Práctica 1: Introducción a Lisp

Ejercicio 1:

Código:

```
(defun newton (f df-dx max-iter x0 &optional (tol-abs 0.0001))
   (let ((df-dx-x0 (funcall df-dx x0))) ;; Calculamos el valor de la derivada en el punto
   (if (> (abs df-dx-x0) 1.0L-15) ;; Si la derivada es 0, devolver NIL
     (let* ((f-x0 (funcall f x0)) (xn1 (- x0 (/ f-x0 df-dx-x0)))) ;; Relaizamos cálculos previos
       (cond ((< max-iter 0) NIL) ((< (abs (- xn1 x0)) tol-abs) xn1) ;; Comprobamos si tenemos que seguir
         (t (newton f df-dx (- max-iter 1) xn1 tol-abs)))))) ;; De ser así seguimos realizando el algoritmo


(defun newton-all (f df-dx max-iter seeds &optional (tol-abs 0.0001))
   (mapcar #'(lambda (x0) (newton f df-dx max-iter x0 tol-abs)) seeds))
   ;; Aplicamos el algoritmo de Newton a todos los elementos de la lista
```

Apartado a)

Este apartado consiste en inplementar el método de Newton-Raphson para encontrar ceros en una función, proporcionando la función, su derivada y un valor inicial. Su funcionamiento es básicamente el uso de la fórmula:

$$x_{n+1} = xn - \frac{f(x_n)}{f'(x_n)}$$

Esta se aplica de forma recursiva hasta que el valor de x_{n+1} sea los suficiente cercano a cero según una determinada tolerancia que se puede introducir en la función como argumento opcional. Además, dado que el método dependiendo del valor inicial que se elija puede no converger, ha de pasarse un número máximo de iteraciones sobre el algoritmo de forma que no se quede haciendo recursión indefinidamente.

Tests:

```
(< (abs (- (newton #'sin #'cos 50 2.0) pi)) 1.0L-6) ;; Test Simple
(< (abs (- (newton #'(lambda (x) (* x x)) #'(lambda(x) (* 2 x)) 50 1.0) 0)) 1.0L-4) ;; Test Simple
```

Apartado b)

Puesto que dependiendo del valor inicial elegido se pueden encontrar distintos ceros, o ninguno si la función no los tiene o el método diverge, se pide crear una función que dada una lista de puntos iniciales devuelva otra con los ceros encontrados aplicando el algoritmo de Newton-Raphson sobre cada uno de ellos.

Tests:

(equal (newton-all #'sin #'cos 50 (mapcar #'eval '((/ pi 2) 1.0 2.0 4.0 6.0))) '(NIL 0.0 3.1415927 3.1415927 6.2831855)) ;; Test Simple

(equal (newton-all #'sin #'cos 50 '() '1) nil) ;; Test con lista vacia de semillas

(equal (newton-all #'sin #'cos 50 nil '1) nil) ;; Test con nil

Ejercicio 2:

Código:

```
(defun combine-elt-lst (elt lst)
  (mapcar #'(lambda (x) (list elt  x)) lst))
  ;; Formamos una lista con el elemento y cada elemento de la lista


(defun combine-lst-lst (lst1 lst2)
  (mapcan #'(lambda (x) (combine-elt-lst x lst2)) lst1))
  ;; Aplicamos la función anterior con cada elemento de la lista1 y la lista 2


(defun combine-list-of-lsts (lstolsts)
  (if (null (rest lstolsts)) ;Si la lista tiene un elemento
    (first lstolsts)
    (if (null (cddr lstolsts)) ; Si la lista tiene dos elementos
      (combine-lst-lst (first lstolsts) (cadr lstolsts))
      (combine-lst-lst-aux (first lstolsts) (combine-list-of-lsts(rest lstolsts))))))


(defun combine-lst-lst-aux (lst1 lst2)
  (mapcan #'(lambda (x) (combine-elt-lst-aux x lst2)) lst1))


(defun combine-elt-lst-aux (el lst)
  (mapcar #'(lambda (x) (cons el  x)) lst))
```

Apartado a)

Este apartado consiste en realizar una lista que contenga las combinaciones de un elemento dado con todos los elementos de otra lista.

