Unidad 1 - Parte 3 – Recorridas sobre Grafos

Recorridas Sobre Grafos

Existen diversas maneras de recorrer un grafo. Esto significa, partir de un vértice dado y moverse a partir de él sobre el resto de los vértices a través de las aristas.

En este apartado se verán dos formas de recorrer grafos. La primera de ellas se denomina DFS (recorrida en profundidad). La segunda se denomina BFS (recorrida en amplitud). Estudiaremos las estrategias para ambas recorridas en forma conceptual y posteriormente daremos una implementación de DFS en C++. BFS no será implementada todavía ya que requiere el estudio de nuevos conceptos que se verán más adelante en el curso.

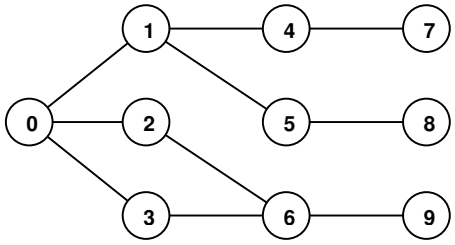
Depth-First Search - DFS

El algoritmo de búsqueda en profundidad (DFS por sus siglas en inglés), está basado en la idea de recorrer el grafo avanzando tanto como se pueda, hasta que en el momento en que no se puede avanzar más, se vuelve algún paso atrás hasta que nuevamente exista una nueva rama del grafo sobre la que se pueda avanzar.

Estrategia:

- Elegir un vértice v en el conjunto de vértices G(V) como punto de partida.
- Marcar el vértice para indicar que ya fue visitado
- Si hay un vértice adyacente a v que no haya sido visitado, elegirlo como nuevo punto de partida y repetir todo el proceso a partir de él. Cuando se retorna al vértice actual, se debe ver si hay otro vértice adyacente a v que aún no esté visitado. Este procedimiento se repite hasta que no haya vértices adyacentes a v sin visitar.
- Si sigue habiendo vértices de G sin marcar, elegir uno de ellos como punto de partida y repetir todo nuevamente.

Ejemplo:



Para este grafo, la recorrida es: 0, 1, 4, 7, 5, 8, 2, 6, 3, 9. Se partió del vértice 0 y se fueron eligiendo vértices siempre en orden creciente de numeración. La primera rama en completarse fue 0, 1, 4, 7. Cuando no se pudo avanzar más, se volvió hacia el vértice 1 y desde allí se continuó por la rama 5, 8. Cuando no se pudo avanzar más, se volvió hacia el vértice 0 y desde allí se continuó por la rama 2, 6, 3. Nótese que no se llegó al 0 nuevamente porque ya había sido visitado. Como no se pudo avanzar más por esa rama, se volvió hacia el vértice 6 y desde allí se continuó por la rama 9. Llegado este punto, no quedaban vértices sin visitar. Nótese que en este algoritmo, ningún vértice es visitado más de una vez.

Implementación en C++:

A continuación presentamos dos implementaciones en C++ para el algoritmo de DFS. La primera de ellas utiliza la representación de Matriz de adyacencia, mientras que la segunda utiliza la representación de Listas de adyacencia.

El lector notará que ambas implementaciones combinan iteración con recursividad. Esto es algo que hasta ahora no había sido visto frecuentemente. La iteración es necesaria para poder recorrer los vértices adyacentes al vértice actual. La recursividad es necesaria para continuar el procesamiento por la subrama correspondiente a cada uno de dichos vértices adyacentes, de acuerdo con la estrategia para DFS presentada anteriormente.

Utilizaremos además un arreglo de booleanos para marcar los vertices que van siendo visitados durante la recorrida. Se asume que dicho arreglo de booleanos se inicializa con todas sus celdas en `false` en forma previa a iniciar la recorrida.

