Rapport de TP 4MMAOD : Génération de patch optimal

DUHART Claudia (IF Groupe 1)
MARTINEZ Cléa (MMIS Groupe 1)

30 novembre 2015

1 Principe de notre programme (1 point)

Nous avons choisi d'implémenter la méthode itérative en stockant tous les coûts $V(i,j)$ dans un tableau.
Dans un premier temps, on parcourt les fichiers F_1 et F_2 pour stocker dans deux listes chaînées (puis des tableaux) les données des fichiers ce qui permet entre autres de sauvegarder le nombre de caractères associé à chaque ligne. Ensuite, on calcule les coûts $V(i,j)$ qu'on stocke dans un tableau, ainsi que les opérations nécessaires pour arriver aux lignes i et j dans un tableau Top.
Un parcours de ces tableaux nous permet ensuite, via de nouveaux tableaux, de reconstruire le patch de coût minimal, et de l'afficher sur la sortie standard.

2 Analyse du coût théorique (3 points)

2.1 Nombre d'opérations en pire cas :

Le nombre d'opérations en pire cas est en $O(n_1 n_2 + c_1 c_2)$

Justification :
La fonction *initList* appelée réalise 4 affectations : $O(1)$
La fonction *getNbCar* fait un accès tableau et une affectation : $O(1)$
La fonction *minimum* fait appel à *compareLignes* qui fait le nombre de caractères de la ligne fois un test d'égalité.
La fonction *listLines* (appelée deux fois au total) fait n appels à *addNext* (fonction en $O(1)$) : $O(n)$
La vérification de l'allocation des tableaux *Tab/Top* fait n_1 tests d'égalité (car les tableaux sont de taille n_1).
Le remplissage des tableaux *Tab et Top fait* $n_1 \times n_2$ *(double boucle) appels à la fonction minimum :* $O(c_1 \times c_2)$.
La fonction build path fait dans le pire des cas $O(max(n_1, n_2))$ affectations.
La fonction *printpath* fait au maximum $O(n_1+n_2)$ tests d'égalité (car la taille du tableau path vaut $= n_1+n_2+2$)

2.2 Place mémoire requise :

La place mémoire requise est en $O(n_1 \times n_2)$.

Justification : On utilise deux tableaux de taille $(n_1 + 1) \times (n_2 + 1)$ (*Tab* et *Top* pour stocker les $V(i,j)$ et les opérations).
On utilise également un tableau de taille $n_1 + n_2 + 2$ pour stocker les opérations conduisant à la construction d'un patch de coût minimal, puis un tableau contenant les minimums des colonnes de *Tab* de taille $n_2 + 1$
Enfin, on stocke dans deux listes chainées(puis des tableaux)de tailles respectives $n_1 + 1$ et $n_2 + 1$ les lignes des fichiers d'entrée et de sortie (de structure *struct line*).

2.3 Nombre de défauts de cache sur le modèle CO :

Notons Z la taille du cache, L la taille des lignes de cache.
Si $Z < max(n_1, n_2)/L$, l'initialisation des tableaux *Tab* et *Top* provoque environ $2(n_1 + n_2)/L$ défauts de cache, sinon 0 défauts de cache.

Si $Z > 2(n_1 n_2)/L$, alors le remplissage des tableaux ne provoque pas de défauts de cache. Sinon, il en provoque $O(n_1 n_2/L)$.
Si $Z > n_1 + n_2 + 2$ le remplissage du tableau *path* ne provoque pa de défaut de cache, sinon il y en a $O((n1 + n2)/L)$.

3 Compte rendu d'expérimentation (2 points)

3.1 Conditions expérimentaless

3.1.1 Description synthétique de la machine :

Processeur : inter CORE I5vPro
Fréquence : 800.292 MHz
Mémoire : 8Go
Système d'exploitation : Ubuntu
Seul firefox était ouvert au moment des mesures pendant les tests.

