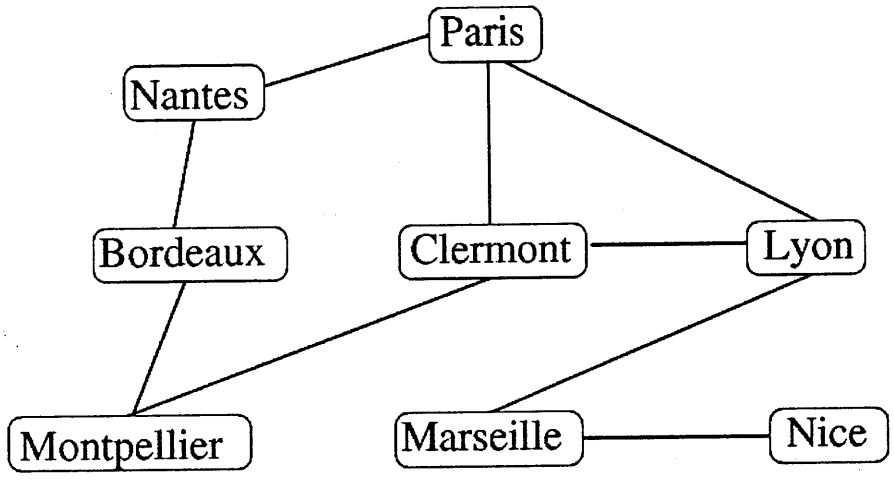
Modéliser une carte routière?

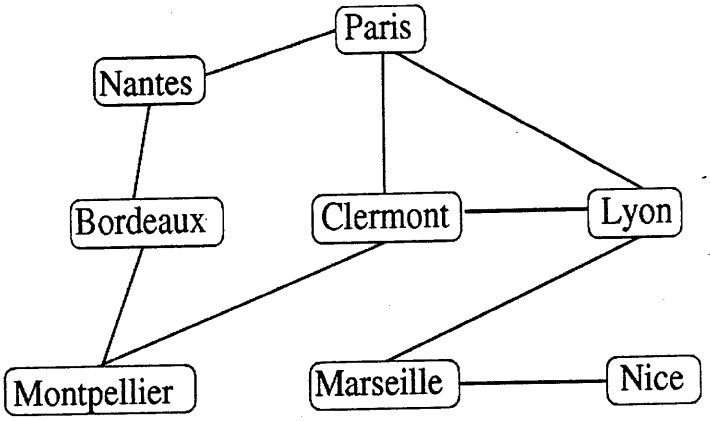
- Soit une carte en deux dimensions
- Une **autoroute** relie une **ville** à une autre
- Sous quelle forme stocker ces informations?
- Traitements possibles
  - Vérifier si toute ville est accessible
  - Trouver des chemins d'une ville à l'autre
  - Trouver quel est le nombre minimum d'autoroute pour que toute ville soit accessible

Représentation



•Une ville est un **noeud** (ou **sommet**)
•Une route est un **arc** (ou **arête**)
•L'ensemble des arcs et noeuds est un **graphe**

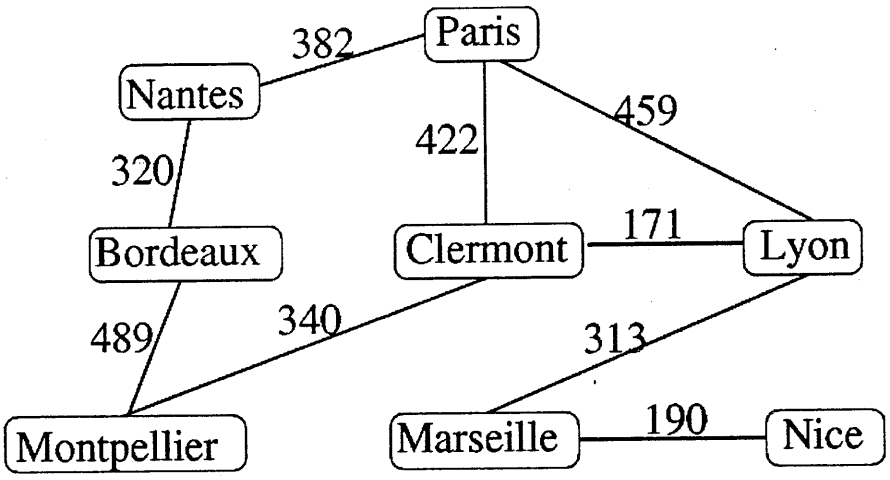
Vocabulaire



- Un **chemin** est une suite de noeuds reliés par des arcs (vert)
- Un **cycle** est un chemin qui part et abouti au même sommet (rouge)
- Un graphe est **connexe** si il existe un (ou plusieurs) chemin entre tout couple de sommet.

