BARREIRO Hugo
RIBEIRO Damien

Projet

Introduction à la Programmation Fonctionnelle

LDD2 IM1 2022-2023

Sommaire :

Introduction :

Le but de ce projet est d'écrire un programme capable de compresser puis décompresser des fichiers en utilisant l'algorithme d'Huffman.

Les commandes de compilation sont :

- Dune clean
- Dune build @all
- Dune exec src/huff.exe

L'exécutable huff.exe se trouve dans _build/default/src

On peut ajouter --stats lors de l'exécution pour afficher les statistiques de la compression. On peut également ajouter --help pour afficher un message d'aide pour utiliser l'exécutable. Ce dernier attend toujours, soit une option, soit un fichier en argument d'exécution.

I) Répartition du travail :

Lors de ce projet, nous avons beaucoup travaillé ensemble, que ce soit lors des séances de TP prévus pour le projet ou depuis chez nous en s'appelant.

A chaque fois qu'un de nous deux écrivait du code, l'autre le relisait afin de l'améliorer ou de corriger d'éventuels bugs.

Néanmoins, Damien a un peu plus travaillé sur l'arbre d'Huffman et la décompression et Hugo a un peu plus travaillé sur la ligne de commande et la compression (même si on a travaillé chacun sur toutes les parties du code). Le tas a été bien travaillé par nous deux.

Pour ce qui est des tests, du débogage et de la mise en ordre du projet, nous avons travaillé main dans la main.

II) Tas :

Nous avons décidé d'implémenter un tas.

Tout d'abord, car celui-ci est adapté pour la création d'un arbre d'Huffman. En effet, la tas permet de trouver et supprimer le minimum en temps logarithmique. Ceci est intéressant pour nous lors de la construction de l'arbre.

De plus, cela était un défi car nous n'avions auparavant jamais implémenté une telle structure de données. Cela nous a également permis d'appliquer la théorie apprise en cours d'Algorithmique Avancée.

Nous avons défini notre type tas comme :

```
type 'a heap =
  { size : int
  ; tas : 'a array
  }
```

Comme vous pouvez le remarquer, il y a deux champs dans notre type. Un entier pour la taille du tas et un tableau de type 'a pour les données.

Stocker la taille des données utilisées nous permet de ne pas allouer un tableau de taille size + 1 a chaque ajout de données dans notre tableau. De plus, cela sert également lors de la suppression d'éléments. Car lorsque notre tableau n'occupe plus qu'un quart de la place allouée, on peut alors allouer un nouveau tableau deux fois moins grand. Le type 'a de notre tableau permet d'avoir un tas plus général et ne fonctionnant pas uniquement que dans notre cas. On pourra donc le réutiliser.

Nous avons décidé d'implémenter notre tas avec un tableau et non un arbre car cela nous a semblé plus simple qu'avec un tableau au niveau de l'implémentation. De plus, le tas avec un tableau est plus efficace qu'avec un arbre.

III) Arbre d'Huffman :

Comme notre compression/décompression repose sur l'algorithme d'Huffman, cette étape était donc indispensable.

Nous avons implémenté notre arbre de cette manière :

```
type 'a tree =
  | Empty
  | Leaf of 'a
  | Node of 'a tree * 'a tree
```

(Note : on aurait pu se passer de Empty mais on trouvait que son utilisation rendait le code un peu plus lisible)

Notre type 'a tree est récursif ce qui va nous permettre d'écrire des fonctions récursives. Ce qui est adapté au langage OCaml et confortable pour travailler sur les arbres.

Avec l'arbre d'Huffman on va se retrouver avec les caractères (uniques) aux feuilles. Ainsi, en parcourant l'arbre, on va retrouver les codes de chaque caractère (0 quand on va à gauche, 1 à droite). Les caractères les plus présents se trouveront aux profondeurs les moins élevées, les moins présents aux profondeurs les plus élevées. Ceci est logique car plus un code est court, moins il prend de place lors de la compression.

Nous avons le type 'a pour avoir un type plus général, c'est-à-dire réutilisable. Ainsi, il ne fonctionne pas uniquement que dans notre cas.

IV) Compression :

Le fonctionnement de notre fonction de compression n'est pas très compliquée à comprendre car nous l'avons décomposée en plusieurs sous-étapes :

```
let compress f stats =
  let in_c = open_in f in
  let h = file_to_heap in_c in
  close_in in_c;

  let tree =
    if heap_size h = 1 then Node (snd (heap_tas h).(0), Empty)
    else heap_to_tree h
  in
  let tab = huff_tab tree in

  let out_c = open_out (f ^ ".hf") in
  let os = of_out_channel out_c in
  let nb_bits_code = write_huffcode tab os in

  let in_c = open_in f in
  let nb_bits_avant, nb_bits_apres = write_compress tab in_c os in

  close_in in_c;
  finalize os;
  close_out out_c;

  if stats then begin
    Printf.printf "Taille de %s : " f;
    print_file_size nb_bits_avant;
    Printf.printf "\n";
    Printf.printf "Taille de %s : " (f ^ ".hf");
    print_file_size (nb_bits_code + nb_bits_apres);
    Printf.printf "\n"
  end
```

