COMPOSITION D'INFORMATIQUE 1 de INF431

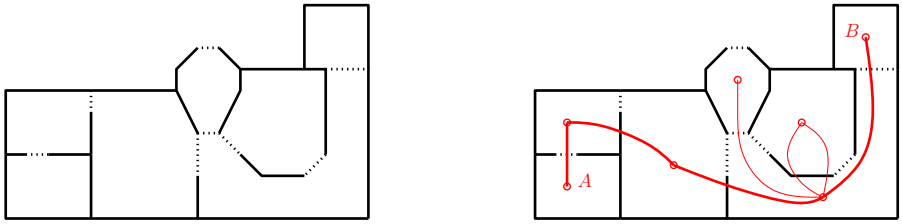
Olivier Devillers, Gilles Schaeffer, François Morain

16 avril 2008

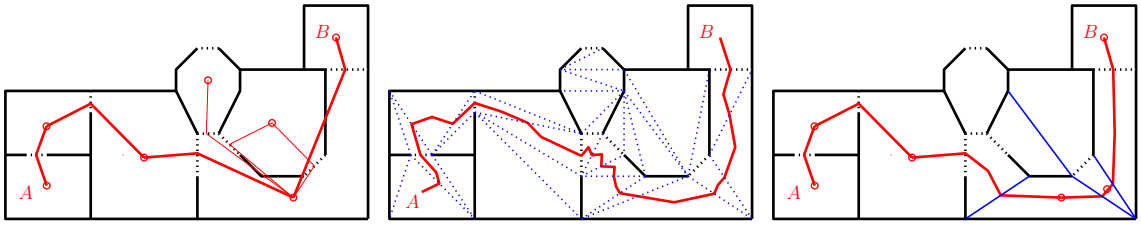
Tous les documents du cours sont autorisés. On attachera une grande importance à la concision, à la clarté, et à la précision de la rédaction.

Introduction

On considère un problème de calcul automatique de chemins dans un bâtiment. On suppose le bâtiment sur un seul niveau, décrit par un plan en coupe horizontale sur lequel apparaissent les éléments constitutifs de base : des cloisons et des portes délimitant des pièces. Afin de faciliter son traitement informatique le plan est donné sous forme d'une structure de *graphe plan* dont la manipulation fait l'objet de la première partie du sujet, et dont les arêtes représentent les cloisons ou portes et les sommets représentent leurs extrémités.



Pour chercher un chemin entre une pièce A et une pièce B dans un bâtiment il est naturel d'y associer un autre graphe dont les sommets sont les pièces et tels que deux pièces voisines par une porte soient reliées par une arête : le problème se ramène alors à chercher un chemin dans ce graphe, joignant les sommets A et B. On obtient ainsi un *chemin combinatoire*, formé d'une suite alternée de pièces et de portes à traverser pour aller de la pièce A à la pièce B. Cependant on se pose le problème de calculer un *chemin géométrique* qu'on puisse dessiner sur le plan original, ou faire parcourir à un robot. Il faut donc donner un lieu géométrique au chemin combinatoire de façon à ce qu'il joigne effectivement un point de la première pièce à un point de la seconde sans rencontrer aucune cloison.



On voit sur l'exemple qu'il ne suffit pas de joindre le barycentre des pièces visitées successivement par le chemin combinatoire même en passant par le milieu des portes ; en particulier, la notion de barycentre n'est pas bien adaptée si la géométrie des pièces est complexe. On explore donc dans la 2ème partie du sujet une solution consistant à trianguler le plan de façon à découper les pièces en régions triangulaires : on peut alors tracer un chemin géométrique en utilisant les barycentres de ces triangles et les milieux des portes.

PARTIE 1 : Représentation des graphes plans

Définitions

Le plan du bâtiment est un *graphe plan* (graphe dessiné dans le plan), avec des *arêtes*, qui sont les cloisons et les portes, et des *sommets*, qui sont les points de jonction ou extrémité de ces cloisons et portes. Le dessin du graphe dans le plan définit aussi des *faces* qui sont les pièces du bâtiment. Il y a une face infinie qui correspond à l'extérieur du bâtiment.

Tous les graphes plans que l'on considère sont supposés connexes.

Du point de vue informatique, nous allons représenter notre graphe à l'aide de classes java `Sommet`, `Face` et `DemiArete`, définies plus loin. À chaque sommet ou face du graphe correspondra un objet de la classe `Sommet` ou `Face`, tandis que chaque arête sera associée à deux demi-arêtes jumelles opposées : on imagine qu'on a fendu chaque arête en deux dans le sens de la longueur, en orientant chaque moitié de façon à ce que sa jumelle soit à droite. On peut noter qu'à chaque face correspond alors un cycle de demi-arêtes qui longe son bord, dans le sens anti-horaire (contraire aux aiguilles d'une montre) pour les faces finies et dans le sens horaire pour la face infinie. Inversement une demi-arête ne touche qu'une seule face, sur sa gauche. Cette représentation est illustrée par la figure ci-dessous :

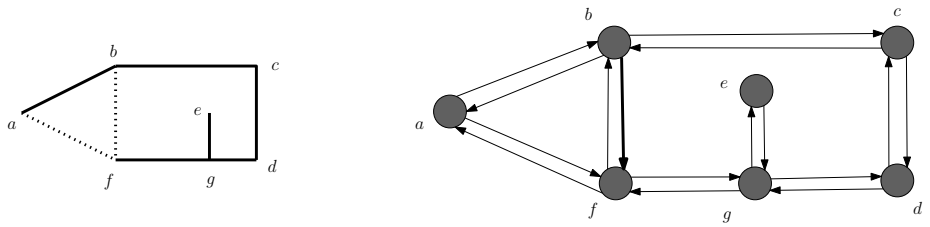


FIG. 1 – Un exemple de bâtiment et son dessin avec les demi-arêtes.

