Javier Delgado del Cerro y Javier López Cano - Pareja 2

Práctica 1

Ejercicio 1

Apartado 1.1

Batería de ejemplos

Comentar antes de empezar este apartado que no incluiremos en ninguna de estas baterías de ejemplos las expresiones pedidas en la memoria, que aparecen explicadas en *Comentarios sobre la impementación*, para evitar repetir información. Además, esta batería de ejemplos está siempre en el mismo orden que la ejecución del programa, y seguida en dicha ejecución por los ejempos pedidos, de forma que sea muy sencillo comprobar su verosimilitud.

En este caso específico, podemos ver que casi todos los casos conflictivos han sido pedidos en el enunciado, con lo que quedaría simplemente comprobar el simétrico de `(cosine-distance nil '(1 2 3))` y comprobar con algunos ejemplos que, efectivamente, se calcula correctamente. Como el resultado coincide en la versión recursiva y en la mapcar, lo expresamos como una única fución:

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

Como esperábamos (y como se puede ver al ejecutar el archivo), todos los resultados son correctos a excepción de la tercera prueba, donde en vez de obtener 0 obtenemos $5.96 * 10^{-8}$, que se debe únicamente a las aproximaciones del procesador (probablemente sobre todo en las raices cuadradas).

Pseudo-código

La única diferencia en el pseudocódigo de las dos funciones pedidas es la forma de calcular el producto escalar, por lo que especificaremos una única función `cosine-distance` que emplee `dot-product`, y dicho producto escalar será uno de los dos explicados.

```
cosine-distance(x, y):
    denominador = raiz(dot-product(x, x))*raiz(dot-product(y, y))
    si denominador == 0:
        devolver null
    si no:
        devolver dot-product(x, y)/denominador

dot-product-rec(x, y):
    si (x vacio) o (y vacio):
        devolver 0
    devolver (primer elem x)*(primer elem y)+dot-product-rec(resto de x, resto de y)

dot-product-mapcar(x, y):
```

```
    sumar(multiplicar el elemento xi por el yi)
```

Comentarios sobre la implementación

Para desarrollar las funciones `cosine-distance-rec` y `cosine-distance-mapcar` creamos dos funciones que nos permiten calcular el producto escalar de dos vectores de forma recursiva y usando mapcar, estas funciones son `dot-product-rec` y `dot-product-mapcar` respectivamente. Además, para evitar repeticiones de código creamos una tercera función `cosine-distance` que recibe como parámetro la función distancia a utilizar, con lo que `cosine-distance-rec` y `cosine-distance-mapcar` son simplemente llamadas a `cosine-distance`. El resultado de la evaluación de los casos indicados sería:

Expresión	Evaluación recurrente	Evaluación mapcar
`(cosine-distance '(1 2) '(1 2 3))`	0.40238577	0.40238577
`(cosine-distance nil '(1 2 3))`	NIL	NIL
`(cosine-distance '() '())`	0	0
`(cosine-distance '(0 0) '(0 0))`	NIL	NIL

Apartado 1.2

Batería de ejemplos

En este caso, la batería de ejemplos que hemos pensado (sin tener en cuenta los pedidos posteriormente, ni los dados en el enunciado como ejemplo, que también funcionan correctamente) serían:

```
(order-vectors-cosine-distance '(1 2 3) '((0 0 0))) ;; --> NIL
(order-vectors-cosine-distance '(0 0 0) '((1 2 3))) ;; --> NIL
(order-vectors-cosine-distance '(1 2 3) '((1 2 3) (2 1 1)) 0.99) ;; --> ((1 2 3))
(order-vectors-cosine-distance '(1 2 3) '((1 2 3) (2 1 1)) 1) ;; --> ((1 2 3))
(order-vectors-cosine-distance '(1 0) '((1 0) (0 1) (1 0.1) (1 0.2))) ;; --> ((1 0) (1
0.1) (1 0.2) (0 1))
(order-vectors-cosine-distance nil '((1 2 3) (2 1 1))) ;; --> NIL
(order-vectors-cosine-distance '(1 2 3) nil) ;; --> NIL
(order-vectors-cosine-distance nil nil) ;; --> NIL
```

Al igual que antes, podemos ver en el tercer caso como, por errores de aproximación, el nivel de confianza 1 no es útil, pues el resultado es NIL en vez de el esperado, al ser la distancia $5.96 * 10^{-8}$ en vez de cero.

Pseudo-código

```
order-vectors-cosine-distance (vector lst-of-vectors confidence-level):
    para cada v en lst-of-vectors:
        resultado.añadir(v, cosine-distance(vector, v))
    resultado.ordenar()
    delvolver resultado.primer-elemento-tuplas()
```

Comentarios sobre la implementación

Para codificar dicha función, es claro que necesitamos obtener la distancia coseno de cada uno de los vectores de `lst-of-vectors` a `vector`, eliminar aquellos cuya distancia sea mayor que `1 - confidence-level` y finalmente, ordenarlos respecto a dicho parámetro de menor a mayor. Para conseguir esto, desarrollamos la funcion `map-vectors-cosine-distance` que se encarga de crear una lista de tuplas (cada una de ellas con un vector y su distancia coseno respecto al vector de referencia) con los vectores cuya distancia coseno es menor o igual que que `1 - confidence-level`. Posteriormente, basta con ordenar esta lista (para lo que usamos `vector-order` y una copia de la lista, pues `sort` es una función destructiva) y coger únicamente el primer elemento de cada tupla. Como alternativa, podríamos haber programado nosotros mismos un algoritmo de ordenación como quicksort y emplear dicha función directamente, pero creemos que la implementación oficial probablemente esté mucho más optimizada que la que podemos desarrollar nosotros.

Al ejecutar los ejemplos dados en el PDF obtenemos exactamente los mismos resultados. Para los casos de prueba pedidos, obtenemos lo siguiente:

Expresión	Evaluación
`(order-vectors-cosine-distance '(1 2 3) '())`	NIL
`(order-vectors-cosine-distance '() '((4 3 2) (1 2 3)))`	NIL

Como era de esperar.

Apartado 1.3

Batería de ejemplos

De nuevo, evitamos probar las expresiones que se piden posteriormente y están explicadas en la tabla del apartado 1.4. Además, usamos la expresión `cosine-distance` por ser el resultado independiente del uso de la función recursiva o la función con mapcar. La batería de ejemplos sería entonces:

```
(get-vectors-category '((1 1 2) (2 2 1)) '((1 2 1) (2 1 2)) #'cosine-distance) ;;; -->
((2 0.0) (1 0.0))
(get-vectors-category '((1 1 2)) '((1 2 1) (2 1 2)) #'cosine-distance) ;;; --> ((1
0.19999999) (1 0.0))
(get-vectors-category '((1 0 0)) '((1 2 1) (2 1 2)) #'cosine-distance) ;;; --; (NIL NIL)
(get-vectors-category '((1 1 2) (2 2 1)) '((1 0 0)) #'cosine-distance) ;;; --; (NIL)
(print (get-vectors-category nil '((1 2 1) (2 1 2)) #'cosine-distance) ;;; --> NIL
(get-vectors-category '((1 2 1) (2 1 2)) nil #'cosine-distance) ;;; --> NIL
(get-vectors-category nil nil #'cosine-distance) ;;; --> NIL
(get-vectors-category '((1 1 2 3) ()) '((1 1 2 3) (2 2 4 6)) #'cosine-distance) ;;; -->
((1 0) (1 0.02536))
(get-vectors-category '((1 4 5 6) (2 2 1 3)) '(() (2 4 5 6)) #'cosine-distance) ;;; -->
(NIL (1 0.0))
(get-vectors-category '(() (1 4 5 6)) '(() (2 4 5 6)) #'cosine-distance) ;;; --> ((NIL
0) (1 0.0))
```

Consideramos que en el caso 3 y 4 tiene que devolver listas de NIL, pues aunque ambas listas con válidas (las te categorías y las de textos), ninguno de los textos tiene una categoría que se le pueda asociar.

Sin embargo, en los casos 5, 6 y 7, como o solo una o ninguna de las listas son válidas, devolvemos nil directamente.

En el caso 8, al igual que ha pasado anteriormente, cabe destacar que no obtenemos exactamente dicho resultado, pues dadas las aproximaciones del ordenador el par `(1 0)` se convierte en `(1 5.9604645E-8)`.

Pseudo-código

```
get-vectors-category (categories texts distance-measure):
    para cada text en texts:
        lista.crear()
        para cada cat en categories:
            lista.añadir(id(cat) distance-measure(vector(text), vector(text)))
        resultado.añadir(lista.coger_minima_distancia())
    devolver resultado
```

Comentarios sobre la implementación

En este tercer apartado, empleamos dos funciones auxiliares para llegar a la pedida. En primer lugar `get-text-category-dist` nos permite, dado un texto y una categoría, obtener una lista con una tupla con el id de la categoria y la distancia del texto a la categoría, y en segundo lugar `get-text-category` nos permite obtener la categoría de un texto. Hacemos que `get-text-category-dist` nos devuelva una lista con una tupla para, de esta forma, poder usar mapcan en `get-text-category`, y controlar los casos en los que alguno de los vectores sea, quitando el id, todo ceros.

De esta forma en `get-vectors-category` simplemente tenemos que llamar a `get-text-category` para cada uno de los textos.

Hemos decidido además que, en el caso de que alguna de las listas pasadas no tenga elementos, la función devuelva nil, pues en ese caso será imposible que la función haga su cometido.

Apartado 1.4

Usando la macro time con algunos ejemplos para medir la diferencia de rendimiento entre ambas funciones, podemos ver que la función recursiva es mucho menos eficiente, como era de esperar, pues sabemos que mapcar paraleliza la ejecución en distintos hilos. Así, por ejemplo, para dos vectores de cuatro elementos como textos y categorías, la función recursiva tarda $8.1 * 10^{-5} s$ mientras que la función mapcar tarda $3.8 * 10^{-5} s$. Si aumentamos los vectores para que sean de cinco elementos, y usamos cuatro de estos vectores para cada una de las listas, vemos que los tiempos en este caso serían $1.91 * 10^{-4} s$ y $7.6 * 10^{-5} s$ respectivamente, por lo que es claro además que el tiempo de la función recursiva crece mucho más rápido que el de la función mapcar.

Además, para los ejemplos planteados obtenemos los siguientes resultados:

Expresión	Evaluación
`(get-vectors-category '(()) '(()) #'cosine-distance)`	((NIL 0))
`(get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance)`	`((2 0.40238577))`
`(get-vectors-category '(()) '((1 1 2 3) (2 4 5 6)) #'cosine-distance)`	(NIL NIL)

Como hemos mencionado en la parte de *batería de ejemplos* del apartado anterior, en el primer y tercer caso, devuelve una lista con NILs porque, para los textos (que pueden o no ser NIL) no hay ninguna categoría.

