I.U.T. de Nice Côte d'Azur
Département Informatique
DUTINFO S4 – IT

2007-2008

Informatique théorique

TD n° 7.2

Décidabilité et complexité

1. Eléments de base

1.1. **Notion de l'algorithme** :

- Informellement, un algorithme consiste en un ensemble fini d'instructions simples et précises qui sont décrites avec un nombre limité de symboles. Un algorithme doit toujours produire le résultat en un nombre fini d'étapes et peut en principe être suivi par un humain avec seulement du papier et un crayon. L'exécution d'un algorithme ne requiert pas d'intelligence de l'humain sauf celle qui est nécessaire pour comprendre et exécuter les instructions.
- Formellement, on peut en revanche *définir* formellement les algorithmes comme les procédés qui peuvent être accomplis par une machine de Turing universelle.

1.2. **Notion de décidabilité et complétude** :

- Le problème d'*équations diophantiennes* posé la première fois par David Hilbert, au Congrès International des mathématiciens qui avait eu lieu à Paris en 1900 :
  on appelle équation diophantienne une équation du genre P(x, y, z, ...) = 0, où P est un polynôme à coefficients entiers. Résoudre une telle équation, c'est chercher les solutions sous forme d'entiers.
  - L'équation $x^2 + y^2 – 1 = 0$ est une équation diophantienne qui admet pour solutions: x = 1, y = 0 et x = 0, y = 1
  - L'équation $x^2 – 991y^2 – 1 = 0$ est également diophantienne mais a des solutions beaucoup plus difficiles à trouver (cf J. P. Delahaye, in "Logique, informatique et paradoxes" ed. Belin), la plus petite est :
    x = 379 516 400 906 811 930 638 014 896 080 et
    y = 12 055 735 790 331 359 447 442 538 767
  - Hilbert posait la question : "existe-t-il un procédé mécanique permettant de dire, après un nombre fini d'étapes, si une équation diophantienne donnée possède des solutions entières?".
  - On doit noter que, parmi ces équations, figurent certaines, qui sont associées à un problème historiquement bien connu: *le problème de Fermat*. Fermat pensait en effet avoir démontré que toute équation de la forme : $x^p + y^p = z^p$ est sans solution pour p > 2. (Pour p = 2, bien sûr... elle en a au moins une, cf: x = 3, y = 4, z = 5). De fait, on ne disposait pas, jusqu'à une date récente (Andrew Whiles, 1993) d'une telle démonstration.
- Formellement, deux types de décidabilité sont définis :
  - *Décidabilité logique* : Une proposition (on dit aussi énoncé) est dite **décidable** dans une théorie axiomatique, si on peut la démontrer ou démontrer sa négation dans le cadre de cette théorie. Un énoncé mathématique est donc indécidable dans une théorie s'il est impossible de le déduire, ou de déduire sa négation, à partir des axiomes.
    - ***Complétude*** : Une théorie mathématique pour laquelle tout énoncé est décidable est dite complète, sinon elle est dite *incomplète*
    - *Théorème d'incomplétude1 (Gödel 1932)* : Dans n'importe quelle théorie récursivement axiomatisable, cohérente et capable de « formaliser l'arithmétique », on peut construire un énoncé arithmétique qui ne peut être ni prouvé ni réfuté dans cette théorie. De tels énoncés sont dits indécidables dans cette théorie. On dit également il est *indépendant* de la théorie.
    - *Théorème d'incomplétude1 (Gödel 1932)* : Si T est une théorie cohérente qui satisfait des hypothèses analogues, la cohérence de T, qui peut s'exprimer dans la théorie T, n'est pas démontrable dans T. Ce théorème d'incomplétude nous garantit que toute théorie axiomatique cohérente, et suffisamment puissante pour représenter l'arithmétique de

Peano (l'arithmétique usuelle), est incomplète, pourvu qu'elle soit axiomatisée de façon que l'on puisse décider au sens algorithmique si un énoncé est ou non un axiome.

