PRÁCTICA 1: LISP

1. **Vector más cercano a un vector dado:**
   1. **Norma p de un vector:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PSEUDOCÓDIGO**

Entrada: x (vector del que se calcula la norma)

p (orden de la norma)

Salida:  n (norma p de x)

Procesamiento:

Si x es vacio,

      evalúa a 0

en caso contrario

evalúa a (|primer-elemento(x)|^p + normap(siguientes-elementos(x))^(1/p)

1. **Usando recursión**

**CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION AUXLIAR**

**;;;lp-rec-aux**

**;;;Funcion que realiza el sumatorio de los elementos del vector**

**;;;elevados al exponente p**

**;;;**

**;;;INPUT : x: vector, en forma de lista**

**;;;p: orden de la norma que se quiere calcular**

**;;;**

**;;;OUTPUT:sumatorio interior de la norma**

**(defun lp-rec-aux(x p)**

**(if (null x)**

**0**

**(+ (expt (abs (first x)) p)**

**(lp-rec-aux (rest x) p))))**

**;;;EJEMPLOS**

**;;;(lp-rec-aux nil 2) ; -> 0 caso especial**

**;;;(lp-rec-aux '(3 4) 2) ;-> 25 caso general**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;; lp-rec (x p)**

**;;; Calcula la norma Lp de un vector de forma recursiva**

**;;;**

**;;; INPUT: x: vector, representado como una lista**

**;;; p: orden de la norma que se quiere calcular**

**;;;**

**;;; OUTPUT: norma Lp de x**

**;;;**

**(defun lp-rec (x p)**

**(expt (lp-rec-aux x p) (/ 1 p)))**

**;;;EJEMPLOS**

**;;;(lp-rec '() 2); -> 0 caso especial**

**;;;(lp-rec '(3 4) 2);-> 5.0 caso general**

1. **Usando mapcar**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;; lp-mapcar (x p)**

**;;; Calcula la norma Lp de un vector usando mapcar**

**;;;**

**;;; INPUT: x: vector, representado como una lista**

**;;; p: orden de la norma que se quiere calcular**

**;;;**

**;;; OUTPUT: norma Lp de x**

**;;;**

**(defun lp-mapcar (x p)**

**(expt (apply #'+ (mapcar #'(lambda (n) (expt (abs n) p)) x)) (/ 1 p)))**

**;;;EJEMPLOS**

**;;;(lp-mapcar '() 2); -> 0 caso especial**

**;;;(lp-mapcar '(3 4) 2);-> 5.0 caso general**

**;;;**

**;;;COMENTARIOS**

**;;;al hacer apply #'+ con un vector es nil da 0**

**COMENTARIOS**

El código con mapcar es mucho más claro y sencillo que el de recursión, aunque el último es más intuitivo puesto que se trata de un sumatorio. Asimismo, cuando usamos mapcar, Lisp se encarga de comprobar los casos en los que la lista sea vacía y por tanto ahorramos comprobaciones.

En la función recursiva hemos necesitado una función auxiliar para calcular el sumatorio. En la función principal ya podemos calcular la raíz una vez hemos llamado a la auxiliar.

* 1. **Norma euclídea y norma 1**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Una vez definidas las dos funciones anteriores, tan solo tenemos que llamarlas utilizando el valor de p que corresponda.

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;; l2-rec (x)**

**;;; Calcula la norma L2 de un vector de forma recursiva**

**;;;**

**;;; INPUT: x: vector**

**;;;**

**;;; OUTPUT: norma L2 de x**

**;;;**

**(defun l2-rec (x)**

**(lp-rec x 2))**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;; l2-mapcar (x)**

**;;; Calcula la norma L2 de un vector usando mapcar**

**;;;**

**;;; INPUT: x: vector**

**;;;**

**;;; OUTPUT: norma L2 de x**

**;;;**

**(defun l2-mapcar (x)**

**(lp-mapcar x 2))**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;; l1-rec (x)**

**;;; Calcula la norma L1 de un vector de forma recursiva**

**;;;**

**;;; INPUT: x: vector**

**;;;**

**;;; OUTPUT: norma L1 de x**

**;;;**

**(defun l1-rec (x)**

**(lp-rec x 1))**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;; l1-mapcar (x)**

**;;; Calcula la norma L1 de un vector usando mapcar**

**;;;**

**;;; INPUT: x: vector**

**;;;**

**;;; OUTPUT: norma L1 de x**

**;;;**

**(defun l1-mapcar (x)**

**(lp-mapcar x 1))**

**COMENTARIOS**

Para probar la norma infinito podemos introducir vectores con valores negativos y ver que el valor absoluto se aplica correctamente. También podemos introducir vectores distintos cuya norma euclídea es igual pero la norma infinito cambia, por ejemplo, (1 1) tiene norma euclídea y (1 0), 1, pero ambos tienen norma infinito 1.

* 1. **Función nearest**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**PSEUDOCÓDIGO**

Entrada: lst-vectors(lista de vectores a comparar)

vector (vector referencia contra el que se compara)

fn-dist: referencia a función para medir distancias

Salida:   vector de entre los de lst-vectors más cercano al de referencia

Procesamiento:

si no( vacío(vector) o vacío(lst-vectors) )

lista-distancias= aplicar (calcular-distancia(lst-vectores))

n=posición(minimo(lista-distancias))

devolver (lst-vectors(n))

**CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION AUXILIAR**

**;;;eval-dist**

**;;;evalua la distancia entre el vector ref y el vectro vec mediente**

**;;;la funcion fn-dist**

**;;;INPUT : ref ; primer vector**

**;;;vec: segundo vector**

**;;;fn-dist: funcion con la que se consigue la distancia entre los vectores**

**;;;**

**;;;OUTPUT: distancia entre los vectores**

**(defun eval-dist(ref vec fn-dist)**

**(funcall fn-dist (mapcar #'- ref vec)))**

**;;;EJEMPLOS**

**;;;(eval-dist '()'(1 0) #'l2-mapcar)-> 0 caso particular**

**;;;(eval-dist '(1 0)'(1 0) #'l2-mapcar)-> 0 caso general**

**;;;(eval-dist '(0 0)'(1 0) #'l2-mapcar)-> 1 caso general**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;; nearest (lst-vectors vector fn-dist)**