Tests:

(equal (combine-elt-lst 1 '(1 2 3 4)) '((1 1) (1 2) (1 3) (1 4))) ;; Test simple

(equal (combine-elt-lst 1 NIL) NIL) ;; Test con lista vacía

(equal (combine-elt-lst '(1 2) '(5 6 7)) '(((1 2) 5) ((1 2) 6) ((1 2) 7))) ;; Test con lista como elemento

(equal (combine-elt-lst NIL '(2 3 4)) '((NIL 2) (NIL 3) (NIL 4))) ;; Test con NIL como elemento

Apartado b)

A continuación se pide realizar una función que realice el producto cartesiano de dos listas, esto es, combinar todos los elementos de una lista, con todos los elementos de la otra. Para ello solo hace falta llamar a la función anterior con cada elemento de la primera lista y la segunda lista.

Tests:

(equal (combine-lst-lst '(1 2) '(a b c)) '((1 a) (1 b) (1 c) (2 a) (2 b) (2 c)));; Test simple

(null (combine-lst-lst '(1 2) NIL)) ;; Test segunda lista vacía

(null (combine-lst-lst NIL '(a b c))) ;; Test primera lista vacía

(equal (combine-lst-lst NIL NIL) NIL) ;; Test ambas listas vacías

Apartado c)

En el último apartado se pide implementar una función que realice el producto cartesiano de un conjunto de listas que se pasan en una lista como argumento de la función. *******Añadir explicación*******

Tests:

(equal (combine-list-of-lsts '((1 2) (3 4) (5 6)))

'((1 3 5) (1 3 6) (1 4 5) (1 4 6) (2 3 5) (2 3 6) (2 4 5) (2 4 6)))

(equal (combine-list-of-lsts '((1 2) (a b) ("hola" "cosa")))

'((1 a "hola") (1 a "cosa") (1 b "hola") (1 b "cosa")

(2 a "hola") (2 a "cosa") (2 b "hola") (2 b "cosa"))) ;; Test simple

(null (combine-list-of-lsts '(NIL '(1 2) '(a b)))) ;; Test con primer argumento NIL

(null (combine-list-of-lsts '(NIL '(1 2) NIL))) ;; Test varios argumentos NIL

(null (combine-list-of-lsts '(NIL NIL NIL NIL))) ;; Test con todos argumentos NIL

Ejercicio 3:

Código:

```
(defun scalar-product (x y)
  (apply #'+ (mapcar #'* x y)))
  ;; Se multiplican las coordenadas de cada vector en orden y se suman los resultados

(defun euclidean-norm (x)
  (sqrt (apply #'+ (mapcar #'(lambda (x0) (* x0 x0)) x))))
  ;; Se elevan al cuadrado las coordenadas, se suman los resultados y se hace la raíz cuadrada

(defun euclidean-distance (x y)
  (euclidean-norm (mapcar #'- x y)))
  ;; Se calcula la norma del vector resta
```

Apartado a)

En primer lugar tenemos que programar una función que calcules el producto escalar de dos vectores. Para ello tiene que ir multiplicando coordenada a cordenada ambos vectores y después sumar los productos. La fórmula que realiza esta operación es:

$$x^T y = \sum_{d=1}^{D} x_d^2$$

Donde D es la dimensión del vector y x^T simboliza el vector traspuesto de x.

Tests:

```
(= (scalar-product '(1 2 3 4) '(1 1 2 1)) 13) ;; Test simple
(= (scalar-product '(1 2 3) '(0 0 0)) 0);; Test con vector 0
```

Apartado b)

La siguiente función tendrá que calcular la norma eulídea de un vector, que se calcula haciendo la raíz cuadrada de la suma de los cuadrados de las coordenadas del vector x, que viene dado por la fórmula:

$$||x|| = \sqrt{\sum_{d=1}^{D} x_d^2}$$

Otra forma de calcularlo utilizando la función anterior es haciendo la raíz cuadrada del producto escalar de x consigo mismo, pero creemos que de esta forma es menos eficiente porque se tendría que recorrer el vector x dos veces por tratarlo como listas distintas la aplicar la función mapcar.

Tests:

(= (euclidean-norm '(-1 2 -5)) (sqrt 30)) ;; Test Simple

(= (euclidean-norm '(0 0 0)) (sqrt 0)) ;; Test con vector 0

Apartado c)

Por último se pide implementar la distancia euclídea entre dos vectores, que no es más que la norma euclídea del vector diferencia de estos. Para ello, solo es necesario llamar a la función anterior con la resta de ambos vectores.

