PROGRAMACIÓN FUNCIONAL

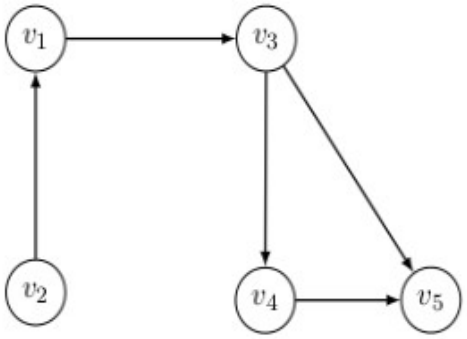
Algorítmos en gráficas

Janeth De Anda Gil

Objetivos: Introducir los conceptos de gráficas e implementar operaciones en programación funcional que manipulen las gráficas, así como también implementar busquedas en las gráficas.

1. Definición de gráfica

Una gráfica o gráfo denotado como G=(V,E), consiste en un conjunto finito de nodos o vértices (V) y un conjunto de aristas E, donde una arista i j representa una conexión entre dos vértices i y j, en la figura 1.1 se presenta una representación visual de una gráfica



$$V=\{v_1, v_2, v_3, v_4, v_5\},$$
$$E=\{(v_2, v_1), (v_1, v_3), (v_3, v_4), (v_3, v_5), (v_4, v_5)\}$$

Figura 1.1 Representación de una gráfica

Las gráficas pueden ser de dos tipos: dirigídas y no dirigídas. En las gráficas dirigídas, las aristas tienen una dirección específica, por ejemplo la figura 1.1 representa una gráfica dirigída.
Un camino de v1 a vn, es una secuencia de nodos unidos por una arista v1,v2,...,vn. Un ciclo es un camino simple que comienza y termina en el mismo nodo.
El conjunto de nodos directamente conectados por una arista a un nodo particular, se dicen que son adyacentes al nodo

2.-Representación de una gráfica

Una gráfica puede ser representada de varias formas como son listas de adyacencia, matríz de adyacencia, Puntero, inductiva y se puede manipular como un ADT (Abstract Data Type). En este caso describiremos gráficas con peso en las aristas.
Para manipular una gráfica en un lenguaje funcional, definiremos el tipo (Graph n w) donde n es el tipo de nodos, es decir, un índice (class Ix) y w es el peso de la arista (class Num).

Las operaciones necesarias para manipular un gráfo ADT son las siguientes :

mkGraph :: (Ix n, Num w) => Bool -> (n,n) -> [(n,n,w)] -> (Graph n w)
Toma un parametro booleano para indicar si una gráfica es dirigida o no, toma los límites inferior y superior del conjunto de vertices, una lista de aristas (dos vértices y el peso), regresa una gráfica

nodes :: (Ix n, Num w) => (Graph n w) -> [n] : regresa los nodos en el gráfo

EdgesD, edgesU :: (Ix n, Num w) => (Graph n w) -> [(n,n,w)]: regresa las aristas de un grafo dirigido y no dirigido

edgeIn :: (Ix n, Num w) => (Graph n w) -> (n,n) -> Bool : regresa True si existe la arista (n,n) en el gráfo

weight :: (Ix n, Num w) => n -> n -> (Graph n w) -> w : regresa el peso de una arista conectada por dos nodos

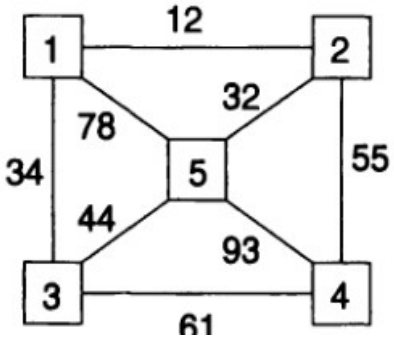
adjacent :: (Ix n, Num w) => (Graph n w) -> n -> [n] : regresa los nodos adyacentes a un nodo en particular

2.1.-Representación puntero.

