Réseaux et théorie des graphes

Le géographe est sans cesse confronté à des réseaux. Certains sont visibles, comme la plupart des réseaux de transport. Mais beaucoup demeurent invisibles, tels les flux d'information, financiers ou certains réseaux sociaux.

Avant d'aborder le traitement de ces réseaux, il est souvent nécessaire de les modéliser sous la forme de graphe pour les analyser à l'aide de la théorie des graphes, puis en leur appliquant d'autres fonctions. CE NOTEBOOK NE TRAITE QUE DE LA STRUCTURE DES RÉSEAUX. Nous aborderons ultérieurement les hypergraphes et les flux dans un réseau.

Construire des graphes de réseaux

Le géographe dispose de données qu'il veut traduire en réseau, par exemple construire un réseau de villes à partir des coordonnées d'un ensemble de villes, extraire un réseau de routes d'une carte ou un réseau fluvial d'un MNT, ou construire un réseau social.

Construire un premier graphe

Les graphes se construisent avec la fonction **Graph**[]. Voici deux graphes. Le deuxième est en partie un graphe orienté, synonyme d'un sens unique. De plus les noeuds sont étiquetés avec l'option *VertexLabels->"Name"*.

```
In[ ]:= Graph[{1 ⟷ 2, 2 ⟷ 3, 2 ⟷ 4, 4 ⟷ 5, 3 ⟷ 1}]
        graphe
```

```
Graph[{1 ↔ 2, 2 ↔ 3, 2 ↔ 4, 4 ↔ 5, 3 ↔ 1}, VertexLabels → "Name"]
graphe                                      étiquettes de sommet
```

Out[ ]=

Out[ ]=

Cette fonction **Graph**[] dispose d'un très grand nombre d'options et de sous-options. Mais, la construction d'un graphe de cette façon peut devenir difficile pour un grand nombre de données, que le géographe a récolté. D'autres solutions sont à envisager.

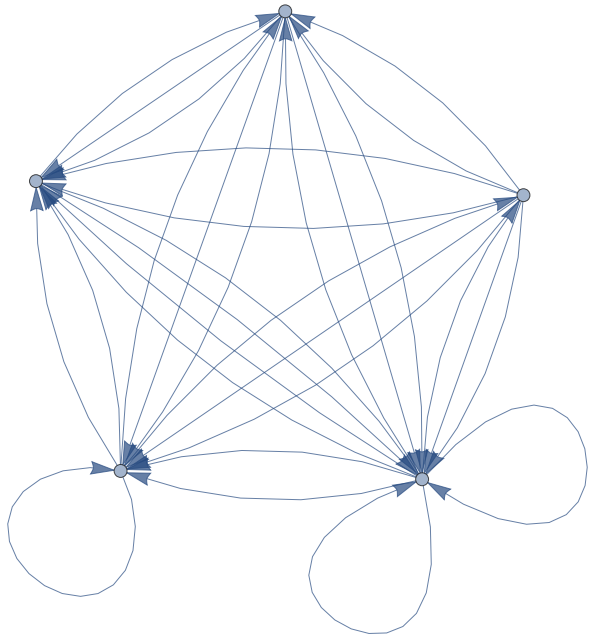
Construire un réseau à partir de données matricielles

Souvent le géographe dispose d'une matrice qui indiquent des relations entre des lieux. Citons les matrices d'adjacence, nominale ou pondérées, et les matrices d'incidences. Les graphes sont élaborés avec les fonctions **AdjacencyGraph**[], **IncidenceGraph**[] ou **WeigthAdjacencyGraph**[]. Le petit programme montre comment construire de tels graphes à partir d'une matrice d'adjacence

aléatoire que nous créons avec la fonction **RandomInteger**[]. Souvent, au grès du tirage au sort, des relations d'un noeuds sur lui-même apparaissent :

```
In[ ]:= mat = RandomInteger[2, {5, 5}]
        nombre entier aléatoire
        AdjacencyGraph[mat]
        graphe d'adjacence
Out[ ]= {{0, 0, 2, 2, 2}, {2, 0, 2, 2, 2}, {1, 0, 0, 2, 0}, {1, 1, 2, 2, 2}, {1, 1, 2, 0, 1}}
```

Out[ ]=



Construire un réseau de villes à partir de leur localisation

Quand on dispose d'un fichier de localisation, les transformations à réaliser dépendent précisément des données. Pour construire le graphe d'un réseau de villes françaises, le programme ci-dessous convient. Nous avons été aidé pour élaborer ce programme. Nous donnons ultérieurement une solution plus brève réalisée avec la version 12.2 du logiciel. En tapant 10 dans la fenêtre qui apparaît sur l'écran, le programme construit le graphe de toutes les relations d'ordre 1 possibles entre les 10 plus grandes villes françaises.

In[ ]:=
```
ClearAll["Global`*"]
```
efface tout

```
ny = Input["Choisir le nombre de villes"] // ToExpression;
```
entrée — en expression

```
cityCoords = (CityData[#, "Coordinates"] & /@ CityData[{All, "France"}])[[ ;; ny]];
```
données de villes — données de v· — tout

```
nom = (CityData[#, "Name"] & /@ CityData[{All, "France"}])[[ ;; ny]];
```
données de villes — données de v· — tout

```
geoGridPos = GeoGridPosition[GeoPosition[#, "WGS84"], {"UTMZone31",
```
position de grille géog··· — position géographique

```
      "CentralScaleFactor" → 0.9996, "GridOrigin" → {500 000, 0}}][[1]] &;
cityCoordsxy = geoGridPos /@ cityCoords;


<< ComputationalGeometry`
delval = DelaunayTriangulation[cityCoordsxy];
del = Flatten[Thread[Rule @@ #] & /@ delval];
```
aplatis — enfile — règle

```
vl = VertexList[Graph[del]];
```
liste de somm··· — graphe

```
noma = Table[nom[[i]], {i, vl}];
```
table

```
coord = Table[cityCoordsxy[[i]], {i, vl}];
```
table

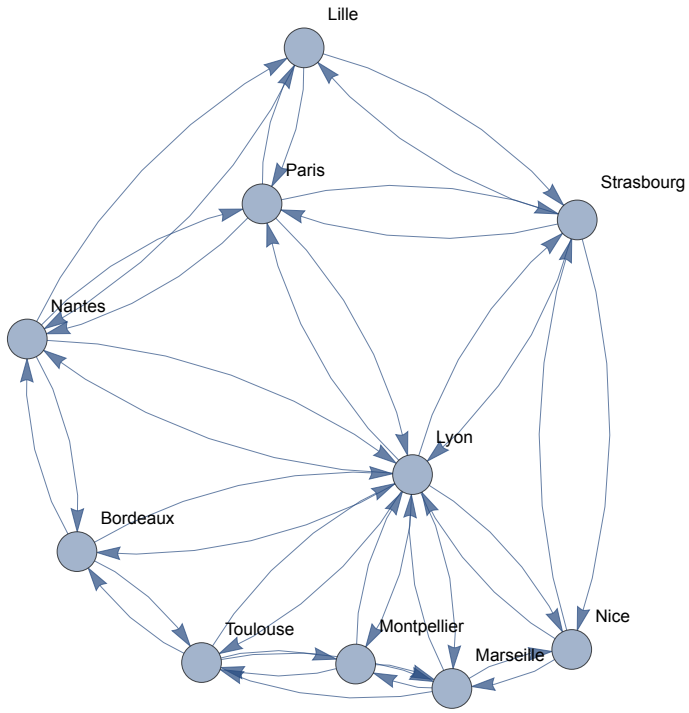
```
delval = DelaunayTriangulation[coord];
del = Flatten[Thread[Rule @@ #] & /@ delval];
```
aplatis — enfile — règle

```
gville = Graph[del, VertexSize → Large,
```
graphe — taille de sommet — large

```
  VertexLabels → Table[i -> noma[[i]], {i, Length[vl]}],
```
étiquettes de sommet — table — longueur

```
  VertexCoordinates → Table[i → coord[[i]], {i, Length[vl]}]]
```
coordonnées de sommet — table — longueur

Out[ ]=



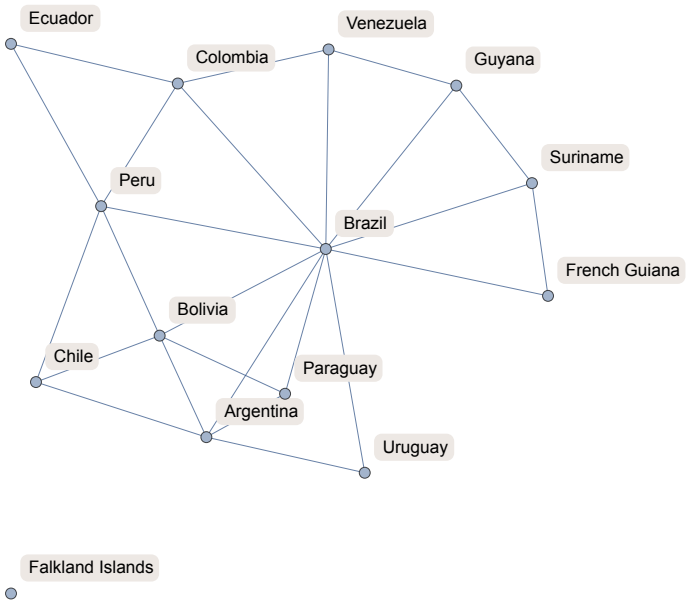
Plusieurs options sont retenues pour bien positionner les villes dans l'espace.

Pour des mailles ou des champs, il est souvent utile de construire le graphe des relations, par exemple les relations frontalières entre des États. Ce qui s'obtient avec la fonction **RelationGraph**[]. Voici un exemple recopié de l'aide disponible pour cette fonction :

In[ ]:=

```
countries = CountryData["SouthAmerica"];
            données de pays
RelationGraph[MemberQ[CountryData[#2, "BorderingCountries"], #1] &,
graphe de relation   appartien···  données de pays
 countries, VertexLabels → "Name"]
            étiquettes de sommet
```

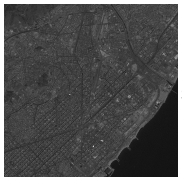
Out[ ]=



Le Brésil a évidemment le plus de frontières avec les autres États, alors que les îles Faklands sont isolées. Enfin, sans être complet, soulignons l'intérêt des graphes de voisinage construits avec la

fonction **NearestNeighborGraph[]**. Ils permettent de relier sous forme de graphe tout objet. Dans le programme ci-dessous l'image satellite (512 x 512 pixels) de la ville de Barcelone est d'abord divisée en 16 images avec la fonction **Partition[]**. Puis un graphe est réalisé entre les images qui se ressemblent. Vous pouvez remplacer l'image initiale par une autre l'image de votre choix et modifier la taille des images issues de la partition (128 pixels dans cet exemple).

