Vecteurs Creux

Jérémie Dumas & Julien Herrmann

Préambule

Tout d'abord, la présente archive comporte deux dossiers, chacun correspondant à des aspects différents du problème de compression de structures creuses : les vecteurs creux (unidimensionnels) et les matrices creuses. L'écriture des fonctions sur les vecteurs creux a constitué la majeure partie du développement de ce projet, et est aboutie et fonctionnelle. La partie sur les matrices creuses a été commencée pour les vacances de Noël, et n'est pas encore tout à fait opérationnelle.

Avec le recul, on peut se demander s'il n'aurait pas été plus instructif de creuser (sans mauvais jeu de mot) la partie vecteur et proposer d'autres implémentations des vecteurs creux (arbres de dictionnaires, tables de hachage, etc.). Nos pistes de réflexion nous ont cependant amené à élargir notre connaissance du sujet à la compression de vecteurs bidimensionnels.

Historique du développement

La principale difficulté du sujet étant le choix judicieux des structures de données utilisées, on retiendra surtout le changement d'approche effectué au cours du développement entre :

- L'*approche statique*. On définit les objets au début des fonctions, en statique, et on devait utiliser des `#define` pour choisir la taille maximal d'un vecteur, etc. Ce n'est pas très pratique et manque de souplesse.
- L'*approche dynamique*. Cette fois on manipule essentiellement des pointeurs, on crée les structures avec malloc, qu'on oublie pas de libérer ensuite, etc.

Après avoir finit d'implémenter les parties proposées, on s'est donc mis à réfléchir sur d'autres moyens de compressions. Julien a essayé de coder des matrices par bloc (BSR), tandis que Jérémie a finalement décidé d'implémenter une nouvelle structure pour les vecteurs creux.

1 Différentes structures pour les vecteurs creux

1.1 Les structures de données

1.1.1 Vecteur

C'est le type de base. On y stock toutes les entrées, y compris les zéros superflus.

Avantages : affectation et recherche en temps constant. Itération d'une fonction sur les éléments dans l'ordre croissant des indices efficace.

Inconvénients : espace mémoire nécessaire trop important. Nombre maximum d'éléments limité.

1.1.2 Vec_comp

C'est un tableau qui contient d'une part les indices des cases non vides du vecteurs, et leur valeur d'autre part. Les éléments sont triés par indices croissants.

Avantages : espace mémoire plus faible. Temps de recherche logarithmique (en fonction du nombre d'éléments non nuls). Itération efficace.

Inconvénients : Temps d'insertion linéaire. Nombre maximum d'éléments non nuls limité.

1.1.3 Vec_list

Le vecteur est vue comme une liste chaînée de ses cases non-vides. Chaque case contient l'indice et la valeur de l'élément dans le tableau non compressé.

Avantages : pas de limitation sur le nombre maximum d'éléments non nuls. Itération efficace.

Inconvénients : Insertion et recherche ont un coût linéaire dans le pire des cas.

1.1.4 Vec_mult

Sans doute la plus originale des 4. On ne l'a pas inventé, mais son fonctionnement nécessite d'être présenté, puisqu'elle n'est pas décrite dans le sujet :

On veut stocker n éléments dans un vecteur. On écrit $n = \overline{n_{k-1}...n_0}^2$ en binaire, où $k = \log_2(n)$. On utilise alors k tableaux $A_{k-1}, ..., A_0$ dans lesquels ont va stocker des éléments par ordre d'indices croissants. La tableau A_i a une taille 2^i, et un tableau A_i est entièrement rempli si et seulement le i-ième bit de n vaut 1 (1). Et ses éléments sont triés (2). Le nombre d'éléments d'un tableau A_i est alors $n_i \cdot 2^i$

On vérifie qu'on stocke bien au final $\sum_{i=0}^{k-1} n_i \cdot 2^i = n$ éléments.

La recherche de l'élément d'indice i se fait en parcourant les k tableaux, dans lequels on fait une recherche dichotomique (les éléments à l'intérieur sont triés). On montre qu'elle a un coût dans le pire des cas $O(\log_2(n)^2)$.

