introducción a dicho lenguaje. Por otra parte, los ejercicio 4 y 5 ya son más avanzados y, posiblemente, nos traerán algún dolor de cabeza.

2 Ejercicio 1

El primer ejercicio se basa en la idea de clasificar una página web en función de la frencuencia de ciertas palabras que suelen abundar en ciertas clases de páginas.
Aparte de ser una buena introducción a dicho lenguaje, también nos sirve como primer acercamiento a los algoritmos de aprendizaje automático que se usan en la industria. Este es un ejemplo sencillo de algoritmo de clasificación.

Comentar que en el apartado 1.2 se ha considerado la confianza como el coseno del ángulo que forman los vectores, es decir:

$$confidence = 1 - cosine\_distance = 1 - (1 - cos(\theta)) = cos(\theta)$$

De esta forma la función de confianza tendrá valores entre -1 y 1. Dos vectores empezarán a considerarse semejantes si la confianza calculada es positiva y no semenajes o de semejanza contraria si su confianza es negativa. Esto no contradice que el nivel de confianza con que se filtra estea entre 0 y 1, ya que si dos vectores son de confianza negativa, ya ni se plantea si pueden ser semejantes o no, directamente son descartados.

Ejemplos de ejecución Ejercicio 1.1 usando recursón

```
(cosine-distance-rec '(1 2) '(1 2 3)) ;; ---> 0.40238577
(cosine-distance-rec nil '(1 2 3)) ;; ---> 1
(cosine-distance-rec '()'()) ;; ---> 0
(cosine-distance-rec '(0 0) '(0 0)) ;; ---> 0
```

Ejemplos de ejecución Ejercicio 1.1 usando mapcar

```
(cosine-distance-mapcar '(1 2) '(1 2 3)) ;; ---> 0.40238577
(cosine-distance-mapcar nil '(1 2 3)) ;; ---> 1
(cosine-distance-mapcar '()'()) ;; ---> 0
(cosine-distance-mapcar '(0 0) '(0 0)) ;; ---> 0
```

Ejemplos de ejecución Ejercicio 1.2

```
(order-vectors-cosine-distance '(1 2 3) '()) ;; ---> NIL
(order-vectors-cosine-distance '() '((4 3 2) '(1 2 3))) ;; ---> NIL
```

Ejemplos de ejecución Ejercicio 1.3

```
(get-vectors-category '(()) '(()) #'cosine-distance);; ---> ((NIL))
(get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'
    cosine-distance) ;; ---> ((1 0.52190864))
(get-vectors-category '(()) '((1 1 2 3) (2 4 5 6)) #'
    cosine-distance) ;; ---> ((NIL 1) (NIL 1))
;; Comentarios sobre las salidas: Al introducir una lista de
    categor as nula, las ditancias entre cualquier vector y el
    vector nulo es 1 debido a como las hemos definido en el
    apartado 1.1
```

Ejemplos de ejecución Ejercicio 1.4

```
 >> (setf categories '((1 43 23 12) (2 33 54 24)))
    ;; ---> ((1 43 23 12) (2 33 54 24))

 >> (setf texts '((1 3 22 134) (2 43 26 58)))
    ;; ---> ((1 3 22 134) (2 43 26 58))

 >> (time (get-vectors-category categories texts #'
     cosine-distance-rec))
    ; 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  0.000000 sec
    ; space allocation:
    ;  12 cons cells, 976 other bytes, 0 static bytes
    ; Page Faults: major: 0 (gc: 0), minor: 0 (gc: 0)
    ;; ---> ((1 0.67117083) (1 0.18444914))

 >> (time (get-vectors-category categories texts #'
     cosine-distance-mapcar))
    ; 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  0.000000 sec
    ; space allocation:
    ;  66 cons cells, 976 other bytes, 0 static bytes
    ; Page Faults: major: 0 (gc: 0), minor: 0 (gc: 0)
    ;; ---> ((1 0.67117083) (1 0.18444914))
```

Como se puede apreciar, los ejemplos de ejecución aportados no son muy pesados, por lo que aportamos nosotros uno con categorías y textos de mayor dimensión.