Ejercicio 2

Apartado 2.1

Batería de ejemplos

En este apartado, los ejemplos que se nos proporcionan en el enunciado para probar el funcionamiento son bastante completos para los casos en los que no se introduce tolerancia, para estos solo faltaría el caso en el que se introduce directamente la raíz de la función. Por tanto en esta batería incluimos este caso y todos los que introducen una tolerancia:

```
(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4))
11)) 20 4.0) ;;---> 4.0
(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4))
11)) 20 0.6 0.001) ;;---> 1.0
(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4))
11)) 30 -2.5 0) ;;---> -3.0
(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4))
11)) 10 100.0 0.005) ;;---> NIL
(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4))
11)) 20 4.0 0.007) ;;---> 4.0
```

En el primer caso la función devuelve 4.0 de forma exacta en la primera iteración ya que directamente se ha introducido como semilla la raíz, lo mismo ocurre en el último caso (igual que el primero pero introduciendo una tolerancia).

En el caso tres la función devuelva un valor muy cercano a 1, 0.99999946, en lugar de exactamente 1 pues $1 - 0.99999946 < 0.001$ que es la tolerancia introducida y por tanto el algoritmo es correcto. En el cuarto caso el resultado debe ser exacto pues la tolerancia introducida es 0.

El penúltimo caso devuelve NIL como se espera ya que no encuentra una raíz que cumpla con la tolerancia pedida en las diez iteraciones que se le permite hacer.

Pseudocódigo

```
newton (funct, deriv, iterations, x0, tol):
    si iterations <= 0:
        devolver null
    si no:
        si valor-absoluto(funct(x0)/deriv(x0)) <= tol:
            devolver x0
        si no:
            devolver newton (funct, deriv, iterations-1, x0-(funct(x0)/deriv(x0), tol)
```

Comentarios sobre la implementación

En este caso, debido a la simplicidad de la función pedida, hemos decidido no emplear funciones auxiliares en su implementación.

Hemos empleado un `let` para guardar el cociente `f(x0)/df(x0)` y así evitar repetir el cáculo varias veces en la función.

Por último, mediante el uso de varios `if` controlamos que el algoritmo (que es recursivo) no realice mas iteraciones de las permitidas (reduciendo en 1 el campo `max-iter` en cada iteración y comprobando que no es cero) , y si el valor introducido como semilla en la iteración (`x0`) cumple la tolerancia (y por tanto es la raíz que buscamos) o no y por tanto se debe realizar una iteración más.

Los resultados para los ejemplos que se nos proponen en el enunciado son los siguientes:

Expresión	Evaluación
`(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 3.0)`	4.000084
`(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 0.6)`	0.99999946
`(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 30 -2.5)`	-3.0000203
`(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 10 100.0)`	NIL

Que, como se puede ver, son los resultados esperados teniendo en cuenta que no son exactos pero satisfacen la tolerancia.

Apartado 2.2

Batería de ejemplos

De nuevo en este caso no repetimos los casos que ya se comprueban con los ejemplos proporcionados en el enunciado para probar el funcionamiento de la función pedida.

Sabiendo esto, la batería de ejemplos propuesta es:

```
(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x
3) 4)) 11)) 7 '(50.0 0.6)) ;;---> 0.99999946
(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x
3) 4)) 11)) 7 '(3.0 50.0)) ;;---> 4.000084
(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x
3) 4)) 11)) 1 '(3.0 -2.5)) ;;---> NIL
(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x
3) 4)) 11)) 20 '(0.6 3.0 -2.5) 0) ;;---> 1.0
```

Podemos observar que en todos los ejemplos salvo el último es aceptable obtener resultados no exactos siempre que cumplan la tolerancia $0,001$ pues esta es la que se establece por defecto si no se introduce ninguna. Por el contrario en el último ejemplo el resultado debe ser exacto pues se introduce una tolerancia exacta. En este último ejemplo podemos ver que al introducir una tolerancia diferente a la que se usa por defecto la función sigue ejecutándose de forma correcta.

En el primer ejemplo vemos que si el algoritmo de newton devuelve NIL con la primera semilla, esta función continua de forma correcta con la semilla siguiente. En el segundo caso comprobamos que si la primera semilla hace que newton devuelva un valor para la raíz el algoritmo termina y devuelve ese valor, y en el tercero vemos que si no encuentra una raíz para ninguna de las semillas en las iteraciones permitidas devuelve NIL.

Pseudocódigo

```
one-root-newton (funct, deriv, iterations, semillas, tol):
    si newton(funct, derivate, iterations, first(semillas), tol) == null:
        si rest(Semillas) == null:
            devolver null
        si no:
            devolver one-root-newton (funct, deriv, iterations, rest(semillas), tol)
    si no:
        devolver newton(funct, derivate, iterations, first(semillas), tol)
```

Comentarios sobre la implementación

De nuevo, en este caso tampoco hemos utilizado funciones auxiliares pero si hemos utilizado la función `let`, en concreto para guardar en una variable local el valor de la llamada a la función `(newton f df max-iter (first semillas) tol)` y así evitar repetición de código.

En este algoritmo hemos decidido emplear varios `if` para controlar cuando debe pararse la ejecución recursiva del algoritmo (cuando la ejecución de newton para alguna semilla produce un resultado distinto de `nil` o cuando `(rest semillas)` es una lista vacía) y para determinar si el algoritmo debe devolver `nil` o el valor que ha devuelto la función newton.

Como se puede observar tanto en el pseudocódigo como en el código, para la implementación de esta función hemos empleado la función newton creada en el apartado anterior.

Al ejecutar las sentencias propuestas en el enunciado obtenemos los siguientes resultados:

Expresión	Evaluación
`(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 -2.5))`	0.99999946
`(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(3.0 -2.5))`	4.000084
`(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(3.0 -2.5))`	NIL

Que, de nuevo, a pesar de no ser exactos, son correctos pues satisfacen la tolerancia.

Apartado 2.3

Batería de ejemplos

En este apartado los ejemplos que se nos proporcionan para probar el funcionamiento del algoritmo pedido ya incluyen todas las posibles variaciones que merece la pena probar (con la tolerancia establecida por defecto), por lo tanto, en esta batería de ejemplos propuesta tan solo añadimos un ejemplo sin introducir tolerancia para probar el correcto funcionamiento del algoritmo, y uno introduciendo tolerancia para ver que también funciona correctamente si se decide introducir una tolerancia específica.

De este modo obtenemos la siguiente batería de ejemplos:

```
(all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(-2.5 3.0 10000.0)) ;;---> (-3.0000203 4.0 nil)
(all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(-2.5 3.0 10000.0) 0) ;;---> (-3.0 4.0 nil)
```

Al ejecutar estos ejemplos observamos que ambos son correctos, y tan solo cabe destacar que, al igual que en los casos anteriores, el caso con tolerancia "0" ofrece resultados exactos mientras que el otro no, pero esto es correcto ya que a pesar de no ser exactos cumplen con la tolerancia.

Pseudocódigo

```
all-roots-newton (funct, deriv, iterations, semillas, tol):
    si rest(semillas) == null:
        devolver lista.crear(newton(funct, deriv, iterations, first(semilla), tol))
    si no:
        devolver lista.añadir(newton(funct, deriv, iterations, first(semilla), tol),
all-roots-newton(funct, deriv, iterations, rest(semillas), tol))
```

Comentarios sobre la implementación

A la hora de implementar el funcionamiento pedido en este apartado, hemos empleado al igual que en el apartado anterior, llamadas a la función newton creada en el apartado 2.1 y hemos almacenado el valor que esta devuelve en una variable local mediante la función `let` para evitar repetir código.

Esta función emplea un if para ver si se trata o no de la ultima iteración del algoritmo recursivo (comprobando si `(rest semillas)` es una lista vacía o no), si lo es, devuelve el valor obtenido de la función newton para esa semilla, y si no lo es, añade el valor obtenido de newton a la lista que se obtiene al llamar de nuevo a la función `all-roots-newton` con `(rest semillas)` (es decir, la lista con los valores de la función newton para el resto de semillas obtenido en las iteraciones siguientes).

De este modo obtenemos a la salida una lista con las raíces que halla newton para las semillas pasadas como argumento (o NIL en caso de que newton no converja para alguna de ellas).

Al ejecutar las expresiones propuestas en el enunciado obtenemos los siguientes resultados:

Expresión	Evaluación
`(all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 -2.5))`	(0.99999946 4.000084 -3.0000203)
`(all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 10000.0))`	(0.99999946 4.000084 NIL)
`(all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(0.6 3.0 -2.5))`	(NIL NIL NIL)

En estos ejemplos se puede ver que el algoritmo es correcto en los casos en que ninguna semilla de la lista hace que el algoritmo de newton converja, en el caso en que todas las semillas hacen que converja, y en el caso en el que algunas hacen que converja y otras no. Es por esto por lo que, como hemos dicho en el apartado de la batería de ejemplos, estos casos no se han incluido ahí.

Podemos observar que los resultados son correctos pues cuando el algoritmo no converge introduce NIL en la lista, y cuando converge los resultados satisfacen la tolerancia establecida por defecto que es de 0,001.

Apartado 2.3.1

Batería de ejemplos

Para probar este apartado concreto se ha decidido emplear la siguiente batería de ejemplos:

```
(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 -2.5)) ;;---> (0.99999946 4.000084 -3.0000203)
(list-not-nil-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 10000.0)) ;;---> (0.99999946 4.000084)
(list-not-nil-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(0.6 3.0 -2.5)) ;;---> NIL
(list-not-nil-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 10000.0) 0) ;;---> (1.0 4.0)
```

En el primer ejemplo propuesto comprobamos que al emplear una lista de semillas en las que todas permiten que el algoritmo newton converja en el número de iteraciones permitido la función no elimina ninguno de los resultados de la lista.

En el segundo empleamos una lista de semillas en las que hay algunas que hacen que el algoritmo converja y otras no y observamos que, como esperábamos, el algoritmo elimina los NIL de la lista resultado.

En el tercer ejemplo vemos que si introducimos una lista de semillas en las que ninguna permite que el algoritmo converja, el resultado de la función es NIL, es decir, la lista vacía, que es correcto.

Por último comprobamos que al introducir una tolerancia diferente a la que se establece por defecto el algoritmo también funciona correctamente. Cabe mencionar que como esperábamos, en este caso con tolerancia 0 los resultados son exactos, mientras que en el resto no es necesario que lo sean siempre que cumplan la tolerancia 0,001 que se establece por defecto.

Pseudocódigo

```
list-not-nil-roots-newton (funct, deriv, iterations, semillas, tol):
    para cada elemento de all-roots-newton(funct, deriv, iterations, semillas, tol):
        si es null:
            no.añadir.elemento.lista()
        si no:
            añadir.elemento.lista()
    devolver lista
```

Comentarios sobre la implementación

En este apartado no hemos empleado recursividad para implementar el funcionamiento pedido, sino que hemos empleado la función `mapcan` para modificar todos los elementos de la lista, como se nos exigía en el enunciado.

Para la implementación del algoritmo hemos empleado llamadas a la función `all-roots-newton` creada en el apartado anterior, y, aplicando una función lambda sobre cada uno de los elementos de la lista que esta función devuelve (usando `mapcan`) eliminamos los posibles NIL de la lista, devolviendo dicha lista modificada.

Esta función lambda empleada sobre la lista obtenida de `all-roots-newton` emplea un `if` para comprobar si el elemento de la lista es NIL, si lo es, lo quita (sustituyéndolo por `'()`), y si no, lo deja como está `(list x)`.