Una manera de representar una gráfica es declarar un árbol general usando lo siguiente:

```
data Graph n w=Vertex n [((Graph,w),w)]
```

En está definición son posibles las referencias circulares usando variables locales para referenciar nodos, por ejemplo en la figura 2.1.1 , se tiene una grafica con su respectiva representación.




```
graphPTR = v1
     where
       v1 = Vertex 1 [(v2,12),(v3,34),(v5,78)]
       v2 = Vertex 2 [(v1,12),(v4,55),(v5,32)]
       v3 = Vertex 3 [(v1,34),(v4,61),(v5,44)]
       v4 = Vertex 4 [(v2,55),(v3,61),(v5,93)]
       v5 = Vertex 5 [(v1,78),(v2,32),(v3,44),(v4,93)]
```

Figura 2.1.1 Gráfica con su representación puntero

Esta representación es muy eficiente en espacio porque atravesar la gráfica no requiere la creación de estructuras de datos. Pero un problema es que si la gráfica no está conectada, cada componente tiene que ser definida por separado, otro problema es que los punteros que llegan a tener lazos o son circulares no pueden ser manipulados diretamente o comparados, tampoco se puede construir o atravesar gráficas de manera sencilla.

2.2 Representación lista de adyacencia

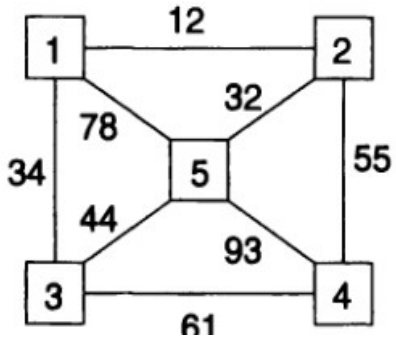
Otra representación es la lista de adyacencia, donde una estructura lineal toma cada nodo de la gráfica junto con sus nodos adyacentes, podemos definir una gráfica de la siguiente manera:

```
data Graph n w= [(n,[(n,w))]
```

En esta representación acceder a la lista de adyacencia de un nodo puede tomar 0(|V|).Una mejor representación es haciendo uso de array, esto permite un acceso constante a cualquier lista de adyacencia, en este caso definimos la gráfica de la siguiente manera:

```
data Graph n w= array n [(n,w))]
```

Un ejemplo de lista de adyacencias se muestra en la siguiente figura




```
graphAL = array (1,5) [(1,[(2,12),(3,34),(5,78)]),
                       (2,[(1,12),(4,55),(5,32)]),
                       (3,[(1,34),(4,61),(5,44)]),
                       (4,[(2,55),(3,61),(5,93)]),
                       (5,[(1,78),(2,32),(3,44),(4,93)])]
```

Figura 2.2.1 Representación de una lista de adyacencia

A continuación se definirán las operaciones para manipular una gráfica con programación funcional.

```
mkGraph:: (Ix n, Num w)=>Bool->(n,n)->[(n,n,w)]->(Graph n w)
```

```
mkGraph dir bounds lista =

     accumArray (\xs x-> x:xs) [] bounds

                  ([(x1,(x2,w))|(x1,x2,w)<-lista]++

                        if dir then []

                        else [(x2,(x1,w))|(x1,x2,w)<-lista, x1/=x2])

adjacent::(Ix n, Num w)=>(Graph n w)->n->[n]

adjacent g v = map fst (g!v)

node::(Ix n, Num w)=>(Graph n w)->[n]

node g = indices g

 edgeIn::(Ix n, Num w)=>(Graph n w)->(n,n)->Bool

edgeIn g (x,y)= elem y (adjacent g x)

weight::(Ix n, Num w)=>n->n->(Graph n w)->w

weight x y g= head [c|(a,c)<-g!x,(a==y)]

edgesD::(Ix n, Num w)=>(Graph n w)->[(n,n,w)]

edgesD g=[(v1,v2,w)|v1 <- node g, (v2,w)<-g!v1]

edgesU::(Ix n, Num w)=>(Graph n w)->[(n,n,w)]

edgesU g=[(v1,v2,w)|v1 <- node g, (v2,w)<-g!v1, v1<v2]
```

