Memoria Inteligencia Artificial (Práctica 1)

Miguel Arconada Manteca
miguel.arconada@estudiante.uam.es

David Cabornero Pascual
david.cabornero@estudiante.uam.es

8 de marzo de 2019

Ejercicio 1

Función que calcula la distancia entre dos vectores

Hemos desarrollado de manera análoga **cosine-distance-rec** y **cosine-distance-mapcar**, de forma que, para dividir en partes más comprensibles la función, también hemos definido sus respectivos **norma** y **producto-escalar**, funciones que son bastante autodescriptivas.

Gracias a la función **let** y a las funciones auxiliares creadas, hemos evitado la repetición de código que se exigía en el ejercicio.

Los resultados solicitados por los dos métodos son los siguientes:

```
(cosine-distance-rec '(1 2) '(1 2 3))
>0.40238577
(cosine-distance-mapcar '(1 2) '(1 2 3))
>0.40238577
(cosine-distance-rec nil '(1 2 3))
>1
(cosine-distance-mapcar nil '(1 2 3))
>1
(cosine-distance-rec '() '())
>0
(cosine-distance-mapcar '() '())
>0
(cosine-distance-rec '(0 0) '(0 0))
>0
(cosine-distance-mapcar '(0 0) '(0 0))
>0
```

Función que devuelve los vectores ordenados por confianza

Nuestra función básica es:

```
1 (defun order-vectors-cosine-distance
2     (vector lst-of-vectors &optional (confidence-level 0))
3   (ordenar-lista (filtrar lst-of-vectors vector confidence-level) vector))
```

Donde ahora tratamos las principales funciones:

- *filtrar*: Una simple recursión en la que se purgan los elementos que no alcanzan la confianza exigida
- *ordenar-lista*: Una recursión algo más sofisticada, en la que se van introduciendo los elementos uno a uno en una lista en la que, cada vez que se inserta un nuevo elemento, se hace insertándolo en el lugar adecuado de acuerdo con el orden por confianza.

Ahora se muestran los ejemplos requeridos:

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

Clasificador por distancia coseno

Lo que se nos pide en este ejercicio es una clara recursión utilizando lo realizado en el apartado anterior:

```
1 (defun get-vectors-category (categories texts distance-measure)
2   (if (null texts)
3       nil
4     (cons
5      (best-category categories (first texts) distance-measure)
6      (get-vectors-category categories (rest texts) distance-measure))))
```

Donde hay dos únicas funciones:

- *get-vectors-category*: Analiza cual es la categoría que mejor se adapta a cada uno de los textos recursivamente.
- *best-category*: Analiza recursivamente cual es la categoría que mejor se adapta a un texto concreto, de forma que cada vez que compara una categoría y un texto, mira a ver si es más pequeña la nueva distancia o el mínimo de las anteriormente miradas.

Aquí se encuentran los ejemplos que se piden comprobar (hay un error en el enunciado, la función es get-vectors-category, no categories):

```
(get-vectors-category '(()) '(()) #'cosine-distance-rec)
>((NIL 0))
(get-vectors-category '(()) '(()) #'cosine-distance-mapcar)
>((NIL 0))
(get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-rec)
>((NIL 1) (NIL 1))
(get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-mapcar)
>((NIL 1) (NIL 1))
(get-vectors-category '(()) '((1 1 2 3) (2 4 5 6)) #'cosine-distance-rec)
>((NIL 1) (NIL 1))
(get-vectors-category '(()) '((1 1 2 3) (2 4 5 6)) #'cosine-distance-mapcar)
>((NIL 1) (NIL 1))
```

Los tiempos son demasiado pequeños como para poder ser medidos por *time*, ya que en ambos casos son idénticamente 0.

