École Polytechnique

Promotion 2010, Année 2011-2012

INF431, Contrôle Classant 1
25 avril 2012

Jean-Pierre Tillich

⋆ ⋆ ⋆

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

L'énoncé comprend 4 parties. Les parties 2, 3 et 4 sont largement indépendantes entre elles (à l'exception des deux dernières questions).

Enfin, on trouvera dans la section 5, en annexe, un rappel de quelques classes JAVA prédéfinies que l'on pourra utiliser librement.

Un code correcteur d'erreurs répond au problème suivant. On veut transmettre sur un canal un message binaire $x_1 \dots x_k$ d'une certaine longueur k. Malheureusement, certains des bits transmis peuvent être effacés durant la transmission : on obtient à la place de 0 ou de 1, le symbole 2 signifiant que le bit envoyé n'a pas été reçu. On veut néanmoins pouvoir récupérer après transmission l'intégralité du message $x_1 \dots x_k$.

Ce problème peut être résolu au moyen de codes correcteurs d'effacements. L'approche utilisée dans ce cas peut être résumée comme suit. Elle consiste en les étapes suivantes :

1. On transforme d'abord le message en question en un message plus long (qui sera donc redondant) par une application injective f qui va de $\{0,1\}^k$ dans $\{0,1\}^n$ avec $n > k$. Cette opération est appelée *l'opération de codage*.
2. Au lieu de transmettre $x_1 \dots x_k$, on émet $y_1 y_2 \dots y_n = f(x_1 x_2 \dots x_k)$.
3. Après transmission, on reçoit un mot $z_1 \dots z_n$ dans $\{0,1,2\}^n$ pour lequel soit $z_i = y_i$, soit $z_i = 2$. L'*algorithme de décodage* consiste à déterminer un mot binaire $x_1 \dots x_k$ qui est tel que $f(x_1 x_2 \dots x_k)$ coïncide avec $z_1 z_2 \dots z_n$ en les positions i pour lesquelles $z_i \neq 2$.

On appelle l'image de f, c'est-à-dire $\{f(x_1, \dots, x_k) | (x_1, \dots, x_k) \in \{0,1\}^k\}$, le *code*. De manière générale, un code *binaire* de *longueur* n et permettant de coder des mots binaires de longueur k est un sous-ensemble de $\{0,1\}^n$ de taille 2^k.

Nous allons étudier un code particulier qui est associé à l'ensemble des cycles d'un graphe. Ce type de code fait partie de la famille des codes LDPC (Low Density Parity Check) qui figurent dans la plupart des normes de codage actuelles.

1 Le code des cycles d'un graphe

On travaille sur des graphes *non orientés*, sans boucles ni arêtes parallèles. On suppose dans la suite qu'un tel graphe est défini au moyen des trois classes `Sommet`, `Arete` et `Graphe`. La classe `Graphe` permet de représenter le graphe au moyen de deux champs `sommets` et `aretes`. Le tableau `sommets` donne la liste des sommets du graphe, le tableau `aretes` donne la liste des arêtes.

```
class Graphe{
    Sommet[] sommets;
    Arete[] aretes;
   //...
}
```

On définit le numéro `i` d'une arête `E` d'un graphe `G` par son indice dans le tableau `G.aretes`. En d'autres termes `E=G.aretes[i]`. De même, le numéro d'un sommet est donné par son indice dans le tableau `sommets`. La classe `Sommet` permet de représenter les sommets du graphe. Elle comprend un champ `voisins` qui est un tableau donnant les numéros des *arêtes* incidentes au sommet auquel il se rapporte. Ainsi, dans l'exemple de la figure 1, le champ `voisins` du sommet central comprend les entrées $6, 7, 8, 9, 10, 11$.

```
class Sommet{
    int[] voisins;
    //...
    }
```

La classe `Arete` représente les arêtes du graphe au moyen de deux champs `g` et `d` de type **int** qui sont les numéros des sommets extrémités de l'arête.

```
class Arete{
    int g;
    int d;
    // ...
}
```

Question 1. Écrire une méthode **int** `degre(int s)` de la classe `Graphe` qui renvoie le degré du sommet de numéro `s`. Écrire une méthode **int** `voisin(int s, int i)` de la classe `Graphe` qui renvoie le numéro du `i`-ème voisin du sommet numéroté `s` (c'est-à-dire le numéro de l'extrémité différente de `s` appartenant à l'arête de numéro `sommet[s].voisins[i]`).