**;;; Calcula de una lista de vectores el vector más cercano a uno dado,**

**;;; usando la función de distancia especificada**

**;;;**

**;;; INPUT: lst-vectors: lista de vectores para los que calcular la distancia**

**;;; vector: vector referencia, representado como una lista**

**;;; fn-dist: referencia a función para medir distancias**

**;;;**

**;;; OUTPUT: vector de entre los de lst-vectors más cercano al de referencia**

**;;;**

**(defun nearest (lst-vectors vector fn-dist)**

**(when vector**

**(let ((distancias (mapcar #'(lambda(x) (eval-dist vector x fn-dist))**

**lst-vectors)))**

**(first (nthcdr (position (apply #'min distancias)**

**distancias) lst-vectors)))))**

**;;;EJEMPLOS**

**;;;(nearest '((2 3) (1 2) ()) '(0 0) #'l2-mapcar)-> NIL caso particular**

**;;;(nearest '(()) '(0 0) #'l2-mapcar)-> NIL caso particular**

**;;;(nearest '((2 3)(1 2) (5 7)) '() #'l2-mapcar)-> NIL caso particular**

**;;;(nearest '((2 3) (1 2)) '(0 0) #'l2-mapcar)-> (1 2) caso general**

**;;;**

**;;;COMENTARIOS**

**;;;la funcion espera una lista de listas y esto no se comprueba,**

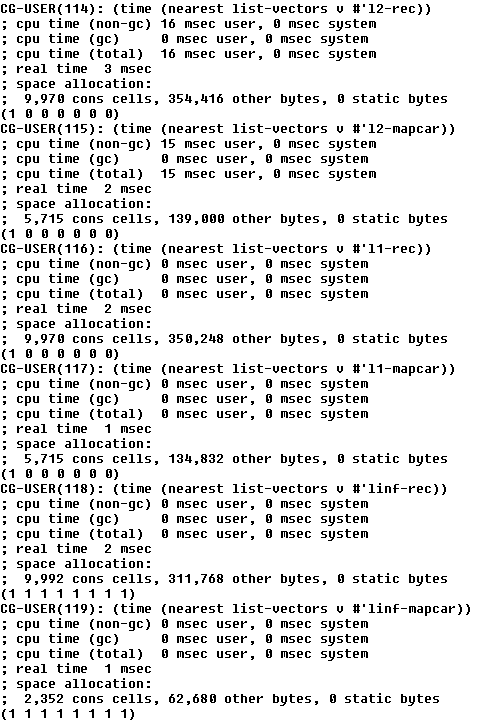
**;;;se sigue el principio RTFM**

**COMENTARIOS**

Se ha utilizado mapcar para la implementación debido a que nos parecía más sencillo que una opción recursiva y por posibles ahorros de memoria (como al final se confirma con las pruebas realizadas a continuación)

Hemos necesitado una función auxiliar que compara un único vector contra el vector referencia.

* 1. **Pruebas función nearest con diferentes normas**



**COMENTARIOS**

Hemos medido los tiempos con veinticinco vectores de dimensión nueve y vemos la distancia al origen. Observamos que la norma euclídea es la más costosa, ya que se ha obtenido un tiempo de 3 ms. Asimismo, en general el mapcar es más eficiente que la recursión, tanto temporal como espacialmente (podemos ver como el número de bloques cons utilizados en todas las funciones recursivas es casi 10.000 mientras que en el mapcar se reduce aproximadamente a la mitad).

1. **Ceros de una función**
   1. **Función : secante**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PSEUDOCÓDIGO**

Siguiendo el pseudocódigo del enunciado se ha implementado el código que se presenta más abajo.

**CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;; secante (f tol-abs max-iter par-semillas)**

**;;; Estima el cero de una función mediante el método de la secante**

**;;;**

**;;; INPUT: f: función cuyo cero se desea encontrar**

**;;; tol-abs: tolerancia para convergencia**

**;;; max-iter: máximo número de iteraciones**

**;;; par-semillas: estimaciones iniciales del cero (x0 x1)**

**;;;**

**;;; OUTPUT: estimación del cero de f, o NIL si no converge**

**;;;**

**(defun secante (f tol-abs max-iter par-semillas)**

**(unless (or (< max-iter 0) (null par-semillas))**

**(let\* ((x0 (first par-semillas)) (x1 (second par-semillas))**

**(fx0 (funcall f x0)) (fx1 (funcall f x1)))**

**(if (< (abs (apply #'- par-semillas)) tol-abs)**

**x1**

**(secante f tol-abs (- max-iter 1)**

**(list x1 (- x1 (\* fx1 (/ (- x1 x0)**

**(- fx1 fx0))))))))))**

**;;;EJEMPLOS**

**;;;caso de no convergencia**

**;;;(setf funcion (lambda (x) (+ (sin x) 1.5)))**

**;;;(secante funcion 1e-3 50 '(1 2))->NIL**

**;;;**

**;;;En el caso general nos remitimos al ejemplo proporcionado en el enunciado**

**COMENTARIOS**

Sobre la implementación elegida destacamos que en cada llamada recursiva a la función creamos una nueva lista donde X1 pasa a ser X0 y el nuevo X1 se calcula mediante la fórmula que se nos ha proporcionado.

Hemos probado funciones que no tienen ceros para ver que ocurría. Una de ellas es sin(x)+1.5 que cumple el máximo de iteraciones y acaba con salida NIL. Otra de ellas es x2+1 que no llega al máximo de iteraciones puesto que las semillas crecen rápidamente y se desborda la memoria.

* 1. **Función : un-cero-secante**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PSEUDOCÓDIGO**

Entrada: f: función

Tol-abs: tolerancia

max-iter: máximo iteraciones

pares-semillas: lista-pares-semillas

Salida: El primer cero encontrado

Procesamiento:

si no (secante (primer-elemento(pares-semillas)))

entonces un-cero-secante(siguientes-elementos(pares-semillas))

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;; un-cero-secante (f tol-abs max-iter pares-semillas)**

**;;; Prueba con distintos pares de semillas iniciales hasta que**

**;;; la secante converge**

**;;;**

**;;; INPUT: f: función de la que se desea encontrar un cero**

**;;; tol-abs: tolerancia para convergencia**

**;;; max-iter: máximo número de iteraciones**

**;;; pares-semillas: pares de semillas con las que invocar a secante**

**;;;**

**;;; OUTPUT: el primer cero de f que se encuentre, o NIL si se diverge**

**;;; para todos los pares de semillas**

**;;;**

**(defun un-cero-secante (f tol-abs max-iter pares-semillas)**

**(when pares-semillas**

**(let ((par1 (secante f tol-abs max-iter**

**(first pares-semillas))))**

**(if par1**

**par1**

**(un-cero-secante f tol-abs max-iter**

**(rest pares-semillas))))))**

**;;;EJEMPLOS**

**;;;caso de no convergencia**

**;;;(setf funcion (lambda (x) (+ (sin x) 1.5)))**

**;;;(un-cero-secante funcion 1e-3 50 '((1 2)(4 5)))->NIL**

**;;;**

**;;;En el caso general nos remitimos al ejemplo proporcionado en el enunciado**

**;;;**

**;;;COMENTARIOS**

**;;;**

**;;;la funcion espera una lista de listas y esto no se comprueba,**

**;;;se sigue el principio RTFM**

**COMENTARIOS**

La función solo admite una lista de listas, en caso contrario se produce un error.

De nuevo hemos probado el caso de no convergencia con sin(x)+1.5 cumple el máximo de iteraciones con cada par de semillas.

