Présentation du projet 2023-2024

XUE Haotian
Chen Shiyao

I. Introduction

Le projet d'ocaml se concentre sur l'exploration et l'implémentation de techniques avancées de manipulation et de compression de données structurées, en particulier les arbres de décision binaires. Ce projet vise à développer des méthodes efficaces pour gérer de grands ensembles de données, en utilisant des structures de données complexes et des algorithmes de compression sophistiqués.

Le cœur du projet repose sur la manipulation de grands entiers et la construction et la gestion d'arbres de décision binaires. Les techniques développées incluent la conversion de données entre différents formats, la construction d'arbres de décision à partir de données brutes, et l'application de règles spécifiques pour la compression des données. L'objectif est de maximiser l'efficacité en termes de temps d'exécution et d'utilisation de la mémoire, tout en assurant un taux de compression élevé.

II. Base de primitives

1. Type de base

```
(**********Question 1.1**********)
type grand_entier = int64 list
```

Ce type représente un grand entier comme une liste d'entiers de type `int64`. Cela permet de gérer des nombres plus grands que ceux que l'on peut stocker dans un seul `int64`.

2. Les primitives

```
let decomposition num = __ (*O(log2(num))*)
```

Convertir un entier `int64` en une liste de booléens représentant sa forme binaire.

Utilisation d'une fonction récursive pour décomposer le nombre. La fonction gère le cas de base (`num = 0L`) et utilise la division et le modulo par 2 pour la décomposition binaire.

```
let completion couple = __ (*O(n*t+n)=>=>O(n*t)*)
```

Compléter une liste binaire jusqu'à une taille donnée avec des `false`.

Ajoute des `false` à la liste jusqu'à ce que sa taille atteigne la valeur spécifiée.

```
let composition lb = (*O(n) : pas prendre en compte la complexité List.rev car il n'est appelée qu'une seule fois au début de la fonction aux *)
```

Convertir une liste de booléens en un entier `int64`.

Parcourt la liste en construisant l'entier, en doublant la valeur accumulée et en ajoutant 1 si l'élément de la liste est `true`.

```
let table x n = completion ((decomposition x),n)
```

Convertir un entier en une liste binaire de taille fixe.

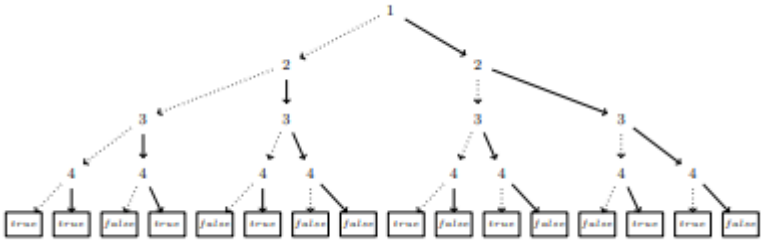
Utilise `decomposition` pour obtenir la forme binaire, puis `completion` pour ajuster la taille de la liste.

```
let genAlea n =  (*O(n) car sa complexité est linéaire et de l'ordre de O(n) dans tt les conditions *)
```

Générer une liste binaire aléatoire de taille donnée.

Génère des nombres aléatoires et les convertit en binaires, en gérant les cas où la taille est supérieure à 32 et 64 pour éviter les dépassements.

III. Arbre de décision



3. Type de l'arbre

```
type decision_binary_tree =
  | Leaf of bool
  | Node of int * decision_binary_tree * decision_binary_tree
```

4. Construction

Pour construire une arbre de décision, on fait d'abord construire une arbre "vide" (tous les feuilles sont initialisées par false)

```
let cons_vide_arbre n = __ (*O(2^n) ici n =hauteur de arbre, alors la complexité de la création de l'arbre est de l'ordre de O(2^n), *)
```

```
let rec log2_arete n = (*O(log2(n))*)
```

Calculer une valeur approchée du logarithme en base 2 d'un nombre, spécifiquement adaptée pour déterminer la hauteur d'un arbre binaire.

```
let arete_arbre lb = log2_arete (List.length lb)
```

Calcule la hauteur d'un arbre binaire basée sur la longueur d'une liste binaire.

```
let cons_chemin_list lb = (*O(n*log2(n))*)
```

Construire une liste de chemins dans un arbre binaire:

- La fonction utilise une approche récursive pour construire une liste de chemins.
- Pour chaque élément `hd` dans la liste `lb`, elle calcule un chemin binaire correspondant basé sur l'indice `cpt` de cet élément dans la liste.
- `decomposition (Int64.of_int cpt)` convertit l'indice `cpt` en sa représentation binaire.
- `completion` est utilisée pour compléter la représentation binaire à une longueur fixe, déterminée par `log2_arete length`.
- `List.rev` est utilisée pour inverser la liste, assurant que les chemins sont construits dans le bon ordre.

```
let rec inserer chemin_bool arbre = __ (*O(log(n)) ou n =nb de noeud*)
```

Insérez un élément dans un arbre en utilisant un chemin booléen. L'argument 'chemin_bool' est de type 'bool list * bool', ce qui signifie que on insère un booléen sélectionné via un chemin spécifié par une liste de booléens ('bool list').

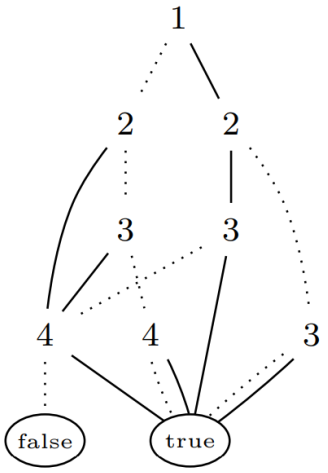
```
let cons_arbre lb = __ (*O(n*log(n) + log(n) + 2^n + nlog(n))==>O(2^n + nlog(n))==> O(2^n)*)
```

La fonction 'inserer' sert à insérer un seul élément dans l'arbre, tandis que 'cons_arbre' met à jour l'arbre avec tous les éléments d'une liste.

```
let liste_feuilles arbre = (*O(n) *)
```

Cette fonction permet d'obtenir une liste contenant tous les éléments booléens situés dans les feuilles de l'arbre.

IV. Compression par liste




```
type node_id_type = GE of grand_entier | BL of bool

type graphe_node = {id:node_id_type }

type elements = {
    entier: node_id_type;
    node_l: graphe_node ref;
    node_r: graphe_node ref;
    depth: int
}
type liste_deja_vus = elements list
```

Dans cette structure “elements”, quatre éléments ont été définis : 'entier', 'node_l', 'node_r' et 'depth'. 'Entier' sert à identifier l'élément, 'node_l' permet d'accéder à son sous-arbre gauche, 'node_r' à son sous-arbre droit, et 'depth' indique la profondeur de cet élément.

```
let rec sup_prefix0 l =(*O(n)*)
```

La fonction OCaml `sup_prefix0` semble bien écrite pour supprimer les zéros en préfixe d'une liste de longs entiers (`int64`, représentés par `0L`, `1L`, etc.).

```
let liste_feuille_to_ge lf =  __ (*O(n*t/64)ici t=n)==>O(n²) *)
```

Transformer une liste de valeurs booléennes en une liste de nombres `int64`.

```
let decomposition_ge ge_num_liste = (*O(n²+n*log(num)) num=val max*)
```

Transformer une liste de nombres `int64` en une liste de valeurs booléennes.

```
let rec check_ldv n ldv = (*O(n) ici n = longueur de ldv*)
```

Cette fonction sert à vérifier si 'n' figure dans la liste 'liste_déjà_vus' et à retourner un booléen.

```
let dbt_to_liste_deja_vus dbt = (*O(n^3)*)
```

Transformer un arbre de décision en une liste des éléments déjà vus. Enregistrer tous les états dans la liste_déjà_vus.

```
let z_sup ldv_ref e_sup replace= (*O(n) où n=len(ldv_ref)*)
```

Selon la règle Z, si un nœud a un sous-arbre droit évalué à 'false', nous allons le connecter à son sous-arbre gauche et ensuite supprimer ce nœud.

```
let rec maj_regle_Z ldv_ref = (*O(n²) où n=len(ldv_ref)*)
```

Cette fonction sert à mettre à jour la liste 'liste_déjà_vus' selon la règle Z.

```
let maj_regle_M ldv_ref = (*O(n^3) car parcourt 2 fois et O(list.mem)=O(n)*)
```

Selon la règle M, nous allons vérifier que tous les éléments de la liste 'liste_déjà_vus' sont distincts.

