Càlcul de comunitats a grafs que representen xarxes socials

Projecte final LP 18-19

Les xarxes socials generen grafs de connexió entre les persones: qui és amic de qui, qui confia en qui etc. Volem fer la implementació d'un graf amb nodes que estan connectats entre si, i poder fer consultes sobre ell. La característica principal d'aquest graf és que és un graf dispers, per això volem utilitzar estructures de dades que no utilitzin una gran quantitat de memòria innecessària.

1. Introducció

Volem calcular les comunitats que apareixen a un graf dispers. Per fer-ho crearem un graf a partir de la matriu dispersa que heu creat a la primera part de la pràctica i després calcularem les comunitats que hi ha representades al graf construint un dendrograma.

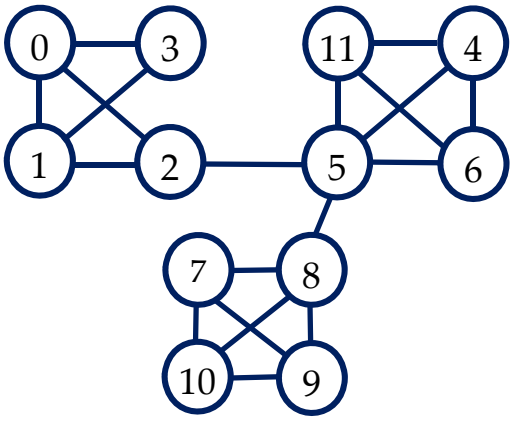
A grans trets una comunitat és un conjunt de nodes que estan més relacionats entre ells que amb la resta de nodes del graf. Si el graf representa una xarxa social serien els conjunts de persones que estan més relacionades entre elles, com per exemple els amics.

Hi ha molts algorismes per trobar aquestes comunitats als grafs però nosaltres hem seleccionat un que treballa amb un ventall d'estructures de dades molt interessant: grafs, arbres i heaps. És l'algorisme de Claudet i Newman: Aaron Clauset, MEJ Newman and Cristopher Moore. Finding community structure in a very large networks. **Phys. Rev. E** 70**, 066111. December 2004.** DOI: https://doi.org/10.1103/PhysRevE.70.066111

1.2 Representació de Grafs

Un graf es pot representar amb una matriu d'adjacència. Una matriu d'adjacència pot representar les connexions entre nodes d'un graf de la següent manera:

Suposem que tenim un graf que representa les relacions entre membres d'una xarxa social. Suposem que les relacions són simètriques. Això ho veiem representat en el graf següent i a la matriu d'adjacència que el representa.



A l'exemple, anomenat ***XarxaCom,*** tenim 3 comunitats representades. La formada pels nodes 0,1,2 i 3; la formada pels nodes 4,5,6 i 11 i finalment la formada pels nodes 7,8,9 i 10. El nostre objectiu és trobar aquestes comunitats.

relacions	0	1	2	3	4	5	6	7	8	9	10	11
0		X	X	X								
1	X		X	X								
2	X	X				X						
3	X	X										
4						X	X					X
5					X		X		X			X
6					X	X						X
7									X	X	X	
8						X		X	X	X	X	
9								X	X		X	
10								X	X	X		
11					X	X	X					

La representació de la matriu d'adjacència com a matriu sparse seria:

	fila	col	v
	0	1	1
	0	2	1
	0	3	1
	1	0	1
	1	2	1
	1	3	1
	2	0	1
	2	1	1
	2	5	1
	3	0	1
	3	1	1
	4	5	1
	4	6	1
	4	11	1
	5	4	1
	5	6	1
	5	8	1
	5	11	1
	6	4	1
	6	5	1
	6	11	1
	7	8	1
	7	9	1
	7	10	1
	8	5	1
	8	7	1
	8	8	1
	8	9	1
	8	10	1
	9	7	1
	9	8	1
	9	10	1
	10	7	1
	10	8	1
	10	9	1
	11	4	1
	11	5	1
	11	6	1