* 1. **Función: todos-ceros-secante**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PSEUDOCÓDIGO**

Entrada: f: función de la que se desea encontrar un cero

tol-abs: tolerancia para convergencia

max-iter: máximo número de iteraciones

pares-semillas: pares de semillas con las que invocar a secante

Salida: Lista con todos los ceros encontrados a partir de las semillas

Procesamiento:

si no (vacío (pares-semillas)

secante(primer-elemento(pares-semillas))

todos-ceros-secante(siguientes-elementos(pares-semillas))

**CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;; todos-ceros-secante (f tol-abs max-iter pares-semillas)**

**;;; Prueba con distintas pares de semillas iniciales y devuelve**

**;;; las raíces encontradas por la secante para dichos pares**

**;;;**

**;;; INPUT: f: función de la que se desea encontrar un cero**

**;;; tol-abs: tolerancia para convergencia**

**;;; max-iter: máximo número de iteraciones**

**;;; pares-semillas: pares de semillas con las que invocar a secante**

**;;;**

**;;; OUTPUT: todas las raíces que se encuentren, o NIL si se diverge**

**;;; para todos los pares de semillas**

**;;;**

**(defun todos-ceros-secante (f tol-abs max-iter pares-semillas)**

**(when pares-semillas**

**(cons (secante f tol-abs max-iter (first pares-semillas))**

**(todos-ceros-secante f tol-abs max-iter**

**(rest pares-semillas)))))**

**;;;EJEMPLOS**

**;;;caso de no convergencia**

**;;;(setf funcion (lambda (x) (+ (sin x) 1.5)))**

**;;;(todos-ceros-secante funcion tol iters semillas)->(NIL)**

**;;;**

**;;;En el caso general nos remitimos al ejemplo proporcionado en el enunciado**

**COMENTARIOS**

Se espera que se conozcan las semillas previamente, ya que la función no calcula las semillas ni los intervalos donde se encuentran las raíces, simplemente las evalúa y devuelve la lista de resultados.

1. **Combinación de listas**
   1. **Combinación elemento-lista**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PSEUDOCÓDIGO**

Entrada: elt: elemento

lst: lista

Salida: lista que combina el elemento con la lista introducida

Procesamiento:

Si no ( null(lst))

par( lista(elt, primer-elemento(lst)), combine-elt-lst(siguientes-elementos(lst))

**CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;; combine-elt-lst (elt lst)**

**;;; Combina un elemento con una lista devolviendo una lista que**

**;;; contiene listas con el formato ((elt first)(elt second)...)**

**;;; donde first y second son respectivamente los 2 primeros elementos**

**;;; de la lista**

**;;;**

**;;; INPUT: elt: elemento**

**;;; lst: lista**

**;;;**

**;;; OUTPUT: lista que combina el elemto con la lista introducida**

**(defun combine-elt-lst (elt lst)**

**(when lst**

**(cons ( list elt (first lst)) (combine-elt-lst elt (rest lst)))))**

**;;;EJEMPLOS**

**;;;(combine-elt-lst 'a '())-> NIL caso particular**

**;;;(combine-elt-lst 5 '(a b))-> ((5 A)(5 B)) caso general**

**COMENTARIOS**

Debido a que no se especifica en el enunciado se admite el caso en el que el elemento es la lista vacía.

* 1. **Combinación lista-lista**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PSEUDOCÓDIGO**

Entrada: lst1: lista1

lst2: lista2

Salida: lista que realiza el producto vectorial de las dos listas entrantes

Procesamiento:

sino( null(lst1) o null(lst2))

une((combine-elt-lst(primer-elemento(lst1), lst2),

combine-elt-lst(siguientes-elementos(lst1),lst2))

**CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;; combine-lst-lst (lst1 lst2)**

**;;; Realiza el producto cartesiano de las dos listas**

**;;;**

**;;; INPUT: lst1: lista1**

**;;; lst2: lista2**

**;;; OUTPUT: lista con las tuplas con las combinaciones producidas por el producto cartesiano**

**(defun combine-lst-lst (lst1 lst2)**

**(unless (or(null lst1) (null lst2))**

**(append (combine-elt-lst (first lst1) lst2) (combine-lst-lst (rest lst1) lst2))))**

**;;;EJEMPLOS**

**;;;(combine-lst-lst '(1 2)'())-> NIL caso particular**

**;;;(combine-lst-lst '()'(1 2))-> NIL caso particular**

**;;;(combine-lst-lst '(1 2)'( a b))->((1 A) (1 B) (2 A) (2 B)) caso general**

* 1. **Combinación lista de listas**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PSEUDOCÓDIGO**

Entrada: lstolsts: lista de listas con las que realizaremos las disposiciones

Salida: lista con las diferentes disposiciones producidas por la función

Procesamiento:

(si formato-okp(lstolsts))

l1=primer-elemento(lstolsts)

lr= siguientes-elementos(lstolsts)

primero= primer-elemento(l1)

combinar-elemento-lista(primero, lr)

combinar-lst-of-lsts(lista(siguientes-elementos(l1),lr))

**CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION AUXLIAR**

**;;;check-lists-okp(listoflsts)**

**;;;Funcion que comprueba que la list-of-lsts no contiene listas vacias**

**;;;**

**;;;INPUT : listoflsts: lista de listas**

**;;;**

**;;;OUTPUT:T si ok, NIL si alguna lista vacia**

**(defun check-lists-okp(listoflsts)**

**(if (null listoflsts)**

**T**

**(when (first listoflsts)**

**(check-lists-okp (rest listoflsts)))))**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;; combine-list-of-lsts (lstolsts)**

**;;; Representa todas las posibles disposiciones de elementos**

**;;; pertenecientes a**

**;;; N listas de forma que en cada disposición aparezca únicamente**

**;;; un elemento de cada lista**

**;;;**

**;;; INPUT: lstolsts: lista de listas con las que realizaremos las disposiciones**

**;;;**

**;;; OUTPUT: lista con las diferentes disposiciones producidas por la funcion**

**(defun combine-list-of-lsts (lstolsts)**

**(when (check-lists-okp lstolsts)**

**(let ((first-lst (first lstolsts)) (other-lsts (rest lstolsts)))**

**(if (null first-lst)**

**'(NIL)**

**(mapcan #'(lambda (x)**

**(mapcar #'(lambda (y) (cons x y))**

**(combine-list-of-lsts other-lsts)))**

**first-lst)))))**

**;;;EJEMPLOS**

**;;;(combine-list-of-lsts '(()))-> (NIL) caso particular**

**;;;(combine-list-of-lsts '((1 2)))-> ((1) (2)) caso particular**

**;;;(combine-list-of-lsts '((1 2)(+)(A B C)))->**

**;;; ((1 + A) (1 + B) (1 + C) (2 + A) (2 + B)**

1. **+ C)) caso general**
2. **Problema SAT**

**4.1. Comprobación de una base de conocimiento**

**4.1.1. Proposición correctamente formada**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PSEUDOCÓDIGO**

Entrada: expresión: lista de listas que representa una proposición en formato prefijo

Salida: T si se sigue el formato especificado, NIL en otro caso

Procesamiento:

if ( átomo(expresión) y no( conector(expresión)))

T

else if ( átomo(expresión) y conector(expresión))

NIL

else

if (conector-unario (primero (expresión ))

comprobar-unario( resto (expresión))

else if (conector-binario (primero (expresión ))

comprobar-binario( resto (expresión))

else if (conector-n-ario (primero (expresión ))

comprobar-n-ario( resto (expresión))

Funciones auxiliares:

comprobar-unario

comprobar-binario

comprobar-n-ario

Procesamiento:

si (comprobar-número-argumentos==OK)

proposición-p(argumentos)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION AUXILIAR**

