TP Algorithmique et Optimisation Discrète

Génération de patch optimal

2015 – 2016

En informatique, un *patch* est une suite d'instructions de transformation appliquées à un flot en entrée (par exemple un fichier) et dont le résultat est écrit sur le flot de sortie. Le flot est ici traité ligne par ligne, de la première à la dernière [1]. Une ligne se termine par le caractère NEWLINE (`\n`). Chaque ligne a un numéro à partir de 1 pour la première ligne ; le début du fichier (avant la première ligne) correspond par convention à une ligne virtuelle de numéro 0.

Un patch est une suite d'instructions, chacune écrite sur une ou deux lignes. La première ligne d'une instruction de patch est composée d'un caractère définissant l'instruction (`+`, `=`, `d` ou `D`), suivi d'un caractère espace puis d'un entier $k \geq 0$ indiquant le numéro de la ligne du flot d'entrée affectée par l'instruction. Cette première ligne de l'instruction peut être :

"`+` k`\n`" *Ajout* : la ligne suivante dans le patch est ajoutée sur le flot de sortie après la ligne k du flot d'entrée. Si $k = 0$, la ligne est insérée avant la première ligne (numérotée 1) du flot d'entrée.

"`=` k`\n`" *Substitution* : la ligne k du flot d'entrée est remplacée sur le flot de sortie par la ligne suivante dans le patch.

"`d` k`\n`" *Destruction* : la ligne k du flot d'entrée n'est pas recopiée sur le flot de sortie.

"`D` k m`\n`" *Multi-destruction* : les m lignes de la ligne k à la ligne $k+m-1$ du flot d'entrée ne sont pas recopiées sur le flot de sortie.

Le fonctionnement est le suivant :

- Toutes les lignes du flot d'entrée qui ne sont pas détruites et dont le numéro n'apparait pas dans une instruction de substitution sont recopiées sur le flot de sortie.
- Les numéros de lignes k dans les instructions du patch apparaissent par ordre croissant.
- Les instructions sont exécutées dans l'ordre du début à la fin ; ainsi deux commandes d'ajout consécutives avec un même numéro de ligne entraînent l'ajout consécutif des deux lignes opérandes dans l'ordre sur la sortie.
- Une instruction de substitution a un numéro de ligne strictement supérieur à l'instruction qui la précède.
- Toutes les instructions qui suivent une instruction de destruction "`d` k" (resp. "`D` k m") ont un numéro de ligne supérieur ou égal à $k+1$ (resp. $k+m$).

La commande fournie `applyPatch` écrit sur la sortie standard le résultat de l'application du patch *patch_file* au fichier en lecture *original_file* :

`applyPatch` *patch_file original_file*

Côut d'un patch. Le coût d'une instruction de patch dont la première ligne commence par "`+`" ou "`=`" est 10+s, où s est le nombre de caractères de la ligne qui suit dans le patch (en incluant `\n`). Le coût d'une instruction de destruction commençant par "`d`" (resp. "`D`") est 10 (resp. 15). Le coût d'un patch est la somme des coûts de ses instructions.

Exemple. Pour le patch P (listing 1) et le fichier F (listing 2), la commande : `applyPatch P F` écrit sur la sortie standard les lignes du listing 3. Le patch P est de coût 67.

```
1 + 0
2 @#?!
3 + 1
4 u
5 = 3
6 v
7 D 4 2
8 + 6
9 ww
```

Listing 1 – Fichier `P` de patch.

```
1 a b
2 ccc
3 d
4 ee
5 ff
6 g
```

Listing 2 – Fichier `F` d'entrée.

```
1 @#?!
2 a b
3 u
4 ccc
5 v
6 g
7 ww
```

Listing 3 – Sortie de `applyPatch P F`

1. La spécification simplifiée dans ce TP diffère des commandes `diff` et `patch` de Unix, mais le principe est similaire.

But du TP : écrire un programme `computePatchOpt` qui prend en entrée deux fichiers F_1 et F_2 en lecture et qui écrit sur la sortie standard un patch P de coût minimal tel que la commande "`applyPatch` P F_1" écrit en sortie un contenu identique à F_2. Ce programme doit implémenter la méthode de programmation dynamique.

Patch restreint Un patch est dit *restreint* si il ne comporte pas d'instruction de multi-destruction (préfixée par `D`). Le *problème du patch restreint* est de calculer un patch restreint de coût minimal.

Exemple. Le patch `R` (listing 5) est un patch restreint de coût 72 équivalent au patch `P` (listing 4).

```
1  + 0
2  @#?!
3  + 1
4  u
5  = 3
6  v
7  D 4 2
8  + 6
9  ww
```

Listing 4 – Fichier de patch `P`.

```
1  + 0
2  @#?!
3  + 1
4  u
5  = 3
6  v
7  d 4
8  d 5
9  + 6
10 ww
```

Listing 5 – Fichier de patch restreint `R` équivalent.

Remarque sur l'implémentation

— Le choix du langage de programmation pour la programmation dynamique est laissé libre ; il sera précisé (et éventuellement argumenté) dans le rapport.

— Le choix du solveur de PL ou PLNE est laissé libre (par exemple GLPK ou PYOMO ou ...) ; il sera précisé (et éventuellement argumenté) dans le rapport.

Questions et barème sur 20 points :

1 point Modéliser le problème restreint sous forme de PLNE. Rendu sous teide avant vendredi 2/10 20h (après la séance 3) ; correction séance 4.
Bonus : on pourra écrire (mais ce n'est pas obligatoire) un programme `computePatchOpt-plne` utilisant un solveur de PLNE (voir rendu final).