Al probar la expresión que se nos propone en el enunciado obtenemos:

Expresión	Evaluación
`(list-not-nil-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 10000.0))`	(0.999999464.000084)

Que es, de nuevo, el resultado que esperábamos obtener.

Ejercicio 3

Apartado 3.1

Batería de ejemplos

Dada la simplicitud del apartado, además de los ejemplos proporcionados en el enunciado y evaluados en el apartado de *comentarios sobre la implementación*, solo falta comprobar su correcto funcionamiento mediante la expresión:

```
(combine-elt-lst 'a '(1 2 3)) ;;; ---> ((A 1) (A 2) (A 3))
```

Pseudo-código

```
combine-elt-lst (elt, lst):
    para cada elemento en lst:
        resultado.añadir(lista(elt lst))
    devolver resultado
```

Comentarios sobre la implementación

En este caso, la implementación ha sido muy sencilla y lo único a destacar es la decisión que hemos tomado de, si o bien el `elt` o bien `lst` son vacíos o nil, devolver nil directamente. Por tanto, la evaluación de las expresiones pedidas es la siguiente:

Expresión	Evaluación
`(combine-elt-lst 'a nil)`	NIL
`(combine-elt-lst nil nil)`	NIL
`(combine-elt-lst nil '(a b))`	NIL

Apartado 3.2

Batería de ejemplos

Al igual que en el ejemplo anterior, todas las expresiones que pueden causar algún tipo de conflicto están evaluadas en el apartado de *comentarios sobre la implementación*, con lo que solo es necesario probar su correcto funcionamiento, para lo que es suficiente la expresión:

```
(combine-lst-lst '(a b c) '(1 2)) ;;; --> ((A 1) (A 2) (B 1) (B 2) (C 1) (C 2))
```

Pseudo-código

Una vez desarrollada la función del aparado anterior, esta es bastante sencilla

```
combine-lst-lst (lst1, lst2):
    para cada elt en lst1:
        resultado.concatenar(combine-elt-lst(elt lst))
    devolver resultado
```

Comentarios sobre la implementación

Como `combine-elt-lst` devuelve una lista, usamos mapcan para aplicar esta función con cada elemento de `lst1` sobre `lst2`. De esta forma, mapcan concatena las listas. Además, en el caso de ser alguno de los campos vacío o NIL, mapcan devuelve simplemente nil, pues no añade nada la lista. Por tanto, la evaluación de las expresiones pedidas es la siguiente:

Expresión	Evaluación
`(combine-lst-lst nil nil)`	NIL
`(combine-lst-lst '(a b c) nil)`	NIL
`(combine-lst-lst nil '(a b c))`	NIL

Apartado 3.3

Batería de ejemplos

De nuevo, todas las expresiones que pueden dar lugar a algún tipo de conflicto están evaluadas en el apartado de *comentarios sobre la implementación*, con lo que basta con ver que funciona correctamente para dos listas (su comportamiento tiene que ser exactamente igual que el de `combine-lst-lst`) y para tres o más listas:

```
(combine-list-of-lsts '((a b c) (1 2))) ;;; --> ((A 1) (A 2) (B 1) (B 2) (C 1) (C 2))
(combine-list-of-lsts '((a b c) (+ -) (1 2))) ;;; --> ((A + 1) (A + 2) (A - 1) (A - 2)
(B + 1) (B + 2) (B - 1) (B - 2) (C + 1) (C + 2) (C - 1) (C - 2))
```

Pseudo-código

Una vez desarrolladas las funciones de los dos apartados anteriores, pensamos en usar simplemente una recursión cuyo caso base sea el caso en el que `lstolsts` tiene únicamente un elemento, pues en dicho caso el resultado sería, usando un ejemplo, `(combine-list-of-lsts '((a b c))) --> ((A) (B) (C))`

```
combine-list-of-lsts (lstolsts):
    si elementos(lstolsts) == 0:
        devolver lista_vacia
    devolver combine-lst-lst(primer elem lstolsts, combine-list-of-lsts(resto lstolsts))
```

Comentarios sobre la implementación

Sin embargo, en la implementación nos encontramos con un pequeño problema, y es que la función `combine-elt-lst` crea una lista para cada par, lo que cumple las especificaciones pedidas para el apartado 3.1, sin embargo, al usar dicha función en este tercer apartado, necesitaríamos concatenar el par en otra lista, obteniendo así una única lista con dos elementos, y no una lista con dos sublistas.

> Un ejemplo sería: queremos obtener `(A + 1)`, pero con la función actual obtenemos `(A (+ (1)))`. Como hemos dicho, la solución pasaría por concatenar los elementos (función `cons`) en vez de crear una lista, pero esto no cumpliría el formato de salida pedido en el enunciado para esta función, pues el resultado de evaluar `(combine-elt-lst 'a '(1 2 3))` sería `((A . 1) (A . 2) (A . 3))` en vez de `((A 1) (A 2) (A 3))`.

Además, el usar mapcan en `combine-lst-lst` haría que el tercer apartado fuese tremendamente ineficiente. Por tanto, creamos dos nuevas funciones auxiliares, una `combine-elt-lst-aux` que simplemente implementa esta concatenación, y otra `combine-elt-lst-aux` cuya función es la misma que `combine-elt-lst` pero llamando en este caso a `combine-elt-lst-aux` de forma recursiva.

Una vez programada la función, comprobamos con el ejemplo dado y con nuestra batería de ejemplos que funciona correctamente, y la evaluación sobre las expresiones pedidas sería:

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

Ejercicio 4

Apartado 4.1

Batería de ejemplos

Para probar el correcto funcionamiento del código pedido en este apartado basta con un ejemplo que contenga todos las combinaciones de operadores que se deban simplificar así como negaciones en un ámbito mayor al de un literal negativos. De este modo podremos saber si las 2 funciones principales implementadas para este apartado (`limpiar` y `expand-truth-tree-aux`) realizan el procedimiento requerido de forma adecuada. Además de este ejemplo "completo" se deben probar también el caso especial de una expresión vacía (`NIL`). Damos por supuesto que las expresiones que se pasan a estas funciones están bien formadas y, por tanto, los casos en que no lo estén no se comprueban.

Basándonos en estos requisitos hemos creado la siguiente batería de ejemplos:

```
(limpiar '(! (<=> (! (^ (! a) b (v c d))) (!(v a (! b c))))))
->(V (^ (V A (! B) (^ (! C) (! D))) (V A (! B))) (^ (^ (! A) B) (^ (! A) B (V C D))))

(limpiar NIL) -> NIL

(expand-truth-tree-aux (limpiar '(! (<=> (! (^ (! a) b (v c d))) (!(v a (! b c))))))
NIL)
->((A A) (A (! B)) (A (! D) (! C)) ((! B) A) ((! B) (! B)) ((! B) (! D) (! C)) (C B (!
A) B (! A)) (D B (! A) B (! A)))

(expand-truth-tree-aux NIL) -> NIL
```

No incluimos "=>" en la expresión completa para probar el funcionamiento ya que no es necesario porque estos están incluidos en el bicondicional ("<=>").

Pseudocódigo

Para la realización de este apartado hemos utilizado dos funciones principales, una función `limpiar` que se encarga de eliminar todas las implicaciones y dobles implicaciones usando las leyes de De Morgan y reduce el ámbito de todas las negaciones al mínimo posible. Su pseudocódigo sería el siguiente.

```
limpiar(fbf):
    si fbf es un literal:
        devolver fbf
    si el primer elemento de fbf es un operador binario:
        devolver limpiar(solve-implication(fbf))
    si el primer elemento de fbf es un operador unario (not):
        devolver limpiar(not-connector(second(fbf)))
    si no:
        para cada elemento (x) de rest(fbf):
            fragmento = limpiar(x)
            añadir fragmento a LISTA
        añadir first(fbf a LISTA)
        devolver LISTA
```

La segunda función principal recibe la base de conocimiento tras ser procesada por la función `limpiar`, y se encarga de crear el árbol a partir de ésta entrada. Su pseudocódigo sería:.

```
expand-truth-tree-aux (fbf tree):
    si la fbf es un and:
        devolver tree
    si la fbf es un literal:
        devolver expand-truth-tree-aux(+and+, add-to-lists(fbf, tree))
    si el primer operador es un or:
        para cada elemento (x) de rest(fbf):
            fragmento = expand-truth-tree-aux(x, tree)
            concatenar fragmento a LISTA
        devolver LISTA
    si el primer operador es un and:
        si no hay segundo elemento en fbf:
            devolver tree
        si no:
            devolver expand-truth-tree-aux(cons(+and+, cddr(fbf)),
                expand-truth-tree-aux(second(fbf), tree))
    si no:
        devolver NIL
```

Por último, como se puede observar en este pseudocódigo, estas funciones necesitan de otras funciones auxiliares para funcionar correctamente.

Las funciones auxiliares para `limpiar` tienen el siguiente pseudocódigo:

```
solve-simple-implication(fbf):
    devolver list(+or+, list(+not+, second(fbf), third(fbf)))


solve-double-implication(fbf):
```

```
    devolver list(+and+, solve-simple-implication fbf,
        solve-simple-implication(list(first(fbf), third(fbf), second(fbf)))


solve-implication(fbf):
    si el primer elemento de fbf es =>:
        devolver solve-simple-implication(fbf)
    si no:
        devolver solve-double-implication(fbf)


negate(fbf):
    si fbf solo tiene 1 elemento:
        devolver list(cons(+not+, fbf))
    si no:
        devolver cons(list(not, first(fbf)), negate(rest(fbf)))


not-connector(fbf):
    si fbf es un literal positivo:
        devolver list(+not+, fbf)
    si el primer elemento de fbf es un not:
        devolver second(fbf)
    si el primer elemento de fbf es un operador binario:
        devolver not-connector(solve-implication(fbf))
    si el primer elemento de fbf es un and:
        devolver cons(+or+, negate(first(fbf)))
    si no:
        devolver cons(+and+, negate(rest(fbf)))
```

Y las funciones auxiliares para `expand-truth-tree-aux` tienen el siguiente pseudocódigo:

```
add-to-lists(elem, lsts):
    si lsts está vacía:
        devolver list(elem)
    si no:
        devolver add-rec(elem, lsts)


add-rec(elem, lsts):
    si lsts está vacía:
        devolver NIL
    si el primer elemento de lsts es un literal:
        devolver cons(list(elem, first(lsts)), add-rec(elem, rest(lsts)))
    si no:
        devolver cons(cons(elem, first(lsts)), add-rec(elem, rest(lsts)))
```

Comentarios sobre la implementación

Para implementar el funcionamiento pedido hemos decidido emplear 2 funciones principales, `limpiar`, que emplea reglas de inferencia para eliminar los condicionales y bicondicionales y reduce las negaciones al menor ámbito posible, es decir, a literales negativos; y `expand-truth-tree-aux` que recibe la expresión "limpiada" y genera a partir de ella el árbol de verdad correspondiente.

Para implementar la función `limpiar` hemos creado varias funciones auxiliares.

En primer lugar `solve-simple-implication` transforma una expresión con un condicional simple en su equivalente (¬A v B), del mismo modo `solve-double-implication` llama 2 veces a la función `solve-simple-impication` para resolver los bicondicionales como si fuesen 2 condicionales simples, por último `solve-implication` determina si la expresión es un condicional o un doble condicional y llama a la función correspondiente para resolverlo.

