Algorithmique (AL5)
Devoir maison : XIPUX install party

Ce devoir maison est à réaliser en **binôme** et à rendre en TD, à votre chargé de TD, ou en amphi, la semaine du **25 novembre**.
Les algorithmes du cours que vous utiliserez devront être nommés (avec la page du poly où les trouver). Bien sûr, vous n'aurez pas besoin de prouver de nouveau leur correction et leur complexité. Il vous suffira, le cas échéant, de rappeler leur complexité.
Pour tout nouvel algorithme que vous présenterez, il faudra justifier sa **correction** et sa **complexité**.
Vous pourrez utiliser des structures de données usuelles, mais dans ce cas, il faudra être explicite sur les fonctions usuelles d'accès à ses structures que vous invoquerez, c'est-à-dire les nommer et préciser leur rôle.

Présentation du problème

On considère le système d'exploitation XIPUX [1]. Il est découpé en *packages*, chaque package contenant divers fichiers et répertoires (exécutables, documentation, fichiers de configuration etc) fournissant un service précis dans le système. Par exemple le package `emacs` fournit l'éditeur de texte utilisé par les vrais programmeurs XIPUX.

On appelle **dépôt** P l'ensemble de tous les packages XIPUX existants. Il y a de l'ordre d'une dizaine de milliers de packages. On ne va pas tous les installer, en particulier parce qu'il y a des **conflits** possibles entre certains packages. Quand deux packages sont en conflit, on ne peut pas avoir les deux en même temps sur une même machine (on peut choisir d'installer l'un des deux, ou aucun des deux). Par exemple le package `xserver` et le package `hardware-teletype` sont en conflit car l'un suppose que la machine sur laquelle il tourne a un écran et une carte graphique, et l'autre qu'elle n'en a pas.

Il y a aussi des **dépendances** entre packages. Par exemple le package `xemacs` dépend du package `emacs` et aussi du package `xserver`. Le package `vi` dépend lui du package `hardware-teletype`.

Etant donné un ensemble de package deja installés, on dit qu'un nouveau package peut être installé si et seulement si :

1. tous les packages dont il dépend sont déjà installés et
2. aucun package en conflit avec lui n'est déjà installé.

Un ensemble de packages $I \subset P$ est une **installation** si il existe un ordre $p_1, p_2, \dots p_s$ sur ses packages tel que tout i, le package p_i peut être installé si les packages $p_1, p_2, \dots p_{i-1}$ sont déjà installés

Modélisation

Un package p du dépôt P sera représenté de la façon suivante :
- son nom,
- sa taille exprimée en octets $w(p)$,
- l'ensemble de dépendances $D(p) \subset P$,
- l'ensemble de conflits $C(p) \subset P$.

Les dépendances et les conflits sont naturellement représentés dans ce problème par deux graphes dont l'ensemble des sommets est l'ensemble de tous les packages du dépôt :

1. *Xipux Is Probably UniX*

- le graphe non orienté représentant les conflits. On met une arête entre deux packages s'ils sont en conflit ;
- le graphe orienté représentant les dépendances. On met un arc orienté du package p vers le package q si le package p dépend du package q.

Un exemple

Exercice 1 : un exemple

Soit la liste de packages suivante :

. Package : **bouml**
taille : 55401
dépendances : libc6, libgcc1, libqt4-network, libqtcore4, libstdc++6

. Package : **libc6**
taille : 12337
dépendances : libgcc1

. Package : **libgcc1**
taille : 116
dépendances : gcc-8-base

. Package : **libqt4-network**
taille : 2143
dépendances : libc6, libgcc1, libqtcore4, libstdc++6, zlib1g

. Package : **libqtcore4**
taille : 5132
dépendances : libc6, libgcc1, libstdc++6, zlib1g

. Package : **libstdc++6**
taille : 2065 kB
dépendances : gcc-8-base, libc6, libgcc1
conflits : scim

. Package : **scim**
taille : 1784 kB
dépendances : libc6, libgcc1, libx11-6
conflits : libstdc++6

. Package : **gcc-8-base**
taille : 254 kB

. Package : **libx11-6**
taille : 1592 kB
dépendances : libc6, libx11-data

. Package : **zlib1g**
taille : 178 kB
dépendances : libc6
conflits : zlib1

. Package : **zlib1**
taille : 197 kB
dépendances : libc6
conflits : zlib1g

. Package : **libx11-data**
taille : 1713 kB

1. Donner la représentation des dépendances et des conflits sous la forme des deux graphes évoqués plus haut.
2. Le package **zlib1** peut-il être installé si le package **scim** n'a pas été installé ? Et si le package **libgcc1** n'a pas été installé ?
3. Les packages **bouml** et **libgcc1** sont-ils installables simultanément ?

Dépôt sans Conflits

Dans un premier temps, on suppose que le dépôt ne contient aucun conflit.

Exercice 2 : cohérence du dépôt

Étant donné un dépôt P, on dit que P est *cohérent* s'il contient uniquement des packages qui peuvent être installés effectivement, c'est-à-dire que pour chaque package il existe au moins une installation qui le contient.

1. Montrer que si le dépôt est cohérent alors le graphe des dépendances ne contient pas de circuit orienté.
2. Montrer la réciproque en montrant que si le graphe ne contient pas de circuit, alors l'ensemble de tous les packages du dépôt est bien une installation.
3. En déduire un algorithme qui, étant donné P, détermine si il est cohérent ou non.