**;;;permite saber si una expresion tiene el**

**;;;formato de una expresion unaria**

**(defun comprobar-unario-p (expresion)**

**(if (null (rest expresion))**

**(let ((primero (first expresion)))**

**(if (listp primero)**

**(proposicion-p primero)**

**T))))**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION AUXILIAR**

**;;;permite comprobar si una expresion tiene el**

**;;;formato de una expresion binaria**

**(defun comprobar-binario-p (expresion)**

**(when (null (third expresion))**

**(and (proposicion-p (first expresion))**

**(proposicion-p (second expresion)))))**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION AUXILIAR**

**;;;comprueba si una expresion tiene el**

**;;;formato de una expresion binaria**

**(defun comprobar-n-ario-p (expresion)**

**(when expresion**

**(mapcan #'proposicion-p expresion)))**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION PROPOSICON\_P**

**;;;muestra si una proposicion tiene el formato correcto para ser**

**(defun proposicion-p(expresion)**

**(let ((atomop (atom expresion)) (conectorp (connector-p expresion)))**

**(cond**

**((and atomop (not conectorp)) t)**

**((and atomop conectorp) nil)**

**(t**

**(let ((op (first expresion)) (expr (rest expresion)))**

**(cond**

**((unary-connector-p op) (comprobar-unario-p expr))**

**((binary-connector-p op) (comprobar-binario-p expr))**

**((n-ary-connector-p op) (comprobar-n-ario-p expr))**

**(t nil)))))))**

**;;;EJEMPLOS**

**;;; (proposicion-p '()) -> T caso particular, el vacio esta bien**

**;;; formado**

**;;; (proposicion-p 'A) -> T caso general**

**;;; (proposicion-p '(H <=> (¬ H)))-> NIL caso general**

**;;; (proposicion-p '(<=> H (¬ H)))-> T caso general**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**COMENTARIOS**

Como caso especial hemos considerado que el vacío es una proposición bien formada, de manera que actúa como elemento neutro.

**4.1.2. Base correctamente formada**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Procedimiento:

Se llama a la función: proposición-p sobre los diferentes elementos de la base de conocimiento. Si todas las llamadas devuelven un resultado satisfactorio entonces se reconoce a la base de conocimiento como correcta

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION BASE-P**

**;;; permite comprobar si una base de conocimiento**

**;;; tiene el formato correcto de una base**

**(defun base-p(lista-exp)**

**(when (listp lista-exp)**

**(mapcan #'proposicion-p lista-exp)))**

**;;;EJEMPLOS**

**;;; (base-p '(())) -> T caso particular, consdieramos de nuevo el**

**;;; vacío como una proposición**

**;;; (base-p 'A) -> NIL caso particular, una proposicion no es una base**

**;;; (base-p '((<=> H (¬ H)))) -> T caso general, base con una ;;;proposicion**

**;;; (base-p '((<=> A (¬ H)) (<=> P (^ A H)) (<=> H P)))-> T caso ;;;general,**

**;;; base con varias proposiciones**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**COMENTARIOS**

De nuevo el vacío es considerado una proposición valida, no obstante para que sea una base se debe introducir el vacío dentro de una lista.

**4.2. Extraer símbolos de una base de conocimiento**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Procedimiento:

Extraer de manera simultánea los diferentes átomos de las diferentes proposiciones que componen la base de conocimiento, después se eliminan los duplicados producidos por las distintas proposiciones. Para ello implementamos una función (extraer-símbolos-aux) con el siguiente pseudocódigo:

**PSEUDOCÓDIGO**

Entrada: kb: lista de listas que representa una base de conocimiento en formato prefijo

Salida: lista con los átomos de la proposición sin repeticiones

Procesamiento:

si(kb!=vacio)

crear-lista(quitar-conector(primero(kb)), extraer-simbolos(siguientes(kb)))

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION AUXILIAR**

**;;; Extrae los simbolos de una proposicion. Como una proposicion**

**;;; puede contener proposiciones dentro necesitamos una llamada a**

**;;; la funcion principal**

**(defun extrae-simbolos-aux (prop)**

**(if (listp prop)**

**(extrae-simbolos**

**(remove-if #'connector-p prop) )**

**(list prop)))**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION EXTRAE-SIMBOLOS**

**;;; funcion que extrae los simbolos de una base de conocimiento**

**;;; sin repeticiones**

**(defun extrae-simbolos (kb)**

**(when kb**

**(remove-duplicates**

**(mapcan #'extrae-simbolos-aux (remove-if #'connector-p kb)))))**

**;;;EJEMPLOS**

**;;; (extrae-simbolos '()) -> NIL Caso particular, la lista vacia no tiene**

**;;; simbolos**

**;;; (extrae-simbolos '(A)) -> A Caso general**

**;;; (extrae-simbolos '((v (¬ A) A B (¬ B)))) -> (A B) caso general**

**4.3. Generar interpretaciones**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Procesamiento:

A partir de una lista de átomos, mediante combinación de las funciones implementadas en el ejercicio tres (combine-elt-lst y combine-list-of-lsts) conseguimos todas las posibles combinaciones de “true” y “nil” para los átomos pasados como argumento.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION GENERA-LISTA-INTERPRETACIONES**

**;;; coge los atomos de la lista pasada por argumento y**

**;;; devuelve todas las posibles interpretaciones**

**(defun genera-lista-interpretaciones (lst)**

**(when lst**

**(combine-list-of-lsts**

**(mapcar #'(lambda (x) (combine-elt-lst x '(T NIL))) lst))))**

**;;;EJEMPLOS**

**;;; (genera-lista-interpretaciones nil)-> NIL caso particular**

**;;; (genera-lista-interpretaciones '(P))-> (((P T) ((P NIL))) Caso**

**;;; general**

**;;; (genera-lista-interpretaciones '(P I))->**

**;;; (((P T) (I T)) ((P T) (I NIL)) ((P NIL) (I T)) ((P NIL) (I NIL)))**

**;;; Caso general**

**4.4. Interpretación-modelo**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**PSEUDOCÓDIGO**

Función: interpretación-modelo-p

Entrada: interp: interpretación de la que se quiere comprobar si la base es modelo

Kb: base de conocimiento

Procesamiento:

Si: lista-vacia(kb)

Devolver TRUE ;;Elemento neutro de AND

En otro caso: Evaluar-proposicion(primero(sustituir-interpretacion(kb))) AND

(interpretación-modelo-p(resto(kb))

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION AUXILIAR**

**;;; Cambia el nombre de un atomo por su valor de verdad.**

**(defun sustituir-val (int kb)**

**(if (not (null int))**

**(sustituir-val( rest int)**

**(subst**

**; cogemos el elemento a sustituir y**

**;ademas le quitamos los parentesis**

**(first(rest (first int)))**

**; el elemento que va a ser sustituido**

**(first (first int))**

**kb))**

**kb))**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION AUXILIAR**

**;;; Una vez que se ha sustituido el valor de verdad en los atomos**

**;;; de una proposicion, esta funcion la evalua haciendo uso de las**

**;;; funciones definidas anteriormente**

**(defun eval-prop-p (prop)**

**(cond**

**((null (listp prop)) prop)**

**((connector-p(first prop)) (funcall (conseguir-op(first prop)) ;funcall**

**(mapcar #'eval-prop-p**

**(rest prop))))**

**(t (first prop))))**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION INTERPRETACION-MODELO-P**

**;;; funcion que comprueba si una interpretacion**

**;;; es modelo de la base de conocimiento**

**(defun interpretacion-modelo-p (interp kb)**

**(when (base-p kb)**

**(reduce #'(lambda(x y) (and (eval-prop-p x) (eval-prop-p y)))**

**(append (sustituir-val interp kb) '(T)))))**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**COMENTARIOS**

En un principio solo íbamos a definir las funciones bicondicional y condicional usando reglas de equivalencia lógicas y para el resto de operadores, usaríamos los nativos de Lisp. Sin embargo nos dimos cuenta de que necesitábamos poder aplicar los operadores sobre listas, ya que al quitar el operador de una proposición nos queda una lista. Por este motivo hemos definido funciones como “myor” o “mynot”.