2.3 Representación matriz de adyacencia

Esta representación usa un arreglo bidimensional , donde la dimensión del arreglo es |V| x |V| , donde ambas coordenadas son los nodos i y j. El principal problema usando está representación es que se necesita un valor que corresponda a una arista que no exista, para esto se usa un tipo algebraico predefinido llamado Maybe con dos constructores, Nothing y Just b, entonces si no hay arista el valor del constructor es Nothing. En este caso definimos la gráfica de la siguiente manera

```
data Graph n w= Array (n,n) (Maybe w)
```

A continuación se definirán las operaciones para manipular una gráfica representada con matriz de adyacencias con programación funcional.

```
mkGraph:: (Ix n, Num w)=>Bool->(n,n)->[(n,n,w)]->(Graph n w)

mkGraph dir bnds@(l,u) lista =

     emptyArray // ([((x1,x2),Just w)|(x1,x2,w)<-lista]++

                          if dir then []

                          else [((x1,x2),Just w)|(x1,x2,w)<-lista,
x1/=x2])

     where emptyArray =array ((l,l),(u,u)) [((x1,x2), Nothing)| x1<-
range bnds, x2<- range bnds]

adjacent::(Ix n, Num w, Eq w)=>(Graph n w)->n->[n]

adjacent g v1 = [v2|v2<-node g,(g!(v1,v2))/=Nothing]

node::(Ix n, Num w)=>(Graph n w)->[n]

node g = range (l,u) where ((l,_),(u,_))= bounds g

edgeIn::(Ix n, Num w, Eq w)=>(Graph n w)->(n,n)->Bool

edgeIn g (x,y)= (g!(x,y))/=Nothing

weight::(Ix n, Num w)=>n->n->(Graph n w)->w

weight x y g= w where (Just w) =g! (x,y)

edgesD::(Ix n, Num w, Eq w)=>(Graph n w)->[(n,n,w)]

edgesD g=[(v1,v2,unwrap(g!(v1,v2)))|v1 <- node g, v2<-node g, edgeIn
g (v1,v2)] where unwrap (Just w) = w

edgesU::(Ix n, Num w, Eq w)=> (Graph n w)->[(n,n,w)]

edgesU g= let (_,(u,_)) = bounds g in [(v1,v2,unwrap(g!(v1,v2)))|v1
<- node g, v2<-range (v1,u), edgeIn g (v1,v2)] where unwrap (Just w)
= w
```

3.- Búsquedas en una gráfica

Se pueden usar dos estrategías para atravesar una gráfica, búsqueda por profundidad (depth first search) y búsqueda por anchura (breadth first search)

3.1 Búsqueda por profundidad

En este caso se visita un nodo seguido de sus adyacencias , vease la figura 3.1.1

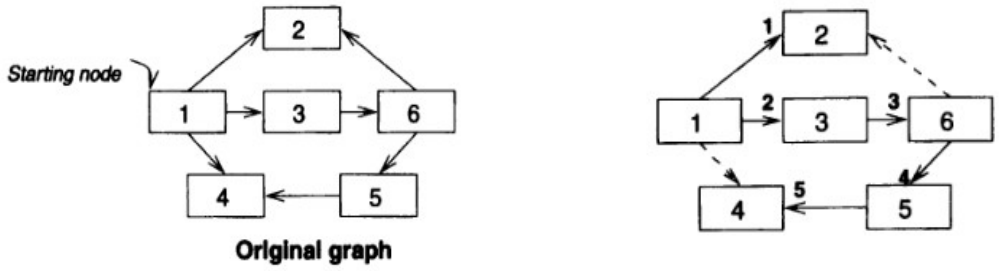


Figura 3.1.1 Representación de la búsqueda por profundidad de una gráfica

La implementación tiene dos listas: nodos visitados y nodos por visitar, en este caso en los nodos por visitar se van colocando las adyacencias al inicio de la lista, a continuación se muestra el código de la implementación, cabe señalar que esta implementación también se puede realizar con una estructura de datos de tipo pila.

```
depthFirstSearch:: (Ix n, Num w)=> n->(Graph n w)->[n]

depthFirstSearch start g =reverse (dfs [start] [])
    where
        dfs[] vis = vis
        dfs (c:cs) vis
            | elem c vis =dfs cs vis
            | otherwise = dfs ((adjacent g c)++cs) (c:vis)
```