Pour le reste de cette section, on supposera que le dépôt est cohérent (ce n'est donc pas à tester).

Exercice 3 : validité

Montrer qu'un ensemble de package P' est une installation (au sens de la définition de l'introduction) si et seulement si la condition suivante est vérifiée

$$\forall p \in P' \; D(p) \subset P'$$

En déduire un algorithme qui, étant donné un ensemble de packages P', répond Vrai si P' est une installation et Faux sinon.

Exercice 4 : dépendances manquantes

Écrire un algorithme qui, étant donné un dépôt P et un ensemble $P' \subset P$, complète P' en une installation I en ajoutant à P' *le moins* de packages possibles.

Temps pour installer un ensemble de packages

Pour les deux exercices qui suivent, on souhaite installer une installation I sur une machine **en minimisant le temps total que cela prendra** [2].

Pour un package p, **le temps** $t(p)$ mis pour le télécharger et l'installer dépend linéairement de sa taille $w(p)$: il existe une constante v, la **vitesse** de la machine (qui dépend entre autres de la vitesse de sa connection Internet), telle que $t(p) = v \cdot w(p)$ exactement. On dénote $disp(p)$ la date que le package p sera installé et disponible. Ainsi, si T est la date à la quelle la machine lance le téléchargement de p, alors $disp(p) = T + t(p)$.

De plus, le processus doit être **interruptible**, c'est-à-dire que si on éteint la machine à tout moment, les packages déjà installés doivent constituer une installation valide. En d'autres termes, **on ne peut démarrer l'installation d'un package que si les packages dont il dépend sont deja installés**.

Exercice 5 : machine séquentielle

Dans cette version, un seul package peut être installé à la fois. La machine (plus exactement, un logiciel d'installation tournant sur une clef USB par exemple) doit télécharger et ensuite installer chaque package de façon atomique : il faut qu'un package soit complètement installé avant de commencer à télécharger (et installer) le suivant.

1. Donner un algorithme qui calcule l'ordre dans lequel les packages doivent être installés (un ordre total sur I) et les dates de disponibilité des packages. L'algorithme produit comme résultat deux tableaux o et $disp$ tel que pour chaque package $p \in I$:
   - son numéro d'**ordre** $o(p)$ vaut entre 1 (pour le package installé en premier) et $|I|$ (pour celui installé en dernier),

2. I est une installation, inutile de revérifier que ses packages sont installables.

– sa **date de disponibilité** $disp(p)$ dénote la date à laquelle le package p aura été installé et sera disponible.

La machine aura donc terminé l'installation de tous les packages et sera totalement opérationnelle à la date $disp(p')$ où p' est le package tel que $o(p') = |I|$, le dernier package installé.

2. Prouver l'optimalité de votre algorithme en montrant que le processus prendra toujours le même temps sur la machine séquentielle, indépendamment de l'ordre choisi pour installer les packages.

Exercice 6 : machine multitâche

Maintenant on considère une machine qui peut installer plusieurs packages de l'installation I à la fois. Pour démarrer l'installation d'un package il faut toujours avoir téléchargé et installé tous ses dépendances.

On veut toujours calculer les dates $disp(p)$ pour tout package p de I. Observer que la sortie de l'algorithme de l'exercice 5 est aussi une réponse valide pour cette machine. Cependant elle n'est pas optimale vu que la machine multitâche peut installer deux (ou plusieurs) packages au même temps si l'un ne dépend pas à l'autre. On veut trouver le **temps minimal** que l'installation de I prendra.

1. Soit $I = p_1, \ldots, p_k$. Exprimer en fonction de $t(p_i)$ le temps optimal pour installer I sur une machine multitâche dans les exemples suivants :
   – l'installation I est constituée de k packages sans aucune dépendance entre eux.
   – l'installation I est constituée de k packages formant un chemin orienté de longueur k dans le graphe de dépendances.

2. Donner un algorithme qui produit en sortie le tableau des dates $disp$, comme vous avez fait dans l'exercice 5. Astuce : il sera utile d'exprimer $disp(p)$ en fonction du $t(p)$ et des $disp(p_i)$ pour les packages $p_i \in D(p)$.

Dépôt ayant des conflits

On considère maintenant la version plus complète du problème où le dépôt a aussi des conflits. On reconsidère les exercices 2 et 3. On dira toujours qu'un dépôt est cohérent si tout package appartient au moins à une installation.

Exercice 7 : validité avec conflits

1. Soit P un dépôt *cohérent* ayant des conflits. Ajouter a la condition donnée dans l'exercice 3 une condition similaire portant sur le graphe des conflits pour obtenir les conditions nécessaires et suffisantes pour qu'un ensemble $P' \subset P$ soit une installation.

2. En utilisant la condition ci-dessus, donner un algorithme qui, étant donné P', répond vrai si P' est une installation, faux si non.

Exercice 8 : cohérence du dépôt avec conflits

1. Trouver la condition à ajouter à celle de l'exercice 2 pour obtenir les conditions nécessaires et suffisantes pour qu'un dépôt soit cohérent quand il possède des conflits.

2. Donner un algorithme qui, étant donné P ayant des conflits, détermine si il est cohérent ou non.