Tests:

(= (euclidean-distance '(1 2 3) '(2 4 -8)) (sqrt 126)) ;; Test Simple

(= (euclidean-distance '(1 1 1) '(1 1 1)) 0) ;; Test con vectores iguales

(= (euclidean-distance '(1 1 1) '(0 0 0)) (sqrt 3)) ;; Test con vector 0

Ejercicio 4:

Código:

```
(defun cosine-similarity (x y)
  (let ((scal-product (scalar-product x y)) ;; Se hacen cálculos previos
    (norm-product (* (euclidean-norm x) (euclidean-norm y))))
    (if (/= 0 norm-product) ;; Si el producto de las normas es 0, devolver NIL
      (/ scal-product norm-product))))


(defun angular-distance (x y)
  (let ((cos-sim (cosine-similarity x y))) ;; Se calcula la similitud del coseno
    (if cos-sim ;; Si no es NIL, se realiza la operación
      (/ (acos cos-sim) pi))))
```

Apartado a)

En este primer apartado hemos programado una función que calcula la similitud coseno entre dos vectores, que viene definida por el coseno del ángulo que forman estos:

$$\text{cosine\_similarity}(x, y) = \frac{x^T y}{||x|| ||y||}$$

Por su definición, esta solo tomará valores en el intervalo [-1, 1], siendo 1 si los vectores son proporcionales o -1 si son proporcionales al inverso.

Tests:

(< (abs (- (cosine-similarity '(1 2 3) '(1 2 3)) 1)) 1.0L-6) ;; Test vectores iguales

(< (abs (- (cosine-similarity '(1 2 3) '(-4.5 7 0)) 0.3051052626)) 1.0L-6) ;; Test simple

(null (cosine-similarity '(0 0 0) '(1 2 3))) ;; Test con vector 0

Apartado b)

A continuación implementamos la función distancia angular entre dos vectores, que utiliza la función anterior y calcula el ańgulo que forman los vectores y los divide después por pi:

$$\text{angular\_distance}(x, y) = \frac{\arccos(\text{cosine\_similarity}(x, y))}{\pi}$$

Por su definición, esta distancia estará en el intervalo [0, 1], siendo 0 si los vectores son proporcionales y 1 si son proporcionales al inverso.

Tests:

(< (abs (- (angular-distance '(1 2 3) '(1 2 3)) 0)) 1.0L-3) ;; Test vectores iguales

(< (abs (- (angular-distance '(1 2 3) '(-4.5 7 0)) 0.4013083507)) 1.0L-3) ;; Test simple

(null (angular-distance '(0 0 0) '(1 2 3))) ;; Test con vector 0

Ejercicio 5:

Código:

```
(defun select-vectors (lst-vectors test-vector similarity-fn &optional (threshold 0))
  (sort ;; Ordena los vectores según su similitud de mayor a menor
    (remove-if #'(lambda (x) (or (null x) (< (second x) threshold)));; Elimina los vectores cuya similitud es menor que el umbral
      (mapcar #'(lambda (lst)
        (let ((simil (funcall similarity-fn lst test-vector))) (unless (null simil) (list lst simil)))) lst-vectors))
#'(lambda (x y) ;; Calcula la similitud de cada vector
        (> (second x) (second y)))))
```

En este ejercicio se pide implementar una función muy simplificada de un algoritmo que sirve para categorizar textos en función de la frecuencia de las distintas palabras que aparecen en el texto. En este caso se utilizarán vectores cuyas coordenadas representarán las frecuencias de cada palabra en el texto, y se compararán con las frecuencias que suele tener cada tipo de texto. De esta forma, la categoría cuyo vetor sea “más parecido” al vector del texto que estamos testeando, se considerará que es la de este. Para determinar el “parecido” de los vectores se podrá utilizar cualquier función de similitud pasada como argumento. La funcion por tanto calculará la similitud del vector test al resto de vectores, se quedará con aquellos que superen un umbral que se podrá introducir de forma opcional y los mostrará ordenados de mayor a menor similitud.

