Memoria P1 IA
Salvador Martín Barcia, Blanca Mercado Molero

Las funciones están comentadas explicando que hace cada parte.
1.
Newton

```
(defun newton (f df-dx max-iter x0 &optional (tol-abs 0.0001))
   "Zero of a function using the Newton-Raphson method


    INPUT:  f:       function whose zero we wish to find
          df-dx:    derivative of f
          max-iter: maximum number of iterations
          x0:       initial estimation of the zero (seed)
          tol-abs:  tolerance for convergence


    OUTPUT: estimation of the zero of f, NIL if not converged"


          (let ((xN (- x0 (/ (funcall f x0) (funcall df-dx x0)))));se asigna a xN para que sea
mas legible
            (cond ((<= max-iter 0);una condicion de salida
                   nil)
                  ((<;la condicion de salida si encuentra el valor
                    (abs (- 0 (/ (funcall f x0) (funcall df-dx x0))))
                    tol-abs)
                     xN)
                  ((>=;el caso base
                    (abs (- 0 (/ (funcall f x0) (funcall df-dx x0))))
                    tol-abs)
                     (newton f df-dx (- max-iter 1) xN tol-abs)))))
```

Newton-all

```
(defun newton-all (f df-dx max-iter seeds &optional (tol-abs 0.0001))
 "Zeros of a function using the Newton-Raphson method


  INPUT:  f:       function whose zero we wish to find
        df-dx:    derivative of f
        max-iter: maximum number of iterations
        seeds:    list of initial estimations of the zeros
        tol-abs:  tolerance for convergence


  OUTPUT: list of estimations of the zeros of f"
```

```
        (cond
         ((null seeds) nil));control de errores
         (mapcar ;llamamos a newton para todas las semillas
          #'(lambda (x0)
            (newton f df-dx max-iter x0 tol-abs))
         seeds))
```

2.

combine-elt-lst

```
(defun combine-elt-lst (elt lst)
  "Combines an element with all the elements of a list


   INPUT:  elt: element
         lst: list


   OUTPUT: list of pairs, such that
           the first element of the pair is elt.
           the second element is an element from lst"

          (if       (null lst)
           nil
           (append ;combinamos el elemento con el primer elemento de lst
     (list (list elt (first lst)))
     (combine-elt-lst elt (rest lst)))));solo pasamos el rest de lst
```

3.

combine-lst-lst

```
(defun combine-lst-lst (lst1 lst2)
  "Calculates the cartesian product of two vectors


   INPUT:  lst1: lst
         lst2: lst


   OUTPUT: list of pairs, such that
           the first element is an element from lst1.
           the second element is an element from lst2"

    (if    (null lst1)
           nil
     (append
       (combine-elt-lst(first lst1) lst2);combinamos el primero de la primera lista con la
funcion anterior
       (combine-lst-lst(rest lst1) lst2))));llamamos a esta funcion quitando el primero de
lst1
```

combine-list-of-lsts-aux

```
(defun combine-list-of-lsts-aux (lst1 all)
     (if (null all)
       lst1
       (combine-list-of-lsts-aux;primer argumento las listas enlazadas lst1 y first de all
        (mapcan
         #'(lambda (x1)
           (mapcar
             #'(lambda (x2)
              (if (listp x1)
                (append x1 (list x2))
                (list x1 x2)))
            (first all)))
        lst1)
        (rest all)));segundo argumento solo el rest de all
 )
```

combine-list-of-lsts

```
(defun combine-list-of-lsts (lstolsts)
   "Combinations of N elements, each of wich


    INPUT:  lstolsts: list of N sublists (list1 ... listN)


    OUTPUT: list of sublists of N elements, such that in each
          sublist the first element is from list1
               the second element is from list 2
               ...
               the Nth element is from list N"
        (if (null lstolsts)
          nil
        (combine-list-of-lsts-aux
          (first lstolsts)
          (rest lstolsts))));llamamos a la funcion aux
```

4.