**4.5. Encontrar modelos**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**PSEUDOCÓDIGO**

Función: encuentra-modelos

Entrada: Kb: base de conocimiento

Variable global: interp = genera-lista-interpretaciones(extraer-simbolos(kb))

Procesamiento:

Si: lista-vacia(kb)

Devolver NIL

En otro caso: Si: interpretación-modelo-p(primero(interp), primero(kb)) es TRUE

Crear-lista(primero(interp), encuentra-modelos(resto(kb))

En otro caso:

Encuentra-modelos(resto(kb))

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;; FUNCION AUXILIAR**

**;;; comprueba de manera recursiba si las interpretaciones**

**;;; pasadas pro argumento son modelo**

**(defun encuentra-modelos-aux (interps kb)**

**(when interps**

**(if (interpretacion-modelo-p (first interps) kb)**

**(cons (first interps) (encuentra-modelos-aux (rest interps) kb))**

**(encuentra-modelos-aux (rest interps) kb))))**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;; FUNCION ENCUENTRA-MODELOS**

**;;; funcion que encuentra los modelos de entre**

**;;; todas las posibles interpretaciones de kb**

**(defun encuentra-modelos (kb)**

**(when (base-p kb)**

**(encuentra-modelos-aux (genera-lista-interpretaciones**

**(extrae-simbolos kb)) kb)))**

**;;;EJEMPLOS**

**;;;(encuentra-modelos '()) -> NIL caso particular**

**;;;**

**;;;(encuentra-modelos '((=> A (¬ H)) (<=> P (^ A H)) (=> H P))) ->**

**;;; -> (((A T) (H NIL) (P NIL)) ((A NIL) (H NIL) (P NIL))) caso general**

**;;;**

**;;;(encuentra-modelos '((=> (^ P I) L) (=> (¬ P) (¬ L)) (¬ P) L)) -> NIL caso general**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**COMENTARIOS**

La función auxiliar es necesaria para pasar la lista de interpretaciones, ya que el prototipo de función que nos dan no tiene una segunda lista. Es la función auxiliar la que realiza el trabajo de interpretación, la principal solo genera la lista de posibles valores para los átomos.

**4.6. Encontrar modelos**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**PSEUDOCÓDIGO**

Función: consecuencia-p

Entrada: prop: proposición de la que se quiere saber si es consecuencia lógica

Kb: base de conocimiento

Procesamiento:

Si: lista-vacia(kb)

Devolver NIL

En otro caso: Modelos = encuentra-modelos(kb)

Para cada m en modelos:

AND evaluar-proposicion(m, prop)

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**CÓDIGO**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION CONSECUENCIA-P**

**;;;comprueba si la proposicion pasada como argumento**

**;;;es consecuencia logica de la kb**

**(defun consecuencia-p (prop kb)**

**(when (base-p kb)**

**(mapcan #'(lambda(x) (interpretacion-modelo-p x (list prop)))**

**(encuentra-modelos kb))))**

**;;;EJEMPLOS**

**;;;(consecuencia-p 'A '()) -> NIL caso particular**

**;;;**

**;;;(consecuencia-p '() '((=> A (¬ H)) (<=> P (^ A H)) (=> H P))) -> NIL caso particular**

**;;;**

**;;;(consecuencia-p 'A '(A)) -> T caso general**

**;;;**

**;;;(consecuencia-p 'A '(¬ A)) -> NIL caso general**

**;;;**

**;;;(consecuencia-p '(¬ H) '((=> A (¬ H)) (<=> P (^ A H)) (=> H P))) -> T caso general**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

