

PRÁCTICA 1

INTELIGENCIA ARTIFICIAL

Jorge Santisteban Rivas
Javier Martínez Rubio

Ejercicio 1

Apartado 1.1

Formula-dest

Pseudocódigo

Entradas x: vector, representado como una lista
y: vector, representado como una lista
prod-esc: función a utilizar para calcular el producto escalar
Salida: Resultado de aplicar la fórmula

Código

```
;;; formula-dest (x y prod-esc)
;;; Aplica la fórmula a x e y
```

```
(defun formula-dst (x y prod-esc)
  (- 1 (/ (funcall prod-esc x y)
            (* (sqrt(funcall prod-esc x x))
               (sqrt(funcall prod-esc y y))))))
```

Comentarios

Esta función simplemente aplica la fórmula necesaria a dos parámetros. La utilizaremos para calcular la distancia coseno

Prod-esc-rec

Pseudocódigo

Entradas x: vector, representado como una lista
y: vector, representado como una lista
Salida: producto escalar entre x e y

Procesamiento

si $x = 0$ o $y = 0$
devuelve 0
en caso contrario,
suma el resultado de multiplicar las primeras coordenadas
de cada lista con el resultado de aplicar la misma función
al resto de cada lista

Código

```
;; product-escalar-rec (x y)
;; Calcula el producto escalar de forma recursiva
;; Se asume que los dos vectores de entrada tienen la misma longitud.
```

```
(defun prod-esc-rec (x y)
  (if (or (null x) (null y))
      0
      (+ (* (first x) (first y))
```

```
(prod-esc-rec (rest x) (rest y))))
```

Comentarios

Es una función que calcula el producto escalar de dos vectores de forma recursiva.

Cosine-distance-rec

Pseudocódigo

Entradas x: vector, representado como una lista

y: vector, representado como una lista

Salida: distancia coseno entre x e y

Procesamiento

si $x = 0$ o $y = 0$

devuelve 0

si la norma de x o la norma de y son 0,

devuelve 0

en caso contrario,

calcula la distancia coseno según su fórmula haciendo

uso de la función auxiliar que calcula el producto escalar
entre dos vectores.

Código

```
;;; cosine-distance-rec (x y)
;;; Calcula la distancia coseno de un vector de forma recursiva
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
```

```
(defun cosine-distance-rec (x y)
  (cond ((= 0 (* (prod-esc-rec x) (prod-esc-rec y y))) 0)
        (t (formula-dst x y #'prod-esc-rec))))
```

Comentarios

Esta función calcula la distancia coseno entre dos listas de manera recursiva. Esta recursividad la vemos en el uso de una función recursiva de cálculo del producto escalar, que hemos programado anteriormente. También hacemos uso de la función, formula-dest, comentada anteriormente y que aplica la fórmula.

Ejemplos

```
3 CL-USER> (cosine-distance-rec '(1 2) '(1 2 3))
4 []
5 0.40238577
6 []
7 CL-USER> (cosine-distance-rec nil '(1 2 3))
8 []
9 []
10 CL-USER> (cosine-distance-rec '() '())
11 []
12 []
13 CL-USER> (cosine-distance-rec '(0 0) '(0 0))
14 []
15 []
16 CL-USER> |
```

Prod-esc-mapcar

Pseudocódigo

Entradas x: vector, representado como una lista

y: vector, representado como una lista

Salida: producto escalar entre x e y

Procesamiento

si x = 0 o y = 0

devuelve 0

en caso contrario,

suma el resultado de multiplicar las primeras coordenadas de cada lista con el resultado de aplicar la misma función al resto de cada lista

Código

```
;;; producto-escalar-rec (x y)
;;; Calcula el producto escalar usando mapcar
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
```

```
(defun prod-esc-mapcar (x y)
  (if (or (null x) (null y))
    0
    (apply #'+ (mapcar #'* x y))))
```

Comentarios

Es una función que calcula el producto escalar de dos vectores sin recursividad, utilizando la función mapcar.

Cosine-distance-mapcar

Pseudocódigo

Entradas x: vector, representado como una lista

y: vector, representado como una lista

Salida: distancia coseno entre x e y

Procesamiento

si x = 0 o y = 0

devuelve 0

si la norma de x o la norma de y son 0,

devuelve 0

en caso contrario,

calcula la distancia coseno según su fórmula haciendo uso de la función auxiliar que calcula el producto escalar entre dos vectores.

Código

```
;;; cosine-distance-mapcar (x y)
;;; Calcula la distancia coseno de un vector usando mapcar
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
```

```
(defun cosine-distance-mapcar (x y)
  (cond ((= 0 (* (prod-esc-mapcar x x) (prod-esc-mapcar y y))) 0)
        (t (formula-dst x y #'prod-esc-mapcar))))
```

Comentarios

Esta función calcula la distancia coseno entre dos listas utilizando la función mapcar, ya que hace uso de una función que calcula el producto escalar usando mapcar.

Ejemplos

```
, Compiling (DEFUN COSINE-DISTANCE-MAPCAR ...) □
CL-USER> (cosine-distance-mapcar '(1 2) '(1 2 3))
□
□□0.40238577□□
□
CL-USER> (cosine-distance-mapcar nil '(1 2 3))
□□0□□
□
CL-USER> (cosine-distance-mapcar '() '())
□□0□□
□
CL-USER> (cosine-distance-mapcar '(0 0) '(0 0))
□□0□□
□
```

Apartado 1.2

Order-lst-vectors

Función auxiliar que hemos creado para que pasándole la lista de vectores ordenados inserte un nuevo vector correctamente

Pseudocódigo

Entrada: vector-ref: vector que representa a una categoría, representado como una lista
vector-insert: vector-insert: dupla con el vector a insertar y su distancia, ya calculada
ord-lst-of-vectors: nuevo vector de vectores ordenados

Salida: la nueva lista ordenada con el vector insertado

Procesamiento:

Si la lista de vectores ordenados es vacía

Insertamos el vector a insertar en la lista

Si la distancia coseno del vector es menor que la del primer elemento de la lista ordenada

Devolvemos la lista ordenada con el vector como primer elemento

En otro caso

Devolvemos la lista con el primer elemento de la lista ordenada y la llamada recursiva de order-lst-vectors aplicada al vector y al *rest* de la lista ordenada.

Código

```
;;;;;;
;;;;; order-lst-vectors
```

```
(defun order-lst-vectors(vector-ref vector-insert ord-lst-of-vectors)
  (cond ((null ord-lst-of-vectors) (cons (second vector-insert) ord-lst-of-vectors))
        ((< (first vector-insert)
              (cosine-distance-mapcar vector-ref (first ord-lst-of-vectors)))
         (cons (second vector-insert) ord-lst-of-vectors))
        (t (cons (first ord-lst-of-vectors)
                  (order-lst-vectors vector-ref vector-insert (rest ord-lst-of-vectors)))))))
```

Comentarios

En este caso hemos utilizado recursividad para ir insertando elementos de manera ordenada a una lista. La mejora que hemos realizado ha sido que el argumento vector-insert lleva ahora su distancia coseno ya asociada, de esta forma no es necesario calcularla siempre.

Order-vectors-cosine-distance

Pseudocódigo

Entrada: vector: vector que representa a una categoría, representado como una lista

lst-of-vectors vector de vectores

confidence-level: Nivel de confianza (parametro opcional)

Salida: Vectores cuya semejanza con respecto a la categoría es superior al nivel de confianza

Procesamiento:

Si $1 - \text{nivel de confianza} \geq \text{distancia del primero de la lista}$

Si $\text{rest} == \text{null}$

Insertamos el primer elemento en una lista vacia (con order-lst-vectors)

En otro caso

Recursión de order-lst-vectors para el resto de elementos

En otro caso

Si $\text{rest} == \text{null}$

Nil

En otro caso

Recursión de la función para aplicarla al resto de los vectores que queremos comparar con el vector referencia

Código

```
;;;;;;;
;;; order-vectors-cosine-distance
;;; Devuelve aquellos vectores similares a una categoría
```

```
(defun order-vectors-cosine-distance (vector lst-of-vectors &optional (confidence-level 0))
```

```
  (if (>= (- 1 confidence-level)
```

```
      (cosine-distance-mapcar vector (first lst-of-vectors)))
```

```
    (if (null (rest lst-of-vectors))
```

```
        (order-lst-vectors vector (list (cosine-distance-mapcar vector (first lst-of-vectors)) (first lst-of-vectors)) '())
        (order-lst-vectors vector (list (cosine-distance-mapcar vector (first lst-of-vectors)) (first lst-of-vectors))
```

```
        (order-vectors-cosine-distance vector (rest lst-of-vectors) confidence-level)))
```

```
(if (null (rest lst-of-vectors))
  nil
  (order-vectors-cosine-distance vector (rest lst-of-vectors) confidence-level))))
```

Comentarios

Hemos utilizado la función auxiliar para ir insertando (si cumple la condición) uno a uno en la lista y de manera ordenada, aplicando recursividad sobre la propia función.

