4MMAOD6 - TD 6 : Dernière séance - Révisions

Résumé du cours : méthodes vues en cours pour écrire des programmes (récursifs et itératifs) d'optimisation discrète performants en analysant leur performance (nombre d'opérations, localité mémoire) :

Mémoisation : identification des appels redondants dans un programme récursif ; élimination des redondances par stockage du résultat de chaque appel réalisé dans une table.

Branch&Bound : exploration arborescente de l'ensemble des solutions admissibles pour mémoriser et mettre à jour la meilleure rencontrée depuis le début de l'exploration. Le programme est itératif et utilise une collection des nœuds restants à explorer (boucle tant que la collection est non vide) :

- ***initialisation*** : pré-calcul de la valeur d'une solution (considérée bonne) et initialisation de la collection avec les racines de l'arbre d'exploration ;
- ***branch*** : fonction d'exploration, retrait et ajout de nœuds dans la collection. Un compromis file de priorité (meilleur d'abord) tant qu'on tient en mémoire puis pile LIFO ensuite permet de réaliser une exploration en mémoire locale ;
- ***bound*** : fonction dite d'évaluation optimiste qui permet d'élaguer un nœud dont les fils ne peuvent pas conduire à une solution améliorant la meilleure solution rencontrée ; en programmant cette fonction, on réalise un compromis entre la rapidité de cette fonction et sa capacité à élaguer beaucoup de nœuds.
- ***anytime*** : on peut stopper à tout instant la boucle ; alors l'écart entre la meilleure solution rencontrée et la meilleure valeur optimiste des nœuds restants à explorer dans la collection permet de majorer l'écart de la solution actuelle avec la valeur d'une solution optimale.

Équation de Bellman : équation récursive (avec conditions initiales) caractérisant la valeur d'une solution optimale à un problème à partir des valeurs optimales de sous problèmes (approche *top-down*).

Programmation dynamique : programme calculant en 2 temps une solution optimale de l'équation de Bellman :

1. ***programmation directe de l'équation de Bellman*** pour calculer la valeur d'une solution optimale, tout en stockant dans une table le choix optimal de chaque sous problème utilisé. Le programme s'écrit assez naturellement :
   - ***récursif*** (top-down) avec mémoisation pour éliminer les appels redondants
   - ***itératif*** (bottom-up) en choisissant l'ordre de résolution des sous problèmes pour éviter les redondances
2. ***remontée d'une solution optimale*** : parcours de la table des choix optimaux pour construire la séquence de choix optimaux à effectuer (chemin optimal) : ce parcours est nécessairement top-down du problème cible à un sous problème terminal (condition d'arrêt).

Localité mémoire : l'écriture de programmes exploitant la hiérarchie mémoire et le parallélisme est critique pour les performances. Partant d'un programme itératif séquentiel :

- ***analyse des défauts de cache*** (*cache miss*) sur modèle simplifié CO (taille de cache Z et de bloc L).
- ***programme itératif par blocs (cache-aware)*** : structuration des itérations en blocs multidimensionnels pour favoriser les accès en cache (*cahe hit*) : on regroupe en blocs les instructions qui accèdent à des données contigües (localité spatiale) ou aux même données (localité temporelle) en choisissant la taille des blocs de données pour que les données accédées dans un bloc de calcul tiennent dans le cache. On obtient des programmes itératifs par blocs (ou super-pas) : chaque bloc est un nid de boucles qui ne génère pas de défaut de cache.
  Cette technique permet souvent aussi d'exhiber des blocs indépendants pouvant être traités en parallèle.
- ***programme récursif par découpe en blocs (cache-oblivious)*** : on programme une découpe récursive des blocs pour obtenir des blocs plus petits à chaque découpe : ainsi, au bout de quelques étapes de découpe récursive, le bloc tient dans le cache et le nombre de défauts de cache est analogue à la programmation cache-aware avec en plus l'avantage d'un programme indépendant de L et Z. On obtient ainsi un programme plus portable et qui de plus fonctionne à chacun des niveaux de la hiérarchie, Un autre intérêt est que le programme met en évidence un parallélisme récursif qui peut être exploité efficacement par des interfaces de programmation parallèle standard (comme OpenMP-4 avec tâches récursives et dépendances ou Cilk par exemple). Important : le règlage du seuil de découpe récursive (en dessous duquel on utilise un programme itératif) permet de limiter le surcout arithmétique.