Réponse :

```
int degre(int s){
     return sommets[s].voisins.length;
 }
int voisin(int s, int i){
     int arete=sommets[s].voisins[i];
     int a=aretes[arete].g;
     int b=aretes[arete].d;
     if (a==s) return b;
     else return a;
 }
```

□

À un graphe donné, on associe le code des cycles avec les caractéristiques suivantes.

1. Notons n le nombre d'arêtes du graphe. Ce nombre n coïncide avec la longueur des mots du code des cycles. Les arêtes du graphe sont donc numérotées $0, 1, \ldots, n-1$. Pour alléger les explications qui suivent, on identifiera une arête avec son numéro. L'ensemble des arêtes associé à un mot binaire $\mathbf{y} = (y_i)_{0 \leq i \leq n-1}$ de $\{0,1\}^n$ est défini comme l'ensemble des arêtes i qui sont telles que $y_i = 1$.
2. Un sous-ensemble E d'arêtes d'un graphe est appelé *pair* si et seulement si le sous-graphe induit par ces arêtes n'a que des sommets de degré pair (voir l'exemple de la figure 1).

3. Le code des cycles du graphe est l'ensemble des mots binaires $\mathbf{y} = (y_i)_{0 \leq i \leq n-1}$ de $\{0,1\}^n$ qui sont tels que l'ensemble des arêtes associé à $\mathbf{y}$ est pair.

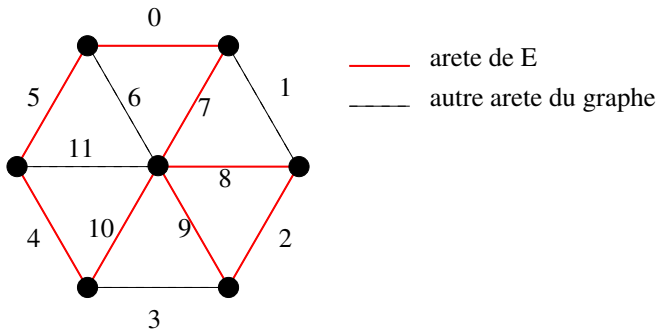

FIGURE 1 – Un exemple de sous-ensemble d'arêtes pair E. La numérotation des arêtes est précisée sur la figure et E correspond au mot binaire $\mathbf{y} = (101011011110)$. Comme $y_0 = y_2 = y_4 = y_5 = y_7 = y_8 = y_9 = y_{10} = 1$ (et les autres y_i sont égaux à 0) E contient exactement les arêtes numérotées 0,2,4,5,7,8,9 et 10.

On représentera dans tout ce qui suit, un mot binaire de longueur n (c'est-à-dire un mot dans $\{0,1\}^n$) et plus généralement un mot de $\{0,1,2\}^n$ par un tableau de type **int[]** de longueur `n`.

Question 2. Écrire une méthode **boolean** `estMotDeCode(int[] mot)` de la classe `Graphe` qui prend en entrée un mot binaire `mot` de longueur égale au nombre d'arêtes du graphe et qui est telle que `G.estMotDeCode(mot)` renvoie **true** si et seulement si `mot` appartient au code des cycles de `G`. On attend ici une solution de complexité linéaire en le nombre d'arêtes du graphe.