Ejemplos de ejecución Ejercicio 1.4 (mayor cómputo)

```
 >> (setf categories '((1 23 45 65 21 34 54 23 34 22) (2 33 52 21
     23 45 47 31 11 1) (3 3 51 6 11 14 64 73 48 29)))
    ;; ---> ((1 23 45 65 21 34 54 23 34 22) (2 33 52 21 23 45 47 31
         11 1) (3 3 51 6 11 14 64 73 48 29))

 >> (setf texts '((1 13 25 35 41 54 64 73 84 92) (2 3 43 61 31 14
     65 87 92 1) (3 31 5 5 11 76 44 38 100 12) (4 3 1 23 67 94 41
     33 34 42)))
    ;; ---> ((1 13 25 35 41 54 64 73 84 92) (2 3 43 61 31 14 65 87
        92 1) (3 31 5 5 11 76 44 38 100 12) (4 3 1 23 67 94 41 33
        34 42))
```

```
6
7  >> (time (get-vectors-category categories texts #'
       cosine-distance-rec))
8 ; cpu time (non-gc) 0.000000 sec user, 0.000000 sec system
9 ; cpu time (gc)     0.000000 sec user, 0.000000 sec system
10 ; cpu time (total)  0.000000 sec user, 0.000000 sec system
11 ; real time  0.003000 sec (  0.0%)
12 ; space allocation:
13 ;  32 cons cells, 3,392 other bytes, 0 static bytes
14 ; Page Faults: major: 0 (gc: 0), minor: 0 (gc: 0)
15 ((1 0.20380217) (1 0.17340422) (1 0.33719987) (1 0.30565596))
16
17  >> (time (get-vectors-category categories texts #'
       cosine-distance-mapcar))
18 ; cpu time (non-gc) 0.000000 sec user, 0.000000 sec system
19 ; cpu time (gc)     0.000000 sec user, 0.000000 sec system
20 ; cpu time (total)  0.000000 sec user, 0.000000 sec system
21 ; real time  0.004000 sec (  0.0%)
22 ; space allocation:
23 ;  572 cons cells, 3,392 other bytes, 0 static bytes
24 ; Page Faults: major: 0 (gc: 0), minor: 0 (gc: 0)
25 ((1 0.20380217) (1 0.17340422) (1 0.33719987) (1 0.30565596))
```

Ahora al menos parece que las ejecuciones han tardo un poco más, al menos apreciable. Resulta que usando `cosine-distance-rec` obtenemos un tiempo real de 0.003 sec y usando `cosine-distance-mapcar` se tiene un tiempo de 0.004 sec. Parece que la implementación recursiva es más eficiente que la que usa la función de lisp mapcar.

3 Ejercicio 2

En este ejercicio se nos pide implementar el conocido método para la búsqueda de un cero de una función llamado método de Newton-Raphson. Es uno de los métodos recursivos por excelencia en matemáticas, lo cuál lo hace buen ejemplo para implementar en Lisp. Además, se generaliza dándole múltiples semillas para aumentar la probabilidad de encontrar todos los ceros. Sobre decisiones de diseño que hayamos tomado de cara a implementar el ejercicio, no hay ninguna lo suficientemente significativa como para ser digna de mención en la memoria.

Ejemplos de ejecución Ejercicio 2

```
1 (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
2 #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 3.0) -> 4.0
3 (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
4 #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 0.6) -> 1.0
5 (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
6 #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 30 -2.5) -> -3.0
7 (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
8 #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 10 100.0) -> NIL
9
10 (one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
11 #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 -2.5)) ->
       1.0
12 (one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
```

```
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(3.0 -2.5)) -> 4.0
(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(3.0 -2.5)) -> NIL

(all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 -2.5)) ->
    (1.0 4.0 -3.0)
(all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 10000)) ->
    (1.0 4.0 nil)
(all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(0.6 3.0 -2.5)) -> (
    nil nil nil)

(list-not-nil-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))
    )
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 10000)) ->
    (1.0 4.0)
```

4 Ejercicio 3

Este ejercicio trata sobre el manejo de listas de Lisp o *list comprehension*. En los dos primeros apartados no hay nada que comentar, todo en orden.
En el tercer apartado la intuición nos decía que se debían utilizar las dos funciones anteriores para implementar la tercera (`combine-lst-of-lsts (lstoflsts) ...`), no obstante, después de unos cuantos minutos de reflexión y pruebas, nos hemos dado de cuenta que no era así. Por esta razón, se han codificado dos funciones auxiliares y además, es obligatorio resaltar el curioso caso base de la función mencionada:

Caso base combine-lst-of-lsts

```
(if (null lstolsts)
    '(NIL)
```

Una manera de ver el por qué de este caso base es hacerse la analogía con el resto de ejercicios. Cuando se recibía una lista de elementos (sobre todo números), el caso base era devolver la lista vacía. En este caso, como la lista es de listas, el caso base es devolver la lista con NIL (no vacía, esta lista tiene el vacío, por lo tanto su cardinal es uno y no cero como puede uno pensar).