La fonction d'insertion doit quant à elle doit conserver les deux propriétés (1) et (2). Pour cela on recherche le premier tableau A_i vide, dans lequel on ajoute un nouvel élément, puis l'on y fusionne les i tableaux précédents $A_0...A_{i-1}$.

Une telle fonction d'insertion a un coût amorti $O(\log_2(n))$. C'est donc une structure de donnée adaptée à de multiples insertions/recherches d'éléments.

La fonction de suppression (pour la remise à zéro d'une valeur), procède de manière analogue. Cependant sont coût amorti est moins intéressant, car selon l'élément supprimé il peut être linéaire. Aussi on considère qu'on peut se permettre de travailler avec quelques zéros pour les algorithmes qui vont l'utiliser, et cette fonction n'a pas été implémentée. C'est une particularité à prendre en compte lors de la partie tests.

Avantages : insertion en temps amorti logarithmique, recherche en temps quasi-logarithmique.

Inconvénients : itération d'une fonction par ordre croissant d'indice difficile.

1.2 Fonctions disponibles

1.2.1 Manipuler un type de donnée

On dispose dans la plupart des cas des fonctions générals suivantes :

- Création, suppression d'un élément
- Remise à zéro d'un élément et remise à zéro rapide. Dans le deuxième cas les valeurs des cases non-utilisées ne sont pas forcément remises à zéro.
- Recherche de la valeur d'une case. Recherche d'un pointeur vers cet élément (pour incrémenter la valeur d'une case par exemple).
- Affectation d'un élément. Insertion aussi d'un nouvel élément pour le type `vec_mult` (il faut vérifier qu'il n'est pas déjà présent, sinon il y a un risque de redondance).

1.2.2 Convertir un type de donnée

Il y a en tout 12 fonctions de conversion, permettant de convertir chaque type en l'un des 3 autres possibles.

Signalons qu'une telle fonction prend en argument un pointeur, une borne éventuelle sur le nombre d'éléments, et qu'elle renvoie un pointeur vers l'instance du nouveau type. Elle ne libère

pas le pointeur donné en argument, il faut donc penser à le faire manuellement. Penser aussi à libérer le pointeur qui recevra le résultat de la fonction de conversion avant l'affectation.

Attention si on essaie de convertir un élément de type `vec_mult` qui contient des zéros ou des redondances, le comportement n'est pas spécifié. S'il y a des zéros additionnels, ceux-ci devraient toutefois être simplement considérés comme des cases non-vides, ce qui n'est pas très gênant.

1.3 Informations complémentaires

1.3.1 Problèmes rencontrés

Le seul problème intéressant rencontré et qui mérite d'être signalé concerne la fonction de conversion `Vec_list_of_vec_mult`.

En effet, au départ l'idée était d'allouer un espace contigu en mémoire pour les n cases de la liste, afin de pouvoir faire des fusions itérées sans utiliser d'emplacement mémoire superflu. On procédait donc comme avec la fonction de conversion `Vec_comp_of_vec_mult`, étant donné qu'on connaissait l'emplacement de chaque cellule.

Le problème vient lorsque l'on souhaite détruire la liste ainsi créer. Il faut garder pour chaque cellule un pointeur correct vers la suivante (pour pouvoir faire des insertions par la suite). Or pour libérer une cellule, si cette cellule a été allouée grâce au `malloc` contigu de la fonction de conversion, il faut libérer toute la plage d'un coup en faisant un `free` de l'adresse de la première cellule. On ne peut pas tester, pour une adresse donnée, si la zone pointée a été allouée par un `malloc`, ou si elle appartient à la plage contigüe décrite précédemment.

1.3.2 Tests effectués

Comme on peut le voir dans le fichier `main.c`, on a effectué tous les tests de conversion possibles sur les différents type de données. On a aussi testé plusieurs affectations (aléatoires) sur des vecteurs de tailles variables. Se reporter au code source pour de plus amples détails.