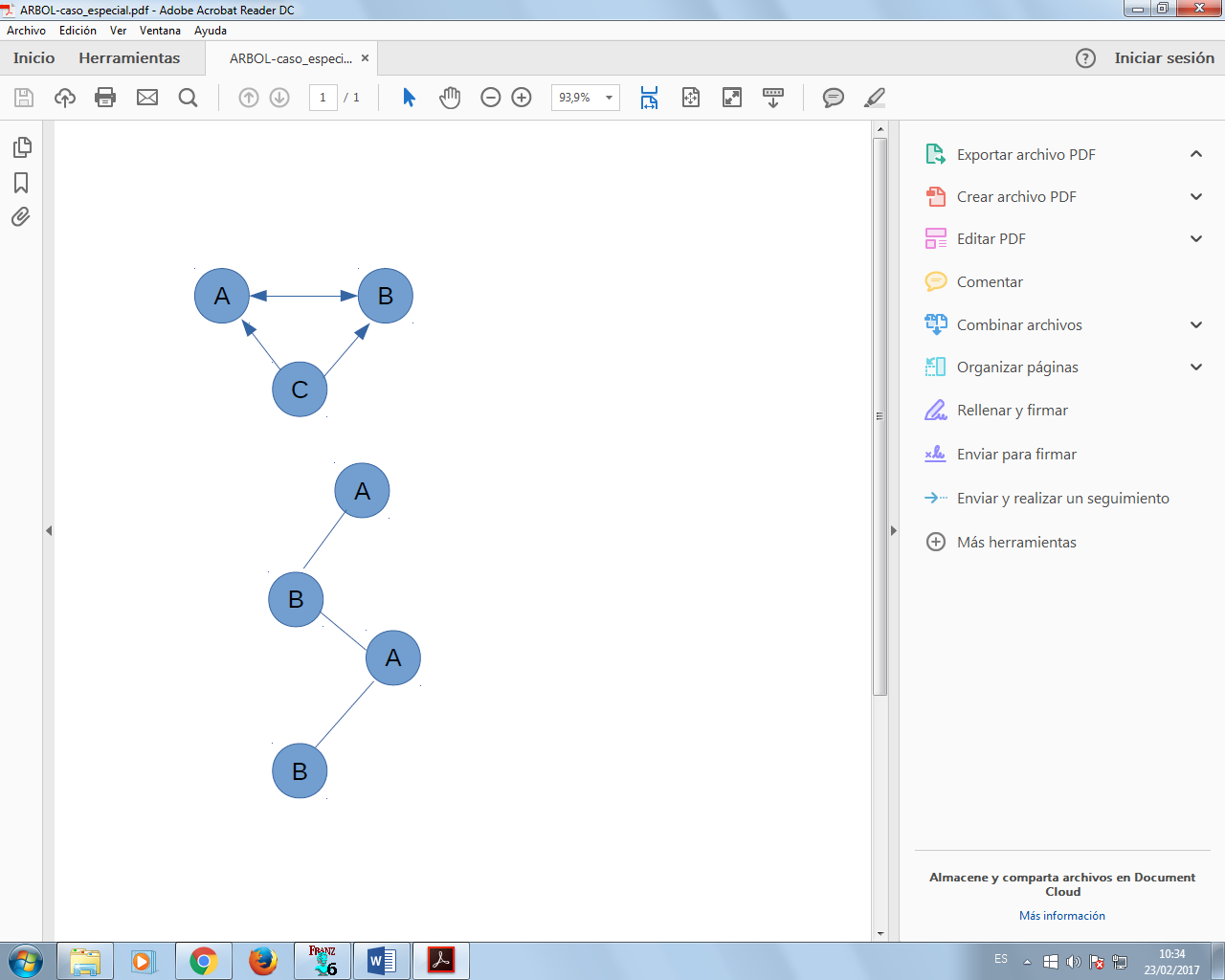
**COMENTARIOS**

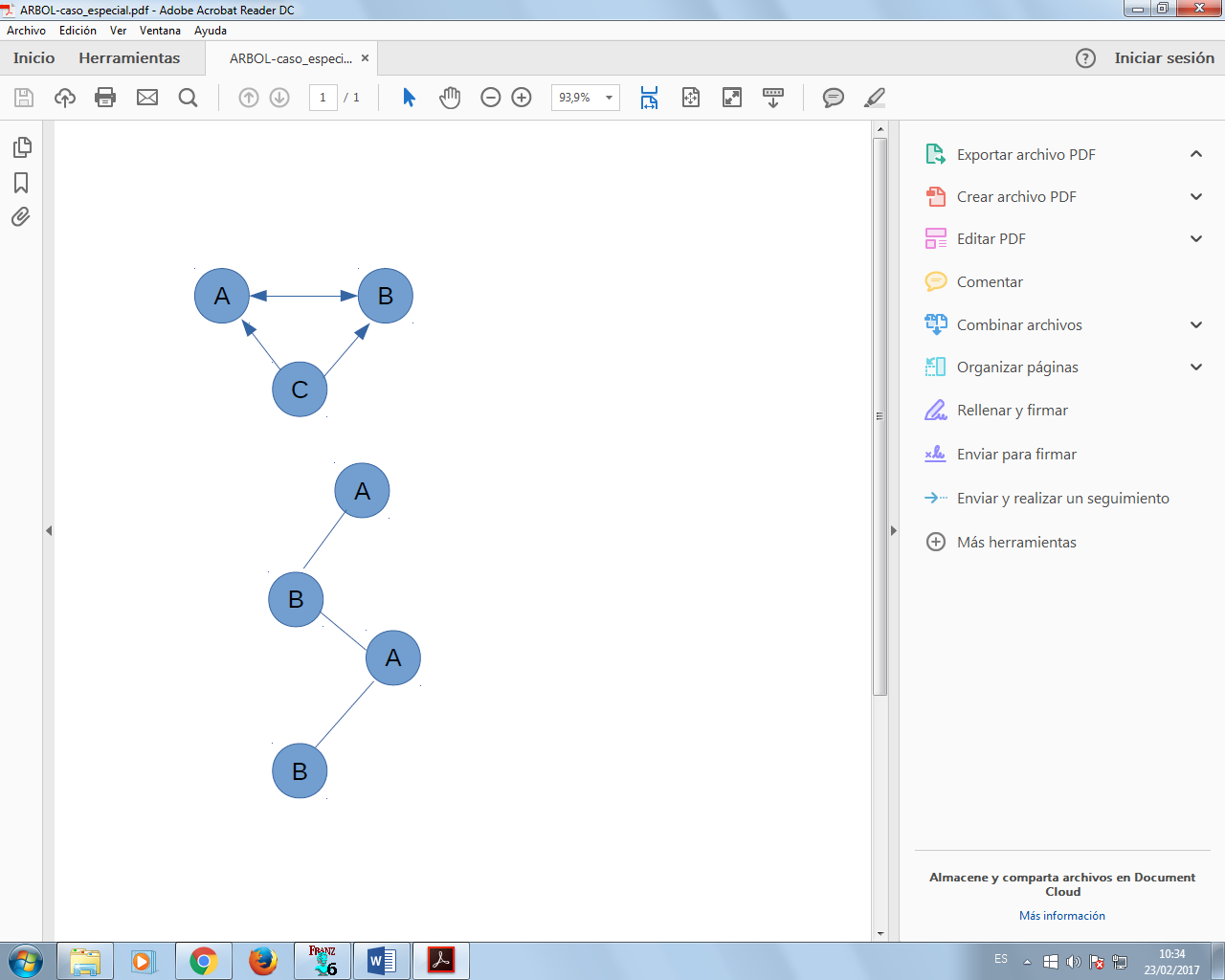
Es necesario que la proposición sea verdadera para todos los modelos de la base, por eso usamos un mapcan. Se podría haber hecho con recursión parando en el primer momento que encontremos un NIL y sería más eficiente, pero complica la programación.

1. **Búsqueda en anchura**

**5.1. Ejemplos de grafos y su árbol de búsqueda en anchura**

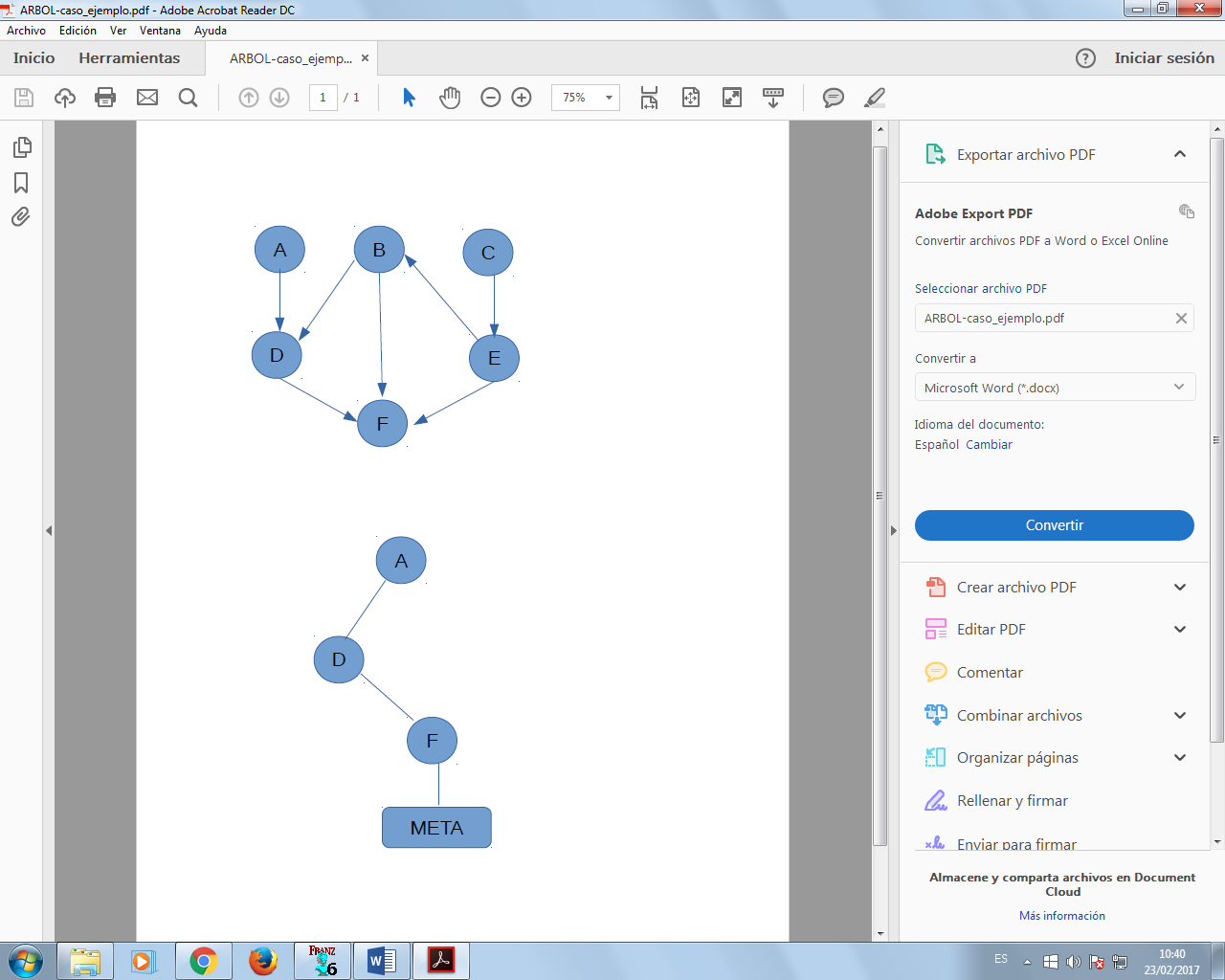
* Caso particular: grafo que no converge