Ejemplos

```
, compiling: ORDER-VECTORS-COSINE-DISTANCE ...]
CL-USER> (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2)) 0.5)
NIL
CL-USER> (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))
CL-USER> (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2)) 0.99)
NIL
CL-USER> (order-vectors-cosine-distance '(1 2 3) '())
NIL
CL-USER> (order-vectors-cosine-distance '() '((4 3 2) (1 2 3)))
((1 2 3) (4 3 2))
```

Apartado 1.3

Get-text-category

Función auxiliar que a partir de un text y una lista de categorías devuelve el id de la categoría que mejor lo aproxima y la distancia a ella.

Pseudocódigo

Entrada: categories: vector de vectores, representado como una lista de listas

text: vector, representado como una lista

distance-measure: función de distancia

min-category: la categoría mínima para comenzar la iteración (la primera categoría por defecto), con su distancia asociada

Salida: Par formado por el vector que identifica la categoría de menor distancia , junto con el valor de dicha distancia.

Procesamiento:

Si lista_categorías == null

devolver una lista con el primer elemento de min-category y la distancia del texto con esa categoría

En otro caso

Si la distancia entre la primera categoría y el texto es menor que la distancia entre la min-category y el texto

Esta pasa a ser la min-category y hacemos una llamada recursiva

En otro caso

Llamada recursiva manteniendo min-category

Código

```
;;;; get-text-category (categories text distance-measure min-category)
;
```

```
;; A partir de un texto devolvemos el identificador de la categoria que lo aproxima y su distancia.
((defun get-text-category (categories text distance-measure min-category)
  (if (null categories)
      min-category
      (if (< (funcall distance-measure (rest (first categories)) (rest text))
              (second min-category))
          (get-text-category (rest categories) text distance-measure (list (first (first categories))
                                         (funcall distance-measure (rest (first categories)) (rest
text))))
          (get-text-category (rest categories) text distance-measure min-category))))
```

Comentarios

En esta función utilizamos recursividad para ir manteniendo siempre la menor categoría, siendo el primer valor de esta el primer elemento de la lista de categorías. Tras haber pasado por todas las categorías devolvemos simplemente el identificador y la distancia a esta. La mejora que hemos realizado ha sido que el argumento min-category lleva ya asociada sus distancia, así no hay que calcularla en cada iteración

Get-vectors category

Pseudocódigo

Entrada: categories: vector de vectores, representado como una lista de listas
 texts: vector de vectores, representado como una lista de listas

Salida: Pares formados por el vector que identifica la categoría de menor distancia , junto con el valor de dicha distancia

Procesamiento:

Con mapcar y una función lambda ir aplicando a todos los elementos de texts la función get-text-category, devolviendo una lista final con todos los ids y las distancias.

Código

```
;;;;;
;; get-vectors-category (categories vectors distance-measure)
( defun get-vectors-category (categories texts distance-measure)
  (if (or (null categories) (null texts))
      NIL
      (mapcar #'(lambda(x) (get-text-category categories x distance-measure (list (first (first categories))
                                         (funcall distance-measure (rest (first categories)) (rest x)))))) texts)))
```

Comentarios

Una vez ya programada la función que averigua la mejor categoría, al final en esta simplemente recurriendo a la función mapcar y una lambda conseguimos una lista con todos los pares de id y distancia.

Apartado 1.4

Utilizamos la función para los siguientes casos, midiendo el tiempo y usando tanto la implementada recursivamente como la que usa mapcar:

1. (get-vectors-categories '() '() #'cosine-distance)

```

110 CL-USER> (time (get-vectors-category '() '() #'cosine-distance-rec))
111 □Evaluation took:
112   0.000 seconds of real time
113   0.000010 seconds of total run time (0.000008 user, 0.000002 system)
114   100.00% CPU
115   8,468 processor cycles
116   0 bytes consed
117
118 □□□((NIL 0))□□□
119 □
120 CL-USER> (time (get-vectors-category '() '() #'cosine-distance-mapcar))
121 □Evaluation took:
122   0.000 seconds of real time
123   0.000009 seconds of total run time (0.000009 user, 0.000000 system)
124   100.00% CPU
125   8,300 processor cycles
126   0 bytes consed
127
128 □□□((NIL 0))□□□

```

Vemos que comparando tanto categorías como textos vacíos ambas funciones se comportan igual, devolviendo NIL como id y 0 como distancia.

2. (get-vectors-categories '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance)

```

260 CL-USER> (time (get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-rec))
261 □
262 Evaluation took:
263   0.000 seconds of real time
264   0.000024 seconds of total run time (0.000021 user, 0.000003 system)
265   100.00% CPU
266   49,604 processor cycles
267   0 bytes consed
268
269 □□□((2 0.40238577))□□□
270 □
271 CL-USER> (time (get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-mapcar))
272 □Evaluation took:
273   0.000 seconds of real time
274   0.000027 seconds of total run time (0.000027 user, 0.000000 system)
275   100.00% CPU
276   54,492 processor cycles
277   0 bytes consed
278
279 □□□((2 0.40238577))□□□
280 □

```

Vemos que para ambos da el mismo resultado pero la recursiva tarda ligeramente menos tiempo que la que utiliza mapcar.

3. (get-vectors-categories '() '((1 1 2 3) (2 4 5 6)) #'cosine-distance)

```

281 CL-USER> (time (get-vectors-category '() '((1 1 2 3) (2 4 5 6)) #'cosine-distance-rec))
282 □Evaluation took:
283   0.000 seconds of real time
284   0.000009 seconds of total run time (0.000009 user, 0.000000 system)
285   100.00% CPU
286   10,524 processor cycles
287   0 bytes consed
288
289 □□□((NIL 0) (NIL 0))□□□
290 □
291 CL-USER> (time (get-vectors-category '() '((1 1 2 3) (2 4 5 6)) #'cosine-distance-mapcar))
292 □Evaluation took:
293   0.000 seconds of real time
294   0.000002 seconds of total run time (0.000002 user, 0.000000 system)
295   100.00% CPU
296   2,968 processor cycles
297   0 bytes consed
298
299 □□□((NIL 0) (NIL 0))□□□
300 □

```

Para esos dos textos vemos que al no haber categorías, los resultados son NIL y 0 para cada uno de los pares. Vemos que en este caso la recursiva tarda mucho mas tiempo que la que usa mapcar.

Ejercicio 2

Apartado 2.1

Newton

Pseudocódigo

Entradas: f: función cuyo cero se desea encontrar
df: derivada de f
max-iter: máximo número de iteraciones
x0: estimación inicial del cero (semilla)
tol: tolerancia para convergencia (parámetro opcional)

Salida: estimación del cero de f o NIL si no converge

Procesamiento

```
si max-iter = -1
devuelve nil
en caso contrario,
se vuelve a llamar a la misma función con max-iter = max-iter -1
y con x0 como la nueva semilla calculada según la fórmula.
```

Código

```
;; newton
;; Estima el cero de una función mediante Newton-Raphson

(defun newton (f df max-iter x0 &optional (tol 0.001))
  (if (= max-iter 0) NIL
    (if (< (abs (funcall f x0)) tol) x0
      (newton f df (- max-iter 1) (- x0 (/ (funcall f x0) (funcall df x0))) tol))))
```

Comentarios

Esta función estima (con una cierta tolerancia), a partir de una semilla dada, la raíz de una función pasada como parámetro utilizando el método de Newton. Es un método iterativo en el cual la semilla va cambiando en cada iteración de acuerdo a una cierta fórmula. En cada una de estas iteraciones, se evalúa la función en la semilla, y si este es 0, se devuelve la semilla como la raíz. Si no lo es, se evalúa la función en la nueva raíz. Además, hay un número máximo de iteraciones, y, en caso de no encontrar ninguna raíz en este periodo, se devuelve NIL.

La función que hemos programado es, por tanto, recursiva. Primero comprobamos si ya se han superado las iteraciones máximas. Ya que hacemos esta comprobación la primera, aparece en el código como max-iter = -1, y no como max-iter = 0. Despues miramos si $f(x_0) = 0$, y en caso de serlo, devolvemos x_0 . En otro caso, hacemos la recursividad, llamando a newton con los nuevos valores en sus argumentos.

