Introducción

En muchos problemas se necesita representar un conjunto de estados y posibles modos de transición de un estado a otro. Una estructura típica para representar estos mecanismos son los grafos.

Un ejemplo particular del uso de grafos consiste en describir una red y buscar un camino dentro de la misma. En caso que se desee encontrar un camino que cumpla con un conjunto de restricciones puede ser necesario revisar gran parte, sino todos, los demás caminos restantes.

El método descrito en este capítulo fue presentado por [Winston,1991] y sigue siendo una de las mejores representaciones del problema.

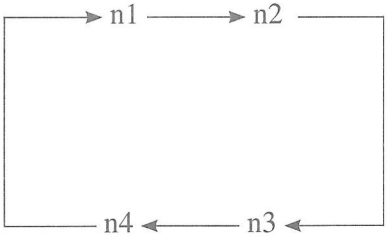
Definición de un grafo

Un grafo G consistirá de un conjunto de nodos que denominaremos N y un conjunto de arcos A. Un arco es un par ordenado de nodos.

Por ejemplo:

Sea G un grafo definido por:
N = (n1 n2 n3 n4)
A = ((n1 n2) (n2 n3) (n3 n4) (n4 n1))

Este grafo se puede representar mediante el siguiente dibujo:

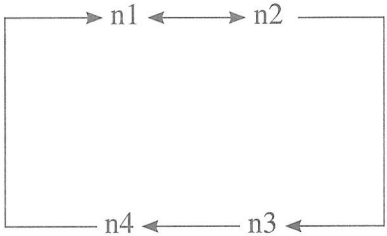


Es posible que un grafo tenga arcos de un nodo v a w, así como del nodo w a v. En estos casos se pueden representarán ambos arcos, o bien, se ignorará la dirección del arco.

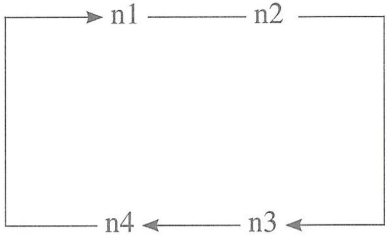
Por ejemplo:

Sea G un grafo definido por:
N = (n1 n2 n3 n4)
A = ((n1 n2) (n2 n1) (n2 n3) (n3 n4) (n4 n1))

Se puede representar como:



O bien se puede representar como:

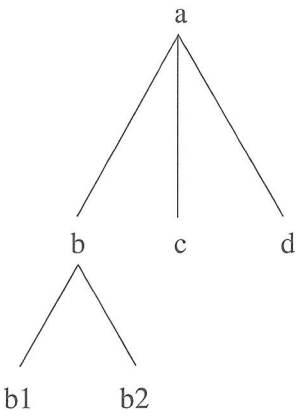


Finalmente se puede mencionar que todo árbol puede representarse como un grafo.

Por ejemplo:

Sea G un grafo definido por:
N = (a b c d b1 b2)
A = ((a b) (a c) (a d) (b b1) (b b2))

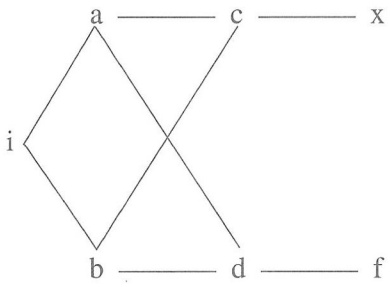
Que se representaría en forma gráfica por un árbol de la forma:



Finalmente se presenta un grafo que servirá como ejemplo a lo largo del resto del capítulo.

Sea G un grafo definido por:
N = (i a b c d x f)
A = ((i (a b))
(a (i c d))
(b (i c d))
(c (a b x))
(d (a b f))
(x (c))
(f (d))
)

Que se puede representar mediante el siguiente dibujo:



El problema principal de buscar todas las rutas

El problema principal que se desea resolver consiste en encontrar TODAS las posibles rutas que lleven desde el nodo i hasta el nodo f.

Este es un problema muy usual e interesante. Suponga que el grafo es una representación de un mapa. De esta forma, cada nodo representa una ciudad y cada arco representa una vía de acceso. Si se cuenta con todos los posibles caminos para llegar de la ciudad i a la ciudad f, sería posible estimar el costo de cada vía de acceso y de esta forma escoger el camino con la distancia más corta.

Para resolver el problema se utilizará la siguiente estrategia.

- Se irán desarrollando las posibles vías de acceso iniciando en i e incrementado el largo de cada vía hasta llegar a f.
- En el proceso de construcción de vías algunas llevarán a caminos equivocados, por ejemplo hacia x, estas vías simplemente se desecharán.
- Se continua el proceso hasta que se llega a una vía que contiene en su trayectoria la ciudad f.

