4MMAOD6 - TD 5 : Branch & Bound

Préambule : Synthèse du cours sur Branch&Bound. Soit A un ensemble fini, potentiellement de très grande taille. On considère le problème de minimisation (un problème de maximisation se traite de manière similaire) : *Trouver* $\bar{x}$ *qui atteint* $\min_{x\in A} f(x)$.
Un algorithme de *Branch&Bound* (ou *séparation - évaluation* en français) consiste à

1. *solution courante* s : partant d'une solution initiale $s \in A$ non nécessairement optimale, mettre à jour s jusqu'à prouver que $f(s)$ est optimale ;
2. *Branch* : explorer de manière arborescente A ; un nœud de l'arbre représente donc un sous-ensemble de A, donc un sous-problème ; un sous-ensemble de cardinal 1 est une feuille. Typiquement, l'opération Branch permet de parcourir les fils d'un nœud.
3. *Bound* : élaguer les nœuds correspondant à un sous ensemble $X \subset E$ dont on peut vérifier facilement que tous les éléments sont moins bons que $f(s)$; i.e. $\forall x \in X : f(x) \geq f(s)$. Pour cela, on utilise une fonction g dite d'*évaluation optimiste* qui prend en entrée un nœud n et retourne une valeur meilleure (donc inférieure pour un problème *Min*, supérieure pour un problème *Max*) que la valeur de f en toute feuille de la sous-arborescence de racine n.
   Autrement dit, g retourne un minorant (resp. majorant) de f pour un problème Min (resp. Max).

Pour le programmer, on utilise une collection E, initialisée avec les racines de l'arborescence (il peut y en avoir une ou plusieurs selon les cas) qui stocke à tout instant l'ensemble des nœuds restant à explorer. Le programme ci-dessous résout un problème **Min** f, la fonction d'évaluation optimiste g retournant un minorant de f :

```
Soit s ∈ A une solution initiale et z̄ := f(s) sa valeur;
E := { racines de l'arbre à explorer };
répéter
    Choisir P ∈ E et le supprimer de E  E := E \ {P} ;
    Évaluer P de manière optimiste : z = g(P);
    si Si on peut montrer z = min_{s∈P} f(s) (par ex. si P ne contient qu'un élément) alors
        mettre à jour z̄ := z et s si besoin
    ;
    sinon si z < z̄ alors
        Branch Décomposer P en k sous-problèmes P_1 ⊔ ··· ⊔ P_k = P;
        E = E ∪ {P_1, ..., P_k};
jusqu'à E = ∅;
```

B&B à Utopia (30' en CTD interactif)

La ville de Utopia possède 4 districts dont les populations sont données dans le tableau qui suit. On veut constituer une assemblée représentative de 11 élus, ce qui fait un représentant pour 100 habitants.
Le problème est de définir le nombre de représentants par district de façon que l'écart maximum de représentation par personne par rapport à l'idéal de un pour cent, soit minimum.
Dans le tableau suivant #idéal représente le nombre (fractionnaire) de représentants qu'il faudrait pour satisfaire la representation idéale par électeur notée : idéal p.p.

District	population	#idéal	idéal p.p.
A	120	1,20	0,01
B	155	1,55	0,01
C	360	3,60	0,01
D	465	4,65	0,01
	———	———	
total	1100	11	

Par exemple si on affecte 1, 1, 4 et 5 représentants aux districts A, B, C et D respectivement, les écarts par rapport à l'idéal sont respectivement de $0,01 - 1/120 = 0,00167$, $0,01 - 1/155 = 0,00355$, $4/360 - 0,01 = 0,00111$ et $5/465 - 0,01 = 0,00075$. Donc l'écart maximum est de $0,00355$.

Question 1 Résoudre par *branch&bound* le problème de trouver le nombre de représentants par district minimisant l'écart maximum.