Figura 3.1.2 Implementación en Haskell de la función de búsqueda por profundidad

```
depthFirstSearch'' start g
                    = reverse (dfs (push start emptyStack) [])
 where
  dfs s vis
   | (stackEmpty s)    = vis
   | elem (top s) vis = dfs (pop s) vis
   | otherwise         = let c = top s
                         in
                          dfs (foldr push (pop s) (adjacent g c))
                              (c:vis)
```

Figura 3.1.3 Implementación en Haskell de la función de búsqueda por profundidad con una pila.

3.2 Búsqueda por anchura

En este caso se visita los nodos por niveles, es decir, se colocan las adyacencias al final , vease la figura 3.2.1

Figura 3.2.1 Representación de la búsqueda por anchura de una gráfica

La implementación tiene dos listas: nodos visitados y nodos por visitar, en este caso las adyacencias de los nodos visitados se colocan al final de la lista, para que se tenga el efecto de atravesar la gráfica por anchura, a continuación se muestra el código de la implementación, cabe señalar que esta implementación también se puede realizar con una estructura de datos de tipo cola.

```
breadthFirstSearch:: (Ix n, Num w)=> n->(Graph n w)->[n]

breadthFirstSearch start g =reverse (bfs [start] [])

    where
        bfs[] vis = vis
        bfs (c:cs) vis
            | elem c vis = bfs cs vis
            | otherwise = bfs (cs++(adjacent g c)) (c:vis)
```

Figura 3.2.2 Implementación de la búsqueda por profundidad

```
breadthFirstSearch start g
                   = reverse (bfs (enqueue start emptyQueue) [])
 where
  bfs q vis
   | (queueEmpty q)= vis
   | elem (front q) vis
                   = bfs (dequeue q) vis
   | otherwise     = let c = front q
                     in
                      bfs (foldr enqueue
                                 (dequeue q)
                                 (adjacent g c))
                          (c:vis)
```

Figura 3.2.3 Implementación en Haskell de la función de búsqueda por profundidad con una cola.

4.- Representación Inductiva

Esta representación tiene un enfoque más adecuado a la programación funcional, desde un punto de vista inductivo, se define el tipo de datos de gráficas finitas dirigidas con nodos y aristas etiquetados como sigue:

-La gráfica vacía es una gráfica

-Sea G una gráfica y v un nuevo nodo entonces, añadir a la gráfica G el nuevo nodo v y sus aristas incidentes (a lo cual llamaremos extensión de G dado un Contexto) también es una gráfica.

Un ejemplo de extensión se muestra en la figura 4.1, donde nuestra gráfica esta formada por los nodos 1(“a”), 2(“b”), 3(“c”) y sus respectivas aristas (imagen del lado izquierdo) y en la imagen del lado derecho, a la gráfica le añadimos una extensión que contiene el nodo 4 (“d”) con sus respectivas aristas.

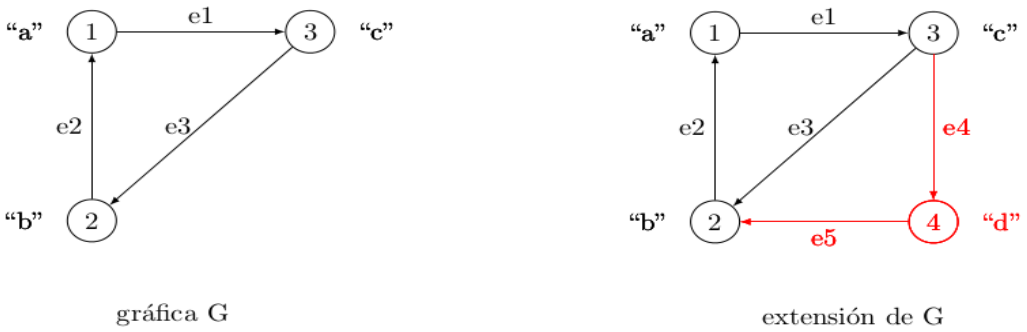


Figura 4.1 Representación de una gráfica con su extensión (contexto)