Matriu Sparse Columnes

IniFiles	Cols	Valors
0	1	1
3	2	1
6	3	1
9	0	1
11	2	1
14	3	1
18	0	1
21	1	1
24	5	1
29	0	1
32	1	1
35	5	1
	6	1
	11	1
	4	1
	6	1
	8	1
	11	1
	4	1
	5	1
	11	1
	8	1
	9	1
	10	1
	5	1
	7	1
	8	1
	9	1
	10	1
	7	1
	8	1
	10	1
	7	1
	8	1
	9	1
	4	1
	5	1
	6	1

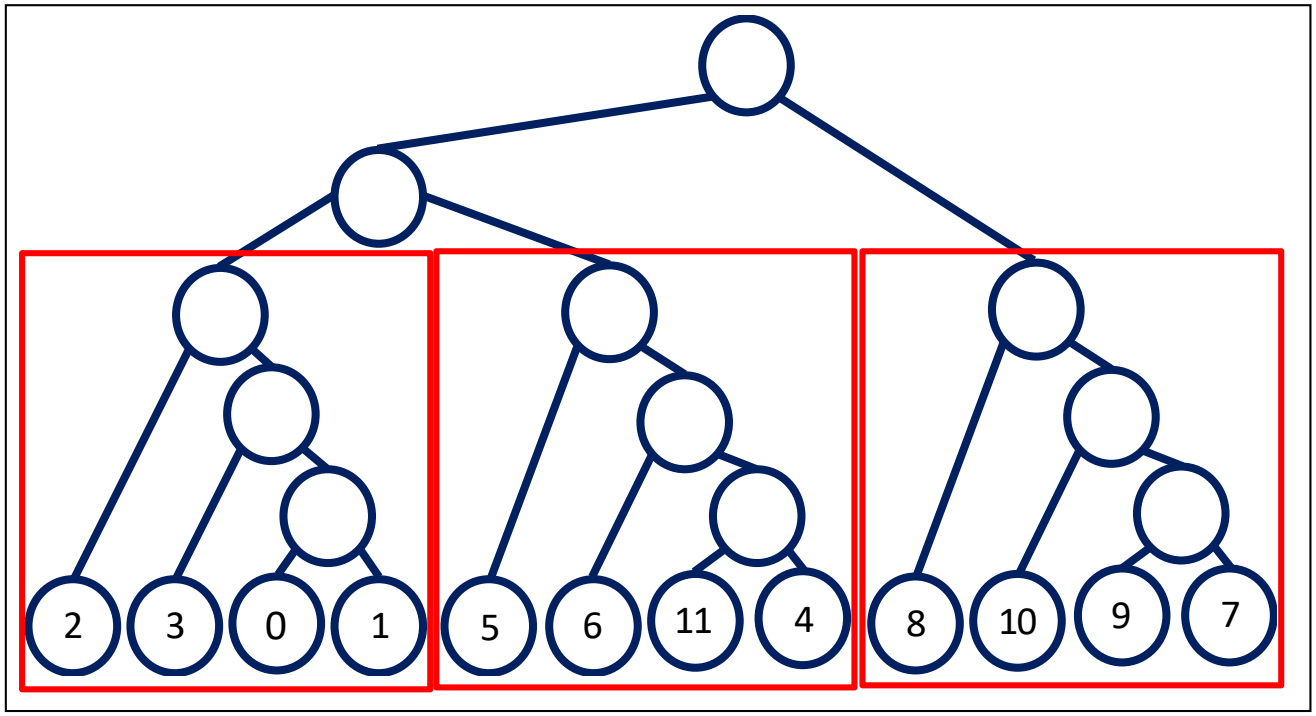
Matriu Sparse CSR

2. Última part de la pràctica

2.0 Algorisme

A partir d'un graf, on els nodes són les persones d'una xarxa social i les arestes les seves relacions, volem calcular quines comunitats existeixen a la xarxa social. Per fer-ho primer agruparem aquests nodes de dos en dos formant comunitats i després les comunitats també de dos en dos per formar comunitats més grans o agruparem comunitats amb nodes.