Le graphe du bâtiment a ici 7 sommets a, b, c, d, e, f et g, 3 faces afb, $bfgegdc$ plus la face extérieure $abcdgf$, et 8 arêtes $\{af\}$, $\{bf\}$, $\{ab\}$, $\{fg\}$, $\{ge\}$, $\{dg\}$, $\{cd\}$ et $\{bc\}$. Sa représentation en machine utilisera donc 7 objets de la classe `Sommets`, 4 de la classe `Face` et 16 de la classe `DemiArete`.

Voici les définitions des trois classes java que nous utilisons :

```
class Sommet {
    double x;
    double y;
    DemiArete arete;
    Sommet() {}
    Sommet(double xx, double yy, DemiArete aa){
        x=xx;y=yy;arete=aa;
    }
}

class DemiArete {
    int type;
    DemiArete jumelle;
    DemiArete suivante;
    DemiArete precedente;
    Sommet origine;
    Face f;
    DemiArete() {}
    DemiArete(DemiArete j, DemiArete s, DemiArete p, Sommet o, Face ff, int t){
       jumelle = j; suivante = s; precedente = p; origine = o; f = ff; type=t;
    }
}
class Face {
    DemiArete arete;
```

```
    Face() {}
}
```

La classe `Sommet` contient deux champs de type **double** pour retenir les coordonnées géométriques du sommet, et un champ de type `DemiArete` indiquant l'*une* des demi-arêtes issues du sommet (on verra que les demi-arêtes sont d'une certaine manière chaînées entre elle, ce qui rend inutile de retenir *toutes* les demi-arêtes issues d'un sommet dans l'objet `Sommet` associé).

La classe `DemiArete` contient un champ de type **int** pour le type (1 : cloison, 2 : porte, 3 : autre), des champs `jumelle`, `origine` et `f` permettant d'indiquer respectivement la demi-arête jumelle, le sommet origine et la face voisine, et enfin deux champs supplémentaires `suivante` et `precedente` qui servent à organiser les demi-arêtes d'une même face en liste circulaire doublement chaînée. Ainsi considérons la demi-arête bf de l'exemple, représentée en gras sur le dessin : pour cette demi-arête, la suivante est la suivante dans la face soit fg, la précédente est cb, la jumelle est fb, la face est $bfgegdc$ et l'origine est b.

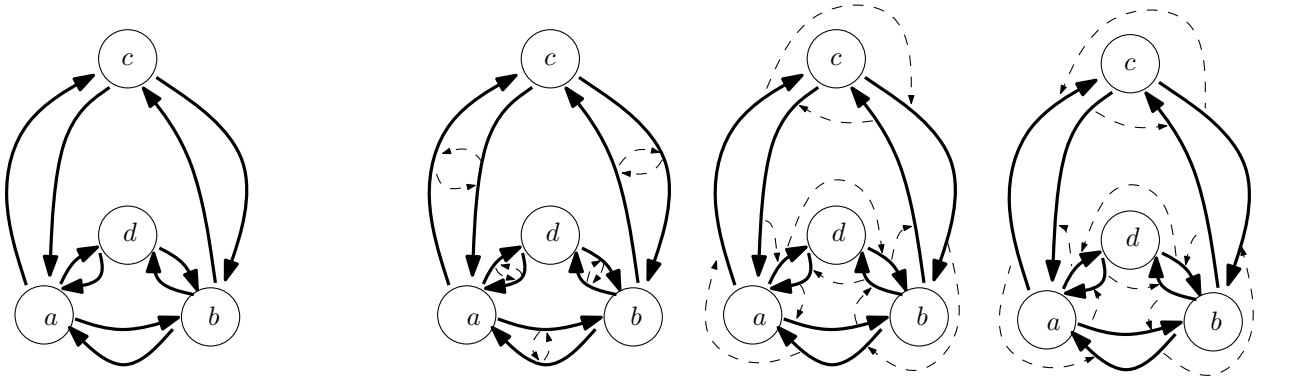
Finalement, la classe `Face` contient un seul champ de type `DemiArete`, qui permet d'indiquer l'une des demi-arêtes de la face : en vertu de la structure doublement chaînées des demi-arêtes d'une même face, cela suffit à pouvoir parcourir la face.

Une représentation machine correcte d'un graphe plan avec n sommets, p faces et m arêtes est donc la donnée de n objets `Sommet`, p objets `Face` et $2m$ objets `DemiArete`, dont les champs sont remplis de manière cohérente avec le graphe plan.

Remarquons enfin qu'on a pas prévu de champ pour les coordonnées des arêtes : les arêtes de nos graphes plans sont des segments, leur description géométrique se déduit donc des coordonnées des sommets origines des deux demi-arêtes associées.

1.1 Question

Voici un petit graphe plan formé de 4 sommets a, b, c et d de 5 arêtes (et donc 10 demi-arêtes) et de 3 faces (en comptant la face infinie). Dans la figure ci dessous vous voyez à gauche la description du graphe et de ses demi-arêtes, puis de gauche à droite sont montrés en pointillés les différents types de relations entre les demi-arêtes.



graphe plan — pointeur d'une demi-arête vers sa jumelle — pointeur d'une demi-arête vers sa suivante — pointeur d'une demi-arête vers sa précédente

Donner les lignes java (une quarantaine) nécessaires à construire en machine une représentation de ce graphe plan en complétant le squelette suivant

```
//retourne ab
static DemiArete question1(){
    Face infini = new Face();
    Face abd = new Face();
    Face adbc = new Face();
    Sommet a = new Sommet(0,0,null);
    Sommet b = new Sommet(2,0,null);
    Sommet c = new Sommet(1,3,null);
    Sommet d = new Sommet(1,1,null);
    DemiArete ab = ...
    ...
    return ab;
```

```
}
```

1.2 Question

Écrire une méthode d'objet

```
public double distanceCarree()
```

de la classe `DemiArete` qui calcule le carré de la longueur d'une arête supposée correctement définie.

