PRACTICA 1: LISP

LEYRE CANALES ANTÓN

GLORIA DEL VALLE CANO

GRUPO 2311

ÍNDICE

1. DISTANCIA COSENO.

1.1. Implementa una función que calcule la distancia coseno entre dos vectores, x e y, de dos formas:

Recursiva:

Función auxiliar encargada de calcular el sumatoria cuadrado de los vectores.

```
;;Sumatorio de vectores (cuadrado)
(defun vct-sum(x)
    (if (null x)
        0
        (+(*(car x)(car x))(vct-sum(cdr x)))))
```

Función auxiliar encargada de calcular el producto cruzado de dos vectores.

```
;;Producto cruzado
(defun cross-prod(x y)
    (cond
        ((null x) 0)
        ((null y) 0)
        (T (+ (* (car x) (car y))
            (cross-prod (cdr x) (cdr y))))))
```

Función principal encargada de calcular la distancia del coseno, donde utilizamos las funciones auxiliares anteriores.

```
(defun cosine-distance-rec (x y)
    (if (or (= (vct-sum x) 0)
            (= (vct-sum y) 0))
        NIL
        (- 1 (/ (cross-prod x y)
            (* (sqrt (vct-sum x))
                (sqrt (vct-sum y)))))))
```

Usando mapcar:

Función auxiliar encargada de calcular el sumatoria cuadrado de los vectores.

```
;;Sumatorio de vectores (cuadrado)
(defun vct-sum-mapcar(x)
    (apply #'+(mapcar #'(lambda (z) (* z z)) x)))
```

Función auxiliar encargada de calcular el producto cruzado de dos vectores.

```
;;Producto cruzado
(defun cross-prod-mapcar (x y)
    (cond
        ((null x) 0)
        ((null y) 0)
        (T (apply #'+ (mapcar #'* x y)))))
```

Función principal encargada de calcular la distancia del coseno, donde utilizamos las funciones auxiliares anteriores.

```
;;Producto cruzado
(defun cross-prod-mapcar (x y)
    (cond
        ((null x) 0)
        ((null y) 0)
        (T (apply #'+ (mapcar #'* x y)))))
```

La evaluación de los casos de prueba es la siguiente:

(cosine-distance '(1 2) '(1 2 3))

```
CG-USER(9): (cosine-distance-rec '(1 2) '(1 2 3))
0.40238577
CG-USER(10): (cosine-distance-mapcar '(1 2) '(1 2 3))
0.40238577
```

(cosine-distance nil '(1 2 3))

```
CG-USER(11): (cosine-distance-rec nil '(1 2 3))
NIL
CG-USER(12): (cosine-distance-mapcar nil '(1 2 3))
NIL
```

(cosine-distance '() '())

```
CG-USER(13): (cosine-distance-rec '() '())
NIL
CG-USER(14): (cosine-distance-mapcar '() '())
NIL
```

(cosine-distance '(0 0) '(0 0))

```
CG-USER(15): (cosine-distance-rec '(0 0) '(0 0))
NIL
CG-USER(16): (cosine-distance-mapcar '(0 0) '(0 0))
NIL
```

1.2. Codifica una función que reciba un vector que representa a una categoría, un conjunto de vectores y un nivel de confianza que esté entre 0 y 1. La función deberá retornar el conjunto de vectores ordenado según más se parezcan a la categoría dada si su semejanza es superior al nivel de confianza.

Función auxiliar recursiva encargada de ir creando los vectores nuevos en los que introducimos el nivel de confianza de ese vector y llamar a la función de ordenar pasándole una lista de vectores creados.

```
(defun cosine-sim-category (vector lista nivel)
    (if (null lista)
        '()
        (let* ((semejanza (- 1 (cosine-distance-rec vector (first lista))))
               (lista-recursiva (cosine-sim-category vector (rest lista) nivel)))
            (if (> semejanza nivel)
                (ins-ord-primero (cons semejanza (first lista)) lista-recursiva)
                lista-recursiva))))
```

Función auxiliar que hemos creado en sustitución del "sort", nos ayuda a ordenar la lista por la clave primera de cada vector.

```
(defun ins-ord-primero (elem lista)
    (if (null lista)
        (list elem)
        (if (> (first elem) (first(first lista)))
            (cons elem lista)
            (cons (first lista) (ins-ord-primero elem (rest lista))))))
```

