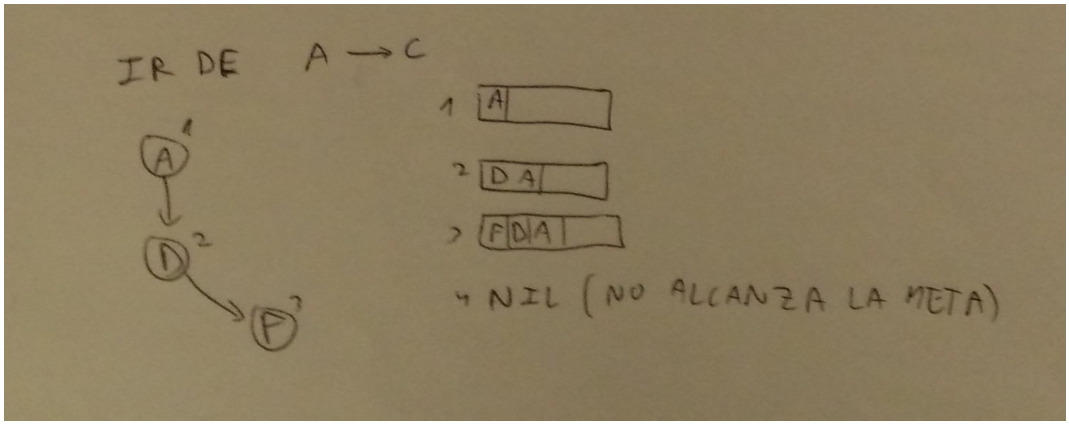
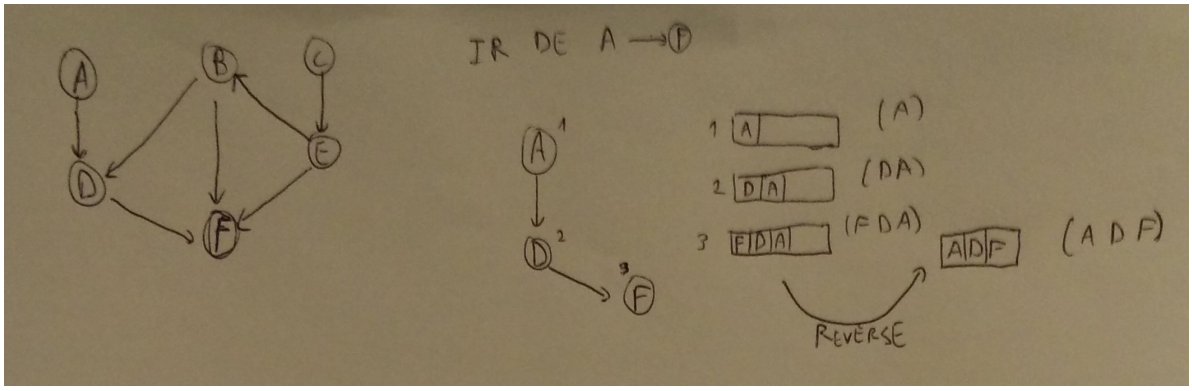
EJERCICIO 1:

i) El algoritmo recibe como argumentos un nodo objetivo, una cola y el grafo en el que buscar.
La primera comprobación que realiza el algoritmo es si la cola es nula. En este caso el algoritmo para y devuelve nil,
En caso de no ser nula realiza dos asignaciones: asigna a path (el camino) el car (parte izquierda) de la cola, y a node el car del path, que sería en la primera iteración el nodo del que partimos. En las siguientes iteraciones este valor será el del adyacente por el que estemos explorando.
El siguiente paso es una comprobación de si hemos alcanzado el nodo objetivo, y, de ser así linvierte la cola y devuelve el camino hasta el objetivo.
Por último, el algoritmo se llama recursivamente manteniendo el valor de los parámetros salvo el de la cola. Este valor se modifica en la función new-path, que devuelve todos los caminos que salen del nodo que estamos tratando. En la función de bfs se realiza un append de la parte derecha de esa cola, para poder operar con los nodos adyacentes al nodo que estábamos explorando.

