Structure de données et algorithmes

Projet 2: bin packing

25 février 2025

L'objectif du projet est d'implémenter, d'analyser théoriquement et de comparer empiriquement différents algorithmes approximatifs pour résoudre le problème de bin packing. L'implémentation de ces différents algorithmes vous permettra de mettre en œuvre différentes structures de données vues au cours (file à priorité et arbres binaires de recherche).

Bin packing

Le SEGI vous appelle pour résoudre le problème algorithmique suivant. La rectrice leur a demandé de faire un backup complet des données de l'université mais aimerait évidemment faire ça à moindre coût. Les données de l'Université sont réparties sur N fichiers de tailles entières strictement positives notées $s_i \in \mathbb{Z}_0^+$, avec $1 \leq i \leq N$. Afin de simplifier l'achat, le SEGI prévoit d'uniquement commander des disques de taille B. Pour faciliter l'archivage, on souhaite également stocker chaque fichier en entier sur un seul disque. On supposera donc que $s_i \leq B$ pour tout i. Afin de minimiser les coûts, le SEGI vous demande de déterminer le nombre minimal de disques qui sont nécessaires pour stocker l'ensemble des fichiers sous ces contraintes et comment répartir les fichiers entre les disques pour arriver à ce nombre minimal. Vous savez déjà bien sûr que ce nombre sera compris entre $\lceil \sum_i s_i/B \rceil$ et N.

Après quelques recherches, vous vous rendez compte que ce problème algorithmique est le problème bien connu de *bin packing* [1] et en lisant sur le sujet, vous apprenez qu'il s'agit d'un problème extrêmement complexe et qu'aucune solution exacte efficace n'existe malheureusement. Vous vous résiliez donc à vous rabattre sur des solutions approximatives basées sur des heuristiques gloutonnes. Votre idée est de les implémenter efficacement (N est potentiellement très grand) et de voir laquelle donnera la meilleure solution sur les données de l'université.

La plupart de ces heuristiques sont basées sur la même approche gloutonne. Les fichiers sont traités séquentiellement. Pour chacun d'eux, on cherche le disque le plus approprié pour y placer le fichier parmi les disques ouverts (c'est-à-dire qui ont déjà reçu un fichier) selon une stratégie particulière détaillée ci-dessous. Si aucun disque ne peut contenir le fichier, on crée un nouveau disque sur lequel on place le fichier et ce disque rejoint la liste des disques ouverts.

Les quatre stratégies de sélection de disque suivantes vous semblent les plus pertinentes:

- **Next-Fit**: On ne maintient qu'un seul disque ouvert qu'on remplit des fichiers parcourus dans l'ordre tant que c'est possible. Dès qu'un fichier ne rentre pas dans le disque courant, on passe à un nouveau disque.
- **Worst-Fit**: On place le fichier courant dans le disque le moins rempli parmi ceux ouverts.
- **Best-Fit**: On place le fichier courant dans le disque le plus rempli qui peut encore contenir le fichier.

[1] https://en.wikipedia.org/wiki/Bin_packing_problem

- **First-Fit**: On place le fichier courant dans le premier disque, dans leur ordre d'ouverture, qui peut contenir le fichier.

Pour obtenir de meilleures performances, vous traiterez les fichiers par *ordre de taille décroissante* dans vos différentes implémentations. Vous avez lu en effet que le traitement des petits fichiers à la fin permettaient de combler les vides laissés par les gros fichiers.

Structures de données

Le nombre de fichiers de l'université étant de plusieurs centaines de milliers, votre implémentation doit être la plus efficace possible pour permettre au SEGI d'obtenir une solution en un temps raisonnable. Cela veut dire que pour certaines stratégies, il faudra utiliser les structures de données les plus appropriées. Certains idées sont données ci-dessous pour chaque stratégie.

Next-Fit. C'est l'approche la plus simple. Il n'y a pas besoin de structure particulière pour l'implémenter efficacement.