3.1.2 Méthode utilisée pour les mesures de temps :

Les mesures de temps ont été effectuées avec la lignes suivante :
`time ./bin/computePatchOpt benchmark/benchmark_n/source benchmark/benchmark_n/target > patch`
Le temps retourné était donc en seconde, nous avons retenu la partie user. Un même test a été exécuté 5 fois de suite pour un même benchmark.

3.2 Mesures expérimentales

	coût du patch	temps min	temps max	temps moyen
benchmark1	2540	0.045	0.096	0.072
benchmark2	3120	0.195	0.255	0.284
benchmark3	809	0.522	0.653	0.584
benchmark4	1708	1.272	1.331	1.309
benchmark5	7553	2.262	2.793	2.387
benchmark6	37027	12.339	12.718	12.496

FIGURE 1 – Mesures des temps minimum, maximum et moyen de 5 exécutions pour les 6 benchmarks en seconde.

3.3 Analyse des résultats expérimentaux

A chaque benchmark, le nombre de lignes augmente, par conséquent le nombre d'opération (dépendant du nombre de lignes et du nombre de caractères des fichiers source et cible) augmente également. De plus, les structures de données utilisées pour stocker les données sont forcément de plus grande taille, et donc le nombre de défauts de cache augmente de manière exponentielle ! ! Les résultats expérimentaux présentés dans le tableau ci-dessous montrent bien cette augmentation exponentielle.

Benchmark	LLi misses	LLd misses
benchmark1	1 123	140 872
benchmark2	1 130	6 832 521
benchmark3	1 122	16 728 779
benchmark4	1 128	31 335 061
benchmark5	1 131	49 768 505
benchmark6	1 134	217 237 598

FIGURE 2 – Nombres de défauts de cache expérimentaux pour chaque benchmark

Mesures effectuées avec cachegrind, selon la commande suivante :
`valgrind --tool=cachegrind computePatchOpt F1 F2`

4 Question : et si le coût d'un patch était sa taille en octets ? (1 point)

Si le coût d'un patch était sa taille en octet, on utiliserait la même méthode que dans notre programme, on changerait seulement l'équation de Bellman pour modéliser le problème (et donc la fonction *mimimum*) de notre programme.

La nouvelle équation de Bellman serait la suivante :

$$\begin{aligned} V(n,m) = min \ \ (\ \ & V(n-1,m-1), \\ & 4 + L(F_2(m)) + V(n,m-1), \\ & 4 + L(F_2(m)) + V(n-1,m-1) \\ & 4 + V(n-1,m), \\ & min_{i\in[2,n]}(6 + V(n-i,m)) \quad) \end{aligned}$$

avec $L(F_2(j))$ le nombre de caractères de la ligne j du fichier F_2, en comptant le caractère de fin de ligne "$\backslash n$"

Si l'on recopie une ligne, alors ça ne coûte rien et on est ramené au même problème, sur des fichiers de longueur $n-1$ et $m-1$.
Si l'on ajoute une ligne, on écrit dans le patch $4 + L(F_2(j))$ caractères et on est ramené au même problème sur des fichiers de longueur n et $m-1$.
Si l'on substitue une ligne, on écrit dans le patch $4 + L(F_2(j))$ caractères et on est ramené au même problème sur des fichiers de longueur $n-1$ et $m-1$.
Si l'on supprime une ligne, on écrit dans le patch 4 caractères et on est ramené au même problème sur des fichiers de longueur $n-1$ et m.
Enfin si l'on supprime i lignes ($i \leq n$), on écrit dans le patch 6 caractères et on est ramené au même problème sur des fichiers de longueur $n-i$ et m.

5 Optimisations possibles

Tout d'abord, nous stockons les lignes dans 2 structures différentes : liste chaînée puis tableau. Cela nous permet un accès plus facile aux valeurs mais augmente la place mémoire nécessaire.
Le stockage de tous les coûts dans notre tableau Tab n'est pas indispensable. En effet, lorsque l'on s'intéresse à une case nous n'avons besoin que de la ligne associée ainsi que la précédente pour les calculs. Ainsi, nous aurions donc pu diminuer la place occupée par ce tableau.