Réponse :

```
boolean estMotDeCode (int[] mot)
{
    for (Sommet sommet : sommets) {
        int parite = 0;
        for (int arete : sommet.voisins)
            parite += mot[arete];
        if (parite % 2 == 1)
            return false;
    }
    return true;
}
```

□

Question 3. Montrer que tout sous-ensemble pair d'arêtes se décompose comme un ensemble de cycles simples n'ayant deux à deux aucune arête commune. (Par exemple, l'ensemble E de la figure 1 est l'union de deux cycles simples n'ayant pas d'arête commune.) On rappelle qu'un cycle simple est un ensemble d'arêtes du graphe formant un graphe connexe de degré 2 (il correspond donc à un chemin qui revient à son origine sans emprunter plusieurs fois la même arête ou le même sommet).

Réponse : On remarque d'abord le fait suivant :

Fait 1 *Un graphe où tous les sommets sont au moins de degré* 2 *contient un cycle simple.*

En effet, on démarre un chemin partant d'un sommet quelconque du graphe et en empruntant à chaque fois une nouvelle arête (c'est possible car le degré de chaque sommet est au moins 2) jusqu'à rencontrer un sommet par lequel on est déjà passé. Le sous-chemin démarrant à la

première fois que l'on a rencontré ce sommet et conduisant de nouveau jusqu'à ce sommet forme un cycle simple.

Un ensemble pair vérifie bien évidemment cette hypothèse et contient donc un cycle simple. Si l'on enlève les arêtes du cycle à l'ensemble pair, cet ensemble reste encore pair. Cela permet de montrer immédiatement la question par récurrence sur le nombre d'arêtes du cycle. □

On supposera à partir de maintenant que le graphe considéré est connexe et contient n arêtes et m sommets.

2 L'algorithme de codage

Nous allons voir dans cette section comment s'effectue l'opération de codage, c'est-à-dire comment on associe de manière injective à un mot binaire d'une certaine longueur k que nous allons préciser un mot binaire du code des cycles d'un graphe G.

Question 4. Combien d'arêtes a un arbre couvrant du graphe ?

Réponse : Un arbre couvrant a $m-1$ arêtes. □

Question 5. Indiquez quelques algorithmes permettant de calculer un arbre couvrant d'un graphe en précisant leur complexité.

Réponse : Ici on parle de graphes non valués, donc des parcours comme DFS ou BFS conviennent (complexité $O(n+m)$). □

Question 6. Soit T un arbre couvrant de G. Montrer que toute arête n'appartenant pas à T peut être complétée par certaines arêtes appartenant à T de manière à former un ensemble pair. *Indication : utiliser le fait qu'un cycle simple est un ensemble pair.*

Réponse : Le graphe composé de cette arête (appelons la E) et des arêtes de T est connexe (puisque T l'est), contient m sommets et m arêtes. Ce n'est donc pas un arbre. Il contient donc un cycle simple. Ce dernier passe par E car T ne contient pas de cycles puisque c'est un arbre. Un tel cycle simple forme un ensemble pair. □

On suppose pour la suite qu'un objet de la classe `Graphe` contient également un champ `arbreCouvrant` représentant un arbre couvrant donné sous la forme d'un tableau d'objets de type `Sommet`. Ce tableau `arbreCouvrant` a la même longueur que `sommets`, c'est-à-dire `m`. Ce qui les différencie est le fait que l'ensemble des arêtes incidentes à `arbreCouvrant[i]` est un sous-ensemble des arêtes incidentes à `sommets[i]`.

```
class Graphe{
    Sommet[] sommets;
    Arete[] aretes;
    Sommet[] arbreCouvrant;
   //...
}
```

Les sommets et les arêtes de l'arbre couvrant d'un graphe `G` sont numérotés comme ceux de `G`, et le champ `voisins` de chaque élément `arbreCouvrant[i]` contient la liste des numéros d'arêtes de `G` qui appartiennent à l'arbre couvrant et qui sont incidents au sommet `i` du graphe.

Question 7. Décrire brièvement puis écrire la méthode correspondant à la question 6, **int[]** `codage(int i)` pour laquelle on fait l'hypothèse que l'arête numérotée `i` n'appartient pas à `arbreCouvrant` et qui est telle que `G.codage(i)` renvoie un mot binaire de longueur n pour lequel
– l'ensemble d'arêtes E associé est pair ;

– E contient l'arête numérotée `i` et un certain nombre d'arêtes de l'arbre couvrant `G.arbreCouvrant`.
On s'attachera ici à donner un algorithme de complexité linéaire dans le nombre de sommets du graphe.