- *Décidabilité algorithmique* : Un problème de décision est dit décidable s'il existe un *algorithme*, une *procédure mécanique* qui termine en un nombre fini d'étapes, qui le décide, c'est-à-dire qui réponde par **oui** ou par **non** à la question posée par le problème. S'il n'existe pas de tels algorithmes, le problème est dit indécidable.
  - Le problème décidable algorithmique réfère à la notion de *calculabilité* en cherchant un algorithme qui s'arrête à un temps fini et fournit la réponse OUI ou NON à la question posée. Autrement dit, le problème P est décidable s'il existe un algorithme qui pour chaque x dit "OUI" ou "NON" à la question : "Est-ce que P(x) est vrai ?"
  - Dire qu'un problème est indécidable ne veut pas dire que les questions posées sont insolubles mais seulement qu'il n'existe pas de méthode unique et bien définie, applicable d'une façon mécanique, pour répondre à toutes les questions, en nombre infini, rassemblées dans un même problème.
- Théorème de Rice : Toute propriété non triviale des langages récursivement énumérables est indécidable.
  - Ce théorème dit que toute propriété non triviale (c.-à-d. qui n'est pas toujours vraie ou toujours fausse) sur la sémantique dénotationnelle d'un langage de programmation Turing-complet est **indécidable**. Il s'agit d'une généralisation du problème de l'arrêt.
  - **Exemple 1 :**
    Par exemple, un ramasse-miettes (logiciel qui libère la mémoire inutilisée) optimal et sûr est impossible. En effet, savoir si l'on doit libérer ou non la mémoire dépend du comportement futur du système, qu'il est impossible de prévoir. Ceci explique que toutes les techniques de ramasse-miettes soient conservatrices, et puissent garder en mémoire des blocs inutiles, permettant ainsi des fuites de mémoire.
  - **Exemple 2 :**
    Ainsi, dans le programme suivant en langage C (supposons que `p` n'intervienne pas ailleurs), déterminer si l'on doit libérer le bloc mémoire sitôt après son allocation est équivalent à décider le problème de l'arrêt sur `f()`.
    ```
    char *p = malloc(1000);
    f();
    *p = 1;
    ```

1.3. **Notion de complexité** :

- **Problème** : La théorie de la **complexité algorithmique** s'intéresse à l'estimation de l'efficacité des algorithmes. Elle s'attache à la question : *entre différents algorithmes réalisant une même tâche, quel est le plus rapide et dans quelles conditions ?*

  - La **théorie de la complexité** s'intéresse à l'étude formelle de la *difficulté* des problèmes en informatique.

  - La théorie de la complexité se concentre donc sur les problèmes qui peuvent effectivement être résolus, la question étant de savoir s'ils peuvent être résolus efficacement ou pas en se basant sur une estimation (théorique) ***des temps de calcul*** et des ***besoins en mémoire*** informatique.

  - Pour qu'une analyse ne dépende pas de la vitesse d'exécution de la machine ni de la qualité du code produit par le compilateur, il faut utiliser comme unité de comparaison des « *opérations élémentaires* » en fonction de la taille des données en entrée (exemple : accès à une cellule mémoire comparaison de valeurs opérations arithmétiques (sur valeurs à codage de taille fixe) opérations sur des pointeurs.

- **Notion de machine déterministe et indéterministe** :

  - Les machines de Turing déterministes font toujours un seul calcul à la fois. Ce calcul est constitué d'étapes élémentaires; à chacune de ces étapes, pour un état donné de la mémoire de la machine, l'action élémentaire effectuée sera toujours la même.

  - Une *machine de Turing non-déterministe* est une variante purement théorique des machines de Turing: on ne peut pas construire de telle machine. À chaque étape de son calcul, cette machine peut effectuer un **choix non-déterministe**: elle a le choix entre plusieurs actions, et elle en effectue une. Si l'un des choix l'amène à accepter l'entrée, on considère qu'elle a fait ce choix-là. Une autre manière de voir leur fonctionnement est de considérer qu'à chaque choix non-

déterministe, elles se dédoublent, les clones poursuivent le calcul en parallèle suivant les branches du choix. Si l'un des clones accepte l'entrée, on dit que la machine accepte l'entrée.

