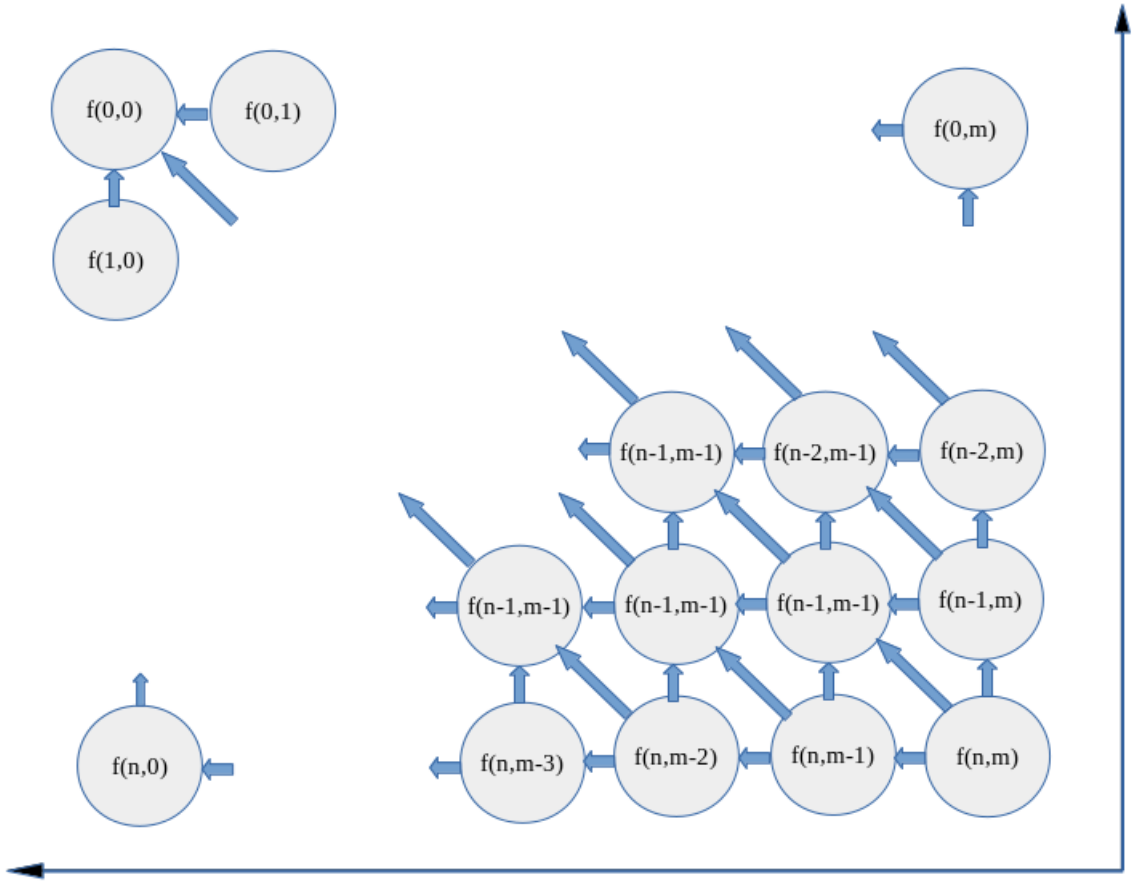
Rapport de TP 4MMAOD : Patch optimal entre deux fichiers

BENJELLOUN Hamza (groupe ISI 2)
AIT LAHMOUCH Nadir (groupe ISI 2)

11 novembre 2019

1 Graphe de dépendances (1 point)

Le graphe de dépendances des appels :



On remarque à partir du graphe d'appel une redondance importante qu'on peut supprimer en initialisant une matrice qui calcule au préalable les valeurs des noeuds avant de faire les appels récursifs. Cette méthode est loin d'être gênante vu que tous les noeuds (appels) seront sollicités, leur calcul donc est nécessaire.