Donat el graf de l'exemple l'algorisme faria el següent:



Això seria un dendrograma que ens mostraria el procés que ha seguit l'algorisme per fer els agrupaments.

Per fer-ho es basarà en el concepte de **Modularitat (Q)**. La idea intuïtiva de la creació de comunitats ens indica que una separació correcte en comunitats donarà grups de nodes que estan molt relacionats entre ells i poc relacionats amb la resta. Per fer-ho, primer considerem que cada node forma una comunitat, i a partir d'aquí calcularem quin increment en aquesta modularitat es produiria si uníssim dos a dos les comunitats que tenim creades en un moment donat. Inicialment la modularitat és **Q=0**. Un cop fet aquest càlcul triem la unió de les comunitats **i** i **j** que incrementa més aquesta modularitat i recalculem els increments de modularitat de les relacions que afecten a les dues comunitats unides eliminant la **i** i recalculant les característiques de la **j** que ara representa a la unió de les dues. Això ho fem fins que la modularitat total es decrementa o fins que ja tot està unit.

Per fer tot això de manera eficient necessitem les següents estructures de dades:

1. Classe Graf: El graf que representa la xarxa. Tindrà la matriu d'adjacència representada per la matriu d'adjacència creada a la primera part de la pràctica (en CSR o columnes).

2. Classe Matriu: La matriu d'adjacència, que és la matriu de la primera part de la pràctica.

3. Classe Comunitat: S'encarregarà de crear les comunitats al graf i tindrà les següents estructures de dades per fer-ho:

3.1 m2: El nombre d'arestes del graf multiplicat per dos.

3.2 k: Un vector on, a cada posició **i,** guardarem la informació de la suma de graus de la comunitat **i**. Inicialment serà el grau dels nodes i a mesura que els anem agrupant s'anirà recalculant.

3.3 a: Un vector amb les dades de les proporcions d'arestes que arriben a una determinada comunitat. Inicialment per un node **j** serà **k[j]/m2** .

3.4 deltaQ: Un vector que contindrà, a cada posició **j,** la informació de l'increment de modularitat que es produiria per cada relació de la comunitat **j** amb cada una de les comunitats amb les que es relaciona. Aquesta informació és la que anirem modificant a mesura que unim comunitats i que necessitarem cercar a partir d'una parella de veïns donada (i,j). Per això la guardarem a **un arbre binari balancejat (map de la stl)** on tindrem un pair<int,int> amb la info dels índexs de la parella relacionada i el valor float del seu increment de modularitat. Els valors estaran ordenats segons <i,j>.

3.5 maxDeltaQ: Un vector que contindrà els valors màxims de cada fila de deltaQ. Amb informació dels nodes relacionats: **<pair<i,j>,deltaQ>.**

3.6 hTotal: Un **heap de màxims** que mantindrà la informació dels màxims de deltaQ de cada fila per tenir a dalt de tot el màxim, i poder-lo escollir a cada pas. Aquesta estructura està implementada a la **Classe Heap** que us donem i té unes particularitats que ens permeten anar modificant-lo sense haver de reconstruir-lo a cada pas. Mireu la descripció de la classe més endavant. Els seus elements seran de la classe **ElemHeap,** que us donem, i que conté un parell de sencers <i,j> que indiquen els índexs de les comunitats que uniríem, i un valor double que indica com s'incrementaria la modularitat si els uníssim. Els elements estaran emplaçats al heap seguint la propietat de màxims del heap en funció del seu valor double.

