1. INTEGRANTES: 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 vértice 0) y que se guarden la cantidad de vértices de cada bipartición en un arreglo de dos posiciones llamado bipartition. Partiendo de la posición 0 en dicho arreglo, cada vez que haya un cambio de nivel en el BFS se adicionan los vértices a la siguiente bipartición y se cambia el índice actual. Al solo haber dos posiciones dicho índice (llamado color) solo podrá estar entre 0 y 1, y dependerá directamente de la bipartición a la que pertenecía el predecesor del vértice (cuya lista está en value). Al final, se guarda en el atributo difference el valor absoluto del mismo.

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.

A continuación se muestra el algoritmo enviado en el Proyecto. Cabe destacar que, debido a que *java* no permite utilizer expresiones lógicas matemáticas como comentarios en el código, se usaron las siguientes dos convenciones:

* ForAll es equivalente a .
* Exists es equivalente a .

/\*\*

\* Verifica que el arreglo dado no es el vector cero.<br>

\* @param bs Arreglo de booleanos.<br>

\* @return Si el arreglo es el vector cero o no.

\*/

private boolean notZero(boolean[] bs) {

//P: 0<=i<bs.length ^ (ForAll j|0<=j<i: !bs[j])

//t:bs.length-i

for(int i=0;i<bs.length;i++)

{

if(bs[i]) return true;

}

//R:0<=i<bs.length ^(ForAll j|0<=j<bs.length: !bs[j])

return false;

}

/\*\*

\*Retorna la diferencia mínima de haber sumado los grafos BC dados en el arreglo.<br>

\* @param graphs Arreglo con grafos que solo poseen la diferencia como atributo.<br>

\* @return Diferencia mínima de la suma de los grafos BC. En el caso de que el algoritmo no resuleva el problema, retorna -1.

\*/

private int minimumDifference(Graph[] graphs)

{

//Variables de máxima diferencia y de diferencia temporal

int maxDifference=0;

int dif=0;

//HashMap para las diferencias de los grafos. Por lo que se entiende, la complejidad de las operaciones de inserción y obtención de elementos del hash es constante.

HashMap<Integer,List<Integer>> hash= new HashMap<>();

//Ciclo para determinar la máxima diferencia O(N)

//P1: 0<=i<N ^ maxDifference=(max k|0<=k<i|:graphs[k].difference)

//t:N-i

for(int i=0;i<graphs.length;i++)

{

//Diferencia temporal

dif=graphs[i].difference;

if(dif>maxDifference)

{

maxDifference=dif;

}

//Si no eciste una lista de esta diferencia creéla

if(!hash.containsKey(dif))

{

hash.put(dif, new ArrayList<>());

}

//Añada a la diferencia un nuevo vértice i

hash.get(dif).add(i);

}

//R1: maxDifference=(max k|o<=k<N|:graphs[k])

//Variable para el arreglo de booleanos

boolean[] b=null;

//Se crea la estructura de la ecuación de recurrencia m usando el número toal de grafos N (graphs.length), la diferencia máxima D

//(maxDifference+1, para tener en cuemta una diferencia de cero) y una vez más el número total N para registrar en un arreglo de

//booleanos que grafos se están usando y cuáles no.

boolean[][][] matrix = new boolean[graphs.length][maxDifference+1][graphs.length];

//Itera únicamente sobre los elementos con diferencias registradas. En el peor de los casos se pasa nuevamente por el número total

//de grafos se asume complejidad O(N).

//P2:Asumiendo que se un conjunto se puede representar como un arreglo y que los valores i son dados en forma secuencial. Se tiene la siguiente invariante:

//0<=k<hash.keySet().size() ^ (ForAll j|0<=j<k: (Exists c in hash.get(j)|0<=c<N: unico(matrix[0][i],c))

//t2:hash.keySet().size()-k

//De esta forma se garantizan los dos casos bases de la ecuación de recurrencia m. Por razones de espacio la fila 0 correspondería a usar 1 grafo,

//la 1 2, y así.

for(Integer i:hash.keySet())

{

matrix[0][i][hash.get(i).get(0)]=true;

}

//Itera sobre todas las filas

boolean exists=true;

//Ciclo por cada fila de la matriz desde una cantidad de 2 grafos o desde la fila 1

//P3: 1<=i<N ^ s={z|0<=z<=D} ^ (ForAll i2|1<=i2<i: (ForAll j|O<=j<=D :

//(ForAll k|0<=k<j ^ matrix[i2][k]!=0:ParPosible(b,d,s)) ^ (ForAll k |0<=k<j ^ matrix[i2][k]==0: !ParPosible(b,d,s))))

//t3: N-i

//Por fuera tiene una complejidad de O(N). AL agregar su complejidad interna se tiene en total O((N\*D)^2)

for(int i=1;i<matrix.length;i++)

{

//P4: O<=j<=D ^ s={z|0<=z<=D} ^ (ForAll k|0<=k<j ^ matrix[i][k]!=0:ParPosible(b,d,s)) ^ (ForAll k |0<=k<j ^

//matrix[i][k]==0: !ParPosible(b,d,s))

//t4: D+1-j

//Por fuera tiene una complejidad de O(D). Al agregar su complejidad interna se tiene en total O(D^2\*N)

for(int j=0;j<matrix[0].length;j++)

{

exists=false;

//En total los ciclos de las invariantes P5.1 y P5.2 generan ua complejidad de O(D) por fuera.