1.3 Question

Écrire une méthode d'objet

```
public int taille()
```

de la classe `Face` qui calcule la taille d'une face (supposée correctement définie), c'est-à-dire le nombre de demi-arêtes qu'on parcourt lorsqu'on fait le tour de la face.

1.4 Question

Écrire une méthode d'objet

```
public int degre()
```

de la classe `Sommet` qui calcule le nombre de demi-arêtes ayant ce sommet pour origine.

On remarquera que si `aa` est une demi-arête d'origine `b` alors `aa.precedent.jumelle` est la demi-arête suivante d'origine `b` en sens anti-horaire autour de `b`.

1.5 Question

Écrire une méthode statique

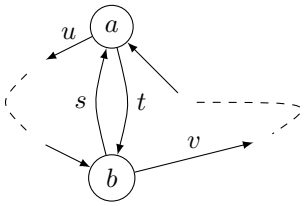
```
public static void joindreSommets(DemiArete u, DemiArete v)
```

de la classe `DemiArete` qui ajoute une arête (formée de deux demi-arêtes `s` et `t`) pour relier les sommets `a=u.origine` et `b=v.origine` en découpant la face `f=u.face=v.face` en deux.

On ne demande pas ici de tester la validité de la structure obtenue : on suppose que `u` et `v` sont des demi-arêtes de la face `f` telles que le segment `s=(a,b)` vérifie les conditions suivantes :

- `s` est strictement à l'intérieur de `f` (autrement dit, `s` est dans la face `f`).
- autour de `a`, `t` est dans le secteur angulaire `(u.precedente.jumelle),u`,
- autour de `b`, `s` est dans le secteur angulaire `(v.precedente.jumelle,v)`.

On veut qu'après insertion, `u.precedente` soit `s` et `v.precedente` soit `t`, et plus généralement que la structure obtenue décrive correctement le graphe plan dans lequel on a inséré l'arête `st`. La fonction devra créer une nouvelle face à laquelle `u` appartiendra, tandis que `v` restera dans l'ancienne face.



1.6 Question

Écrire une méthode

```
public static void aretePendante(double x, double y, DemiArete u)
```

de la classe `DemiArete` qui crée et insère un sommet `b` de coordonnées `(x,y)` dans la face associée à l'arête courante `u` et crée une arête entre `b` et `a=u.origine`. Ici encore on ne demande pas de tester la validité de la structure obtenue : on suppose que les coordonnées du point `b=(x,y)` sont telles que le segment `ab` soit dans la face `u.face`.

Un exemple d'arête pendante est l'arête $\{ge\}$ de la figure 1.

1.7 Question

On se donne une méthode

```
static DemiArete premiereArete(double ax, double ay, double bx, double by)
{
  Face infini = new Face();
  Sommet a = new Sommet(ax,ay,null);
  Sommet b = new Sommet(bx,by,null);
  DemiArete u = new DemiArete(null,null,null,a,infini,1);
  DemiArete v = new DemiArete(u,u,u,b,infini,1);
  u.jumelle = v;
  u.precedente = v;
  u.suivante = v;
  a.arete = u;
  b.arete = v;
  infini.arete = u;
  return u;
}
```

qui crée la structure associée au graphe plan avec une unique arête d'extrémités `(ax,ay)` et `(bx,by)`.

Expliquez ce que fait le code suivant :

```
DemiArete ab = DemiArete.premiereArete(0, 0, 1, 1);
DemiArete fa = DemiArete.aretePendante(1, -1, ab);
DemiArete.joindreSommets(ab.jumelle, fa);
Face infinie = ab.face;
```

1.8 Question

Utiliser la méthode `premiereArete` ainsi que les méthodes `aretePendante` et `joindreSommets` pour écrire une méthode qui crée la structure associée à l'exemple du début de cette partie (Figure 1).

1.9 Question

Le but de la question est de programmer un parcours en profondeur d'abord (dfs) d'un graphe plan en utilisant la structure associée.

On suppose que les trois classes `DemiArete`, `Face` et `Sommet` disposent en plus du champ `marque`.

Écrire une méthode statique

```
public static void marque(Sommet s)
```

de la classe `DemiArete` qui réalise un parcours en profondeur d'abord partant du sommet `s`. On suppose que toutes les marques de sommets sont initialisées à **true**. La fonction doit faire passer les marques de tous les sommets, demi-arêtes et faces du graphe plan à **false**.

PARTIE 2 : Triangulation

Une triangulation géométrique est un graphe plan dont toutes les faces finies sont des triangles.

2.1 Question

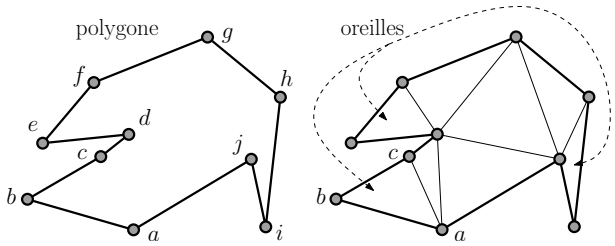
On note n le nombre de sommets, k le nombre de sommets sur le bord de la face infinie m le nombre d'arêtes ($2m$ le nombre de demi-arêtes) et f le nombre de faces.

Utiliser le fait que dans une triangulation toutes les faces sont des triangles sauf la face infinie, et la relation d'Euler pour trouver le nombre de demi-arêtes, d'arêtes et de triangles en fonctions de n et k.