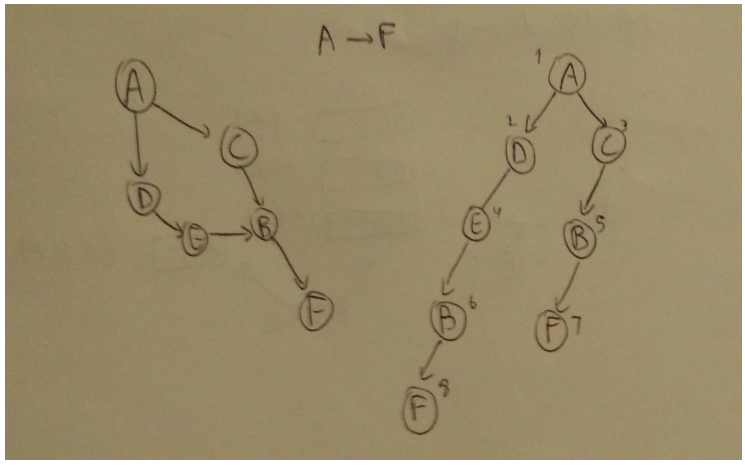
a)



b)



c)



ii) Encuentra el camino más corto ya que lo que hace es una llamada al algoritmo de búsqueda, que encuentra este camino.

Iii)

Función: bfs (end queue net)
Entrada: end [nodo objetivo]
queue [cola]
net [grafo]
Salida: Lista que contiene el camino más corto del nodo inicial al nodo objetivo.

Pseudocódigo:

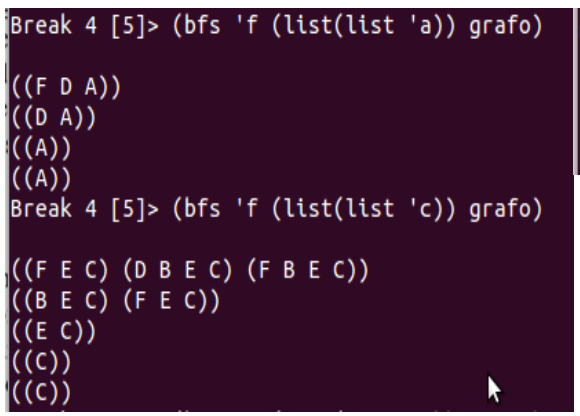
Si la lista es vacía,
devuelve nil.
En caso contrario
Se vinculan los valores de path con el car de queue y de node con el car de path.
Si el nodo es el objetivo,
se invierte la cola y se devuelve el camino encontrado.
En caso contrario:
Llamada recursiva pasando como parámetros end, net y el resultado de modificar la cola mediante la función new-paths, seguido de un append de cdr y la cola.

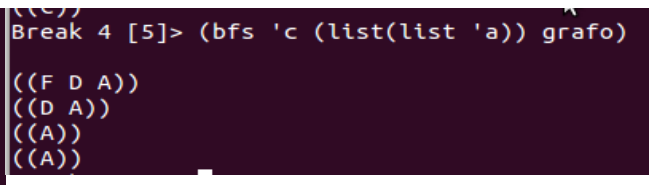
iv)

```
(defun bfs (end queue net)
  (if (null queue)                      ;Caso Base
      nil
              ;;;Se vinculan los valores de path y de node
    (let ((path (car queue)))
      (let ((node (car path)))
        (if (eql node end)              ;Caso Base
            (reverse path)
                              ;;;Recursion, se exploran los caminos con origen en el nodo con que estamos trabajando
          (bfs end
               (append (cdr queue)
                       (new-paths path node net))  ;Esta llamada nos devuelve la cola modificada con los caminos sacados de new_paths
               net))))))

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

v) El funcionamiento del código con los ejemplos está en el ejercicio (i).
También adjuntamos una captura de pantalla de la evolución del algoritmo, tanto en caso de error como en dos casos típicos, uno de ellos el propuesto en el ejercicio.





vi)

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

b) (FBAC)

vii)

a) Utilizando el grafo del ejemplo 1, dirigiendo F a D y haciendo una búsqueda de A a C desborda la pila al encontrarse en un bucle infinito debido a los estados repetidos.
(shortest-path 'a 'f '((a d) (b d f) (c e) (d f) (e b f) (f d)))

b) El código para este ejercicio es el mostrado en el anexo.

PSEUDOCÓDIGOS

Función 1: bfs-improved (end queue net)

