TP : la classe NP

1. Qu'est-ce qu'une propriété NP ?

Q 1. Définir une notion de certificat pour le problème.

Données:

- *m, un entier positif – un nombre de machines*
- *n, un entier positif – un nombre de tâches*
- *ai, ti, pour chaque tâche i, 0 ≤ i ≤ n − 1, sa date d'arrivée et sa durée (toutes les deux entières).*
- *D, l'attente maximale autorisée.*

Certificat : Une liste de tâches triées en fonction de leur ordre d'exécution.

Quelle sera la taille d'un certificat?

Taille du certificat = O(n*log(n)) avec log(n) qui correspond à l'encodage d'un entier.

La taille des certificats est-elle bien bornée polynomialement par rapport à la taille de l'entrée ?

Oui, car elle dépend du nombre de tâches données : **O(n) <= n** *avec n=nombre de tâches*

Proposez un algorithme de vérification associé. Est-il polynomial ?

Oui c'est bien polynomiale, ça dépend de la taille des données, plus spécifiquement de la taille de n et de m , cet algo de vérif a une complexité en O((n-m)*m).

L'algo proposé est dans la fonction *`bool verificationAlgorithm(Configuration config, vector <int> certificat)`*

```
bool verificationAlgorithm(JSP instance, vector <int> certificat) {
    vector<Task> m;
    for(int i = 0; i<instance.m ; i++) {
        m.push_back(instance.tasks[certificat[i]]);
    }

    for(int i = instance.m ; i< instance.n;i++) { // (n - m) loop
        int busyMachine = 0;
        bool alreadyChange = false;
       for(int j = 0 ; j < m.size() ; j++) { // m(=number of machine) loop
           int duration = m[j].a + m[j].t;
            if(duration <= instance.tasks[certificat[i]].a + instance.D ) {
                if(alreadyChange==false) {
                    if(instance.tasks[certificat[i]].a < duration) {
                    instance.tasks[certificat[i]].a = duration;
                    }
                m[j] = instance.tasks[certificat[i]];
                alreadyChange = true;
                }
            }else {
                busyMachine++;
            }
       }

       if(busyMachine==instance.m) { // if the number of busy machine is == at m means that the certificat is false
           return false;
       }
       alreadyChange = false;
    }
    return true;
}
```

Q 2.1 Proposez un algorithme de génération aléatoire de certificat, i.e. qui prend en entrée une instance du problème, et génère aléatoirement un certificat de façon à ce que chaque certificat ait une probabilité non nulle d'apparaître.

```
// Q 2.1.Génération aléatoire d'un certificat
vector<int> generationAleatoireDeCertificat(JSP instance) {
    vector<int> certificat;

    while(certificat.size() < instance.n) {
        int nb = rand() % instance.n; //on choisit un nombre aleatoire qui est compris entre 0 et n
        vector<int>::iterator it = find(certificat.begin(), certificat.end(), nb);
        if(it == certificat.end()) { // on vérifie que ce numéro n'est pas déjà dans le certificat
            //(on ne veut pas avoir deux fois la même tâche)
            certificat.push_back(nb); //on place le numéro dans le certificat
        }
    }
    return certificat;
}
```

Q 3.1. Pour une instance donnée, pouvez-vous donner un ordre de grandeur du nombre de certificats ?

Ordre de grandeur = toutes les combinaisons possibles en fonction du nombre de tâches (=n) donc **!n**

Q 3.2. Enumération de tous les certificats ou l'algorithme du British Museum

Quel ordre proposeriez-vous ?

On commence avec une liste croissante : 1,2,3,…, n, puis on fait on fait toute les permutations possibles lorsque 1 est à l'index, ensuite toutes les permutations possibles lorsque 2 est à l'index 0, ...

Comment déduire de ce qui précède un algorithme pour tester si le problème a une solution ?

Il faut tester toutes les possibilités si on a une possibilité qui est correcte ça veut dire que le problème a une solution sinon il n'en a pas. Il faut tout tester dans l'ordre cité plus haut et ainsi être sûr que toutes les possibilités ont été testées.

Quelle complexité temporelle a cet algorithme ?

On utilise la méthode std::next_permutation four par la librairie `<algorithm>` en C++.
Selon la doc on est en complexité dans le pire des cas en O(n) où n représente la taille du tableau donc ici le nombre de tâche.

Quelle est sa complexité spatiale ?

Celui-ci est en temps constant O(1).

Implémentation

Voir code répo gitlab.

Pour l'utiliser en ligne de commande : //CLI:

- command to compile this file: g++ -std=c++11 main.cpp -o NomExe
- to run exe: ./NomExe nomFichierDeDonneesTest -option (int)
- exemple chemin de fichier : donneesTestNP/donnee3
- options possibles: -verif -nondet -exhaust
- int: donnée D du problème JSP (pas obligatoire: si rien D=0)

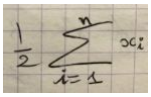
2. Réductions polynomiales

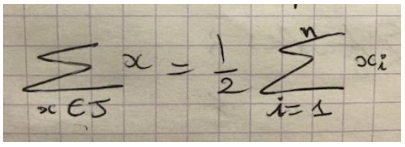
Q 1. Une première réduction

Soit le problème de décision *Partition* défini par :
Donnée :
n –un nombre d'entiers
$x_1, \cdots, x_n$ –les entiers
Sortie : Oui, si il existe un sous-ensemble de $[1..n]$ tel que la somme des x_i correspondants soit exactement la moitié de la somme des x_i, i.e. $J \subset [1..n]$, tel que $\sum_{i \in J} x_i = \sum_{i \notin J} x_i = \frac{\sum_{i=1}^{n} x_i}{2}$

Q 2. Montrer que Partition se réduit polynomialement en JSP.