//Internamente se realizan dos recorridos de complejidad O(N) en el peor de los casos, y ya que debe verificar si la matriz no es el vector cero

//y además revisar los grafos que hacen parte de la diferencia indicada en caso de que la primera condición sea cierta. Igual en el peor de los casos

// se topó con una diferencia que contiene todos los grafos dados por lo que se haría un segundo recorrido de O(N). Por esto se plantea una

//complejidad de O(N)+O(N), osea O(N)

//Teniendo en cuenta la complejidad interna se genera un total de O(N\*D)

//P5.1: exists=ParPosible(b,d,s) ^ O<=k<j-k ^ s={z|0<=z<k:z}U{z|j-k<=z<j|z}  ^ !exists

//t5.1: j/2-k

for(int k=0;k<j-k && !exists;k++)

{

//Evalua la posibilidad de que m(i-1)(k) no sea el vector cero y de que hallan grafos con diferencia j-k

if(hash.get(j-k)!=null && notZero(matrix[i-1][k]))

{

//Revisa los grafos con la diferencia dada

for(Integer g:hash.get(j-k))

{

//Si no se ha usado el grafo se añade a un nuevo arreglo de booleanos y se rompe el ciclo.

if(!matrix[i-1][k][g])

{

b=Arrays.copyOf(matrix[i-1][k], matrix[0][0].length);

matrix[i][j]=b;

matrix[i][j][g]=true;

exists=true;

break;

}

}

}

//Evalua la posibilidad de que m(i-1)(j-k) no sea el vector cero y de que hallan grafos con diferencia k

else if(hash.get(k)!=null && notZero(matrix[i-1][j-k]))

{

//Revisa los grafos con la diferencia dada

for(Integer g:hash.get(k))

{

if(!matrix[i-1][j-k][g])

{

//Si no se ha usado el grafo se añade a un nuevo arreglo de booleanos y se rompe el ciclo.

b=Arrays.copyOf(matrix[i-1][j-k], matrix[0][0].length);

matrix[i][j]=b;

matrix[i][j][g]=true;

exists=true;

break;

}

}

}

}

//R5.1: exists=(ParPosible(b,d,{i|0<=i<j})=(m(i,j)!=0))

//P5.2: exists=ParPosible(b,d,s) ^ j<=k<=D ^ s={z|0<=z<j:z}U{z|j<=z<=k}  ^ !exists

//t5.2: D-k

for(int k=j;k<matrix[0].length && !exists;k++)

{

//Evalua la posibilidad de que m(i-1)(k) no sea el vector cero y de que hallan grafos con diferencia k-j

if(hash.get(k-j)!=null && notZero(matrix[i-1][k]))

{

//Revisa los grafos con la diferencia dada

for(Integer g:hash.get(k-j))

{

//Si no se ha usado el grafo se añade a un nuevo arreglo de booleanos y se rompe el ciclo.

if(!matrix[i-1][k][g])

{

b=Arrays.copyOf(matrix[i-1][k], matrix[0][0].length);

matrix[i][j]=b;

matrix[i][j][g]=true;

exists=true;

break;

}

}

}

//Evalua la posibilidad de que m(i-1)(k-j) no sea el vector cero y de que hallan grafos con diferencia k

else if(hash.get(k)!=null && notZero(matrix[i-1][k-j]))

{

//Revisa los grafos con la diferencia dada

for(Integer g:hash.get(k))

{

//Si no se ha usado el grafo se añade a un nuevo arreglo de booleanos y se rompe el ciclo.

if(!matrix[i-1][k-j][g])

{

b=Arrays.copyOf(matrix[i-1][k-j], matrix[0][0].length);

matrix[i][j]=b;

matrix[i][j][g]=true;

exists=true;

break;

}

}

}

}

//R5.2: exists=(ParPosible(b,d,{i|0<=i<=D})=(m(i,j)!=0))

}

}

//P6: 0<=j<D+1 ^ (ForAll k|0<=k<j: !notZero(matrix[N-1][k])

//En otras palabras, para todas las diferencias antes de la que se está analizando, el arreglo de booleanos debe ser el

//vector cero. Si no se retorna el valor de la diferencia buscada.

//t6:D+1-j

for(int j=0;j<matrix[0].length;j++)

{

if(notZero(matrix[matrix.length-1][j]))

{

//j sería la mínima diferencia alcanzable

return j;

}

}

//R6: 0<=j<D+1 ^ (ForAll k|0<=k<=D: !notZero(matrix[N-1][k])

//En el caso improbable de que el algoritmo no encuentre alguna diferencia alcanzable con los datos dados retorna -1

return -1;

}

1. EXPLICACIÓN DE COMPLEJIDAD TEMPORAL:
   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).

EXPLICACIÓN DE COMPLEJIDAD ESPACIAL:

* Para la creación del grafo la estructura que ocupa más espacio es una matriz de adyacencia de tamaño ***V^2.*** Este espacio es fijo y no cambia durante la ejecución de la determinación de biparticiones.
* El máximo espacio ocupado depende de la estructura tridimensional creada para la programación dinámica del algoritmo, que ocupa un tamaño de ***N^2\*D***. Este espacio es fijo y no cambia a lo largo de la ejecución del algoritmo.

Por lo anterior se asume que a nivel de espacio, la complejidad ocupada del algoritmo es de ***N^2\*D***.

1. COMENTARIOS FINALES:

Llama bastante la atención que a pesar de ser un algoritmo con complejidad polinomial con potencia 4 se demore muy poco tiempo en correr. Se considera que esto se debe a que se utiliza una estructura de *HashMap* que minimiza la longitud del recorrido realizado y a su vez a que, en el momento en que encuentra una solución a la recurrencia, rompe los ciclos correspondientes y sigue, evitándose en muchas situaciones el peor caso calculado en la complejidad.