3.7 indexComs: Un vector de parelles de sencers que junt amb un sencer **primCom** ens indicaran quina és la relació de comunitats que estan actives i ens permetran recórrer el vector deltaQ visitant només les comunitats actives. Per fer-ho inicialment **primCom valdrà 0**, i cada posició de indexComs valdrà: **indexComs[i].first=i-1** ; **indexComs[i].second=i+1.** Així recorrerem totes les comunitats (inicialment els nodes del graf). En el moment que dues comunitats **i,j** s'ajuntin, quedarà només la j activa. En aquest moment la comunitat anterior a la i **(indexComs[indexComs[i].second)** haurà d'estar encadenada amb la comunitat següent a la i **(indexComs[indexComs[i].first)**. Per això:

indexComs[indexComs[i].first].second = indexComs[i].second i indexComs[indexComs[i].second].first = indexComs[i].first. L'índex **primCom** s'actualitzarà només si la comunitat eliminada és la primera.

3.8 vDendrogrames: Un vector d'apuntadors a Tree que mantindrà el dendrograma que estem creant amb les unions de comunitats. Per cada posició **j** del vector tindrem l'arbre que representa la comunitat que s'està creant en aquella posició **vDendrogrames[j]= adreça del Tree**. La idea és anar ajuntant els arbres a mesura que unim comunitats i fer que per les comunitats que desapareixen apuntin a NULL.

Algorisme:

1. Calculem m2: nombre d'arestes*2

2. Creem el vector k: k[j]= grau del node j

3. Creem el vector a: a[j]= k[j]/m2

4. Creem el vector IndexComs: on a cada posició:

indexComs[i].first=i-1 ; indexComs[i].second=i+1.

5. Inicialitzem l'índex **primCom a 0**.

6. Calculem deltaQ i maxDeltaQ: per cada fila **i** mirem les relacions del node amb la resta de nodes i per cada relació amb el node calculem **dQ=(1/m2) – ((k[i]*k[j]) / (m2*m2))** i l'afegim al map corresponent a la fila **deltaQ[i]** l'element **pair<pair<i,j>,dQ>** . A mesura que calculem els valors d'una fila anem guardant quin és el màxim que s'està generant. A mesura que fem aquest càlcul guardem el màxim de cada fila a **maxDeltaQ[i]** i l'afegim a **hTotal**.

8. Creem el vector **vDendrogrames,** creant un arbre (i reservant memòria per a ell per a cada un dels nodes del graf. L'arbre serà de doubles, pels nodes que representin les fulles (els nodes del graf) guardarem el seu índex, i per la resta la Q que tindríem en aquell moment de l'agrupament.

9. Mentre tinguem més d'una comunitat i no haguem acabat

7.1 Escollim i esborrem el màxim al heap.

7.2 si el màxim fa que la Q augmenti de valor

7.2.1 Fusionem les dues comunitats

7.3 Si no hem acabat.

10. Retornem la llista dels dendrogrames de vDendrogrames que pertanyen a comunitats actives segons **el vector IndexComs**.

El procés de **Fusionar** les comunitats i i j és com segueix: Volem eliminar la comunitat i i fer que la j sigui la que es mantingui com a fusió de les dues.

7.2.1.1 Recorrem els elements de **deltaQ[i] i deltaQ[j]** i mirem quins veïns tenen en comú i quins no:

A. Per cada veí de la i i per cada veí de la j mirem:

A. 1. Si és un veí en comú amb la j: anem a la seva deltaQ[vei] i busquem l'element <vei,i> agafant el seu valor d'increment de modularitat **deltaM(vei,i)** i el mateix per vei,j agafant el seu valor **deltaM(vei,j)**. Amb aquests valors modifiquem la relació del veí amb la j com a **deltaM(vei,j)**. = **deltaM(vei,i)+ deltaM(vei,j)**. I esborrem la relació del veí amb la i que ja desapareix. També anem a deltaQ[j] busquem la relació amb el veí i modifiquem la seva **deltaM(j,vei)** = **deltaM(i,vei)+ deltaM(j,vei)**.