scalar-product

```
(defun scalar-product (x y)
  "Calculates the scalar product of two vectors

  INPUT:  x: vector, represented as a list
        y: vector, represented as a list

  OUTPUT: scalar product between x and y


  NOTES:
     * Implemented with mapcar"
```

```
      (cond
       ((null x) nil)
       ((null y) nil)
       (T (apply #'+ ;sumamos todos los elementos de la lista
        (mapcar #'* x y)))));multiplicamos cada xn de la lista por yn
```

euclidean-norm

```
(defun euclidean-norm (x)
 "Calculates the euclidean (l2) norm of a vector

   INPUT:  x: vector, represented as a list


   OUTPUT: euclidean norm of x"
   (if (null x)
    nil
    (sqrt (scalar-product x x))));raiz cuadrada de producto escalar
```

euclidean-distance

```
(defun euclidean-distance (x y)
 "Calculates the euclidean (l2) distance between two vectors

   INPUT: x: vector, represented as a list
        y: vector, represented as a list


   OUTPUT: euclidean distance between x and y"
   (cond
    ((null x) nil)
    ((null y) nil)
    (T (euclidean-norm
        (mapcar #'- x y)))));la norma de la resta de cada elemento de x y
```

5.

select-vectors

```
(defun select-vectors (lst-vectors test-vector similarity-fn &optional (threshold 0))
   "Selects from a list the vectors whose similarity to a
    test vector is above a specified threshold.
    The resulting list is ordered according to this similarity.

    INPUT:  lst-vectors:   list of vectors
          test-vector:   test vector, representad as a list
          similarity-fn: reference to a similarity function
          threshold:     similarity threshold (default 0)

    OUTPUT: list of pairs. Each pair is a list with
          a vector and a similarity score.
          The vectors are such that their similarity to the
          test vector is above the specified threshold.
          The list is ordered from larger to smaller
```

```
        values of the similarity score

    NOTES:
      * Uses remove-if and sort"

      (cond
       ((null lst-vectors) nil)
       ((null test-vector) nil)
       (t (sort;ordenamos por el segundo elemento del par
       (remove-if;borramos los que sean menores que el corte
        #'(lambda (x1) (< (second x1) threshold))
         (mapcar #'(lambda (x0)
           (list x0 (funcall similarity-fn test-vector x0)))
          lst-vectors))
        #'> :key #'second))))
```

6.

nearest-neighbor

```
(defun nearest-neighbor (lst-vectors test-vector distance-fn)
  "Selects from a list the vector that is closest to the
   reference vector according to the specified distance function

   INPUT:  lst-vectors:   list of vectors
         ref-vector:    reference vector, represented as a list
         distance-fn:   reference to a distance function

   OUTPUT: List formed by two elements:
         (1) the vector that is closest to the reference vector
            according to the specified distance function
         (2) The corresponding distance value.


   NOTES:
     * The implementation is recursive
     * It ignores the vectors in lst-vectors for which the
       distance value cannot be computed."

      (cond
       ((null lst-vectors) nil)
       ((null test-vector) nil)
       (t (let
        ((x0;asignamos el primer par
          (list (first lst-vectors)
            (funcall distance-fn test-vector (first lst-vectors)))))
        (if (null (rest lst-vectors))
          x0;devolvemos ese par si no hay mas (caso base)
          (first;si no el mas pequeño de este y el de la siguiente recursion
            (sort
```

```
            (list
              x0
              (nearest-neighbor
                (rest lst-vectors)
                test-vector
                distance-fn))
           #'< :key #'second)))))))
```

7.

backward-chaining-aux

```
(defun backward-chaining-aux(goal lst-rules pending-goals)

    (some;miramos si se cumple para alguna de las reglas
      #'(lambda (x0) (not (null x0)))
      (mapcar #'(lambda (rule)
        (when (eq goal (second rule))
          (every;miramos si se cumple para todas los literales positivos de una regla
            #'(lambda (x1) (not (null x1)))
            (mapcar #'(lambda (go);llamamos para cada literal de una regla
              (backward-chaining-aux
                go
                (remove rule lst-rules :test #'equal)
                (if (null pending-goals)
                  (list goal)
                  (cons goal pending-goals))))
              (first rule)))))
        lst-rules)))
```

backward-chaining

```
(defun backward-chaining (goal lst-rules)
  "Backward-chaining algorithm for propositional logic

   INPUT: goal:      symbol that represents the goal
          lst-rules: list of pairs of the form
                     (<antecedent>  <consequent>)
                     where <antecedent> is a list of symbols
                     and  <consequent> is a symbol


   OUTPUT: T (goal derived) or NIL (goal cannot be derived)


   NOTES:
      * Implemented with some, every"


      (if (null lst-rules)
        nil
        (backward-chaining-aux goal lst-rules NIL)));llamamos a la aux
```

8.

bfs-improved

```
(defun bfs-improved (end queue net)
  (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 path node net))
              (remove (assoc node net) net  :test #'equal))))));borramos el
               ;que se acaba de utilizar para elimiar bucles infinitos
```