```
Entrada:  end  (nodo objetivo)
          queue(cola con los caminos)
          net: (grafo en el que operamos)
Salida:   Camino al nodo objetivo

Procesamiento:

     Si queue es nula,
           evalúa a NIL

     en caso contrario:
          Se vinculan los valores de path con el car de queue y de
          node con el car de path.

     Si el nodo es el objetivo,
           se invierte la cola y se devuelve el camino encontrado.
     en caso contrario:
          Llamada recursiva pasando como parámetros end, net y el
          resultado de modificar la cola mediante la función new-
          paths-improved, seguido de un append de cdr y la cola.
```

Función 2: new-paths-improved (path node net)

```
Entrada:  path (lista actual)
          node(nodo del que sacar caminos)
          net: (grafo en el que operamos)
Salida:   Lista con caminos que salen de node

Procesamiento:

     Si la llamada a la funcion unique-p sobre path devuelve NIL,
           evalúa a NIL

     en caso contrario:
           añadimos a path todos los caminos asociados a node
```

Función 3: shortest-path-improved (end queue net)

Entrada: end (nodo objetivo)
queue(cola con los caminos)
net: (grafo en el que operamos)
Salida: Camino al nodo objetivo

Procesamiento:

Llamada a la funcion de busqueda en anchura mejorada

Función 4: unique-p (list)

Entrada: list (lista en la que comprobar)
Salida: T si hay elementos repetidos o NIL si no

Procesamiento:

Comprobar que la lista no es nula

en caso contrario:

evaluar un elemento de la lista y ver si se encuentra en la parte ya analizada de la lista

Llamada recursiva con el siguiente elemento de la lista

EJERCICIO 2:

El código de las funciones está en el anexo. En esta memoria están los pseudocódigos.

(i)

Función 1: euclidean-rec (x y)

```
Entrada:  x    (vector)
          y    (vector)
Salida:   Distancia euclidea entre 2 vectores

Procesamiento:

     Si x o y son nulos,
          evalúa a 0
     en caso contrario:
          llamar a suma-sqr y hacer la raiz cuadrada del resultado
```

Función 2: suma-sqr (x y)

```
Entrada:  x    (vector)
          y    (vector)
Salida:   Distancia euclidea al cuadrado entre 2 vectores

Procesamiento:

     Si x o y son nulos,
          evalúa a 0
     en caso contrario:
          si hemos llegado al final de la lista,
               hacer el cuadrado de la resta de los elementos de
los vectores
          en caso contrario:
               Hacer la suma del cuadrado de la resta de la
               posición en la que estamos con el  resultado de una
               llamada recursiva a suma-sqr.
```

Comentarios:

Si una de la listas tiene un numero de elemento mayor que la otra devolverá NIL.

Función 3: AD-recur(x y)

```
Entrada:  x    (vector)
          y    (vector)
Salida:
     Distancia absoluta

Procesamiento:

     Si x es NULL,
          evalua a nil
     Si y es NULL,
          evalua a nil
     Si estamos en el ultimo elemento de las listas
          Hacemos el valor absoluto de la resta del elemento en el
          que nos encontramos
     En caso contrario
          Hacemos la suma del valor absoluto de la resta de los
          elementos de la lista x y en que nos encontramos con el
          el resultado de la llamada recursiva al siguiente
          elemento de la lista.
```

(ii)

Función 1: euclidean-mapcar (x y)

```
Entrada:  x    (vector)
          y    (vector)
Salida:   la distancia euclídea entre los vectores