Para resolver las negaciones de modo que se reduzcan al menor ámbito posible hemos creado la función `not-connector` que recibe la expresión interior de un "not" (es decir, la fbf sin el not), y la resuelve de la manera conveniente. Si la expresión es un or, la transforma en la negación de los ands, si es un and, en la negación de los ors y si es otra negación, elimina la doble negación. Para facilitar el caso de los or y and esta función llama a otra `negate` que recibe una lista de expresiones y las niega todas.

Finalmente la función `limpiar` determina si la expresión general (la más amplia) es un or, un and, un condicional o un bicondicional y llama a la función correspondiente para resolverlo, esta función luego se va llamando a sí misma recursivamente para repetir el proceso con las subexpresiones interiores de menor ámbito, obteniendo finalmente la expresión sin condicionales y con negaciones solo en literales negativos.

Para implementar `expand-truth-tree-aux` también hemos empleado una función auxiliar `add-to-lists` a la que se le pasa un elemento y una lista de expresiones y añade el elemento a todas las expresiones de la lista. Si la lista es `NIL` la función devuelve `NIL`, y si no llama a otra función auxiliar `add-rec` que emplea recursión para ir añadiendo el elemento a todas las sub-listas.

Finalmente `expand-truth-tree-aux` recibe la expresión que debe expandir y un "árbol" inicialmente vacío que va rellenando y que al final será el árbol de verdad de la expresión. En primer lugar la función determina si la expresión ya "limpia" es un and, un `or` o un literal. Si es un literal añade este al árbol resultado (con `add-to-lists`) y se llama a si misma con este árbol y con la expresión `^` que es el caso base, si la expresión es `^` se devuelve el árbol que se tenga (`tree`), si es un `or` emplea un `mapcan` para "expandir" (llamándose a si misma) cada una de las sub-expresiones del `or`, y si es un and, emplea recursión para expandir los subelementos.

El caso del and, que es el más complicado, es el que necesita explicarse con más detalle. En caso de que la expresión sea un and, la función se llama de nuevo a sí misma con `fbf` la misma expresión sin el primer elemento del and, es decir si la expresión era `(^ a b c d)` el nuevo `fbf` será `(^ b c d)`, y se establece como `tree` otra llamada a la función `expand-truth-tree-aux` con `fbf` el elemento que hemos extraído de la lista (en el caso del ejemplo la "a"), y con `tree` el árbol que teníamos. De este modo se van evaluando y expandiendo 1 a 1 las distintas sub-expresiones del and y se van añadiendo al árbol.

Mediante esta implementación acabamos consiguiendo un árbol de verdad con el formato:

```
((a b c d...) ( e f g h...)...)
```

donde cada uno de los elementos de la lista externa es una de las ramas del árbol de verdad y cada uno de los elementos de las listas internas (las letras) son los literales (que pueden ser tanto positivos como negativos) que deben cumplirse (todos) para que dicha rama no tenga contradicciones.

Apartado 4.2

Batería de ejemplos

Para probar la correcta implementación del algoritmo pedido debemos comprobar que la función implementada `truth-tree` funcione correctamente. Para ello, sabiendo que las funciones del apartado 4.1 son correctas, tan solo sería necesario probar 2 ejemplos, uno sat y otro unsat de modo que veamos si la función determina correctamente si son posibles o no. Además probamos también el caso especial de que la expresión que se le pase a la función sea `NIL`.

Con este fin hemos desarrollado la siguiente batería de ejemplos:

```
(truth-tree '(^ (v a (! b)) (v (! a) b))) -> T
(truth-tree '(^ (v a (! b)) (v (! a) b) c (! c))) -> NIL
(truth-tree NIL) -> NIL
```

Pseudocódigo

Para determinar si la expresión proporcionada es SAT o UNSAT empleamos la función `truth-tree`, que llama a las funciones del apartado anterior para limpiar la expresión y crear el árbol, así como a otra función nueva llamada `sat` que determina si el árbol es sat o no; esta última emplea a su vez una función auxiliar llamada `contradiction` que recibe una rama del árbol y determina si hay o no contradicciones en ella.

El pseudocódigo de estas funciones es:

```
truth-tree(fbf):
    devolver sat(expand-truth-tree-aux(limpiar fbf) NIL)
```

```
sat(fbf):
    si fbf está vacía:
        devolver NIL
    si contradiction(NIL, NIL, first(fbf)) es true:
        devolver sat(rest(fbf))
    si no:
        devolver true
```

```
contradiction(pos, neg, fbf):
    si fbf está vacía:
        devolver NIL
    si fbf es un literal:
        devolver NIL
    si el primer elemento de fbf es un literal positivo:
        si pertenece a neg:
            devolver true
        si no:
            añadir elemento a pos
            devolver contradiction(pos, neg, rest(fbf))
    si el primer elemento de fbf es un literal negativo:
        si pertenece a pos:
            devolver true
```

```
        si no:
            añadir elemento (sin la negación) a neg
            devolver contradiction(pos, neg, rest(fbf))
    si no:
        devolver NIL
```

Comentarios sobre la implementación

Para implementar el algoritmo que determine si una expresión es sat o no hemos implementado la función `truth-tree` que llama a las funciones `expand-truth-tree-aux` y `limpiar` del apartado anterior para generar el árbol de verdad correspondiente a la expresión, y tras esto llama a otra función auxiliar `sat` que determina si este árbol es sat o no.

Para determinar si un árbol es sat o no, la función `sat` comprieba si el árbol es `NIL` y si lo es devuelve `NIL`, tras esto comprueba si la primera rama del árbol (el primer elemento de la lista con el formato explicado en el apartado anterior) tiene contradicciones a través de la función auxiliar `contradiction` explicada a continuación, y si no las tiene determina que el árbol es sat devolviendo T, y de lo contrario se llama de nuevo a sí misma con el resto del árbol (el árbol sin la primera rama ya estudiada).

Para determinar si hay contradicciones en una rama, la función `contradiction` emplea una lista `pos` donde almacena los literales positivos que hay en la rama y otra lista `neg` donde almacena los negativos (sin la negación), ambas listas comienzan estando vacías. En primer lugar la función comprueba si la rama está vacía y si lo está devuelve `NIL` (caso base), tras esto comprueba si la rama es un literal (ya sea positivo o negativo) y si lo es devuelve `NIL` (pues no hay contradicciones). Si no se trata de un literal, comprueba si el primer elemento de la rama es un literal positivo, y si lo es mira si está en la lista de negativos `neg`, si lo está implica que en la misma rama está el mismo literal negado y no negado y por tanto hay una contradicción por lo que devuelve T, si por el contrario no pertenece a `neg` no hay contradicción en la rama por el momento y la función se llama a sí misma con el resto de la rama (sin el elemento ya analizado) añadiendo este elemento a la lista de positivos `pos`. Si por el contrario el elemento es un literal negativo, comprueba si está en la lista de positivos `pos`, si está hay contradicción y devuelve T, y si nó se llama a sí misma de la misma forma que en el caso anterior pero añadiendo el elemento a la lista `neg` en lugar de a la de positivos (en este caso sin añadir la negación, se añade solo la parte positiva del literal para facilitar la comparación). De este modo consigue saber si la rama tiene o no comparaciones pues cuando la rama acaba, la función se llamará a sí misma con `NIL` como rama y devolverá `NIL` (como debe ser pues no hay contradicciones). Permitiendo a `sat` saber si hay contradicciones o no para determinar si el árbol es sat o unsat.

Finalmente `truth-tree` devuelve el resultado de la función `sat` sobre el árbol que genera `expand-truth-tree-aux` a partir de la expresión ya "limpia" gracias a `limpiar`.

Preguntas

> 1) Si en lugar de (∧ A (∨ B C)) tuviésemos (∧ A (¬ A) (∨ B C)), ¿qué sucedería?

Como A y (¬A) son incompatibles (^ A (¬A)) no es puede cumplirse, por lo que `truth-tree` devolverá `NIL`.

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

De nuevo para que esa expresión se cumpla debe cumplirse (^ A (¬A)) que es imposible, por lo que de nuevo se devolverá `NIL`.

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

La función `expand-truth-tree` devuelve como va evolucionando cada una de las ramas del árbol de verdad según este se va expandiendo.

Ejercicio5

Apartado 5.1

Casos especiales

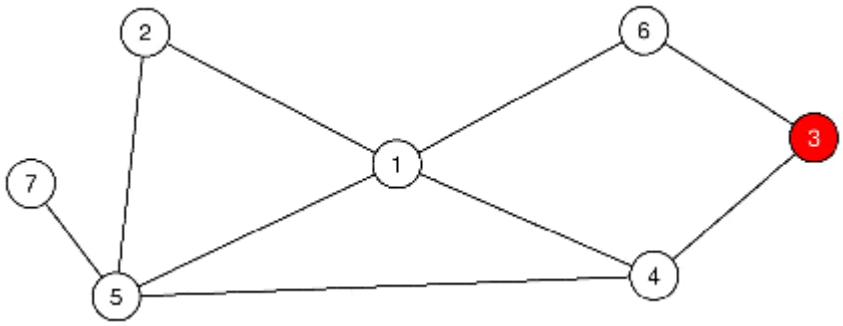
Grafo vacío

En un grafo vacío el algoritmo no tiene grafo que analizar y por tanto no existirá ningún camino sean cuales sean los nodos inicial y final propuestos.

Grafo cuyo inicio es el final.

En este caso el primer nodo que el algoritmo comprobaría sería el nodo inicial, detecta que es el final, y por tanto ha encontrado un camino que es el nodo.

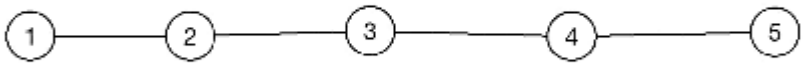
En este ejemplo se quiere ir del nodo 3 al nodo 3.



El camino encontrado es en sí, el nodo 3.

Grafo cadena

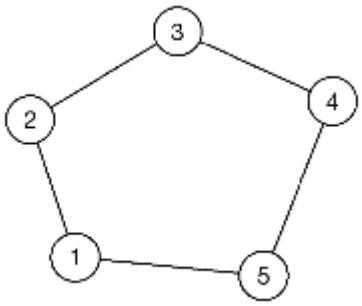
En n grafo cadena solo existe un camino posible entre 2 nodos, que es el que encontraría el algoritmo ya que iría analizando un nodo en cada dirección desde el origen, y cuando llegase al nodo destino habrá encontrado el camino. Si antes de llegar al destino se encuentra con que en una de las 2 direcciones no quedan nodos por explorar, esa será la dirección errónea y continuará con los únicos nodos que puede explorar que son los que le queden en la otra dirección, hasta encontrar el camino.



En este caso, por ejemplo, si se quisiese ir del nodo 2 al 4, el único camino posible sería 2, 3, 4, y, por tanto, este sería el camino que encontraría BFS.

Grafo cíclico

En un grafo cíclico existen siempre 2 caminos entre 2 nodos. En este caso el algoritmo encontrará el más corto de los 2 ya que al ir analizando en una cola fifo, irá comprobando cada vez un nodo de cada uno de los caminos, y, cuando encuentre el nodo final parará.