A.2. Si només és un veí de la i: anem a la seva deltaQ[vei] i busquem l'element <vei,i> agafant el seu valor d'increment de modularitat **deltaM(vei,i).** Afegim un element a **deltaQ[vei] que té com a índexs de les comunitats (vei,j)** i com a valor d'increment de la modularitat **deltaM(vei,j)**. = **deltaM(vei,i) – 2 A[j]* A[vei]**. I esborrem la relació del veí amb la i que ja desapareix. També anem a deltaQ[j] i afegim una relació amb el veí (j,vei) afegint el seu increment de modularitat **deltaM(j,vei) = deltaM(i,vei)- 2 A[j]* A[vei]**.

Un cop fet això mirem els veïns de j

A.3. Per cada veí que només està a j. Anem a la seva deltaQ[vei] i busquem l'element <vei,j> agafant el seu valor d'increment de modularitat **deltaM(vei,j).** Modifiquem la relació del veí amb la j com **deltaM(vei,j)**. = **deltaM(vei,j) – 2 A[i]* A[vei]**. També anem a deltaQ[j] i modifiquem el seu valor d'increment de modularitat amb el veí com a **deltaM(j,vei) = deltaM(j,vei)- 2 A[i]* A[vei]**.

A4. Mantenim el màxim de j i de cada veí al vector **maxDeltaQ** i modifiquem el seu valor de deltaM al heap hTotal si és necessari.

A5. Eliminem tots els veïns de i, per deixar la posició i de deltaQ buida..

B. Un cop fet això:

B1. Marquem la comunitat **i** com a esborrada al **vector IndexComs**.

B2. Fusionem els arbres de les posicions **i** i **j** del vector **vDendrogrames** posant-los a la posició **j** del vector.

B3 Recalculem la **A[j]=A[j]+A[i]**

2.1 Classe Graf

Us donem la classe graf que té com a matriu d'adjacència la matriu d'adjacència creada a la primera part de la pràctica per vosaltres.

```
#include "Comunitat.h"

class Graph
{
public:
	Graph() {};
	Graph(string nomFitxerRels):mAdjacencia(nomFitxerRels){ };
	~Graph() {};
	friend std::ostream& operator<<(std::ostream& os, const Graph& g)
	{ os << g._mAdjacencia; return os; };
	void calculaComunitats(list<Tree<double>*>& listDendrogram);
	void clear() { m_mAdjacencia.clear(); }
private:
	MatriuSparseCSR _mAdjacencia;
};
```

Volem afegir el mètode calculaComunitats:

```
void calculaComunitats(list<Tree<double>*>& listDendrogram);
```

que definirà un element de tipus Comunitats i cridarà al mètode de la classe comunitats del mateix nom.

Hem afegit mètode clear per alliberar memòria i poder fer proves encadenades de diferents grafs.

2.2 Classe ElemHeap

Us donem la classe ElemHeap per guardar els elements (i,j,deltaQ) a Heap, on i,j seran els índexs de dues comunitats i deltaQ l'increment de modularitat si els agrupéssim.

```
class ElemHeap
{
public:
	ElemHeap() { m_Val = 0; m_Pos = { 0, 0 }; }
	ElemHeap(double val, pair<int, int> pos) { m_Val = val; m_Pos = pos; };
	double getVal() const { return m_Val; }
	pair<int,int> getPos() const { return m_Pos; }
	bool operator<(const ElemHeap& e) { return (m_Val < e.m_Val); }
	bool operator<=(const ElemHeap& e) { return (m_Val <= e.m_Val); }
	bool operator>(const ElemHeap& e) { return (m_Val > e.m_Val);}
	bool operator>=(const ElemHeap& e) { return (m_Val >= e.m_Val); }
	bool operator==(const ElemHeap& e) { return ( m_Val == e.m_Val);  }
	~ElemHeap() {};
	ElemHeap& operator=(const ElemHeap& e);
	friend std::ostream& operator<<(std::ostream& out, const ElemHeap& elHeap);
private:
	double m_Val;
	pair<int, int> m_Pos;
};
```

Aquesta classe guarda els índexs (i,j) de les comunitats que volem agrupar i el seu valor deltaQ associat. Veiem que els operadors de comparació actuen només sobre el valor deltaQ i per això si volem cercar un valor de la parella(i,j) no ho podem fer directament.