Tests:

(equal (select-vectors '((-1 -1 -1) (-1 -1 1) (-1 1 1) (1 1 1))

'(1 1 1) #'cosine-similarity 0.2) (list (list '(1 1 1) '1.0) (list '(-1 1 1) '0.33333334))) ;; Test Simple

(equal (select-vectors nil '(1 1 1) #'cosine-similarity 0.2) nil) ;; Test Vectores a nil

(equal (select-vectors '((-1 -1 -1) (-1 -1 1) (-1 1 1) (1 1 1))

nil #'cosine-similarity 0.2) nil) ;; Test con Vector Comparador a nil

(equal (select-vectors '((-1 -1 -1) (-1 -1 1) (-1 1 1) (1 1 1))

'(0 0 0) #'cosine-similarity 0.2) nil) ;; Test con vetor nulo

Ejercicio 6:

Código:

```
(defun nearest-neighbor (lst-vectors test-vector distance-fn)
  (if (and lst-vectors test-vector) ; Comprueba que los argumentos no son nil
    (if (null (rest lst-vectors)) ; Hace el caso de que solo se pase un vector para comparar
      (list (first lst-vectors) (funcall distance-fn (first lst-vectors) test-vector)) ; devuelve la distancia entre el unico vector y el test
      (if (< (funcall distance-fn (first lst-vectors) test-vector) (funcall distance-fn (second lst-vectors) test-vector));comprueba si hay mayor distancia con el primer vector o con el segundo
        (nearest-neighbor (remove (second lst-vectors) lst-vectors) test-vector distance-fn); si es con el segundo lo borra de la lista de vectores y vuelve a llamar a la funcion sin ese vector
        (nearest-neighbor (remove (first lst-vectors) lst-vectors) test-vector distance-fn))))); si es con el primero igual
```

En este ejercicio se pide implementar una función que dada una lista de vectores, un vector de referencia, y una función de similitud, halle el vector mas parecido a la referencia que se da.

Se pide realizarla de forma recursiva, por lo que la la forma de realizarlo que hemos elegido consiste en comparar los dos primeros vectores de la lista con el vector test y nos quedamos con el más

“cercano”. A continuación volvemos a llamar a la función (recursiva) pasando como lista de vectores la lista inicial con el vector mas lejano de los dos comparados eliminado.

Tests:

(equal (nearest-neighbor '((1 2 3) (-1 2 5) (-2 -3 1)) '(1 1 1) #'angular-distance) '((1 2 3) 0.12337583)) ;; Test simple

(null (nearest-neighbor NIL '(1 2 3) #'angular-distance)) ;; Lista de vectores vacía

(null (nearest-neighbor '((1 2 3) (4 5 6)) NIL #'angular-distance)) ;; Vector test NIL

Ejercicio 7:

Código:

```
(defun backward-chaining (goal lst-rules)
  (backward-chaining-aux goal lst-rules NIL))

(defun backward-chaining-aux (goal lst-rules pending-goals)
  (if (member goal pending-goals) NIL ;; Si el goal está en la lista de pendientes se devuelve NIL
    (some #'(lambda (x) ;; Se comprueba si se puede inferir el goal a partir de alguna regla
      (cond ((null x) NIL) ((equal (second x) goal) ;; Comprueba que no es NIL y si el goal se puede inferir con la regla x
        (if (null (car x)) t ;; De ser así, si la condición es NIL, es decir, es un hecho, se devuelve true
        (every #'(lambda (y) ;; Si no, se llama a la función con cada átomo que conforma la regla
          (backward-chaining-aux y (remove x lst-rules) (cons goal pending-goals))) (car x))))))
      lst-rules)))
```

A continuación se propone un ejercicio que consiste en programar una función que determine si un determinado átomo se puede determinar a partir de una serie de reglas y hechos. Dichas reglas son de la forma:

$$\left(A \wedge B \wedge C \Rightarrow D\right)$$

Donde A, B, C y D son átomos. En Lisp, las reglas son representadas como una lista cuyo primer elemento es una lista con los requisitos y el segundo el átomo que implican, en nuestro ejemplo sería ((A B C) D). De la misma forma, el hecho A se representará como una implicación que no tiene requisitos, es decir, (NIL A).

Esta función se implementa de forma recursiva de forma que para cada átomo que se quiere inferir, en primer lugar se busca alguna regla o hecho que permita inferirlo. Si se encuentra un hehco, ya hemos terminado y si se encuentra una regla, se llamará de nuevo a la función para intentar inferir los

requisitos para inferir el objetivo, añadiendo el objetivo actual a una lista de objetivos pendientes. La función devolverá verdadero si consigue inferir el átomo objetivo, o falso si no. La lista de objetivos pendientes se utiliza para evitar entrar en bucles e intentar inferir un átomo que ya se está intentando inferir en llamadas previas a la función. Para evitar que al llamar a esta función haya que introducir como argumento esta lista ya que no es parte de la información del ejercicio si no una herramienta para realizarlo, dentro de la función solo se llamará a una función auxiliar con esta lista que será la que realice todo el proceso.