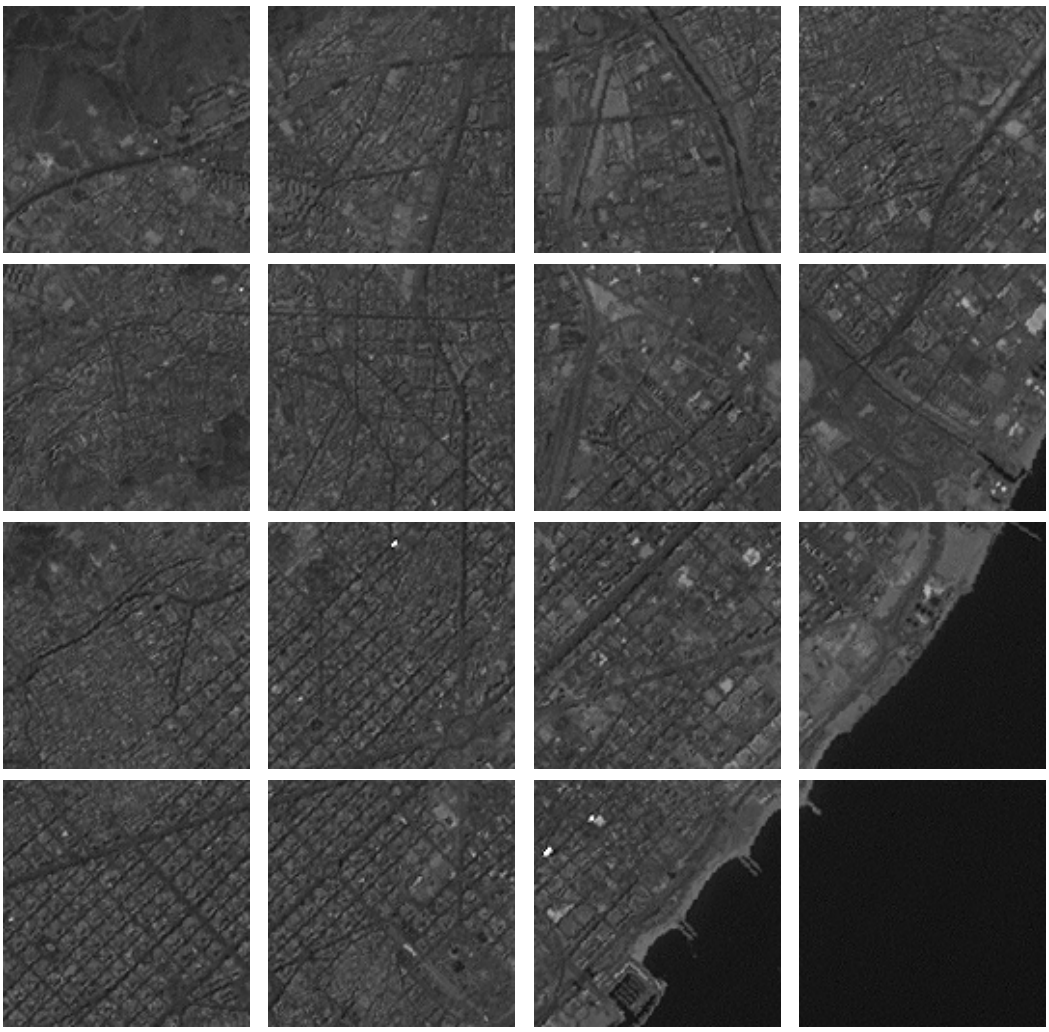
In[ ]:=
```
(g = ImagePartition[ , 128]) // Grid
```
partition d'image — grille

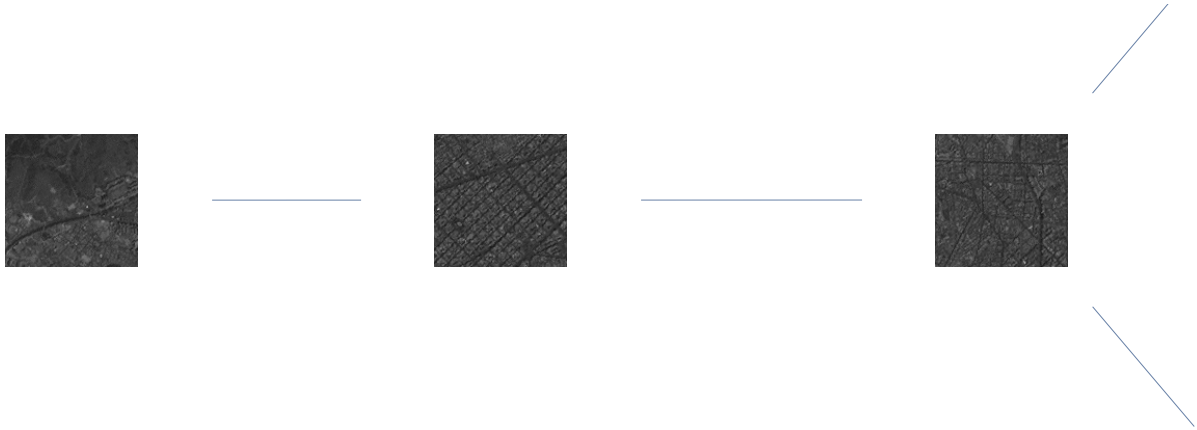


```
NearestNeighborGraph[Flatten@g, 1,
```
graphe du plus proche voisin — aplatis

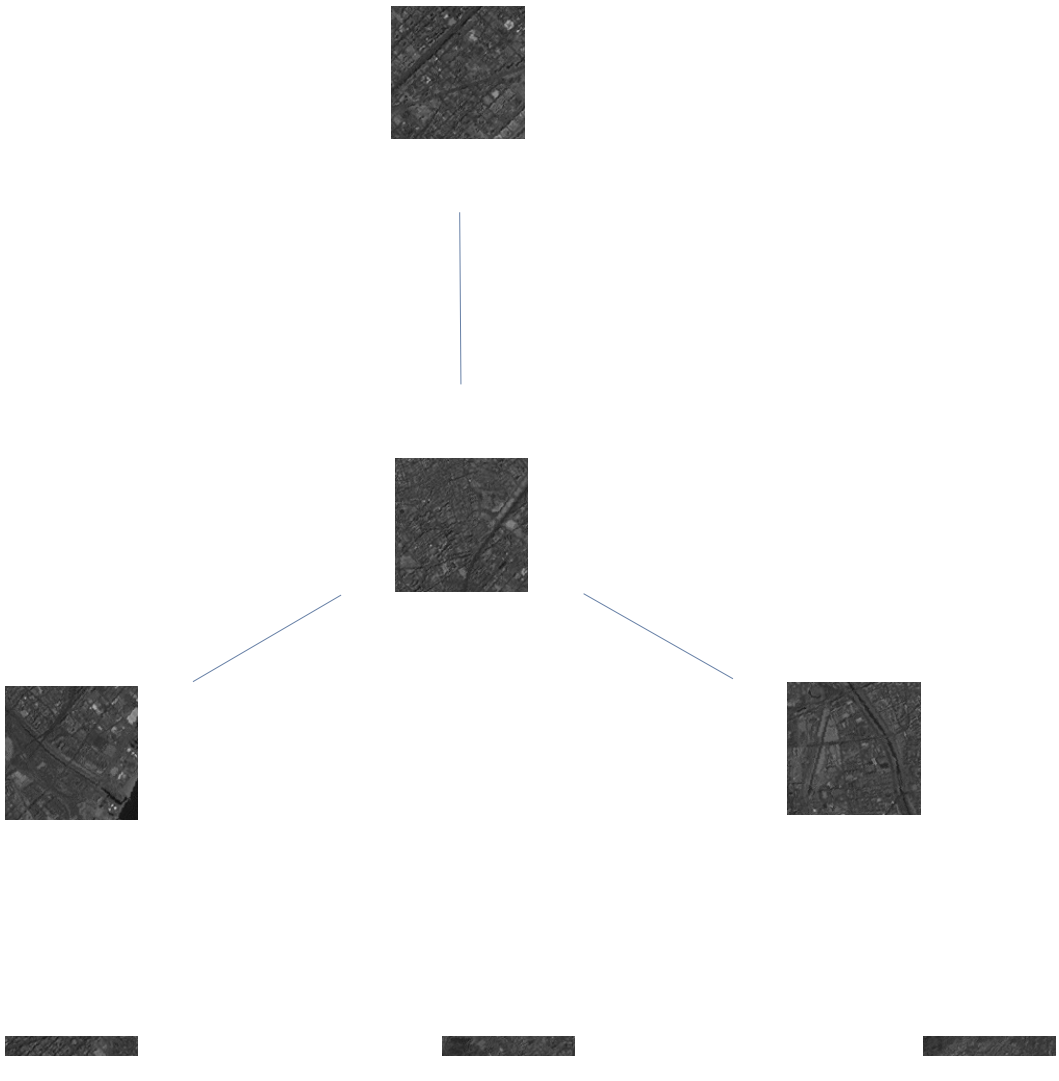
```
 VertexShapeFunction -> "Name", VertexSize → Large]
```
fonction de forme de sommet — taille de sommet — large

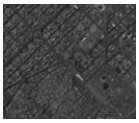
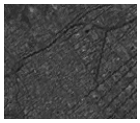
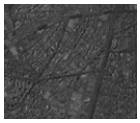
Out[ ]=

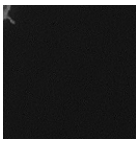


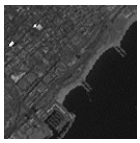


Out[ ]=







Une fonction similaire **MeshConnectivityGraph**[] construit un graphe entre des espace voisins, par exemple ceux issus d'un traitement par les polygones de Voronoï.

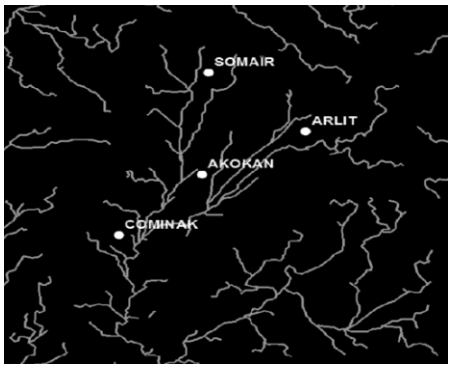
Extraire un réseau d'une carte ou d'une image

Les outils de traitement d'images permettent d'extraire un réseau qui est présent sur une image, que cette image soit une carte ou une photographie satellitaire. Généralement, il convient au préalable de binariser l'image, puis de se servir de la fonction **MorphologicalGraph**[], une fonction de morphologie mathématique. À partir d'une image récupérée dans Internet, nous pouvons extraire le graphe de son réseau hydrographique avec les deux lignes de code ci-dessous :

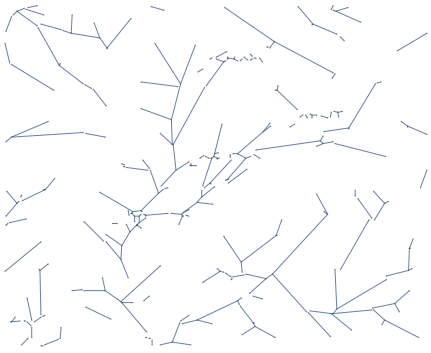
In[ ]:=
```
image = ColorNegate[ColorSeparate[ ]][[2]]
```
couleur en négatif | sépare couleur

```
g = MorphologicalGraph[image]
```
graphe morphologique

Out[ ]=



Out[ ]=



La figure affichée montre que quelques retouches sont à apporter. Ce serait facile à réaliser avec les outils de la morphologie mathématique.

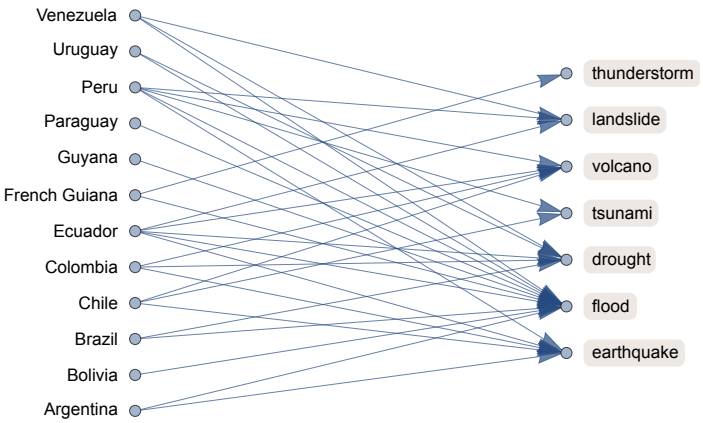
Des graphes bipartites

Il est tout aussi facile d'élaborer des graphes bipartites qui mettent en relation deux ensembles de données. Le graphe bi-partite ci-dessous associe à chaque pays d'Amérique du Sud les risques naturels encourus.