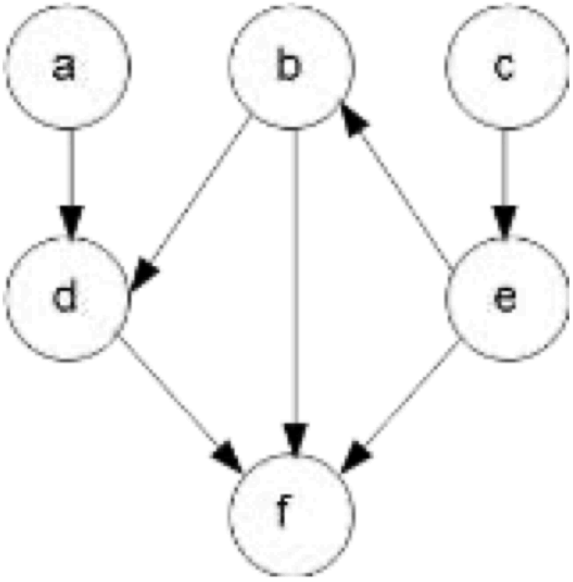
shortest-path-improved

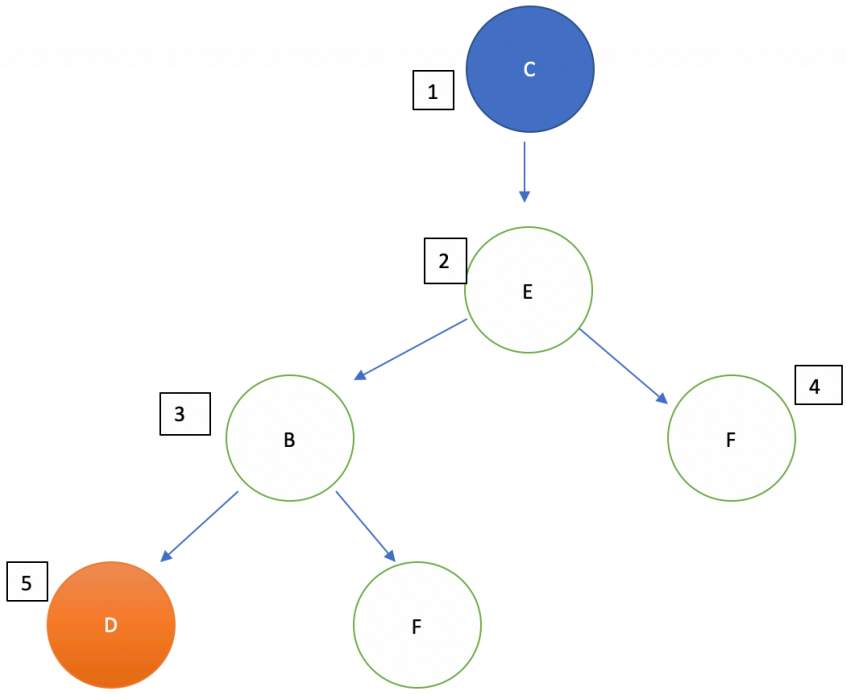
```
(defun shortest-path-improved (start end net)
  (bfs-improved end (list (list start)) net))
```

a)

La función *bfs* lo primero que hace es comprobar el caso de salida de la recursividad, sale en el caso de que no haya comprobado todos los nodos desde el primero y no encuentre la solución. Lo siguiente que hace es guardar en la variable *path* el siguiente path que se va a analizar que es una lista en la cual el primer elemento es el siguiente nodo para analizar, este también lo guarda en *node*. Comprueba si *node* es el objetivo en cuyo caso devuelve el reverse del *path* para indicar cuales son los nodos a seguir para encontrar la solución, en caso de que no lo sea llama otra vez a la función cambiando los parámetros de manera que, el nodo final sigue siendo el mismo, la lista de paths va a ser los que nos quedan por analizar de ese nivel, seguido por los nodos del siguiente nivel que vamos a analizar, es importante que estén en ese orden ya que es una búsqueda en anchura, **de esta tarea se encarga new-paths**, y por último le pasa el mismo árbol de la anterior recursión.

b) Empieza en C y acaba en D:





Los números indican en qué orden se han explorado los nodos, el nodo azul es el nodo inicial y el naranja el final.