2.3 Classe Heap

Us donem la classe Heap per guardar els màxims per cada fila de (i,j,deltaQ)

```
class Heap {
public:
	Heap() { m_act = -1; m_maxEls = 0; };
	Heap(int maxEls);
	Heap(const Heap& h);
	~Heap() {};
	Heap& operator=(const Heap& h);
	ElemHeap& max() { return m_data[0]; }
	int size() { return m_act+1; }
	friend std::ostream& operator<<(std::ostream& out, const Heap& h);
	void insert(const ElemHeap& el);
	void resize(int mida);
	void delMax();
	void modifElem(const ElemHeap& nouVal);
	void clear();
	bool operator==(const Heap& h);
private:
	vector<ElemHeap> m_data;
	//Guardem indexs del vei inicial per cada un dels valors que tenim guardats
	vector<int> m_index;
	int m_maxEls; //indica nombre total de nodes: array va de 0 a m_maxEls-1
	int m_act; //indica posicio ultima ocupada: inicialment -1
	int pare(int pos) const { return ((pos - 1) / 2); }
	int fillEsq(int pos) const  { return ((2 * pos) + 1); }
	int fillDret(int pos) const { return ((2 * pos) + 2); }
	void intercanvia(int pos1, int pos2);
	std::ostream& printRec(std::ostream& out,int pos,int n) const;
	void ascendir(int pos);
	void descendir(int pos);
};
```

Guarda valors ElemHeap i manté un vector amb les posicions al vector m_data dels valors <i,j> deltaQ per cada i que tenim, on la i és el valor de la comunitat:

- `void resize(int mida)`: Fer un resize de mida per poder inicialitzar la mida del vector que guardarà les dades. Inicialment necessitem que sigui de la mida del nombre de nodes que té el graf.
- `void insert(const ElemHeap& el)`: Permet inserir un element de tipus ElemHeap seguint les propietats de heap de màxims pel seu valor de deltaQ.
- `ElemHeap& max()`: retorna el màxim del heap.
- `void delMax()`: Elimina el màxim recomposant el Heap perquè compleixi les propietats de HEap de màxims.
- `void modifElem(const ElemHeap& nouVal)`: Modifica el valor j i deltaQ d'un element (i,j,deltaQ). La idea és que al heap tinguem una entrada per cada comunitat i, i sabem en tot moment a quina posició del vector es troba gràcies al vector m_index. Per això podem modificar els valors associats a una comunitat i. És a dir quina és la deltaQ màxima que pot generar en un moment donat i amb quina altra comunitat la genera. Aquesta modificació es fa mantenint les propietats de heap de màxims.
- `void clear()`: Allibera la memòria del Heap i el deixa buit.

2.4 Classe Tree

Us donem la classe Tree

```
template <class T>
class Tree {
public:

	Tree();
	Tree(const T& data);
	Tree(const Tree<T>& t);
	Tree(string nomFitxer);
	~Tree();
	bool isLeave() const { return ((m_left == NULL) && (m_right == NULL)); }
	bool isEmpty() const { return (m_data == NULL);}
	Tree<T>* getRight(){return m_right;}
	Tree<T>* getLeft() { return m_left; }
	void setRight(Tree<T>* tR);
	void setLeft(Tree<T>* tL);
	T& getData() { return (*m_data); }
	friend std::ostream& operator<<(std::ostream& out, const Tree<T>& t);
private:
	Tree<T>* m_left;
	Tree<T>* m_right;
	Tree<T>* m_father;
	T* m_data;
	void TreeRec(ifstream& fitxerNodes, int h, Tree<T>* father);
	std::ostream& coutArbreRec(int n, std::ostream& out) const;
};
```