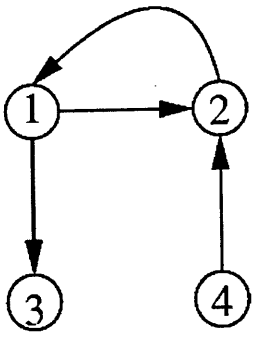
Graphe valué (ou étiqueté)

- Mettre des valeurs sur les arcs!
- L'étiquette peut être de n'importe quel type
- Le noeud peut aussi contenir une valeur (ici, nom de la ville)



Graphes orientés

- Les arcs sont fléchés et ne marchent que dans un seul sens (pour choisir un chemin par exemple!
- Exemple: les automates, pour reconnaître un langage (graphe valué et orienté)



- Le noeud 1 est le **prédécesseur** du noeud 3
- Le noeud 3 est un **successeur** du noeud 1

Matrice d'adjacence

	Lyon	Marseille	Nice	Clermont	Paris	Bordeaux	Nantes	Montpellier
Lyon	–	313	–	171	459	–	–	–
Marseille	313	–	190	–	–	–	–	–
Nice	–	190	–	–	–	–	–	–
Clermont	171	–	–	–	422	–	–	340
Paris	459	–	–	422	–	–	382	–
Bordeaux	–	–	–	–	–	–	320	489
Nantes	–	–	–	–	382	320	–	–
Montpellier	–	–	–	340	–	489	–	–

- Si graphe non orienté –> Symétrique
- Si graphe non valué –> Booléens dans le tableau (l'arc existe?)

Progammation avec matrice

Version pour un nombre borné (n) de sommets

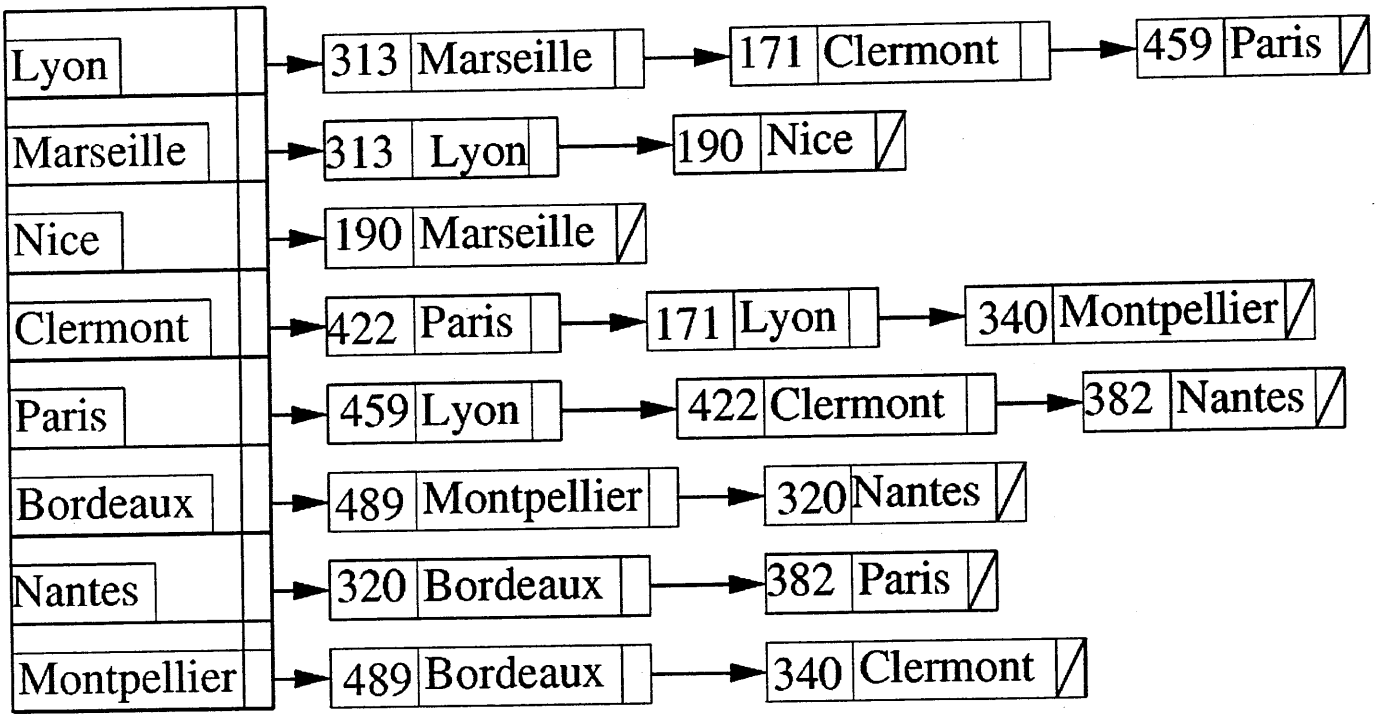
```
class Etiquette {...};
class Graphe {
   - private int n; //nb sommets
   - private Etiquette matrix[][];
   - public Etiquette arc(int s1, int s2){...}
   - public Graphe(int n) {...}
   - public void ajouteArc(int s1, int s2,
     Etiquette e);
   - public void supprimeArc(int s1,s2);
   - public int préd(int s, int i); //i^ème
     prédecesseur de s
   - public int succ(int s, int i); //i^ème
     success de s
};
```