Función principal en la que realizamos el control de errores y llamamos a una de las funciones auxiliares anteriores haciendo un "rest" de lo devuelto para eliminar de cada vector de la lista que nos devuelve el primer elemento que corresponde al nivel de confianza y dejar los vectores como al principio.

```
(defun order-vectors-cosine-distance (vector lst-of-vectors &optional (confidence-level 0))
  (if (or (<= (length lst-of-vectors) 1)
          (< (length vector) 1)
          (> confidence-level 1)
          (< confidence-level 0))
      NIL

          (mapcar #'rest (cosine-sim-category vector lst-of-vectors confidence-level))))
```

La evaluación de los casos de prueba es la siguiente:

En los dos casos sale NIL debido a que en nuestra comprobación de errores no permitimos listas vacías.

```
CG-USER(20): (order-vectors-cosine-distance '(1 2 3) '())
NIL
CG-USER(21): (order-vectors-cosine-distance '() '((4 3 2) (1 2 3)))
NIL
```

En estos tres casos nos sale la salida que esperamos según el enunciado proporcionado.

```
CG-USER(22): (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2)) 0.5)
((4 2 2) (32 454 123))
CG-USER(23): (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2)) 0.3)
((4 2 2) (32 454 123) (133 12 1))
CG-USER(24): (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2)) 0.99)
NIL
```

1.3. Clasificador por distancia coseno. Codifica una función que reciba un conjunto de categorías, cuyo primer elemento es su identificador, un conjunto de vectores que representan textos, cuyo primer elemento es su identificador, y devuelva una lista que contenga pares definidos por un identificador de la categoría que minimiza la distancia coseno con respecto a ese texto y el resultado de la distancia coseno. El tercer argumento es la función usada para evaluar la distancia coseno (mapcar o recursiva).

Función auxiliar que hemos creado para calcular la distancio, es decir aplicar la función a los vectores y llama a la función de ordenar.

```
(defun clasificar-mejor (categoria texto distancia)
    (if (or (null categoria) (null texto))
        '()
        (let* ((calculo-distancia (mapcar distancia (list(rest(first categoria)))
                                          (list(rest(first texto))))))
            (ins-ord-segundo (cons (first(first categoria)) calculo-distancia)
                             (clasificar-mejor (rest categoria) texto distancia)))))
```

Función que hemos creado en sustitución al “sort”, nos ayuda a ordenar la lista por la clave segunda de cada vector.

```
(defun ins-ord-segundo (elem lista)
    (if (null lista)
        (list elem)
        (if (< (second elem) (second (first lista)))
            (cons elem lista)
            (cons (first lista) (ins-ord-segundo elem (rest lista))))))
```

Función principal en la que añadimos un control donde categorías y textos no pueden ser vacíos, y de las listas que obtenemos llamando a una de las funciones auxiliares las unimos.

```
(defun get-vectors-category (categories texts distance-measure)
    (if (or (null categories) (null texts))
        '()
        (let* ((lista-segunda (clasificar-mejor categories (rest texts) distance-measure))
               (lista-primera (clasificar-mejor categories texts distance-measure)))
            (if (null lista-segunda)
                (list (first lista-primera))
                (append (list (first lista-primera))
                        (list (first lista-segunda)))))))
```

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.

Declaración de las categorías y los textos que se nos proporcionan en el enunciado.

```
CG-USER(28): (setf categories '((1 43 23 12) (2 33 54 24)))
((1 43 23 12) (2 33 54 24))
CG-USER(29): (setf texts '((1 3 22 134) (2 43 26 58)))
((1 3 22 134) (2 43 26 58))
```

Tras ejecutar los dos casos propuestos con la macro time, podemos decir que utilizando la función de mapcar es mínimamente mas lenta que la recursiva.

```
CG-USER(35): (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:
;  3,033 cons cells, 67,848 other bytes, 0 static bytes
; Page Faults: major: 0 (gc: 0), minor: 0 (gc: 0)
((2 0.5101813) (1 0.18444914))
CG-USER(36): (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  0.001000 sec (  0.0%)
; space allocation:
;  2,241 cons cells, 82,056 other bytes, 0 static bytes
; Page Faults: major: 0 (gc: 0), minor: 0 (gc: 0)
((2 0.5101813) (1 0.18444914))
```