1 point Modéliser le problème général par équation de Bellman. Rendu sous teide avant vendredi 6/11 20h (après la séance 7) ; correction séance 8.

3 points Résolution par programmation dynamique : écrire un programme exécutable `computePatchOpt` qui résout le problème général par programmation dynamique. Le programme est noté sur 2 points (clarté et efficiacité) ; sa documentation sur 1 point.

8 points Le rendu final doit être déposé sous teide avant le 27/11/2015 ; ce rendu `TPAOD.tgz` est l'archive [2] d'un répertoire `TPAOD` (cf contenu en fin d'énoncé) incluant notamment un `Makefile`, les sources pour construire l'exécutable `bin/computePatchOpt` et sa documentation `doc/index.html` ainsi qu'un rapport de 4 pages maximum `rapport/rapport.pdf`. Ce rapport doit suivre le modèle fourni ; il décrit les choix d'implémentation et répond dans l'ordre aux questions suivantes :

1. (1 point) explication brève du principe de votre programme en précisant la méthode implantée (récursive, itérative) ;
2. (3 points) analyse du coût théorique de votre programme en fonction des nombres de lignes n_1 et n_2 (resp. de caractères c_1 et c_2) des deux fichiers F_1 et F_2 en entrée :
   — nombre d'opérations en pire cas ;
   — place mémoire requise ;
   — analyse des défauts de localité sur le modèle CO (argumenter en s'appuyant sur le programme) ;
3. (2 points) compte rendu d'expérimentation :
   i. la description synthétique de la machine et des conditions dans lesquelles les mesures ont été effectuées (pour permettre la reproductibilité des mesures) ;

2. cette archive peut être générée par la commande Unix : `tar cvfz TPAOD.tgz TPAOD`

ii. un tableau donnant les temps d'exécution mesurés pour chaque benchmark indiqué (temps minimum, maximum et moyen sur 5 exécutions) ;

iii. une réponse justifiée à la question : les temps mesurés correspondent ils à votre analyse théorique (nombre d'opérations et défauts de cache) ?

4. (1 point) Quelle méthode utiliseriez vous pour résoudre le problème si le coût d'un patch était défini comme sa taille en nombre d'octets (i.e. taille du fichier patch) ?
On ne demande pas de programme ni d'algorithme, mais juste de préciser le principe de la résolution choisie (parmi celles vues en cours) ; on précisera soit les équations de base pour la résolution, soit les modifications à apporter à votre programme si il peut être adapté à cette fonction de coût.
5. (1 point) pour la qualité globale du rapport (présentation, concision et clarté de l'argumentation).

7 points Évaluation automatique : un programme automatique testera la correction des programmes. Tous les tests échoués recevront automatiquement la note 0 (par exemple si le nom ou la spécification du programme ne sont pas respectés, ou si le patch généré est invalide ou pas de coût minimal, etc).
NB La moitié de ces tests (3.5 points) admettent un patch restreint comme solution optimale.

Bonus 1 point Écriture d'un programme exécutable `computePatchOpt-plne` qui résout le problème restreint et comparaison du résultat obtenu et des performances expérimentales avec la résolution par programmation dynamique : cette comparaison doit alors figurer dans le rapport, en dernière question traitée.

Contenu du répertoire TPAOD : règles à respecter strictement pour l'évaluation automatique. Votre dépôt final teide doit être une archive `TPAOD.tgz` qui suit le modèle de l'archive squelette fournie. À la racine de ce répertoire doivent figurer :

— un fichier `Makefile` standard qui sera exécuté par le programme d'évaluation en premier via la commande

```
make -f TPAOD/Makefile
```

et qui peut donc être utilisé pour générer les exécutables, la documentation et le rapport demandés ;

— un répertoire `src` contenant l'ensemble des sources développées ;

— un répertoire `bin` contenant le fichier exécutable `computePatchOpt` ; et, si vous l'avez réalisé, le programme `computePatchOpt-plne` ; vous pouvez mettre d'autres fichiers, mais ils ne seront pas appelés directement par le programme d'évaluation automatique. L'exécutable `computePatchOpt` respecte la spécification suivante :

```
computePatchOpt F1 F2
```

écrit sur la sortie standard un patch de coût minimal transformant via `applyPatch` le fichier `F1` en `F2`.
Le programme optionnel `computePatchOpt-plne` a la même spécification mais génère un patch restreint de coût minimal ;

— un répertoire `rapport` qui contient un fichier `rapport.pdf` correspondant au rapport en `pdf` (format exigé, 4 pages maximum). Il est recommandé d'utiliser LaTeXen complétant le fichier fourni ; mais vous pouvez utiliser un autre logiciel de traitement de texte à condition de respecter le squelette (pdf) de rapport fourni :

— un répertoire `doc` avec en particulier un fichier html `index.html` contenant la documentation des programmes développés (ce fichier peut être généré par exemple avec doxygen).

Le programme d'évaluation automatique (qui sera lancé depuis un répertoire plus haut dans la hiérarchie) exécute d'abord la commande : `make -f TPAOD/Makefile`
Après exécution de cette commande, les fichiers `TPAOD/bin/computePatchOpt`, `TPAOD/rapport/rapport.pdf` et `TPAOD/doc/index.html` doivent exister. Le programme d'évaluation lance ensuite une séquence de commandes de la forme

```
TPAOD/bin/computePatchOpt F1 F2 > patchF1F2 ; verifPatch F1 F2 patchF1F2
```

où la commande (non fournie) `verifPatch` vérifie la validité [3] et l'optimaité en coût du patch `patchF1F2`.

3. En particulier il vérifie qu'après la commande : `applyPatch patchF1F2 F1 > F2patched ; diff F2 F2patched > erreurs` le fichier `erreurs` est vide.