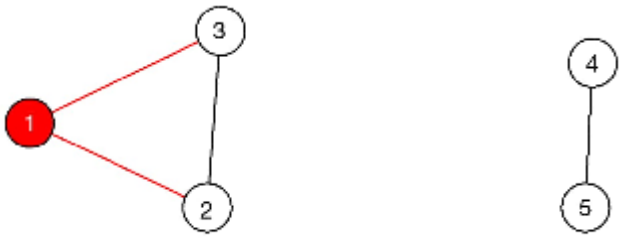
En este caso, si se desease ir del nodo 3 al 5, habría 2 caminos posibles. Uno de ellos sería 3, 4, 5, y el otro sería 3, 2, 1, 5. De entre estos 2 FBS encontraría 3, 4, 5 por ser el más corto.

Grafo no conexo que debe ir desde uno de los sub-grafos conexos al otro

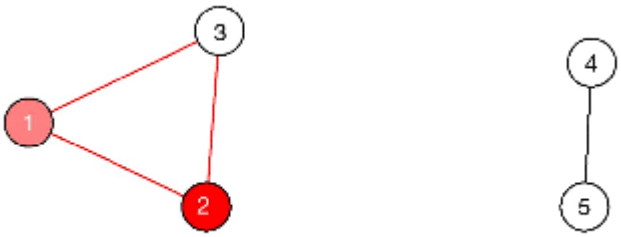
En este caso, BFS recorre el sub-grafo conexo al que pertenece el origen, y luego para pues todos los nodos que puede explorar ya han sido explorados antes. Por tanto no existe camino entre el origen y el destino.

En este ejemplo tratamos de ir del nodo 1 al 4.

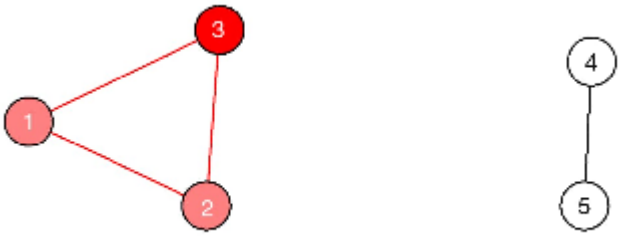
1º Se expande el nodo 1 añadiendo los nodos 2 y 3 a la cola (sus vecinos).



2º Se expande ahora el nodo 2, el 3 no se añade a la cola pues ya ha sido descubierto.



3º Se expande el nodo 3, no se añaden nodos a la cola pues todos sus vecinos ya han sido descubiertos.



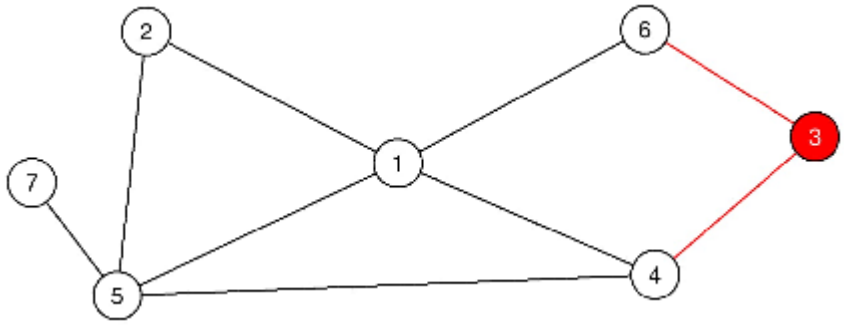
No quedan nodos que explorar en la cola, luego el algoritmo acaba sin encontrar ningún camino.

Casos Típicos

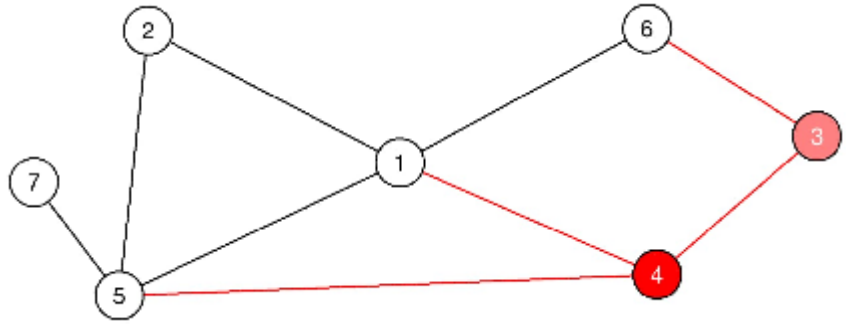
1) Grafo no dirigido

Tomamos como primer caso "típico" un grafo no dirigido en el que se quiere ir del nodo 3 al nodo 7. En este caso el algoritmo se llevaría a cabo de la siguiente forma:

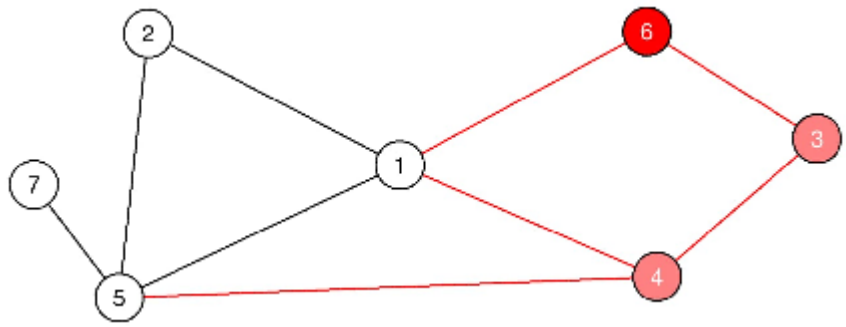
1º Se expande el nodo 3 descubriendo sus vecinos y añadiéndolos a la cola (de menor a mayor).



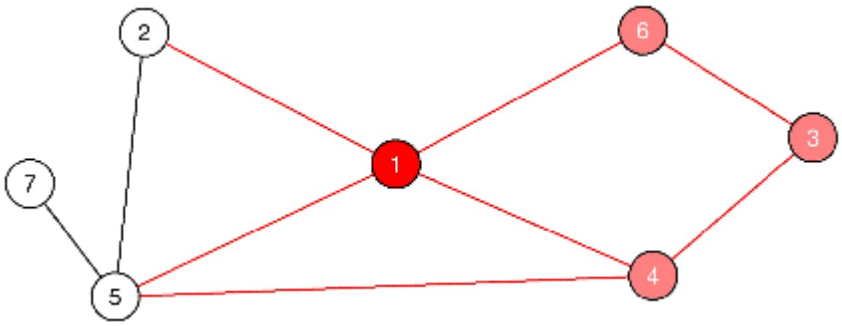
2º Se explora el primer nodo de la cola (4) y se expande, de nuevo se añaden los vecinos a la cola.



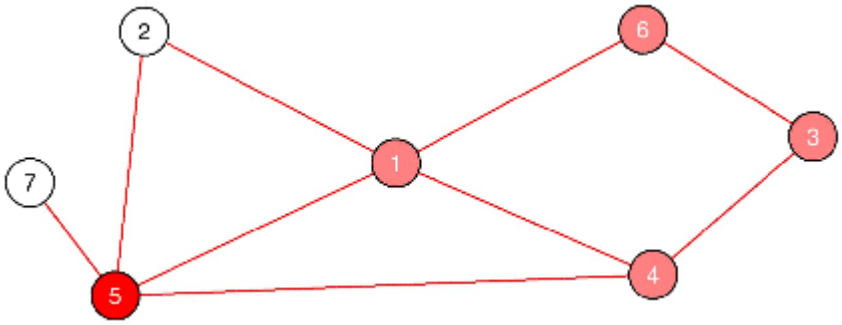
3º Le toca el turno al 6 pero el 1 no se añade a la cola pues ya se había añadido antes.



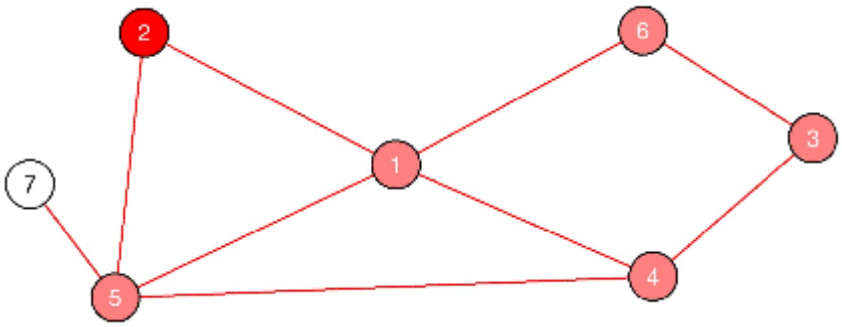
4º El siguiente en la cola es el 1, al explorarlo tan solo se añade el 2 a la cola.



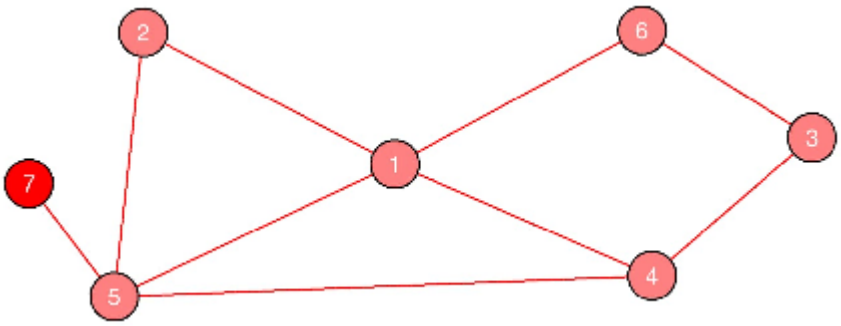
5º Procedemos a explorar el 5, añadimos el 7 a la cola pues el 7 ya estaba en ella.



6º Exploramos el próximo de la cola (2) pero ninguno de sus vecinos se añade a la cola pues ya han sido explorados.



7º Toca ahora explorar el número 7, que es el destino, luego el algoritmo finaliza.



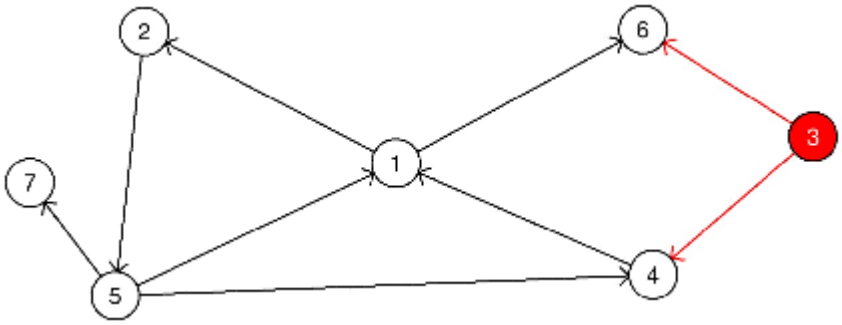
FBS acaba encontrado un camino, que es 3, 4, 5, 7.