En la siguiente imagen se puede ver dos ejemplos de la contrucción de una gráfica

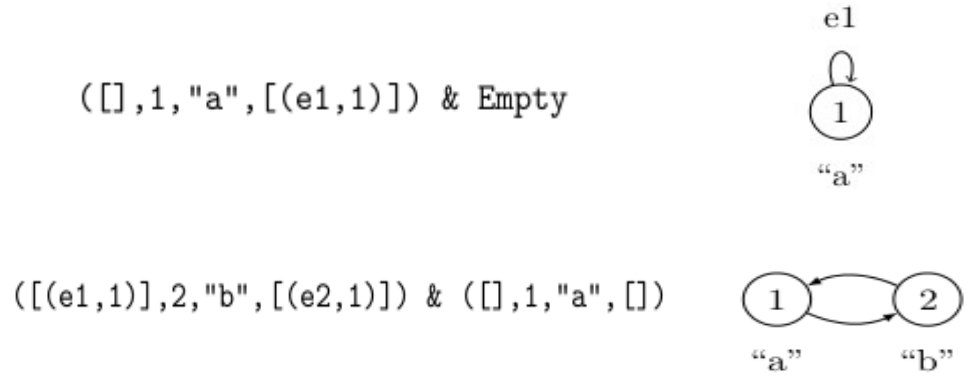


Figura 4.2 construcción de gráficas con contextos

Para construir gráficas se utilizará:

El tipo de dato Graph a b que representa las gráficas cuyos nodos de tipo a y aristas etiquetadas con elementos de tipo b.

El sinónimo de tipo Context que es una cuádrupla formada por: la Adyacencia a nodos prede- cesores, un entero que representa al nodo en cuestión, la etiqueta del nodo y la Adyacencia a nodos sucesores.

El sinónimo de tipo Adyacencia (Adj b) que es una lista de tuplas utilizada en el tipo Contexto (asociado, digamos, al nodo v), dependiendo si se trata de la primer o última entrada, esta lista se refiere a los nodos predecesores o sucesores respectivamente. La tupla consiste en la primer entrada de la etiqueta de una arista y en la segunda entrada de un nodo adyacente a v predecesor o sucesor, según sea el caso.

Toda esta información es utilizada en el paso inductivo que genera la extensión de una gráfica. Los contextos no son utilizados únicamente en la construcción de graficas, también son empleados en las operaciones de descomposición. La descomposición de gráficas puede fallar por muchas razones, por ello las operaciones de descomposición y sus tipos deben estar prevenidos para esos casos, se cuenta entonces con los tipos llamado MContext (Maybe-Context), Decomp y GDecomp representando respectivamente un posible contexto, un par que en su primer entrada contiene un posible contexto y en la segunda una gráfica y por último un par que contiene un contexto y una gráfica.

```
data  Graph a b    = Empty | Context a b & Graph a b

      type  Node         = Int
      type  LNode  a     = (Node,a)

      type  Edge         = (Node,Node)
      type  LEdge  b     = (Node,Node,b)

      type  Adj b        = [(b,Node)]
      type  Context a b  = (Adj b, Node, a, Adj b)
      type  MContext a b  = Maybe (Context a b)

      type Decomp g a b = (Mcontext a b , g a b)

      type GDecomp g a b = (Context a b , g a b)
```

El tipo de datos de gráficas finitas dirigidas y etiquetadas (a las cuales nos referiremos únicamente como gráficas) se define recursivamente como sigue:

1. La gráfica vacía es una gráfica y se denota por empty.

2. Si c es un contexto y g es una gráfica entonces c&g es una gráfica.

3. Sólo éstas son gráficas.

Para demostrar propiedades acerca de este tipo de datos es necesario recurrir al principio de inducción estructural, que está basado en las reglas base y recursiva de la definición expuesta anteriormente, así como el análisis de los patrones básicos inducidos. La definición recursiva de gráficas genera la siguiente versión del principio de inducción estructural para gráficas:

Sea P una propiedad acerca de gráficas. Si se desea probar P(g) para toda gráfica g basta proceder como sigue:

Base de la inducción: probar que se cumple P(empty) directamente.