Ejercicio 2

Función que encuentra una raíz

Se adjunta la función pedida a continuación, donde se realiza una simple recursión en la que se realiza el algoritmo del método de Newton:

```
1 (defun newton (f df max-iter x0 &optional (tol 0.001))
2   (cond ((<= tol 0) NIL)
3         ((< max-iter 0) NIL)
4         ((= max-iter 0) x0)
5         (T
6          (ultima-iteracion f df (newton-rec f df (- max-iter 1) x0) tol))))
```

A continuación se explican las funciones indicadas:

- *newton*: Función principal, mira casos base y errores
- *newton-rec*: Hace el método de Newton recursivamente. Las operaciones matemáticas de cada iteración las hace *iteracion*
- *ultima-iteracion*: Realiza lo mismo que *newton-rec*, pero solo realiza una iteración y se percata de si el resultado pedido cumple el criterio de tolerancia

Se adjuntan algunos ejemplos interesantes ya resueltos en el enunciado:

```
(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 3.0)
>4.0
(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 30 -2.5)
>-3.0
```

Función que encuentra una semilla para la cual Newton converja

Esta función es extremadamente simple, únicamente hay que hacer una recursión hasta encontrar una semilla que no de NIL. En caso de que se acabe la lista de semillas, se devuelve NIL. Se adjunta a continuación el código de la función principal:

```
1 (defun one-root-newton (f df max-iter semillas &optional (tol 0.001))
2   (if (null semillas)
3       NIL
4     (let ((nx (newton f df max-iter (first semillas) tol)))
5       (if (null nx)
6           (one-root-newton f df max-iter (rest semillas) tol)
7         nx))))
```

Se adjuntan de nuevo ejemplos ejecutados por nuestro programa:

```
(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))
>1.0
(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(3.0 -2.5))
>NIL
```

Función que devuelve la raíz encontrada para cada semilla

De nuevo muy simple, aún más incluso. Únicamente hay que crear una lista con las raices encontradas por cada semilla. Se adjunta el código a continuación:

```
1 (defun all-roots-newton (f df max-iter semillas &optional (tol 0.001))
2   (if (null semillas)
3       NIL
4     (cons (newton f df max-iter (first semillas) tol)
5           (all-roots-newton f df max-iter (rest semillas) tol))))
```

Adjuntamos a continuación de nuevo unos descriptivos ejemplos:

```
(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))
>(1.0 4.0 -3.0)
(all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 10000))
>(LISP no termina, por culpa del propio intérprete. Trabaja con fracciones enormes aunque
los números no son demasiado altos)
```

Prueba con distintas raíces y solo se queda con las que converjan

Programa muy similar al anterior, en el que evitamos que nos salgan los NIL producidos por las semillas que no convergían gracias al mapcan. A continuación se muestra el código:

```
1 (defun list-not-nil-roots-newton (f df max-iter semillas &optional (tol 0.001))
2   (mapcan (lambda (x) (if (null x) NIL (list x)))
3           (all-roots-newton f df max-iter semillas tol)))
```

A continuación se muestra de nuevo un ejemplo de prueba:

```
(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
```

Ejercicio 3

Combina elemento con lista

Nuestra función es:

```
1  (defun combine-elt-lst (elt lst)
2    (cond
3     ((and (null elt) (null lst)) nil)
4     ((null lst) nil)
5     ((null elt) (cons
6                   (cons (first lst) nil)
7                   (combine-elt-lst elt (rest lst))))
8     (T (cons
9         (cons elt (cons (first lst) nil))
10        (combine-elt-lst elt (rest lst))))))
```

El caso base será cuando reciba la lista vacía, pero comprueba si el elemento es NIL también, por lo tanto, las salidas preguntadas en el enunciado son:

```
(combine-elt-lst 'a nil)
>NIL
(combine-elt-lst nil nil)
>NIL
(combine-elt-lst nil '(a b))
>((A) (B))
```

Combina dos listas

Para este apartado hemos programado la siguiente función:

```
1  (defun combine-lst-lst (lst1 lst2)
2    (cond
3     ((and (null lst1) (null lst2)) nil)
4     ((null lst1) nil)
5     (T (append (combine-elt-lst (first lst1) lst2)
6                (combine-lst-lst (rest lst1) lst2)))))
```

Esta función llama a la función del apartado anterior.

Los casos del enunciado evalúan a:

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

Todos los casos anteriores son casos baseen nuestra función o en nuestra función auxiliar.

