Memoria Practica 1 Inteligencia Artificial

1. Distancia coseno

1.1. Responde en la memoria con el resultado de evaluar los siguientes casos:

1.- (cosine-distance ‘(1 2) ‘(1 2 3)): El resultado de evaluar este caso es tanto en la funcion que hace uso de mapcar como en la recursiva de 0.40238565.

2.- (cosine-distance nil ‘(1 2 3)): El resultado de evaluar este caso es en ambas funciones 0. Al principio obteniamos un error (division por cero), ya que solo contemplabamos el caso base en la funcion de recursion, pero lo hemos corregido aniadiendo un control de erroes tambien en la funcion principal.

3.- (cosine-distance nil nil): El resultado obtenido es el mismo que en el apartado anterior, ya que al no pasar el primer filtro de que x debe ser distinto de nil en la primera recursion, devolvemos 0.

4.- (cosine-distance ‘(0 0) ‘(0 0)): El resultado que obteniamos al principio un error debido a la division entre 0. Para abordar este problema decidimos usar un let que guardara el divisor y antes de dividir realizamos una comprobacion de si este es 0.

1.2. Responde en la memoria con el resultado de evaluar los siguientes casos:

1.- (order-vectors-cosine-distance ’(1 2 3) ’()) : El resultado de evaluar este caso es nil.
2.- (order-vectors-cosine-distance ’() ’((4 3 2) (1 2 3))): El resultado de evaluar este caso es ((4 3 2) (1 2 3))

1.4. Haz pruebas llamando a get-vectors-category con las distintas variantes de la distancia coseno y para varias dimensiones de los vectores de entrada. Verifica y compara tiempos de ejecución. Documenta en la memoria las conclusiones a las que llegues.

Los resultados obtenidos al ejecutar el siguiente codigo:

```
(setf categories '((1 43 23 12) (2 33 54 24)))
(setf texts '((1 3 22 134) (2 43 26 58)))
(time (get-vectors-category categories texts #'cosine-distance-rec))
(time (get-vectors-category categories texts #'cosine-distance-mapcar))
```

Real time: 9.3E-5 sec.
Run time: 0.0 sec.
Space: 2624 Bytes
Real time: 5.1E-5 sec.
Run time: 0.001 sec.
Space: 3648 Bytes

En los tiempos de ejecucion podemos observar que la primera llamada a la funcion get-vectors-category es mas optima que la segunda. Creemos que esto se puede deber o bien a que la primera funcion esta programada de manera mas optima o que al hacer uso de mapcar en la segunda funcion, esta sea menos optima al tener que hacer llamadas a librerias del sistema. De todas maneras pensamos que son datos poco significativos y deberiamos hacer pruebas con vectores mas grandes para obtener datos mas concluyentes.

Responde en la memoria con el resultado de evaluar los siguientes casos:

1.- (get-vectors-category ‘(()) ‘(()) #’cosine-distance-rec): El resultdo obtenido es ((NIL 0)).

2.- (get-vectors-category ‘((1 4 2) (2 1 2)) ‘((1 1 2 3)) #’cosine-distance-rec): El resultado obtenido es ((2 0.39753592)).

3.- (get-vectors-category ‘(()) ‘((1 1 2 3) (2 4 5 6)) #’cosine-distance-rec): El resultado obtenido es ((NIL 0) (NIL 0))

3. Combinacion de listas

3.1 Responde en la memoria con el resultado de evaluar los siguientes casos:

1.- (combine-elt-lst ‘a nil): El resultado obtenido es NIL.

2.- (combine-elt-lst nil nil): El resultado obtenido es NIL

3.- (combine-elt-lst nil ‘(a b)): El resultado obtenido es: ((NIL A) (NIL B))

3.2 Responde en la memoria con el resultado de evaluar los siguientes casos:

1.- (combine-lst-lst nil nil): El resultado obtenido es NIL.

2.- (combine-lst-lst ‘(a b c) nil): El resultado obtenido es NIL.

3.- (combine-lst-lst nil ‘(a b c)): El resultado obtenido es NIL.

3.3 Responde en la memoria con el resultado de evaluar los siguientes casos:

1.- (combine-list-of-lsts ‘(() (+ -) (1 2 3 4))): El resultado obtenido es NIL.

2.- (combine-list-of-lsts ‘((a b c) () (1 2 3 4))): El resultado obtenido es NIL.

3.- (combine-list-of-lsts ‘((a b c) (1 2 3 4) ())): El resultado obtenido es NIL.

4.- (combine-list-of-lsts ‘((1 2 3 4))): El resultado obtenido es ((1) (2) (3) (4)).

5.- (combine-list-of-lsts ‘(nil)): El resultado obtenido es NIL.

6.- (combine-list-of-lsts nil): El resultado obtenido es NIL.

4. Arboles de verdad en logica proposicional

Pregunta 1: si en lugar de (∧ A (∨ B C)) tuviésemos (∧ A (¬ A) (∨ B C)), ¿qué sucedería?

El resultado que obtenemos es NIL, ya que nos encontramos ante una contradiccion, A ∧ !A.

Pregunta 2:¿Y en el caso de (∧ A (∨ B C) (¬ A))?

El resultado que obtenemos es el mismo, NIL, ya que al ser una puerta logica AND es necesario que se cumplan todos los terminos que se ven afectados por la misma para que sea SAT.

Pregunta 3: estudia la salida del trace mostrada más arriba. ¿Qué devuelve la función expand-truthtree?

La funcion expand-truth-tree devuelve los literales que se van obteniendo de la base del conocimiento para luego proceder a su evaluacion en truth-tree.

5. Busqueda en anchura

5.2 Escribe el pseudocódigo correspondiente al algoritmo BFS.

BFS:

CREAMOS COLA Q

ANIADIMOS EL ORIGEN A LA COLA Q

MARCAMOS ORIGEN COMO VISITADO

MIENTRAS Q NO ESTE VACIO:

V = OBTENEMOS ELEMENTO DE LA COLA

BUCLE EN EL QUE RECORREMOS LOS VECINOS DE V -> W:

SI W NO HA SIDO VISITADO:

MARCAMOS W COMO VISITADO

INSERTAMOS W DENTRO DE LA COLA Q

5.6 Ilustra el funcionamiento del código especificando la secuencia de llamadas a las que da lugar la evaluación:

Trace: (SHORTEST-PATH 'A 'F '((A D) (B D F) (C E) (D F) (E B F) (F)))

Trace: (BFS 'F '((A)) '((A D) (B D F) (C E) (D F) (E B F) (F)))

Trace: (BFS 'F '((D A)) '((A D) (B D F) (C E) (D F) (E B F) (F)))

Trace: (BFS 'F '((F D A)) '((A D) (B D F) (C E) (D F) (E B F) (F)))

Trace: BFS ==> (A D F)

Trace: BFS ==> (A D F)

Trace: BFS ==> (A D F)

Trace: SHORTEST-PATH ==> (A D F)

5.7 Utiliza el código anterior para encontrar el camino más corto entre los nodos B y G en el siguiente grafo no dirigido. ¿Cuál es la llamada concreta que tienes que hacer? ¿Qué resultado obtienes con dicha llamada?

La llamada correcta para encontrar el nodo G desde B es:

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

El resultado obtenido es (B E G)