Worst-Fit. Cette solution demande de pouvoir accéder pour chaque nouveau fichier au disque parmi ceux ouverts qui est le moins rempli. Il doit être possible également de mettre à jour l'espace vide du disque. Cela semble correspondre très bien au cahier des charges d'une file à priorité.

Best-Fit. On doit pouvoir accéder ici au disque d'espace libre le plus faible qui peut recevoir le fichier. Ca vous semble possible si vous stockez les disques ouverts dans un arbre binaire de recherche en utilisant comme clé la taille de l'espace libre sur le disque (voir la figure 1 et la question 3 de l'analyse théorique). Pour garantir un code efficace, il est cependant nécessaire d'utiliser une implémentation d'arbre équilibré.

First-Fit. C'est l'approche la plus compliquée à implémenter efficacement. Il faut pouvoir trouver le premier disque ouvert qui peut contenir le fichier. Il y a donc deux critères à prendre en compte: la taille de l'espace vide sur le disque et le moment auquel le disque a été ouvert. Une approche possible est d'utiliser un arbre binaire de recherche utilisant comme clé l'indice du disque dans l'ordre d'ouverture. En ajoutant en chaque nœud de cet arbre, la capacité maximale des disques dans le sous-arbre dont ce nœud est la racine, il est possible de retrouver rapidement le disque ciblé (voir la figure 2 et la question 4 de l'analyse théorique).

Implémentation

Nous vous fournissons les fichiers suivants:

- `File.h/File.c`: un type de données simple pour représenter un fichier, caractérisé par un nom et une taille (de type `size_t`)
- `Disk.h/Disk.c`: un type de données simple pour représenter un disque, caractérisé par sa capacité, une taille d'espace libre, une taille d'espace utilisé et une liste de fichier.
- `LinkedList.h/LinkedList.c`: une implémentation de liste, qui sera utilisée pour fournir les données à vos fonctions et récupérer les résultats. L'implémentation contient également une fonction de tri de la liste qui vous sera utile pour trier les fichiers en ordre décroissant.
- `main.c`: un fichier principal qui permet de lire un fichier d'entrée contenant une liste de fichiers au format csv pour tester vos implémentations. Il est également possible de créer une liste de fichiers aléatoires pour faire des tests à plus grande échelle.

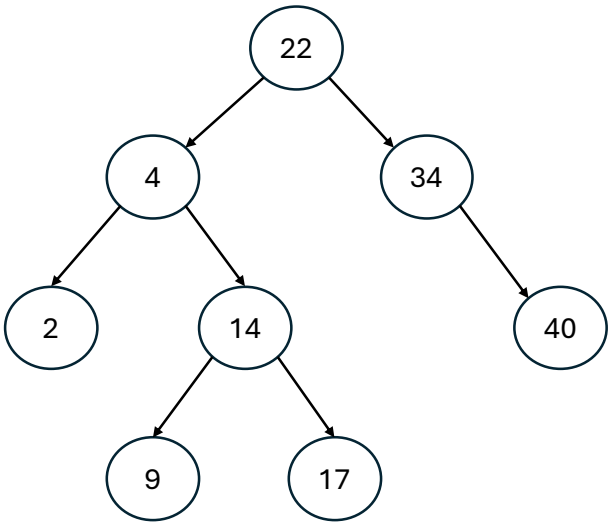

Figure 1: Un arbre binaire de recherche dont les clés représentent l'espace restant dans 8 disques ouverts. Avec la stratégie Best-Fit, un fichier de taille 12 devra par exemple être stocké dans le disque de taille 14, un fichier de taille 17 dans le disque de taille 17, et un fichier de taille 41 mènera à la création d'un nouveau disque.