2 Extensions aux matrices

2.1 Matrices par blocs

Après avoir cherché de nouvelles façons de stocker les vecteurs creux, nous nous sommes donc intéressés aux vecteurs de dimension $p \geqslant 2$, ou dans le cas présent aux matrices creuses ($p = 2$). Le format de stockage choisi est appelé BSR (BLOCK COMPRESSED SPARSE ROW). On stock dans la structure les dimensions de la matrice ainsi que de ses blocs, puis on crée 3 tableaux :

- `values` : contient les éléments des blocs non-nuls (comportant au moins un élément non-nul) de la matrice.
- `columns` : `columns[i]` est le numéro de la colonne où se trouve le i-ème bloc non-nul de la matrice.
- `rowindex` : `rowindex[i]` est le numéro du premier bloc non-nul de la i-ème ligne de la matrice par bloc (vaut -1 si cette ligne n'a que des bloc nuls). On ajoute à la fin de ce tableau le nombre de blocs non-nuls de la matrice.

Exemple

$$M = \begin{bmatrix} 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 & 2 & 2 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

On peut la stocker de plusieurs façons :

```
M= {hauteur  = 6; hauteur_block = 2;
    longueur = 6; largeur_block = 2;
    values   = [0;1;0;1;2;0;0;0;0;0;2;3;0;1;0;1;2;2;0;0];
    columns  = [0;1;2;1;2];
    rowindex = [0;2;3;5]}

M= {hauteur  = 6; hauteur_block = 3;
    longueur = 6; largeur_block = 3;
    values   = [0;1;2;0;1;0;0;0;0;0;2;3;1;2;2;1;0;0];
    columns  = [0;1];
    rowindex = [0;1;2]}
```

2.2 Type de données

Si on choisit de n'utiliser que des tableaux, lors de l'appel de la fonction `Affect_block_matrix`, la taille de ces derniers peut être peut-être modifier (car on peut créer de nouveaux blocs non-nuls). Cela pose donc des problèmes pratiques, et il est difficile d'insérer des éléments au milieu d'un tableau. Nous avons donc choisit d'utiliser des listes pour stocker les éléments de `values`,`columns` et `rowindex`. Julien a donc écrit les fonctions de bases (création, destruction, affectation, valeur, insertion, etc.) pour un nouveau type de liste, analogue au `vec_list` des vecteurs creux.

2.3 Informations complémentaires

2.3.1 Problèmes rencontrés

Il semblerait que les fonctions de conversion compilent mais aient un comportement étrange. Il y a sans doute une erreur au niveau des indices, mais les raisons précises de l'erreur n'ont pas encore pu être identifiées.

2.3.2 Perspectives

Pour améliorer le stockage on peut remarquer que, dans le cas des matrices creuses, le tableau `values` peut être lui-même un vecteur creux, étant donné que l'on stocke les éléments de bloc qui contiennent au moins un élément non-nul (et qui donc peuvent contenir des zéros). Une première piste pour avoir un stockage encore plus efficace est donc d'utiliser une des structures de vecteurs creux précédentes pour stocker le tableau `values`. On pourrait également envisager d'écrire un algorithme qui détermine quelles sont les dimensions optimales des blocs. En effet, comme on peut le constater sur l'exemple, une modification de la taille des blocs entraine un changement dans la taille des tableaux stockés. Ainsi pour stocker le moins de valeurs possibles, on peut jouer sur les dimensions des blocs.

3 Bilan

D'une manière générale, ce rapport reflète bien les difficultés auxquelles on est confronté lors de la manipulation des structures de données du type vecteur creux. La partie vecteur fonctionne bien tandis que la partie matrice aurait mérité un peu plus de temps pour être finalisée.

Pour aller plus loin, signalons qu'une structure de donnée n'est efficace que par rapport aux algorithmes qui vont l'exploiter. En effet, on peut vouloir optimiser un tableau pour le parcours de ses éléments (itération d'une fonction), ou bien pour la recherche/l'insertion, voire simplement l'union (problème union/find). Ainsi il aurait pu être intéressant de se concentrer sur d'autres structures pour les vecteurs creux (dictionnaires ou tables de hachage cités plus haut), mais le temps manque malheureusement si on n'y songe pas dès le début.