Implementación usando Matriz de adyacencia:

```
const int N = /* cantidad de vertices */
typedef int Grafo[N][N];


void DFS (Grafo G, int verticeActual, boolean visitado[N])
{
      /* marco el vértice actual como visitado */
      visitado[verticeActual] = true;

      /* aquí va el procesamiento que le queramos dar
         al vértice actual antes de continuar por las
         subramas de sus vecinos */

      /* recorremos los vecinos del vértice actual */
      for (int j=0; j<N; j++)
      {
            /* si j es adyacente al vértice actual */
            if (G[verticeActual][j] == 1)
            {
                  /* si j no fue visitado, sigo por la
                     subrama que parte desde él */
                  if (!visitado[j])
                        DFS(G, j, visitado);
            }
      }

      /* aquí va el procesamiento que le queramos dar
         al vértice actual luego de procesadas las
         subramas de sus vecinos. */
}
```

Implementación usando Listas de adyacencia:

```
typedef struct nodo{ int vert;
                     nodo * sig;
                   }
typedef nodo * ListaAdy;

const int N = /* cantidad de vertices */
typedef ListaAdy Grafo [N];


void DFS (Grafo G, int verticeActual, boolean visitado[N])
{
      /* marco el vértice actual como visitado */
      visitado[verticeActual] = true;

      /* aquí va el procesamiento que le queramos dar
         al vértice actual antes de continuar por las
         subramas de sus vecinos */

      /* recorremos los vecinos del vértice actual */
      ListAdy aux = G[verticeActual];
      while (aux != NULL)
      {
            /* si aux->vert no fue visitado, sigo por la
               subrama que parte desde él */
            if (!visitado[aux->vert])
               DFS(G, aux->vert, visitado);

            aux = aux->sig;
      }

      /* aquí va el procesamiento que le queramos dar
         al vértice actual luego de procesadas las
         subramas de sus vecinos. */
}
```

Orden del algoritmo:

Es posible demostrar (pero no lo vamos a hacer) que el orden del algoritmo DFS es $O(\max\{|G(V)|, |G(A)|\})$ suponiendo un grafo conexo. Esto es, el máximo entre la cantidad de vértices y la cantidad de aristas del grafo. Esto bajo la hipótesis de que el procesamiento que le queramos dar al vértice actual es de O(1). En otro caso, se debe multiplicarle al orden de DFS el orden de ejecución de dicho procesamiento.

DFS es un caso particular de *backtracking*. Los algoritmos de backtracking consisten en la búsqueda exhaustiva de soluciones a problemas que pueden modelarse en forma arborescente, y en muchos casos, mediante grafos. Usualmente tienen orden de ejecución exponencial $O(a^n)$ que es mucho mayor que todos los órdenes estudiados en cursos anteriores. Sin embargo, DFS tiene un orden de ejecución notoriamente menor que muchos otros algoritmos de backtracking debido a que, por su naturaleza, aplica una estrategia que descarta muchas recorridas, reduciendo notoriamente su tiempo de ejecución.

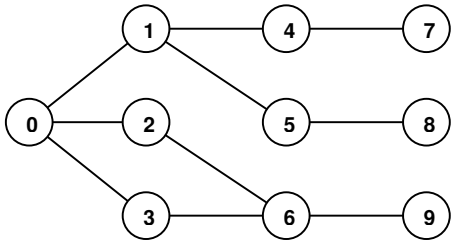
Breadth-First Search - BFS

El algoritmo de búsqueda en amplitud (BFS por sus siglas en inglés), está basado en la idea de recorrer el grafo visitando primero todos los vecinos posibles. Una vez visitados, se continúa por cada uno de ellos, repitiendo el mismo proceso. En cada paso, se descarta la visita a vértices que ya hayan sido visitados anteriormente.

Estrategia:

- Elegir un vértice v en el conjunto de vértices G(V) como punto de partida.
- Marcar el vértice para indicar que ya fue visitado
- Visitar todos los vecinos de v que aún no hayan sido visitados y marcarlos como visitados.
- Luego ir recorriendo dichos vecinos uno por uno en el mismo orden en que fueron visitados. En cada uno de ellos, repetir el punto anterior. En cada paso, descartar aquellos vértices que hayan sido visitados anteriormente.
- Si sigue habiendo vértices de G sin marcar, elegir uno de ellos como punto de partida y repetir todo nuevamente.

Ejemplo:



Para este grafo, la recorrida es: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Se partió del vértice 0 y se fueron eligiendo vértices siempre en orden creciente de numeración. Primeramente se visitaron los vecinos 1, 2, 3. Luego se fue al vértice 1 y desde allí se visitaron 4, 5. Luego se fue al vértice 2 y desde allí se visitó 6. Luego se fue al vértice 3, pero como su vecino 6 ya había sido visitado, no se visitó nuevamente. Después de eso se continuó por el segundo nivel de vecinos, respetando el orden en el cual fueron visitados los del primer nivel. Esto es, se fue al vértice 4 y desde allí se visitó el 7. Luego se fue al vértice 5 y desde allí se visitó el 8. Por último, se fue al vértice 6 y desde allí se visitó el 9. Llegado este punto, no quedaban vértices sin visitar. Nótese que en este algoritmo, ningún vértice es visitado más de una vez, al igual que en DFS.