Combina lista de listas

Nuestra función para este apartado es la siguiente:

```
1 (defun combine-list-of-lsts (lstoflsts)
2   (cond
3    ((null lstoflsts) nil)
4    ((null (first lstoflsts)) '(nil))
5    (T (combine-lst-with-list-of-cons (first lstoflsts) (combine-list-of-lsts (rest lsto
```

Esta función llama a la función auxiliar *combine-lst-with-list-of-cons*, que combina una lista con una lista de listas, y combina la lista con cada una de las listas de la lista final. Esta función es:

```
1 (defun combine-lst-with-list-of-cons (list list-of-cons)
2   (cond
3    ((null list) nil)
4    ((null (first list-of-cons))
5     (cons (cons (first list) nil)
6           (combine-lst-with-list-of-cons (rest list) list-of-cons)))
7    (T (append
8        (combine-elt-cons (first list) list-of-cons)
9        (combine-lst-with-list-of-cons (rest list) list-of-cons)))))
```

Esta función llama a su vez a más funciones auxiliares, pero no las explicamos por ser muy sencillas.

Los casos de prueba del enunciado para esta función evalúan a:

```
(combine-list-of-lsts '(() (+ -) (1 2 3 4)))
>(NIL)
(combine-list-of-lsts '((a b c) () (1 2 3 4)))
>((A) (B) (C))
(combine-list-of-lsts '((a b c) (1 2 3 4) ()))
>((A 1) (A 2) (A 3) (A 4) (B 1) (B 2) (B 3) (B 4) (C 1) (C 2) ...)
(combine-list-of-lsts '((1 2 3 4)))
>((1) (2) (3) (4))
(combine-list-of-lsts '(nil))
>(NIL)
(combine-list-of-lsts nil)
>NIL
```

Vemos que es la salida esperada, puesto que va recorriendo la lista hasta que se encuentra $'(nil)$, que es un caso base

Ejercicio 4

Funciones que aplican las reglas de derivación

Para este apartado desarrollamos varias funciones, que se encargan de aplicar fórmulas concretas para ir desarrollando la expresión hasta tener una que cumpla las reglas necesarias para construir el árbol de verdad:

- *negar*: Se encarga de desarrollar la expresión hasta que las únicas negaciones que aparecen son sobre átomos, no sobre expresiones. Para ello, hace uso de las reglas de De-Morgan.
- *desarrollar-bicond* y *desarrollar-cond*: Estas funciones "deshacen" cada condicional o bicondicional en expresiones unidas por *or*s o *and*s. Se llama primero la correspondiente a los bicondicionales, ya que los desarrolla de la siguiente forma:

  $$A <=> B \equiv (A => B) \wedge (!A =>!B)$$

  Estas ecuaciones parten de que el primer símbolo de la expresión es aquél que se desea expandir.
- *sintetizar*: Esta función se encarga de ir recorriendo la expresión inicial, y cada vez que se encuentra con un símbolo que desea modificar, invoca a las funciones *sintetizar-not*, *sintetizar-bicond* o *sintetizar-cond*, que son las que se encargan de rehacer la expresión incorrecta.

Como la función de *sintetizar* se encarga de recorrer la expresión, dentro de las otras funciones no comprobamos que, efectivamente, la expresión es incorrecta, ya que solo son invocadas por esa función, y confiamos que las llama con expresiones correctas.

Árbol de verdad

Nuestra función principal es la siguiente:

```
1 (defun truth-tree (expresion)
2   (lista-satisfacible (expand-truth-tree-aux (sintetizar expresion))))
```

Esta función simplemente se encarga de, lo primero, "arreglar" la cadena hasta el formato deseado. Después, llama a la función *expand-truth-tree-aux*, que recibe una expresión, y devuelve una lista de todos los nodos hoja del árbol de verdad correspondiente a la expresión. Por último, llama a la función *lista-satisfacible* sobre este resultado, que se encarga de comprobar los nodos, buscando uno que sea SAT, para deducir si la expresión es SAT o UNSAT.

Nuestra función *expand-truth-tree-aux* es la más compleja, ya que se encarga, de forma recursiva, de analizar el tipo de expresión que recibe (unión de *and*'s, *or*'s, átomo, etc), y llamar a las funciones auxiliares correspondientes.

Esta función asume que se le pasa como argumento una expresión con formato correcto (empieza por un *and* y solo contiene *and*s, *or*s y átomos negados), pues siempre la llamamos después de "sintetizarla".

