GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: REPRESENTACIÓN COMPUTACIONAL DE LOS GRAFOS

1. Introducción

La teoría de grafo es un campo de estudio de las matemáticas y las ciencias de la computación, que estudia las propiedades de los grafos estructuras que constan de dos partes, el conjunto de vértices, nodos o puntos; y el conjunto de aristas, líneas o lados (edges en inglés) que pueden ser orientados o no. Por lo tanto también está conocido como análisis de redes.

La teoría de grafos es una rama de las matemáticas discretas y de las matemáticas aplicadas, y es un tratado que usa diferentes conceptos de diversas áreas como combinatoria, álgebra, probabilidad, geometría de polígonos, aritmética y topología.

2. Conocimientos previos

2.1. Struct

Las estructuras (también llamadas **struct**) son una forma de agrupar varias variables relacionadas en un solo lugar. Cada variable en la estructura se conoce como un miembro de la estructura.

A diferencia de una matriz, una estructura puede contener muchos tipos de datos diferentes (int, string, bool, etc.).

Para crear una estructura, use la palabra clave **struct** y declare cada uno de sus miembros entre llaves.

Después de la declaración, especifique el nombre de la variable de estructura (myStructure en el ejemplo siguiente):

```
struct{              // Declaracion de la estructura
   int myNum;          // Miembro (int variable)
   string myString;    // Miembro (string variable)
} myStructure;        // variable de tipo estructura
```

Para acceder a los miembros de una estructura, use la sintaxis de punto (.):

```
// Crear estructura variable llamada myStructure
struct {
   int myNum;
   string myString;
} myStructure;

// Asignar los valores a los miembros de myStructure
myStructure.myNum = 1;
myStructure.myString = "Hello World!";

// Imprimir los miembros myStructure
cout << myStructure.myNum << "\n";
cout << myStructure.myString << "\n";
```

2.1.1. Estructuras con nombre

Al darle un nombre a la estructura, puede tratarla como un tipo de datos. Esto significa que puede crear variables con esta estructura en cualquier parte del programa en cualquier momento.

Para crear una estructura con nombre, coloque el nombre de la estructura justo después de la palabra clave **struct**:

```
struct myDataType { // Esta estructura es llamada "myDataType"
   int myNum;
   string myString;
};
```

Para declarar una variable que usa la estructura, use el nombre de la estructura como el tipo de datos de la variable:

```
// Declarar una estructura llamada "car"
struct car {
   string brand;
   string model;
   int year;
};

int main(){
   //Crear una estructura de tipo carro y almacenar en la variable myCar
   car myCar1;
   myCar1.brand = "BMW";
   myCar1.model = "X5";
   myCar1.year = 1999;

   //Crear otra estructura de tipo carro y almacenar en la variable myCar2;
   car myCar2;
   myCar2.brand = "Ford";
   myCar2.model = "Mustang";
   myCar2.year = 1969;

   //Imprimir los miembros de una estructuras
   cout << myCar1.brand << " " << myCar1.model << " " << myCar1.year << "\n";
   cout << myCar2.brand << " " << myCar2.model << " " << myCar2.year << "\n";

   return 0;
}
```

2.2. Matrices

Una matriz es un arreglo de areglos fila, o más en concreto un arreglo de referencias a los arreglos fila. Con este esquema, cada fila podría tener un número de elementos diferente.