In[•]:=
```
risk = CountryData["SouthAmerica", "NaturalHazards"];
       (* données de pays *)
nom = CountryData["SouthAmerica", "Name"];
      (* données de pays *)
mat = Transpose[{nom, risk}];
      (* transpose *)
mat /. {country_, disaster_} :> (Rule[country, #] & /@ disaster)
                                 (* règle *)
graphData = Flatten[%, 2];
            (* aplatis *)
Graph[graphData, VertexLabels → "Name",
(* graphe *)      (* étiquettes de sommet *)
 GraphLayout → "BipartiteEmbedding", ImagePadding → 60]
 (* disposition de graphe *)          (* garnissage d'image *)
```

Out[•]=
```
{{Argentina → earthquake, Argentina → flood},
 {Bolivia → flood}, {Brazil → drought, Brazil → flood},
 {Chile → earthquake, Chile → tsunami, Chile → volcano},
 {Colombia → drought, Colombia → earthquake, Colombia → volcano},
 {Ecuador → drought, Ecuador → earthquake,
  Ecuador → flood, Ecuador → landslide, Ecuador → volcano}, {},
 {French Guiana → flood, French Guiana → thunderstorm}, {Guyana → flood},
 {Paraguay → flood}, {Peru → earthquake, Peru → flood, Peru → landslide,
  Peru → tsunami, Peru → volcano}, {}, {Uruguay → drought, Uruguay → flood},
 {Venezuela → drought, Venezuela → flood, Venezuela → landslide}}
```

Out[ ]=



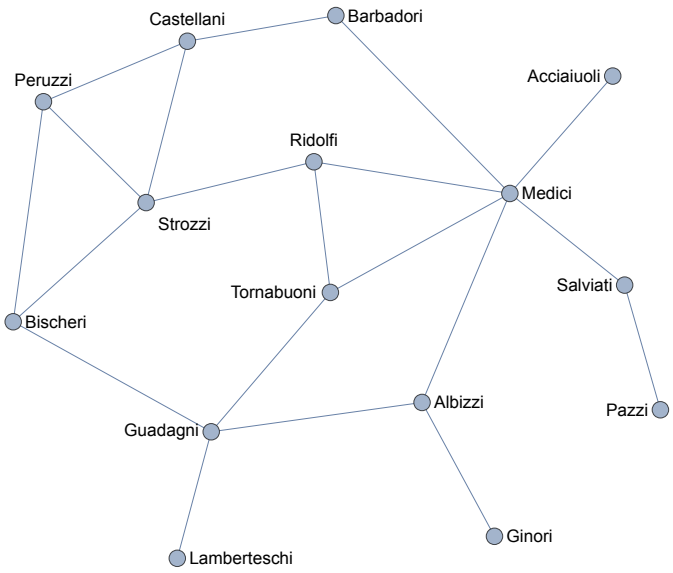
Attention, pour l'Europe ou d'autres ensembles d'États, certains pays ne sont pas représentés, et il faut en tenir compte pour que ce programme issu de l'aide fonctionne bien.

Il est aussi possible de construire des graphes de réseaux sociaux avec la fonction **SocialMediaData**[]. Mais, l'auteur de ce notebook est quelque peu réfractaire à s'inscrire sur ce type de réseau. Le lecteur peut cependant en importer avec la fonction **ExampleData**[].

Voici le graphe du réseau des grandes familles florentines à la Renaissance :

In[2]:=
```
societeFlorence = ExampleData[{"NetworkGraph", "FlorentineFamilies"}]
```
données d'exemples

Out[2]=



Après obtention d'un graphe, il est facile d'en appréhender la structure, puis suivant le type de graphe de conduire une analyse de flux. Les applications étant innombrables, nous présentons uniquement quelques applications, les plus fréquentes dans les études géographiques.

Analyse Structurelle de réseaux

Comme pour une série statistique, un réseau peut être étudié avec des indices globaux et/ou locaux. Dans le guide "*GraphPropertiesAndMeasurements*", le lecteur peut se rendre compte de la richesse des fonctions disponibles. Mais, pour des exercices plus réalistes, le graphe d'origine des 10 villes est transformé en graphe des liaisons aériennes, aller-retour, en retranchant et ou ajoutant des arêtes au graphe des 10 villes. ATTENTION LES CORRECTIONS NE SONT VALABLES QUE POUR CE RÉSEAU DES 10 VILLES. D'autres corrections seraient nécessaires pour construire un réseau plus vaste. Avec les fonctions **EdgeAdd**[] et **EdgeDelete**[], on obtient le graphe *liensAeriens*.

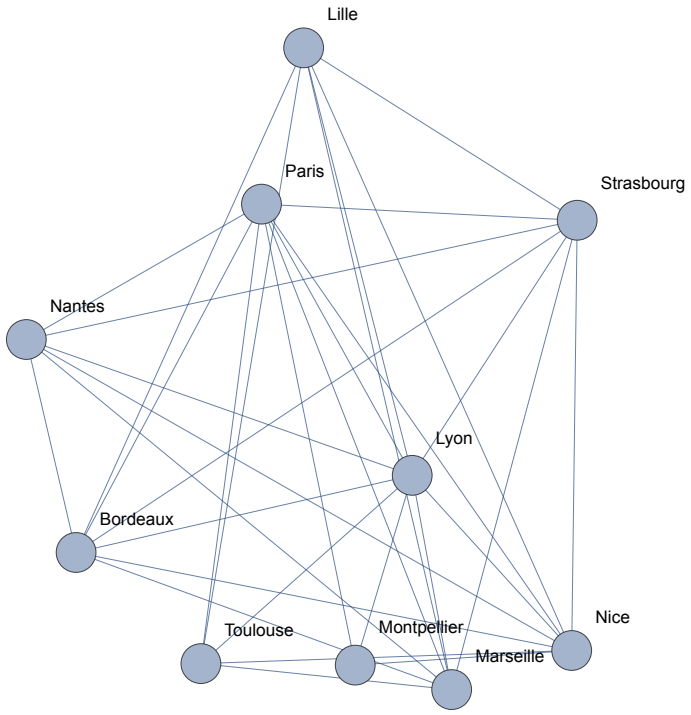
In[•]:=
```
gville1 = UndirectedGraph[gville];
```
graphe non dirigé

```
gville3 =
  EdgeAdd[gville1, {10 ↔ 6, 1 ↔ 7, 5 ↔ 3, 5 ↔ 7, 5 ↔ 10, 5 ↔ 9, 5 ↔ 6, 7 ↔ 2, 7 ↔ 10,
```
ajoute arête

```
    7 ↔ 9, 4 ↔ 2, 1 ↔ 10, 10 ↔ 4, 6 ↔ 4, 6 ↔ 2, 8 ↔ 1, 7 ↔ 8, 1 ↔ 9, 1 ↔ 6}];

liensAeriens = EdgeDelete[gville3, {1 ↔ 5, 2 ↔ 5, 9 ↔ 10, 8 ↔ 9, 6 ↔ 8, 6 ↔ 7}]
```
supprime arête

Out[•]=



Tester quelques propriétés élémentaires

Il est d'abord possible de s'interroger sur quel type de graphe le géographe analyse, avec des fonctions comme **DirectedGraphQ**[], **EulerianGraphQ** ou **HamiltonianGraphQ**[], puis réaliser les premières analyses avec des indicateurs globaux. Le petit programme ci-dessous applique quelques fonctions de tests au réseau des liaisons aériennes entre les 10 plus grandes villes françaises :

In[ ]:=
```
Print["Le réseau urbain est-il un graphe direct :  ",
 DirectedGraphQ[liensAeriens]]
Print["Le réseau urbain est-il un graphe connecté :",
 ConnectedGraphQ[liensAeriens]]
Print["Le réseau urbain est-il un graphe eulerien :",
 EulerianGraphQ[liensAeriens]]
Print["Le réseau urbain est-il un graphe acyclique :",
 AcyclicGraphQ[liensAeriens]]
Print["Le réseau urbain est-il un graphe bipartite :",
 BipartiteGraphQ[liensAeriens]]
```
imprime / graphe dirigé ? / imprime / graphe connexe ? / imprime / graphe eulérien ? / imprime / graphe acyclique ? / imprime / graphe bipartite ?

```
Le réseau urbain est-il un graphe direct :  False
Le réseau urbain est-il un graphe connecté :True
Le réseau urbain est-il un graphe eulerien :False
Le réseau urbain est-il un graphe acyclique :False
Le réseau urbain est-il un graphe bipartite :False
```

Puis, nous calculons quelques indicateurs globaux de ce même réseau :.

In[ ]:=
```
Print["Nombre d'arêtes du réseau :  ", EdgeCount[liensAeriens]]
Print["Nombre de hubs dans le réseau :  ", GraphHub[liensAeriens]]
Print["Distance moyenne du réseau :  ", MeanGraphDistance[liensAeriens] // N]
Print["Diametre du réseau :  ", GraphDiameter[liensAeriens] // N]
Print["Densité du réseau :  ", GraphDensity[liensAeriens] // N]
Print["Efficience en lien du réseau :  ",
 GraphLinkEfficiency[liensAeriens] // N]
Print["Coefficent global de clustering :  ",
 GlobalClusteringCoefficient[liensAeriens] // N]
Print["Coefficient de partionnement moyen :  ",
 MeanClusteringCoefficient[liensAeriens] // N]
Print["Connectivité des sommets :  ", VertexConnectivity[liensAeriens] // N]
Print["Connectivité d'arêtes :  ", EdgeConnectivity[liensAeriens] // N]
```

```
Nombre d'arêtes du réseau :  33
Nombre de hubs dans le réseau :  {3}
Distance moyenne du réseau :  1.2666667
Diametre du réseau :  2.
Densité du réseau :  0.73333333
Efficience en lien du réseau :  0.96161616
Coefficent global de clustering :  0.75757576
Coefficient de partionnement moyen :  0.7997619
Connectivité des sommets :  3.
Connectivité d'arêtes :  3.
```

Il est alors intéressant de calculer quelques indicateurs locaux de ce même réseau des liaisons aériennes et d'en proposer des illustrations. Parmi ces coefficients locaux, ceux de centralité ou inversement d'excentralité, d'homophilie pour les réseaux sociaux, de réciprocité sont les plus généralement étudiés. En voici quelques-uns, toujours pour le réseau aérien des 10 villes françaises. Attention ces calculs ne tiennent pas compte du poids de chaque aéroport.

In[•]:=
```
Print["Degré de sommet"]
```
imprime

```
g1 = VertexDegree[liensAeriens]
```
degré de sommet

```
HighlightGraph[liensAeriens, VertexList[liensAeriens],
```
mets graphe en surbrillance — liste de sommets

```
  VertexSize → Thread[VertexList[liensAeriens] → Rescale[g1]]]
```
taille de sommet — enfile — liste de sommets — rééchelonne

```
Print["Centralité de proximité"]
```
imprime

```
g1 = ClosenessCentrality[liensAeriens]
```
centralité de proximité

```
HighlightGraph[liensAeriens, VertexList[liensAeriens],
```
mets graphe en surbrillance — liste de sommets

```
  VertexSize → Thread[VertexList[liensAeriens] → Rescale[g1]]]
```
taille de sommet — enfile — liste de sommets — rééchelonne

```
Print["Centralité de degré"]
```
imprime

```
g1 = DegreeCentrality[liensAeriens]
```
centralité de degré

```
HighlightGraph[liensAeriens, VertexList[liensAeriens],
```
mets graphe en surbrillance — liste de sommets

```
  VertexSize → Thread[VertexList[liensAeriens] → Rescale[g1]]]
```
taille de sommet — enfile — liste de sommets — rééchelonne

```
g1 = EigenvectorCentrality[liensAeriens]
```
centralité de vecteurs propres

```
HighlightGraph[liensAeriens, VertexList[liensAeriens],
```
mets graphe en surbrillance — liste de sommets

```
  VertexSize → Thread[VertexList[liensAeriens] → Rescale[g1]]]
```
taille de sommet — enfile — liste de sommets — rééchelonne

```
Print["Coefficient de partitionnement local"]
```
imprime — coefficient

```
g1 = LocalClusteringCoefficient[liensAeriens] // N
```
coefficient de partitionnement local — valeur numérique