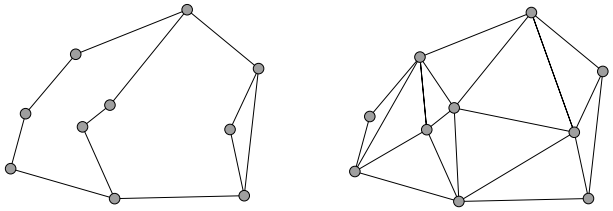
(si vous n'avez pas bien regardé le poly ! la relation d'Euler c'est $n - m + f = 2$)

Triangulation d'un graphe plan

Pour transformer un graphe plan en triangulation on va trianguler successivement toutes ses faces. Pour trianguler une face (un polygone) ayant au moins 4 sommets, on va s'appuyer sur le fait qu'une triangulation d'un polygone a toujours au moins une *oreille*, c'est-à-dire un triangle dont deux cotés sont sur le bord du polygone. Plus précisément, on admet le résultat suivant : « Il existe `a,b,c` trois sommets consécutifs d'un polygone P tels que `ab` et `bc` sont des arêtes de P et le segment ouvert `ac` est strictement à l'intérieur de P. » Une fois trouvé de tels sommets `a,b,c` ajouter l'arête `ac` sépare P en un triangle et un polygone P' ayant une arête en moins que P auquel on peut appliquer le résultat pour poursuivre la triangulation à son tour.



La figure ci-dessous montre un graphe plan non triangulé à gauche et une possibilité de graphe plan triangulé à droite.



On se donne une méthode statique (qu'il n'est pas demandé d'écrire)

```
public static int orientation(Sommet a, Sommet b, Sommet c)
```

de la classe `Sommet` qui retourne 1 si le triangle `abc` est direct -1 si il est indirect et 0 si les trois points sont alignés. Le triangle est direct si en tournant dans le sens `abc` on tourne dans le sens anti-horaire. (Pour information, une façon d'écrire une telle méthode est d'utiliser le fait que le déterminant de la matrice $\begin{pmatrix} x_0 & x_1 \\ y_0 & y_1 \end{pmatrix}$ est égal à l'aire du parallélépipède de sommets $\binom{0}{0}$, $\binom{x_0}{y_0}$, $\binom{x_0+x_1}{y_0+y_1}$, $\binom{x_1}{y_1}$ si celui ci est direct, et à son opposé s'il est indirect.)

2.2 Question

Utiliser la méthode `orientation` pour écrire une méthode statique

```
public static boolean coupe(Sommet a, Sommet b, DemiArete u)
```

de la classe `Sommet` qui teste si le segment `ab` coupe l'arête `u`. On supposera que trois sommets différents du graphe plan ne sont jamais parfaitement alignés (cette hypothèse n'est pas très réaliste pour un plan de bâtiment, mais le traitement correct des cas d'alignements est un peu trop fastidieux pour être demandé en temps limité).

Pour simplifier la suite, si `a` (resp. `b`) est une extrémité de `u`, alors on convient que `ab` ne coupe pas `u`.

2.3 Question

Écrire une méthode d'objet

`public boolean oreille()`

de la classe `DemiArete` qui crée une oreille en utilisant la demi-arête courante et la suivante si cela ne crée pas d'intersection et retourne **true** dans ce cas. S'il n'est pas possible de rajouter l'arête, la méthode retourne **false**.

2.4 Question

Écrire une méthode d'objet

`public void triangulePolygone()`

de la classe `DemiArete` qui triangule la face associée à la demi-arête courante.

2.5 Question

Écrire une méthode statique

`public static void triangule(Sommet s)`

de la classe `DemiArete` qui triangule le graphe plan auquel appartient le sommet `s`. On peut supposer toutes les marques initialisées à **false** (et on doit les remettre à **false** à la fin).

2.6 Question

Quelle est la complexité de votre méthode de triangulation géométrique en fonction de n le nombre de sommets ?

Même question, si on suppose que dans la triangulation de départ la taille des faces est bornée par une constante κ.

2.7 Question

Donner un algorithme en pseudocode qui, étant données deux faces `f`, `g` d'une structure de graphe plan triangulé renvoie un chemin géométrique allant d'un point de la face `f` à un point de la face `g`. Le chemin sera donné par une liste de coordonnées `(x_i,y_i)` des extrémités des segments qui le composent.

On utilisera le fait que si deux triangles `abc` et `abd` sont adjacents par une arête `ab` alors le chemin allant du barycentre de `abc` au barycentre de `abd` en passant par le milieu de `ab` ne rencontre que l'arête `ab`.

CORRIGÉ de la COMPOSITION D'INFORMATIQUE 1 de INF431

Olivier Devillers, Philippe Chassignet

2008

PARTIE 1 : Représentation des cartes planaires

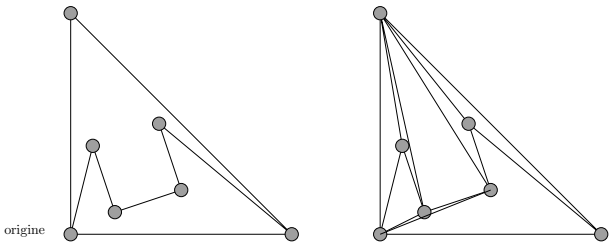
Voir le code java à la fin.

1.7

Le code créé un triangle abf en partant de ab puis en rajoutant une arête pendante fa. La méthode `joindreSommets` fait exactement ce qu'il faut pour terminer. Enfin on affecte à la variable `infini` la face infinie. Les 3 premières lignes peuvent servir de début pour la question suivante.