Cada ruta se representará mediante una lista, de esta forma TODAS las posibles soluciones para ir del nodo i al nodo f se presentan a continuación:

```
> '((i a c b d f)
    (i a d f)
    (i b c a d f)
    (i b d f))
```

Una representación computacional para el Grafo

Inicialmente se debe buscar una representación adecuada para el grafo, se utilizará una lista de listas, donde el primer elemento indicará el nodo y una lista posterior indicará los nodos que son accesibles mediante un arco.

```
;;  Representación del grafo
;;  se guarda en la variable gg
;;
(define gg '( (i (a b))
              (a (i c d))
              (b (i c d))
              (c (a b x))
              (d (a b f))
              (x (c))
              (f (d))
              ))
```

Operaciones elementales sobre la representación

La primera operación se llamará *(solución? fin ruta)* y devolverá el valor de #f si la ruta propuesta llega hasta el nodo fin, en caso contrario devolverá el valor de #f. Por efecto de eficiencia las rutas se irán construyendo al revés, de modo tal, el primer nodo de la lista representa el último nodo visitado.

```
;;  Se obtiene una solución cuando
;;  se ha llegado al nodo destino
;;
(define (solución? fin ruta)
  (equal? fin (car ruta)))
```

De esta forma, esta funcion trabajará de la siguiente manera:

```
>  (solución? 'f '(f d a i))
#t

>  (solución? 'f '(d a i))
#f
```

A continuación se construirá una función de nombre *(vecinos ele grafo)*, esta función toma un nodo particular *(ele)* y retorna todos los vecinos inmediatos. En caso que el nodo no exista devuelve el valor de #f.

Observe que para la construcción de esta función se utiliza la primitiva *(asoc ele lista)* la cual realiza gran parte del trabajo deseado.

```
;;  Una función que retorna los vecinos inmediatos
;;
(define (vecinos ele grafo)
  (let ( (resultado (assoc ele grafo))
       )
    (cond ( (equal? resultado #f)
            #f)
          ( else
           (cadr resultado)))))
```

Ejemplos:

```
> (vecinos 'i gg)
(a b)

> (vecinos 'a gg)
(i c d)

> (vecinos 'z gg)
#f
```

La función de nombre (extender ruta grafo) toma una ruta parcial, denominada ruta, y construye con base en ella varias posibles caminos alternativos. Las rutas se construyen con el nodo inicial de último y el nodo más recientemente visitado de primero.

Al extender una trayectoria debe revisar que el nodo que está agregando no exista en la trayectoria parcial, de ser así se podría crear un ciclo infinito de la forma ir hacia a, ir hacia b, ir hacia a, ir hacia b, etc.

```
;;  Una función que crea nuevas trayectorias
;;  Si no puede crear nuevas trayectorias devuelve #f
;;
(define (extender ruta grafo)
  (apply append
```

```
(map (lambda(x)
       (cond  ((miembro? x ruta) '())
              (else (list (cons x ruta)))))
     (vecinos (car ruta) grafo)))
```

La función extender produciría los siguientes resultados:

```
> (extender '(i) gg)
((a i) (b i))

> (extender '(a i) gg)
((c a i) (d a i))

> (extender '(d a i) gg)
((b d a i) (f d a i))
```

Encontrar las rutas mediante profundidad primero

Es posible ahora escribir una función que encuentre UNA ruta que nos lleve desde i hasta f. La función (prof ini fin) realiza esto. El procedimiento de prof es el siguiente:

a. Inicia al hacer de ini un recorrido parcial.

b. Si el recorrido parcial contiene al nodo final se acaba el proceso.

c. De lo contrario toma el inicio del recorrido parcial y busca sus vecinos.

d. Extiende la trayectoria parcial para cada vecino.

e. Vuelve a (2) e inicia de nuevo.

El nombre de la función es prof porque esta función toma una posible ruta y la extiende continuamente hasta llegar al final o fracasar en su intento de encontrar una solución. Si fracasa continua extendiendo la siguiente ruta parcial. Por ello se dice que este procedimiento de construcción es de profundidad primero.

```
;;  Escribimos ahora un procedimiento que recorre el grafo
;;  utilizando la técnica de profundidad primero
;;
(define (prof ini fin grafo)
  (prof-aux (list (list ini)) fin grafo))
```

```
(define (prof-aux rutas fin grafo)
  (cond ( (null? rutas)
          '())
        ( (solución? fin (car rutas))
          (reverse (car rutas)))
        ( else
          (prof-aux (append (extender (car rutas) grafo)
                            (cdr rutas))
                    fin
                    grafo))))
```