Ejemplos de ejecución Ejercicio 3.1

```
(combine-elt-lst 'a nil) ;; ---> NIL
(combine-elt-lst nil nil) ;; ---> NIL
(combine-elt-lst nil '(a b)) ;; ---> ((NIL A) (NIL B))
;; Comentarios: en el primer ejemplo se evalua a NIL porque la
    lista introducida es NIL. En cambio, en el tercer ejemplo, se
    combina el ELEMENTO NIL con la lista (a b), de ah  que si se
    obtenga un resultado no nulo.
```

Ejemplos de ejecución Ejercicio 3.2

```
(combine-lst-lst nil nil) ;; ---> NIL
```

```
(combine-lst-lst '(a b c) nil) ;; ---> NIL
(combine-lst-lst nil '(a b c)) ;; ---> NIL
```

Ejemplos de ejecución Ejercicio 3.3

```
(combine-list-of-lsts '(() (+ -) (1 2 3 4))) ;; ---> NIL
(combine-list-of-lsts '((a b c) () (1 2 3 4))) ;; ---> NIL
(combine-list-of-lsts '((a b c) (1 2 3 4) ())) ;; ---> NIL
(combine-list-of-lsts '((1 2 3 4))) ;; ---> (1 2 3 4)
(combine-list-of-lsts '(nil)) ;; ---> NIL
(combine-list-of-lsts nil) ;; ---> NIL
```

5 Ejercicio 4

Y llegamos al "hueso" de la práctica; su valoración de 5 puntos ya hacía sospechar en un primer momento, algo que se confirmó cuando nos pusimos manos a la obra.

Sin lugar a dudas lo más complicado era hacerse la idea de como podría que estar representado el árbol de verdad en el código. Eso y junto con el díficil manejo de las listas de este lenguaje, al cual aún no estamos (¿estábamos?) acostumbrados, hizo que este ejercicio se resistiera por unos días.

Poco a poco se fue avanzando, diseñando y luego implementando las funciones de inferencia necesarias para reducir cualquier expresión introducida a una expresión con solo operandores *and* y *or* (sin contar que algún literal podería estar negado).

Por último, se unió todo en una única función recursiva (`expand (...)`) que expandía por completo, haciendo llamadas anidadas a las funciones de derivación, la expresión introducida y se construía el árbol de verdad en su totalidad.

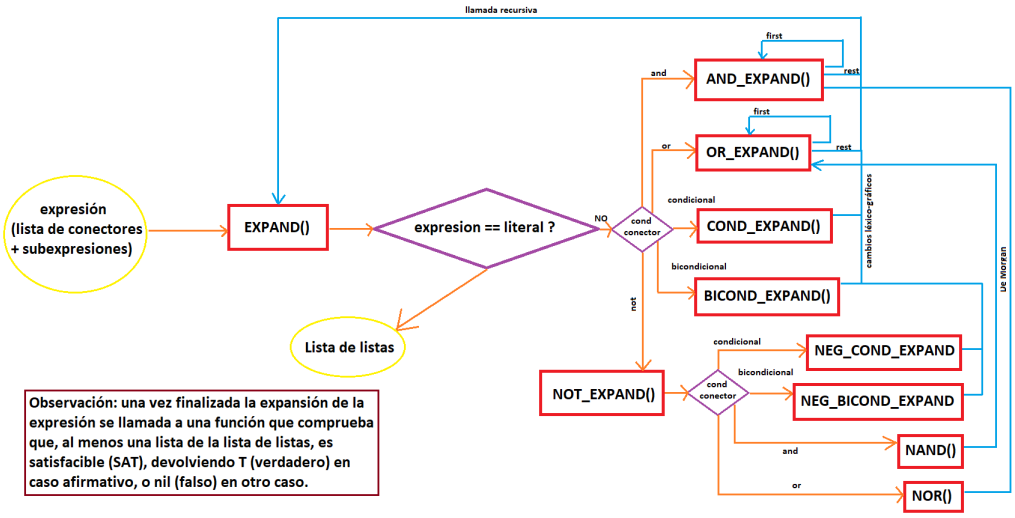
En nuestro caso el árbol estaba representado como una lista de listas, donde cada sublista representaba una rama (o camino desde nodo raíz a cualquier nodo terminal).