En pseudo code, une méthode itérative serait donc :

def mainFunction(parameters...) :
"Initialisation de la matrice"
for index-lines-target in range(n) :
for index-lines-source in range(m) :
memValues[index-lines-target][index-lines-source] = value
"La valeur à chaque itérarion est calculée à partir des valeurs déjà exixtantes dans la matrice et la fonction minimum, comme cela est présenté dans le graphe d'appels"
"On peut appeler désormais la fonction récursive pour écrire dans le patch"
WritePatchRecursif(memValues,....)

Cette fonction récursive prendra aussi en paramètre deux int i et j pour signifier où on en est dans l'écriture du patch est dans le graphe d'appel. A chaque appel, si les paramètres ne vérifient les conditions d'arrêt (i=0 ou j=0 ou (i=0 j=0)), on recalculera les trois valeurs suceptibles à être égales memValue[i][j] et avec des tests d'égalité on réappelle la fonction récursive avant d'écrire dans le patch l'op correspondante.

2 Principe de notre programme (1 point)

Notre programme se décompose en deux fonctions principales utilisant d'autres fonctions annexes.

Une fonction "algo-iter" qui commence par récupérer les fichiers source et destination après un traitement avec des fonctions implémentées dans notre fichier outils.c. Elle remplit ensuite une matrice avec les valeurs des noeuds nécessaires pour un remplissage optimal du patch. Et enfin elle appelle la deuxième fonction "write-patch-recur" dans une double boucle lui donnant en paramètre la matrice et deux indices i et j qui s'incrémentent au fur et à mesure.

La deuxième fonction quant à elle écrit récursivement dans le patch selon les valeurs de la matrice. Elle suit ainsi le chemin optimal pour l'implémenter.

3 Analyse du coût théorique (3 points)

3.1 Nombre d'opérations en pire cas :

Notre programme commence par transformer le fichier source et target en tableau de caractères avec mmap puis on le parcours avec la fonction lines() qui a $O(C)$ comme complexité (avec C le nombre de caractéres dans le fichier).

On remplit ensuite la matrice qui memorise les valeurs nécessaires pour construire un patch optimal et cette étape coûte $4(n_1+1)(n_2+1)$ en complexité.

On appelle dès lors la fonction récursive write-patch-recur(). Cette fonction prend en pire de cas $(n_1+1)(n_2+1)$: C'est le cas qui correspond a la destruction de toutes les lignes du fichier source puis l'ajout de toutes les lignes du fichier target ou le contraire.

Donc théoriquement notre algorithme a un coût de :

$$Coût = C_1 + C_2 + 5(n_1+1)(n_2+1)$$

3.2 Place mémoire requise :

La mémoire théorique requise pour le bon fonctionnement de notre algorithme correspond à la mémoire prise par la matrice.

$$MemoireRequise \quad = \quad 4n_1m_1 \quad octets$$

4 correspond à la taille des int (sizeof(int) = 4 octets).

Ainsi, notre coût mémoire est en $O(nm)$

3.3 Nombre de défauts de cache sur le modèle CO :

Le calcul théorique des défauts de cache donne :

En prenant le modèle de cache simplifié CO de taille Z avec une taille L pour une ligne du cache, on trouve que le nombre de défauts de caches est :

si $n_1 < Z$:

$$\frac{n_1 n_2}{L}$$

Sinon le nombre devient :

$$n_1 n_2$$

Les défauts de cache calculés avec cachegrind :

Pour d'abord le test benchmark00 on trouve 36 539 défauts de cache :

```
==31074== I   refs:      130,813,025
==31074== I1  misses:          1,299
==31074== LLi misses:          1,275
==31074== I1  miss rate:        0.00%
==31074== LLi miss rate:        0.00%
==31074==
==31074== D   refs:       40,652,301  (40,253,817 rd   + 398,484 wr)
==31074== D1  misses:         13,402  (     6,115 rd   +   7,287 wr)
==31074== LLd misses:          9,679  (     2,619 rd   +   7,060 wr)
==31074== D1  miss rate:         0.0% (       0.0%     +     1.8%   )
==31074== LLd miss rate:         0.0% (       0.0%     +     1.8%   )
==31074==
==31074== LL refs:            14,701  (     7,414 rd   +   7,287 wr)
==31074== LL misses:          10,954  (     3,894 rd   +   7,060 wr)
==31074== LL miss rate:          0.0% (       0.0%     +     1.8%   )
```