Esta función hace uso de las funciones que nos fueron dadas en el enunciado, como *negative-literal-p*, etc, que comprueban el tipo de cada elemento de la expresión, así como de funciones auxiliares de elaboración propia, como *combinar-listas-and* o *combinar-listas-or*, que combinan las listas de nodos hoja de dos expresiones menores, si estas están unidas por cada uno de los operadores en la expresión global.

Funciones que comprueban la lista de hojas del árbol de verdad de la expresión:

Para terminar, desarrollamos un conjunto de funciones que se encargan de analizar la lista de nodos hoja del árbol de verdad, para encontrar alguno que sea SAT. La función general para esta

finalidad es la siguiente:

```
1 (defun lista-satisfacible (lista)
2   (cond
3     ((null lista) T)
4     ((par-satisfacible (first lista))
5      (lista-satisfacible (rest lista)))
6     (T nil)))
```

Preguntas del enunciado

Pregunta 1: Si en lugar de (∧ A (v B C)) tuviésemos (∧ A (! A) (v B C)), ¿qué sucedería?

Si llamamos a la función que hemos desarrollado en la práctica con esta expresión obtenemos:

```
(truth-tree '(∧ A (! A) (v B C)))
>NIL
```

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

Volvemos a llamar a la función con esta entrada:

```
(truth-tree '(∧ A (v B C) (! A)))
>NIL
```

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

Viendo los retornos de la función con cada entrada, y las llamadas recursivas que hace, deducimos que devuelve los nodos raíz de la expresión que se le pida analizar, en su caso teniendo en cuenta los nodos ya analizados (primer argumento)

Una decisión que tomamos en este ejercicio fue que, para facilitar un poco el problema, nuestra función genera el árbol completo, sin "podar" aquellas ramas que antes presenten una contradicción. Esto hace que nuestro programa se ejecute un poco más lento, pero en cambio, los pasos intermedios del algoritmo no tienen en cuenta las contradicciones, por lo que son más sencillos de programar.

También decidimos representar cada nodo como un par de listas, una con los átomos positivos y otra con los negativos, para así hacer más fácil la comprobación de si un átomo es UNSAT, reduciéndose a comprobar si algún átomo está en las dos listas.

Si llamamos a la función con la misma entrada que en el enunciado y comprobamos el trace, obtenemos lo siguiente:

```
1 >> (truth-tree '(=> A (^ B (! A))))
```

```
 0[6]: (EXPAND-TRUTH-TREE-AUX (^ (^ (V (! A) (^ B #)))))
   1[6]: (EXPAND-TRUTH-TREE-AUX (^ (! A)))
   1[6]: returned ((NIL (A)))
   1[6]: (EXPAND-TRUTH-TREE-AUX (^ (V (^ B (! A)))))
     2[6]: (EXPAND-TRUTH-TREE-AUX (^ (^ B (! A))))
       3[6]: (EXPAND-TRUTH-TREE-AUX (^ B))
       3[6]: returned (((B) NIL))
       3[6]: (EXPAND-TRUTH-TREE-AUX (^ (! A)))
       3[6]: returned ((NIL (A)))
     2[6]: returned (((B) (A)))
     2[6]: (EXPAND-TRUTH-TREE-AUX (^ (V)))
     2[6]: returned ((NIL NIL))
   1[6]: returned (((B) (A)))
 0[6]: returned ((NIL (A)) ((B) (A)))
T
```

Se puede ver que es una llamada recursiva correcta. No es igual a la del enunciado, pues nosotros codificamos los retornos de la función de forma diferente, y, además, no hacemos poda intermedia.

Ejercicio 5

Ejemplos ilustrativos

Grafos especiales

- Grafo vacío: En este caso, el árbol resultante sería vacío
- Grafo de un solo elemento: El árbol resultante sería solo de ese elemento.
- Grafo no conexo: El árbol no puede contener nodos de partes no conexas del grafo, pues no existe un camino entre ellas.