PARTIE 2 : Triangulation

Voir le code java à la fin sauf pour les questions ci dessous. Le programme principal donné dans le code construit la carte et la triangulation de la figure suivante :



2.1

La relation d'Euler est $f-m+n=2$ (polycopié). Le nombre de demi-arêtes est bien évidemment $2m$, par ailleurs comme toutes les faces sont des triangles on peut compter les demi-arêtes par $3(f-1)+k$, en effet chacun des $f-1$ triangles a 3 demi-arêtes et la face infinie en a k.

De $f-m+n=2$ et $2m=3f+k-3$ on obtiens le nombre de triangle $f-1=2n-2-k$ et $m=3n-3-k$.

2.6

La méthode `Oreille` doit vérifier les intersections avec les autres arêtes de la face. Cela peut prends un temps $O(\kappa)$. κ peu être de l'ordre de n dans le cas le pire.

Si on s'y prends bien, la triangulation d'une face fait un nombre linéaire (dans la taille de la face) d'appels à `oreille`. En effet soit un appel à `oreille` est fructueux et la taille du morceau de face restant à trianguler décroît, soit un appel à `oreille` est infructueux, mais cette oreille ne sera plus jamais candidate (à condition que en cas d'appel fructueux à `oreille` on prenne la peine de démarrer la recherche de l'oreille suivante au bon endroit).

Le coût global de la fonction `triangulePolygone` est proportionnel au nombre d'appels à `oreille` donc $O(m)$.

Le coût global de la fonction `trianguleR` est proportionnel au nombre de fois où elle est appelée. À part le premier appel, on appelle `trianguleR(s)` pour chaque demi-arête arrivant en `s` soit $2m$ fois.

Comme $m = O(n)$ on a un algorithme linéaire en nombre d'appels à `oreille` et on a donc une complexité $O(n\kappa)$ qui donne $O(n^2)$ en l'absence d'hypothèse sur κ. Obtenir une complexité $O(n)$ $(\forall\kappa)$ est possible mais c'est beaucoup plus compliqué.

2.7

la fonction suivante va imprimer le chemin cherché (`g` en premier et `f` en dernier).

```
boolean chemin(f,g) {
// retourne vrai si un tel chemin existe
// on suppose les sommets sont marqués initiallement à faux
// il faudrait les remarquer faux après en appellant marque
  si f est marqué vrai retourner faux;
  marquer f à vrai;
  si f=g imprimer barycentre(f) et retourner vrai;
  pour chaque demi-arete a sur le bord de f qui est une porte{
     h = face de la jumelle de a; // h est une piece voisine de f
     si chemin(h,g) est vrai alors {
           imprimer milieu(a);
           imprimer barycentre(f);
           retourner vrai;
     }
  }
  retourner faux;
}
```