Réponse : On effectue un parcours de l'arbre couvrant pour trouver le chemin (dans l'arbre) joignant les deux extrémités de l'arête i.

```
int extremite(int arete, int origine){
    int a=aretes[arete].g;
    int b=aretes[arete].d;
    if (a==origine) return b;
    else return a;
}

int[] codage(int i){
    int[] resultat = new int[aretes.length];
    resultat[i] = 1;
    explore(aretes[i].g,aretes[i].d,-1,resultat);
    return resultat;
}
boolean explore(int depart,int but, int areteInterdite, int[] codage){
    Sommet s = arbreCouvrant[depart];
    for (int arete : s.voisins){
        if (arete!=areteInterdite){
            int suite=extremite(arete,depart);
            if (suite==but||explore(suite,but,arete,codage)){
                codage[arete]=1;
                return true;
            }
        }
    }
    return false;
}
```

□

Question 8. Soit E et F deux sous-ensembles d'arêtes de G. On note $E \oplus F$ la différence symétrique de E et de F, c'est-à-dire l'ensemble des arêtes qui appartiennent à E ou à F mais qui n'appartiennent pas à l'intersection $E \cap F$. Que peut-on dire de $E \oplus F$ si E et F sont tous deux pairs ?

Réponse : Soit s un sommet du graphe. Notons E_s et F_s l'ensemble des arêtes incidentes à s appartenant à E et F respectivement. Remarquons que l'ensemble des arêtes incidentes à s appartenant à $E \oplus F$ est égal à la différence symétrique $E_s \oplus F_s$. Ce dernier ensemble est de cardinalité paire puisque E_s et F_s sont tous deux de cardinalité paire. Il en résulte que $E \oplus F$ est pair. □

Question 9. Soit T un arbre couvrant de G. Montrer que tout sous-ensemble d'arêtes n'appartenant pas à T peut s'étendre en rajoutant éventuellement certaines arêtes appartenant à T de manière à former un ensemble pair. *Indication : utiliser les questions 6 et 8.*

Réponse : C'est une utilisation directe des deux questions précédentes : l'ensemble d'arêtes F cherché peut s'écrire $\oplus_{i \in E} F_i$ où E est le sous-ensemble d'arêtes n'appartenant pas à T considéré et F_i est le sous-ensemble pair d'arêtes ne contenant que l'arête i et un certain nombre d'arêtes de T obtenu par le procédé de la question 7. □

Question 10. On admet dans cette question que les $n - m + 1$ arêtes qui ne sont pas dans l'arbre couvrant sont numérotées $0, 1, \ldots, n - m$. Décrire brièvement un procédé de codage (on rappelle que ce dernier doit être injectif) qui à tout mot binaire de longueur $n - m + 1$ associe un ensemble d'arêtes de G pair en utilisant les questions 7 et 9. Écrire la méthode correspondante **int**[] `codage(int mot[])` qui est telle que `G.codage(mot)` renvoie un mot binaire de longueur n (où n est le nombre d'arêtes de `G`), pour lequel l'ensemble d'arêtes associé est pair. Quelle est la complexité de votre procédé de codage ?
NB : si vous bloquez sur cette question, il vous est conseillé de passer directement aux questions suivantes, elle n'est en effet pas utilisée dans la suite.