Caso típico (ejemplo)

El grafo del ejemplo es:

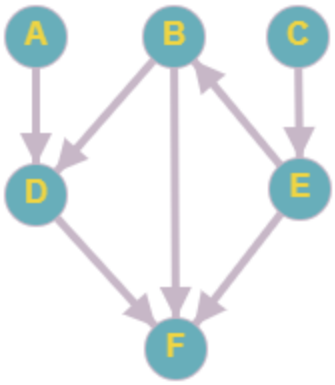

Figura 1: Grafo del enunciado

Si hacemos el algoritmo para cada nodo inicial obtenemos:

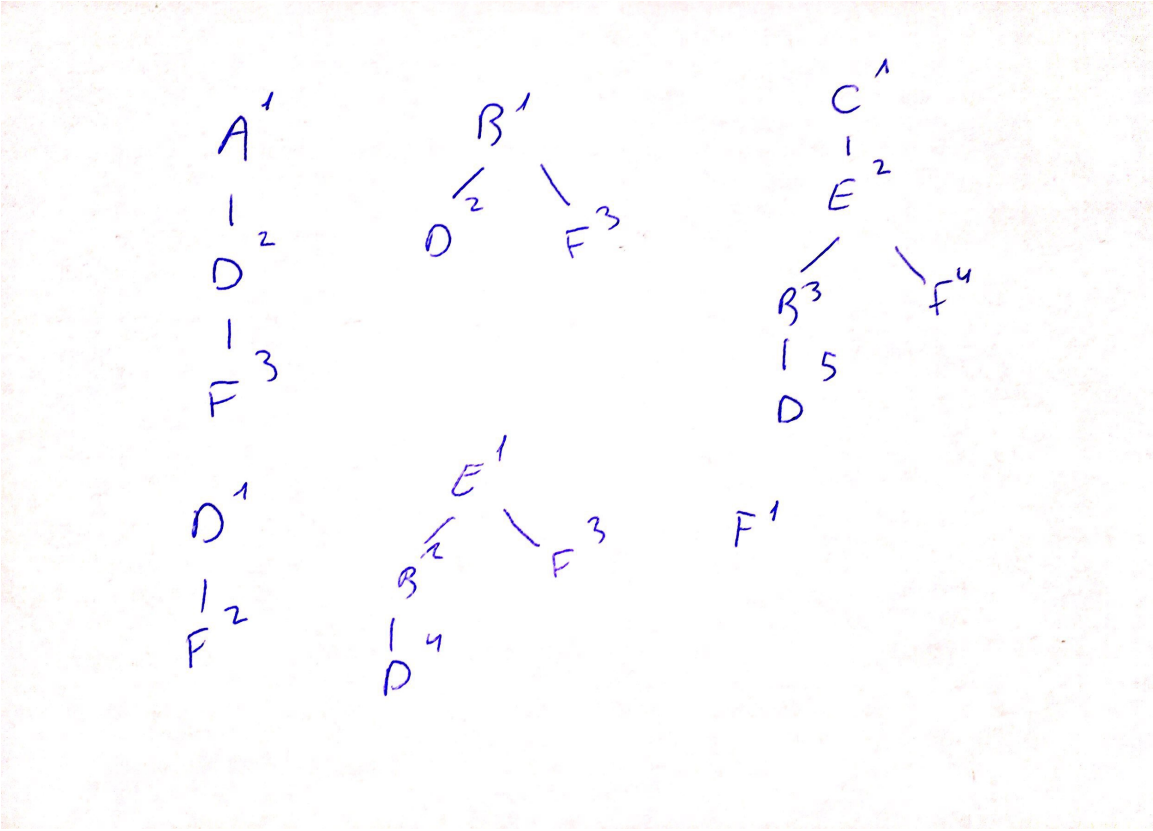

Figura 2: Árboles para cada nodo de inicio

Caso típico (distinto ejemplo)

Con grafos especiales asumimos que nos estamos refiriendo a grafos con un nodo o ninguno, y en ambos casos el algoritmo es el trivial.