Propriétés du stockage 'matrice'

- Intérêt: très rapide pour
  - accéder à un arc,
  - aux successeurs (ou voisins) d'un noeud
  - aux prédécesseurs d'un noeud
- Inconvénient: Taille (n^2), quelque soit le nombre d'arc
  - OK si graphe **dense** (bcp d'arcs)
  - PB si peu d'arcs

Stockage en liste d'adjacence

- Liste chaînée de successeurs



Programmation par liste d'adjacence

Version pour un nombre borné (n) de sommets

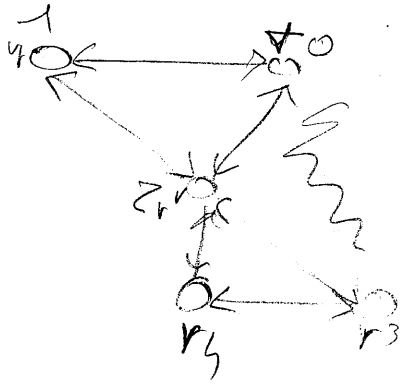
```
class Etiquette {...};
class Graphe {
   - private int n; //nb sommets
   - private Vector adjacence[];
   - public Etiquette arc(int s1, int s2){...}
   - public Graphe(int n) {...}
   - public void ajouteArc(int s1, int s2,
     Etiquette e);
   - public void supprimeArc(int s1,s2);
   - public int préd(int s, int i); //i^ème
     prédecesseur de s
   - public int succ(int s, int i); //i^ème
     success de s
};
```

Propriétés des listes d'adjacence

- Peu coûteux en termes d'espace
  - (O(m), nb d'arcs)
- Parcours des successeurs rapide
- Parcours des prédécesseurs coûteux
- Existence d'un arc coûteuse

Parcours d'un graphe en profondeur

- Le canif suisse des algorithmes de graphe
- int DFS(Noeud u, int k)
  - u.marque=visité
  - Pour tous les successeurs **s** de **u** faire
    - si u.marque==non–visité faire
      - k=DFS(s)
  - u.marque=traité
  - u.ordre=k
  - return k+1;



Parcours DFS (suite)

- Répéter DFS pour les sommets non atteints!
- allDFS(Graphe G)
  - k=0
  - pour tous les noeuds **u** de G faire
    - u.marque=non–visité
  - pour tous les noeuds **u** de G faire
    - si u.marque==non–visité faire
      - k=DFS(u,k)

DFS & recherche de cycle

- bool verifAcyclique(Graph G)
  - allDFS(G)
  - pour tous les noeuds **u** de G faire
    - Pour tous les successeurs **s** de **u** faire
      - Si u.ordre<s.ordre alors
        - return FALSE
  - return TRUE