Una vez hecho esto, se programaron otras funciones que comprobaban si era alguna rama SAT o, por el contrario, si todas eran UNSAT. Combinando la función que desarrolla el árbol, junto con la/las función/es que determinan si es SAT o UNSAT, se pudo completar el ejercicio propuesto.

Es importante comentar que la función que construye el árbol (`expand()`) puede ser mejorada implementando la tala de las ramas que ya se considerarían UNSAT, pero no se ha optimizado debido a que el usuario puede desear obtener la expresión del árbol en su totalidad, por lo que se ha omitido dicha mejora, dejándole el trabajo sucio a las funciones que comprueban si la expresión es satisfacible o insatisfacible analizando la expresión del árbol.

Llegado este momento se obligatorio comentar que no se ha registrado ni el código implementado ni su pseudocódigo ya que las funciones programadas son relativamente sencillas y el nombre de las variables y/o funciones ya describen bastante bien el funcionamiento del algoritmo.

No obstante, para este ejercicio no podemos sostener esto, por lo que hemos diseñado un pequeño y esquematizado diagrama de flujo de la función más complicada (a nuestro parecer) de la práctica: `expand()`.

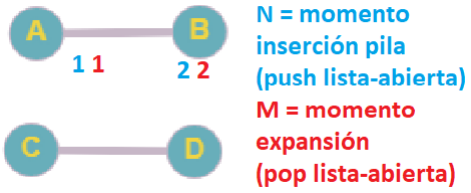


En el diagrama podemos percibir cual el la secuencia de llamadas entre funciones auxiliares para expandir la expresión. Comentar que en el diagrama no se recogen los caso base de las funciones `and_expand()` y `or_expand()`, pero son básicamente devolver **'(NIL)** y **NIL** respectivamente.

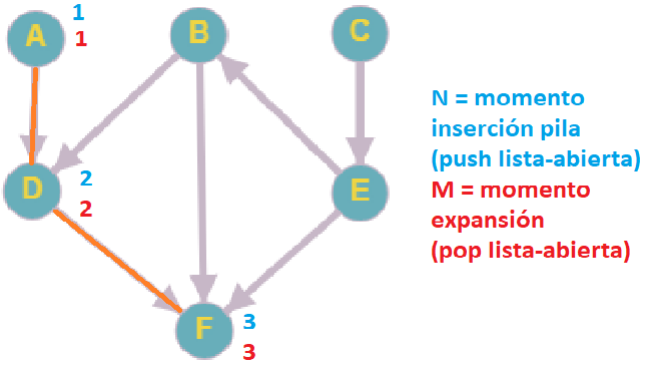
6 Ejercicio 5

6.1 Ejercicio 5.1

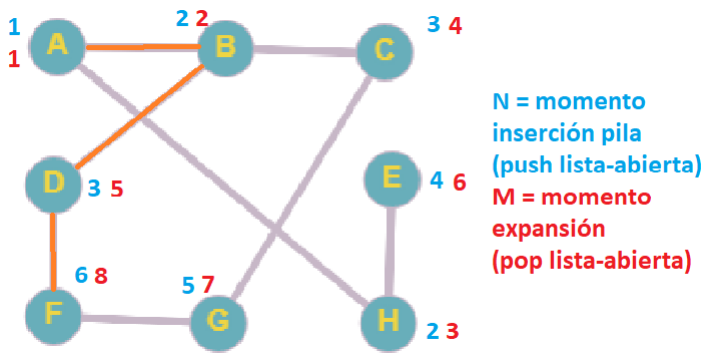
a) Grafos especiales: En primer lugar comentamos los casos en los que el grafo es vacío, tiene un elemento o es no conexo. En el caso de que el grafo sea vacío, el algoritmo no entra siquiera en el bucle ya que la lista es vacía, devolviendo error. Por otro lado, cuando el grafo contiene un solo nodo, si el nodo objetivo es diferente al único, se evalúa error y en caso contrario, devuelve el nodo inicial (que es el único camino).
Por último, en el caso de un grafo no conexo, si se pide como nodo final uno que no está en la componente conexa del nodo raíz evalúa error. Mostramos abajo una imagen ejemplificado esto, buscando desde el nodo A el nodo C (mismo resultado con D como nodo final).