- **Complexité en temps :** La complexité d'un algorithme se mesure essentiellement en calculant le *nombre d'opérations élémentaires* pour traiter une donnée de t*aille n*. Les opérations élémentaires considérées sont
  - le nombre de comparaisons (algorithmes de recherche)
  - le nombre d'affectations (algorithmes de tris)
  - Le nombre d'opérations (+,*) réalisées par l'algorithme (calculs sur les matrices ou les polynômes).
  - Pour les machines déterministes, on définit la classe **TIME(t(n))** des problèmes qui peuvent être résolus en temps **t(n)**. C'est-à-dire pour lesquels il existe au moins un algorithme sur machine déterministe résolvant le problème en temps t(n) (le temps étant le nombre de transitions sur machine de Turing).
  - Pour les machines non déterministes, on définit la classe **NTIME(t(n))** des problèmes qui peuvent être résolus en temps **t(n)**.
  - Le coût d'un algorithme A pour une donnée d est le nombre d'opérations élémentaires n nécessaires au traitement de la donnée d et est noté $COUT_A(d)$
    - Complexité dans le pire des cas : $Max_A(n) = \max \{COUT_A(d), d \in Dn\}$ (Dn est ensemble de données de taille n)
    - Complexité dans le meilleur des cas : $Min_A(n) = \min \{COUT_A(d), d \in Dn\}$
    - Complexité en moyenne $Moy_A(n) = \sum_{d \in Dn} p(d)COUT_A(d)$ où p(d) est la probabilité d'avoir en entrée la donnée d parmi toutes les données de taille n. Si toutes les données sont équiprobables, alors on a, $Moy_A(n) = (1/|Dn|) \sum_{d \in Dn} COUT_A(d)$
- **Complexité en espace mémoire :** Il est quelquefois nécessaire d'étudier la complexité en mémoire lorsque l'algorithme requiert de la mémoire supplémentaire (tableau auxiliaire de même taille que le tableau donné en entrée par exemple)
  - La complexité en espace évalue l'espace mémoire utilisé en fonction de la taille des données ; elle est définie de manière analogue :
    - *SPACE(s(n)) = { L | L peut être décidé par une machine déterministe en utilisant au plus s(n) cellules de mémoire}*
    - *NSPACE(s(n)) = { L | L peut être décidé par une machine non-déterministe en utilisant au plus s(n) cellules de mémoire}*
- **Exemple**
  - **Problème :** Soit T un tableau de taille N contenant des nombres entiers de 1 à k. Soit a un entier entre 1 et k. La fonction suivante retourne la position du premier a rencontré s'il existe, et 0 sinon.

```
Trouve:=proc(T,n,a)
local i;
for i from 1 to n do
        if T[i]=a then RETURN(i) fi;
RETURN(0);
end;
```

    - Cas le pire : N (le tableau ne contient pas a)
    - Cas le meilleur : 1 (le premier élément du tableau est a)
    - Complexité moyenne : Si les nombres entiers de 1 à k apparaissent de manière équiprobable, on peut montrer que le cout moyen de l'algorithme est $k(1 - (1 - 1/k)^N)$

2. Techniques de calcul de la complexité

2.1. Analyse asymptotique

2.1.1. Les concepts

On étudie systématiquement la complexité *asymptotique*, notée grâce aux *notations de Landau*.

- La notion de grand **O**, aussi appelée **symbole de Landau**, est un symbole utilisé en théorie de la complexité, en informatique, et en mathématiques pour décrire le comportement asymptotique des fonctions. Fondamentalement, elle indique avec quelle rapidité une fonction « augmente » ou « diminue ».
  - Informellement, cette notion vient de deux idées simples

        - idée 1 : évaluer l'algorithme sur des données de grande taille. Par exemple, lorsque n est assez '***grand***', $3n^3 + 2n^2$ est essentiellement $3n^3$.
        - idée 2 : on élimine les constantes multiplicatrices, car deux ordinateurs de puissances différentes diffèrent en temps d'exécution par une constante multiplicatrice. De $3 * n^3$, on ne retient que n^3
        - L'algorithme est dit en $O(n^3)$.
    - L'idée de base est donc qu'un algorithme en $O(n^a)$ est « meilleur » qu'un algorithme en $O(n^b)$ si $a < b$.

- ***Notion de Grand O***
    - Supposons que f et g soient deux fonctions définies sur une partie de l'ensemble des nombres réels. Nous écrivons : $f(x) = O(g(x))$ (ou $f(x) = O(g(x))$ quand $x \to \infty$ pour être plus précis) si et seulement s'il existe des constantes N et C telles que pour tout $x>N$, $|f(x)| \leq C\,|g(x)|$.
    - Plus généralement, si a est un nombre réel, nous écrivons $f(x) = O(g(x))$ quand $x \to a$ si et seulement s'il existe des constantes $d > 0$ et C telles que pour tout x tel que $|x-a| < d$, $|f(x)| \leq C\,|g(x)|$.
    - Remarques
        - La relation O n'est pas symétrique
        - La première définition est la seule utilisée en informatique (où typiquement seules les fonctions positives à variable entière n sont considérées; les valeurs absolues peuvent être ignorées), tandis que les deux définitions sont utilisées en mathématiques.