Permet crear un arbre i assignar-li els seus fills esquerre i dret sense crear nous arbres. El que volem es poder crear el dendrograma sense haver de reconstruir cada cop els arbres. El que farem és crear els nodes intermedis però aprofitarem els ja creats per anar-los agrupant en arbres més grans. Per això hem definit els mètodes setLeft i setRight que no reserven nova memòria:

```
template<class T>
void Tree<T>::setLeft(Tree<T>* tL)
{	m_left = tL;
	if (m_left != NULL)
	{	//Fem que this sigui el pare de left
		m_left->m_father = this;
	}
}
template<class T>
void Tree<T>::setRight(Tree<T>* tR)
{	m_right = tR;
	if (m_right != NULL)
	{	//Fem que this sigui el pare de right
		m_right->m_father = this;
	}
}
```

2.5 Classe Matriu

Heu d'afegir els següents mètodes a la classe matriu que heu implementat:

- `int getNValues() const { return _Valors.size(); }`: ens retorna el nombre d'arestes que hi ha al graf.
- `void calculaGrau(vector<int>& graus) const`: calcula el grau de cada node i el retorna a un vector. Serà el nostre k inicial.
- `void creaMaps(vector<map<pair<int, int>, double>>& vMaps) const`: Per cada node ens retorna un map amb un parell d'enters que representen els indexos de les comunitats (en aquest cas encara nodes) i un 0 com a deltaQ. És important que ho faci la matriu perquè és qui te la informació de les relacions entre nodes.
- `void calculaDendrograms(vector<Tree<double>*>& vDendograms) const`: Genera els primers dendrogrames, és a dir un vector amb tantes posicions com nodes té el graf i un arbre a cada una que conté un sol node amb la informació de l'índex del node corresponent.
- `void clear()`: Allibera tot l'espai de la Matriu i la deixa buida.

2.5 Classe Comunitat

La classe comunitat és la que heu d'implementar vosaltres.

```
#include "MatriuSparse.h"
#include <list>
#include "Heap.h"
class Comunitat
{
public:
	Comunitat(MatriuSparse* pMAdj);
	~Comunitat();
	void calculaM2() { m_M2 = m_pMAdj->getNValues(); }
	void calculaK() { m_pMAdj->calculaGrau(m_k); };
	void calculaA();
	void creaDeltaQHeap();
	void creaIndexComs();
	void InicialitzaDendrograms(){ m_pMAdj->calculaDendograms(m_vDendrograms); }
	void calculaComunitats(list<Tree<double>*>& listDendrogram);
	void fusiona(int com1, int com2);
	void modificaVei(int com1, int com2, int vei, int cas);
	int getM2() { return m_M2; }
	vector<int> getK() { return m_k; }
	vector<double> getA() { return m_A; }
	vector<map<pair<int, int>, double>> getdeltaQ() {return m_deltaQ;}
	Heap gethTotal() {return m_hTotal;}
	vector<pair<int, int>> getIndexComs() { return m_indexComs; }
	void clear();
private:
	//vector de maps per cada fila de deltaQij
	vector<map<pair<int, int>, double>> m_deltaQ;
	//vector mante llista de comunitats actives amb un parell que indica anterior i seguent activa.
	//la primera te com anterior -1 i la ultima com a seguent la mida del vector
	vector<pair<int, int>> m_indexComs;
	//vector que mante el maxim de deltaQij d'una fila, per no recalcular innecesariament
	vector<pair<int, double>> m_maxDeltaQFil;
	//index que indica quina es la primera comunitat activa
	int m_primComdeltaQ;
	//Vector d'arbres per dendrograma. Inicialment vector amb un arbre per cada node del graf.
	//Cada node fulla te index del node, cada node intern i l'arrel te el Q en el moment de la unio
	vector<Tree<double>*> m_vDendrograms;
	//Vector de graus dels nodes
	vector<int> m_k;
	//Vector index a de cada aresta
	vector<double> m_A;
	//Heap maxims deltaQ per fila. Num elems = comunitats actives (a inici = num nodes graf)
	Heap m_hTotal;
	//Modularitat Q de la particio en comunitats calculada
	double m_Q;
	//Nombre d'arestes *2 del graf
	int m_M2;
	MatriuSparse* m_pMAdj;
};
```