DFS & tri topologique

- Permet de faire un tri topologique (cf. le fameux numéro d'ordre de chaque sommet) si le graphe est acyclique
- Exemple du diagramme de tâches
  - Détecte les problèmes impossible (cycle)
  - Tri topologique: une solution au pb!

```
package graphe;

import java.util.*;
import java.io.*;

/** Classe Sommet <p> Cette classe ne stocke pour le moment qu'une
    chaine de caractère.
*/
public class Sommet {
    /** Sommet
        <p> Constructeur unique de la classe sommet
    */

    public Sommet(String libelle, int numero)
    {
        this.libelle = libelle;
        this.numero = numero;
        ordre = -1;
        etat = 0;
    }

    /** Methode toString
        @return la version 'chaine de caractèré de l'objet
    */
    public String toString()
    {
        return libelle+"("+numero+"."+ordre+")";
    }


    /** Méthode getNum()
        À tout sommet est associé un entier unique.
        @return renvoie l'entier associé au sommet */
    public int getNum()
    {
        return numero;
    }
    /** Méthode equals
        Redéfinit l'égalité pour ne plus comparer que les numéros
    */
    public boolean equals(Sommet s2)
    {
        return (s2.numero == numero);
    }

    public int getOrdre()
    {
        return ordre;
    }

    private String libelle;
    protected int numero;
    protected int ordre;
    protected int etat;
}
```



```
package graphe;

import java.io.*;
import java.util.*;

/** Classe Etiquette
    <p> Dans notre cas, l'étiquette (la valeur portée par l'arc) sera
     un entier
*/

public class Etiquette {

    public Etiquette(int valeur)
    {
        this.valeur = valeur;
    }
    /** Attibut valeur
        <p> Attention: aucun controle n'est exercé sur cet attribut (public)
     */
    public int valeur;
}
```

```
package graphe;

import java.io.*;
import java.util.*;

/** Classe Graphe
    <p> Stock un graphe sous forme de liste d'adjacence
*/
public class Graphe {
    /** classe Arc
        <p> Arc est une classe INTERNE à graphe, parce que seul graphe doit accé
der à Arc
    */
    public class Arc {
        protected Sommet destination;
        protected Etiquette etiquette;
        public Arc(Sommet destination, Etiquette etiquette)
        {
            this.destination = destination;
            this.etiquette = etiquette;
        }
        protected Arc(Sommet destination)
        {
            this.destination = destination;
            this.etiquette = null;
        }
    }


    /** constructeur graphe
        argument: le nb max de sommets
     */
    public Graphe(int taille)
    {
        this.nbSommets = 0;
        this.taillePhysique = taille;
        adj = new Vector [taille];
        sommets = new Sommet[taille];
    }
    /** méthode ajouteArc
        <p> Ajoute un arc de s1 vers s2, portant l'étiquette e,
 au graphe
 @return Une référence sur le sommet crée
    */
    public Sommet ajouteSommet(Sommet s)
    {
        if(nbSommets ≡ taillePhysique) {
            System.out.println("Dépassement capacité sommet");
            return null;
        }

        int i = nbSommets++;
        s.numero = i;
        sommets[i] = s;
        adj[i] = new Vector();
        return sommets[i];
    }

    /** méthode ajouteArc
        <p> Ajoute un arc de s1 vers s2, portant l'étiquette e,
 au graphe
    */
```



```
    public void ajouteArc(Sommet s1, Sommet s2, Etiquette e)
    {
        Arc nouvelArc = new Arc(s2,e);
        adj[s1.numero].addElement(nouvelArc);
    }

    /** méthode enleveArc
        <p> Enleve un arc de s1 vers s2, portant l'étiquette e,
au graphe
    */
    public void enleveArc(Sommet s1, Sommet s2)
    {
        adj[s1.numero].removeElement(s2);
    }

    /** Methode getEtiquette
        @return <li>
        <ul> l'étiquette de l'arc présent entre les deux sommets
        <ul> "null" si cet arc n'existe pas
        </li>
     */
    public Etiquette getEtiquette(Sommet s1, Sommet s2)