Réponse : À un mot $\mathbf{x} \in \{0,1\}^{n-m+1}$ on associe l'ensemble E d'arêtes n'appartenant pas à T qui est défini comme l'ensemble des arêtes dont le numéro i vérifie $x_i = 1$. Cette ensemble d'arêtes peut être complété en un ensemble pair F d'après la question 9. Le mot binaire $\mathbf{y}$ associé sera le codage de $\mathbf{x}$. Comme F contient E par définition, le sous-mot de $\mathbf{y}$ obtenu en prenant ses $n - m + 1$ premières positions est égal à $\mathbf{x}$. Le codage est donc bien injectif. Par ailleurs, rappelons que l'ensemble F est donné par $F = \oplus_{i \in E} F_i$ où F_i est le sous-ensemble pair d'arêtes ne contenant que l'arête i et un certain nombre d'arêtes de T. $\mathbf{y}$ est donc obtenu par l'addition modulo 2 des mots binaires `codage(i)` où i appartient à E. Ceci conduit donc au procédé de codage suivant.

```
void arrayXor (int[] left, int[] right)
{
  for (int i = 0; i < left.length; i++)
    left[i] ^= right[i];
}

  int[] codage(int[] mot)
{
  int[] resultat = new int[aretes.length];
  for (int i = 0; i < mot.length; i++)
    if (mot[i] == 1)
      arrayXor(result, codage(i));
  return resultat;
}
```

Le codage est de complexité $0(n^2)$. □

3 Détection de petits cycles

On se propose dans cette partie de calculer la taille du plus petit cycle simple du graphe.

Dans le cas de la figure 1 par exemple, la taille du plus petit cycle simple vaut 3.

Question 11. Écrire une méthode **int**[] `distance(int s)` qui renvoie un tableau des distances du sommet `s` à tous les sommets du graphe.

Réponse : On effectue une BFS depuis s. □

Soit `s` un sommet. On appelle « cycle sans demi-tour partant de `s` », un chemin non-vide, démarrant en `s` et revenant en `s` qui n'emprunte pas de manière consécutive deux fois la même arête (le chemin correspondant x_0 =`s`, x_1,...,x_ℓ =`s` est donc tel que $x_{i-1}x_i$ et x_ix_{i+1} sont deux arêtes distinctes pour tout i dans $\{1, \ell - 1\}$).

Question 12. Soit C un plus petit cycle sans demi-tour partant de `s`. Que peut-on dire des sommets les plus éloignés de `s` dans C lorsque C contient un nombre impair de sommets ? Et un nombre pair ? On ne demande pas de démonstration détaillée.

Réponse : Ils sont à la fois voisins et à une même distance d de `s` dans le second cas. Dans le premier, x_i, x_{i+1} et x_{i+2} sont à des distances respectives de d, $d+1$ et d de `s`. □

Question 13. Décrire brièvement, puis écrire, une méthode **int** `taillePlusPetitCycleDepuis(int s)` qui renvoie la taille du plus petit cycle sans demi-tour partant du sommet de numéro `s`. On pourra rendre `Integer.MAX_VALUE` lorsqu'il n'y a pas de tel cycle.

Réponse : II suffit de détecter les deux configurations de la question précédente pour trouver les points extrémaux d'un cycle sans demi-tour. Par soucis d'exhaustivité, on donne la justification complète ici :

Il reste à voir que l'on obtient bien la plus petite taille d'un cycle sans demi-tour. Considérons un cycle sans demi-tour de taille minimale passant par `s`. Deux cas sont possibles :

(i) Il est de taille paire $2e$. On considère l'unique sommet antipodal `t` du cycle (c'est le sommet auquel on arrive après avoir emprunté les e premières arêtes du cycle). Soit d la distance de `s` à `t`. On a $d \leq e$, puisque e est une borne supérieure sur la distance entre `s` et `t`. Il n'est pas possible que $d < e$, sinon on pourrait construire un cycle sans demi-tour passant par `s` de taille inférieure à $2e$. On a donc $d = e$ et il y a nécessairement deux sommets `u` et `v` sur le cycle qui sont à la fois voisins de `t` et à distance $d-1$ de `s`. Cette configuration est considérée dans la méthode `taillePlusPetitCyclePassantPar(int s)`.