2) Grafo dirigido

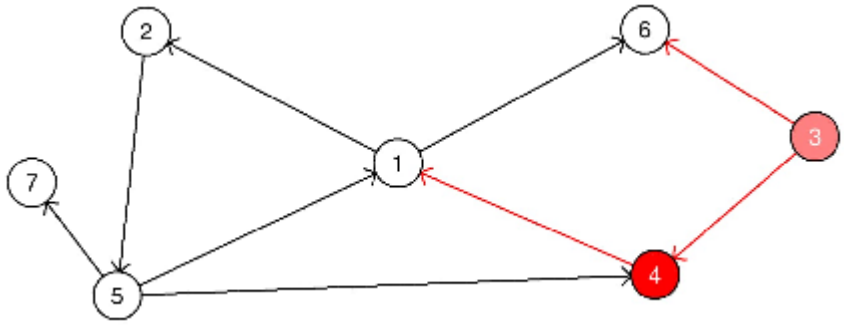
Como segundo caso típico tomamos el mismo grafo pero, esta vez, con aristas dirigidas. De nuevo intentamos ir del nodo 3 al nodo 7.

El desarrollo es muy similar pero al expandir los nodos se consideran vecinos solo los que tienen enlaces desde el nodo que se esta expandiendo hasta él, y no al revés.

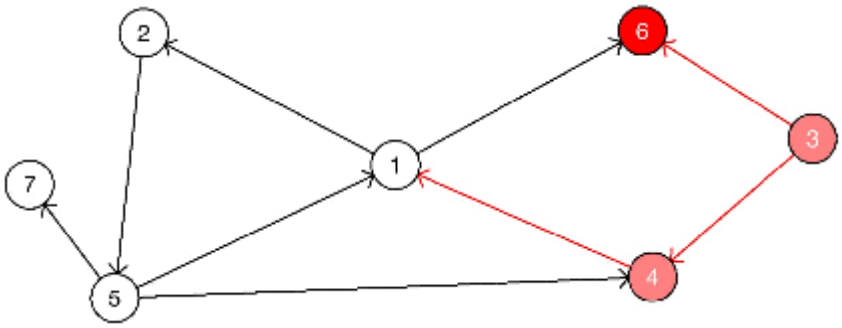
1º Se expande el nodo 3 y se añaden el 4 y el 6 a la cola.



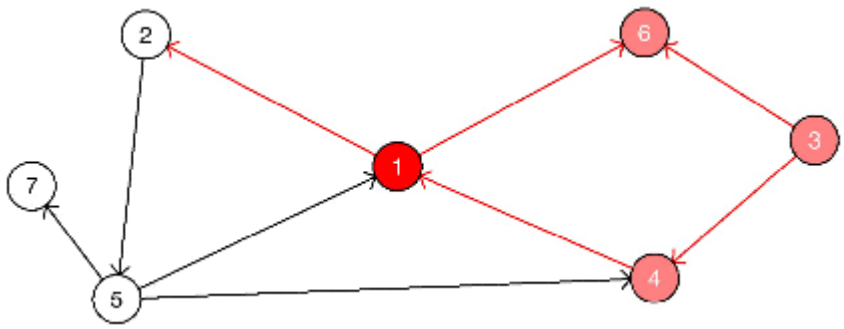
2º Se expande el nodo 4 y se añade el 1 a la cola.



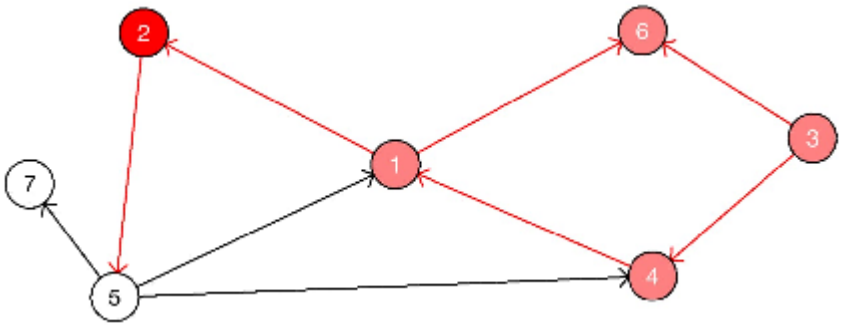
3º Se expande el nodo 6 pero no se añade ninguno a la cola pues no tiene vecinos conectados en el sentido adecuado.



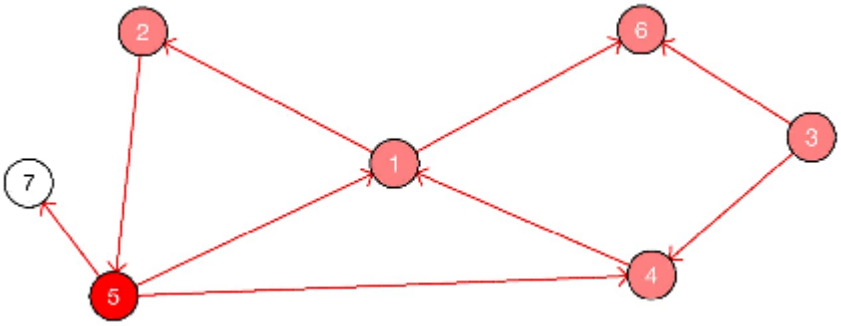
4º Se expande el nodo 1 y se añade el 2 a la cola.



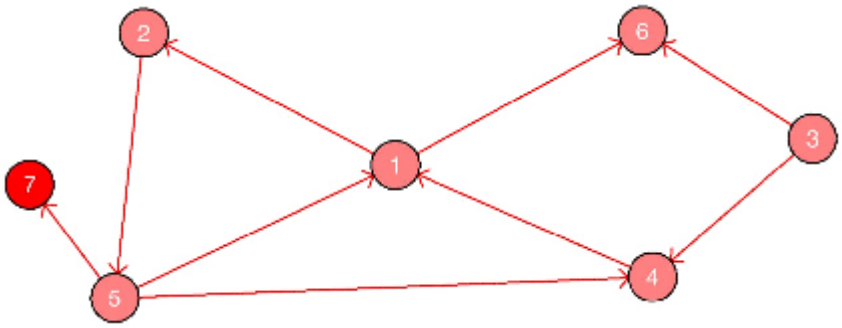
5º Se expande el nodo 2 añadiendo el 5 a la cola.



6º Se expande el nodo 5, los nodos 1 y 4 ya han sido explorados por lo que no se añaden a la cola, pero el nodo 7 si.



7º Se expande el nodo 7 que es el destino por lo que el algoritmo termina.



En este caso también se ha encontrado un camino, que es 3, 4, 1, 2, 5, 7.

Apartado 5.2

El pseudocódigo del algoritmo BFS es:

```
BFS(grafo, inicio, destino):
    Q = nueva-cola()
    lista_explorados = nueva-lista()
    añadir inicio a Q
    mientras Q no esté vacía:
        sacar el primer nodo de Q
        si nodo == destino:
            devolver ÉXITO
        para cada vecino de nodo:
            si vecino no esta en lista_explorados:
            añadir vecino a Q
        añadir nodo a lista_explorados
    devolver FALLO
```

Apartado 5.5

La función:

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

Soluciona el problema de encontrar el camino más corto entre 2 nodos del grafo ya que realiza el algoritmo BFS en el grafo estableciendo como elemento inicial de Q el nodo desde el que se quiere iniciar, haciendo que el primer elemento que se extraiga de Q en el algoritmo sea este primero, de modo que el algoritmo comience en este nodo y termine en el nodo que se introduzca como end.

Apartado 5.6

La secuencia de llamadas que produce la evaluación:

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

es:

```
(shortest-path 'a 'f '((a d) (b d f) (c e) (d f) (e b f) (f)))
    (bfs 'f '((a)) '((a d) (b d f) (c e) (d f) (e b f) (f)))
        (bfs 'f '((d a)) '((a d) (b d f) (c e) (d f) (e b f) (f)))
            (bfs 'f '((f d a)) '((a d) (b d f) (c e) (d f) (e b f) (f)))
            -> (a d f)
        -> (a d f)
    -> (a d f)
-> (a d f)
```

obteniendo, finalmente que el camino más corto para este grafo entre los nodos a y f es (a d f).

Apartado 5.7

Para hallar el camino más corto en el grafo propuesto entre los nodos B y G se debe realizar la siguiente llamada:

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

y esta llamada produce la salida (b d g), que, efectivamente es el camino más corto.

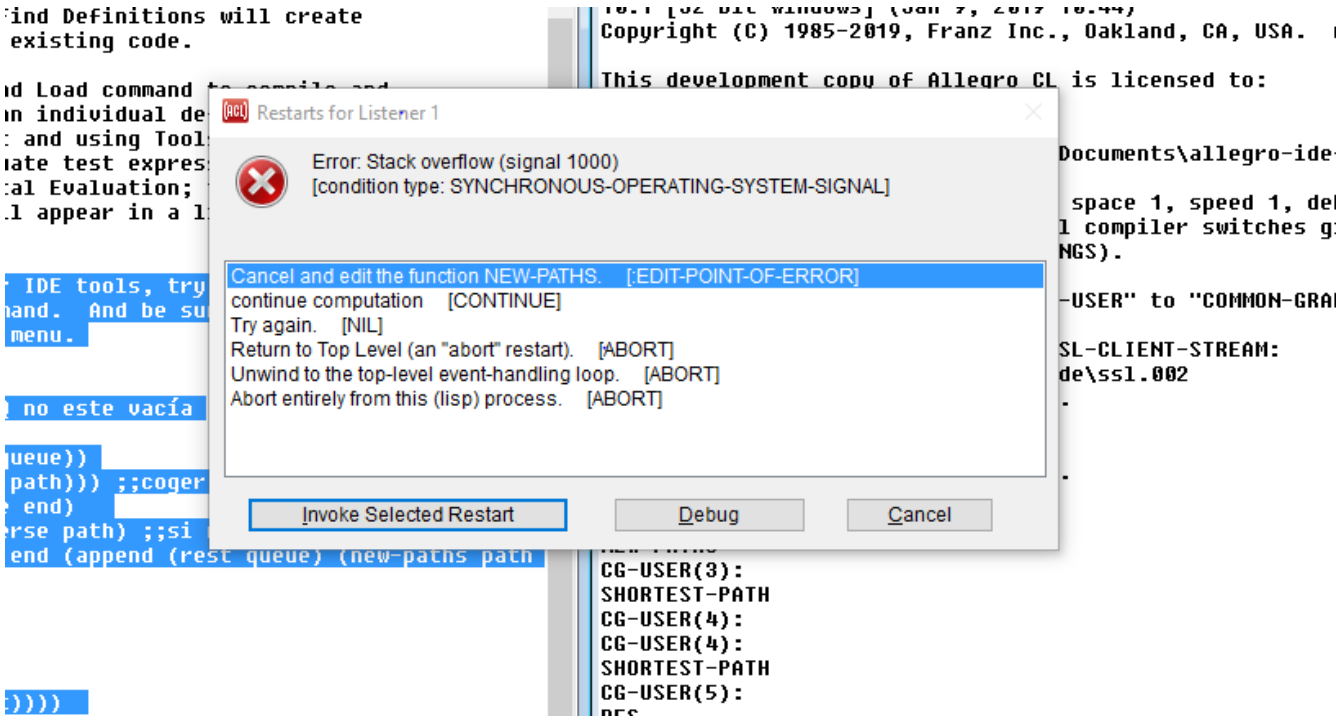
Apartado 5.8

Ejemplo

Para dar un ejemplo de que el algoritmo empleado en los apartados anteriores da error para grafos con bucles ya que entra en una recursión infinita vamos a probar a ejecutar el siguiente ejemplo:

```
;;Grafo inconexo con ciclos donde el destino no esta conectado al origen
(shortest-path 'a 'f '((a b c) (b a d) (c a e) (d b e) (e c d) (f))))
```

Al probar a ejecutar este algoritmo en allegro obtenemos el siguiente error:



Esto se debe a que al no controlar si el nodo que se está expandiendo ha sido ya explorado o no, el algoritmo entra en un bucle infinito, y allegro, tras 1000 iteraciones determina que es un bucle infinito y devuelve este error.