Ejemplos

```

;;: compiling (DEFUN NEWTON ...)

CL-USER> (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 3.0)
NIL
CL-USER> (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 0.6)
0.99999946
CL-USER> (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 -2.5)
-3.0000203
CL-USER> (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 10 100)
NIL

```

Apartado 2.2

One-root-newton

Pseudocódigo

Entradas: f: funcion cuyo cero se desea encontrar
df: derivada de f
max-iter: maximo numero de iteraciones
semillas: semillas con las que invocar a Newton
tol: tolerancia para convergencia (parametro opcional)
Salida: el primer cero de f que se encuentre , o NIL si se diverge para todas las semillas

Procesamiento

```

si semillas = nil
devuelve nil
si one-root-newton != nil
devuelve newton f df max-iter (first semillas) tol
en caso contrario,
se hace one-root-newton f df max-iter (rest semillas) tol

```

Código

```

;;; one-root-newton
;;; Prueba con distintas semillas iniciales hasta que Newton
;;; converge

```

```

(defun one-root-newton (f df max-iter semillas &optional (tol 0.001))
  (cond ((null semillas) nil)
        ((newton f df max-iter (first semillas) tol)
         (newton f df max-iter (first semillas) tol))
        (t (one-root-newton f df max-iter (rest semillas) tol))))

```

Comentarios

Esta función es también recursiva, ya que tiene que evaluar la función newton en cada una de las semillas de la lista pasada como argumento hasta que se encuentra una que no sea nil. Simplemente se comprueba esta condición, y si no se cumple se vuelve a llamar a la misma función con el resto de la lista de semillas.

Ejemplos

```

2 ; Compiling (DEFUN ONE-ROOT-NEWTON ...)
3 CL-USER> (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))
4 []
5 ()|0.9999946|()
6 []
7 CL-USER> (one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(3.0 -2.5))
8 ()|4.0|()
9 []
10 CL-USER> (one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(3.0 -2.5))
11 ()|NIL|()
12 []
13 CL-USER>

```

Apartado 2.3

ALL-roots-newton

Pseudocódigo

Entradas: f: funcion cuyo cero se desea encontrar
df: derivada de f
max-iter: maximo numero de iteraciones
semillas: semillas con las que invocar a Newton
tol: tolerancia para convergencia (parametro opcional)
Salida: las raices que se encuentren para cada semilla o nil
si para esa semilla el metodo no converge

Procesamiento

aplicamos newton pasando como semilla cada elemento de la lista
pasada como argumento

Código

```

;; all-roots-newton
;; Prueba con distintas semillas iniciales y devuelve las raices
;; encontradas por Newton para dichas semillas

```

```
(defun all-roots-newton (f df max-iter semillas &optional ( tol 0.001))
  (mapcar #'(lambda(x) (newton f df max-iter x tol)) semillas))
```

Comentarios

En esta función aplicamos un mapcar para así simplificar. También tuvimos en cuenta implementarlo de manera recursiva, pero decidimos que esta manera era mucho más simple.

Ejemplos

```

2 ; Compiling (DEFUN ALL-ROOTS-NEWTON ...)
3 CL-USER> (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))
4 []
5 ()|0.9999946 4.0 -3.0000203|()
6 []
7 CL-USER> (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))
8 ()|0.9999946 4.0 NIL|()
9 []
10 CL-USER> (all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(0.6 3.0 10000.0))
11 ()|NIL NIL NIL|()
12 []
13 CL-USER>

```

Apartado 2.3.1

list-not-nil-roots-newton

Pseudocódigo

Entradas: f: funcion cuyo cero se desea encontrar
df: derivada de f
max-iter: maximo numero de iteraciones
semillas: semillas con las que invocar a Newton
tol: tolerancia para convergencia (parametro opcional)
Salida: las raices que se encuentren para cada semilla

Procesamiento

aplicamos all roots newton pero solo seleccionamos las que el resultado no sea nil

Código

```
;;;;; list-not-nil-newton
```

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

Comentarios

En esta función aplicamos un mapcar para así simplificar. Además gracias al unless solo seleccionamos los resultados que no sean nil. También tuvimos en cuenta implementarlo de manera recursiva, pero decidimos que esta manera era mucho más simple.

Ejemplos

```
CL-USER> (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))
NIL
CL-USER> (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))
NIL
CL-USER> |
```

Ejercicio 3

Apartado 3.1

Combine-elt-lst

Pseudocódigo

Entrada: elt: elemento a combinar con la lista
lst: lista a combinar con el elemento

Salida: Devuelve las listas resultantes de combinar el elemento con cada uno de los elementos de la lista.

Procesamiento:

Si el elemento o la lista es null

Nil

En otro caso, usando mapcar combinamos uno a uno cada uno de los elementos de la lista con el elemento pasado como argumento.

Código

```
;;;;; combine-elt-lst
;;;;; Combina un elemento dado con todos los elementos de una lista

(defun combine-elt-lst (elt lst)
  (cond ((or (null elt) (null lst))
          nil)
        (t (mapcar #'(lambda(x) (list elt x)) lst))))
```

Comentarios

En lugar de utilizar la recursión, a partir de la función mapcar y una función lambda que crea una lista a partir de dos elementos vamos recorriendo la lista elemento a elemento y generando todas las listas demandadas.

Ejemplos

```
1 ;;; Computing (DEFUN COMBINE-LIST-OF-ELTS)
2 CL-USER> (combine-elt-lst 'a nil)
3 []
4 []
5 []
6 CL-USER> (combine-elt-lst nil nil)
7 []
8 []
9 []
10 CL-USER> (combine-elt-lst nil '(a b))
11 []
12 []
13 CL-USER> (combine-elt-lst 'a '(1 2 3))
14 []
15 []
16 CL-USER> |
```

Apartado 3.2

Combine-lst-lst

Pseudocódigo

Entrada: lst1: primera lista

lst2: segunda lista

Salida: lista con las combinaciones del elemento con cada uno de los de la lista

Procesamiento:

Si alguna de las listas es null

Nil

En otro caso, utilizando mapcan y una función lambda que va elemento a elemento de la primera lista aplicando combine-elt-lst sobre la segunda lista

Código

```
;;;;; combine-lst-lst
;;;;; Calcula el producto cartesiano de dos listas
```

```
(defun combine-lst-lst (lst1 lst2)
  (cond ((or (null lst1) (null lst2))
         nil)
        (t (mapcan #'(lambda(x) (combine-elt-lst x lst2)) lst1))))
```

Comentarios

De igual manera que en el 3.1 aplicamos mapcan y vamos creando todos los pares de combinaciones de las dos listas.

Ejemplos

```
CL-USER> (combine-lst-lst '(a b c) '(1 2))
((A 1) (A 2) (B 1) (B 2) (C 1) (C 2))
CL-USER> (combine-lst-lst '(a b c) nil)
NIL
CL-USER> (combine-lst-lst nil nil)
NIL
CL-USER> (combine-lst-lst nil '(a b c))
NIL
```

Apartado 3.3

Combine-list-of-lsts

(utilizamos dos funciones auxiliares iguales que las explicadas en el 3.1 y 3.2 pero para combinar un elemento con una lista utilizamos cons).

Pseudocódigo

Entrada: lst1: primera lista

lst2: segunda lista

Salida: producto cartesiano de las dos listas

Procesamiento: Si alguna de las listas devolvemos nil. En otro caso hacemos recursión, donde, mediante la función combine-cons-elt-lst, vamos combinando elemento a elemento de la primera lista con los de la segunda para al final guardar cada uno de los resultados en una lista final que será el resultado que devolveremos.

Código

```
;;;;; combine-append-lst-lst
;;;;; Calcula el producto cartesiano de dos listas (utilizando append)
```

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

Pseudocódigo

Entrada: lstolsts: lista de listas

Salida: lista con todas las posibles combinaciones de elementos

Procesamiento:

Si la lista de listas es null

Nil

En otro caso, llamamos a la función que combina dos listas, cuyos argumentos serán la primera lista y la recursión de la función pero respecto al *rest* de la lista de listas.

Código

```
;;;;; combine-list-of-lsts
;;;;; Calcula todas las posibles disposiciones de elementos
;;;;; pertenecientes a N listas de forma que en cada disposicion
;;;;; aparezca unicamente un elemento de cada lista
(defun combine-list-of-lsts (lstolsts)
  (cond ((null lstolsts)
         (list nil))
        (t (combine-cons-lst-lst (first lstolsts)
                                  (combine-list-of-lsts (rest lstolsts))))))
```

Comentarios

En este caso hemos utilizado la recursión para poder ir combinando todas las listas a partir de la función que da todas las combinaciones de dos listas.