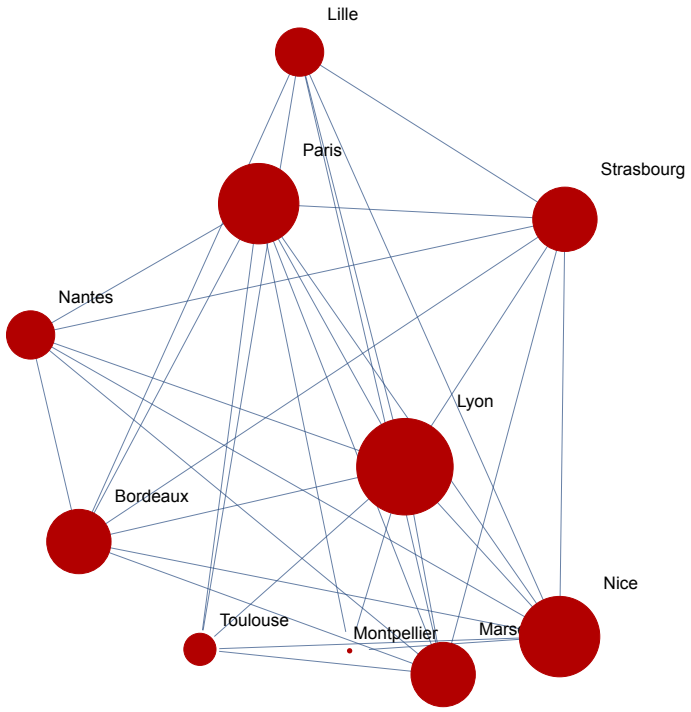
```
HighlightGraph[liensAeriens, VertexList[liensAeriens],
```
mets graphe en surbrillance — liste de sommets

```
  VertexSize → Thread[VertexList[liensAeriens] → Rescale[g1]]]
```
taille de sommet — enfile — liste de sommets — rééchelonne

```
Degré de sommet
```

Out[•]=
```
{8, 6, 9, 7, 6, 7, 5, 3, 7, 8}
```

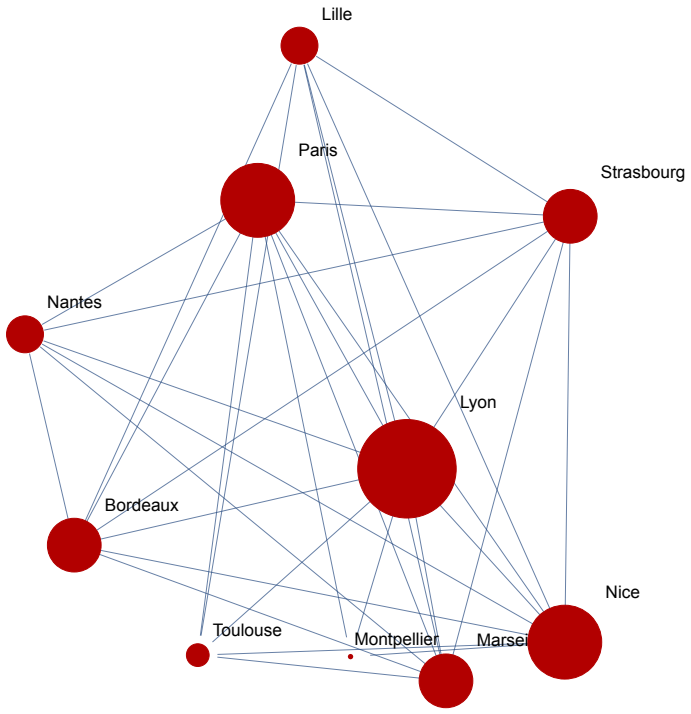
Out[ ]=



Centralité de proximité

Out[ ]= {0.9, 0.75, 1., 0.81818182, 0.75, 0.81818182, 0.69230769, 0.6, 0.81818182, 0.9}

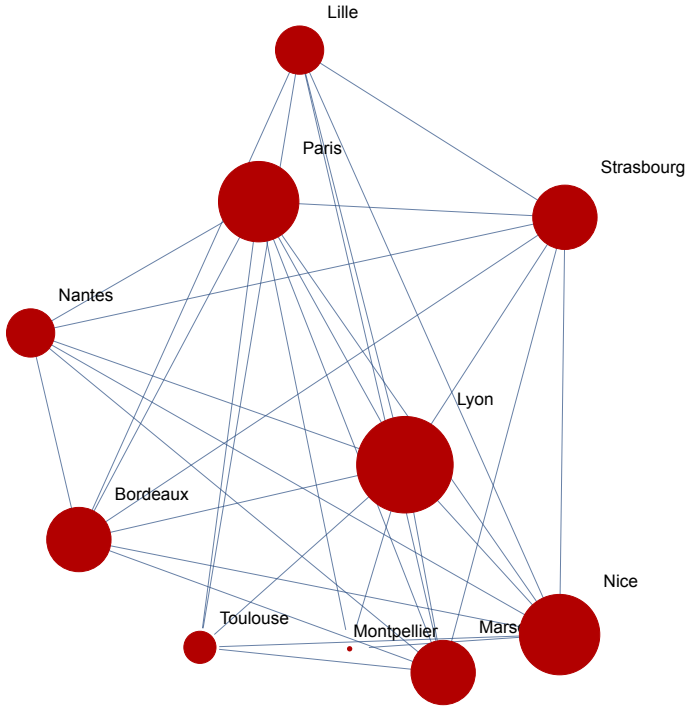
Out[ ]=



Centralité de degré

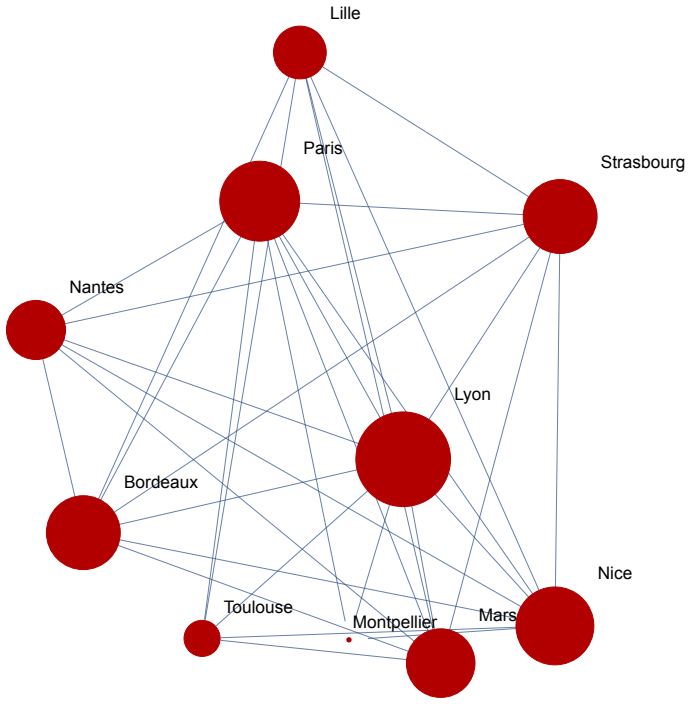
Out[ ]= {8, 6, 9, 7, 6, 7, 5, 3, 7, 8}

Out[ ]=



Out[ ]= {0.11445289, 0.097955544, 0.12617967, 0.10968232, 0.092937141, 0.10968232, 0.07969941, 0.051045501, 0.10549711, 0.11286808}

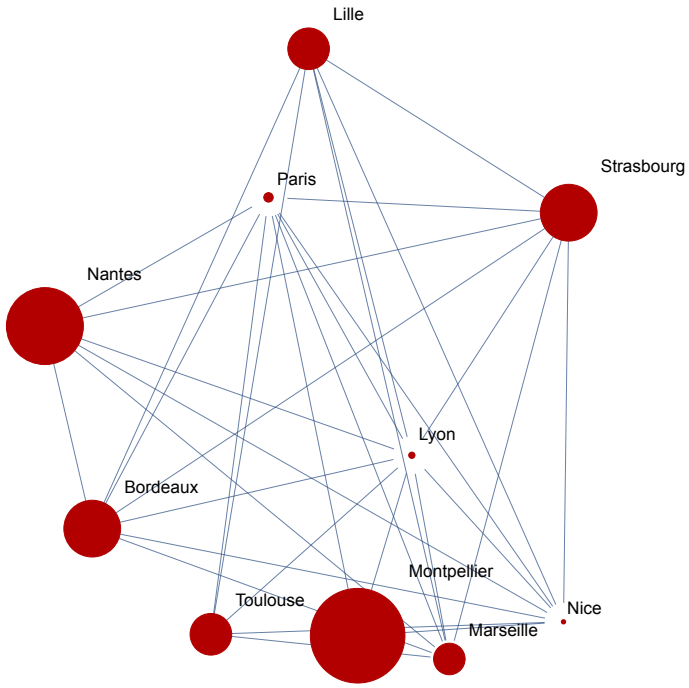
Out[ ]=



Coefficient de partitionnement local

Out[ ]= {0.67857143, 0.93333333, 0.66666667, 0.85714286, 0.8, 0.85714286, 0.8, 1., 0.76190476, 0.64285714}

Out[•]=



Outre ces indicateurs globaux et locaux, il est nécessaire de s'interroger sur la connectivité, connectivité qui peut concerner des composantes du graphe, mais aussi les noeuds ou les arêtes. Mathematica offre plus d'une vingtaine de fonctions pour analyser la connectivité.

Composantes et connectivité du graphe

Il existe de nombreux algorithmes pour définir les composantes d'un graphe. Et, il en va de même pour les analyses de connectivité. Procédons d'abord aux analyses de composantes, de simples partitions, des cliques ou des K-composantes.

In[•]:=
```
Print["Partitions du réseau"]
```
imprime

```
gg = FindGraphPartition[liensAeriens]
```
trouve partition de graphe

```
HighlightGraph[liensAeriens, Map[Subgraph[liensAeriens, #] &, gg]]
```
mets graphe en surbrillance — app··sous-graphe

```
Print["Communautés du réseau"]
```
imprime

```
gg = FindGraphCommunities[liensAeriens]
```
trouve communautés de graphe

```
HighlightGraph[liensAeriens, Map[Subgraph[liensAeriens, #] &, gg]]
```
mets graphe en surbrillance — app··sous-graphe

```
CommunityGraphPlot[liensAeriens, FindGraphCommunities[liensAeriens]]
```
tracé de graphe de communauté — trouve communautés de graphe

```
Print["Clique du réseaux"]
```
imprime

```
gg = FindClique[liensAeriens]
```
trouve clique

```
HighlightGraph[liensAeriens, Subgraph[liensAeriens, gg]]
```
mets graphe en surbrillance — sous-graphe

```
Print["Kplex d'ordre 2"]
```
imprime

```
gg = FindKPlex[liensAeriens, 2]
```
trouve k-plex

```
HighlightGraph[liensAeriens, Subgraph[liensAeriens, gg]]
```
mets graphe en surbrillance — sous-graphe

```
Print["K-Core Composantes du réseau"]
```
imprime

```
gg = KCoreComponents[liensAeriens, 3]
```
composantes k-cœur

```
HighlightGraph[liensAeriens, gg]
```
mets graphe en surbrillance

```
Print["Composantes d'arêtes du réseau"]
```
imprime

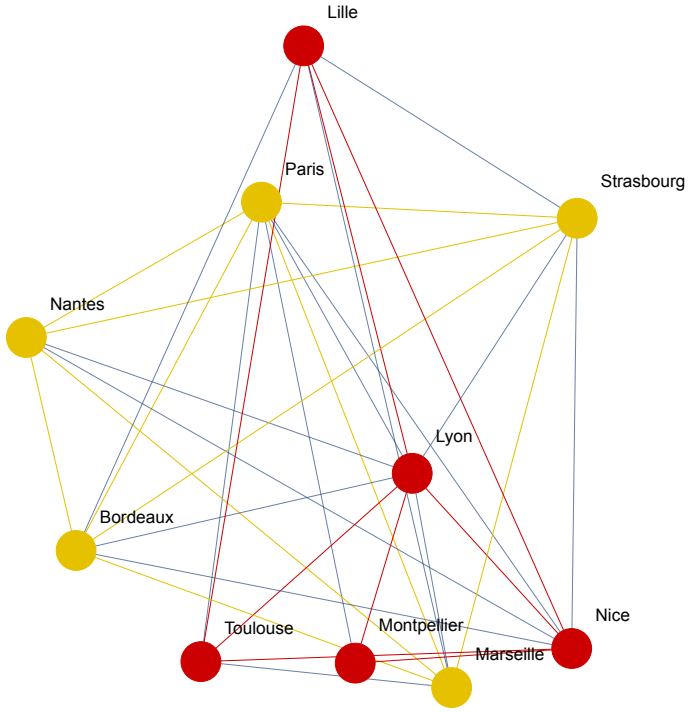
```
gg = VertexComponent[liensAeriens, {1}]
```
composantes de sommet

```
HighlightGraph[liensAeriens, gg]
```
mets graphe en surbrillance

```
Partitions du réseau
```