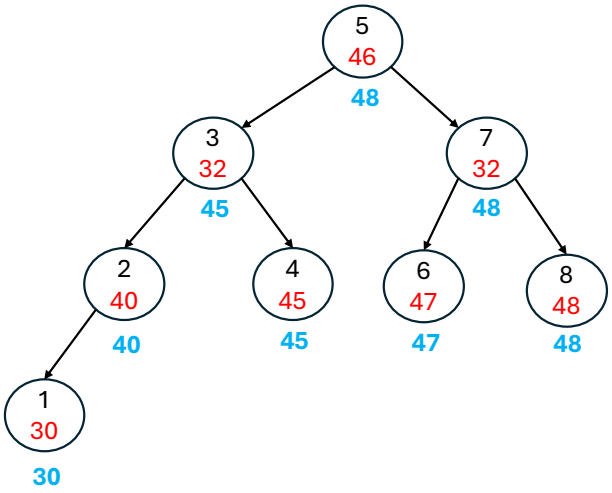

Figure 2: Une arbre binaire de recherche pour implémenter l'approche First-Fit. La clé, en noir, est l'indice de création du disque, la valeur en rouge est l'espace libre sur le disque, la valeur en bleu est l'espace disque maximal parmi tous les nœuds du sous-arbre dont le nœud est la racine. Avec la stratégie First-Fit, un fichier de taille 32 devra être stocké dans le disque n°2 et un fichier de taille 44 dans le disque n°4. Cette représentation permet de trouver le disque ciblé en parcourant une seule branche dans l'arbre.

- `Makefile`: permettant de compiler tous les fichiers et de les tester.

Vous devez implémenter les quatres fichiers suivants:

`BP_nextfit.c`, `BP_worstfit.c`, `BP_bestfit.c`, `BP_firstfit.c`.

Chacun de ces fichiers doit implémenter la solution correspondant au problème de bin packing par le biais de l'unique fonction `binpacking` déclarée dans le fichier d'entête `BP.h`, qui prend en entrée une taille de disque (B plus haut), une liste de fichiers, et une liste de disques, vide initialement. La fonction doit renvoyer le nombre de disques trouvé par l'algorithme et placer ces disques, avec leurs fichiers, dans la liste donnée en troisième argument. Une solution `BP_naive.c` vous est fournie comme exemple, qui crée simplement un nouveau disque pour chaque fichier en les prenant dans l'ordre initial.

Pour implémenter `BP_worsfit.c`, vous devez vous baser sur une file à priorité que vous devrez implémenter par vous-même dans le fichier `PQ.c` selon l'interface définie dans `PQ.h`. Pour implémenter `BP_bestfit.c` et `BP_firstfit.c`, vous pouvez vous baser sur une implémentation existante d'arbre binaire de recherche générique parmi ces deux-ci:

- `avl_bf.c/avl_bf.h`, obtenue ici https://github.com/xieqing/avl-tree/.
- `treap.c/treap.h`, obtenue ici https://github.com/matthewclegg/treap.

La première est une implémentation d'AVL telle que vue au cours. La deuxième est une implémentation de ce qu'on appelle un Treap ou bien un arbre randomisé[2]. Contrairement à l'AVL, cette approche ne garantit pas du log N dans le pire cas mais seulement en moyenne. Cette implémentation donne cependant des performances suffisantes pour notre application et est un peu plus simple que l'AVL. Vous pouvez modifier ces implémentations comme vous le souhaitez pour vos implémentations des fichiers `BP_bestfit.c` et `BP_firstfit.c`. Votre code d'arbre modifié devra être fourni respectivement dans les fichiers `BST_bestfit.c/.h` et `BST_firstfit.c/.h` qui sont compilés par le `Makefile` fourni.