2.2.1. C++

Los arreglos bidimensionales o matrices en C++ se pude declarar similar a como se hace un arreglo unidimensional.

```
/*Se conoce de antemano las dimensiones esta manera es
estatica se recomienda que sea dinamica*/
int mat [3][4];

/*De forma dinamica*/
int ** mat;
mat = new int * [ncolumns];
for(int i=0;i<ncolumns;i++)
   mat[i]=new int [nfilas];

/*Con los valores conocidos*/
double carrots[3][4] {{2.5, 3.2, 3.7, 4.1},// primera fila
   {4.1, 3.9, 1.6, 3.5},// segunda fila
   {2.8, 2.3, 0.9, 1.1} // tercera fila
};
```

2.2.2. Java

Los arrays bidimensionales de Java se crean de un modo muy similar al de C++ (con reserva dinámica de memoria). En Java una matriz se puede crear directamente en la forma,

```
int [][] mat = new int[3][4];
```

o bien se puede crear de modo dinámico dando los siguientes pasos:

1. Crear la referencia indicando con un doble corchete que es una referencia a matriz,

   ```
   int[][] mat;
   ```

2. Crear el vector de referencias a las filas,

   ```
   mat = new int[nfilas][];
   ```

3. Reservar memoria para los vectores correspondientes a las filas,

   ```
   for(int i=0; i<nfilas; i++);
      mat[i] = new int[ncols];
   ```

A continuación se presentan algunos ejemplos de creación de arrays bidimensionales:

```
// crear una matriz 3x3
```

```
// se inicializan a cero
double mat[][] = new double[3][3];
int [][] b = {{1, 2, 3},
   {4, 5, 6}, // esta coma es permitida
};
int c = new[3][]; // se crea el array de referencias a arrays
c[0] = new int[5];
c[1] = new int[4];
c[2] = new int[8];
```

3. Desarrollo

Existen diferentes formas de representar un grafo (simple), además de la geométrica y muchos métodos para almacenarlos en una computadora. La estructura de datos usada depende de las características del grafo y el algoritmo usado para manipularlo. Entre las estructuras más sencillas y usadas se encuentran las listas y las matrices, aunque frecuentemente se usa una combinación de ambas. Las listas son preferidas en grafos dispersos porque tienen un eficiente uso de la memoria. Por otro lado, las matrices proveen acceso rápido, pero pueden consumir grandes cantidades de memoria.

La estructura de datos usada depende de las características del grafo y el algoritmo usado para manipularlo. Entre las estructuras más sencillas y usadas se encuentran las listas y las matrices, aunque frecuentemente se usa una combinación de ambas. Las listas son preferidas en grafos dispersos porque tienen un eficiente uso de la memoria. Por otro lado, las matrices proveen acceso rápido, pero pueden consumir grandes cantidades de memoria.

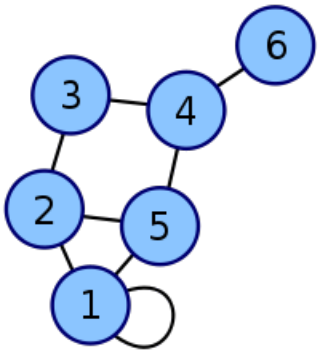

Figura 1: Ejemplo de grafo

Estructura de lista

- **Lista de incidencia:** Las aristas son representadas con un vector de pares (ordenados, si el grafo es dirigido), donde cada par representa una de las aristas.
- **Lista de adyacencia:** Cada vértice tiene una lista de vértices los cuales son adyacentes a él.

Esto causa redundancia en un grafo no dirigido (ya que A existe en la lista de adyacencia de B y viceversa), pero las búsquedas son más rápidas, al costo de almacenamiento extra. ListAdy={ {1,2,5}, {3,5}, {4}, {5,6} }

- **Lista de grados:** También llamada secuencia de grados o sucesión gráfica de un grafo no-dirigido es una secuencia de números, que corresponde a los grados de los vértices del grafo.LisGra=(4,3,3,3,2,1).

Estructuras matriciales

- **Matriz de adyacencia:** El grafo está representado por una matriz cuadrada M de tamaño n^2, donde n es el número de vértices. Si hay una arista entre un vértice x y un vértice y, entonces el elemento m x, y es 1, de lo contrario, es 0.

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Figura 2: Matriz de adyacencia del grafo

- **Matriz de incidencia:** El grafo está representado por una matriz de A. (aristas) por V (vértices), donde [vértice, arista] contiene la información de la arista (1 - conectado, 0 - no conectado).