Tindrem els següents mètodes i els que puguin sorgir a partir d'ells:

- Comunitat(MatriuSparse* pMAdj): Inicialitzarà atributs i m_pMAdj al valor que li passem. Aquest punter a matriu el fem servir per poder fer els càlculs parcials del main sense tenir un graf associat.
- ~Comunitat();
- void calculaM2() { m_M2 = m_pMAdj->getNValues(); }: Calcula el valors de les connexions * 2 del graf, ho fa cridant a un mètode de la matriu (que haureu d'implementar).
- void calculaK() { m_pMAdj->calculaGrau(m_k); };: Calcula el grau de cada node del graf. Ho fa cridant a un mètode que s'haurà d'implementar a la Matriu.
- void calculaA(): Calcula el vector A tal com s'especifica a l'apartat 2.0.
- void creaDeltaQHeap();:Crea el vector m_deltaQ. Per fer-ho necessitarà cridar al mètode creaMaps de la matriu d'adjacència, que generarà tots els elements (i,j) dels maps de cada fila. Després de cridar a aquest mètode omplirà els valors del map creat calculant deltaQ. A mesura que es faci això anirem omplint el vector m_maxDeltaQFil amb els màxims corresponents a cada fila. A mesura que omplim cada fila i tinguem el seu màxim, afegirem aquest màxim al heap m_hTotal amb els seus valors inicials. Tot això tal com s'explica a l'apartat 2.0 d'inicialitzacions de les diferents estructures.
- void creaIndexComs();: Inicialitza el vector IndexComs tal com s'especifica a l'apartat 2.0.
- void InicialitzaDendrograms(){ m_pMAdj->calculaDendograms(m_vDendrograms); }
- void calculaComunitats(list<Tree<double>*>& listDendrogram); Calcularà les comunitats i les retornarà a una llista d'apuntadors a arbres que seran els dendrogrames. Es calcularà seguint l'algorisme presentat a l'apartat 2.0.
- void fusiona(int com1, int com2); Fusionarà dos comunitats recalculant les seves dades tal com s'explica a l'apartat 2.0.
- void modificaVei(int com1, int com2, int vei, int cas);: Modificarà els veïns de dues comunitats fusionades.
- int getM2() { return m_M2; }
- vector<int> getK() { return m_k; }
- vector<double> getA() { return m_A; }
- vector<map<pair<int, int>, double>> getdeltaQ() {return m_deltaQ;}
- Heap gethTotal() {return m_hTotal;}
- vector<pair<int, int>> getIndexComs() { return m_indexComs; }
- void clear();: Allibera tota la memoria de la comunitat i la buida.

3. Dades

Utilitzarem 3 grafs representats de forma ordenada com els que teníem a la part de Matrius però aquest cop simètrics. Per tant sempre trobarem al fitxer les relacions del node i al j i del j a l'i.

1. XarxaCom.txt: Té l'estructura de la xarxa de l'exemple amb 12 nodes.

2. ZackaryKarate.txt: Té l'estructura d'una comunitat de karate molt utilitzada en construcció de comunitats. Els nodes estan numerats de l'1 al 34 i volem mantenir aquesta numeració per això us sortirà sempre un node addicional 0 que no està relacionat amb cap altre.

3. EpinionsSimetricOrdenat.txt: És la xarxa que ja coneixeu però aquest cop l'hem fet simètrica per tal de poder utilitzar aquest algorisme.