Tout d'abord, grâce à la fonction file_to_heap, on va stocker dans un tableau le nombre d'occurrences de chaque caractère dans le fichier à compresser. Chaque indice i du tableau correspond au caractère codé par la valeur i (comprises entre 0 et 255 inclus) Puis, on va créer un tas, qui va se trier automatiquement en fonction du nombre d'occurrences par les propriétés du tas, avec des paires (nombre_d'occurrences, caractère) pour les caractères apparaissant au moins une fois dans le fichier.

Ensuite, grâce à ce tas, nous pouvons créer l'arbre d'Huffman. Pour ce faire, on va créer l'arbre d'Huffman "par le bas". Ainsi, on va retirer le minimum (soit m1), puis encore une fois le minimum (soit m2) et on va créer un nœud (m1,m2) et ainsi de suite jusqu'à vider le tas. A la fin, on va avoir la forme d'arbre voulue, avec les caractères plus présents vers le haut de l'arbre et les moins présents vers le bas de l'arbre.

On peut, à présent, associer un code pour chaque caractère. On va ainsi parcourir l'arbre fraîchement créé et associer à chaque caractère son code. Pour les stocker de manière efficace et y accéder rapidement, on va utiliser un tableau d'indice 0 à 255 inclus. Ainsi, on pourra accéder au code d'un caractère codé habituellement par la valeur i avec T[i].

Avant de compresser le fichier, on va écrire les codes d'Huffman dans le fichier compressé afin de les retrouver lors de la décompression. On va écrire la longueur du code puis le code.

Enfin, on va parcourir le fichier à compresser octet par octet et écrire le code d'Huffman du caractère lu grâce au tableau créé. On va également compter le nombre de bits lus et écrits afin d'afficher les stats si demandé.

(On affiche, à la fin, les stats si cela à été demandé)

Si on note n la taille du fichier à compresser, notre algorithme a une complexité en temps en O(n), pour être plus précis, elle est environ égale à : n + 256*log2(256) + 256*log2(256) + log(256) + 256 + n = 2n + 2*256*8 + 264 = 2n + 4360 = 2n + cout constant

V) Décompression :

Nous avons également décomposé notre fonction de décompression en plusieurs étapes :

```
let decompress f =
  let in_c = open_in f in
  let is = of_in_channel in_c in

  let list_data = extr_data is in
  let tree =
    match recreate_tree list_data with
    | Leaf c ->
      Node (Leaf c, Empty)
    | t -> t
  in

  let out_c = open_out ("decompressed_" ^ Filename.remove_extension f) in
  decomp_file is out_c tree;
  close_in in_c
```

Tout d'abord, on récupère les codes d'Huffman. On les stockera dans une structure de données nommée data_decompress composée de 3 trois champs. Le champ long correspond à la longueur d'un code de huffman, le champ code correspond au code en base 10 et le champ tree correspond à un arbre de caractère associé au code et la longueur (ex: si longueur = 1 et code = 1 alors tree correspond au descendant droit de la racine de l'arbre d'Huffman avant compression).

Pour récupérer les codes, on fait 256 fois : longueur = lire 4 bits, code = lire longueur bits, stocker dans une liste de type data_decompress. Puis, on va trier cette liste dans l'ordre décroissant des longueurs des codes (c'est-à-dire en fonction du champ longueur).

Ensuite, grâce à cette liste, on peut recréer l'arbre d'Huffman qui va nous permettre de décompresser le fichier de manière efficace. Pour ce faire, on va regrouper les éléments de même longueur, puis les assembler ensemble pour créer des nœuds. Pour y parvenir, on prend le

premier élément et on cherche un élément qui a le même code à l'exception du dernier bit et qui ont la même longueur.

(remarque: la recherche de cet élément analogue est plus rapide grâce au fait que la liste est triée. Que l'on trie par longueur car plusieurs éléments peuvent avoir le même code mais avec des longueurs différentes. Pour conclure cette aparté sur le tri, on a choisi l'ordre décroissant car l'on ne dispose que des feuilles au début de la reconstruction de l'arbre. On a ainsi tous les sous-arbres de profondeur maximale au début du programme et on obtient ceux de profondeur inférieure à force de rappeler la fonction.)

On groupe les arbres dans un nœud, on décrémente la longueur des codes par 1, on divise le code par 2 pour que le code soit de bonne longueur et on insère l'élément dans le tableau en s'assurant que la liste reste triée.

On va répéter cette opération tant qu'il y a plus d'un élément dans l'arbre.

