Rapport de projet
Compression de texte

Lisa Ceresola et Elisa Ciocarlan

Sommaire

I. Types de données

a) File de priorité

Dans un premier temps, notre but était de créer l'arbre de Huffman. Or, la construction d'un tel arbre repose sur la file de priorité. Nous avons dû trouver un type afin de stocker les caractères ainsi que leurs occurrences. Nous avons fait le choix d'utiliser le type List afin de faciliter l'utilisation et la manipulation des informations récoltées lors de la lecture de notre fichier. Ainsi notre file de priorité qui nous permet de créer l'arbre de Huffman est une liste de paires (occurrence, caractère en ASCII) triée dans l'ordre croissant en fonction de l'occurrence de notre caractère. Ainsi pour créer notre arbre de Huffman il nous suffit de prendre les deux premières paires de notre liste pour créer une nouvelle paire (occurrence, arbre) et ainsi la replacer dans la liste en fonction de la nouvelle occurrence calculée.

b) Arbre de Huffman

L'arbre de Huffman est représenté par un type somme, l'entier stocké dans la feuille correspond au code en ASCII d'un caractère de notre fichier lu préalablement. Notre arbre obtenu est l'arbre final lorsque notre liste est un singleton c'est-à-dire que tous les caractères ont été intégrés à l'arbre de Huffman.

c) Codes donnés par l'arbre de Huffman

Afin de pouvoir stocker les codes de nos caractères après parcours de l'arbre de Huffman nous avons opté pour un tableau de taille 256 de listes initialisées à vide. Ainsi, nous avons ordonné notre tableau de sorte que tab.(caractère en ASCII) nous donne le code calculé grâce à l'arbre de Huffman. Si un caractère n'est pas dans notre fichier, nous aurons alors comme code une liste vide.

Nous avons choisi d'utiliser une liste pour représenter nos codes et non par une chaine de caractères 0 1 car il semblait plus simple de parcourir une liste qu'une chaine de caractères. Or, il est primordial pour la suite de notre projet de pouvoir correctement manipuler les codes afin d'effectuer la compression et la décompression de notre fichier.

d) Choisir le bon caractère lors de la décompression

Lors de la décompression, nous avons parcouru notre fichier afin de reconnaitre des codes et grâce à notre arbre de Huffman, savoir à quel caractère ce code est associé.

Nous avons décidé d'utiliser des List pour stocker notre code. Étant donné que les codes de l'arbre de Huffman sont eux aussi stockés dans des listes, il était donc assez rapide de parcourir nos deux listes en parallèle et savoir à quel caractère le code trouvé appartient.

II. Implémentation de ligne de commande

Nous avons décidé d'expliciter le fonctionnement de la fonction ligne_de_commande () dans le fichier huff.ml car elle occupe une place centrale dans le fonctionnement du programme global et fait appel à toutes les fonctions implémentées dans ce projet. De plus, pour l'implémenter il faut garder en tête plusieurs exceptions, ce qui la rend d'autant plus intéressante.

a) Résultat attendu

Avant de commencer l'implémentation de cette fonction, il a fallu se poser pour réfléchir aux réponses que notre programme doit apporter en fonction des différentes requêtes de l'utilisateur. D'après l'énoncé, on doit exécuter une action en fonction des requêtes suivantes :

1) « huff fichier » : compression de fichier
2) « huff fichier.hf » : décompression de fichier.hf
3) « huff –stats fichier » : compression de fichier et affichage des statistiques liées à cette compression
4) « huff –help » : affichage de l'aide

On remarque ainsi que la ligne de commande doit contenir au moins 3 chaines de caractères délimitées par un espace en comptant la commande d'exécution « ./huff.exe » et au plus 4 chaines de caractères. De plus, une option commence nécessairement par ' -- ' et on distingue un fichier à compresser d'un fichier à décompresser à l'aide de l'extension « .hf ». Cependant on ne peut pas utiliser trop naïvement l'appel String.split_on_char '.' fichier car on va être amené à compresser des fichiers avec des extensions « .txt » par exemple.