Application à la programmation efficace d'algorithmes d'optimisation discrète : la programmation dynamique se prête bien à une programmation *cache-oblivious* ; partant d'un programme itératif issu de l'équation de Bellman, on écrit un programme :

- récursif qui découpe l'espace d'itération (ie le tableau des choix optimaux) en sous bloc (généralement sur la plus grande dimension) jusqu'à un seuil d'arrêt (pour éviter le surcout)
  - en dessous du seuil, les blocs sont traités séquentiellement de manière itérative (le seuil est suffisamment petit pour tenir en cache et grand pour amortir le surcout arithmétique des appels récursifs)
    - le parallélisme des appels récursifs peut être exploité grâce à des interfaces de programmation parallèles standard (comme OpenMP avec les tâches récursives, en choisissant un seuil d'arrêt pour le parallélisme).

1 Branch&Bound

Question 1 On arrête en cours d'exécution un algorithme de Branch&Bound après quelques fractions de seconde de calcul. La figure 1 représente l'arbre de branchement : dans cette figure, la valeur sous le nœud indique son évaluation optimiste ; la valeur au dessus d'un nœud indique la meilleure valeur jusque là (ie la valeur de la solution courante).

1. Est-ce un problème de MIN ou de MAX ?
2. Quelle est la valeur de la solution courante ?
3. Quelle est la valeur de la solution initiale ? Que cela signifie-t-il ?
4. Parmi les nœuds actifs 3, 5, 6, 7 et 8, quels sont ceux restants à explorer et pourquoi ?
5. Quel(s) nœud(s) explorer en priorité et pourquoi ?
6. Quel est l'écart maximal entre la solution courante et une solution optimale ?

1. Problème MIN (valeur solution courante décroissante, évaluations optimistes croissantes)
2. valeur de la solution courante = 31
3. valeur de la solution initiale ? $i + \infty$; cela signifie qu'il n'y avait pas de solution initiale (valeur par défaut)
4. Parmi les nœuds actifs 3, 5, 6, 7 et 8, quels sont ceux restants à explorer et pourquoi ? Les noeuds 3 et 5 doivent encore être explorés car ils sont susceptibles de contenir des solutions meilleures que la courante (31). Le noeud 6 peut être élagué par borne, le noeud 7 par optimalité et le noeud 8 par faisabilité.
5. Le nœud 5 car c'est celui de meilleure évaluation (27) et le plus profond.
6. La solution courante de valeur 31 est à 4 de la meilleure évaluation optimiste 27. La garantie de performance est de $\frac{31-27}{27} = 14.81\%$

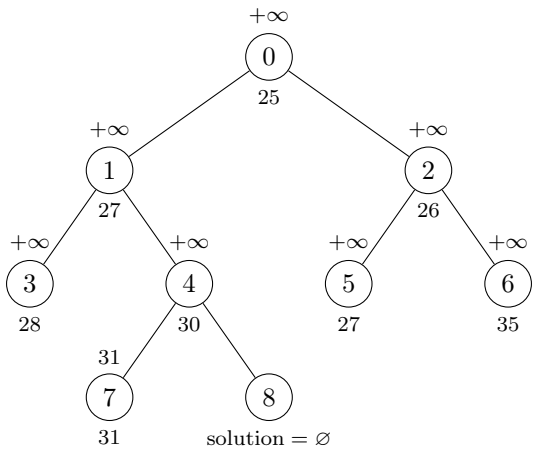

FIGURE 1 – L'arbre de recherche associé à notre instance du problème de placement de sites de secours.

2 Programmation dynamique

Un système est formé de N composants en série. Pour $1 \leq i \leq N$, le composant i a une probabilité p_i d'être en panne. De façon à augmenter la fiabilité du système, le composant i peut être *répliqué* : au lieu d'un seul exemplaire du composant i, on en met $n_i \geq 1$ en parallèle. Ainsi, la probabilité que le système fonctionne est $\prod_{i=1}^{N}(1 - p_i^{n_i})$. La figure 2 montre un exemple d'un tel système, dans lequel $N = 3$.

Chaque composant i a de plus un coût entier $c_i \geq 1$.

On se donne un budget maximal $B \geq \sum_{i=1}^{N} c_i$ pour construire le système ; B est un un entier.