Question 2 Plus généralement, on veut distribuer d députés dans les n premiers districts en minimisant l'écart maximum de représentation par personne par rapport à l'idéal de un pour cent, avec la contrainte d'au moins un député par district. Proposer une carcatérisation récursive de la valeur minimale de cet écart maximum. Combien coûterait en mémoire et en ordre un programme qui implémenterait cette caractérisation ?

B&B avec espace mémoire borné et cache (30')

Pour atteindre le plus rapidement possible une solution optimale et donc l'arborescence critique, l'ordre de parcours de la collection des sous-problèmes à explorer joue un rôle curcial. Cet ordre de parcours est défini par les méthodes `add()` et `pop()` de la collection, qui respectivement y ajoute et en extrait un élément.
Dans toute la suite on suppose que les collections `Stack` et `Queue` et `PriorityQueue` sont fournies et implémentent respectivement :

— pile (`Stack`) : l'élement extrait est le dernier ajouté (LIFO) ;
— file (`Queue`) : l'élement extrait est le premier ajouté (FIFO) ;
— file de priorité (`PriorityQueue`) : l'élément extrait est celui de plus forte priorité.

Question 3 (10') Stockage des nœuds à explorer dans une file de priorité ;

1. Entre deux sous-problèmes d'évaluations optimistes différentes et de même profondeur, lequel explorer en priorité ?
2. Entre deux sous-problèmes de même évaluation optimiste et de profondeurs différentes, lequel explorer en priorité ?
3. Quelle priorité utiliser lorsqu'on stocke les nœuds dans une file de priorité ?
4. Quelle place mémoire est requise en pire cas pour une file de priorité ?

Question 4 (15') **B&B avec contrainte mémoire et localité**. Soit T la taille mémoire requise pour stocker un nœud de l'arbre du B&B et soit d la profondeur maximum de l'arbre.
Pour s'exécuter dans un espace mémoire limité de taille M, (on suppose [1] $d \times T \ll M$s), on procède comme suit :

— Quand l'arbre contient moins de $M/T - d$ nœuds, l'ajout et le retrait de nœuds est géré par une file de priorité ;
— quand l'arbre contient plus de $M/T - d$ nœuds, l'ajout et le retrait de nœuds est géré par une pile (profondeur).

1. Implémenter une telle collection, à savoir l'initialisation (constructeur), la fermeture (destructeur), les méthodes `add`, `top`, `pop`, `estVide`.
2. Sur le modèle CO (cache de taille Z et lignes de cache de taille L remplacées par politique LRU), comment choisir M pour limiter le nombre de défauts de cache du B&B et quel est ce nombre ?
3. L'appel système `getpagefaults` donne le nombre de défauts de page depuis l'initialisation du sytème. Comment implanter l'algorithme ci-dessus en utilisant `getpagefaults` sans connaître Z, L ?

Question 5 (15') **B&B anytime**.

1. On stoppe l'algorithme en cours d'exécution ; donner un algorithme qui calcule l'écart maximal de la valeur de la solution courante avec la valeur optimale.
2. Donner un algorithme qui donne une solution à un écart au plus e de l'optimum et qui explore au plus 2 fois plus de nœuds que nécessaire avec le même parcours.

Sac à dos binaire

On considère un problème de sac à dos binaire : maximiser la valeur du sac de volume V en le remplissant avec des objets choisis parmi n ; pour $i = 0 \dots n-1$, l'objet i est de volume v_i et de valeur g_i.

Question 6 Instancier l'algorithme *branch&bound* générique pour résoudre le sac à dos binaire. Expliciter : la représentation d'un sous-problème ; la fonction d'évaluation optimiste.

Question 7 La collection est une file (FIFO) : analyser temps d'exécution, place mémoire et localité en pire cas.

Question 8 La collection est une pile (LIFO) : analyser temps d'exécution, place mémoire et localité en pire cas.

Voyageur de commerce (TSP)

Donner le principe d'un algorithme *branch&bound* pour résoudre le problème du TSP : donner le principe des méthodes de branchement et d'évaluation optimiste (on ne demande pas de programme ici).

1. Cette hypothèse est raisonnable sinon même un parcours en profondeur de l'arbre est impossible.