Procesamiento:
     Si x o y es NULL,
          evalua a nil
     En caso contrario:
          realizamos la raíz cuadrada de la suma de los cuadrados
          de la resta de dos elementos que se encuentran en la
          misma posicion.
```

Función 2: AD-mapcar (x y)

```
Entrada:  x    (vector)
          y    (vector)
Salida:   Distancia absoluta entre 2 vectores

Procesamiento:
     Si x o y es NULL,
          evalua a nil
     En caso contrario
```

```
     Se devuelve el valor de la suma de los valores absolutos de
     restar los elementos de la posición x con los de la y
```

EJERCICIO 3:

El código de los ejercicios está en el anexo. Aquí están los pseudocódigos.

3.1)

Función: combine-elt-lst (elt lst)

```
Entrada:  elt  (elemento a combinar)
          lst  (lista a combinar)
Salida:   Lista con listas combinadas

Procesamiento:
     Si lst es nula

          evalúa a NIL

     En caso contrario:

          Se crea una lista con el elemento y el primer elemento de
          la lista

          Caso recursivo: Se llama a la función para combinar el
          resto de elementos
```

3.2)

Función: combine-lst-lst (lst1 lst2)

```
Entrada:  lst1 (lista a combinar)
          lst2 (lista a combinar)
Salida:   Producto cartesiano de las dos listas

Procesamiento:
     Si lst1 es nula

          evalua a NIL

     En caso contrario:

          Se crea una lista la combinacion del primer elemento de
          la primera lista con los elementos de la segunda.

          Caso recursivo: Se llama a la funcion para combinar el
          resto de elementos
```

3.3)

Función: combine-list-of-lsts (lstolsts)

```
Entrada:  lstolsts  (lista de listas)
Salida:   Lista con listas combinadas

Procesamiento:
     Si nos encontramos en el ultimo elemento

          Devolvemos una lista con el first de todos los elementos

     En caso contrario

          Juntamos los primero elementos de las listas con su resto
          donde la lista es la combinación de listas de aquella en
```

la que estamos posicionado con el resultado de una llamada recursiva a la función con el resto de la lista

EJERCICIO 4:

El código se encuentra en el anexo, aquí pseudocódigo.

4.1)

Función: extrae-simbolos (expr)

```
Entrada:  expr (lista con la FBF)
Salida:   Lista con los símbolos atómicos de la FBF

Procesamiento:
     Si expr es nula
          evalua a NIL
     En caso contrario:
          Se pasan todos los elementos de expr a una sola lista.
          Se eliminan todos los elementos que no sean símbolos
          atómicos y se eliminan duplicados en caso de haberlos.
```

Función: aplana (expr)

```
Entrada:  expr (lista con la FBF)
Salida:   Lista con los símbolos atómicos de la FBF

Procesamiento:
     Si x es nula
          evalua a NIL
     En caso contrario: caso recursivo
          Si un elemento es un simbolo atomico se llama a la
          funcion pasando el resto de la lista para que el
          resultado se junte con el elemento
     En caso contrario: caso recursivo
          Si un elemento es una lista se llama a la funcion para
          acabar teniendo un elemento y no una lista
```

4.2)

Funcion: genera-lista-interpretaciones (lst)

```
Entrada: lst (lista)
Salida:   Lista de listas con las posibles combinaciones
          con T y NIL en grupos de n, donde n es el numero de
          elementos de la lista.
```

Procesamiento:

Si lst es NULL

Devuelve NIL

En caso contrario

Generamos en una lista de listas con 2 sublistas una con la variable y el valor T y otra con la variable y el valor NIL. Siendo el numero de listas el total de elementos de lst.

Con estos valores generamos las ditintas combinaciones llamando a la función combine-list-of-lsts que nos mezclara las distintas posibles combinaciones de lo valores de verdad.

Comentarios: La salida seria como si la parte izquierda fuese la posición de un bit y la parte derecha un bit de 0 o 1 y se tuviese que sacar una lista decremental de valores en binario.