        - Grand-O est la notation asymptotique la plus utilisé

> $f(n) = O(g(n))$ quand $n \to \infty$ si et seulement s'il existe des constantes N et C telles que pour tout $n>N$, $f(n) \leq C\,g(n)$.

- ***Notion de Grand notion* Ω**
    - Supposons que f et g soient deux fonctions définies sur une partie de l'ensemble des nombres réels. Nous écrivons : $f(x) = \Omega(g(x))$ (ou $f(x) = \Omega(g(x))$ quand $x \to \infty$) si et seulement s'il existe des constantes N et C telles que pour tout $x>N$, $C\,|g(x)| \leq |f(x)|$.
    - Remarques
        - La relation Ω n'est pas symétrique
        - La première définition est la seule utilisée en informatique (où typiquement seules les fonctions positives à variable entière n sont considérées; les valeurs absolues peuvent être ignorées), tandis que les deux définitions sont utilisées en mathématiques.

- ***Notion de Grand notion notion* Θ**
    - Supposons que f et g soient deux fonctions définies sur une partie de l'ensemble des nombres réels. Nous écrivons : $f(x) = \Theta(g(x))$ (ou $f(x) = \Theta(g(x))$ quand $x \to \infty$) si et seulement s'il existe des constantes N et $C1$ et $C2$ telles que pour tout $x>N$, $C1\,|g(x)| \leq |f(x)| \leq C2\,|g(x)|$
    - Remarques
        - La relation Θ est symétrique
        - Après grand-O, les notations Θ et Ω sont les plus utilisées en informatique

- ***Notion de Grand notion petit* o**
    - Dans le même esprit de manipulation des ordres de grandeur, la notation $f(x) = o(g(x))$ (cette fois avec un ***petit o***) signifie que la fonction f est négligeable devant la fonction g, quand x tend vers une valeur particulière. Formellement, pour $a \in \mathbb{R} \cup \{-\infty, +\infty\}$ et pour f et g deux fonctions de la variable réelle x, avec g qui ne s'annule pas sur un voisinage de a, on dit que $f(x) = o(g(x))$ quand $x \to a$ si et seulement si $f(x)/g(x) \to 0$ quand $x \to a$.
    - Remarques
        - Le petit-o est courant en mathématique mais plus rare en informatique

2.1.2. Application d'analyse asymptotique

Voici une liste de catégories de fonctions qui sont couramment rencontrées dans les analyses d'algorithmes. Les fonctions de croissance les plus lentes sont listées en premier. *c* est une constante arbitraire.

notation	complexité
$O(1)$	constante
$O(\log(n))$	logarithmique
$O((\log(n))^c)$	polylogarithmique
$O(n)$	linéaire
$O(n \log(n))$	parfois appelée « linéarithmique »
$O(n^2)$	quadratique
$O(n^c)$	polynomiale, parfois « géometrique »
$O(c^n)$	exponentielle
$O(n!)$	factorielle

Remarque :

- Notons que $O(n^c)$ et $O(c^n)$ sont très différents. Le dernier exprime une croissance bien plus rapide, et ce pour n'importe quelle constante $c>1$. Une fonction qui croît plus rapidement que n'importe quel polynôme est appelée *super-polynomiale*. Une fonction qui croît plus lentement que toute exponentielle est appelée *sous-exponentielle*. Il existe des fonctions à la fois super-polynômiales et sous-exponentielle comme par exemple la fonction $n^{\log(n)}$. Certains algorithmes ont un temps de calcul de ce type, comme celui de la factorisation d'un nombre entier.

- Remarquons aussi, que $O(\log n)$ est exactement identique à $O(\log(n^c))$. Les logarithmes diffèrent seulement d'un facteur constant, et que la notation grand « ignore » les constantes. De manière analogue, les logarithmes dans des bases constantes différentes sont équivalents.

- La liste précédente est utile à cause de la propriété suivante : si une fonction *f* est une somme de fonctions, et si une des fonctions de la somme grimpe plus vite que les autres, alors celle qui croît le plus vite détermine l'ordre de $f(n)$.