Tests

(backward-chaining 'Q '((NIL A) (NIL B) ((P) Q) ((L M) P) ((B L) M) ((A P) L) ((A B) L))) ;; Test que sí encuentra

(not (backward-chaining 1 '((NIL 3) (NIL 5) ((2 3) 4) ((5 6) 2) ((3 7) 6) (NIL 8)))) ;; Test que no encuentra

(not (backward-chaining 'Q NIL)) ;; Test con nil como lista

Ejercicio 8:

Código:

```
(defun bfs-improved (end queue net); misma implementacion, solo cambia que llama a la funcion new-paths-improved
  (if (null queue)
      NIL
    (let* ((path (first queue))
           (node (first path)))
      (if (eql node end)
          (reverse path)
        (bfs-improved end
            (append (rest queue)
                    (new-paths-improved path node net))
            net)))))


(defun new-paths-improved (path node net)
                (add-vecinos path (rest (assoc node net))));Llama add-vecinos


(defun add-vecinos (path vecinos);añade los vecinos
  (mapcar #'(lambda(x)
    (add-to-path path x)) vecinos))
```

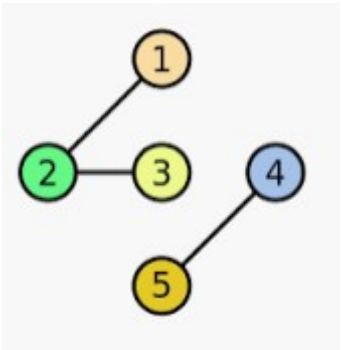
```
(defun add-to-path (path n)
  (if (null (member n path)); Solo cambia que introduce el nodo como vecino si este no se ha visitado antes para que no haya ciclos
    (cons n path)))


(defun shortest-path-improved (end queue net)
  (bfs-improved end (list(list start)) net)) ; misma implementacion que la dada
```

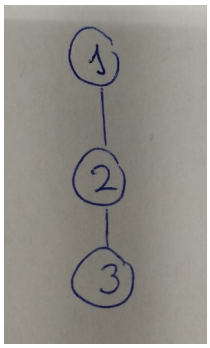
A) La función BFS recibe tres argumentos: end (el nodo final), queue (EL camino recorrido al que se van añadiendo los nodos que visitamos). Lo primero, se comprueba que el argumento *queue* no es *null* para que pueda empezar la búsqueda por un nodo (nodo raíz). A continuación define las variables locales *path* (que será el camino que va a seguir) y *node* (que será el nodo que está visitando).

Si el nodo a visitar es la meta (end) acabamos la búsqueda y devolvemos el camino dado la vuelta (*reverse path*). Si no volvemos a llamar a la función *bfs* (recursiva) con la misma meta (*end*), la cola (*queue*) añadiendo lo que devuelve la función *new-paths*, y la lista de adyacencia del grafo.

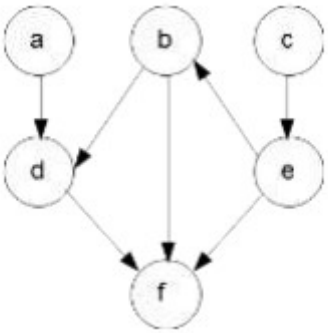
La funcion *new-paths* añade al camino los hijos del nodo que se está visitando usando la función *assoc,* que los busca en la lista de adyacencia y *rest* para no incluir el que se está visitando.

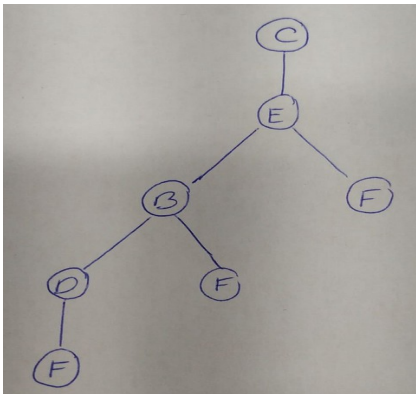
B) Grafo Especial, BFS desde el nodo 1: (grafo no conexo)