Implementación en C++:

Como se dijo al inicio del capítulo, no se verá todavía una implementación para BFS porque requiere el estudio de nuevos conceptos que se verán más adelante en el curso.

Árboles de Cubrimiento de Costo Mínimo (ACCM)

Recordemos la definición de árbol de cubrimiento estudiada al inicio del curso:

> Sea G=(V, A) un grafo conexo. Se dice que G' = (V', A') es un árbol de cubrimiento de G si se cumple que G es un árbol, V' = V y A' $\subseteq$ A

Consideremos ahora que cada arista tiene asociado un costo. Esto puede modelarse de diversas formas. Una de ellas es considerar una función C: A $\rightarrow$ R tal que a cada arista de G le hace corresponder su costo asociado (un número real):

Se define el costo de un grafo como la suma de los costos de todas sus aristas:

$$C(G) = \sum_{a \in A} C(a)$$

Un problema importante en teoría de grafos consiste en encontrar un árbol de cubrimiento de un grafo G cuyo costo sea el menor posible. A este árbol se le suele llamar árbol de cubrimiento de costo mínimo (ACCM). Existen varios algoritmos que permiten encontrar dicho ACCM. En este curso estudiaremos uno de ellos, llamado Algoritmo de Kruskal.

Algoritmo de KRUSKAL

La manera en que funciona el algoritmo de Kruskal es ir agregando una a una las aristas a un bosque hasta llegar a un árbol que es la solución buscada.

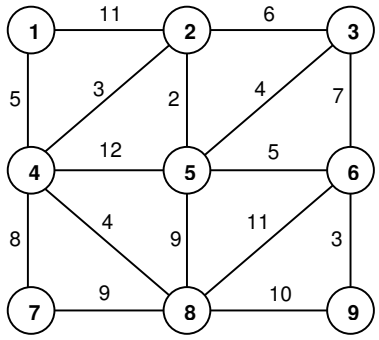
Estrategia:

- Se parte inicialmente de un bosque con todos los vértices del grafo G pero ninguna arista.
- Primero, se van eligiendo las aristas de G en forma ascendente según su costo.
- Se toma una arista en dicho orden, si el hecho de agregar esa arista al bosque introduce un ciclo, se descarta.
- En caso contrario se agrega al bosque. Así, se toman en este orden una a una cada una de las aristas, repitiendo el procedimiento hasta que el bosque sea conexo.

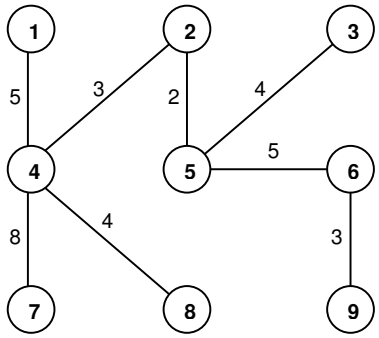
Seudocódigo:

```
Entrada:    Grafo G
Salida:     Grafo T (ACCM)
Método:     Inicializar T con los mismos vértices de G, pero sin aristas.
            MIENTRAS T sea disconexo HACER
                    Elegir la arista de G con menor costo y eliminarla de G.
                    SI dicha arista no genera un ciclo en T ENTONCES
                            Incorporar dicha arista a T
                    FIN
            FIN
```

Ejemplo:



Presentamos el ACCM resultante de aplicar el Algoritmo de Kruskal al grafo propuesto. Es tarea del lector verificar que efectivamente se trata de un ACCM.



Las aristas resultantes en el ACCM son las siguientes (fueron agregadas en orden creciente de costo):

- {2,5} de costo 2
- {2,4} de costo 3
- {6,9} de costo 3
- {3,5} de costo 4
- {4,8} de costo 4
- {1,4} de costo 5
- {5,6} de costo 5
- {4,7} de costo 8

Nótese que las aristas {2,3} y {3,7} del grafo original (con costos 6 y 7 respectivamente) fueron descartadas por generar ciclos.