    {
        Arc partiel = new Arc(s1); // Arc incomplet pour recherche

        int index = adj[s1.numero].indexOf(partiel);
        if(index≡-1)
            return null;
        return ((Arc)adj[s1.numero].elementAt(index)).etiquette;
    }


    /** Methode successeur
        @return <li>
        <ul> le ieme successeur suivant sucesseur du sommet s
       <ul> null si s n'a pas de successeurs
        </li>
    */
    public Sommet successeur(Sommet s, int i)
    {
        Vector l = adj[s.numero]; // Simplifie la notation + gain tps
        if(l.size()≤i)
            return null;
        return ((Arc)l.elementAt(i)).destination;
    }

    /** Methode DFS
        <p> Effectue un parcours en profondeur d'abord sur le graphe et numérote
les sommets visités
        @return le numéro d'ordre à attribuer prochainement
    */
    protected int DFS(Sommet s, int k)
    {
        s.etat = 1; // visité

        int j = 0;
        Sommet succ;
        do {
            succ = successeur(s,j++);
            if(succ≠null ∧ succ.etat≡0)
                k = DFS(succ,k);
        } while (succ≠null);
```

```
        s.ordre = k;
        s.etat = 2;

        return k+1;
    }

    /** Methode allDFS
        <p> Effectue autant de parcours DFS que nécessaire pour numéroter tous l
es sommets du graphe
    */
    public void allDFS()
    {
        for(int i=0; i<nbSommets; i++)
            sommets[i].etat=0; // Marque le sommet comme non visité

        int k = 0;
        for(int i=0; i<nbSommets; i++)
            if(sommets[i].etat==0)
                k = DFS(sommets[i],k);

    };

    /** Methode acyclique
        @return vrai si le graphe est acyclique
    */
    public boolean acyclique()
    {
        allDFS();
        for(int i=0; i<nbSommets; i++)
            {
                Sommet s = sommets[i];
                int j = 0;
                Sommet succ;
                do {
                    succ = successeur(s,j++);
                    if(succ≠null ∧ succ.ordre ≥ s.ordre)
                        return false;
                } while (succ≠null);
            }
        return true;
    }

    private int nbSommets;
    private int taillePhysique;
    private Vector adj[];
    private Sommet sommets[];
}
```



```
import graphe.Sommet;
import graphe.Graphe;
import graphe.Etiquette;
import java.io.*;
import java.util.*;

class TestGraphe {
    public static void main(String argv[])
    {
        int nbSommets = 8;
        Graphe g = new Graphe(nbSommets);
        Sommet s[] = new Sommet[nbSommets];
        for(int i=0; i<nbSommets;i++)
            s[i] = g.ajouteSommet(new Sommet("",i));

        Etiquette e = new Etiquette(1);
        g.ajouteArc(s[0],s[1],e);
        g.ajouteArc(s[0],s[2],e);
        g.ajouteArc(s[0],s[7],e);
        g.ajouteArc(s[1],s[5],e);
        g.ajouteArc(s[1],s[6],e);
        g.ajouteArc(s[1],s[4],e);
        g.ajouteArc(s[2],s[3],e);
        g.ajouteArc(s[3],s[4],e);
        g.ajouteArc(s[6],s[2],e);
        g.ajouteArc(s[7],s[5],e);
        g.ajouteArc(s[5],s[2],e);
        /*
          L'ajout d'un de ces arcs rend le graphe acyclique
        g.ajouteArc(s[1],s[1],e);
        g.ajouteArc(s[7],s[7],e);
        g.ajouteArc(s[1],s[2],e);
        g.ajouteArc(s[4],s[7],e);
        g.ajouteArc(s[7],s[0],e);
        g.ajouteArc(s[3],s[6],e);
        */
        for(int i=0; i<nbSommets;i++)
            {
                System.out.print("Successeurs de "+s[i]+": ");

                int j = 0;
                Sommet succ;
                do {
                    succ = g.successeur(s[i],j++);
                    if(succ≠null)
                        System.out.print(""+succ+",");
                } while (succ≠null);
                System.out.println(" ("+(j-1)+" successeurs)");
            }
        if(g.acyclique())
            System.out.println("Graphe acyclique");

        for(int i=0; i<nbSommets;i++)
            System.out.println(""+s[i]);
    }
}
```