Soit I une instance de Partition définie par n, un nombre d'entier et E un ensemble de n entiers :
I: n et E = {x1, ... , xn}
On construit une instance de JSP à partir des données de I :
On prend :

- m = 2
- nombre de tâches n (de JSP) = n (de I) + 2
- ai = 0 pour toutes tâches
- deux tâches avec ti = D
- et ti = xi pour le reste de n tâches
- D = 

Soit un ensemble J inclu dans E tel que 

Les tâches correspondant aux xi de l'ensemble J s'exécutent sur une machine et le reste des tâches sur l'autre machine sans jamais attendre plus de temps que la valeur de D.

- Soit I est valide : on peut diviser les xi en 2 groupes dont la somme des xi de chaque groupe est égale et cette somme est égale à ½ de la somme de tous les xi de E.
- Soit red(I) est valide : les tâches s'exécutent en se répartissant sur 2 machines et n'attendent jamais plus que D secondes avant de s'exécuter.

Donc I est valide si et seulement si red(I) est valide et la réduction est polynomiale.

Q 2.1. Qu'en déduire pour JSP ?

Par définition, Une propriété R est dite NP-dure Si toute propriété NP se réduit polynomialement en R.

Le problème de Partition est un problème connu et on sait qu'il est NP complet (NP + NP-dure), donc on peut en déduire que JSP est NP-dur

Q 2.2. Pensez-vous que JSP se réduise polynomialement dans Partition? Pourquoi?

Oui, car les deux problèmes sont NP-durs, nous avons prouvé que Partition se réduisait polynomialement en JSP donc l'inverse est également possible puisque la réduction polynomiale est réciproque. En effet, nous pouvons passer de n'importe quelle instance de JSP vers Partition et inversement.

Q 2.3. [à coder] Implémenter la réduction polynomiale de Partition dans JSP. L'utiliser pour résoudre Partition à partir de votre code pour résoudre JSP. Vous pouvez tester avec les données fournies.

Pour tester les algorithmes JSP à partir de données de Partition il faut mettre **-partition** à la place de la valeur de D à la fin de la ligne de commande pour lancer le programme. De cette manière on lit le fichier de donnée de Partition et on le transforme en donnée de type JSP et on peut utiliser les algorithmes en précisant les options : -verif, -nondet, -exhaust.

Par exemple : *./NomExe exPart2 -verif -partition*

Code transformation de donnée Partition en donnée JSP :

```
if(!strcmp(argv[3], "-partition")) { //si on test des données de Partition avec les algo de JSP
//on prend m=2 et on construit toutes les tâches avec ai=0 et on prend D=somme totale des xi/2
    m = 2;
    n = stoi(dataPartition[0]);
    for(unsigned int i=1; i<=n; i++) {
        int t = stoi(dataPartition[i]);
        Task task(0, t);
        tasksList.push_back(task);
        D+=t;
    }
    D=D/2;
    //ajout des deux tâches en plus avec ai=0 et ti=D:
    n+=2;
    Task task(0, D);
    tasksList.push_back(task);
    tasksList.push_back(task);
```

Q 3.1. Montrer que SUM est NP.

Données:

- *n – un nombre d'entiers*
- *x1, · · · , xn –les entiers*
- *s –un entier cible*

Certificat **:** Un sous-ensemble J ⊂ [1..n] de SUM

Algo de vérif : On parcourt le sous-ensemble J, on additionne ses valeurs et on regarde si le résultat est égal à l'entier cible s.

```
verificationAlgorithm(s, verctor<int> J)
{
    int sum = 0;

    for (int i = 0; i < J.size();i++) sum +=J[i];

    if(sum==s) return true;

    return false;
}
```

Sum est bien NP car on peut vérifier un certificat en temps polynomial, cet algorithme est en O(n) où n est la taille du sous ensemble J de SUM donc cet algorithme est polynomialement borné par la taille de la donnée.

Q 3.2. Montrer que SUM se réduit polynomialement en Partition.

Il faut prouver que n'importe quelle donnée de SUM se réduit polynomialement en Partition

Soit I = instance de SUM :

- n = 3 et
- Une liste L d'entier : {4,5,7}
- Entier cible s = 9

Sortie : Oui, le sous-ensemble J : [4,5], 4 + 5 = 9.

Pour la réduction polynomiale de SUM dans Partition, prenons la valeur sum = la somme des entiers de la liste *L*, créons la liste L' dans laquelle on ajoute ces deux valeurs : *sum + s, 2sum - s* à *L*. Le nombre d'entrées de Partition devient *n' = n + 2*.