Al ejecutarse (prof 'i 'f) se produce el siguiente resultado:

```
> (prof 'i 'f gg)
(i a c b d f)
```

Este proceso produce la siguiente lista de rutas parciales.

```
> (prof 'i 'f gg)
  rutas: ((i))
  rutas: ((a i) (b i))
  rutas: ((c a i) (d a i) (b i))
  rutas: ((b c a i) (x c a i) (d a i) (b i))
  rutas: ((d b c a i) (x c a i) (d a i) (b i))
  rutas: ((f d b c a i) (x c a i) (d a i) (b i))
  (i a c b d f)
```

Es fácil extender el procedimiento anterior para que en lugar de devolver una única solución sea capaz de devolver TODAS las soluciones. Una vez que se ha encontrado la primera solución esta se guarda dentro de una lista que se lleva para tal fin. Posteriormente es posible desarrollar las demás rutas parciales.

```
;; Encuentra TODAS ruta entre ini y fin en un grafo
;; Utilizndo Profundidad Primero
;;
(define (prof-todas ini fin grafo)
  (prof-todas-aux (list (list ini)) fin grafo '()))

(define (prof-todas-aux rutas fin grafo total)
  (cond ( (null? rutas)
          (map reverse total))
```

```
    ( (solución? fin (car rutas))
      (prof-todas-aux (cdr rutas)
                      fin
                      grafo
                      (cons (car rutas) total)))
  ( else
      (prof-todas-aux (append (extender (car rutas) grafo)
                              (cdr rutas))
                      fin
                      grafo
                      total))))
```

De esta forma se cuenta con la solución final.

```
> (prof-todas 'i 'f gg)
( (i b d f)
  (i b c a d f)
  (i a d f)
  (i a c b d f) )
```

Encontrar las rutas mediante anchura primero

Es posible implementar un proceso de búsqueda de las rutas que en lugar de extender siempre la primera ruta de la lista de rutas posibles vaya extendiendo todas las rutas de forma más simétrica. Este método se conoce con el nombre de anchura primero.

El algoritmo anterior se puede modificar fácilmente para realizar este proceso de búsqueda. Una vez que se ha extendido una ruta parcial esta se coloca al final de la lista de manera tal que primero se extenderán las demás rutas antes de volver a extender ésta.

A continuación se muestra el código.

```
;;  Encuentra TODAS ruta entre ini y fin en un grafo
;;  Utilizando Anchura Primero
;;
(define (anchura-todas ini fin grafo)
  (anchura-todas-aux (list (list ini)) fin grafo '()))
```

```
(define (anchura-todas-aux rutas fin grafo total)
  (cond ( (null? rutas)
          (map reverse total))
        ( (solución? fin (car rutas))
          (anchura-todas-aux  (cdr rutas)
                              fin
                              grafo
                              (cons (car rutas) total)))
        ( else
          (anchura-todas-aux (append
                               (cdr rutas)
                               (extender (car rutas) grafo))
                             fin
                             grafo
                             total))))
```

Este nuevo procedimiento producirá el siguiente conjunto de soluciones:

```
> (anchura-todas 'i 'f gg)
( (i b c a d f)
  (i a c b d f)
  (i b d f)
  (i a d f))
```


Resumen

- Es fácil representar estructuras de grafos en scheme.
- Un problema típico es encontrar todas las rutas para ir desde un nodo de inicio hasta un nodo final.
- Las técnicas de construcción de rutas con profundidad primero y anchura primero difieren en el modo en que construyen las rutas.

Ejercicios

Ejercicio N° 1.

Bajo que condiciones utilizaría usted en un problema profundidad primero?

Bajo que condiciones utilizaría usted en un problema anchura primero?

En general, cuál de estos dos procedimientos cree que es mejor?

Ejercicio N° 2.

Reescriba el procedimiento que busca todas las soluciones en profundidad primero de manera tal que ahora en lugar de utilizar una estructura de control basada en recursión lo haga con una estructura de control basada en repetición.

Ejercicio N° 3.

Póngale un valor a cada uno de los arcos de grafo. Una vez que cuente con todas las rutas posibles para ir de i hacia f escriba un procedimiento que toma el valor de cada arco y encuentra la ruta cuya suma es la de menor valor.

Para realizar este procedimiento puede hacerlo de dos formas.

- La primera modifique la estructura del grafo para incluir el valor de ir a cada uno de los nodos vecinos. Para ello también tiene que modificar el procedimiento de vecinos y extender.
- La segunda utilice una segunda estructura donde se guarden los datos de los valores. En este caso se toman las soluciones y se operan con esta segunda estructura.

Ejercicio N° 4.