Ejemplos

```
110 CL-USER> (combine-list-of-lsts '((() (+ -) (1 2 3 4)))
111  ()NIL()
112  )
113  []
114 CL-USER> (combine-list-of-lsts '((a b c) () (1 2 3 4)))
115  ()NIL()
116  []
117 CL-USER> (combine-list-of-lsts '((a b c) (1 2 3 4) ()))
118  ()NIL()
119  []
120 CL-USER> (combine-list-of-lsts '(((1 2 3 4)))
121  ()((1) (2) (3) (4))()
122  []
123 CL-USER> (combine-list-of-lsts '(nil))
124  ()NIL()
125  []
126 CL-USER> (combine-list-of-lsts nil)
127  ()(NIL)()
128  []
129 CL-USER>
```

Ejercicio 4

Nuestro planteamiento del problema es el siguiente: Primero, cogemos todas las expresiones de la base de conocimiento y las transformamos a expresiones con solamente disyunciones y conjunciones. Una vez tenemos toda la base transformada, hacemos el árbol de verdad y concluimos si la base es SAT o UNSAT.

Nota: En nuestra implementación, a la función truth-tree le pasamos como argumento una lista de listas, en la que cada una de estas es una fbf de la base de conocimiento. Esto quiere decir que, aunque la base esté compuesta de una sola fbf, habrá que pasarle a truth-tree la mencionada lista de listas, aunque esta esté solamente formada por una sublista, que será la fbf en cuestión.

Apartado 4.1

Para este apartado hemos utilizado funciones auxiliares que hacen las equivalencias de las expresiones particulares. A partir de estas funciones más simples, hemos creado una función que combina todas ellas para transformar una expresión compleja.

Double negation

Pseudocódigo

Entrada: fbf - Formula bien formada (FBF) a analizar

Salida: fbf-modified FBF sin doble negación

Procesamiento:

Devuelve el segundo elemento de la lista, es decir, el átomo

Código

```
;;;;;;  
;;; double-negation  
;;; Recibe una expresion con una doble negacion y aplica la regla de derivacion adecuada
```

```
(defun double-negation(fbf)  
  (second fbf))
```

Comentarios

Esta función es muy simple, simplemente te devuelve el átomo en positivo, ya que es una doble negación.

Bicond

Pseudocódigo

Entrada: fbf - Formula bien formada (FBF) a analizar

Salida: fbf-modified FBF sin doble condicional

Procesamiento:

devuelve una lista con $(V (\wedge A B) (\wedge !A !B))$

Código

```
;;;;;  
;;; bicond  
;;; Recibe una expresion con una doble condición y aplica la regla de derivacion adecuada
```

```
(defun bicond(fbf)  
  (list +or+ (list +and+ (second fbf) (third fbf))  
        (list +and+ (list +not+ (second fbf)) (list +not+ (third fbf)))))
```

Comentarios

Esta función simplemente aplica la transformación adecuada del bicondicional.

Conditional

Pseudocódigo

Entrada: fbf - Formula bien formada (FBF) a analizar

Salida: fbf-modified FBF sin condicional

Procesamiento:

devuelve una lista con $(V !A B)$

Código

```
;;;;;
```

;;; conditional
;;; Recibe una expresion con una condición y aplica la regla de derivacion adecuada

```
(defun conditional(fbf)
  (list +or+ (list +not+ (second fbf)) (third fbf)))
```

Comentarios

Esta función simplemente aplica la transformación adecuada del condicional.

Neg-bicond

Pseudocódigo

Entrada: fbf - Formula bien formada (FBF) a analizar

Salida: fbf-modified FBF sin doble condicional negado

Procesamiento:

devuelve ina lista con $(V (\wedge A !B) (\wedge !A B))$

Código

```
;;;;;;;
```

;;; neg-bicond

;;; Recibe una expresion con una doble condición negada y aplica la regla de derivacion adecuada

```
(defun neg-bicond(fbf)
  (list +or+ (list +and+ (second fbf) (list +not+ (third fbf)))
        (list +and+ (list +not+ (second fbf)) (third fbf))))
```

Comentarios

Esta función simplemente aplica la transformación adecuada del doble condicional negado.

Neg-cond

Pseudocódigo

Entrada: fbf - Formula bien formada (FBF) a analizar

Salida: fbf-modified FBF sin condicional negado

Procesamiento:

devuelve una lista con $(\wedge A !B)$

Código

```
;;;;;;;
```

;;; neg-cond

;;; Recibe una expresion con una condición negada y aplica la regla de derivacion adecuada

```
(defun neg-conditional(fbf)
  (list +and+ (second fbf) (list +not+ (third fbf))))
```

Comentarios

Esta función simplemente aplica la transformación adecuada del condicional negado.

De-morgan-and

Pseudocódigo

Entrada: fbf - Formula bien formada (FBF) a analizar

Salida: fbf-modified FBF sin la conjunción negada

Procesamiento:

si el resto de la lista es null,
 devuelve una lista de listas con un not y el primer elemento de fbf,
 si el primer elemento es un and o un or,
 hace la recursividad con el resto de la lista,
 en caso contrario,
 hace un append de un not con el primer elemento de la lista y el resultado
 de aplicar la misma función al resto de la lista

Código

```
;;;;; de-morgan-and
;;;; Recibe una expresion con una conjunción negada y aplica la regla de derivacion adecuada
```

```
(defun de-morgan-and (fbf)
  (cond ((null (rest fbf)) (list (list +not+ (first fbf))))
        ((n-ary-connector-p (first fbf)) (cons +or+ (de-morgan-or (rest fbf))))
        (t (append (list (list +not+ (first fbf))) (de-morgan-or (rest fbf))))))
```

Comentarios

Esta función aplica la transformación adecuada de la conjunción negada. El problema aquí es que la conjunción es una operación n-aria, es decir, tiene tantos argumentos como queramos. Es por ello por lo que es necesario hacer recursividad, para que vaya negando todos y cada uno de estos argumentos y se queden como una disyunción.

De-morgan-or

Pseudocódigo

Entrada: fbf - Formula bien formada (FBF) a analizar
Salida: fbf-modified FBF sin disyunción negada

Procesamiento:

si el resto de la lista es null,
 devuelve una lista de listas con un not y el primer elemento de fbf,
 si el primer elemento es un and o un or,
 hace la recursividad con el resto de la lista,
 en caso contrario,
 hace un append de un not con el primer elemento de la lista y el resultado
 de aplicar la misma función al resto de la lista

Código

```
;;;;; de-morgan-or
;;;; Recibe una expresion con una disyunción negada y aplica la regla de derivacion adecuada
```

```
(defun de-morgan-or (fbf)
  (cond ((null (rest fbf)) (list (list +not+ (first fbf))))
        ((n-ary-connector-p (first fbf)) (cons +and+ (de-morgan-or (rest fbf))))
        (t (append (list (list +not+ (first fbf))) (de-morgan-or (rest fbf))))))
```

Comentarios

Esta función aplica la transformación adecuada de la disyunción negada. El problema aquí es que la disyunción es una operación n-aria, es decir, tiene tantos argumentos como queramos. Es por ello

por lo que es necesario hacer recursividad, para que vaya negando todos y cada uno de estos argumentos y se queden como una conjunción.

Ya que todas estas funciones son auxiliares y se utilizan en una función más compleja, hemos visto innecesario incluir ejemplos de todas ellas. Como es obvio, las exemplificaremos en la siguiente función.

Convert

Pseudocódigo

Entrada: fbf - Formula bien formada (FBF) a analizar

Salida: fbf modificada

Procesamiento:

si fbf es un conector o un literal, lo devuelve
dependiendo del primer elemento de la lista, hace la recursividad con
la correspondiente transformación como argumento
en caso de ser un conector n-ario, realiza la recursividad de cada uno
de sus argumentos utilizando un mapcar

Código

```
;;;;; convert
;;;; Recibe una expresion y la convierte en una expresion con and's y or's
```

```
(defun convert (fbf)
  (cond ((or (literal-p fbf) (connector-p fbf)) fbf)
        ((bicond-connector-p (first fbf)) (convert (bicond fbf)))
        ((cond-connector-p (first fbf)) (convert (conditional fbf)))
        ((unary-connector-p (first fbf))
         (cond ((bicond-connector-p (first (first (rest fbf))))
                (convert (neg-bicond (second fbf))))
               ((cond-connector-p (first (first (rest fbf))))
                (convert (neg-conditional (second fbf))))
               ((eql +or+ (first (first (rest fbf))))
                (convert (de-morgan-or (second fbf))))
               ((eql +and+ (first (first (rest fbf))))
                (convert (de-morgan-and (second fbf))))
               ((unary-connector-p (first (first (rest fbf))))
                (convert (double-negation (second fbf))))
               (t fbf)))
         ((n-ary-connector-p (first fbf))
          (mapcar #'(lambda(x) (convert x)) fbf))
         (t fbf))
      )
```

Comentarios

Esta función transforma una expresión compleja, es decir, una expresión que es combinación de muchas expresiones simples, en una expresión que contenga solamente disyunciones y conjunciones, que es lo que buscamos. Para ello, utilizamos la recursión, volviendo a llamar a la función con las expresiones simples ya convertidas cada vez que encontramos una. De esta forma,

vamos transformando todas poco a poco, hasta que acabamos transformando todas. El caso base de la recursión es cuando a la función le llega un literal o un conector, en cuyo caso lo devuelve tal y como le llega. De esta forma, nos aseguramos que ninguna expresión queda sin ser transformada.