- exemple : si $f(n) = 10 \log(n) + 5 (\log(n))^3 + 7\, n + 3\, n^2 + 6\, n^3$,
  alors $f(n) = O(n^3)$.

2.1.3. Propriétés de grand O et Θ

1- $f \in \Theta(g) \Leftrightarrow f \in O(g)$ et $g \in O(f)$

2- $n^p \in O(n^{p+1})$ mais $n^p \notin \Theta(n^{p+1})$

3- Pour les notations Grand O et Θ, le coefficient d'un monôme est négligeable devant l'exposant, cequi limite donc la validité de la notion pour les faibles valeurs de n, c'est-à-dire pour les problèmes de petite taille. Par exemple : $f = n^2 + 200000000.n$ alors $f \in O(n2)$.

4- Les sommes de puissances d'entiers interviennent souvent dans le calcul :

5- $S_{1,n} = \sum_{i=1,\,n} i = 1+2+3+\ldots+n = n(n+1)/2$ $\quad S_{1,n} \in O(n^2)$
$S_{2,n} = \sum_{i=1,\,n} i^2 = 1+4+9+\ldots+n^2 = n(n+1)(2n+1)/6$ $\quad S_{2,n} \in O(n^3)$
$S_{k,n} = \sum_{i=1,\,n} i^k = 1^k+2^k+3^k+\ldots+n^k$ $\quad S_{k,n} \in O(n^{k+1})$ (démontrable par récurrence)

6- Symétrie : $f \in \Theta(g) \Rightarrow g \in O(f)$

7- Transitivité : $\{f \in O(g) \wedge g \in O(h)\} \Rightarrow g \in O(h)$ et $\{f \in \Theta(g) \wedge g \in \Theta(h)\} \Rightarrow g \in \Theta(h)$

8- $f \in O(g) \Rightarrow \lambda.f \in O(g), \lambda \in \mathbb{R}^{+*}\} \; g \in O(h)$

9- $\{f_1 \in O(g_1) \wedge f_2 \in O(g_2)\} \Rightarrow f_1 + f_2 \in O(\max(g_1, g_2))$

10- $\{f_1 - f_2 \geq 0 \wedge f_1 \in O(g_1)\} \Rightarrow f_1 - f_2 \in O(g_1)$

11- $\{f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2)\} \Rightarrow f_1 + f_2 \in \Theta(\max(g_1, g_2))$

12- $\{f_1 - f_2 \geq 0 \wedge f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) \wedge \lim_{n \to \infty}(g_2(n) / g_1(n)) = 0\} \Rightarrow f_1 - f_2 \in \Theta(g_1)$

13- $\{f_1 \in O(g_1) \wedge f_2 \in O(g_2)\} \Rightarrow f_1 . f_2 \in O(g_1 . g_2)$

14- $\{f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2)\} \Rightarrow f_1 . f_2 \in \Theta(g_1 . g_2)$

15- $\lim_{n \to \infty}(f(n) / g(n)) = a \neq 0\} \Rightarrow f \in \Theta(g)$

16- $\lim_{n \to \infty}(f(n) / g(n)) = 0\} \Rightarrow f \in O(g) \wedge f \notin \Theta(g)$

17- $\lim_{n \to \infty}(f(n) / g(n)) = \infty\} \Rightarrow g \in O(f) \wedge g \notin \Theta(f)$

2.1.4. Evaluation de la complexité d'un algorithme

A compléter

2.2. Principe de transformation de problème :

Le principe consiste simplement à transformer le problème à classer en un problème déjà classé. On utilise les propriétés des classes de complexité pour transformation.

2.2.1. Classes de complexité

La théorie de la complexité repose sur la définition de classes de complexité qui permettent de classer les problèmes en fonction de la complexité des algorithmes qui existent pour les résoudre.

- **Classe L (LOGSPACE)** : Un problème de décision qui peut être résolu par un algorithme déterministe en espace *logarithmique* par rapport à la taille de l'instance est dans L.

- **Classe NL** : Cette classe s'apparente à la précédente mais pour un algorithme non-déterministe.

- **Classe P** : Un problème de décision est dans *P* s'il peut être décidé par un algorithme déterministe en un temps *polynomial* par rapport à la taille de l'instance. On qualifie alors le problème de polynomial.