En este grafo queremos llegar de A a C, lo cual es imposible, además hay un bucle entre A y B. Si este bucle no estuviera el algoritmo pararía en B y devolvería FALSO. Al haber un bucle, se da una recursión infinita.



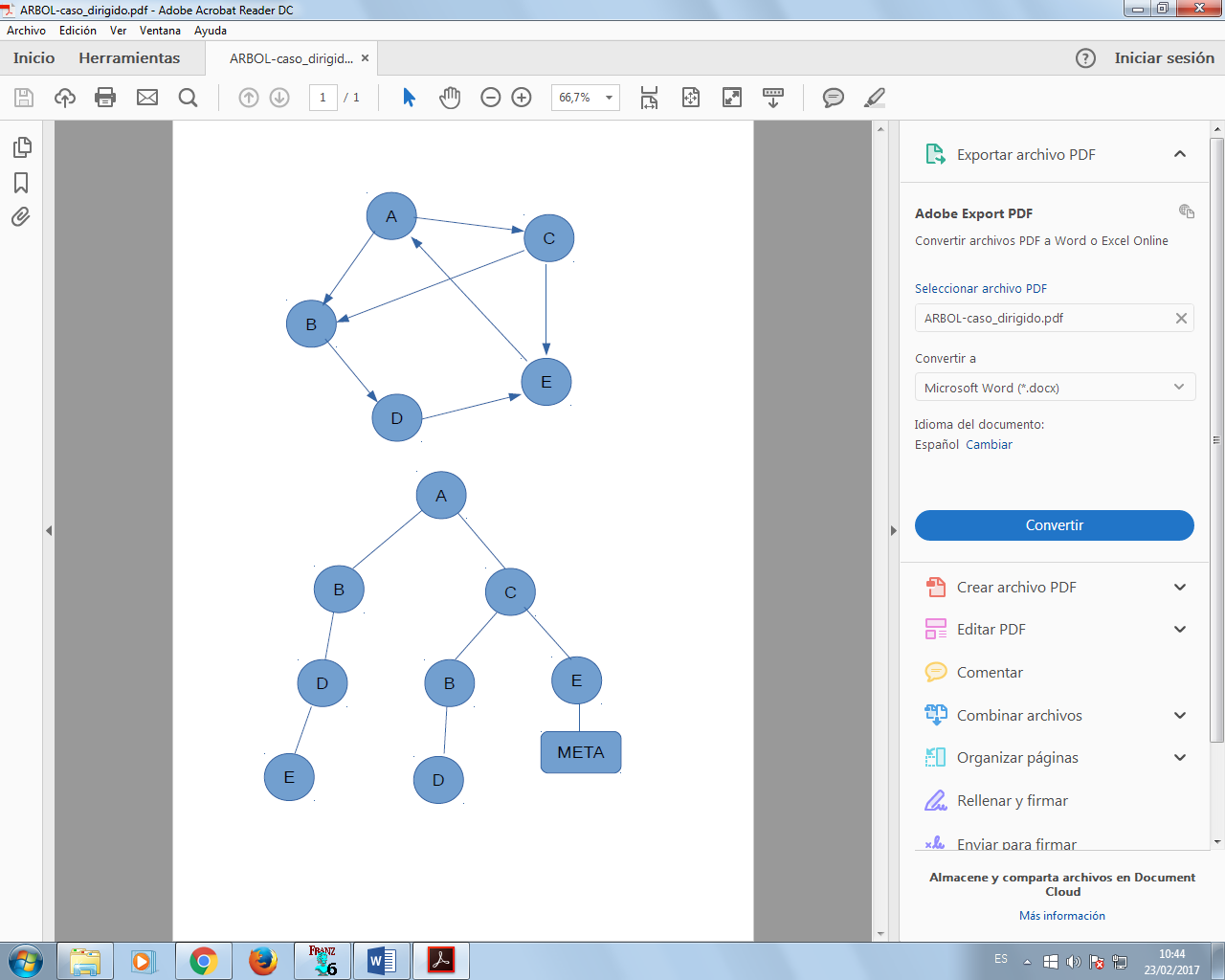
El árbol sigue este patrón infinitamente.

* Caso general: ejemplo propuesto



Queremos ir de A a F. En este grafo podría darse el caso del anterior, ya que hay nodos que no son accesibles desde otros nodos del grafo. No obstante cuando vamos de A a F el algoritmo converge, además sin repetición de nodos.

El árbol resultante.

* Caso general: otro ejemplo

Tenemos el grafo dirigido en el que queremos llegar de A a E.

El árbol resultante converge, pero hay repetición de nodos.

**5.2. Algoritmo de BFS**

Función: bfs

Entrada: test-objetivo, indica cuál es el nodo meta

Lista-abiertos, cola FIFO en la que se guardan los nodos abiertos

Grafo

Retorno: FALSO en caso de que no haya solución

Camino desde el nodo inicial hasta la meta

Procesamiento:

Si: lista-abieros es vacía

Devolver: FALSO

Si: nodo satisface test-objetivo

Devolver: dar-vuelta(camino)

En otro caso:

Eliminar(nodo, lista-abiertos)

Añadir(hijos(nodo), lista-abiertos)

**5.3. Implementación del algoritmo en Lisp**

**COMENTARIOS:**

Hemos cambiado el doble let del algoritmo por un let\*.