Out[•]=
```
{{3, 5, 9, 8, 7}, {1, 2, 4, 10, 6}}
```

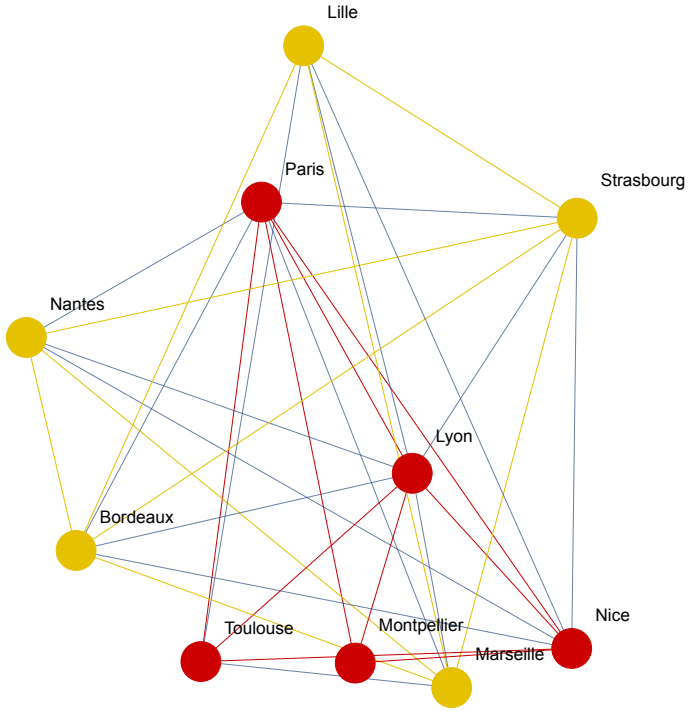
Out[ ]=



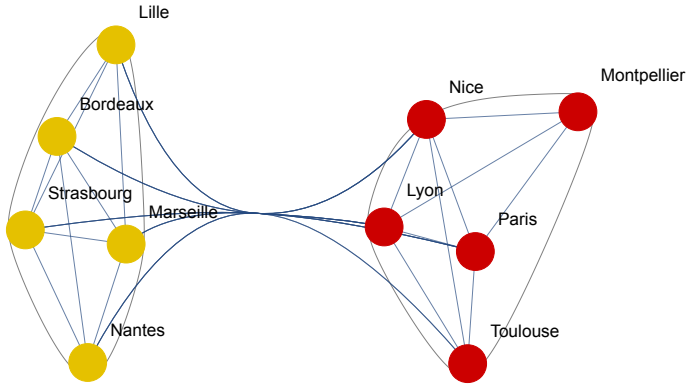
Communautés du réseau

Out[ ]= {{1, 3, 9, 8, 7}, {2, 4, 5, 10, 6}}

Out[ ]=



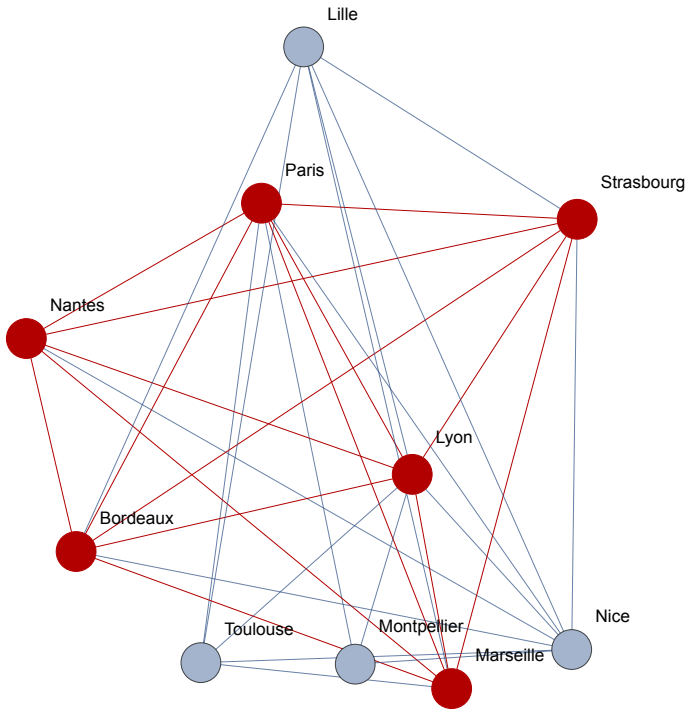
Out[ ]=



Clique du réseaux

Out[ ]= {{1, 2, 3, 4, 10, 6}}

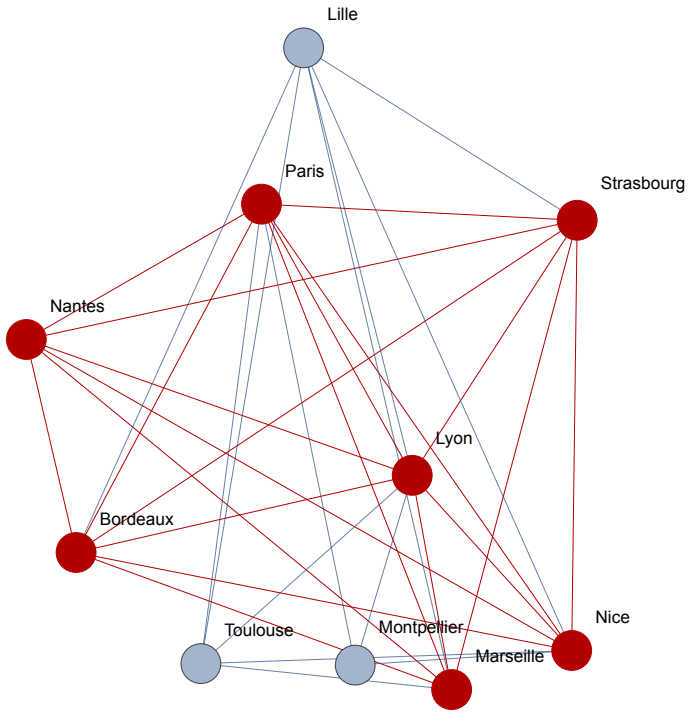
Out[ ]=



Kplex d'ordre 2

Out[ ]= {{1, 2, 3, 4, 10, 6, 7}}

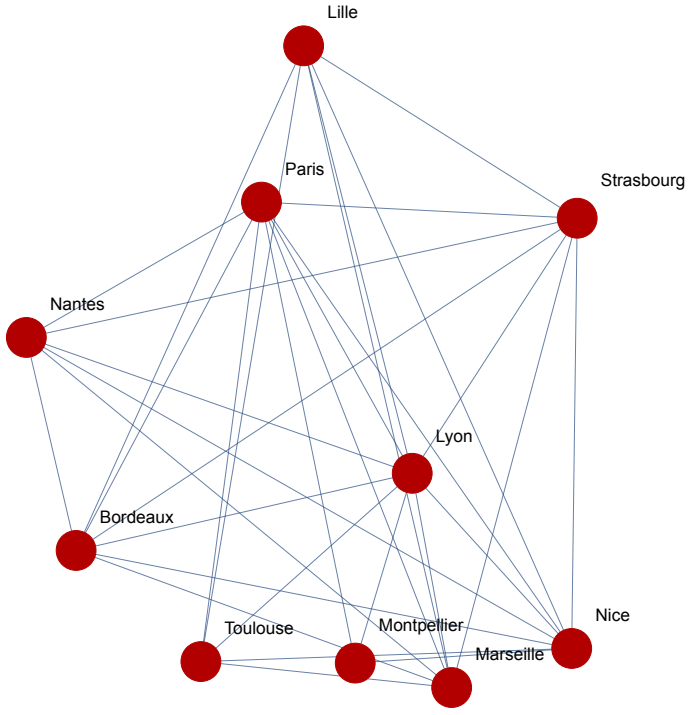
Out[ ]=



K-Core Composantes du réseau

Out[ ]= {{1, 2, 3, 4, 5, 10, 9, 8, 6, 7}}

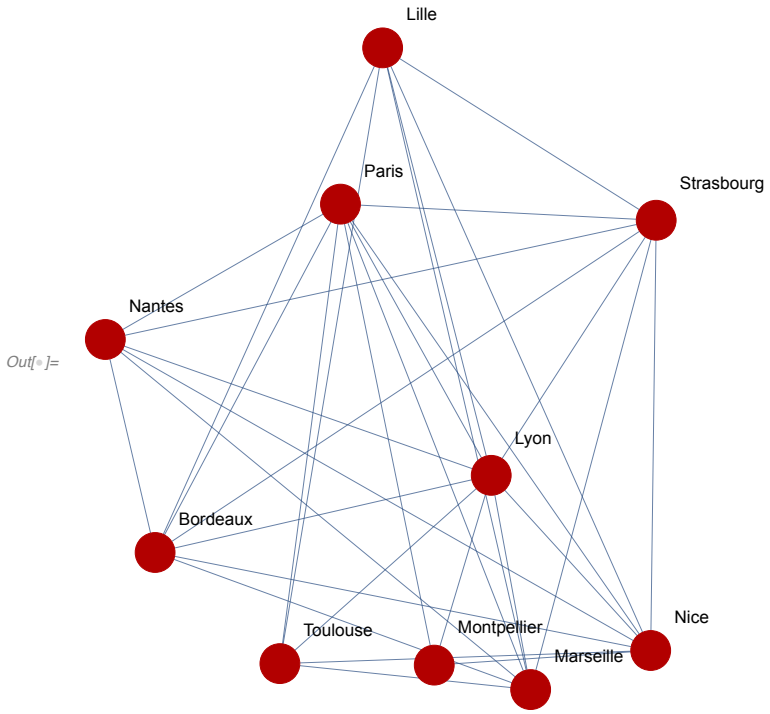
Out[ ]=



Composantes d'arêtes du réseau

Out[ ]= {1, 2, 3, 4, 10, 9, 8, 6, 7, 5}

Out[ ]=

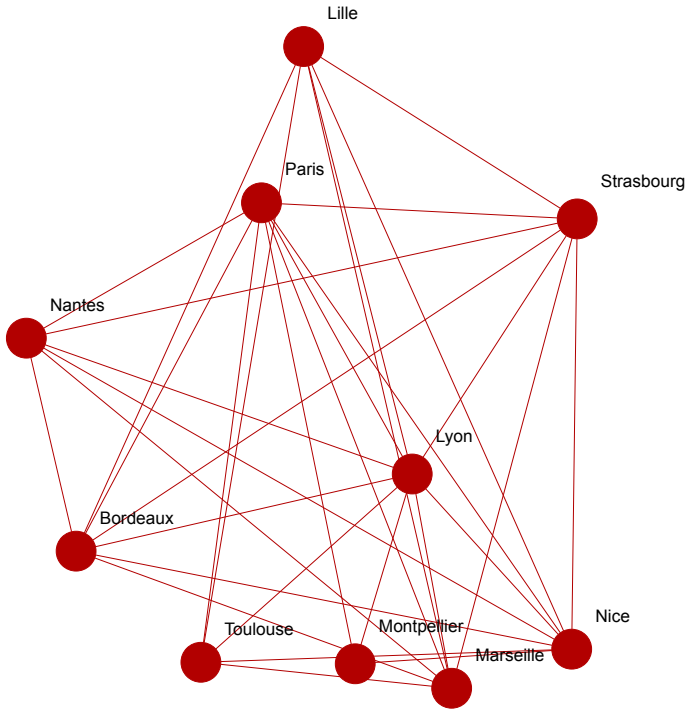


Puis examinons quelques indicateurs de connectivité

In[ ]:=
```
Print["Composantes connectées du réseau"]
```
imprime
```
gg = ConnectedGraphComponents[liensAeriens];
```
composantes de graphe connexe
```
HighlightGraph[liensAeriens, gg]
```
mets graphe en surbrillance
```
Print["Connectivité des noeuds du réseau"]
```
imprime
```
gg = VertexConnectivity[liensAeriens]
```
connectivité de sommet
```
Print["Connectivité des arêtes du réseau"]
```
imprime
```
gg = EdgeConnectivity[liensAeriens]
```
connectivité d'arête

```
Composantes connectées du réseau
```