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Figura 3: Matriz de de incidencia del grafo

4. Implementación

4.1. C++

```
/*
Esta variante es quizas la mas sencilla. Un grafo va ser un arreglo de una
   estructura llamada Node la cual tiene como atributo un vector con los
   indices de los nodos adyacentes al nodo almacenado en la posicion inesima
```

```
   del arreglo. La definicion de MAX\_N debe ser un valor igual o mayor que
   la cantidad de nodos del grafo que vas a representar con esta variante. Es
    muy utilizada cuando realizamos BFS y DFS.
*/
struct Node{
   vector<int> adj;
};
Node graf[MAX_N];

/*Es una extension de la primera en esta incorporamos un vector de los pesos
   de las aristas de los nodos adyacentes. Otra adecuacion es la variable
   dist que almacenara la distancia minima existen entre un nodo X y el que
   ocupa la posicion inesima. Esta variante es muy utilizada para un Dijkstra
   .*/
struct Node{
   int dist;
   vector<int> adj;
   vector<int> weight;
};
Node graf[MAX_N];

/*En determinados algoritmos de grafos se hace necesario trabajar con las
   aristas del grafo y para este elemento tambien podemos representarlo
   mediante la siguiente estructura. En este caso las variables a y b de la
   estructura son los nodos que componen la aristas mientras la variable
   weight contiene el valor de la aristas. E es el arreglo de las aristas del
    grafo mientras MAX\_N tiene que ser definida con valor igual o mayor a la
    cantidad de aristas del grafos.*/
struct Edge{
        int a, b;
        int weight;
};
Edge E[MAX_N];


/*Aqui cuando las aristas son dirigidas src es e origen y dst el destino*/
typedef int Weight;

struct Edge {
        int src, dst;
        Weight weight;
        Edge (int src, int dst, Weight weight):
        src (src), dst (dst), weight (weight) {}
};

bool operator <(const Edge & e , const Edge & f) {
        return e.weight != f.weight ? e.weight > f.weight : //!!INVERSE!!
        e.src != f.src ? e.src < f.src : e.dst < f.dst;
}
typedef vector <Edge> Edges;
```

```
typedef vector <Edges> Graph;
```

4.2. Java

```
class Graph {

  private int V;
  private LinkedList<Integer> adj[];

  public Graph(int v) {
     V = v;
     adj = new LinkedList[v];
     for(int i = 0; i < v; ++i) {
        adj[i] = new LinkedList();
     }
  }

  void addEdge(int v, int w) {
     adj[v].add(w);
  }
}
```

5. Complejidad

En este caso la complejidad no esta enfocado en cuanto al tiempo sino al uso de memoria. La representación de matrices siempre ocupa tiene un consumo de O($n * m$) o O($n * n$) donde n es la cantidad de vértices y m cantidad de aristas aunque no siempre se utilice toda la memoria. En caso de la representación de lista aunque su consumo puede llegar al mismo nivel que la representacion de matrices pero en la práctica es más eficiente que las matrices.

6. Aplicaciones

Lo tratado en esta guía tiene un grupo aplicaciones importante ya que es la base para la representación de los grafos sobre los cuales se van a modelar una determinada situación para ser resuelta con la utilización de un determinado algoritmo.

7. Ejercicios propuestos

Para este tema no existe ejercicios propuestos pero si es importante dominar el contenido abordado en esta guía ya que es la base para solucionar los problemas de teoría de grafos los cuales son algoritmos que parten de trabajar con un grafo el cual debe estar representado en una de las variantes abordadas en esta guía.