On veut calculer par programmation dynamique le nombre de chaque composant i pour construire le système qui a la plus grande probabilité de fonctionner sachant que le coût total du système ne doit pas dépasser le budget B.

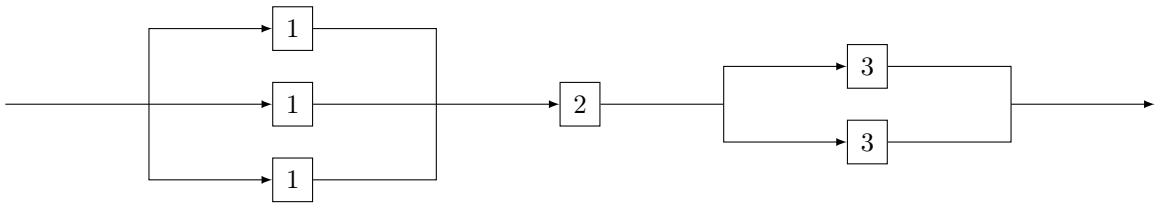

FIGURE 2 – Exemple de configuration de système à 3 composants en série (6 composants au total). La probabilité que ce système fonctionne est $(1-p_1^3)\times(1-p_2)\times(1-p_3^2)$.

Question 2 Soit $C(k,b)$ le taux de fiabilité maximal que l'on peut atteindre avec un budget $b \leq B$ en n'utilisant que des composants choisis parmi les k premiers (en autorisant bien entendu la réplication de certains d'entre eux).

1. Justifier brièvement la formule de récurrence suivante, et qu'il s'agit d'une équation de Bellman :

$$C(k,b) = \max_{1\leq x\leq \left\lfloor \frac{b}{c_k}\right\rfloor} \Big((1-p_k^x)\times C(k-1, b-x\cdot c_k) \Big).$$

2. Comment initialisez-vous la récurrence ?

1. *Justification de la formule :* sous-structure optimale.
   Supposons que le système à n composants est optimal avec un budget B et en répliquant x fois le composant n. Alors il existe une solution optimale où le sous système formé par les $n-1$ premiers composants est aussi optimal avec un budget $B - x.c_n$ (car sinon, on améliorerait la solution en remplaçant ce sous-système par un autre optimal).
2. *Conditions initiales.* $C(0,b) = 1$ si $b \geq 0$ et $C(0,b) = 0$ si $b < 0$.

Question 3 Écrire un algorithme itératif (en pseudo-code) qui calcule le coût $C(N,B)$ d'une solution optimale pour N composants et un budget de B en faisant $O\left(N\cdot B^2\right)$ opérations. Pensez à commenter votre code et à justifier son coût en nombre d'opérations. (NB on suppose disposer d'une opération `puissance(a,b)` qui calcule a^b en temps constant.)

On suppose $B \leq \sum_{i=1}^{n} c_i$ car sinon il n'y a pas de solution.
Le calcul itératif du tableau $C[0..n, 0..B]$ est direct :

```
for (b = 0 ;  b <= B; i++)  C[0,b] = 1 ; // Initialisation
for (k = 1 ;  k <= N; k++)  // Boucle sur les composants
   for (b = 0 ;  b <= B; b++)  // Boucle sur le budget
      if (b <  c[k])  C[k,b] = 0 ;
      else {
         opt = (1 - p[k]) * C[k-1, b - c[k] ] ;
         for ( x =2; x <=  b / c_k ; x++)
            if ( opt < (tmp=(1-puissance(p[k],x))*C[k-1, b-x*c[k]])) {
               opt = tmp ;
            }
         C[k,b] = opt ;
      }
```

Le nombre d'appels à puissance est $\sum_{k=1}^{N}\sum_{b=0}^{B}\frac{b}{c_k} \simeq \frac{B^2}{2}\sum_{k=1}^{N}\frac{1}{c_k} \leq NB^2$, donc le cout est $O(NB^2)$.

Question 4 Ecrire un algorithme (en pseudo-code) qui calcule pour $1 \leq i \leq N$, le nombre n_i d'exemplaires du composant i dans un système de fiabilité maximale pour un budget B : utilisez votre programme précédent sans le réécrire, en expliquant comment le modifier si besoin.