Hipótesis de inducción: suponer que se cumple P(g1) donde g1 es una gráfica que ha sido construida previamente.

Paso inductivo: sea c un contexto, probar que es válido P(c&g1), es decir, probar que se cumple la propiedad para una gráfica g1.

Conclusión: En tal caso, el principio de inducción estructural sobre gráficas nos permite concluir válido :

$$\forall g\ P(g)\ .$$

Con el fin de construir de una manera adecuada una gráfica, evitando ciertos errores se introducen dos funciones: Empty es una función que regresa una gráfica vacía y & una función que recibe una gráfica y un contexto, regresa una gráfica, si es que esta se puede construir. La función & produce error cuando se quiere agregar el contexto de un nodo que ya está presente en la gráfica o bien cuando se hace referencia en la lista de sucesores o predecesores a un nodo que aún no está en la gráfica.

4.1 Casación de patrones en gráficas (Pattern matching)

La definición recursiva de gráficas con la que contamos - la cual, por supuesto, ha generado un principio de inducción estructural- permite definir funciones sobre gáficas empleando la técnica de casación de patrones (pattern matching): cada cláusula de la definición del tipo de datos intruduce un patrón, el cual es un esquema sintáctico bien definido que se utiliza para definir un caso de la función en cuestión.

```
isEmpty :: gr a b -> Bool

matchAny ::gr a b -> GDecomp gr a b
```

De donde necesitaremos agregar:

```
type Decomp g a b = (MContext a b,gr a b)
```

La función isEmpty recibe una gráfica y nos dice si es vacía o no. Por otro lado matchAny recibe una gráfica e intenta descomponerla, regresando un par con un contexto y la gráfica restante. Esta función reporta un error, por ejemplo, cuando es aplicada a una gráfica vacía pues no se puede obtener un contexto.

La operación matchAny g devolverá (en alguno de sus casos) (c1,g1) donde c1 es el contexto. MatchAny no devuelve los contextos exactamente en el orden inverso en el que fueron insertados en la gráfica. Sin embargo, habrá situaciones en las que se requiera

obtener contextos en algún orden específico; para ello se define la función match que recibe como parámetros, v un nodo, g una gráfica y devuelve el contexto asociado a v si es que lo hay. Esto es:

```
match :: Node-> gr a b -> GDecomp gr a b
```

4.2 Búsqueda en profundidad

La búsqueda en profundidad (Depth First Search) es un algoritmo que revela información sobre la estructura interna de una gráfica; la información obtenida es utilizada para implementar otros algoritmos como ordenamiento topológico o computar componentes fuertemente conexos.

La búsqueda a profundidad permite recorrer todos los nodos de una gráfica de manera ordenada pero no uniforme. El algoritmo consiste en visitar cada nodo de la gráfica una única vez, visitando siempre el primer sucesor no visitado del nodo actual antes que hermanos.El algoritmo recibe como parámetro una gráfica y regresa una lista con los nodos en el orden que fueron visitados.

A continuaciión se explica la implementación de la búsqueda a profundidad en programación funcional.

Los parámetros para esta función serán por supuesto, la gráfica a ser procesada y una lista de nodos siendo el primer elemento de la lista el nodo donde inicia la búsqueda. Esta lista se hace necesaria para el caso de procesar una gráfica no conexa donde, después de haber explorado uno de los componentes de la gráfica, será necesario algún nodo de otro componente conexo para continuar con la búsqueda.