b) Caso típico (grafo dirigido ejemplo): En la siguiente imagen ilustramos la ejecución del algoritmo *Breadth First Search* con el grafo ejemplo aportado. El nodo raíz es A y el nodo objetivo es F. Para otros nodos inicio-objetivo el algoritmo se comportaría de forma análoga, salvo que no exista camino al tratarse de un grafo dirigido.



c) Caso típico (propia cosecha): Esta vez hemos diseñado nosotros el grafo (no dirigido). En él ilustramos la ejecución de dicho algoritmo comenzando con el nodo A y buscando el nodo F. En este caso, al tratarse de un grafo no dirigido y conexo, siempre existe un camino entre dos nodos dados, y la ejecución sería completamente análoga.



Aclaración: Todas estas ejecuciones y comentarios están basados en el algoritmo `bfs` mejorado con eliminación de estados repetidos.

6.2 Ejercicio 5.2

Pseudocódigo BFS

```
Sea C cola y n nodo inicial
C.encolar(n)
camino.add(n)
mientras C no vacia:
    m = C.pop()
    for v vecinos de m:
        C.encolar(v)
        camino.add(v) //Se van generando distintos posibles caminos
             en cada ramificacion
        if v es destino
            return camino
return error
```

6.3 Ejercicio 5.5

Shortest path

```
( defun shortest-path ( start end net )
( bfs end ( list ( list start )) net ))
```

Esta función encuentra el camino más corto entre *start* y *end*, ya que llama a bfs con único nodo inicial desde el que queremos empezar, y el algoritmo bfs va a ir explorando los nodos a distancia 1 buscando el nodo end, si no lo encuentra va a los nodos a distancia 2 de start... Así hasta llegar a end, y será el camino más corto porque ha ido comprobando todas las posibilidades de caminos que tiene a cada paso. Es decir, si hubiese otro camino más corto, tendría que haberlo encontrado en pasos anteriores.

6.4 Ejercicio 5.6

El resultado de hacer trace de todas las funciones implicadas es el siguiente:

Trace

```
>> ( shortest-path 'a 'f '(( a d ) ( b d f ) ( c e ) ( d f ) ( e b
    f ) ( f )))
 0[6]: (SHORTEST-PATH A F ((A D) (B D F) (C E) (D F) (E B F) (F)))
   1[6]: (BFS F ((A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
     2[6]: (NEW-PATHS (A) A ((A D) (B D F) (C E) (D F) (E B F) (F))
         )
     2[6]: returned ((D A))
     2[6]: (NEW-PATHS (D A) D ((A D) (B D F) (C E) (D F) (E B F) (F
         )))
     2[6]: returned ((F D A))
   1[6]: returned (A D F)
 0[6]: returned (A D F)
(A D F)
```

Con esto vemos que el programa hace, en efecto, lo que se esperaba. Expande A, teniendo como único camino posible D. Tras ello, vuelve a expandir, ahora D, que nos da F como único camino, y como es el destino, devuelve el camino que ha ido formando.

6.5 Ejercicio 5.7

La llamada que hay que hacer es:

Llamada

```
(shortest-path 'b 'g '((a b c d e) (b a d e f) (c a g) (d a b g h)
    (e a b g h) (f b h) (g c d e h) (h d e f g)))
```

Y el resultado es:

Resultado

```
(b d g)
```

6.6 Ejercicio 5.8

Un ejemplo que ilustra el fallo del código proporcionado puede ser la siguiente llamada a shortest-path:

Llamada

```
(shortest-path 'a 'c ((a b) (b a) (c d) (d c)))
```

Es decir, el problema se da, como se dice en el enunciado, cuando se pide buscar un camino inexistente en un grafo con ciclos.

Para solucionarlo, lo que hemos decidido implementar es ir guardando en una lista auxiliar los nodos ya expandidos, y haciendo una comprobación antes de expandir cada nodo de si está en dicha lista o no. En caso de que esté en la lista,

simplemente no hacemos nada con él, y en caso de que la cola quede vacía evaluamos a NIL, porque habremos llegado a una situación en la que la conclusión es que no hay camino posible entre los dos nodos suministrados.