Tandis que pour le test benchmark01 on a 47 473 666 132 défauts de cache :

```
==31194== I   refs:      42,448,008,739
==31194== I1  misses:             1,341
==31194== LLi misses:             1,330
==31194== I1  miss rate:           0.00%
==31194== LLi miss rate:           0.00%
==31194==
==31194== D   refs:      13,561,110,867  (13,371,379,737 rd   + 189,731,130 wr)
==31194== D1  misses:     4,737,871,887  ( 4,733,115,122 rd   +   4,756,765 wr)
==31194== LLd misses:         4,745,147  (        32,641 rd   +   4,712,506 wr)
==31194== D1  miss rate:           34.9% (          35.4%     +         2.5%   )
==31194== LLd miss rate:            0.0% (           0.0%     +         2.5%   )
==31194==
==31194== LL refs:        4,737,873,228  ( 4,733,116,463 rd   +   4,756,765 wr)
==31194== LL misses:          4,746,477  (        33,971 rd   +   4,712,506 wr)
==31194== LL miss rate:             0.0% (           0.0%     +         2.5%   )
```

4 Compte rendu d'expérimentation (2 points)

4.1 Conditions expérimentaless

Les mesures se font sur une machine de l'ensimag. Pour calculer le temps que prend chaque benchmark, on utilise deux variables start et end de type clock() de <time.h>. Le temps final est calculé par la formule "(end - start) / CLOCKS-PER-SEC", CLOCKS-PER-SEC pour le convertir en secondes.

4.1.1 Description synthétique de la machine :

Caractéristiques de la machine sur laquelle on exécute les tests :
Machine de l'ensimag : Precision 3430
Système d'exploitation : CentOS Linux 7
Processeur : Intel® Core™ i5-8500 CPU @ 3.00GHz 6
Mémoire : 31.1 GiB

4.1.2 Méthode utilisée pour les mesures de temps :

Pour calculer le temps que prend chaque benchmark, on utilise deux variables start et end de type clock() de <time.h>. Le temps final est calculé par la formule "(end - start) / CLOCKS-PER-SEC", CLOCKS-PER-SEC pour le convertir en secondes.

Les 5 exécutions ont été faites 5 fois de suite pour chacun des tests.

4.2 Mesures expérimentales

Compléter le tableau suivant par les temps d'exécution mesurés pour chacun des 6 benchmarks imposés (*temps minimum, maximum et moyen sur 5 exécutions*)

	coût du patch	temps min	temps max	temps moyen
benchmark_00	10474	0.00	0.01	0.005
benchmark_01	12699	1.94	1.96	1.958
benchmark_02	3626	86.83	87.26	87.042
benchmark_03	-	-	-	-
benchmark_04	-	-	-	-

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

4.3 Analyse des résultats expérimentaux

Les résultats expérimentaux traduisent bien ce qu'on a prévu théoriquement, si on augmente les nombres de lignes du fichier source ou target, on commet plus de défauts de cache en remplissant et en parcourant la matrice (car théoriquement le nombre de défauts de cache est équivalent à $n_1 n_2$). Aussi, et de même on prend plus de temps.

5 Coût du patch en octet (1 point)

Si on définit le coût de patch comme étant sa taille on octet, il faudrait tout simplement changer les équations de bellman et laisser le même algorithme.
Les équations de Bellman deviennent dans ce cas :

$$f(i,j) = min \begin{cases} f(i-1,j-1) + C(i,j) \\ f(i-1,j) + 3 + log_{10}(i) + L(A,i)) \\ f(i,j-1) + 3 + log_{10}(i) + L(B,j) \end{cases}$$

avec C est :

$$C(i,j) = \begin{cases} 0 \quad si\ A[i] = B[j] \\ 3 + log_{10}(i) + L(B,j) + L(A,i) \quad sinon \end{cases}$$

et pour les conditions initiales :

$$\begin{cases} f(0,0) = 0 \\ f(0,j) = \sum_{k=1}^{j} (2 + log_{10}(i) + L(B,k)) \\ f(i,0) = \sum_{k=1}^{j} (2 + log_{10}(i) + L(A,k)) \end{cases}$$