Un gran número de problemas se pueden resolver mediante la técnica anterior de extender soluciones hasta encontrar una solución final. A continuación se describe un problema clásico conocido como el problema de las tres monedas.

Se tienen monedas que pueden colocarse en corona (C) o escudo (E). El problema de las tres monedas consiste en lo siguiente:

Se tienen tres monedas que marcan (C C E)

Se busca llegar a uno de los dos siguientes estados (C C C) o (E E E)

Construya un programa que utilice un proceso de búsqueda para localizar TODAS las formas en que estas tres monedas pueden llegar al estado final.

Ejercicio N° 5.

Se tienen dos contenedores de agua a los que llamaremos A y B. Al recipiente A le caben 5 litros de agua y al recipiente B 3 litros de agua. El objetivo final del juego es dejar una cantidad 2 litros de agua en el recipiente A.

Para representar este problema se utilizará una lista de la forma (A B). Inicialmente los dos recipientes se encuentran vacíos por los que el estado inicial es (0 0). Se busca entonces iniciar en (0 0) y terminar en (2 x) con x cualquier valor.

Las únicas operaciones válidas entre estos contenedores son:

1 Llenar el recipiente A.

2. Llenar el recipiente B.

3. Vaciar el recipiente A.

4. Vaciar el recipiente B.

5. Mover el contenido de A hacia B.

6. Mover el contenido de B hacia A.

Construya una función de nombre (rep A B C), que en nuestro caso se invocará como (rep 5 3 2) la cual debe producir una solución para este problema. Por ejemplo:

```
> (rep 5 3 2)
((0 0) (5 0) (2 3))
```

Construya una función de nombre (trep A B C) que se invocará como (trep 5 3 2) la cual debe producir TODAS las posibles soluciones para este problema.

Ejercicio N° 6.

Otro ejercicio típico es el problema de las n reinas. Se define un tablero de largo y anchura n. El problema consiste en colocar n reinas en dicho tablero de manera tal que ninguna de ellas ataque a las otras.

Para representar la posición de una reina dentro del tablero se utilizara un lista de dos elementos de la forma (i j), donde i representará la fila y j la columna donde se encuentra la reina.

Por ejemplo, suponga que se trata de un tablero 4x4. Las únicas maneras de colocar 4 reinas en dicho tablero sin que ninguna ataque a las otras es:

```
> (reinas 4)
( ((1 3) (2 1) (3 4) (4 2))
  ((1 2) (2 4) (3 1) (4 3)))
```

Ejercicio N° 7.

Otra forma de resolver el problema de las reinas es utilizando un algoritmo de permutaciones. Evidentemente cada reina deberá estar colocada en una fila diferente, bastaría entonces encontrar todas las posibles permutaciones de las columnas. Cada una de ellas se prueba y se eliminarán aquellas permutaciones que estén en conflicto.

Escriba un procedimiento de permutaciones y utilícelo para resolver el problema de las n reinas.

Ejercicio N° 8.

En un tablero de ajedrez de tamaño NxN se coloca un caballo de ajedrez en la posición (i j). Se desea que el caballo recorra todo el tablero sin caer dos veces en una misma casilla.

Construya un programa en que indique los movimientos que debe realizar el caballo partiendo de (i j) para recorrer todo el tablero. En esta caso debe construir únicamente una solución, no todas las soluciones posibles.

Ejercicio N° 9.

Utilizando los algoritmos descritos en esta sección es posible construir todas las formas en que se puede jugar una partida de ajedrez??

Se puede realizar ese programa??

En caso afirmativo que impide realizar dicho programa??

Ejercicio N° 10.

Un puzzle (¿¿acertijo??) es un juego infantil que consta de una superficie cuadrada que ha sido subdividida en cuadrados más pequeños. Finalmente uno de los cuadrados se ha removido para movimientos entre las piezas.

Utilice un método de profundidad primero o de anchura primero para determinar todas las posibles formas de ordenar un puzzle.

La representación que se utilizará consistirá en una matriz de tamaño NxN. La solución final consistirá en tener el puzzle ordenado.

En el caso de un puzzle de 3x3 este sería el estado final:

```
( (1 2 3)
  (4 5 6)
  (7 8 0))
```

Dos posibles estados iniciales para el problema serían:

```
( (6 7 5)
  (4 0 2)
  (8 1 3))
```

```
( (6 0 5)
  (4 7 2)
  (8 1 3))
```

Observe que del segundo estado se pueden construir los siguientes movimientos:

```
( (0 6 5)
  (4 7 2)
  (8 1 3))
```

```
( (6 5 0)
  (4 7 2)
  (8 1 3))
```

```
( (6 7 5)
  (4 0 2)
  (8 1 3))
```