(ii) Il est de taille impaire, disons $2e+1$. On considère les deux sommets antipodaux `t` et `u` du cycle, `t` est le sommet auquel on arrive après avoir emprunté les e premières arêtes du cycle, `u` est le sommet qui permet d'aboutir à `s` après avoir emprunté les e dernières arêtes du cycle. On montre de la même manière que la distance de `t` à `s` et la distance de `t` à `u` sont toutes les deux égales à e. La configuration correspondante est donc considérée par la méthode `taillePlusPetitCyclePassantPar(int s)`.

```
int taillePlusPetitCyclePassantPar(int s){
     int taille=Integer.MAX_VALUE;
    int n0,n1,d;
     int[] distances=distance(s);
     for (int x=0;x<sommets.length;x++){
         n0=0;
         n1=0;
         for (int j=0;j<degre(x);j++){
             if (distances[x]>distances[voisin(x,j)])
                 n1++;
             else if (distances[x]==distances[voisin(x,j)]) n0++;
         }
         if (n1>1)
             {if (taille>2*distances[x]) taille=2*distances[x];}
         else if (n0>=1)
             if (taille>2*distances[x]+1) taille=2*distances[x]+1;
     }
     return taille;
}
```

□

Question 14. Décrire brièvement puis écrire une méthode **int** `taillePlusPetitCycle()` de la classe `Graphe` qui renvoie la taille du plus petit cycle simple du graphe et préciser sa complexité. *Indication : remarquer que le plus petit cycle sans demi-tour dans le graphe est un cycle simple.*

Réponse : Le plus petit cycle sans demi-tour est nécessairement un cycle simple, la distance minimale est donc obtenue en prenant le minimum sur tous les sommets `i` de `taillePlusPetitCyclePassantPar(i)`.

```
int taillePlusPetitCycle(){
    int d=Integer.MAX_VALUE;
    for (int i=0;i<sommets.length;i++){
        d = (int) Math.min(taillePlusPetitCyclePassantPar(i),d);
    }
    return d;
}
```

La méthode `taillePlusPetitCyclePassantPar` a pour complexité $O(n)$, par conséquent la complexité de `distanceMinimale()` est en $O(nm)$. □

4 L'algorithme de décodage

On admet à partir de maintenant que :

1. l'émetteur et le récepteur se sont mis d'accord sur un graphe G donné,
2. l'émetteur envoie un mot binaire $\mathbf{y}$ du code des cycles de G,
3. le récepteur reçoit un mot $\mathbf{z}$ de longueur n appartenant à $\{0,1,2\}^n$ pour lequel soit $z_i = y_i$, soit $z_i = 2$ (ce qui correspond à un effacement dans la séquence $\mathbf{y}$),
4. le récepteur choisit un mot du code des cycles de G qui coïncide avec $\mathbf{z}$ en les positions pour lesquelles $z_i \neq 2$. Si il y a plusieurs possibilités, cette dernière est choisie de manière arbitraire.

Pour formaliser cet algorithme de décodage on introduit la notion suivante. Soit $\mathbf{z} \in \{0,1,2\}^n$. On dit que i est une position *effacée* de $\mathbf{z}$, si et seulement si $z_i = 2$. On dit qu'un mot $\mathbf{z} \in \{0,1,2\}^n$ est *valide* si et seulement si il existe un mot binaire $\mathbf{y}$ du code des cycles du graphe qui est tel que $y_i = z_i$ pour toutes les positions i qui ne sont pas effacées. Ainsi le récepteur reçoit toujours un mot valide à l'issue d'une transmission d'un mot du code des cycles du graphe. On dit qu'un sommet j est *intéressant* pour un mot $\mathbf{z} \in \{0,1,2\}^n$ si et seulement si une arête i incidente à j, et une seulement, est telle que $z_i = 2$ (et correspond donc à une position effacée).

Question 15. Écrire une méthode **void** `corrigeUnEffacement(int i, int s, int[] z)` de la classe `Graphe` qui

1. attend en entrée un mot valide `z` dans $\{0,1,2\}^n$ pour lequel `i` est une position effacée et `s` est une des extrémités de `i` qui est intéressante pour `z` ;
2. modifie la valeur de `z` en `i` pour qu'elle devienne binaire et que `z` reste valide.