Por tanto es necesario implementar un nuevo código que solucione este fallo teniendo en cuenta que nodos han sido ya explorados con anterioridad para evitar bucles infinitos.

Batería de ejemplos

Para probar el correcto funcionamiento de la modificación pedida del algoritmo bfs, hemos diseñado la siguiente batería de ejemplos:

```
;;Grafo inconexo con ciclos donde el destino no esta conectado al origen
(shortest-path-improved 'a 'f '((a b c) (b a d) (c a e) (d b e) (e c d) (f)))) -> NIL

;;Grafo con ciclos donde el destino no es un nodo del grafo
(shortest-path-improved 'a 'f '((a b c) (b a d) (c a) (d b e) (e d))) -> NIL

;;Grafo con ciclos donde el origne no pertenece al grafo
(shortest-path-improved 'f 'a '((a b c) (b a d) (c a) (d b e) (e d))) -> NIL

;;Grafo con ciclos y un solo camino entre origen y destino
(shortest-path-improved 'a 'f '((a b c) (b a d) (c a e) (d b e) (e c d f) (f e)))      -
> (a c e f)

;;Grafo con ciclos y varios caminos entre origen y destino
(shortest-path-improved 'a 'f '((a b c) (b a d f) (c a e) (d b e) (e c d f) (f e b)))  -
> (a b f)
```

```
;;Grafo nulo (vacío)
(shortest-path-improved 'a 'f NIL) -> NIL
```

Pseudocódigo

El pseudocódigo de esta nueva implementación es:

```
bfs-improved-aux(end, queue, net, explored)
    si queue está vacía:
        devolver lista vacía
    si la primera sublista de queue está vacía
        devolver NIL
    si no:
        path = primer elemento de queue
        node = primer elemento de path
        si node == end:
            devolver dar-vuelta(path)
        si node pertenece a explored:
            devolver bfs-improved-aux(end, concatenar(rest(queue), new-paths(path,
                node, net), net, cons(node, explored))
        si no:
            devolver bfs-improved-aux(end, rest(queue), net, explored)
```

Donde la función new-paths es la misma que en el caso anterior.

Comentarios sobre la implementación

Para implementar este nuevo funcionamiento se ha decidido partir del código para el bfs que se nos proporcionaba y hacerle una simple modificación que consiste en añadir un nuevo parámetro llamado `explored` que es una lista donde vamos añadiendo los nodo que ya han sido explorados, de este modo antes de llamar recursivamente a la función (es decir, antes de explorar los vecinos de un nodo) comprobamos si el nodo en el que estamos está o no en la lista de explorados, si está, llamamos a la función de nuevo pero sin el primer elemento de `queue` (el camino que se está estudiando en ese momento), y si no está lo añadimos a la lista y llamamos a la función de forma recursiva del mismo modo que en el método "no mejorado" de bfs.

De este modo evitamos que la función pueda evaluar caminos que pasen 2 veces por el mismo nodo (formando un ciclo) pues estos caminos nunca van a ser la solución óptima de BFS.

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 1
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Apartado 1.1
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; cosine-distance (x y)
;;; Calcula la distancia coseno de un vector de forma recursiva
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;;
;;; INPUT: x: vector, representado como una lista
;;;         y: vector, representado como una lista
;;; OUTPUT: distancia coseno entre x e y
;;;
(defun cosine-distance (x y distance-measure)
  (if (and (null x) (null y))
      0
      (let ((denominator (* (sqrt (funcall distance-measure x x)) (sqrt (funcall
distance-measure y y)))))
      (if (= denominator 0)
        nil
        (- 1 (/ (funcall distance-measure x y) denominator))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; dot-product-rec (x y)
;;; Calcula el producto escalar de los vectores x e y de forma recursiva
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;;
;;; INPUT: x: vector, representado como una lista
;;;         y: vector, representado como una lista
;;;
;;; OUTPUT: producto escalar del vector x por el vector y
;;;
(defun dot-product-rec (x y)
  (if (or (null x) (null y))
    0
    (+ (* (first x) (first y)) (dot-product-rec (rest x) (rest y)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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.
;;;
;;; INPUT: x: vector, representado como una lista
;;;         y: vector, representado como una lista
;;; OUTPUT: distancia coseno entre x e y
;;;
(defun cosine-distance-rec (x y)
  (cosine-distance x y #'dot-product-rec))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; dot-product-mapcar (x y)
;;; Calcula el producto escalar de los vectores x e y usando mapcar
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;;
;;; INPUT: x: vector, representado como una lista
```

```
;;;         y: vector, representado como una lista
;;;
;;; OUTPUT: producto escalar del vector x por el vector y
;;;
(defun dot-product-mapcar (x y)
    (apply #'+ (mapcar #'* x y)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; cosine-distance-mapcar
;;; Calcula la distancia coseno de un vector usando mapcar
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;;
;;; INPUT:  x: vector, representado como una lista
;;;         y: vector, representado como una lista
;;; OUTPUT: distancia coseno entre x e y
;;;
(defun cosine-distance-mapcar (x y)
  (cosine-distance x y #'dot-product-mapcar))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Apartado 1.2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; map-vectors-cosine-distance
;;; Devuelve una lista de tuplas de vectores con su distancia coseno
;;; a vector, siempre que esta distancia sea menor que
;;; confidence-level
;;; INPUT:  vector: vector que representa a una categoria,
;;;                 representado como una lista
;;;         lst-of-vectors vector de vectores
;;;         confidence-level: Nivel de confianza (parametro opcional)
;;;         dist-func: Funcion usada para medir la distancia coseno
;;;                    (parametro opcional)
;;; OUTPUT: Lista de tuplas de vectores y su distancia coseno a vector,
;;;         siempre que esta sea menor que confidence-level
;;;
(defun map-vectors-cosine-distance (vector lst-of-vectors &optional (confidence-level 0))
    ;;; Tenemos que comprobar que la distancia es menor o igual que 1 - el nivel de
confianza. Te esta forma, sabemos
    ;;; que el elemento de la lista está lo suficientemente cerca del vector referencia.
    ;;; Si la distancia es nil o se cumple la condición anterior, el lambda devuelve nil
y por tanto mapcan no lo añade a la lista
    ;;; mapcan concatena los elemetos, por lo que tenemos que hacer (list (list v
confidence))
    (mapcan (lambda (v) (let ((distance (cosine-distance-mapcar v vector))) (when (and
distance (<= distance (- 1 confidence-level))) (list (list v distance))))) lst-of-
vectors))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; order-vectors-cosine-distance
;;; Devuelve aquellos vectores similares a una categoria
;;; INPUT:  vector: vector que representa a una categoria,
```

```
;;;                 representado como una lista
;;;         lst-of-vectors vector de vectores
;;;         confidence-level: Nivel de confianza (parametro opcional)
;;; OUTPUT: Vectores cuya semejanza con respecto a la
;;;         categoria es superior al nivel de confianza,
;;;         ordenados de mayor a menor confianza
;;;
(defun order-vectors-cosine-distance (vector lst-of-vectors &optional (confidence-level
0))
    ;;; Ordenamos una copia del vector por ser sort una función destructiva
    (mapcar #'first (sort (copy-list (map-vectors-cosine-distance vector lst-of-vectors
confidence-level)) #'> :key #'second)))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Apartado 1.3
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; get-text-category_dist
;;; Devuelve una lista con una tupla con el id de la categoría
;;; y su distancia al texto
;;;
;;; INPUT : category:   lista que representa la categoría
;;;         text:       lista que representa al texto
;;;         distance-measure: funcion de distancia
;;; OUTPUT: lista con una tupla formada por el id de la categoría
;;;         y su distancia al texto
;;;
( defun get-text-category-dist (category text distance-measure)
    (let ((distance (funcall distance-measure (rest text) (rest category))))
        (if (null distance)
            nil
            (list (list (first category) distance)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; get-text-category
;;; Asigna a cada texto la categoría que le corresponde
;;;
;;; INPUT : categories: vector de vectores, representado como
;;;                     una lista de listas
;;;         text:       lista que representa al texto
;;;         distance-measure: funcion de distancia
;;; OUTPUT: Par formado por el vector que identifica la categoria
;;;         de menor distancia , junto con el valor de dicha distancia
;;;
( defun get-text-category (categories text distance-measure)
    ;;; Ordenamos una copia del vector por ser sort una función destructiva
    (first (sort (copy-list (mapcan (lambda (cat) (get-text-category-dist cat text
distance-measure)) categories)) #'> :key #'second)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; get-vectors-category (categories vectors distance-measure)
;;; Clasifica a los textos en categorias .
;;;
;;; INPUT : categories: vector de vectores, representado como
;;;                     una lista de listas
;;;         texts:      vector de vectores, representado como
;;;                     una lista de listas
;;;         distance-measure: funcion de distancia
;;; OUTPUT: Pares formados por el vector que identifica la categoria
;;;         de menor distancia , junto con el valor de dicha distancia
;;;
( defun get-vectors-category (categories texts distance-measure)
    (if (or (null categories) (null texts))
        nil
        (mapcar (lambda (text) (get-text-category categories text distance-measure))
texts)))




;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Apartado 2.1
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; newton
;;; Estima el cero de una funcion mediante Newton-Raphson
;;;
;;; INPUT : f: funcion cuyo cero se desea encontrar
;;; df: derivada de f
;;; max-iter : maximo numero de iteraciones
;;; x0: estimacion inicial del cero ( semilla )
;;; tol: tolerancia para convergencia ( parametro opcional )
;;; OUTPUT : estimacion del cero de f, o NIL si no converge
;;;
(defun newton (f df max-iter x0 &optional ( tol 0.001))
    (if (<= max-iter 0)
        NIL
        (let ((division (/ (funcall f x0) (funcall df x0))))
            (if (<= (abs division) tol)
                x0
                (newton f df (- max-iter 1) (- x0 division) tol)))))