Ejemplos

Comenzamos con ejemplos sencillos:

```
55  □
56 CL-USER> (convert '(^ A B))
57 □□(^ A B)□□□
58 □
```

```
48  □
49 CL-USER> (convert '(> A B))
50 □□(V (! A) B)□□□
51 □
```

Vamos complicando un poco la expresión

```
45 CL-USER> (convert '(> A (^ B (! A))))
46 □
47 □□(V (! A) (^ B (! A)))□□□
48 □
```

```
58  □
59 CL-USER> (convert '(> (> (^ P Q) R) (^ P Q)))
60 □□(V (^ (^ P Q) (! R)) (^ P Q))□□□
61 □
62 CL-USER>
```

Finalmente, introducimos una expresión bastante compleja

```
51  □
52 CL-USER> (convert '<=> (> (^ P Q) R) (> P (v (! Q) R)))
53 □□(V (^ (V (V (! P) (! Q)) R) (V (! P) (V (! Q) R)))
54 (^ (^ (^ P Q) (! R)) (^ P (^ Q (! R)))))□□□
55 □
56 CL-USER> (convert 'L(^ A B))
```

Como podemos comprobar en estos ejemplos, tanto las funciones auxiliares como la más compleja funcionan correctamente, y realizan lo que deseamos, es decir, convertir cualquier expresión en su equivalente con solo disyunciones y conjunciones.

Expand

Pseudocódigo

Entrada: fbfs – Lista de formulas bien formadas (FBF) a analizar

Salida: fbfs modificadas

Procesamiento:

devuelve la transformación de cada una de las fbf's recibidas como argumento como una conjunción

Código

```
;;;; expand  
;; Recibe la base de conocimiento y la convierte en una expresión con and's y or's
```

```
(defun expand (fbfs)  
  (cons '+and+ (mapcar #'(lambda(x) (convert x)) fbfs)))
```

Comentarios

Esta función aplica la función convert, anteriormente comentada, a una lista de fbf's, que en nuestro caso será la base de conocimiento que queremos analizar, con un signo “and” al principio de todas. Este signo lo ponemos porque esto es lo que pasaremos a la función truth-tree-aux.

Ejemplos

```
61  CL-USER> (expand '((=> A (^ B (! A)))))  
62  (^ (V (! A) (^ B (! A))))  
63  CL-USER>
```

Vemos como, a pesar de que solo haya una expresión, hay que pasarlo como lista de listas.

```
64  CL-USER> (expand '((=> A (^ B (! A))) (=> A B) (^ A B)))  
65  (^ (V (! A) (^ B (! A))) (V (! A) B) (^ A B))  
66  CL-USER>
```

En este caso, hemos metido tres expresiones, devolviendo la función toda la base transformada y expandida (como conjunción).

Apartado 4.2

Para este apartado hemos realizado también funciones auxiliares. Entre ellas se encuentran funciones que sirven para construir el árbol de verdad y funciones cuyo objetivo es ver si hay contradicciones o no en una rama, y por tanto determinan si la base de conocimiento es SAT o UNSAT

Truth-tree-aux

Pseudocódigo

Entrada: fbf's - Formulas bien formadas (FBF) que forman una base

lst - lista de atomos

Salida: literales - Listas de atomos que habrá que analizar después

Procesamiento:

si fbf's es un signo and, se devuelve lst

si fbf's es un literal

 si lst está vacía, hacemos la recursividad con fbf's como lst y and como fbf's

 si no, hacemos la recursividad combinando el literal con cada uno de los elementos que haya en lst

si el primer elemento es un signo and
 si es el único elemento, devolvemos lst
 si no, hacemos la recursividad con una nueva lst y el siguiente argumento del
 and
 si el primer elemento es un or,
 aplicamos mapcar a cada argumento del or
 en caso contrario,
 devolvemos nil

Código

```
;;;;; truth-tree-aux
;;;; Funcion auxiliar que recibe una expresion y construye su arbol de verdad para
;;;; determinar si es SAT o UNSAT
```

```
(defun truth-tree-aux (lst fbfs)
```

```
(cond ((eql +and+ fbfs) lst)
      ((literal-p fbfs)
       (if (null lst)
           (truth-tree-aux (list fbfs) +and+)
           (truth-tree-aux (insert-elt fbfs lst) +and+)))
      ((eql +and+ (first fbfs))
       (if (null (second fbfs))
           lst
           (truth-tree-aux (truth-tree-aux lst (second fbfs)) (expand (cddr fbfs))))
       )
      ((eql +or+ (first fbfs))
       (mapcar #'(lambda(x) (truth-tree-aux lst x)) (rest fbfs)))
      (t nil)
    )
```

```
)
```

Comentarios

Esta función tiene como objetivo devolver una lista de átomos, a partir de una base de conocimiento, que habrá que analizar para comprobar si se encuentra alguna contradicción. Esta lista de átomos es en realidad una lista de listas, en la que cada una de ellas es cada rama del árbol de verdad. La idea principal que hemos utilizado para construir el árbol ha sido la de hacer una recursión por cada uno de los elementos en caso de que fuera un or (de esta forma creábamos tantas listas, es decir, tantas ramas como hiciera falta), y, en el caso de que fuera un and, combinar cada uno de los argumentos con cada lista de átomos. Para ello hemos utilizado una función auxiliar muy parecida a la utilizada en el ejercicio tres de esta práctica, que combinaba un elemento con una lista. De esta forma, conseguimos que la salida de la función sea la lista de listas (que representan cada rama), que habrá que analizar después.

Ejemplos

```

148  []
149  CL-USER> (truth-tree-aux nil '(^ (V (! A) (^ B (! A))))) 
150  [](((! A)) (((! A) B))) []
151  []
152  CL-USER> (truth-tree-aux nil '(^ (^ (v A B)))) 
153  []((A) (B)) []
154  []
155  CL-USER>

```

LF ⚠ 1 deprecation UTF-8 Lisp REPL

Insert-elt

Pseudocódigo

Entrada: elt- Elemento a combinar

lst- Lista con la que combinar

Salida: Lista con el resultado de combinar el elemento con cada uno de los elementos de la lista lst

Procesamiento:

si lst es nil, devuelve nil

si el primer elemento de lst es un literal, hace cons de combinar en una lista elt con el elemento más la recursiva del resto de lst

en otro caso, hace cons de combinar en un cons elt con el elemento más la recursiva del resto de lst

Código

```
;;;;; insert-elt
;;;;; Recibe un elemento y una lista y devuelve el elemento combinado con cada elemento de la lista
```

```
(defun insert-elt (elt lst)
  (cond ((null lst) nil)
        ((literal-p (first lst))
         (cons (list elt (first lst)) (insert-elt elt (rest lst))))
        (t (cons (cons elt (first lst)) (insert-elt elt (rest lst)))))
        )
  )
```

Comentarios

Esta función, como ya hemos comentado, es muy parecida a las implementadas en el ejercicio 3. En este caso, la utilizamos en truth-tree-aux para meter un literal en más de una lista distinta, lo que ocurre en el caso que nos encontramos un and y en la lista de átomos haya más de una lista, lo que significa que en el árbol hay más de una rama, pero el elemento en cuestión es común a todas ellas.

Ejemplos

```

157 []
158 CL-USER> (insert-elt 'A '(B C D))
159 []((A B) (A C) (A D)) []
160 []

```

En este ejemplo podemos ver claramente como es su comportamiento.