Il suffit d'avoir un tableau auxiliaire `choixOpt[0..n, 0..B]` où `choixOpt[k,b]` stocke n_k qui donne la fiabilité optimale avec un budget b et un circuit limité au composants 1 à k.
Pour le programme, on remplace le corps de boucle par :

```
      else {
         opt = (1 - p[k]) * C[k-1 ][ b - c[k] ] ;
         xopt= 1 ;
         for ( x =2; x <=  b / c[k]; ++x )
            if ( opt < (tmp=(1-puissance(p[k],x))*C[k-1, b-x*c[k]])) {
               opt = tmp ;
               xopt= x ;
             }
         C[k,b] = opt ;
         choixOpt[k,b] = xopt ;
      }
```

Le calcul du nombre de réplicats est très simple, en remontant à partir de $C[n, B]$:

```
  for (k=N, brem=B; k >0;  --k) {
     cout << "Nombre de composants " << k << "=" << choixOpt[k][brem] << endl;
     brem -=  choixOpt[k][brem] * c[k] ;
  }
```

Question 5 Préciser le coût total en nombre d'opérations (travail en notation Θ) et l'espace mémoire requis par votre programme pour calculer une solution optimale à partir des entrées.

Travail : un coût $O(N)$ pour la remontée soit $O(B^2N)$ en tout.
Mémoire : $N(B+1)$ entiers pour calculer le tableau `choixOpt` des indices ; et $N.(B+1)$ entiers pour le tableau C (mais on peut aussi calculer en place C dans un seul vecteur de taille N).

Question 6 On considère un cache de taille Z auquel on accède avec une taille L de ligne de cache (modèle CO présenté en cours). Les 3 questions suivantes sont indépendantes.

1. On suppose ici $Z \gg N \cdot B$ entiers (par exemple, $Z > 10 \cdot N \cdot B$). Combien de défauts de cache (en ordre de grandeur, notation Θ) effectue votre programme ?
2. On suppose maintenant $Z \ll B$. Combien de défauts de cache en pire cas (notation Θ) effectue votre programme ?
3. Comment programmeriez-vous l'algorithme pour faire $O\left(\frac{NB}{L} + \frac{NB^2}{ZL}\right)$ défauts de cache ? Expliquer brièvement le principe et analysez le nombre de défauts de cache.

1. si $Z \gg NB$: les 3 tableaux contigus en mémoire $C[k-1][0..B]$ et $C[k][0..B]$ et aussi choixOpt$[k][0..B]$ tiennent en cache ensemble ; on a donc $O\left(\frac{NB}{L}\right)$ défauts de cache. Idem, si $Z \gg BN$.
   Remarque : en ne stockant qu'un ou 2 vecteurs pour C, on ne fait que $(N + O(1))\frac{B}{L}$ défauts.
2. si $Z \ll B$: on a $O\left(\frac{B}{L}\right)$ défauts pour calculer chaque élément (boucle interne), donc $O\left(\frac{NB^2}{L}\right)$
3. On fait un calcul par blocs de taille K similaire aux boucles type tri par insertion vues en TD. Pour chacune des N lignes, la mise à jour du bloc i ($i = 0 \dots B/K$) requiert en pire cas (si tous les $c_k = 1$ par exemple) d'accéder chacun des premiers blocs j de la ligne précédente pour $0 \leq j \leq i$; chaque mise à jour du bloc i avec le bloc j requiert $\frac{2K}{L}$ défauts. D'où $\sum_{k=1}^{N} \sum_{i=0}^{B/K} \sum_{j=0}^{i} 2\frac{K}{L} \simeq \frac{NB^2}{KL}$ défauts de cache, On choisit donc $K \simeq Z/2 - O(1)$ (si on fait un calcul par lignes) pour que deux blocs tiennent en cache et on obtient le résultat avec en plus les $\frac{NB}{L}$ défauts incontournables sur choixOpt.
   On peut aussi faire une programmation cache-oblivious en coupant récursivement en 2 le plus grand des deux blocs (ou même en faisant une découpe bi-dimensoionnelle sur B et N qui en plus dégage du parallélisme).

3 Cache et localité

On considère un cache de taille Z mots, accèdé par lignes de cache de taille L mots (modèle CO). Soit $A[n][m]$ une matrice de $n \times m$ mots stockée dans un tableau contigu en mémoire, ligne après ligne (comme en C, Java ou Python) : pour $0 \leq i < n$ et $0 \leq j < m$, $A[i][j]$ est stocké à l'indice $i \times m + j$ du tableau.

