1. Juan Camilo Ruiz y Sergio Guzmán
2. ALGORITMO DE SOLUCIÓN
   1. CONSTRUCCIÓN DEL GRAFO:

Para construir el grafo se recibe la línea de entrada planteada en el ejercicio que describe a cada grafo y se guarda en el atributo difference la diferencia entre la cantidad de vértices de una bipartición con respecto a la otra. La estrategia para hallar la bipartición es aplicar un BFS desde cualquier vértice (en este caso se escoge el primero que aparece en la lista de aristas o el vértice 0 si no hay aristas). Cada vez que haya vértices de la primera bipartición aumenta el valor del atributo descrito antes, y cada vez que haya vértices de la segunda bipartición la decrece. Al final, se guarda en el atributo difference el valor absoluto del mismo. Cabe destacar que, al ser un grafo bipartito, cada vez que cambie de nivel en la búsqueda por anchura se le adicionan vértices a la siguiente bipartición que, al ser solo dos, se turnan de forma respectiva.

Para este grafo se utilizaron las siguientes funciones matemáticas de verificación del programa:

-

-

-

-

//Difference of the bipartition

public int difference;

/\*\*

\* Graph constructor that uses a given line with the structure (v e x1 y1 x1 y2... xk yk) to create a graph bipartition.<br>

\* @param line Given line with the form described above

\*/

public Graph(String line)

{

int[] bipartition;

//Q: The graph is bipartite

//Data split O(n)

String[] data=line.split(" ");

//Number of vertexes

int v=Integer.parseInt(data[0]);

//Adjacency matrix

boolean[][] adj= new boolean[v][v];

//Number of edges

int e=Integer.parseInt(data[1]);

//P1:2<=i<data.length ^ 0<=e<Integer.parseInt(data[1]) ^ AdjPairs(data,i,adj)

//t1:data.length-e

for(int i=2,j=0;i<data.length && j<e;i+=2,j++)

{

adj[Integer.parseInt(data[i])][Integer.parseInt(data[i+1])]=true;

adj[Integer.parseInt(data[i+1])][Integer.parseInt(data[i])]=true;

}

//R1: AdjPairs(data,data.length,adj)

//Marked array for each pair until data.length there is an edge in adj

boolean[] marked=new boolean[v];

bipartition= new int[2];

//Regular queue with O(1) operations

Queue<Integer> agenda=new LinkedList<>();

//By default the source is 0. Because the graph is connected, it must be able to get to all vertexes in time.

int source=0;

//Array that stores the value of the vertexes partition, which is either 0 or 1.

int[] value=new int[v];

//Adds one element to the first number of the bipartition

difference=1;

bipartition[0]=1;

marked[source]=true;

agenda.add(source);

int color=0;

value[source]=0;

//Because it expects connected graphs, the agenda will harness at most all vertexes (V)

//P2:!agenda.isEmpty() ^ bip(marked,value,0)=bipartition[0]^bip(marked,value,1)=bipartition[1]

//t2:agenda.size()

while(!agenda.isEmpty())

{

//New source of bfs

source=agenda.poll();

//Changes the value of the given vertex to the other possible bipartition for the children found in the bfs

color=(value[source]+1)%2;

//Cycle that goes through all the adjoint vertexes. O(V)

//P3:0<=i<v ^ markedValue(adj,source,i,marked,value)

//t3:v-i

for(int i=0;i<v;i++)

{

if(adj[source][i] && !marked[i])

{

bipartition[color]++;

marked[i]=true;

agenda.add(i);

value[i]=color;

}

}

//R3:markedValue(adj,source,v,marked,value)

}

//R2: agenda.isEmpty() ^ bip(marked,value,0)=bipartition[0]^bip(marked,value,1)=bipartition[1] ^ generalMarked(adj,v,marked,value)

//Difference stores the absolute value of the difference between both partitions

difference=Math.abs(bipartition[1]-bipartition[0]);

* 1. ALGORITMO: Para el algoritmo se utiliza la técnica de programación dinámica mirando si con una cantidad *N* de grafos se puede llegar a una diferencia *d* planteada. Al final, la respuesta será aquella que utilice una cantidad *N* de grafos en el menor *d* posible. Con el fin de plantear esta situación se tienen las siguientes expresiones y convenciones:
     1. h: Tabla de *hash* que tiene como llaves las diferencias planteadas y como valores una lista de los grafos que cumplen con esa condición (dichos grafos están enumerados de 0 hasta N-1).
     2. N: Cantidad total de grafos.
     3. D: Diferencia máxima de biparticiones encontrada en todos los grafos.
     4. b: Arreglo de booleanos que es de norma N y en cada posición i indica si el grafo se está utilizando para llegar a ese estado (*true*) o no (*false*). Dentro de los posibles estados de este arreglo se tienen los vectores **0** (todas las posiciones en *false*) y **1** (todas las posiciones en *true*).

Por último, se tiene la ecuación de recurrencia planteada:

Donde:

En resumidas cuentas, la ecuación siempre retorna un arreglo de booleanos en donde en cada posición se tiene si se está o no utilizando un grafo con el identificador dado para la solución. Por la forma en que se construye la ecuación, al consultar todas las soluciones m(N,d) para algún *d* entre *0* y *D*, solo se tendrán vectores **0** o **1**. Debido a esto, solo se debe consultar el menor d que no posea un vector **0** generando la solución buscada.

1. EXPLICACIÓN DE COMPLEJIDAD:
   1. Para lo referente a la construcción del grafo la operación más costosa que se tiene es un recorrido doble para realizar el BFS (que posee una complejidad máxima de O(V^2) donde V es el número de vértices del grafo) y a nivel espacial se tiene un espacio constante de tamaño V^2 en el peor caso (matriz de adjuntas de vértices). Como se deben construir n grafos, la complejidad estimada de este paso es de O(n\*V^2).
   2. En lo referente a la complejidad, la operación más costosa es el ciclo para los casos que no son base de la ecuación de recurrencia. En total, como debe revisar todos los números de grafos a utilizar, para cada uno analizar cada diferencia hasta llegar a D, y por cada uno de estos análisis revisar todas las diferencias del número anterior de grafos en el peor caso, en teoría se estima una complejidad de =(N\*D^2). Sin embargo, debido al uso del lenguaje *java*, para revisar que un arreglo de booleanos no sea el vector cero, se debe hacer un recorrido sobre todas las posiciones del arreglo en el peor caso (O(N)). Por ello, se estima en total una complejidad de O((N\*D)^2)