Las siguientes funciones sirven para determinar si hay contradicciones o no en un árbol de verdad, el cual ya hemos construido con la función truth-tree-aux

Check-lst-lst

Pseudocódigo

Entrada: list-lista a analizar

Salida: T – No es una lista de listas

F – Es una lista de listas

Procesamiento:

si lst es nil, devuelve true

si el primer elemento de lst es un literal, hace la recursividad con el resto de la lista
en caso contrario devuelve nil

Código

```
;;;; check-lst-lst
;;; Recibe una lista y comprueba si es una lista de listas
```

```
(defun check-lst-lst (lst)
  (if (null lst)
      t
      (if (literal-p (first lst))
          (check-lst-lst (rest lst))
          nil)))
```

Comentarios

Esta función sirve para comprobar si una lista es una lista de listas o no. Nos será muy útil a la hora de llamar a otras funciones auxiliares.

Ejemplos

```
169  []
170 CL-USER> (check-lst-lst '(A B C))
171  []
172 CL-USER> (check-lst-lst '((A B C) (D E F)))
173  []
174 CL-USER> (check-lst-lst 'NIL)
175  []
```

En estos ejemplos vemos claramente su funcionamiento

Truth-tree-check

Pseudocódigo

Entrada: list-conjunto de listas a analizar

Salida: T – No hay contradicciones

F – Hay contradicciones

Procesamiento:

si lst es nil, devuelve nil

si el primer elemento de lst es un literal, devuelve true

si lst no es una lista de listas, mira a ver si hay contradiccion en la lista
si hay más de una lista en lst, hace la recursiva con el rest

Código

```
;;;;; truth-tree-check
;;;; Recibe una expresión y comprueba si hay contradicciones

(defun truth-tree-check (lst)
  (cond ((null lst) nil)
        ((literal-p lst) t)
        ((check-lst-lst lst)
         (contradiction lst))
        ((null (truth-tree-check (first lst)))
         (truth-tree-check (rest lst)))
        (t t)
      )
)
```

Comentarios

Esta función comprueba si hay contradicciones en un árbol. Su funcionamiento consiste en ver si hay contradicción en la primera lista, y si no la hay, directamente devuelve true ya que con que una rama esté bien sirve para que la base sea SAT. Si hay contradicción mira a ver la siguiente lista, y en caso de no encontrar ninguna que devuelva true, devuelve nil, ya que significaría que no hay ninguna rama SAT, y por lo tanto la base es UNSAT.

Ejemplos

```
318 
319 CL-USER> (truth-tree-check '(((! A) (((! A) B))))
320 
321 
322 CL-USER> (truth-tree-check '(((! A) (((! A) A))))
323 
324 
325 CL-USER> (truth-tree-check '(((! A) A) (((! A) A))))
326 
327 
328 CL-USER>
```

LF □ 1 deprecation UTF-8 Lisp

Estos ejemplos, aunque simples, son muy ilustrativos de como funciona la función.

Nos quedan, por tanto, las funciones necesarias para determinar si en una lista específica hay contradicciones o no. Para ello utilizamos dos funciones más.

Contradiction-aux

Pseudocódigo

Entrada: list-expresión a analizar
 frst-elemento a comparar
Salida: T – No hay contradicciones
 F – Hay contradicciones

Procesamiento:

si lst es nil, devuelve true
 si el frst es un literal negativo
 si el siguiente elemento es el mismo en positivo, devuelve nil
 si no, hace la recursividad con el siguiente elemento de la lista
 si el frst es un literal positivo y el segundo de la lista es negativo
 si es el mismo literal, devuelve nil,
 si no, hace la recursividad con el siguiente elemento de la lista
 en caso contrario, hace la recursividad con el resto de la lista

Código

```
;;;;;;
;; contradiction-aux
;; Funcion auxiliar de contradiction
```

```
(defun contradiction-aux (frst lst)
  (cond ((null lst) t)
        ((negative-literal-p frst)
         (if (eql (second frst) (second lst))
             nil
             (contradiction-aux frst (rest lst))))
        ((and (positive-literal-p frst) (negative-literal-p (second lst)))
         (if (eql frst (second (second lst)))
             nil
             (contradiction-aux frst (rest lst))))
        (t (contradiction-aux frst (rest lst)))
  )
)
```

Comentarios

Esta función comprueba si hay contradicciones en una lista comparando cada elemento con uno pasado por parámetro. Básicamente, su funcionamiento es el siguiente: Dada una lista y un elemento, si el elemento es positivo y en la lista se encuentra el mismo elemento en negativo devuelve ni, al igual que ocurriría si el elemento fuera negativo y se encontrara con un elemento positivo. Si se llega al final de la lista y no ha encontrado contradicción, entonces devuelve true.

Veremos ejemplos de esta función con la siguiente, que es la general.

Contradiccion

Pseudocódigo

Entrada: list-lista a analizar
Salida: T – No hay contradicciones
 F – Hay contradicciones

Procesamiento:

si lst es nil, devuelve true
 si el primer elemento no tiene ninguna contradicción, hace la recursiva del rest de lst
 en caso contrario, devuelve nil

Código

```
;;;;;;
;; contradiction
;; Recibe una expresion y devuelve si es contradictoria o no
```

```
((defun contradiction (lst)
```

```
(cond ((null lst) t)
      ((contradiction-aux (first lst) lst)
       (contradiction (rest lst)))
      (t nil)))
```

Comentarios

Esta función comprueba si hay contradicciones en una lista. Para ello, simplemente se ocupa de ir variando el elemento frst de la función auxiliar que hemos comentado antes, ya que se tiene que comprobar con todos los elementos de la lista. Si llegamos al final, devolvemos true, y, al primer caso en el que la función auxiliar nos de una contradicción, devolvemos nil

Ejemplos

Vamos a ver algunos ejemplos con la traza de las funciones para así ver mejor su funcionamiento

```

590 []
591 CL-USER> (contradiction '(A (! A)))
592 [] 0: (CONTRADICTION (A (! A)))
593 [] 1: (CONTRADICTION-AUX A (A (! A)))
594 [] 1: CONTRADICTION-AUX returned NIL
595 [] 0: CONTRADICTION returned NIL
596 []NIL[]

597 []
598 CL-USER> (contradiction '(A (! A) B C))
599 [] 0: (CONTRADICTION (A (! A) B C))
600 [] 1: (CONTRADICTION-AUX A (A (! A) B C))
601 [] 1: CONTRADICTION-AUX returned NIL
602 [] 0: CONTRADICTION returned NIL
603 []NIL[]

604 []
605 CL-USER> (contradiction '(A B C (! C)))

```

LF ⚠ 1 deprecation

```

04 []
05 CL-USER> (contradiction '(A B C (! C)))
06 [] 0: (CONTRADICTION (A B C (! C)))
07 [] 1: (CONTRADICTION-AUX A (A B C (! C)))
08 [] 2: (CONTRADICTION-AUX A (B C (! C)))
09 [] 3: (CONTRADICTION-AUX A (C (! C)))
10 [] 4: (CONTRADICTION-AUX A (!! C))
11 [] 5: (CONTRADICTION-AUX A NIL)
12 [] 5: CONTRADICTION-AUX returned T
13 [] 4: CONTRADICTION-AUX returned T
14 [] 3: CONTRADICTION-AUX returned T
15 [] 2: CONTRADICTION-AUX returned T
16 [] 1: CONTRADICTION-AUX returned T
17 [] 1: (CONTRADICTION (B C (! C)))
18 [] 2: (CONTRADICTION-AUX B (B C (! C)))
19 [] 3: (CONTRADICTION-AUX B (C (! C)))
20 [] 4: (CONTRADICTION-AUX B (!! C))
21 [] 5: (CONTRADICTION-AUX B NIL)
22 [] 5: CONTRADICTION-AUX returned T
23 [] 4: CONTRADICTION-AUX returned T
24 [] 3: CONTRADICTION-AUX returned T
25 [] 2: CONTRADICTION-AUX returned T
26 [] 2: (CONTRADICTION (C (! C)))
27 [] 3: (CONTRADICTION-AUX C (C (! C)))
28 [] 3: CONTRADICTION-AUX returned NIL
29 [] 2: CONTRADICTION returned NIL
30 [] 1: CONTRADICTION returned NIL
31 [] 0: CONTRADICTION returned NIL
32 []NIL[]
33 []

```

En estos ejemplos vemos claramente el funcionamiento de la función. Nótese que en los dos primeros, la contradicción está muy al principio de la lista. Sin embargo, en el segundo, ésta se encuentra al final de la lista, por lo que se ve obligado a trabajar más para llegar a ella.

Finalmente, llegamos a la última función, la cuál es la más simple de todas, ya que hace uso de todas las immplementadas hasta ahora.