El resultado de la búsqueda a profundidad será una lista de nodos en el orden en el que fueron visitados. Considérese entonces el siguiente algortimo:

```
dfs :: Graph gr => [Node] -> gr a b  -> [Node]
dfs  [ ]      g    = [ ]
dfs   _      empty = [ ]
dfs (v:vs) (c &v g) = v : dfs (suc c ++ vs) g
dfs (v:vs)    g    = dfs vs g
```

El algoritmo trabaja de la siguiente manera:

Si ya no hay nodos para ser visitados, es decir, que la lista de nodos sea vacía o bien la gráfica a ser procesada es una gráfica

vacía , dfs se detiene y regresa una lista sin nodos. Si aún hay nodos para ser visitados se utiliza la cabeza de la lista, digamos v, dfs intenta obtener el contexto asociado a v en la gráfica que es pasada como argumento. Si es posible obtener el contexto asociado a v, digamos c, que será el caso cuando v es un elemento de la gráfica, entonces: v se agrega v a la lista resultante y se continúa la búsqueda sobre la gráfica restante g visitando primero los sucesores de v antes que los nodos en la cola de la lista pasada como par ámetro vs; esto último se logra al llamar dfs con primer parámetro (suc c ++ vs).

Justo el hecho de colocar los sucesores del nodo al inicio de la lista en la siguiente llamada para ser visitados antes que los nodos en la lista restante, es lo que permite que se realice la búsqueda a profundidad.

Finalmente, si el nodo v no pudo ser encontrado en la gr áfica, es decir, no se obtuvo un cotexto asociado a v, dfs continúa la búsqueda con la cola de la lista pasada como parámetro vs y la gráfica original g. Nótese que se llega a éste caso únicamente cuando v no está en la gráfica pues de lo contrario la ejecución habría entrado en el segundo caso.

Se muestra a continuación la implementación de dfs utilizando la función match:

```
dfs :: Graph gr => [Node] -> gr a b  -> [Node]
dfs  vs g | null vs || isEmpty g  = [ ]
dfs (v:vs)   g  = case match v g of
                  (Just c , g1) -> v:dfs (suc c ++ vs) g1
                  (Nothing , g) -> dfs vs g
```

4.3 Búsqueda por amplitud

La implementación del algoritmo de búsqueda en amplitud en programación funcional tiene una estructura y mecanismo similar a la implementación funcional de la búqueda a profundidad, excepto por la forma de procesar los nuevos sucesores encontrados, lo cual refleja precisamente el orden en que los nodos a ser visitados son ordenados. En dfs se emplea un mecanismo de pila, en donde los nuevos descubiertos se colocan frente a los previamente descubiertos; en bfs los nuevos nodos alcanzados son guardados en una cola por lo que serán procesados después de los que han sido previamente almacenados:

```
bfs :: Graph gr => [Node] -> gr a b  -> [Node]
bfs  vs g | null vs || isEmpty g  = [ ]
bfs (v:vs)   g  = case match v g of
                  (Just c , g1) -> v:bfs (vs ++ suc c) g1
                  (Nothing , g) -> bfs vs g
```

4.3 Observaciones

bfs y dfs requieren que se visiten los nodos una única vez, es interesante apreciar la facilidad con que se cubre este requerimiento en lenguaje funcional, es decir, basta eliminar de la gráfica el nodo que ha sido visitado, garantizando que una vez visitado un nodo, no se repetirá en la lista resultado. En el algoritmo en lenguaje imperativo presentado, sucede que pueden efectuarse operaciones-iteraciones innecesarias llegando a un nodo que ya ha sido visitado y que hasta que se revisa la marca del nodo que indica que ha sido visitado se procesa el siguiente nodo.

Estas operaciones-iteraciones innecesarias se ahorran en la definición funcional. Por otro lado considerese el código en haskell de dfs a el paso a bfs , hablando estrictamente en código, consistió únicamente en cambiar el orden en el que se agregan los sucesores a la lista -proporcionando de hecho, el mecanismo de cola o pila - esto es, en dfs se utilizó dfs (suc c ++ vs) g1 mientras que en bfs se empleó bfs (vs ++ suc c) g1. Esta similitud en la estructura del código de dfs y bfs nos permite apreciar más claramente la relación y las diferencias entre los algoritmos, situación que resulta ligeramente más complicada con los códigos correspondientes en lenguaje imperativo.