c)

```
CL-USER> (trace bfs)
WARNING: BFS is already TRACE'd, untracing it first.
(BFS)
CL-USER> (shortest-path 'c 'd '((a d) (b d f) (c e) (d f) (e b f) (f)))
  0: (BFS D ((C)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
    1: (BFS D ((E C)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
      2: (BFS D ((B E C) (F E C)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
        3: (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)))
          4: (BFS D ((D B E C) (F B E C)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
          4: BFS returned (C E B D)
        3: BFS returned (C E B D)
      2: BFS returned (C E B D)
    1: BFS returned (C E B D)
  0: BFS returned (C E B D)
(C E B D)
```

d)

i. end es el nodo final de la búsqueda.
ii. queue es una lista de listas que contienen el camino a llegar desde el nodo mas profundo (también siendo el nodo que se va a evaluar en cada caso) hasta el principio del árbol
iii. net es la lista de adyacencia del árbol de búsqueda.
iv. path es la siguiente rama del árbol que se va a examinar.
v. node es el nodo que se va a evaluar de la rama que toca en ese momento de la recursión.
vi. new-paths es la función que sacara los siguientes paths que saldrán del nodo que se esta evaluando en ese momento.
vii. shortest-path realiza un algoritmo de búsqueda en anchura, y para un árbol con los mismos valores en cada arista este algoritmo encuentra el camino mas corto. En concreto esta lista de listas nos va poniendo cada nodo que se va a analizar, pero con todos los demás nodos que se han recorrido para llegar hasta este último.

e)

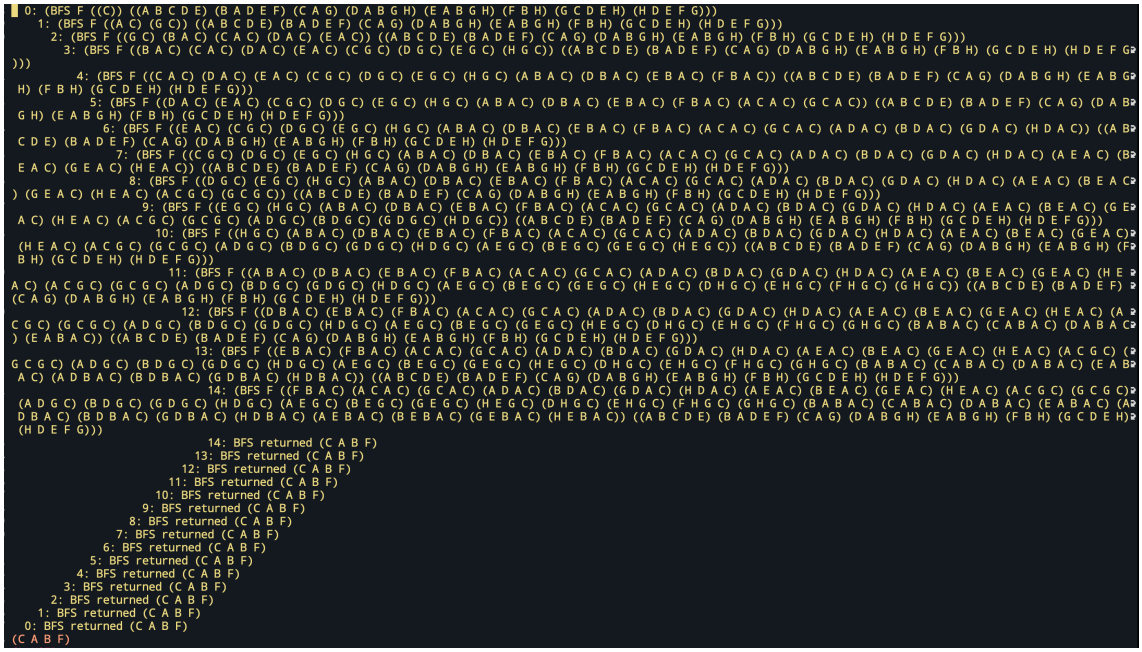
```
insert_cola(nodo ini)
mientras(!cola_vacia)
        n=extract_queue()
        insert_hijos_en_cola(n)
```

f)

El algoritmo BFS realiza una búsqueda en anchura por tanto partiendo del nodo inicial sacaremos todos sus hijos, una vez tengamos todos los examinaremos uno a uno guardando sus hijo para su posterior examinación después de que se hayan examinado los de profundidad anterior, hasta llegar a la solución.

g)

(shortest-path 'c 'f '((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) Con este ejemplo entra en un bucle infinito

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