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

```
;; Apartado 2.2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; one-root-newton
;;; Prueba con distintas semillas iniciales hasta que Newton
;;; converge
;;;
;;; INPUT : f: funcion de la que se desea encontrar un cero
;;; df: derivada de f
;;; max-iter : maximo numero de iteraciones
;;; semillas : semillas con las que invocar a Newton
;;; tol: tolerancia para convergencia ( parametro opcional )
;;;
;;; OUTPUT : el primer cero de f que se encuentre , o NIL si se diverge
;;; para todas las semillas
;;;
(defun one-root-newton (f df max-iter semillas &optional ( tol 0.001))
    (let ((newton-res (newton f df max-iter (first semillas) tol)))
        (if (null newton-res)
            (if (null (rest semillas))
                NIL
                (one-root-newton f df max-iter (rest semillas) tol))
            newton-res)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Apartado 2.3
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; all-roots-newton
;;; Prueba con distintas semillas iniciales y devuelve las raices
;;; encontradas por Newton para dichas semillas
;;;
;;; INPUT : f: funcion de la que se desea encontrar un cero
;;; df: derivada de f
;;; max-iter : maximo numero de iteraciones
;;; semillas : semillas con las que invocar a Newton
;;; tol: tolerancia para convergencia ( parametro opcional )
;;;
;;; OUTPUT : las raices que se encuentren para cada semilla o nil
;;; si para esa semilla el metodo no converge
;;;
(defun all-roots-newton (f df max-iter semillas &optional ( tol 0.001))
  (mapcar #'(lambda (x) (newton f df max-iter x tol)) semillas))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Apartado 2.3.1
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; list-not-nil-roots-newton
;;; Quita los nil de la lista que devuelve all-roots-newton
;;; con los parámetros pasados como argumento
;;;
;;; INPUT : f: funcion de la que se desea encontrar un cero
;;; df: derivada de f
;;; max-iter : maximo numero de iteraciones
;;; semillas : semillas con las que invocar a Newton
;;; tol: tolerancia para convergencia ( parametro opcional )
;;;
;;; OUTPUT : las raices que se encuentren para cada semilla sin nil
;;;
(defun list-not-nil-roots-newton (f df max-iter semillas &optional ( tol 0.001))
    (mapcan (lambda (x)
        (if (null x)
            '()
            (list x)))
        (all-roots-newton f df max-iter semillas tol)))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 3
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Apartado 3.1
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-elt-lst
;;; Combina un elemento dado con todos los elementos de una lista
;;;
;;; INPUT: elem: elemento a combinar
;;;        lst: lista con la que se quiere combinar el elemento
;;;
;;; OUTPUT: lista con las combinacion del elemento con cada uno de los
;;;         de la lista. Si alguno de los elementos es nil, devolvemos
;;;         nil
(defun combine-elt-lst (elt lst)
    (if (null lst))
        nil
        (mapcar (lambda (x) (list elt x)) lst))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Apartado 3.2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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

```
;;; combine-lst-lst
;;; Calcula el producto cartesiano de dos listas
;;;
;;; INPUT: lst1: primera lista
;;;        lst2: segunda lista
;;;
;;; OUTPUT: producto cartesiano de las dos listas
(defun combine-lst-lst (lst1 lst2)
    ;;; Usamos mapcan porque combine-elt-lst devuelve una lista, y mapcan concatena estas
listas
    (mapcan (lambda (elt) (combine-elt-lst elt lst2)) lst1))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Apartado 3.3
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-elt-lst-aux
;;; Combina un elemento dado con todos los elementos de una lista
;;;
;;; INPUT: elem: elemento a combinar
;;;        lst: lista con la que se quiere combinar el elemento
;;;
;;; OUTPUT: lista con las combinacion del elemento con cada uno de los
;;;         de la lista, como una concatenación de ambos.
;;;         Si alguno de los elementos es nil, devolvemos nil
(defun combine-elt-lst-aux (elt lst)
    (if (or (null elt) (null lst))
        nil
        (mapcar (lambda (x) (cons elt x)) lst)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-lst-lst-aux
;;; Calcula el producto cartesiano de dos listas
;;;
;;; INPUT: lst1: primera lista
;;;        lst2: segunda lista
;;;
;;; OUTPUT: producto cartesiano de las dos listas
(defun combine-lst-lst-aux (lst1 lst2)
  (if (or (null lst1) (null lst2))
    nil
    (append (combine-elt-lst-aux (first lst1) lst2) (combine-lst-lst-aux (rest lst1)
lst2))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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
;;;
;;; INPUT: lstolsts: lista de listas
```

```
;;;
;;; OUTPUT: lista con todas las posibles combinaciones de elementos
(defun combine-list-of-lsts (lstolsts)
  (if (null lstolsts)
    (list nil)
    (combine-lst-lst-aux (first lstolsts) (combine-list-of-lsts (rest lstolsts)))))




;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; defino operadores logicos
(defconstant +bicond+ '<=>)
(defconstant +cond+   '=>)
(defconstant +and+    '^)
(defconstant +or+     'v)
(defconstant +not+    '!)

;; definiciones de valores de verdad, conectores y atomos
(defun truth-value-p (x)
  (or (eql x T) (eql x NIL)))

(defun unary-connector-p (x)
  (eql x +not+))

(defun binary-connector-p (x)
  (or (eql x +bicond+)
      (eql x +cond+)))

(defun n-ary-connector-p (x)
  (or (eql x +and+)
      (eql x +or+)))

(defun bicond-connector-p (x)
  (eql x +bicond+))

(defun cond-connector-p (x)
    (eql x +cond+))

(defun connector-p (x)
  (or (unary-connector-p  x)
      (binary-connector-p x)
      (n-ary-connector-p  x)))

(defun positive-literal-p (x)
  (and (atom x)
       (not (truth-value-p x))
       (not (connector-p x))))

(defun negative-literal-p (x)
```

```
  (and (listp x)
       (eql +not+ (first x))
       (null (rest (rest x)))
       (positive-literal-p (second x))))

(defun literal-p (x)
  (or (positive-literal-p x)
      (negative-literal-p x)))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Apartado 4.1
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; solve-simple-implication
;;; Funcion que dada una implicación (=> A B) devuelve (v (! A) B).
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar de
;;;           la forma (=> A B).
;;; OUTPUT : (v (! A) B)
;;;
(defun solve-simple-implication (fbf)
    (list +or+ (list +not+ (second fbf)) (third fbf)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; solve-double-implication
;;; Funcion que dada una doble implicación (<=> A B) devuelve
;;; (^ (v (! A) B) (v (! B) A)).
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar de
;;;           la forma (<=> A B).
;;; OUTPUT : (^ (v (! A) B) (v (! B) A))
;;;
(defun solve-double-implication (fbf)
    (list +and+ (solve-simple-implication fbf) (solve-simple-implication (list (first
fbf) (third fbf) (second fbf)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; solve-implication
;;; Funcion que dada una implicación o doble implicacion
;;; devuelve su equivalente.
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar de
;;;           la forma (=> A B) o (<=> A B).
;;; OUTPUT : Equivalente de la expresion
;;;
(defun solve-implication (fbf)
    (if (cond-connector-p (first fbf))
        (solve-simple-implication fbf)
        (solve-double-implication fbf)))
```

```
(defun negate(fbf)
    (if (null (rest fbf))
        (list (cons +not+ fbf))
        (cons (list +not+ (first fbf)) (negate (rest fbf)))))


(defun not-connector (fbf)
    (cond ((positive-literal-p fbf) (list +not+ fbf))
        ((unary-connector-p (first fbf)) (second fbf))
        ((binary-connector-p (first fbf)) (not-connector (solve-implication fbf)))
        ((eql +and+ (first fbf)) (cons +or+ (negate(rest fbf))))
        (t (cons +and+ (negate(rest fbf))))))


(defun limpiar (fbf)
    (cond ((literal-p fbf) fbf)
        ((binary-connector-p (first fbf)) (limpiar (solve-implication fbf)))
        ((unary-connector-p (first fbf)) (limpiar (not-connector (second fbf))))
        (t (cons (first fbf) (mapcar (lambda (x) (limpiar x)) (rest fbf))))))



(defun add-to-lists (elem lsts)
    (if (null lsts)
        (list elem)
        (add-rec elem lsts)))


(defun add-rec (elem lsts)
    (cond ((null lsts) nil)
        ((literal-p (first lsts)) (cons (list elem (first lsts)) (add-rec elem (rest
lsts))))
        (t (cons (cons elem (first lsts)) (add-rec elem (rest lsts))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; exapand-truth-tree-aux
;;; Función resursiva que ecibe una expresion y construye su arbol
;;; de verdad para determinar si es SAT o UNSAT
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar.
;;; OUTPUT : T   - FBF es SAT
;;;          N   - FBF es UNSAT
;;;
(defun expand-truth-tree-aux (fbf tree)
   (cond ((eql fbf +and+) tree)
        ((literal-p fbf) (expand-truth-tree-aux +and+ (add-to-lists fbf tree)))
        ((eql (first fbf) +or+) (mapcan #'(lambda (x) (expand-truth-tree-aux x tree))
(rest fbf)))
        ((eql (first  fbf) +and+)
            (if (null (second fbf))
                tree
```

```
                (expand-truth-tree-aux (cons +and+ (cddr fbf)) (expand-truth-tree-aux
(second fbf) tree))))
        (t NIL)))



(defun contradiction (pos neg fbf)
    (cond ((null fbf) NIL)
        ((positive-literal-p fbf) NIL)
        ((positive-literal-p (first fbf))
            (if (member (first fbf) neg)
                t
                (contradiction (cons (first fbf) pos) neg (rest fbf))))
        ((negative-literal-p (first fbf))
            (if (member (second (first fbf)) pos)
                t
                (contradiction pos (cons (second (first fbf)) neg) (rest fbf))))
        (t NIL)))


(defun sat (fbf)
    (cond ((null fbf) NIL)
        ((contradiction '() '() (first fbf)) (sat (rest fbf)))
        (t t)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; truth-tree
;;; Recibe una expresion y construye su arbol de verdad para
;;; determinar si es SAT o UNSAT
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar.
;;; OUTPUT : T   - FBF es SAT
;;;          N   - FBF es UNSAT
;;;
(defun truth-tree (fbf)
    (sat (expand-truth-tree-aux (limpiar fbf) '())))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 5
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Breadth-first-search in graphs
;;;
(defun bfs (end queue net)
    (if (null queue) ;;mientras Q no este vacía
        '()
        (let* ((path (first queue))
            (node (first path))) ;;coger primer nodo de Q
            (if (eql node end)
                (reverse path) ;;si nodo == fin: devolver dar.vuelta(camino)
```

```
                (bfs end (append (rest queue) (new-paths path node net)) net))))) ;;si
no: se llama a si mismo añadiendo los vecinos a Q, los vecinos se cogen con new-paths.


(defun new-paths (path node net)
    (mapcar #'(lambda (n)
        (cons n path))
        (rest (assoc node net))))


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

;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; shortest-path-improved
;;; Version de busqueda en anchura que no entra en recursion
;;; infinita cuando el grafo tiene ciclos
;;; INPUT:   end: nodo final
;;;          queue: cola de nodos por explorar
;;;          net: grafo
;;; OUTPUT: camino mas corto entre dos nodos
;;;         nil si no lo encuentra



(defun bfs-improved (end queue net)
    (bfs-improved-aux end queue net NIL))


(defun bfs-improved-aux (end queue net explored)
    (if (null queue)
        '()
        (if (null (first queue))
            NIL
            (let* ((path (first queue))
                (node (first path)))
                (if (eql node end)
                    (reverse path)
                    (if (null (member node explored))
                        (bfs-improved-aux end (append (rest queue) (new-paths path node
net)) net (cons node explored))
                        (bfs-improved-aux end (rest queue) net explored)))))))


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