Vamos a utilizar este grafo como grafo dirigido:

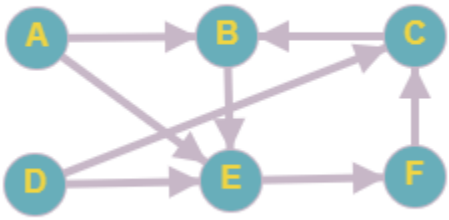

Figura 3: Grafo de ejemplo

Si desarrollamos el árbol para varios nodos iniciales obtenemos los siguientes resultados:

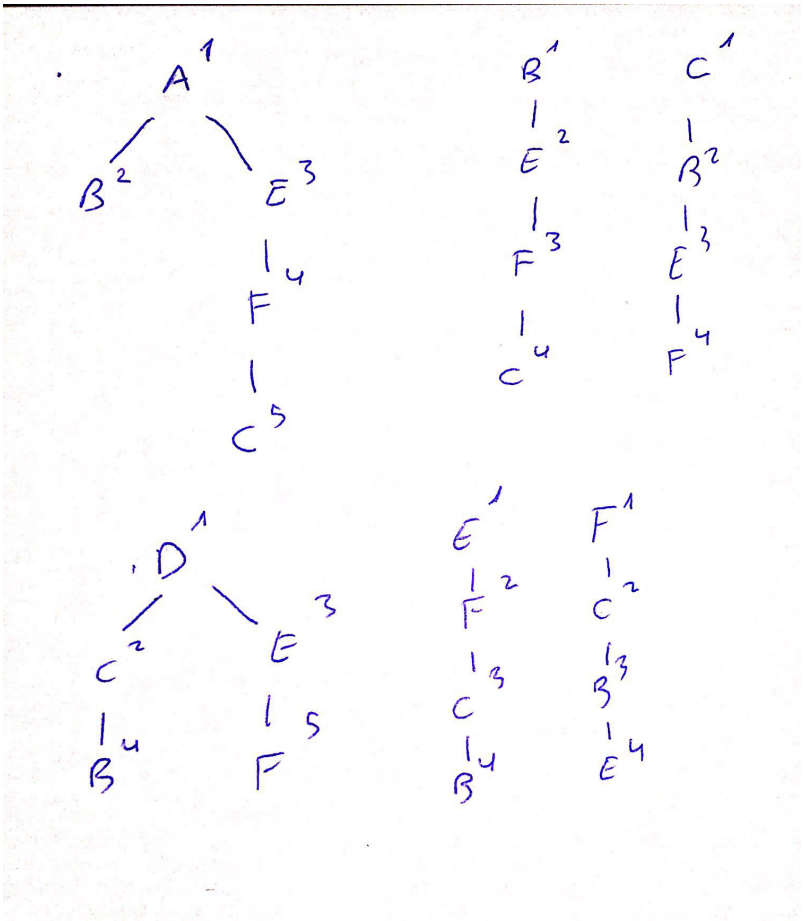

Figura 4: Árboles para cada nodo de inicio

Pseudocódigo

```
BFS(grafo graph, nodo source, nodo destino):
    Queue queue = crearCola()
    for vertice in vertices(graph):
        visitado(vertice) = False
    visitado(source) = True
    add_queue(queue, source)
    padre(source) = null
    while (is_empty(queue) == False):
        vertice = extract(queue)
        for new in adyacentes(vertice):
            if(visitado(new) == False):
                visitado(new) = True
                padre(new) = vertice
                if(new == destino):
                    return path(new) ;;Devuelve el camino de padres
                add_queue(queue,new)
    return null
```

Interpretación del código

```
(defun bfs (end queue net)
  (if (null queue) '()  ;;No se ha encontrado el nodo por BFS
    (let* ((path (first queue)) ;;Path es el camino de nodos padre del nodo
           (node (first path))) ;;en cuesti n , que es node
      (if (eql node end) ;;Si encuentra el camino devuelve los
          (reverse path) ;;sucesivos padres
        (bfs end    ;;Siguiente iteraci n del for, se a aden los nodos hijo
                    ;;a los caminos posibles
             (append (rest queue)
                     (new-paths path node net))
             net)))))

(defun new-paths (path node net) ;;Crea tantos caminos nuevos como nodos
  (mapcar #'(lambda (n)          ;;hijo haya respectoa al nodo, juntando
              (cons n path))     ;;el camino de antes con cada nuevo nodo
    (rest (assoc node net))))
```

Implementación de Graham

Podemos ejecutar la función con grafos sencillo, para ver su funcionamiento:

Si tomamos el grafo ((a b) (b c) (b d)), que se puede ver en la siguiente imagen

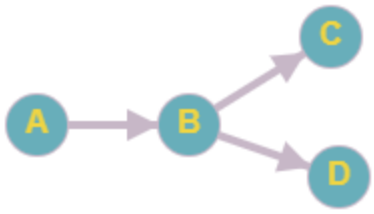

Figura 5: Grafo de ejemplo

```
(bfs 'c '((a)) '((a b) (b c) (b d)))
>(A B C)
(bfs 'c '((d)) '((a b) (b c) (b d)))
>NIL
(bfs 'c '(nil) '((a b) (b c) (b d)))
>NIL
```

Podemos ver a partir de estos ejemplos, que, a partir de una lista de rutas ya descubiertas, el algoritmo expande el árbol de búsqueda hasta el nodo end, según el grafo de la entrada. Cuando llega a end, devuelve la ruta que le ha llevado hasta él.

Ruta más corta

Esta función lo que hace es llamar a la función que expande el nodo, de forma que parte de la lista solo con el nodo inicial (lo obliga a ramificarse de él) y hasta el nodo final, de modo que cuando haya una ruta, al ser de coste uniforme, es la más "rápida".

Secuencia de llamadas

Si hacemos uso de la macro *trace* para la llamada `(shortest-path 'a 'f '((a d) (b d f) (c e) (d f) (e b f) (f)))` obtenemos la siguiente salida:

```
1  0[6]: (SHORTEST-PATH A F ((A D) (B D F) (C E) (D F) (E B F)))
2    1[6]: (BFS F ((A)) ((A D) (B D F) (C E) (D F) (E B F)))
3    1[6]: returned (A D F)
4  0[6]: returned (A D F)
5  (A D F)
```

Sin embargo, esta no es toda la recursividad de esta llamada, pues *bfs* es una función recursiva, y no hace solo una llamada. No sabemos por qué ocurre esto, pero nos pasa en los dos intérpretes que usamos: Allegro y LispWorks.

Para ver esta recursividad, lo que hacemos es ir ejecutando nosotros el código manualmente viendo las llamadas recursivas. De esta manera, podemos ver que son llamadas correctas, y se resuelve el problema de la forma que muestra la salida de esta ejecución.

Grafo no dirigido

Para hallar la ruta más corta entre B y G en ese grafo, debo llamar a:
`(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)))`

que representa el nodo anterior (representando cada arista no dirigida como un par de aristas dirigidas entre dos nodos, una en cada sentido)

El resultado que obtengo cuando ejecuto esto es:

```
 (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)))
>(B D G)
```

Recursión infinita

El código anterior presenta problemas cuando hay ciclos en el grafo, y no hay solución para el problema, como por ejemplo en el grafo siguiente si se quiere ir de A a C.

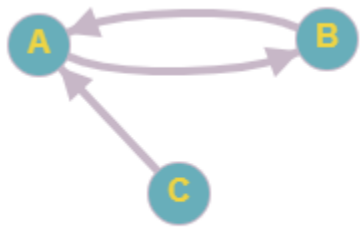

Figura 6: Grafo con recursión infinita

En este caso, el árbol de BFS quedaría de la siguiente forma:

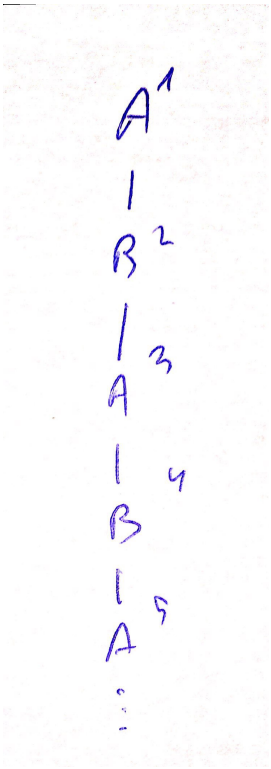

Figura 7: Árbol infinito

La solución que implementamos nosotros es dar un caso base al algoritmo, mediante el uso de una lista de nodos visitados. Usando esta herramienta, no puede entrar en recursión infinita, ya que una vez que visita un nodo, sus hijos no van a cambiar, por lo que se marca como visitado, y no se vuelve a expandir. Si no tiene solución el algoritmo, llegará un momento que la lista de nodos a visitar esté vacía, lo que implica que no hay solución posible.