Truth-tree

Pseudocódigo

Entrada: fbfs - Formula bien formada (FBF) a analizar

Salida: T – Fbfs es SAT

F – Fbfs es UNSAT

Procesamiento:

si fbfs es nil, devuelve nil

si no, devuelve el resultado de aplicar truth-tree-check al resultado de truth-tree-aux, a la cual se le pasa como argumento el resultado de aplicar expand a fbfs

Código

```
;;;;; truth-tree
;;;; Recibe una expresion y construye su arbol de verdad para
;;;; determinar si es SAT o UNSAT
```

```
(defun truth-tree (fbfs)
  (if (null (first fbfs)) nil
    (truth-tree-check (truth-tree-aux nil (expand fbfs))))
```

```
)
```

Comentarios

Esta función es la que determina si la base es SAT o UNSAT. Para ello, primero hacemos expand de la base, para sacar la expresión equivalente, luego construimos el árbol, y finalmente vemos a ver si hay contradicciones en él.

Ejemplos

```
600  []
61 CL-USER> (truth-tree '((())))
62  []NIL[]
63 []
64 CL-USER> (truth-tree '(((^ (v A B)))))
65  []T[]
66 []
67 CL-USER> (truth-tree '(((^ (! B) B))))
68  []NIL[]
69 []
70 CL-USER> (truth-tree '(((<=> (=> (^ P Q) R) (=> P (v (! Q) R))))))
71  []T[]
72 []
73 CL-USER> (truth-tree '(((>= A (^ B (! A))))))
74  []T[]
75 []
76 CL-USER> (truth-tree '(((>= (<=> A (^ A B)) B))))
77  []T[]
78 []
79 CL-USER> (truth-tree '(((>= A B) (^ A (! B)))))
80  []NIL[]
81 []
82 CL-USER> (truth-tree '(((>= A B) (^ A (! B)) (V A B)))
83  []NIL[]
84 .
```

Ejercicio 5

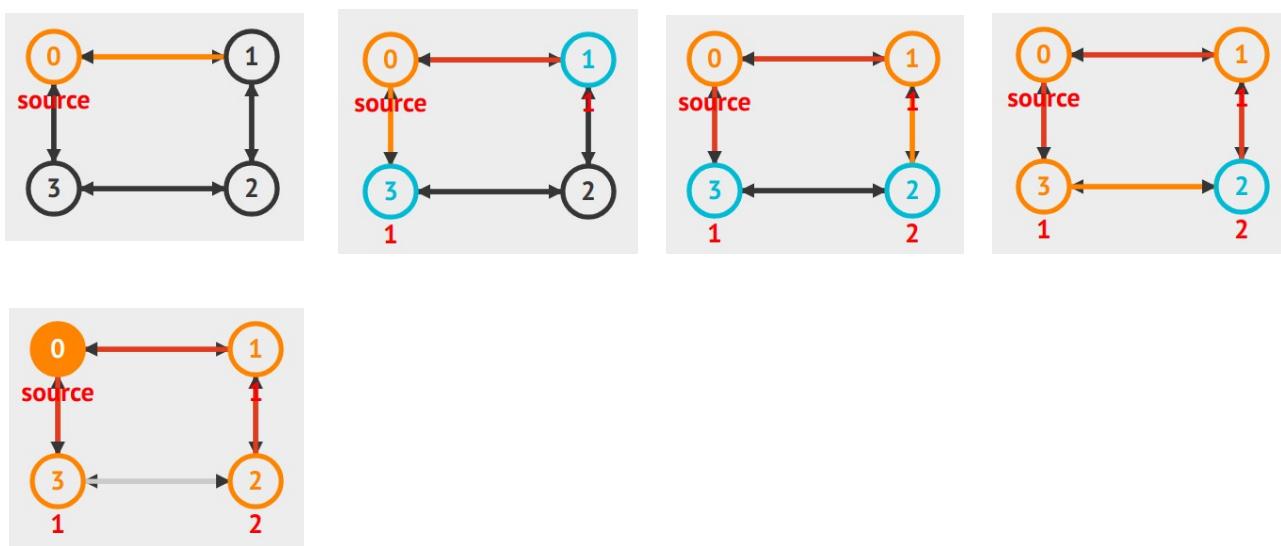
Leyenda: Nodo blanco borde negro(no visitado, no explorado); Nodo blanco borde azul(nodo visitado, no explorado); Nodo naranja(nodo visitado, explorado).

5.1

- Grafos especiales: Podemos encontrar grafos como los cíclicos o los no conexos:

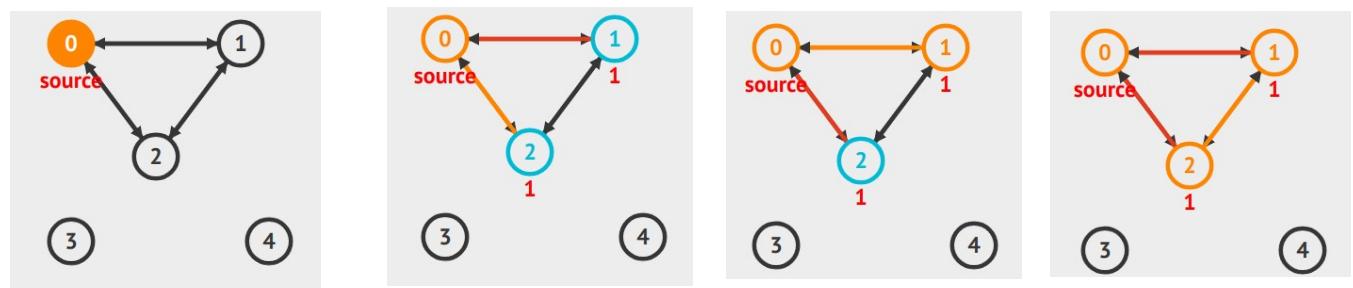
- Cíclicos: Se trata de “*un grafo que se asemeja a un polígono de n lados. Consiste en un camino cerrado en el que no se repite ningún vértice a excepción del primero que aparece dos veces como principio y fin del camino.*” ¹. Pongamos como ejemplo un grafo cuadrado con cuatro nodos no dirigido:

Lista de adyacencias: ((0 1 3) (1 0 2) (2 1 3) (3 0 2))



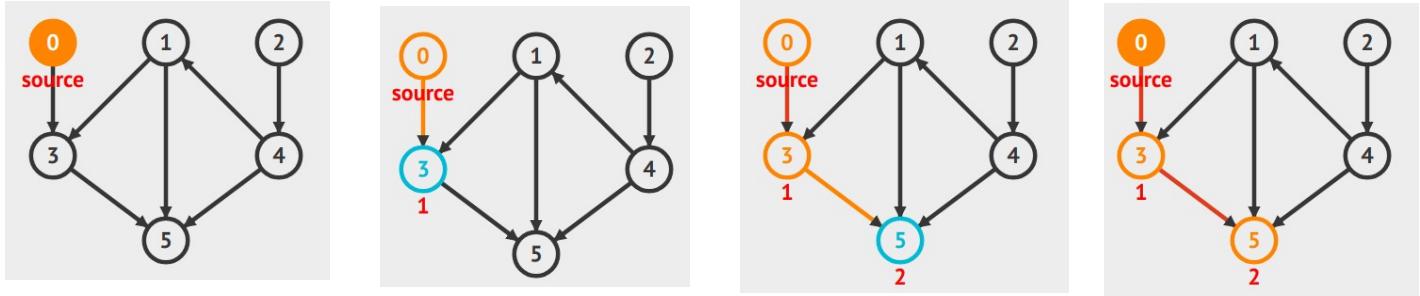
- No conexos:

Lista de adyacencias: ((0 1 2) (1 0 2) (2 0 1) (3) (4))