b) Code brut

Voici le code de la fonction en question :

```
let ligne_de_commande() =
  if Array.length Sys.argv < 2 then
    Printf.printf "Nombre de paramètres sur la ligne de commande invalide. Veuillez recommencer en entrant dans la ligne de commande 'huff' suivi de l'option souhaitée.\n"
  else
    let file_ok f =
      try (Sys.file_exists f)
      with
        Sys_error f -> false
    in
    let affichage_aide ()=
      Printf.printf "\n                                    ** AIDE **\n
    Les options proposées par ce programme sont les suivantes:
      – huff fichier : pour compresser le fichier donné en argument et obtenir un fichier fichier.hf
      – huff fichier.hf : pour décompresser le fichier donné en argument et obtenir un fichier fichier
      – huff --stats fichier : pour compresser le fichier et aussi afficher des statistiques sur ce dernier
      – huff --help : pour afficher un message d'aide sur les différentes options\n\n"
    in
    let match_file f s=
      let cas_hf nom =
        if not (file_ok nom) then
          Printf.printf "Le fichier \" %s \" n'existe pas\n" nom
        else
          if s=1 then begin
            Printf.printf "Il n'est pas possible de décompresser un fichier en affichant les statistiques, consulter \"huff --help\" pour avoir la liste des options possibles.\n"
          end
          else begin
            Printf.printf "Decompression en cours…\n";
            Huffman.decompress nom
          end
      in
      let cas_f nom =
        if not (file_ok nom) then
```

```
33            Printf.printf "Le fichier \" %s \" n'existe pas\n" nom
34          else
35            if s=0 then begin
36              Printf.printf "Compression en cours…\n";
37              Huffman.compress nom 0
38            end
39            else begin
40              Printf.printf "Compression en cours…\n";
41              Huffman.compress nom 1
42
43            end
44        in
45        match f with
46            name::["hf"] -> cas_hf name
47            | name::ext::["hf"] -> let nom= name^"."^ext in
48                                   cas_hf nom
49            | name::[] -> cas_f name
50            | name::ext::[] -> let nom = name^"."^ext in
51                               cas_f nom
52            |_ -> Printf.printf "Syntaxe invalide. Pour avoir de l'aide, entrer \"huff --help\".\n"
53      in
54      let h = Sys.argv.(1) in
55      if h<>"huff" || Array.length Sys.argv = 2 then
56        Printf.printf "Veuillez entrer dans la ligne de commande \"huff\" suivi de l'option souhaitée.\nPour
    avoir de l'aide, entrer \"huff --help\".\n"
57      else if Array.length Sys.argv = 3 then
58        let f= Sys.argv.(2) in
59        if f="--help" then
60          affichage_aide ()
61        else if f.[0]='-' && f.[1]='-' then
62              Printf.printf "L'option n'a pas été reconnue, pour avoir de l'aide, entrer \"huff --help\".\n"
63              else
64                let file_hf = String.split_on_char '.' f in
65                match_file file_hf 0
66        else
67          let o = Sys.argv.(2) in
68          let f = Sys.argv.(3) in
69          if o <> "--stats" then
70            Printf.printf "L'option n'a pas été reconnue, pour avoir de l'aide, entrer \"huff --help\".\n"
71          else
72            let file_hf = String.split_on_char '.' f in
73            match_file file_hf 1
```

c) Explication du code

Nous allons d'abord expliciter le squelette principal de cette fonction. Afin de récupérer la requête de l'utilisateur, on utilise la fonction Sys.argv qui renvoie un tableau du contenu de la ligne de commande. Si ce dernier est de taille 1, alors l'utilisateur a exécuté le programme sans mentionner

l'option souhaitée et on envoie donc un message d'erreur (l.1). Sinon, on analyse le contenu de la ligne de commande pour réaliser l'action souhaitée.

Pour ce faire, on vérifie que l'utilisateur a bien entré « huff » suivi d'une autre chaîne de caractères correspondant à un nom de fichier ou à une option (l.55-56). Si une de ces conditions n'est pas respectée, on affiche un message d'erreur qui indique la commande « huff - -help » pour que l'utilisateur ait de l'aide.

Sinon on regarde si on est dans le cas où il y au total 3 chaînes de caractères et on récupère la chaine de caractère donnée après « huff » à l'aide de l'appel Sys.argv.(2) (l.57-58). On l'analyse alors : s'il s'agit de l'option « --help », on est dans le cas 4) et on affiche l'aide dans le terminal en appelant la sous-fonction imbriquée affiche_aide() (l.10-17). Sinon, si on observe que l'utilisateur a voulu appeler une option (l.61), alors on affiche dans le terminal que l'option n'a pas été reconnue. On suppose sinon que l'utilisateur a voulu entrer le nom d'un fichier à compresser ou à décompresser et on récupère le nom du fichier dans un tableau en séparant les caractères à l'aide de la fonction String.split_on_char '.' fichier puis ce tableau sera analysé par la fonction match_file fichier 0 qui effectuera l'action demandée. Lorsqu'on appelle cette fonction, on donne en argument l'entier 0 qui signale qu'il ne faudra pas afficher de statistiques.