Out[ ]=




```
Connectivité des noeuds du réseau
```

Out[ ]= 3

```
Connectivité des arêtes du réseau
```

Out[ ]= 3

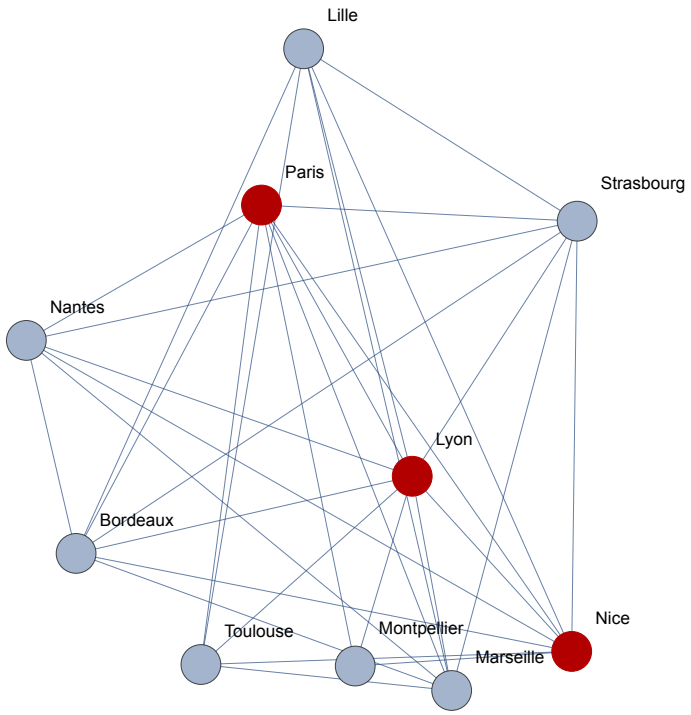
Il est même possible de reconnaître les noeuds ou les arêtes à couper pour désorganiser totalement ce réseau. Ce qui est très utile dans une recherche sur les risques.

In[•]:=
```
Print["Coupure des noeuds du réseau"]
```
imprime
```
gg = FindVertexCut[liensAeriens]
```
trouve coupe de sommet
```
HighlightGraph[liensAeriens, gg]
```
mets graphe en surbrillance
```
Print["Coupure des arêtes du réseau"]
```
imprime
```
gg = FindEdgeCut[liensAeriens]
```
trouve coupe d'arête
```
HighlightGraph[liensAeriens, gg]
```
mets graphe en surbrillance
```
Print["Coupures minimales du réseau"]
```
imprime
```
gg = FindMinimumCut[liensAeriens]
```
trouve coupe minimale
```
HighlightGraph[liensAeriens, Map[Subgraph[liensAeriens, #] &, Last[gg]]]
```
mets graphe en surbrillance | app·· | sous-graphe | dernier
```
Print["Coupures maximales du réseau"]
```
imprime
```
gg = FindMaximumCut[liensAeriens]
```
trouve coupe maximale
```
HighlightGraph[liensAeriens, Map[Subgraph[liensAeriens, #] &, Last[gg]]]
```
mets graphe en surbrillance | app·· | sous-graphe | dernier

```
Coupure des noeuds du réseau
```

Out[•]= {1, 3, 7}

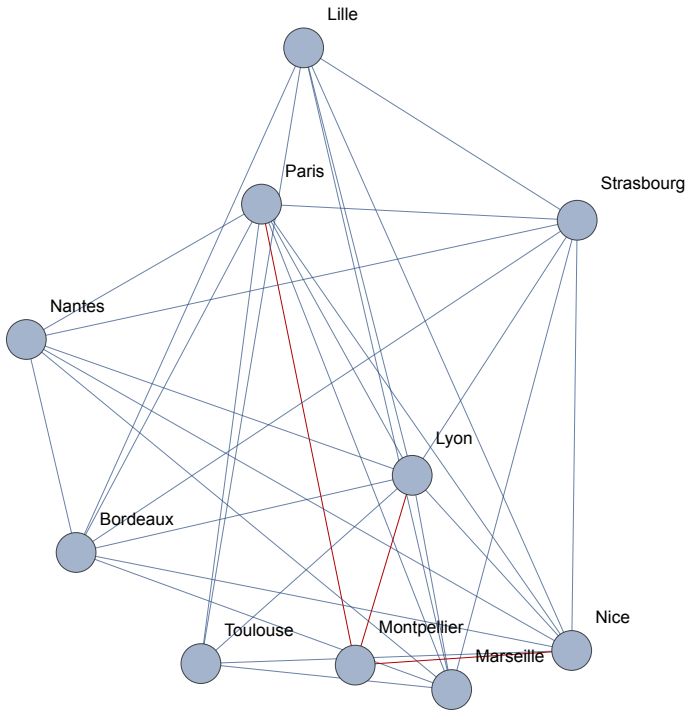
Out[•]=




```
Coupure des arêtes du réseau
```

Out[•]= {1 ⟷ 8, 3 ⟷ 8, 8 ⟷ 7}

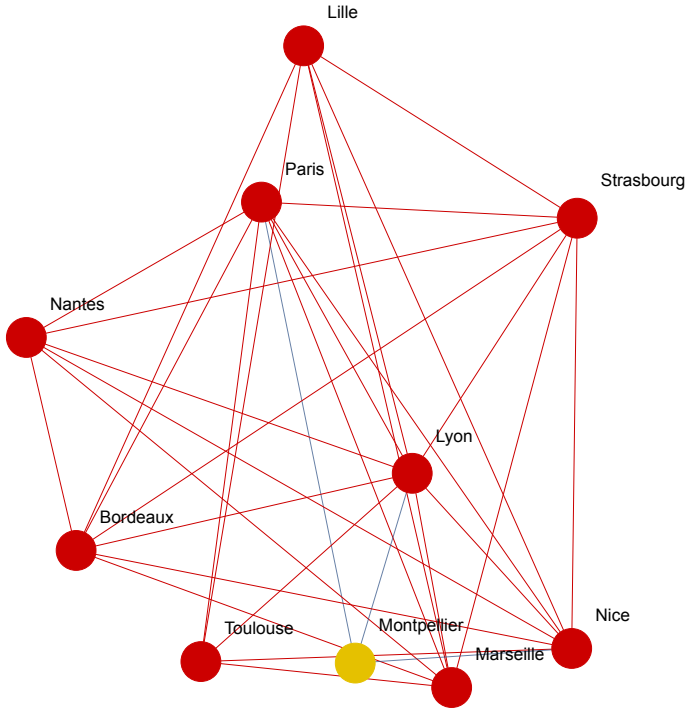
Out[ ]=



Coupures minimales du réseau

Out[ ]= {3, {{1, 2, 3, 4, 5, 10, 9, 6, 7}, {8}}}

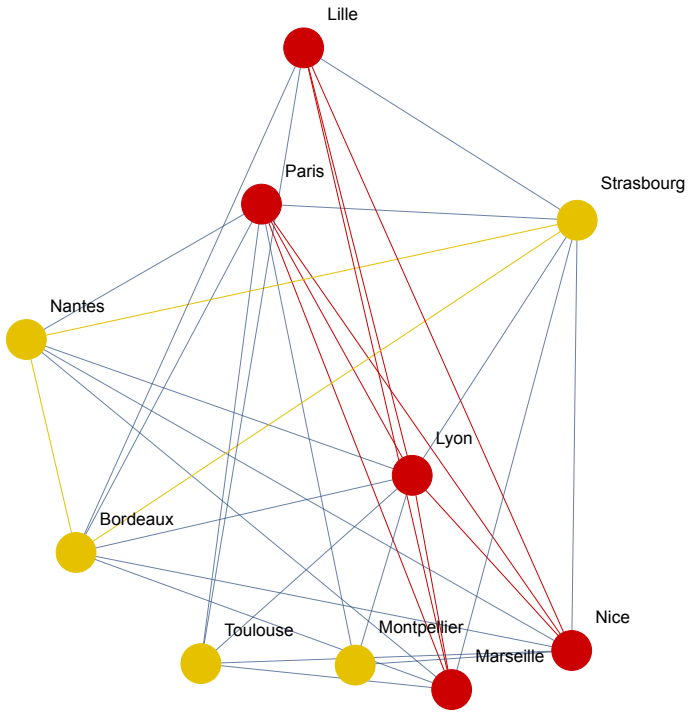
Out[ ]=



Coupures maximales du réseau

Out[ ]= {22, {{1, 3, 5, 6, 7}, {2, 4, 10, 9, 8}}}

Out[ ]=



Bien d'autres fonctions sont disponibles. Et le géographe peut construire ses propres fonctions, par exemple en partant des matrices d'incidences ou adjacentes, qui s'obtiennent facilement avec les fonctions **AdjacencyMatrix[]** et **IncidenceMatrix[],** et même **WeightedAdjacencyMatrix**[] pour des données pondérées par l'intensité des flux ou le poids des noeuds. Ci-dessous la simple matrice adjacente du réseau des relations aériennes :

In[ ]:=
```
AdjacencyMatrix[liensAeriens] // MatrixForm
```
matrice d'adjacence — forme matricielle

Out[ ]//MatrixForm=

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

Enfin, toujours dans une analyse structurelle, le géographe s'intéresse aux chemins formés par une suite d'arêtes.

Chemins et cycles d'un réseau

Pour naviguer d'un point à un autre d'un réseau, on cherche le plus court chemin, tandis que pour traverser un réseau, on s'intéresse plutôt aux cycles. Les fonctions **FindShortestPathe**[], puis **FindHamiltonianPath**[], qui visite exactement une fois seulement chaque noeud, trouve les chemins les plus courts entre les noeuds d'un réseau.