Suggestions: Pour `BP_bestfit.c`, il suffit d'utiliser l'arbre pour stocker les disques et d'ajouter à l'interface une fonction permettant de retrouver le disque approprié (c'est l'objet de la troisième question théorique ci-dessous). Pour `BP_firstfit.c`, il faut ajouter aux nœuds de l'arbre deux informations: l'espace libre sur le disque au nœud et l'espace disque maximal parmi tous les disques du sous-arbre dont ce nœud et la racine. Il est possible alors de retrouver le disque approprié efficacement (voir la quatrième question théorique). La difficulté principale est cependant de maintenir l'information sur les espaces disque à jour lors de l'insertion en prenant en compte les rotations. Notez que pour `BP_firstfit.c`, vous n'avez pas besoin en principe de faire de suppression de disques dans l'arbre (si la capacité d'un disque vaut 0, il ne sera jamais choisi). Vous pouvez donc enlever cette fonction du code et n'avez donc pas besoin de la modifier pour prendre en compte l'information sur les espaces disques.

Analyse théorique

Dans votre rapport, répondez aux question suivantes:

1. Donnez une idée approximative de la complexité d'une solution au problème par recherche exhaustive (en fonction de N).
2. Prouvez, en donnant un contre-exemple, qu'au moins deux des quatres solutions gloutonnes proposées ne sont pas optimales.
3. Donnez le pseudo-code d'une fonction `Tree-Search-BF(T, k)` retrouvant dans un arbre binaire de recherche représenté comme au cours, la clé la plus petit de l'arbre plus grande ou égale à k. Analysez sa complexité en temps au pire cas en fonction de N.

[2] https://en.wikipedia.org/wiki/Treap

4. Donnez le pseudo-code d'une fonction `Tree-Search-FF(T, size)` retrouvant dans un arbre binaire de recherche tel que représenté à la figure 2 le nœud correspondant au premier disque qui peut contenir une fichier de taille `size`. Soit un nœud `x`. `x.key`, `x.space`, `x.maxspace` contiennent respectivement le numéro du disque, l'espace libre sur ce disque et l'espace maximum parmi les disques dans le sous-arbre dont `x` est la racine. Analysez sa complexité en temps au pire cas en fonction de N.

5. Analysez la complexité en temps au pire cas des quatre algorithmes en fonction du nombre de fichiers N en prenant en compte (et en précisant dans le rapport) vos choix d'implémentation.

Analyse empirique

Dans le rapport, répondez aux questions suivantes:

1. En utilisant le fichier `main.c` fourni, tracez sur un graphe l'évolution du gaspillage obtenu par chacune des solutions en fonction de N. Le gaspillage est défini par la différence entre la somme des tailles des disques utilisés et la taille totale des fichiers. Vous pouvez fixer la taille des disques à 1000000. Faites des courbes moyennes sur au moins 5 expériences. Discutez brièvement des différences entre méthodes.

2. Pour les mêmes expériences, tracez également l'évolution des temps de calcul moyens en fonction de N. Discutez la concordance entre ces courbes et l'analyse théorique de la section précédente.

Date de remise et soumission

Le projet est à réaliser ***seul ou en binôme*** pour le ***dimanche 11 mai 2025 à 23h59*** au plus tard. Le projet est à remettre via Gradescope (http://gradescope.com, code cours **42V2RV**)

Les fichiers suivants doivent être remis:

1. votre rapport (6 pages maximum) au format PDF et nommé `rapport.pdf`. Soyez bref mais précis et respectez bien la numération des (sous-)questions;

2. les quatre fichiers `BP_nextfit.c`, `BP_worstfit.c`, `BP_bestfit.c`, `BP_firstfit.c`.

3. L'implémentation de la file à priorité `PQ.c`,

4. les deux implémentations d'arbres `BST_bestfit.c/.h` et `BST_firstfit.c/.h`.

Respectez bien les extensions de fichiers ainsi que les noms des fichiers, en ce compris les minuscules/majuscules. N'incluez aucun fichier supplémentaire.

Un projet non rendu à temps recevra une cote globale nulle. La soumission du projet suppose que vous avez pris connaissance des règles en terme de plagiat rappelées sur Ecampus. En cas de plagiat[3] avéré, le groupe se verra affecter une cote nulle à l'ensemble du cours.

Les critères de correction sont précisés sur Ecampus.

Bon travail !

[3]Des tests anti-plagiat seront réalisés.