Code java

Vous pouvez le prendre sur la page web, le compiler et l'essayer (la sortie est un peu pauvre je le reconnais).

```
import java.util.*;

class Sommet {
    boolean marque;
    double x;
    double y;
    DemiArete arete;
    Sommet() {;}
    Sommet(double xx, double yy, DemiArete aa) {x=xx;y=yy;arete=aa;}

    public boolean valide()
    {
        if (arete.origine != this)
            { print(); System.out.println(" pas bonne arete");return false;}
        return true;
    }

    public int degre()
    {
        int d=0;
        DemiArete aa = arete;
        do {
            ++d;
            aa = aa.precedente.jumelle;
        }while(aa != arete);
        return d;
    }
```

```
    public static int orientation(Sommet a, Sommet b, Sommet c)
    {
        double x0 = b.x - a.x; double x1 = c.x - a.x ;
        double y0 = b.y - a.y; double y1 = c.y - a.y ;
        double det = x0 * y1 - x1 * y0 ;
        return ( (det>0)? 1 : ( (det<0) ? -1 : 0) ) ;
    }

    public static boolean coupe(Sommet a, Sommet b, DemiArete e)
    {// ne marche pas si les 4 sommets sont alignés
        return
            (orientation( a , b, e.origine)
             != orientation( a , b, e.jumelle.origine))
            && (orientation( e.origine, e.jumelle.origine, a)
                != orientation( e.origine, e.jumelle.origine, b) ) ;
    }

    public void print(){ System.out.print("("+x+", "+y+")"); }
}

class Face {
    boolean marque;
    int couleur;
    DemiArete arete;
    Face() {;}
    public int taille()
    {
        int t=0;
        DemiArete aa = arete;
        do {
            ++t;
            aa = aa.suivante;
        }while(aa != arete);
        return t;
    }
    public boolean valide()
    {
        if (arete.face != this)
            { arete.print();System.out.println("Face pas bonne arete");return false;}
        return true;
    }
}

class DemiArete {
    int type;
    boolean marque;
    DemiArete jumelle;
    DemiArete suivante;
    DemiArete precedente;
    Sommet origine;
    Face face;
    DemiArete() {};
    DemiArete(DemiArete j, DemiArete s, DemiArete p, Sommet o, Face ff, int t)
    {jumelle = j; suivante = s; precedente = p; origine = o; face = ff; type=t;}

    public boolean valide()
    {
        if (jumelle.jumelle != this)
            { print(); System.out.println(" pas bonne jumelle");return false;}
        if (suivante.precedente != this)
            { print(); System.out.println(" pas bonne suivante");return false;}
```

```
    if (precedente.suivante != this)
        { print(); System.out.println(" pas bonne precedente");return false;}
    return origine.valide() && face.valide() ;
}

public double distanceCarree()
{
    double xx = origine.x - jumelle.origine.x;
    double yy = origine.y - jumelle.origine.y;
    return xx*xx + yy*yy;
}

public static void joindreSommets(DemiArete u, DemiArete v)
{
    // creer la nouvelle arete et la nouvelle face
    DemiArete t = new DemiArete(null,v,u.precedente,u.origine,v.face,1);
    Face nouvFace = new Face();
    DemiArete s = new DemiArete(t,u,v.precedente,v.origine,nouvFace,1);
    // mettre les pointeurs faces aretes
    nouvFace.arete = s;
    v.face.arete = t;
    // raccrocher la jumelle
    t.jumelle = s;
    // raccrocher les anciennes aretes
    u.precedente = s;
    v.precedente = t;
    t.precedente.suivante = t;
    s.precedente.suivante = s;
    // mettre a jour les pointeurs aretes / faces
    for ( ; u != s; u = u.suivante) u.face = nouvFace;
}

static DemiArete premiereArete(double ax, double ay, double bx, double by)
{
    Face infini = new Face();
    Sommet a = new Sommet(ax,ay,null);
    Sommet b = new Sommet(bx,by,null);
    DemiArete u = new DemiArete(null,null,null,a,infini,1);
    DemiArete v = new DemiArete(u,u,u,b,infini,1);
    u.jumelle = v;
    u.precedente = v;
    u.suivante = v;
    a.arete = u;
    b.arete = v;
    infini.arete = u;
    return u;
}

public static DemiArete aretePendante(double x, double y, DemiArete u)
{
    DemiArete ab = new DemiArete(null, null, u.precedente, u.origine, u.face, 1);
    DemiArete ba = new DemiArete(ab, u, ab, null, u.face, 1);
    Sommet b = new Sommet(x, y, ba);
    // on complete a la place des null
    ab.jumelle  = ba;
    ab.suivante = ba;
    ba.origine = b;
    // on raccroche les vieilles aretes
    ab.precedente.suivante = ab;
    u.precedente = ba;
    return ba;
```

```
}

public static void marque(Sommet s)
{
    if (s.marque == false) return; // sommet deja visite

    s.marque = false;
    DemiArete debut = s.arete;
    DemiArete aa = debut;
    do {
        aa.marque = false;
        aa.jumelle.marque = false;
        aa.face.marque = false;
        marque(aa.jumelle.origine);
        aa = aa.precedente.jumelle;
    }while(aa != debut);
}

public static void print(Sommet s)
{
    printR(s);
    marque(s);
}
public void print(){
        origine.print();
        System.out.print("--> ");
        jumelle.origine.print();
        System.out.println();
}
public static void printR(Sommet s)
{
    if (s.marque == true) { // sommet deja visite
        return;
    }

    s.marque = true;
    DemiArete debut = s.arete;
    DemiArete aa = debut;
    do {
        aa.print();
        printR(aa.jumelle.origine);
        aa = aa.precedente.jumelle;
    }while(aa != debut);
}

public static void toutValide(Sommet s)
{
    toutValideR(s);
    marque(s);
}
public static void toutValideR(Sommet s)
{
    if (s.marque == true) { // sommet deja visite
        return;
    }

    s.marque = true;
    DemiArete debut = s.arete;
    DemiArete aa = debut;
    do {
        aa.valide();
```

```
        toutValideR(aa.jumelle.origine);
        aa = aa.precedente.jumelle;
    }while(aa != debut);
}

public boolean oreille()
{
    DemiArete aa = suivante.suivante.suivante;
    if (1 != Sommet.orientation(origine,
                                suivante.origine,
                                suivante.suivante.origine))
        return false;
    //l'angle est pas convexe ca peut pas etre une oreille

    if ((1 != Sommet.orientation(origine,
                                 suivante.suivante.origine,
                                 aa.origine))
        &&(1 == Sommet.orientation(suivante.origine,
                                   suivante.suivante.origine,
                                   aa.origine)))
        return false;
    //le sommet suivant est dans l'oreille


    // maintenant on teste toutes les aretes non adjacentes
    // a l'oreille candidate pour test d'intersection
    while(aa!=precedente) {
        if ( Sommet.coupe(origine, suivante.suivante.origine, aa) ){
            return false;
        }
        aa = aa.suivante;
    }
    joindreSommets(this, this.suivante.suivante);
    return true;
}

public void triangulePolygone()
{
    if ( suivante.suivante.suivante == this )return;  // test si triangle
    if ( oreille()) {
    // on essaye de couper l'oreille, si ca marche on recommence
    // dans le morceau restant en demarrant a un endroit astucieux
    // (le demarrage est juste important pour la complexite)
        precedente.jumelle.precedente.triangulePolygone();
    }
    else {
    // si ca marche pas on tourne dans la face pour essayer
    // l'oreille suivante
     suivante.triangulePolygone();
    }
}

public static void triangule(Sommet s)
{
    trianguleR(s);
    marque(s);
}
public static void trianguleR(Sommet s)
{
    if (s.marque == true) return; // sommet deja visite
```

```
        s.marque = true;
        DemiArete debut = s.arete;
        DemiArete aa = debut;
        do {
            aa.triangulePolygone();
            trianguleR(aa.jumelle.origine);
            aa = aa.precedente.jumelle;
        }while(aa != debut);
    }

}




class Pale {

    static DemiArete question8()
    {//retourne ab
        DemiArete fa = DemiArete.premiereArete( 1, -1, 0, 0);
        DemiArete ba = DemiArete.aretePendante( 1,  1, fa);
        DemiArete cb = DemiArete.aretePendante( 3,  1, ba);
        DemiArete dc = DemiArete.aretePendante( 3, -1, cb);
        DemiArete gd = DemiArete.aretePendante( 2, -1, dc);
        DemiArete eg = DemiArete.aretePendante( 2,  0, gd);
        ba.joindreSommets(ba        , fa); // fb
        ba.joindreSommets(eg.jumelle, fa); // fg
        //maintenant dire ou sont les portes
        fa.type = 2;
        fa.jumelle.type = 2;
        ba.precedente.type = 2;            //fb
        ba.precedente.jumelle.type = 2;
        return ba.jumelle;
    }

    static DemiArete question1()
    {//retourne ab
        Face infini = new Face();
        Face abd = new Face();
        Face adbc = new Face();
        Sommet a = new Sommet(0,0,null);
        Sommet b = new Sommet(2,0,null);
        Sommet c = new Sommet(1,3,null);
        Sommet d = new Sommet(1,1,null);
        DemiArete ab = new DemiArete(null,null,null,a,abd,1);
        DemiArete bd = new DemiArete(null,null, ab ,b,abd,1);
        DemiArete da = new DemiArete(null, ab , bd ,d,abd,1);
        ab.suivante   = bd;
        ab.precedente = da;
        bd.suivante   = da;
        DemiArete ad = new DemiArete( da ,null,null,a,adbc,1);
        DemiArete db = new DemiArete( bd ,null, ad ,d,adbc,1);
        DemiArete bc = new DemiArete(null,null, db ,b,adbc,1);
        DemiArete ca = new DemiArete(null, ad , bc ,c,adbc,1);
        da.jumelle    = ad;
        bd.jumelle    = db;
        ad.suivante   = db;
        ad.precedente = ca;
        db.suivante   = bc;
        bc.suivante   = ca;
```

```
        DemiArete ac = new DemiArete( ca ,null,null,a,infini,1);
        DemiArete cb = new DemiArete( bc ,null, ac ,c,infini,1);
        DemiArete ba = new DemiArete( ab , ac , cb ,b,infini,1);
        ca.jumelle    = ac;
        bc.jumelle    = cb;
        ab.jumelle    = ba;
        ac.suivante   = cb;
        ac.precedente = ba;
        cb.suivante   = ba;
        infini.arete=ac;
        abd.arete=ab;
        adbc.arete=ad;
        a.arete=ab;
        b.arete=bc;
        c.arete=ca;
        d.arete=da;
        return ab;
    }

    public static void main(String[] args) {


        System.out.println("......Question 1.1......");
        DemiArete q1 = question1();
        DemiArete.toutValide(q1.origine);
        DemiArete.print(q1.origine);

        System.out.println("longueur carree d'une arete : "+q1.distanceCarree() );
        System.out.println("taille d'une face (abd) : "+q1.face.taille() );
        System.out.println("taille d'une face (adbc): "+q1.suivante.jumelle.face.taille() );
        System.out.println("degre d'un sommet (d) : "+q1.precedente.origine.degre() );
        System.out.println("degre d'un sommet (a) : "+q1.origine.degre() );

        System.out.println("......Question 1.8......");
        DemiArete q8 = question8();
        DemiArete.toutValide(q8.origine);
        DemiArete.print(q8.origine);

        System.out.println("......Carte planaire......");
        DemiArete a1 = DemiArete.premiereArete(0,0,10,0);
        Sommet s1 = a1.origine;
        DemiArete.aretePendante(0,10,a1);
        a1.joindreSommets(a1.jumelle,a1.precedente);
        DemiArete.aretePendante(1,4,a1);
        DemiArete.aretePendante(2,1,a1.precedente);
        DemiArete.aretePendante(5,2,a1.precedente.precedente);
        DemiArete.aretePendante(4,5,a1.precedente.precedente.precedente);
        DemiArete.joindreSommets(a1.suivante,
                        a1.precedente.precedente.precedente.precedente);

        DemiArete.print(s1);
        DemiArete.toutValide(s1);

        System.out.println("......Triangulation geometrique......");

        DemiArete.triangule(s1);
        DemiArete.print(s1);
        DemiArete.toutValide(s1);
    }
}
```