In[•]:=
```
gg = FindShortestPath[liensAeriens, 8, 5]
```
trouve le plus court chemin

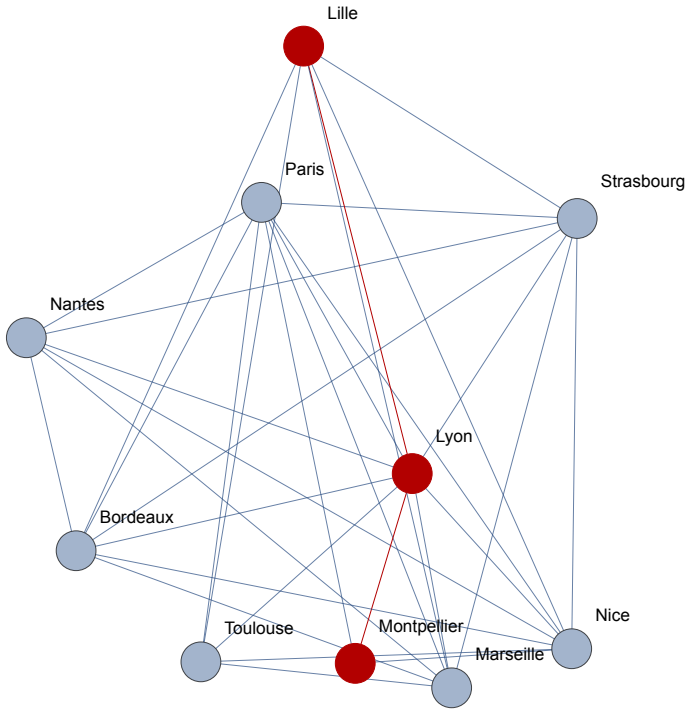
```
HighlightGraph[liensAeriens, PathGraph[gg]]
```
mets graphe en surbrillance — graphe chemin

```
gg = FindHamiltonianPath[liensAeriens]
```
trouve chemin hamiltonien

```
HighlightGraph[liensAeriens, PathGraph[gg]]
```
mets graphe en surbrillance — graphe chemin

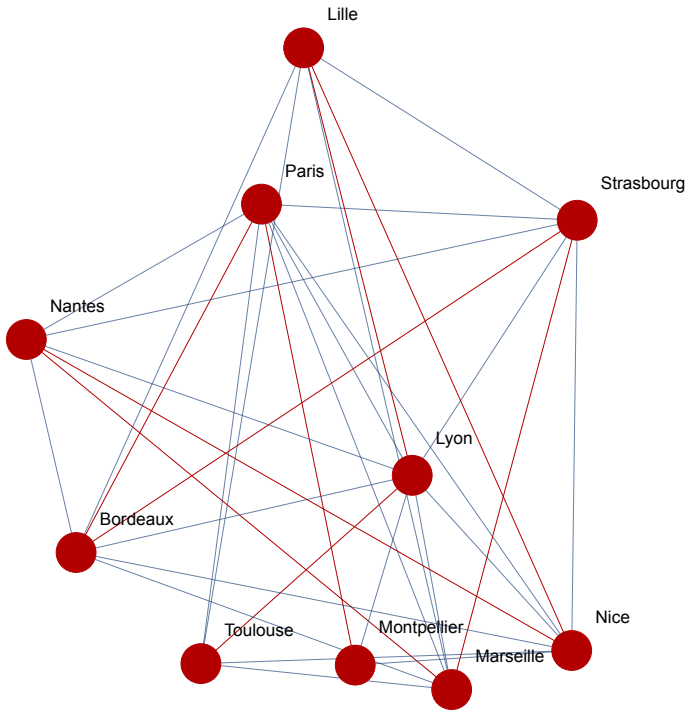
Out[•]= {8, 3, 5}

Out[•]=



Out[•]= {9, 3, 5, 7, 2, 6, 4, 10, 1, 8}

Out[ ]=



Ces fonctions sont généralisables. Voici l'exemple théorique d'un touriste qui désire trouver le plus court chemin pour visiter tous les pays d'Europe en ne passant qu'une seule fois dans chacun d'eux. Ce petit programme est directement emprunté de l'aide consacrée à la fonction **FindHamiltonianPath[]**.

In[ ]:=
```
europe = CountryData["Europe"];
pos = GeoPosition[CountryData[#, "CenterCoordinates"]] & /@ europe;
adj = Table[QuantityMagnitude@GeoDistance[i, j], {i, pos}, {j, pos}] /.
    Quantity[0., _] → Infinity;
g = WeightedAdjacencyGraph[europe, adj];
FindHamiltonianPath[g, Greece COUNTRY, Germany COUNTRY] // Shallow

GeoGraphics[{Thick, Red, GeoPath[%]}]
```

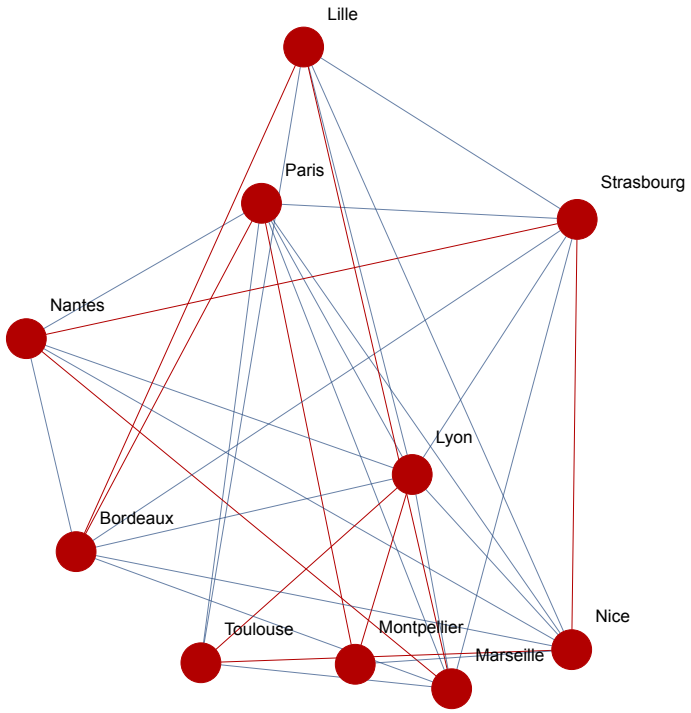
Out[ ]//Shallow=
{Greece, Cyprus, Bulgaria, North Macedonia, Kosovo, Serbia, Bosnia and Herzegovina, Montenegro, Albania, Malta, <<41>>}

Out[ ]=

Des fonctions similaires servent pour analyser les tours ou les cycles.

```
In[•]:= gg = FindHamiltonianCycle[liensAeriens]
        trouve cycle hamiltonien
        HighlightGraph[liensAeriens, PathGraph[First[gg]]]
        mets graphe en surbrillance   graphe chemin  premier
Out[•]= {{1 ↔ 10, 10 ↔ 5, 5 ↔ 6, 6 ↔ 2, 2 ↔ 4, 4 ↔ 7, 7 ↔ 9, 9 ↔ 3, 3 ↔ 8, 8 ↔ 1}}
```

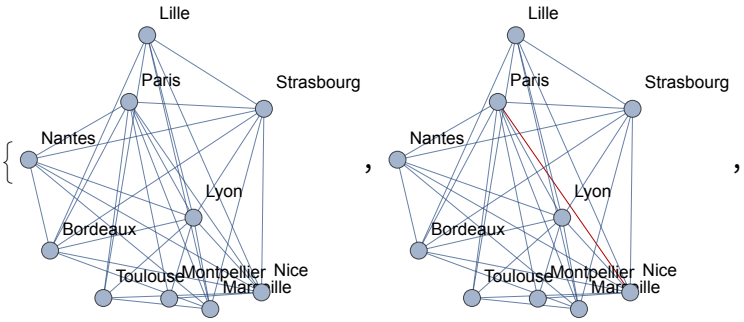
Out[•]=

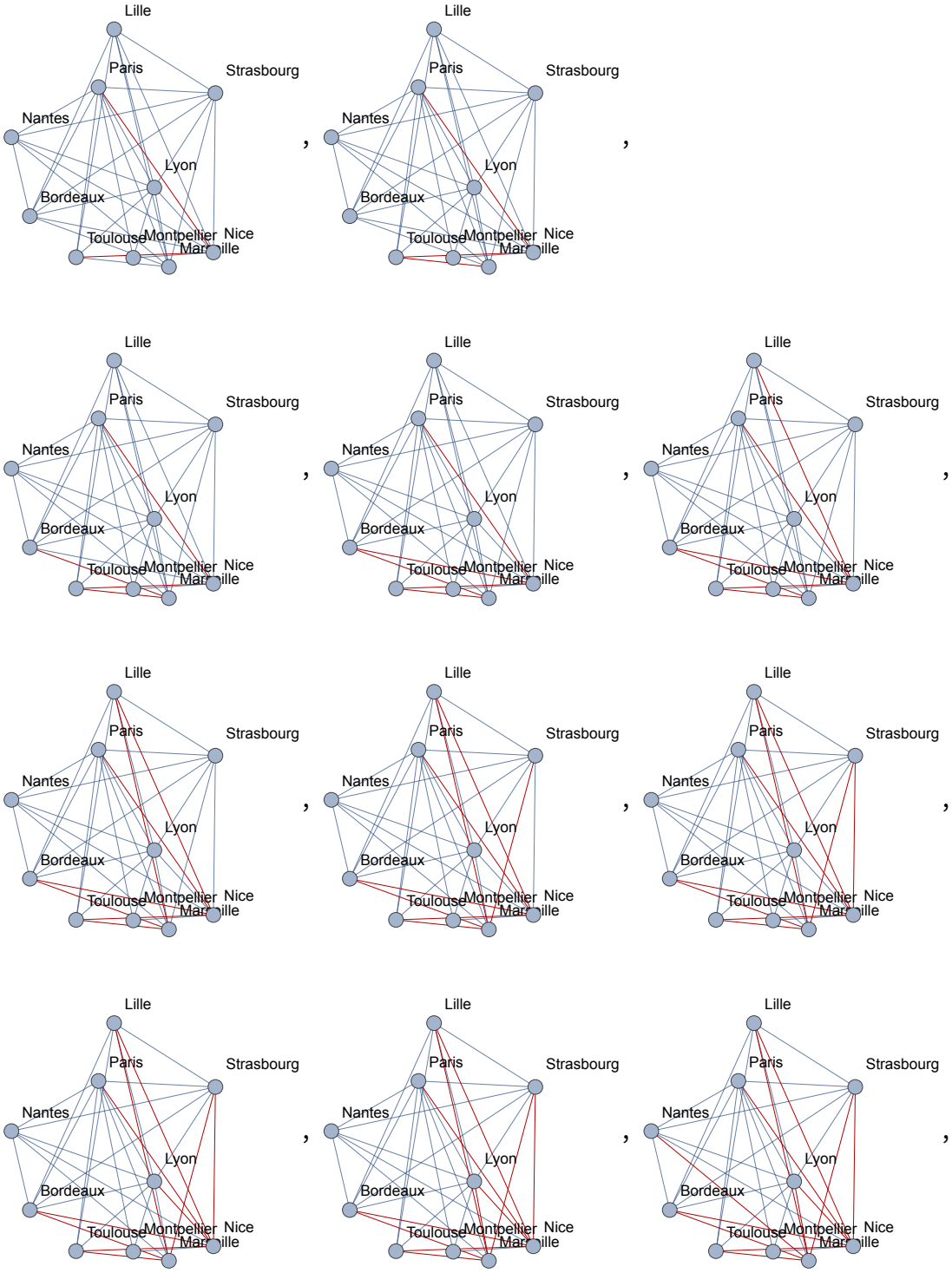


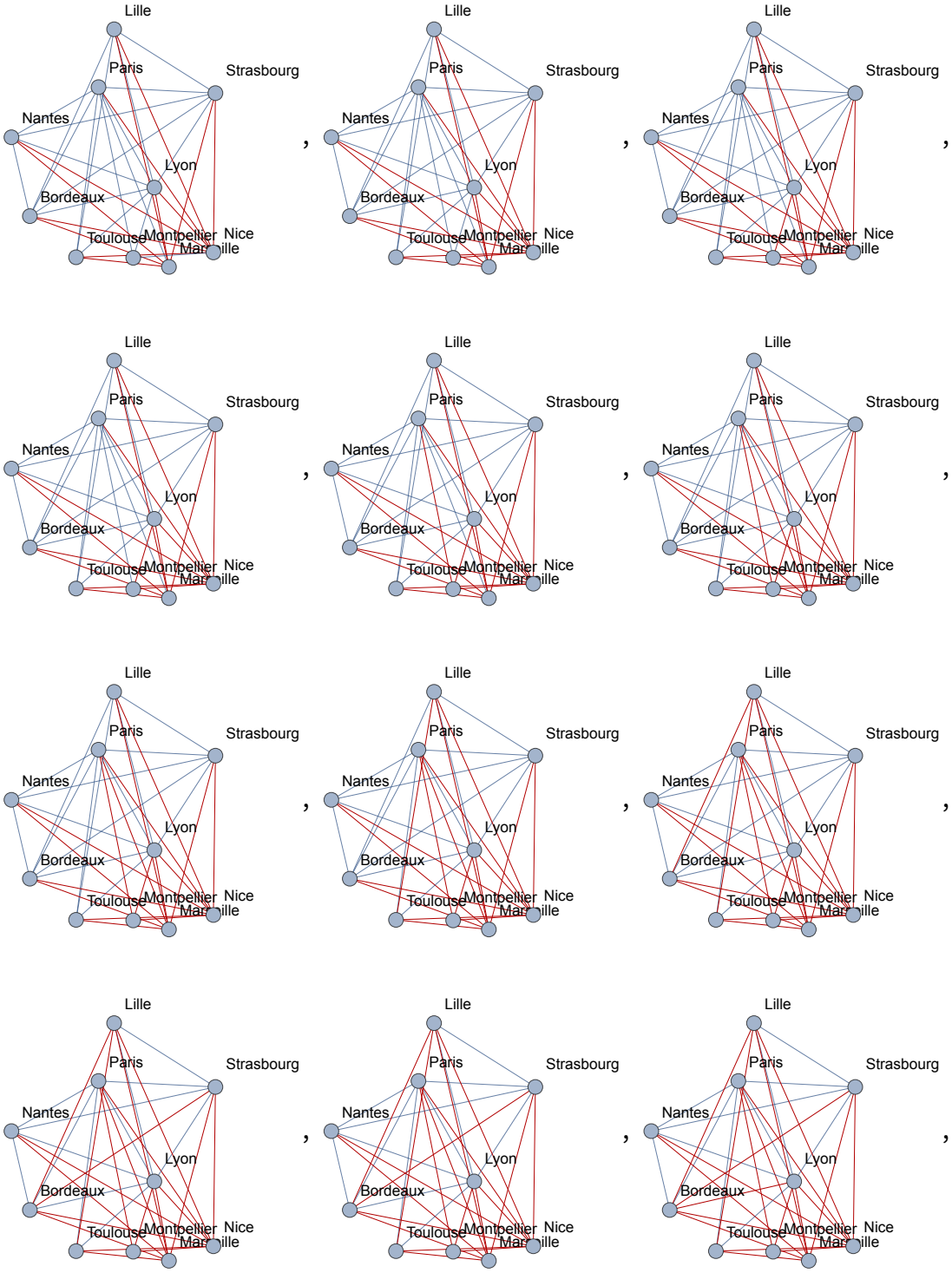
Et, le tour du postier emprunte chaque arête au moins une fois. Le programme ci-dessous illustre les déplacements successifs de ce processus.