S'il y a plus de 3 chaînes de caractères, on vérifie qu'il s'agit du cas 4) en récupérant la supposée option à l'indice 2 et le supposé fichier à l'indice 3 (l.77-80). Si l'option récupérée ne correspond pas à « --stats », on envoie un message d'erreur, sinon on appelle la fonction match_file fichier 1 , le 1 indiquant qu'il faudra afficher les statistiques.

Nous allons maintenant donner des précisions sur la sous-fonction imbriquée match_file : string list -> int -> unit (l.18-44) qui prend en argument un tableau du nom du fichier et un entier servant de drapeau qui indique s'il faut afficher les statistiques. On cherche d'abord à déterminer si le fichier est à compresser ou à décompresser en regardant si l'extension « .hf » apparait dans le nom. Pour cela on utilise un match … with (l.45). Si le fichier est à décompresser, on appelle la fonction cas_hf nom, sinon la fonction cas_f nom ou bien on affiche une erreur si la syntaxe est invalide.

Ces sous-fonctions (l.19-30 et l.31-44) ont un fonctionnement très similaire. En effet, on regarde d'abord que le fichier existe dans le répertoire courant à l'aide de la sous-fonction file_ok f (l.7-11) qui fait un try … with en appelant la fonction Sys.file_exists f , et renvoie un booléen true si le fichier existe, false sinon. Si le nom n'est pas valide, on affiche un message d'erreur, sinon on regarde le drapeau pour voir s'il faut afficher des statistiques. Dans la fonction cas_hf nom, on affiche un message d'erreur si le drapeau est à 1 car le programme n'affiche pas de statistiques dans le cas d'une décompression et sinon il s'agit du cas 2) on effectue la décompression en appelant Huffman.decompress nom(l.28). Dans la fonction cas_f nom, on effectue la compression (cas 2) ou 3)) en appelant la fonction Huffman.compress nom 0 (l.37 et 41) et si le drapeau n'est pas à 0 , ie cas 3), on affiche également les statistiques en appelant la fonction Huffman.compress nom 1 (l.42). En effet, l'entier en argument de la fonction compress indique qu'il faut afficher les statistiques s'il est à 1.

III. Jeux de tests

a) Fichiers ajoutés pour tester nos fonctions
   - test.txt : fichier comportant le mot 'satisfaisant' pour vérifier si les résultats trouvés étaient similaires à ceux donnés dans le sujet
   - testvide.txt : fichier vide pour vérifier que nos fonctions ne plantent pas sur un fichier vide
   - test2.txt : fichier en plus de petite taille avec des caractères très variés
   - test3.txt : fichier de taille moyenne

- Amis.txt : fichier plus long pour voir si notre programme n'est pas trop naïf

b) Tests heap.ml

Pour tester les fonctions implémentées dans le fichier heap.ml, nous avons créé une fonction une fonction test let test() = dans le fichier testé. Nous avions créé une fonction let () = Heap.test() dans huff.ml. Ainsi, pour tester, dans le terminal nous avons effectué la commande « ./huff.exe ».

Nous avons dans un premier tester les fonctions de bases implémentées au début afin de savoir si celles-ci étaient fonctionnelles.

Les tests suivants étaient notamment utilisés pour visualiser les codes émis par notre arbre de Huffman afin de savoir si ces derniers étaient corrects et aussi pour nous permettre d'améliorer parfois certaines de nos fonctions. Nous avons donc implémenté différentes fonctions d'affichage, leurs définitions ont été écrites à la fin fichier Heap.mli. Par ailleurs, les tests de ces fonctions ont été effectués avant la finalisation de la ligne de commande.

c) Tests ligne de commande

Afin de tester la ligne de commande, nous avons exécuté dans le terminal les différentes options possibles pour voir si on obtenait bien le résultat attendu. C'est en testant dans le terminal que nous avons notamment pu comprendre le fonctionnement de certaines fonctions telles que Sys.argv.

d) Test compression et décompression

Pour tester nos fonctions de compression et de décompression nous devions dans un premier tester nos fonctions auxiliaires, c'est-à-dire s'assurer que tout compile et que les résultats attendus étaient ceux renvoyés par nos fonctions. Ces tests passés, nous avons donc créer nos fonctions de compression et de décompression. Pour les tester nous avons donc téléchargé ou créé des fichiers textes de tailles variées.