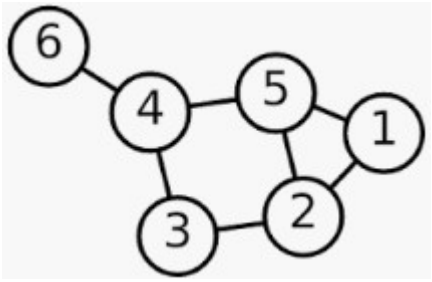


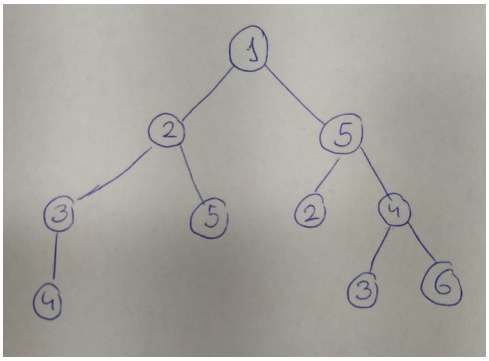
Grafo del ejemplo anterior, BFS desde el nodo C:





Grafo diferente, BFS desde el nodo 1:





C) Evaluamos las llamadas a BFS usando *trace:*

```
Break 4 [5]> (trace bfs)
;; Rastreando la función BFS.
(BFS)
Break 4 [5]> (shortest-path 'c 'd '((a d) (b d f) (c e) (d f) (e b f) (f)))
1. Trace: (BFS 'D '((C)) '((A D) (B D F) (C E) (D F) (E B F) (F)))
2. Trace: (BFS 'D '((E C)) '((A D) (B D F) (C E) (D F) (E B F) (F)))
3. Trace: (BFS 'D '((B E C) (F E C)) '((A D) (B D F) (C E) (D F) (E B F) (F)))
4. Trace:
(BFS 'D '((F E C) (D B E C) (F B E C))
 '((A D) (B D F) (C E) (D F) (E B F) (F)))
5. Trace: (BFS 'D '((D B E C) (F B E C)) '((A D) (B D F) (C E) (D F) (E B F) (F)))
5. Trace: BFS ==> (C E B D)
4. Trace: BFS ==> (C E B D)
3. Trace: BFS ==> (C E B D)
2. Trace: BFS ==> (C E B D)
1. Trace: BFS ==> (C E B D)
(C E B D)
```

Se busca el camino de C a D, por lo que llama a BFS con el nodo C como nodo raíz. LA siguiente llamada a *BFS*, recibirá la cola ((E, C)), por lo que el nodo que se explorará ahora es el nodo E. Ocurrirá lo mismo para la siguiente llamada siendo la cola que se le pasa a *BFS* ((B, E, C)), hasta que finalmente se alcanza D, que es el nodo objetivo y finaliza la búsqueda devolviendo el camino recorrido dado la vuelta: (C E B D).

D) Apartados de i. a vi. resueltos en apartado A. La función *sorthest-path* lo único que hace es llamar a la función *BFS*, pasando el nodo *start* como el único de *queue* (nodo raíz). Encuentra el camino más corto del nodo raíz al nodo *end* haciendo la búsqueda en anchura explicada anteriormente.

E)

```
método BFS(Grafo, cola, origen):
    cola_insertar(origen)
    origen = visitado

    mientras cola no vacía:
        v = cola_extraer(cola)

        para todo w hijo de v en el grafo:
            si w = no visitado:
                w = visitado
                cola_insertar(w)
```

F)

```
(defun bfs (end queue net)
  (if (null queue)
      NIL
    (let* ((path (first queue)) ;; En la cola incluye ya el nodo raiz
           (node (first path)))
      (if (eql node end)
          (reverse path)  ;; Cuando encuentre el nodo acaba
        (bfs end
             (append (rest queue)  ;;Añade a la cola solo los hijos no visitados
                     (new-paths path node net))  ;; llama a new paths que inserta los hijos no visitados
             net)))))

(defun new-paths (path node net)
  (mapcar #'(lambda(n)
        (cons n path))
          (rest (assoc node net)))) ;; Obtiene los hijos
```

G) (shortest-path 'A ' H '((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)))

H) Caso problematico: grafo con ciclos sin solución

(shortest-path 'C 'F '((A B C D E) (B A D E) (C A G) (D A B G H) (E A B G H) (F H) (G C D E H) (H D E G)))