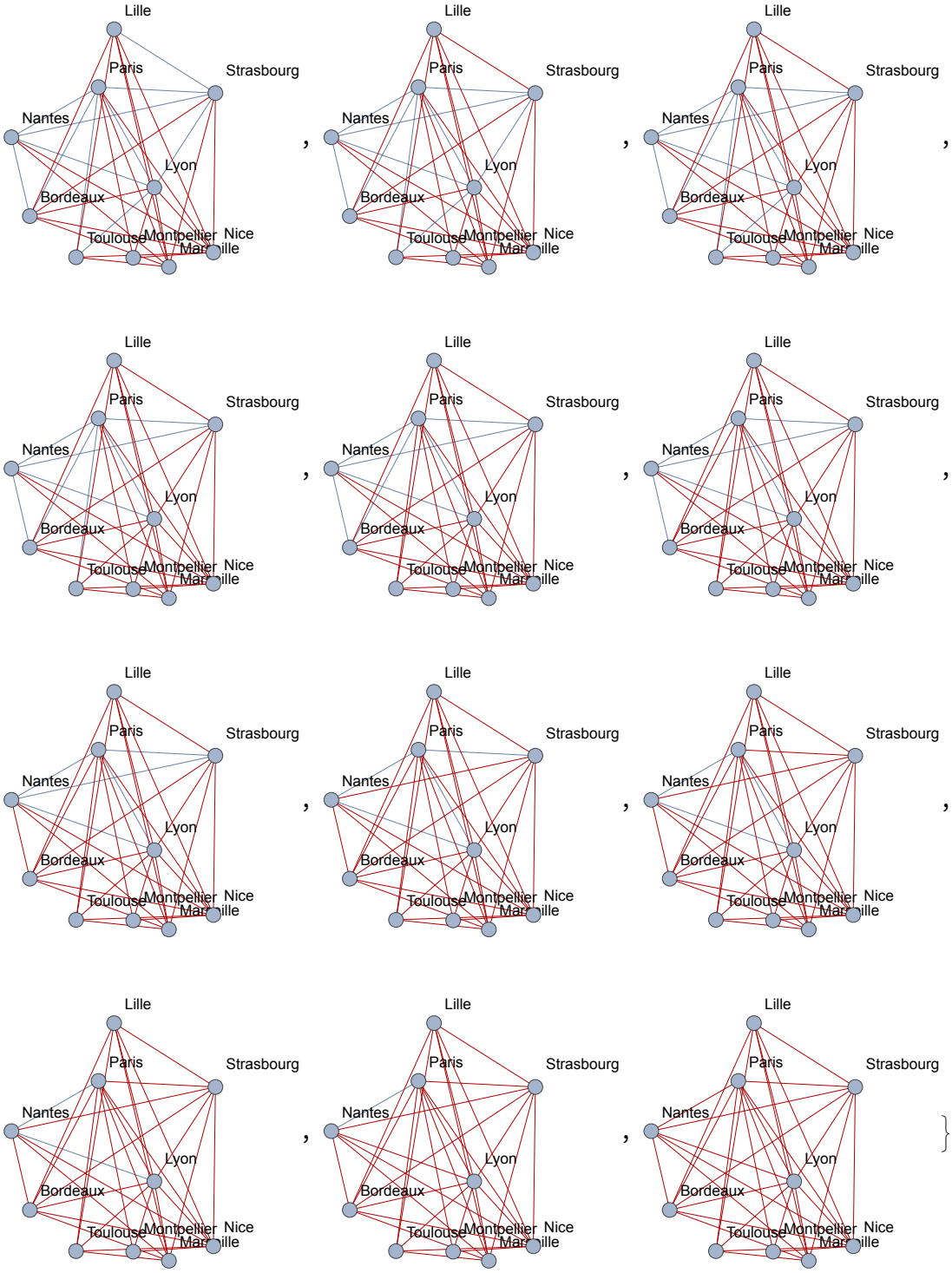
```
In[•]:= gg = FindPostmanTour[liensAeriens] // First
        trouve circuit de facteur                premier
        Table[HighlightGraph[liensAeriens, gg[[1 ;; i]]], {i, 0, Length[gg]}]
        table   mets graphe en surbrillance                    longueur
Out[•]= {1 ↔ 7, 7 ↔ 9, 9 ↔ 6, 6 ↔ 10, 10 ↔ 7, 7 ↔ 5, 5 ↔ 6, 6 ↔ 4,
         4 ↔ 7, 7 ↔ 3, 3 ↔ 6, 6 ↔ 2, 2 ↔ 7, 7 ↔ 8, 8 ↔ 3, 3 ↔ 8, 8 ↔ 1, 1 ↔ 6,
         6 ↔ 9, 9 ↔ 5, 5 ↔ 10, 10 ↔ 4, 4 ↔ 10, 10 ↔ 3, 3 ↔ 5, 5 ↔ 4, 4 ↔ 3,
         3 ↔ 9, 9 ↔ 1, 1 ↔ 10, 10 ↔ 2, 2 ↔ 4, 4 ↔ 1, 1 ↔ 3, 3 ↔ 2, 2 ↔ 1}
```

Out[•]=




Lille
Paris
Strasbourg
Nantes
Lyon
Bordeaux
Toulouse
Montpellier
Nice
Marseille


Lille
Paris
Strasbourg
Nantes
Lyon
Bordeaux
Toulouse
Montpellier
Nice
Marseille



Le lecteur trouvera bien d'autres fonctions et des exemples géographiques dans les aides de chacune de ces fonctions.

Modèles de graphes de réseaux

En statistique, le géographe recherche la loi de probabilité qui correspond à ses données. La même logique reste valable pour les réseaux. À quel type de réseau théorique correspond le réseau étudié. Deux approches sont généralement proposées. Soit se servir d'indicateurs notamment ceux

relatifs aux noeuds et aux arêtes. Soit, comparer le graphe-données à divers graphes théoriques. Le logiciel Wolfram Mathematica en propose des centaines. Ces deux approches sont souvent complémentaires. La première étant utilisée plutôt pour modéliser la croissance d'un réseau dans le temps, par exemple la croissance d'Internet. La seconde approche est fondée sur le concept d'isomorphisme. Deux graphes isomorphes ont le même nombre de sommets et sont connectés de la même façon. Les instructions ci-dessous testent si le graphe des liaisons aériennes est isomorphe à deux graphes aléatoires avec la fonction **IsomorphicGraphQ[]**. Remarquez que nous donnons le même nombre de sommets et d'arêtes aux graphe aléatoire qu'au graphe du réseau aérien. Comme le réseau aérien n'est pas assimilable ni à un modèle purement aléatoire, ni un modèle de Bernoulli, le programme donne la réponse *False*. Puis, la fonction **FingGraphIsomorphism[]** donne un ensemble vide {}.

In[•]:=
```
h = RandomGraph[{10, 33}]
```
graphe aléatoire

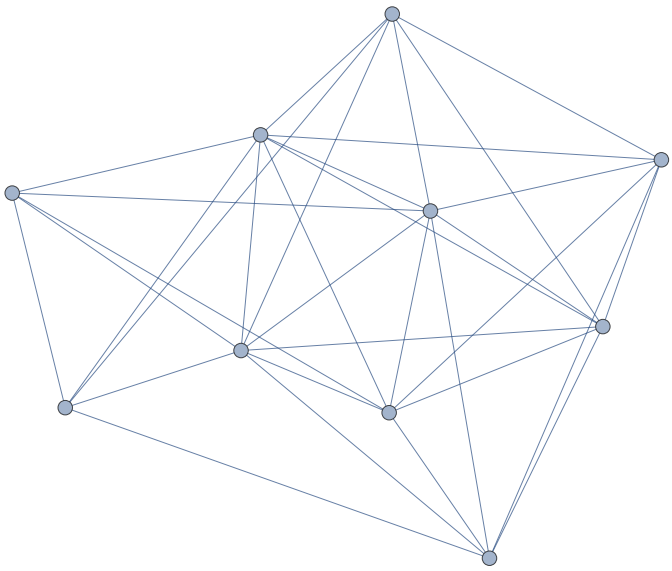
```
h1 = RandomGraph[BernoulliGraphDistribution[10, 0.9]]
```
graphe aléatoire | loi de graphe de Bernoulli

```
IsomorphicGraphQ[liensAeriens, h]
```
graphe isomorphe ?

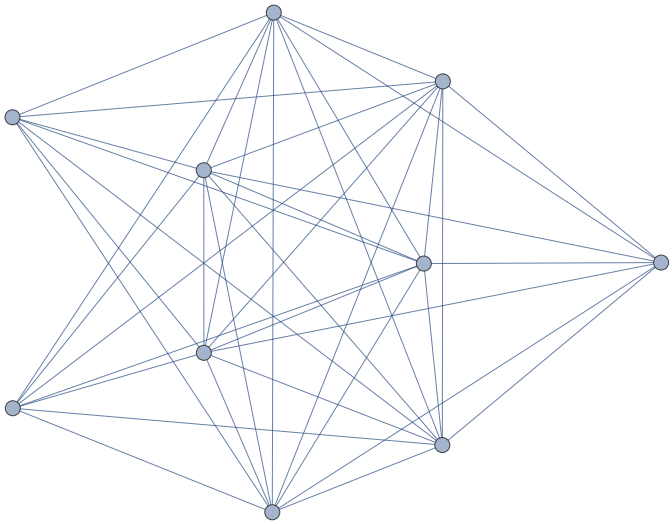
```
IsomorphicGraphQ[liensAeriens, h1]
```
graphe isomorphe ?

```
FindGraphIsomorphism[h, liensAeriens]
```
trouve isomorphisme de graphes

Out[•]=



Out[ ]=



Out[ ]= False

Out[ ]= False

Out[ ]= { }

Conclusion

Bien qu'élémentaire, la présentation de ces fonctions montre la richesse des outils utilisables pour étudier la structure des réseaux. Il convient d'ajouter qu'outre les options, les objets graphiques obtenus se combinent facilement avec les autres fonctions intégrées dans le logiciel, par exemple avec la cartographie. Enfin, les Resources Function proposent quelques fonctions pour l'étude des hypergraphes qui généralisent le concept de graphe.