Nous avons utilisé les petits fichiers notamment :

- Pour vérifier si nos arbres étaient corrects grâce à une fonction d'affichage de l'arbre que nous avons implémentée
- Lors de l'implémentation de nos fonctions, il était plus simple de voir s'il y avait des erreurs dans le fichier compressé (à l'aide la commande *xxd -b nomfichier*) notamment pour l'écriture des bits et octets,
- Pour vérifier si notre fichier décompressé est correct (à l'aide de la commande *cat nomfichier).*

Les plus gros fichiers nous ont permis de savoir si nos programmes étaient assez efficaces pour effectuer les taches voulues sur ces fichiers.

En ce qui concerne les commandes utilisées, pour compiler nous avons utilisé *dune build* et pour exécuter nous avons utilisé celles de la ligne de commande.

e) Tests affichage des statistiques

Nous avons testé l'affichage des statistiques en vérifiant que les informations affichées étaient cohérentes. Notamment en ce qui concerne les temps d'exécutions, celui pour l'algorithme de Huffman doit être inférieur à celui pour l'algorithme de compression. Pour vérifier que la taille d'un fichier était correctement lue et affichée, nous avons vérifié dans les propriétés des fichiers que la valeur était identique.

IV. Analyse des statistiques

Nous obtenons les statistiques suivantes sur nos 3 fichiers de « test » :

- Cas 1 : fichier très court contenant « satisfaisant »

```
Statistiques lors de la compression de test.txt :
    - Temps d'exécution de l'algorithme de compression: 0.005065s
    - Temps d'exécution de l'algorithme de huffman: 0.002468s
    - Taille de l'ancien fichier: 12o
    - Taille du nouveau fichier compressé: 13o
    - Taux de compression: 108.333333%
```

- Cas 2 : fichier plutôt court mais avec des caractères très variés

```
Statistiques lors de la compression de test2.txt :
    - Temps d'exécution de l'algorithme de compression: 0.010691s
    - Temps d'exécution de l'algorithme de huffman: 0.005118s
    - Taille de l'ancien fichier: 245o
    - Taille du nouveau fichier compressé: 262o
    - Taux de compression: 106.938776%
```

- Cas 3 : fichier de longueur moyenne contenant un poème, ie des caractères qui varient peu

```
Statistiques lors de la compression de test3.txt :
    - Temps d'exécution de l'algorithme de compression: 0.004206s
    - Temps d'exécution de l'algorithme de huffman: 0.002070s
    - Taille de l'ancien fichier: 5272o
    - Taille du nouveau fichier compressé: 3034o
    - Taux de compression: 57.549317%
```

Nous observons que sur des fichiers courts la taille augmente lors de la compression, ce qui s'explique par le fait que l'arbre de Huffman prend de la place dans le fichier compressé. Alors que sur des fichiers moyens ou longs la taille diminue. Ainsi, sur un fichier de taille moyenne nous obtenons un taux de compression de 57%, ce qui n'est pas négligeable.

En ce qui concerne le temps d'exécution de l'algorithme de compression, nous observons que plus les caractères contenus varient, plus le temps augmente. En effet, on obtient le temps d'exécution le plus important dans le cas n°2 alors qu'il ne s'agit pas du fichier le plus grand parmi les 3 cas présentés. Cela s'explique par le fait que l'arbre d'Huffman est d'autant plus complexe qu'il y a des caractères différents.

En conclusion, on observe que notre algorithme de compression est obsolète sur des fichiers très peu volumineux en raison du fait que nous stockons notre arbre de Huffman dans le fichier compressé. De plus, le temps d'exécution de notre algorithme dépend essentiellement du nombre de caractères différents contenus dans le fichier à compresser et non pas de la longueur du fichier.

V. Répartition du travail

Nous avons décidé de nous repartir le travail de la manière suivante :

- Ligne de commande : Elisa
- Construction de l'arbre de Huffman et récupération des codes : Lisa
- En ce qui concerne la compression et la décompression le travail a été partagé dans l'ensemble, mais plus particulièrement de la manière suivante :
    - Elisa : manipulation des fichiers (ouverture, fermeture, écriture, lecture, création des fichiers compressés et décompressés)
    - Lisa : manipulation de l'arbre de Huffman (sérialisation, désérialisation, trouver les caractères associés aux bits lus du fichier compressé)
- Affichage des statistiques : Elisa