Réponse :

```
void corrigeUnEffacement(int i, int s, int[] z){
    for (int j=0;j<degre(s);j++)
        z[i] += z[sommets[s].voisins[j]];
    z[i] = z[i]%2;
}
```

À la fin de la méthode, `z[i]` est égal à deux fois la valeur initiale de `z[i]` plus la somme sur les autres arêtes `j` incidentes à `s` des `z[j]`, le tout modulo 2. Par conséquent, `z[i]` est égal à la somme modulo 2 de ces `z[j]`. C'est clairement la seule valeur qui remplit le cahier des charges de la méthode. □

Question 16. Écrire une méthode **void** `corrige(int [] z)` de la classe `Graphe` qui prend en entrée un mot `z` dans $\{0,1,2\}^n$ qui est valide, qui ne le modifie que dans les positions `i` effacées de telle manière qu'en sortie il reste valide et qu'il ne reste aucun sommet intéressant pour `z`. On essaiera ici de donner une implémentation qui soit de complexité linéaire vis à vis du nombre d'arêtes du graphe.

Réponse :

```
void metDansLaPileSiInteressant(int s, int[] z, Stack<Couple> p){
    int compteur=0;
    int arete=-1;
    for (int i=0;i<degre(s);i++){
        if (z[sommets[s].voisins[i]]==2) {
            compteur++;
            arete=sommets[s].voisins[i];
        }
    }
    if (compteur==1) p.push(new Couple(arete,s));
}
class Couple{
    int a;
    int s;
    Couple (int va, int vs){a=va;s=vs;}
}
void corrige(int z[]){
    int res;
    Stack<Couple> pile=new Stack<Couple>();
    for (int i=0;i<sommets.length;i++)
        metDansLaPileSiInteressant(i,z,pile);
    while (!pile.empty()){
        Couple c = pile.pop();
        corrigeUnEffacement(c.a,c.s,z);
        metDansLaPileSiInteressant(aretes[c.a].g,z,pile);
        metDansLaPileSiInteressant(aretes[c.a].d,z,pile);
    }
}
```

La complexité de `corrige(int z[])` est bien en $O(n)$, puisque
(i) la boucle sur le nombre de sommets est de complexité $O(m)$ qui est bien de la forme $O(n)$ car $m \leq n+1$, le graphe étant connexe ;
(ii) la boucle sur la pile ne fait au plus que $O(m)$ itérations, chaque sommet ne pouvant être intéressant qu'une fois. □

Question 17. Soit E un ensemble d'arêtes associé à un ensemble de positions effacées dans un mot valide $\mathbf{z} \in \{0,1,2\}^n$ ayant au moins une position effacée et pour lequel aucun des sommets n'est intéressant. Montrer que E contient un cycle.

Réponse : L'ensemble des arêtes correspondant à ces positions forme un sous-graphe dans lequel chaque sommet est de degré au moins 2 (sinon il contiendrait un sommet qui serait intéressant). On utilise le fait 1 pour conclure. □

Question 18. Soit un mot `z` valide qui est tel que l'ensemble des arêtes correspondant aux positions effacées ne contient pas de cycle. Montrer que `G.corrige(z)` transforme `z` en un mot binaire qui appartient au code des cycles de `G`. Montrer qu'il n'y a aucun autre mot du code du code des cycles qui coïncide avec `z` sur les positions non effacées.

Réponse : La méthode `corrige` termine avec un mot $\mathbf{z}'$ valide et qui ne contient plus aucun sommet intéressant. $\mathbf{z}'$ ne peut plus contenir de positions effacées, sinon on aurait une contradiction avec la question précédente. Par conséquent $\mathbf{z}'$ est binaire. Le point crucial est qu'à chaque fois que l'on corrige un effacement avec la méthode `corrigeUnEffacement`, la nouvelle valeur prise par `z` en `i` est la seule valeur compatible avec un mot du code des cycles qui coïncide avec `z` sur les positions non effacées. Par conséquent, puisque par hypothèse $\mathbf{z}$ coïncidait au

départ avec un mot $\mathbf{y}$ du code des cycles sur les positions non effacées, $\mathbf{z}'$ est égal à ce mot $\mathbf{y}$ et $\mathbf{y}$ est le seul mot du code des cycles ayant cette propriété. □