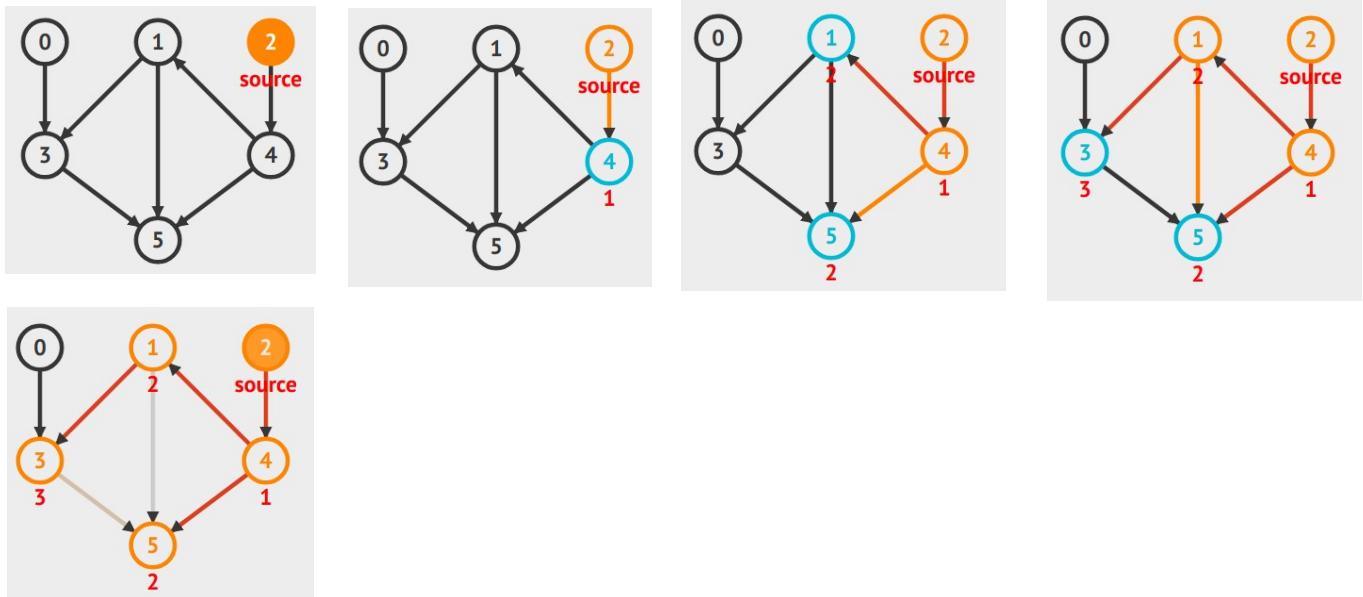


- Ejemplo: Empezamos en el nodo 0:

Lista de adyacencias: ((0 3) (1 3 5) (2 4) (3 5) (4 1 5) (5))

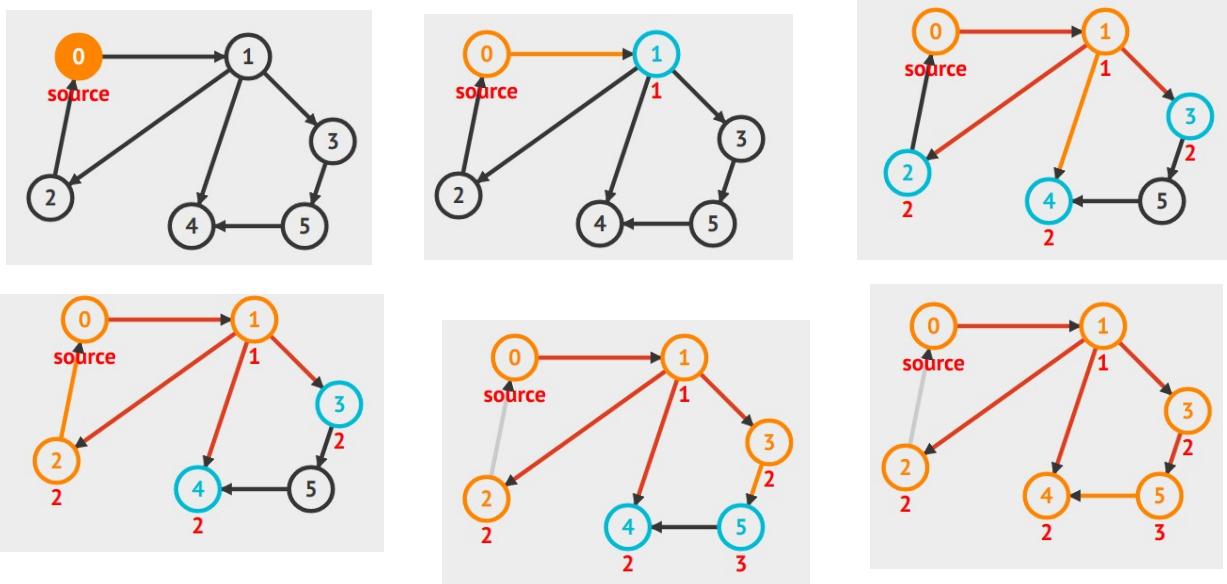


Empezamos en el nodo 2:



- Otro ejemplo:

Lista de adyacencias: ((0 1) (1 2 3 4) (3 1 5) (4) (5 4))



5.2

```
BFS (G, s)          //Where G is the graph and s is the source node2
    let Q be queue.
        Q.enqueue( s ) //Inserting s in queue until all its neighbour
        vertices are marked.

        mark s as visited.
        while ( Q is not empty)
            //Removing that vertex from queue,whose neighbour will be
            visited now
            v = Q.dequeue( )

            //processing all the neighbours of v
            for all neighbours w of v in Graph G
                if w is not visited
                    Q.enqueue( w )           //Stores w in Q to
                    further visit its neighbour
                    mark w as visited.
```

5.4

```
; ;;;;;;;;;;;;;;;;;;;
;;; 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 ))))

;;
; ;;;;;;;;;;;;;;;;;;
```

El argumento "queue" lo tenemos que inicializar como el nodo origen. Definiremos el path como el primer camino que metamos en la queue, donde guardaremos todos los caminos recorrido y el nodo como primer elemento del path. Si el nodo es el nodo final (end) devolvemos el path dado la vuelta para que recupere el orden. Si no, hacemos una llamada recursiva a la función pero siendo la queue la union del resto de elementos de la queue y la llamada a los new-paths (nuevos caminos) desde el nodo. Esta función devolverá todos los caminos descubiertos pero no explorados (añadidos a la cola). Por lo que en bfs vamos añadiendo a la lista los elementos descubiertos pero no explorados y en este algoritmo vamos añadiendo los caminos descubiertos pero no explorados. El algoritmo

acaba cuando el primer elemento del camino es el nodo final (se hace así por la forma en la que añadimos los caminos a la lista).

5.5

La función *shortest-path* lo único que hace es que nos convierte el nodo origen como una lista de listas, es decir que el primer elemento de queue es una lista con el nodo origen dentro. De esta manera podremos aplicar BFS con nodo origen y nodo final.

5.6

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

Empezamos en A y llamamos a la función new-paths que nos devuelve las listas de A y los nodos con los que está conectado (en este caso solo devuelve (D A) ya que solo está conectado con él). Ahora este será nuestra nueva cola, y exploraremos los caminos desde D empezando por A, donde el único que existe es (F D A). Finalmente como F es el nodo final el algoritmo devuelve ese camino dado la vuelta para que este en orden de exploración.

5.7

El grafo ((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 F) (H D E F G)), el nodo inicial será B y el nodo final será G.

```
] CL-USER> (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 F) (H D E F G)))
] (B D G) ]]
```

5.8

Bfs-improved

Pseudocódigo

Entrada: end: nodo final

queue: cola de nodos por explorar
net: grafo

Salida: camino más corto entre dos nodos

nil si no lo encuentra

Procesamiento:

Código

```
; ;;;;;;;;;;;;;;;;;;;;
; ; B r e a d t h - f i r s t - s e a r c h in graphs
```

```
( defun bfs-improved-aux ( end queue net)
  ( if ( null queue ) '()
    ( let* (( path ( first queue ))
           ( node ( first path )))
      ( if ( eql node end )
          ( reverse path )
          (if (null (find node rpt-lst))
              (bfs-improved end (append (rest queue) (new-paths path node net)) net (cons node rpt-lst)))
              (bfs-improved end (rest queue) net rpt-lst))))))
```

Comentarios

El único cambio respecto a la función bfs original es que esta función es capaz de detectar si un nodo ya ha sido explorado previamente. Para ello simplemente lo que hacemos es que la función reciba un cuarto argumento que será una lista donde iremos añadiendo los nodos explorados. Si el nodo ya está en la lista simplemente llamamos a la función con el resto de la cola. Si no está, aplicamos la búsqueda recursiva del bfs original y añadimos el nodo a la lista de nodos ya explorados. De esta manera no entrará en un bucle con grafos como ((A B) (B A) (C)) y busquemos el camino entre A y C.

Ejemplos

```
CL-USER> (shortest-path-improved '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 F) (H D E F G)) '())
NIL
CL-USER> (shortest-path-improved 'B 'G '((A B) (B C) (C)) '())
NIL
CL-USER> (shortest-path-improved '0 '4 '((0 1) (1 2 3 4) (3 1 5) (4) (5 4)) '())
NIL
CL-USER> (shortest-path-improved '0 '3 '((0 3) (1 3 5) (2 4) (3 5) (4 1 5) (5)) '())
NIL
CL-USER> (shortest-path-improved '0 '2 '((0 1 2) (1 0 2) (2 0 1) (3) (4)) '())
NIL
CL-USER> |
```

REFERENCIAS

1. https://es.wikipedia.org/wiki/Grafo_ciclo
2. <https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/>