    - Exemple :
      Prenons par exemple le problème de la connexité dans un graphe. Étant donné un graphe à *s* sommets, il s'agit de savoir si toutes les paires de sommets sont reliées par un chemin. Pour le résoudre, on dispose de l'algorithme de parcours en profondeur qui va construire un arbre couvrant du graphe à partir d'un sommet. Si cet arbre contient tous les sommets du graphe, alors le graphe est connexe. Le temps nécessaire pour construire cet arbre est au plus s^2, donc le problème est bien dans la classe P.

    - Les problèmes dans P correspondent en fait à tous les problèmes facilement solubles.

- **Classe NP** : Un problème NP est *N*on-déterministe *P*olynomial (et non pas *Non polynomial*, erreur très courante).

    - La classe NP réunit les problèmes de décision pour lesquels la réponse *oui* peut être décidée par un algorithme non-déterministe en un temps polynomial par rapport à la taille de l'instance.

    - De façon équivalente, c'est la classe des problèmes qui admettent un algorithme dans *P* qui, étant donné une solution du problème *NP* (un *certificat*), est capable de répondre *oui* ou *non*.

- Intuitivement, les problèmes dans *NP* sont tous les problèmes qui peuvent être résolus en énumérant l'ensemble des solutions possibles et en les testant avec un algorithme polynomial.

- Exemple
  Par exemple, la recherche de cycle hamiltonien dans un graphe peut se faire avec deux algorithmes :

  - le premier génère l'ensemble des cycles (ce qui est exponentiel) ;

  - le second teste les solutions (en temps polynomial).

- **Classe Co-NP (Complémentaire de NP)** : C'est le nom parfois donné pour l'équivalent de la classe *NP*, mais avec la réponse *non*.

- **Classe PSPACE :** Elle regroupe les problèmes décidables par un algorithme déterministe en espace polynomial par rapport à la taille de son instance.

- **Classe NSPACE ou NPSPACE** :
  Elle réunit les problèmes décidables par un algorithme non-déterministe en espace polynomial par rapport à la taille de son instance

- **Classe EXPTIME**
  Elle rassemble les problèmes décidables par un algorithme déterministe en temps exponentiel par rapport à la taille de son instance.

- **Classe NEXPTIME**
  Elle rassemble les problèmes décidables par un algorithme non déterministe en temps exponentiel par rapport à la taille de son instance.

2.2.2. Propriété des Classes de complexité

- **Complexité en temps et en espace séparément**
  - P ⊆ NP et symétriquement P ⊂ co-NP
  - P ⊆ NP ⊆ EXPTIME
  - LOGSPACE ⊆ PSPACE = NPSPACE
- **Liaison entre temps et espace**
  - Si $SPACE(s(n)) \geq n$ alors $SPACE(s(n)) \leq TIME(s(n))$.

  (Si M fonctionne en temps t(n) alors M fonctionne en espace au plus t(n))
- **Complexité en temps et espace**
  - LOGSPACE ⊆NLOGSPACE ⊆ P ⊆ NP ⊂ PSPACE = NPSPACE
  - et symétriquement Co-NP ⊂ PSPACE
  - P ⊆ NP⊂ PSPACE = NPSPACE ⊆ EXPTIME

- **Problème C-Complet et réduction**

  - **C-Difficile**
    Un problème est **C-difficile** si ce problème est au moins aussi dur que tous les problèmes dans **C**.

  - **C-Complet**
    Soit **C** une classe de complexité (comme **P**, **NP**, etc.). On dit qu'un problème est **C-complet** si il est dans **C**, et il est **C-difficile** (on utilise parfois la traduction incorrecte **C-dur**).

2.2.3. Exemple

Démontrons ici que le problème de la recherche de cycle hamiltonien dans un graphe orienté est NP-Complet.

- Le problème est dans *NP* : On peut trouver de façon évidente un algorithme pour le résoudre avec une machine non-déterministe, par exemple en énonçant tous les cycles puis en sélectionnant le plus court.

- Nous disposons du problème de la recherche de cycle hamiltonien pour les graphes non-orientés. Un graphe non-orienté peut se transformer en un graphe orienté en « doublant » chaque arête de manière à obtenir, pour chaque paire de nœuds adjacents, des chemins dans les deux sens. Il est donc possible de ramener le problème connu, NP-difficile, au problème que nous voulons classer. **Le nouveau problème est donc NP-difficile**.

- Le problème étant dans *NP* et *NP-difficile*, il est **NP-complet**.