Question 19. Voyez-vous l'intérêt d'avoir calculé la longueur du plus court cycle simple de G ? (partie 3)

Réponse : Soit d la longueur du plus court simple de G. S'il y a moins de d bits effacés, les positions effacées ne peuvent former de cycle (tout cycle contient au moins un cycle simple). On peut donc toujours corriger au moins d effacements. □

Question 20. Les questions précédentes vous suggèrent-elles une méthode plus efficace pour effectuer le codage tel qu'il est décrit en partie 2 ?

Réponse : La méthode `corrige` termine avec un mot $\mathbf{z}'$ valide et qui ne contient plus aucun sommet intéressant. $\mathbf{z}'$ ne peut plus contenir de positions effacées, sinon on aurait une contradiction avec la question précédente. Par conséquent $\mathbf{z}'$ est binaire. Le point crucial est qu'à chaque fois que l'on corrige un effacement avec la méthode `corrigeUnEffacement`, la nouvelle valeur prise par `z` en `i` est la seule valeur compatible avec un mot du code des cycles qui coïncide avec `z` sur les positions non effacées. Par conséquent, puisque par hypothèse $\mathbf{z}$ coïncidait au départ avec un mot $\mathbf{y}$ du code des cycles sur les positions non effacées, $\mathbf{z}'$ est égal à ce mot $\mathbf{y}$ et $\mathbf{y}$ est le seul mot du code des cycles ayant cette propriété.

On utilise ce que l'on vient de démontrer de la manière suivante. On choisit un arbre couvrant T du graphe. On suppose que les arêtes qui ne sont pas dans T sont numérotées $0, 1, \ldots, n-m$. On code un mot binaire $\mathbf{x}$ dans $\{0,1\}^{n-m+1}$ en construisant le mot $\mathbf{z} \in \{0,1,2\}^n$ de la manière suivante :

$$\begin{aligned} z_i &= x_i \text{ pour } i \in \{0, \ldots, n-m\} \\ z_i &= 2 \text{ sinon.} \end{aligned}$$

D'après la question 6, $\mathbf{z}$ est un mot valide. Appliquer la méthode `corrige` conduit donc à un mot du code des cycles. Sa complexité est cette fois-ci en $O(n)$, ce qui améliore la complexité de la méthode de codage vue précédemment. □

5 Annexe

On rappelle ici quelques classes prédéfinies dont on pourra faire librement usage dans le contrôle.

5.1 `Integer`

Cette classe permet de manipuler un entier comme une variable de type objet et dispose entre autres des méthodes suivantes :
- `Integer(int i)` : le constructeur renvoyant un objet de type `Integer` représentant `i` ;
- `intValue()` : renvoie la valeur de l'entier représenté par l'objet de type `Integer` auquel on applique la méthode.

Elle dispose aussi des champs **static int** `MAX_VALUE` et **static int** `MIN_VALUE` représentant respectivement la plus grande et la plus petite valeur possible d'une variable de type **int**.

5.2 `LinkedList<E>`

La classe générique prédéfinie `LinkedList<E>` implémente l'interface `Collection<E>` et possède entre autres les méthodes suivantes :
- `LinkedList<E>()` : le constructeur qui renvoie une nouvelle liste vide ;

– **void** `add(E e)` : ajoute `e` en queue de liste ;
– `E removeFirst()` : supprime le premier élément de la liste et le renvoie ;
– `isEmpty()` : renvoie vrai si et seulement si la liste est vide.

5.3 `Stack<E>`

La classe générique prédéfinie `Stack<E>` est une autre implémentation de l'interface `Collection<E>` qui possède entre autres les méthodes suivantes :
– `Stack<E>()` : le constructeur qui renvoie une nouvelle pile vide ;
– **void** `push(E e)` : ajoute `e` en haut de la pile ;
– `E pop()` : supprime l'élément du haut de la pile et le renvoie ;
– `empty()` : renvoie vrai si et seulement si la pile est vide.