On aurait alors comme données pour Partition :

- n = 5 = n'
- 4, 5 , 7, 23, 25 = L'

Sortie : oui, 25 + 7 = 4 + 5 + 23 car le sous ensemble *J + 2sum – s = sum + s + le reste*.

Donc SUM se réduit polynomialement en Partition l'algorithme de réduction polynomiale est en O(n) où n est la taille de L (=la liste d'entiers de SUM).

Q 3.3. Implémenter la réduction ci-dessus de SUM dans Partition.

```
//Partition
struct Partition {
    int n;
    vector <int> L;

    Partition(int number, vector <int> list ) {
        n = number;
        L = list;
    }
};

Partition sumIntoPartition(int n, vector <int> L, int s) {
    int sum = 0;
    for(int i = 0; i < L.size(); i++) {
        sum += L[i];
    }

    vector <int> newL = L;
    newL.push_back(sum + s);
    newL.push_back(2*sum - s);

    Partition partition(n+2, newL);

    return partition;
}
```

Q 4. Composition de réductions

En utilisant les deux réductions précédentes, implémenter une réduction de SUM dans JSP.

Si red1 est une réduction polynomiale de Sum dans Partition , et red2 est une réduction polynomiale de Partition vers JSP, alors red1 ◦ red2 est une réduction polynomiale de Sum vers JSP.

Implémentation :

```cpp
// Reduction Partition into JSP
JSP partitionIntoJsp(Partition partition) {
    int m = 2;
    //ajout des deux tâches en plus avec ai=0 et ti=D:
    int n = partition.n +2;
    vector <Task> tasksList;
    int D = 0;
    for(unsigned int i=1; i<=n; i++) {
        int t = partition.L[i];
        Task task(0, t);
        tasksList.push_back(task);
        D+=t;
    }

    D=D/2;
    Task task(0, D);
    tasksList.push_back(task);
    tasksList.push_back(task);

    JSP instance(m,n,tasksList,D);

    return instance;
}

//Reduction SUM to Partition, Params corresponds to a instance of SUM
void sumIntoPartition(int n, vector <int> L, int s) {
    int sum = 0;
    for(int i = 0; i < L.size(); i++) {
        sum += L[i];
    }

    vector <int> newL = L;
    newL.push_back(sum + s);
    newL.push_back(2*sum - s);

    Partition partition(n+2, newL);

    partitionIntoJsp(partition);
}
```

3. Optimisation versus Décision

Par définition, si on avait un algorithme P pour le problème d'optimisation, on en aurait un pour celui de décision.

JSPOpt1

Donnée

m, un entier positif – un nombre de machines

n, un entier positif – un nombre de tâches

a_i, t_i : pour chaque tâche i, $0 \leq i \leq n-1$, sa date d'arrivée et sa durée (toutes les deux entières).

Sortie l'attente minimale d'un ordonnancement correct pour le problème, i.e. le plus petit D telle qu'il existe un ordonnancement correct tel que chaque tâche attende au plus D.

JSPOpt2

Donnée

m, un entier positif – un nombre de machines

n, un entier positif – un nombre de tâches

a_i, t_i : pour chaque tâche i, $0 \leq i \leq n-1$, sa date d'arrivée et sa durée (toutes les deux entières).

Sortie Un ordonnancement correct pour le problème, et optimal pour l'attente.

Q1. **Montrer que si JSPOpt1 (resp. JSPOpt2) était P, la propriété JSP le serait aussi; qu'en déduire pour JSPOpt1 (resp. JSPOpt2)?**

Si on suppose que JSPOpt1 (resp. JSPOpt2) est P, étant donné que ces deux problèmes d'optimisation sont associés au problème de JSP on peut déjà en déduire que JSP serait également P. En effet, si on a un algorithme A, qui résout JSPOpt1 de manière polynomiale, et qui retourne une valeur D, cela signifie donc que JSP retourne Oui, car il existe un ordonnancement correct avec D JSP correspondant à D JSPOpt1.

Q2. Montrer que si la propriété JSP était P, JSPOpt1 le serait aussi.

S'il existait un algorithme polynomial appelé A pour trouver un ordonnancement correct qui garantit une attente maximale d'au plus D secondes, cela signifierait donc que JSP appartient à P. On pourrait alors utiliser cet algorithme A pour trouver l'attente minimal D d'un ordonnancement (JSPOpt1). On applique cet algorithme A, on retrouve l'ordonnancement optimal et on retourne la valeur D de cet ordonnancement tel que chaque tâche attende au plus D secondes.

Q3. Montrer que si la propriété JSP était P, JSPOpt2 le serait aussi.

Si JSP est P, cela signifie qu'il existe un algorithme polynomial permettant de trouver un ordonnancement correct en fonction d'une valeur D fournie. Si cet algo retourne "Oui" avec cette instance JSP avec D, il suffit de vérifier s'il n'existe pas de solution au problème JSP avec D - 1 si oui on regarde avec D - 2, ... jusqu'à arriver au cas où c'est non et là on retourne la valeur de D + 1. Ce qui nous donnera la valeur de D optimale.