La evaluación de los casos de prueba es la siguiente:

Esta prueba da NIL debido a que tanto textos como categorías son dos listas vacías.

```
CG-USER(31): (get-vectors-category '(()) '(()) #'cosine-distance-mapcar)
((NIL NIL))
```

```
CG-USER(32): (get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-rec)
((2 0.40238577))
```

Esta prueba devuelve NIL debido a que categorías es una lista de vectores vacía.

```
CG-USER(33): (get-vectors-category '(()) '((1 1 2 3) (2 4 5 6)) #'cosine-distance-rec)
((NIL NIL) (NIL NIL))
```

2. RAÍCES DE UNA FUNCIÓN.

2.1. Implementar una función newton que aplique el método de Newton-Raphson para encontrar una raíz de una función.

Función principal recursiva que hemos creado y se encarga de calcular la raíz de una función.

```
(defun newton (f df max-iter x0 &optional (tol 0.001))
    (let* ((ff (funcall f x0))
                (fprime (funcall df x0))
                (step (- x0 (/ ff fprime)))
                (dif (abs (- x0 step))))
                (cond
                    ((< dif tol) x0)
                    ((= max-iter 0) NIL)
                    (t (newton f df (- max-iter 1)
                            step
                            tol)))))
```

La evaluación de los casos de prueba es la siguiente:

Los resultados de los casos de prueba propuestos en el enunciado coinciden con los que se nos dan.

```
CG-USER(38): (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 3.0)
4.000084
CG-USER(39): (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 0.6)
0.99999946
CG-USER(40): (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 30 -2.5)
-3.0000203
CG-USER(41): (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 10 100.0)
NIL
```

2.2. Implementar una función one-root-newton que recibe una función, su derivada, una lista de semillas y una tolerancia (opcional). Si el método de Newton encuentra una raíz a partir de alguna de las semillas, la función devuelve el valor de la primera raíz encontrada. Si no converge para ninguna semilla, devuelve nil.

Función principal recursiva que hemos creado para encontrar la raíz a partir de la semilla para la que converge, para ello utilizamos la recursión para ir comprobando con las diferentes semillas que nos dan y utilizamos la función de newton anterior para calcular la raíz.

```
(defun one-root-newton (f df max-iter semillas &optional (tol 0.001))
    (if (null semillas)
            nil
    (let ((operacion (newton f df max-iter (first semillas) tol)))
    (if (null operacion)
        (one-root-newton f df max-iter (rest semillas) tol)
    operacion))))
```

La evaluación de los casos de prueba es la siguiente:

Los resultados de los casos de prueba propuestos en el enunciado coinciden con los que se nos dan.

```
CG-USER(43): (one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 -2.5))
0.99999946
CG-USER(44): (one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(3.0 -2.5))
4.000084
CG-USER(45): (one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(3.0 -2.5))
NIL
```

2.3. Implementar una función all-roots-newton que recibe una función, su derivada, una lista de semillas y la tolerancia (opcional). La función devuelve una lista con las raíces encontradas o nil, a partir de las semillas.

Función principal recursiva que hemos creado para encontrar la lista de raíces para las que converge a través de las semillas que nos dan. La diferencia con la función anterior es que esta devuelve todas las raíces y no solo la primera encontrada por lo que hemos hecho un “append” para unir todas en una lista.

```
(defun all-roots-newton (f df max-iter semillas &optional ( tol 0.001))
  (if (null semillas)
    nil
    (append (list (newton f df max-iter (first semillas) tol))
            (all-roots-newton f df max-iter (rest semillas) tol))))
```

La evaluación de los casos de prueba es la siguiente:

Los resultados de los casos de prueba propuestos en el enunciado coinciden con los que se nos dan.

```
CG-USER(47): (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))
(0.99999946 4.000084 -3.0000203)
CG-USER(48): (all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 10000.0))
(0.99999946 4.000084 NIL)
CG-USER(49): (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)
```