Algunos de los problemas más conocidos de grafos son:

- Conectividad Simple: Consiste en estudiar si el grafo es conexo, es decir, si existe al menos un camino entre cada par de vértices.
- Detección de Ciclos: Consiste en estudiar la existencia de al menos un ciclo en el grafo.
- Camino Simple: Consiste en estudiar la existencia de un camino entre dos vértices cualquiera.
- Camino de Euler: Consiste en estudiar la existencia de un camino que conecte dos vértices dados usando cada arista del grafo exactamente una sola vez. Si el camino tiene como inicio y final el mismo vértice, entonces se desea encontrar un tour de Euler.
- Camino de Hamilton: Consiste en estudiar la existencia de un camino que conecte dos vértices dados que visite cada nodo del grafo exactamente una vez. Si el camino tiene como inicio y final el mismo vértice, entonces se desea encontrar un tour de Hamilton.
- Conectividad Fuerte en Dígrafos: Consiste en estudiar si hay un camino dirigido conectando cada par de vértices del dígrafo. Inclusive se puede estudiar si existe un camino dirigido entre cada par de vértices, en ambas direcciones.
- Clausura Transitiva: Consiste en tratar de encontrar un conjunto de vértices que pueda ser alcanzado siguiendo aristas dirigidas desde cada vértice del dígrafo.
- Árbol de Expansión Mínima: Consiste en encontrar, en un grafo pesado, el conjunto de aristas de peso mínimo que conecta a todos los vértices.
- Caminos cortos a partir de un mismo origen: Consiste en encontrar cuales son los caminos más cortos conectando a un vértice v cualquier con cada uno de los otros vértices de un dígrafo pesado. Este es un problema que por lo general se presenta en redes de computadores, representadas como grafos.
- Planaridad: Consiste en estudiar si un grafo puede ser dibujado sin que ninguna de las líneas que representan las aristas se intercepten.
- Pareamiento (Matching): Dado un grafo, consiste en encontrar cual es el subconjunto más largo de sus aristas con las propiedad de que no haya dos conectados al mismo vértice. Se sabe que este problema clásico es resoluble en tiempo proporcional a una función polinomial en el número de vértices y de aristas, pero aun no existe un algoritmo rápido que se ajuste a grandes grafos.
- Ciclos Pares en Dígrafos: Consiste en encontrar en un dígrafo un camino de longitud par. Este problema puede lucir simple ya que la solución para grafos no dirigidos es sencilla. Sin embargo, aun no se conoce si existe un algoritmo eficiente para resolverlo.
- Asignación: Este problema se conoce también como pareamiento bipartito pesado (bipartite weigthed matching). Consiste en encontrar un pareamiento perfecto de peso mínimo en un grafo bipartito. Un grafo bipartito es aquel cuyos vértices se pueden separar en dos conjuntos, de tal manera que todas las aristas conecten a un vértice en un conjunto con otro vértice

en el otro conjunto.

- Conectividad General: Consiste en encontrar el número mínimo de aristas que al ser removidas separarán el grafo en dos partes disjuntas (conectividad de aristas). También se puede encontrar el número mínimo de nodos que al ser removidos separarán el grafo en dos partes disjuntas (conectividad de nodos).
- El camino más largo: Consiste en encontrar cual es el camino más largo que conecte a dos nodos dados en el grafo. Aunque parece sencillo, este problema es una versión del problema del tour de Hamilton y es NP-hard.
- Colorabilidad: Consiste en estudiar si existe alguna manera de asignar k colores a cada uno de los vértices de un grafo, de tal forma de que ninguna arista conecte dos vértices del mismo color. Este problema clásico es fácil para k=2 pero es NP-hard para k=3.
- Conjunto Independiente: Consiste en encontrar el tamaño del mayor subconjunto de nodos de un grafo con la propiedad de que no haya ningún par conectado por una arista. Este problema es NP-hard.
- Clique: Consiste en encontrar el tamaño del clique (subgrafo completo) más grande en un grafo dado.
- Isomorfismo de grafos: Consiste en estudiar la posibilidad de hacer dos grafos idénticos con solo renombrar sus nodos. Se conocen algoritmos eficientes para solucionar este problema, para varios clases particulares de grafos, pero no se tiene solución para el problema general. Este problema es NP-hard .