Question 7 Le programme suivant remplit le tableau S (préalloué de taille n) avec le max de chaque ligne : pour $0 \leq i < n$, $S[i] = \max_{j=0}^{m-1} \{A[i][j]\}$. Combien engendre-t-il de défauts de cache ?

```
for (int  i=0; i<m ; ++i)  {
  S[i] = A[i][0] ;
  for (int j=1; j<n ; ++j)
    if (S[i] < A[i][j]) S[i] = A[i][j];
}
```

La ligne de cache qui accède $S[i]$ reste en cache et la matrice est parcourue de manière contiguë. Le programme effectue nm comparaisons et $\frac{nm}{L} + \frac{n}{L} + O(1)$ défauts de cache.

Question 8 Écrire un programme qui remplit le tableau T (préallloué de taille m) avec le min de chaque colonne, i.e. pour $0 \leq j < m$, $T[j] = \min_{i=0}^{n-1} \{A[i][j]\}$; votre programme doit effectuer $O\left(\frac{nm}{L}\right)$ défauts de cache. Préciser le nombre de comparaisons effectuées et le nombre de défauts de cache.

```
      for (int j=0; j<m ; ++j) T[j] = A[0][j] ;
      for (int  i=0; i<n ; ++i)
        for (int j=0; j<m ; ++j)
          if (T[j] >  A[i][j] )  T[j] = A[i][j] ;
```

Le programme effectue $n(m-1)$ comparaisons et $\frac{nm}{L}$ défauts de cache sur A et $\frac{m}{L}$ défauts de cache sur T dans la bloucle d'initialisation et à chaque passage dans la boucle interne, soit $2\frac{nm}{L} + \frac{m}{L} = O(nm/L)$ défauts de cache.

Question 9 Comment calculeriez-vous ensemble les deux tableaux S et T pour faire, lorsque n et m sont grands, un nombre de défauts de cache inférieur à $(1+\varepsilon)\frac{nm}{L} + O\left(\frac{n+m}{L}\right)$ où ε est une constante qui tend vers 0 quand Z tend vers l'infini ; on ne demande pas le programme, seulement le principe.

Réponse attendue : cache aware. On parcourt A par blocs de taille $K \times K$ et S et I par blocs de taille K, avec K maximum tel que $K^2 + 2K + O(1) < Z$; soit $K \simeq \sqrt{Z}$.
Autre solution possible : cache oblivious : on peut aussi écrire un programme récursif qui met à jour S et T en découpant récursivement en deux la sous-matrice en entrée sur sa plus grande dimension, jusqu'à atteindre une dimension minimale $s \times s$ en dessous de laquelle on parcourt la sous-matrice en faisant la mise à jour des blocs correspondants de S et T. Le nombre de défauts de cache est le même (à $O\left(\log \frac{nm}{s^2}\right)$ près pour les défauts de cache sur la pile des appels) mais sans avoir à connaître Z.

Détails : Le nombre de défauts est inférieur à $\left\lceil \frac{nm}{K^2} \right\rceil \left(\left\lceil \frac{K^2}{L} \right\rceil + 2\left\lceil \frac{K}{L} \right\rceil + K + 2 + O(1) \right) = \frac{nm}{L}\left(1 + \frac{2+L}{\sqrt{Z}} + O\left(\frac{L}{Z}\right)\right) = (1+\epsilon)\frac{nm}{L}$ avec ϵ qui tend vers 0 quand Z augmente.

```
  for (int i=0; i<n ; ++i) S[i]= A[i][0] ;
  for (int j=0; j<m ; ++j) T[j]= A[0][j] ;
  for (int I=0; I<n ; I+=K )
    for (int J=0; J<m ; J+=K ) {
      int maxi = (n < I+K) ? n : I+K ;
      int maxj = (m < J+K) ? m : J+K ;
      for (int i=I; i<maxi ; ++j)
        for (int j=0; j<maxj ; ++j) {
          if (S[i] < A[i][j]) S[i] = A[i][j] ;
          if (T[j] >  A[i][j] )  T[j] = A[i][j] ;
        }
    }
```

4 Modélisation par équation de Bellman

Exercice des skieurs (TD3)