Implementa una función haciendo uso de mapcan que pase la salida de all-roots-newton a una lista de semillas (sin nil).

```
(defun list-not-nil-roots-newton (f df max-iter semillas &optional (tol 0.001))
    (mapcan #'(lambda (x)
                    (if x (list x)))
                    (all-roots-newton
                        f df max-iter semillas tol)))
```

La evaluación de los casos de prueba es la siguiente:

Los resultados de los casos de prueba propuestos en el enunciado coinciden con los que se nos dan.

```
CG-USER(51): (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 10000.0))
(0.99999946 4.000084)
```

3. COMBINACIÓN DE LISTAS.

3.1. Define una función que combine un elemento dado con todos los elementos de una lista.

Función principal en la que utilizamos mapcar para aplicar un elemento dado a los diferentes elementos de una lista.

```
(defun combine-elt-lst (elt lst)
    (if (or (equal elt nil) (equal lst nil))
        nil
        (mapcar #'(lambda (x) (list elt x)) lst))
  )
```

La evaluación de los casos de prueba es la siguiente:

El resultado de este caso de prueba coincide con el resultado que nos dan.

```
CG-USER(53): (combine-elt-lst 'a '(1 2 3))
((A 1) (A 2) (A 3))
```

En estos casos de prueba devuelve NIL debido a que en el primero la lista es NIL, en el segundo son NIL la lista y el elemento a combinar y en el tercero es NIL el elemento y en nuestra función comprobamos que cuando alguna sea NIL devolvemos NIL

```
CG-USER(54): (combine-elt-lst 'a nil)
NIL
CG-USER(55): (combine-elt-lst nil nil)
NIL
CG-USER(56): (combine-elt-lst nil '(a b))
NIL
```

3.2. Diseña una función que calcule el producto cartesiano de dos listas.

Función principal recursiva que hemos creado y para hacer el producto cartesiano utilizamos la función "append" y la función de combinar elemento con lista anterior, además de una recursividad de llamadas a la función de combinar listas.

```
(defun combine-lst-lst (lst1 lst2)
    (if (or (null lst1) (null lst2))
        nil
        (append (combine-elt-lst (first lst1) lst2) (combine-lst-lst (rest lst1) lst2)))
  )
```

La evaluación de los casos de prueba es la siguiente:

El resultado de este caso de prueba coincide con el resultado que nos dan.

```
CG-USER(58): (combine-lst-lst '(a b c) '(1 2))
((A 1) (A 2) (B 1) (B 2) (C 1) (C 2))
```

En estos casos de prueba devuelve NIL debido a que en el primero las dos listas son NIL, en el segundo la segunda lista es NIL y en el tercero la primera lista es NIL.

```
CG-USER(59): (combine-lst-lst nil nil)
NIL
CG-USER(60): (combine-lst-lst '(a b c) nil)
NIL
CG-USER(61): (combine-lst-lst nil '(a b c))
NIL
```

3.3. Diseña una función que calcule todas las posibles disposiciones de elementos pertenecientes a N listas de forma que en cada disposición aparezca únicamente un elemento de cada lista

Función principal recursiva, con caso base de lstolsts es NIL devolvemos NIL y si el "rest" de lstolsts es NIL devolvemos una lista con el primero de lstolsts.

```
(defun combine-list-of-lsts (lstolsts)
  (if (null (rest lstolsts))
    (mapcar #'list (first lstolsts))
    (if (null lstolsts)
        nil
        (mapcar #'(lambda (x) (apply #'cons x))
            (nconc (combine-lst-lst (first lstolsts) (combine-list-of-lsts (rest lstolsts))))))))
```

La evaluación de los casos de prueba es la siguiente:

El resultado de este caso de prueba coincide con el resultado que nos dan.

```
CG-USER(64): (combine-list-of-lsts '((a b c) (+ -) (1 2 3 4)))
((A + 1) (A + 2) (A + 3) (A + 4) (A - 1) (A - 2) (A - 3) (A - 4) (B + 1) (B + 2)
 ...)
```

Algunos de estos casos de prueba nos devuelven NIL debido a que introducimos listas vacías o NIL y para estos casos nosotros hacemos devolver NIL.

```
CG-USER(65): (combine-list-of-lsts '(() (+ -) (1 2 3 4)))
NIL
CG-USER(66): (combine-list-of-lsts '((a b c) () (1 2 3 4)))
NIL
CG-USER(67): (combine-list-of-lsts '((a b c) (1 2 3 4) ()))
NIL
CG-USER(68): (combine-list-of-lsts '((1 2 3 4)))
((1) (2) (3) (4))
CG-USER(69): (combine-list-of-lsts '(nil))
NIL
CG-USER(70): (combine-list-of-lsts nil)
NIL
```

4. ÁRBOLES DE VERDAD EN LÓGICA PROPOSICIONAL.