**5.4. Algoritmo de BFS comentado**

**;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;**

**;;;FUNCION BUSQUEDA EN ANCHURA COMENTADA**

**;;;Breadth-first-search in graphs**

**(defun bfs (end queue net)**

**(if (null queue) nil ; queue es la lista de abiertos**

**(let\* ((path (car queue))(node (car path)))**

**(if (eql node end) (reverse path) ; Si el nodo satisface el**

**; test objetivo devolvemos**

**; el camino hasta el**

**(bfs end (append**

**(cdr queue) ; eliminamos el nodo de la lista abierta**

**(new-paths path node net)) ;añadimos los nodos**

**;resultantes de la expasion**

**net)))))**

**(defun new-paths (path node net) ;expansion del nodo**

**(mapcar #'(lambda(n)**

**(cons n path))**

**(cdr (assoc node net)))) ;conseguimos los hijos**

**5.5. Camino más corto entre dos nodos**

El algoritmo resuelve el problema del camino más corto entre dos nodos ya que búsqueda en anchura es completo, esto es, si el camino entre los dos nodos dados existe, BFS lo encuentra.

**5.6. Ejecución del algoritmo de BFS**

Ejecutamos el comando trace con bfs. Tras llamar a la función shortest-path obtenemos la siguiente salida:

**0[1]: (BFS F ((A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))**

La función shortes-path llama a bfs poniendo F en el test-objetivo e introduciendo A en la lista de abiertos. Se procede a expandir a, generando su único sucesor: D

**1[1]: (BFS F ((D A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))**

Segunda llamada recursiva. Se introduce el nodo D en la lista de abiertos y se procede a su expansión. En teoría hemos visto que una vez expandido A este se debería sacar de la lista, pero el algoritmo dado no lo hace ya que devuelve el camino y para eso mantiene los nodos en la lista y posteriormente le da la vuelta.

**2[1]: (BFS F ((F D A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))**

Tercera llamada recursiva se introduce F en la lista de abiertos y al ir a expandirlo se satisface el test objetivo y el algoritmo acaba.

**2[1]: returned (A D F)**

**1[1]: returned (A D F)**

**0[1]: returned (A D F)**

**(A D F)**

Se cierran las llamadas recursivas y se va devolviendo el camino hasta el nodo.

**5.7.**

Debemos realizar la siguiente llamada. El resultado obtenido es F B A C.

**CG-USER(59): (setf net '((a d c b e) (b f d e a) (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)))**

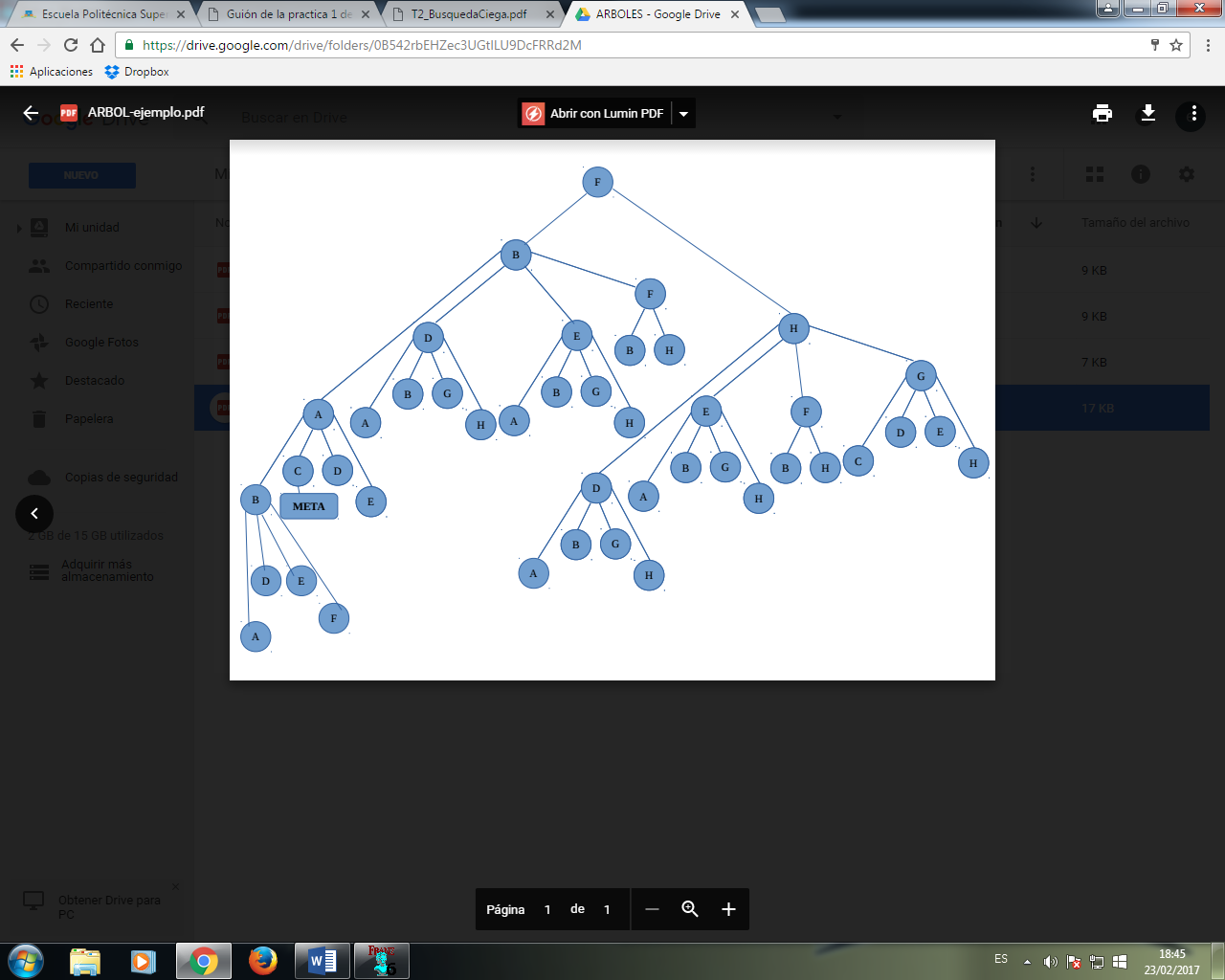
**((A D C B E) (B F D E A) (C A G) (D A B G H) (E A B G H) (F B H)**

**(G C D E H) (H D E F G))**

**CG-USER(60): (shortest-path 'f 'c net)**

**(F B A C)**

Hemos realizado el árbol resultante:



Podemos ver que al ser un grafo en que todos los enlaces son dobles, se generan muchos estados repetidos

**5.8. Mejora del algoritmo**

El ejemplo propuesto es el del apartado 5.1, donde pusimos el grafo dirigido de tres nodos en el que la meta era inalcanzable desde el inicio. Como ilustramos en el árbol se van expandiendo los nodos A y B infinitamente, o en el caso de una máquina, hasta que se desborda la pila de llamadas.

Nuestra propuesta para evitar este problema es: una vez expandido el nodo, borramos tanto su lista de adyacencia, como su presencia en las listas de adyacencia de los demás nodos, de tal manera que nunca volveremos a expandir ese nodo.