COMMENTAIRES de correction sur la COMPOSITION D'INFORMATIQUE 1 de INF431

Olivier Devillers

2008

Erreurs d'énoncés

- En dessous de la figure 1, il est écrit «4 de la classe `Face`» au lieu de «3 de la classe `Face`»
- Dans la classe `DemiArete` le champs pour la face incidente est appelé soit `f` soit `face` (certains l'on signalé, je ne pense pas que cela est perturbé quelqu'un).
- La méthode `aretePendante` est définie **void** mais utilisée comme retournant l'arête `ba` dans la question 1.7. Il a été indiqué en séance de prendre partout la deuxième version.
- On ne précisait pas le type à donner aux arêtes dans les différentes fonctions (1.1, 1.5, 1.6). Le seul endroit où j'en ai tenu un peu compte dans la correction c'est à la question 1.8 pour les deux «portes».

Remarques générales

La première partie a été traité au moins jusqu'à la question 1.8 par la plupart des élèves. La deuxième partie a été peu abordée et avec des résultats plutôt décevants.

Chaque question est notée sur 1 et après un grand tableau (sur la page web du cours) mélange tout ça avec des coefficients.

Je n'écris pas forcément sur les copies. Vous avez les notes question par question dans le tableau et vous avez la correction. Si c'est juste ou si c'est faux d'une manière standard (cf ci dessous) il est fort possible que rien ne soit écrit sur votre copie, ou un minimum.

Erreurs fréquentes et remarques du correcteur

- **Lire l'énoncé !**
  1.2 le carré de la distance, pas la distance.
  1.3 la taille de la face c'est son nombre d'arêtes, pas son périmètre.
  1.8 la figure 1, c'est la figure 1, pas une autre (et pas la figure de la question 1.1 qui n'était pas numérotée).
  2.1/2.2 la fonction `orientation` c'était une fonction supposée fournie. Il n'était pas demandé de l'écrire (et c'était explicitement précisé !).
- Lorsque la même erreur se répète plusieurs fois, elle peut être moins sanctionné par la suite car dans les questions suivantes la difficulté se situe ailleurs.
- Pour les boucles circulaires où on reviens à son point de départ (tourner autour d'un sommet) la boucle **do** `{...` while(..) ; est beaucoup plus sympa que les boucles **for** ou **while**. Bien entendu, c'est juste quand même, si vous n'oubliez pas de regarder le premier élément (et que vous n'écrivez pas une boucle où on ne rentre jamais).
- On n'est pas obligé d'initialiser une variable à sa déclaration. En tout cas

```
Sommet s = new Sommet();
s = u.origine;
```

  est à proscrire. On a appelle un constructeur pour rien puisque l'on oublie tout de suite l'objet créé.