Para el planteamiento de este ejercicio hemos ideado la solución de la siguiente manera:

1. Recibimos una FBF, por tanto debe ser completamente FBF
2. Esta expresión será SAT si existe una rama que sea todo valor TRUE
3. Traducimos la expresión a una simplificada
4. Evaluamos las operaciones por separado
5. Juntamos los valores finales

4.1. Implementar funciones que apliquen las reglas de derivación necesarias para construir el árbol de verdad.

Funciones auxiliares para encontrar dos literales en una misma expresión para encontrar una contradicción acerca de ellos:

```
(defun evaluate (lst)
  (reduce #'(lambda (x y) (or x y))
          (mapcan #'(lambda (elt) (evaluate-aux elt lst)) lst)))

(defun evaluate-aux (elt lst)
  (mapcar #'(lambda (comp) (equal (negate elt) comp)) lst))
```

Expansión del árbol:

```
(defun expand (exp)
  (cond
    ((literal-p exp) (list exp))

    ((and-p (first exp))
     (expand (rest exp)))

    ((or-p (first exp))
     (mapcan #'expand (rest exp)))

    ((cond-connector-p (first exp))
     (expand (translate-cond (rest exp))))

    ((bicond-connector-p (first exp))
     (expand (translate-bicond (rest exp))))

    ((and (unary-connector-p (first exp))
                  (listp (cadr exp)))
                      (expand(list (negate (cadr exp)) (negate (rest (rest exp))))))

    ;;((and (unary-connector-p (first exp)) (listp (second exp)))
    ;;          (expand (negate (rest exp))))
    (t exp)))
```

Funciones auxiliares para la traducción de operadores:

```
(defun negate (elt)
  (if (negative-literal-p elt)
      (second elt)
      (list +not+ elt)))


(defun translate-bicond (lst)
  (list +and+
        (translate-cond lst)
        (list +or+
              (cons +not+ (rest lst))
              (first lst))))


(defun translate-cond (lst)
 (list +or+
       (list +not+ (first lst))
       (second lst)))
```

4.2. Implementar las funciones que construyan el árbol de verdad a partir de una base de conocimiento.

Encuentra las contradicciones del árbol a medida que se va expandiendo:

```
(defun truth-tree (exp)
  (unless (null exp)
    (not (evaluate (expand exp)))))
```

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

```
CL-USER 57 : 4 > (truth-tree '(∧ A (! A) (∨ B C)))
    2 TRUTH-TREE > ...
      >> EXP : (∧ A (! A) (∨ B C))
      3 EXPAND > ...
        >> EXP : (∧ A (! A) (∨ B C))
      3 EXPAND < ...
        << VALUE-0 : (∧ A (! A) (∨ B C))
    2 TRUTH-TREE < ...
      << VALUE-0 : NIL
NIL
```

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

```
CL-USER 58 : 4 > (truth-tree '(∧ A (∨ B C) (! A)))
    2 TRUTH-TREE > ...
      >> EXP : (∧ A (∨ B C) (! A))
      3 EXPAND > ...
        >> EXP : (∧ A (∨ B C) (! A))
      3 EXPAND < ...
        << VALUE-0 : (∧ A (∨ B C) (! A))
    2 TRUTH-TREE < ...
      << VALUE-0 : NIL
NIL
```

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

```
CL-USER 64 : 7 > (expand '(<=> (A B)))
((V (! (A B)) NIL) (V (!) (A B)))

CL-USER 65 : 7 > (expand '(=> A (^ B (! A))))
((! A) B (! A))
```

5. BUSQUEDA EN ANCHURA.

5.1. Ilustra el funcionamiento del algoritmo resolviendo a mano algunos ejemplos ilustrativos.

Caso típico con el grafo dirigido del ejemplo:

A:
A -> D -> F
B:
B -> D -> F
C:
C -> E -> B -> F -> D
D:
D -> F

E:
    E -> B -> F -> D
F:
    F

Caso típico con el siguiente grafo:
((A B C) (B D E) (C F G) (D) (E) (F) (G))

A:
    A -> B -> C -> D -> E -> F -> G
B:
    B -> D -> E
C:
    C -> F -> G
D:
    D
E:
    E
F:
    F
G:
    G

5.2. Escribe el pseudocódigo correspondiente al algoritmo BFS.

Suponemos que en la lista de nodos por explorar se encuentra ya el nodo origen:

1. Si lista de nodos por explorar está vacía

   Finalizar.

2. Si no

   Sacamos el primer nodo de la lista de nodos por explorar

3. Exploramos dicho nodo y lo metemos en la lista de nodos explorados
4. Añadimos los nodos descubiertos a la lista de nodos por explorar si no se encuentran en la lista de nodos explorados

   Repetimos los pasos hasta finalizar

5.3. Estudia con detalle la siguiente implementación del algoritmo BFS, tomada del libro "ANSI Common Lisp" de Paul Graham

(http://www.paulgraham.com/acl.html). Asegúrate de que comprendes su funcionamiento con algún grafo sencillo.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Breadth-first-search in graphs
;;;
(defun bfs (end queue net)
  (if (null queue) '()
    (let* ((path (first queue))
           (node (first path)))
      (if (eql node end)
          (reverse path)
        (bfs end
             (append (rest queue)
                     (new-paths path node net))
             net)))))
(defun new-paths (path node net)
  (mapcar #'(lambda(n)
              (cons n path))
          (rest (assoc node net))))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.4. Pon comentarios en el código anterior, de forma que se ilustre cómo se ha implementado el pseudocódigo propuesto en 4.2).

(if (null queue)) '() -----> corresponde a nuestro pseudocódigo al paso 1.

(let * ((path (first queue)) ....) ------> corresponde al paso 2.

(bfs end .....) --------> corresponde al paso 3.

(new-paths path node net)) ---------> corresponde al paso 4.

5.5. Explica por qué esta función resuelve el problema de encontrar el camino más corto entre dos nodos del grafo.

Porque el algoritmo de búsqueda en anchura es un algoritmo completo y óptimo, por lo que si existe un camino a la meta se encontrará y además en caso de que existan varios devolverá aquel que sea óptimo.

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

Primero explora el nodo a:
(bfs 'f '((a)) '((a d) (b d f) (c e) (d f) (e b f) (f)))
Después explora el nodo d:
(bfs 'f '((d)) '((a d) (b d f) (c e) (d f) (e b f) (f)))
Después explora el nodo f que es la meta y finaliza:
(bfs 'f '((f)) '((a d) (b d f) (c e) (d f) (e b f) (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 es (shortest-path 'f 'c ((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)))

No obtenemos ningún resultado debido a que entra en un bucle infinito.

5.8. El código anterior falla (entra en una recursión infinita) cuando hay ciclos en el grafo y el problema de búsqueda no tiene solución. Ilustra con un ejemplo este caso problemático y modifica el código para corregir este problema.

El problema del código es que no existe una lista de nodo explorados por lo que entra en bucle infinito, una solución sería añadir una comprobación de si el nodo ya se ha explorado no introducirlo en la lista de nodos a explorar.

```
(defun bfs-improved (end queue net)
  (if (null queue)
    nil
    (let((path (car queue)))
        (let ((node (car path)))
        (if (eql node end)
            (reverse path)
            (bfs-improved end
                (append (cdr queue)
                    (new-paths-improved path node net))
                net))))))

(defun new-paths-improved (path node net)
    (if (null (evaluar-repeticion path))
        nil
        (mapcar #'(lambda(x) (cons x path))
                        (cdr (assoc node net)))))

(defun evaluar-repeticion (list)
    (or (null list)
        (and (not (member (first list) (rest list)))
            (evaluar-repeticion (rest list)))))

(defun shortest-path-improved (end queue net)
  (bfs-improved end queue net))
```