Enfin, on peut écrire le fichier décompressé. Pour effectuer cette tâche on va lire bit par bit le fichier compressé (on reprend là où on avait arrêté de lire lorsqu'on a récupéré les codes d'Huffman). Si on lit un bit 0, on descend à gauche dans l'arbre, à droite sinon. Dès qu'on arrive à une feuille, on écrit le caractère se trouvant à cette feuille. Puis, on repart du haut de l'arbre. On répète ceci jusqu'à arriver à la fin du fichier compressé. Notre fichier décompressé est alors écrit.

Il est important de noter que la fonction decomp_file est buguée. Lors de la décompression d'un fichier écrit avec de nombreux caractères différents, la profondeur de notre arbre va être élevée et le nombre de bit à lire pour déchiffrer certains caractères sera aussi plus élevé. Lorsque que le code à écrire dépasse 8 bits, read_bit va lever l'exception Invalid_stream (ligne 50 bs.ml : if num_bits >= 8 then invalid_stream () ; il y a trop de caractère dans le buffer de read_bit). Dans le temps qui nous était imparti, nous n'avons pas trouvé de solution pour éviter cette erreur.

On gère cette exception en affichant un message à l'utilisateur. Lorsque l'exception est levée, seul une partie (ou aucune) du fichier est alors décompressé.

On note n la taille du fichier compressé pour la partie des données (on ne compte pas les codes d'Huffman au début du fichier). La complexité exacte de la décompression varie en fonction de la taille du fichier, du nombre de caractères différents et du nombre d'occurrence des caractères. Il est assez difficile de trouver une approximation de cette dernière mais on peut essayer de trouver une majoration.

La complexité de extr_data dépend de la taille des codes de caractères, on peut majorer la fonction par 256*3+''nombre de caractères différents"*max{"tailles caractères"}).

La complexité de recreate_tree varie en fonction du nombre de caractère différent ; on peut la majorer par O("nombre de caractères différents''²) comparaison pour cette opération. (On prend un élément du tableau, on trouve l'élément correspondant à l'arbre frère, on va faire au plus O("nombre de caractères différents") comparaison et à chaque itération de recreate_tree on va retirer 1 élément de la liste "nombre de caractère différents''-1 iteration.

La complexité de decomp_file varie en fonction de la taille du fichier et de la longueur des codes de arbres. On peut majorer (très éloignés du résultat exact) en considérant que chaque caractères s'écrit avec le code le plus long possible. Il y a (n / 8) caractère donc la complexité est majorable par (n / 8)*max{"tailles caractères"}.

Ici, elle est majoré par :

256*3
- + "nombre de caractères différents"*max{"tailles caractères''}
- + "un certain terme O("nombre de caractères différents''²)"
- + (n / 8)*max{"tailles caractères"}

VI) Ligne de commande et options :

Afin de réaliser cette ligne de commande et de gérer les options d'exécution, nous avons notamment utilisé le module d'OCaml : Arg.

Celui-ci permet de récupérer les options et le nom du fichier donné en argument. Si on utilise l'option --help, alors on affiche le message d'aide. Si on utilise l'option --stats, alors on initialise et appelle la fonction compress_with_stats avec le nom du fichier donné en argument lors de l'exécution. S'il n'y a pas d'option, alors si le fichier donné en argument se termine par .hf, alors on le décompresse sinon on le compresse. Ces fonctions sont initialisées et appelées toujours grâce au module Arg.

De plus, on lève une erreur s'il n'y a pas de fichier donné en argument lors de l'exécution et pas d'option --help.

VII) Tests :

Tout d'abord, nous avons testé nos fonctions individuellement grâce à des affichages dans le main ou directement à l'intérieur des fonctions. Néanmoins, beaucoup de nos tests n'ont pas été sauvegardés dans le git.

Ensuite, une fois que chaque fonction était écrite et testée, on a testé la compression/décompression avec et sans option sur des fichiers .txt (qui sont sur le dépôt git).

Par exemple, on a pu appeler des fonctions avec en paramètres des valeurs particulières visant à vérifier un résultat particulier. Ou encore prendre des données, puis vérifier qu'on les retrouve bien à la fin après plusieurs manipulations de fonctions.

Conclusion :

Ce projet nous a permis de nous familiariser encore plus avec la programmation fonctionnelle et le langage OCaml. Même si on n'a pas encore tous les réflexes de programmation OCaml, nous avons appris à utiliser davantage les spécificités syntaxiques et sémantiques de ce langage.

Nous avons également appris quelques bases sur la compression et décompression. Comment attribuer des codes, les stocker de manière efficace mais également comment manipuler des fichiers en lisant et écrivant de différentes manières (octet par octet, bit par bit, n bits par n bits, etc…)

Enfin, on a pu implémenter des structures de données qu'on avait auparavant jamais utilisées en pratique (tas, arbre). Cela nous a également permis de développer notre autonomie (pas suivre tout bêtement un TP guidé, par exemple) mais également notre attitude à chercher de manière efficace des ressources sur internet (sur les différentes fonctions, modules ou spécificités de OCaml, par exemple).