– Question 1.1

```
DemiArete ab = new DemiArete();
DemiArete ba = new DemiArete(ab,....);
DemiArete ab = new DemiArete(ba,....);
```

À la 2ème ligne `ba` va stocker une référence au `ab` créé à la 1ère ligne. À la 3ème ligne on créé un **nouveau** `ab`. Bref ça marche pas du tout.
C'est pareil pour la variante

```
DemiArete ba = new DemiArete(new DemiArete(ba,....)  ,....);
```

Ne pas oublier les champs `arete` des `Sommets` et des `Faces`.
Cette question était longue et fastidieuse, l'objectif était de bien vous faire comprendre comment ça marche avant de passer à la suite. Pour ceux qui ont écrit un morceau de la fonction et on terminé avec un «pareil pour la suite» 1- ils n'ont pas tous les points 2- si il y a la moindre erreur dans ce qu'il faut considérer «pareil» ! c'est évidemment pas très bon.
– Question 1.2
x^2 n'est pas du java, il faut écrire `x*x`. `Math.pow(x,2)` est un peu lourd. Et sinon `sqrt` c'est la racine carrée et ici on demandait le carré de la distance.
Pour accéder à l'autre extrémité de l'arête on peut utiliser indifféremment `jumelle.origine` ou `suivante.origine`.
– Question 1.3
Si vous démarrez votre tour de la face en gd sur la Fig 1, quand vous retrouvez g vous êtes en ge et vous n'avez pas fait tout le tour. Donc faire la condition d'arrêt de la boucle sur les sommets était moins bien que sur les demi-arêtes.
– Question 1.4
Ici le test sur les sommets était maladroit, mais plus raisonnable qu'à la question précédente.
– Question 1.5
Les notations introduites dans l'énoncé ne sont pas connues automatiquement. Une déclaration du genre `Sommet a=u.origine` est nécessaire si on veut utiliser une variable `a`.
Ne pas oublier le `v.face.arete=v;` sinon le pointeur arête de l'ancienne face peut rester positionné sur une demi-arête de la nouvelle face.
Il a été indiqué en séance que cette fonction retournait une demi-arête. `ba` était ce qui était cohérent avec la question 1.7, mais si vous avez préférer returner `ab` il fallait essayer d'être cohérent à la question 1.8.
– Question 1.8
Pour accrocher une arête pendante, il ne suffit pas que le point de départ soit bon.

```
DemiArete ab = premiereArete(0,0,1,1);
DemiArete fa = aretePendante(1,-1,ab);
DemiArete.joindreSommets(ab.jumelle,fa); //créé triangle afb
DemiArete cb = aretePendante(x,y,ab.jumelle);
```

la dernière ligne crée bien une arête cb attachée au point b mais à l'intérieur du triangle afb, il faut utiliser

```
DemiArete cb = aretePendante(x,y,ab.suivante);
```

De même pour appliquer le deuxième `joindreSommet` il faut que les deux `DemiArete` en arguments soient dans la même face (pour le premier `joindreSommet` il n'y a qu'une face, donc on ne peut pas se tromper).
Sinon il était précisé au début de l'énoncé «Tous les graphes...sont supposés connexes», et par ailleurs le nom de la fonction `premiereArete` laisse bien penser qu'il faut l'utiliser pour la première arête (et pas les suivantes), enfin les spécifications de la fonction `joindreSommet` disait bien que ça devait couper une face en deux. Bref créer plusieurs «premières» arêtes et les joindre après était en contradiction avec tout ça et est une mauvaise réponse.
Ne pas oublier non plus de dire où sont les portes et les cloisons.
– Question 1.9
Un assez bête DFS sur lequel les résultats sont plutôt mauvais.

Il était clair que c'était les sommets qui devait servir à parcourir puisque l'hypothèse du marquage initial portait sur ces seuls sommets (et au passage on pouvait marquer les faces et demi-arêtes sans se préoccuper de savoir si on les marquait plusieurs fois ou pas).
Utiliser la fonction degré dans une boucle

```
for (int i=0, a=s.arete; i<s.degre(); i++, a=a.precedente.jumelle)
```

est maladroit car le degré est alors recalculé à chaque passage dans la boucle ce qui rend le truc quadratique dans le degré. Préférer :

```
d=s.degre; for(...; i<d; ...)
```

ou mieux :

```
a=s.arete; do {...; a=a.precedente.jumelle;} while(a!=s.arete);
```

et évidemment pas :

```
a=s.arete.precedente.jumelle; while(a!=s.arete) {...; a=a.precedente.jumelle;}
```

qui oublie de traiter `a.arete`.

– Question 2.1
Il y en a pour ne pas savoir résoudre 2 équations linéaires à deux inconnues. Vérifier sur un exemple : 1 triangle, 3 sommets, 3 arêtes et 2 triangles, 4 sommets, 5 arêtes (un quadrilatère et sa diagonale) pour être sur de pas se gourer pourrait être un bon réflexe.
Et la question était le nombre de triangles, pas celui de faces (décompter la face infinie).

– Question 2.2
Pour que 2 segments ab et cd se coupent, il faut que a et b soient de part et d'autre de la droite cd **et aussi** que c et d soient de part et d'autre de la droite ab.