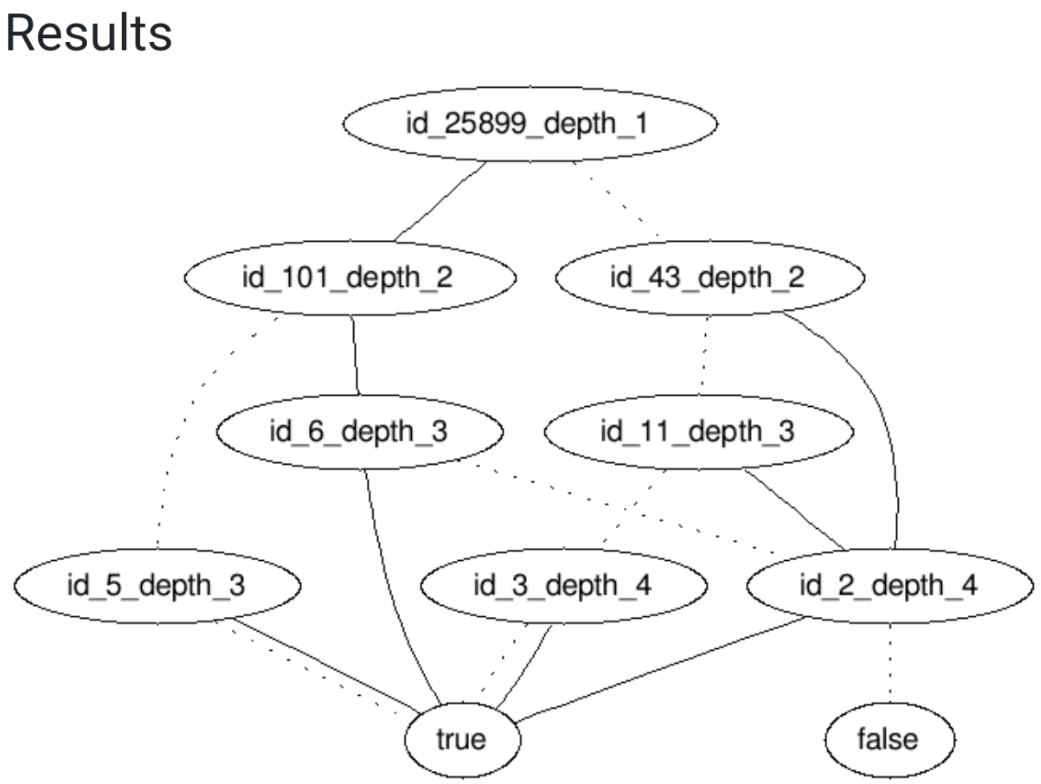
```
let compressionParListe arbre_decision = (*O(n^3)*)
```

La fonction 'compressionParListe' a pour but d'appliquer les deux fonctions 'maj_regle_M' et 'maj_regle_Z'.

```
let cons_graphe_dot_string_liste arbre_de_decision =  __
```

```
let dot fichier arbre_de_decision =  =  __
```

Ces fonctions ont pour but de créer un fichier texte contenant la représentation du graphe de 'liste_déjà_vus', formatée selon la syntaxe de GraphViz.



V. Compression par arbre

```
type arbre_deja_vus =
  | Noeud of elements * arbre_deja_vus * arbre_deja_vus
  | Feuille
```

La structure mentionnée ci-dessus est similaire à 'liste_deja_vus'.

```
let  add_true_false adv = (*O(n) n=nb noeud d'arbre*)
```

Cette fonction a pour but d'ajouter des feuilles 'true' et 'false' dans l'arbre 'arbre_deja_vus'.

```
let dbt_to_arbre_deja_vus dbt = (*O(n^3)*)
```

Cette fonction sert à construire l'arbre 'arbre_deja_vus' en se basant sur l'arbre de décision.

```
let rec is_in_tree ge tree = (*O(n)*)
```

Cette fonction a pour but de vérifier la présence de l'élément 'ge' dans l'arbre 'arbre_deja_vus'.

```
let inserer_node_in_tree node tree = (*O(n+2^N)*)
```

Cette fonction est conçue pour insérer un nœud de type 'éléments' dans l'arbre 'arbre_deja_vus'.

```
let regle_M adv = __ (*O(n²)*)
```

Conformément à la règle M, nous vérifions que tous les éléments de l'arbre 'arbre_deja_vus' sont distincts.

```
let to_list adv = (*O(n) n=nb noeud de adv*)
let supp_in_list l ge = (*O(n)*)
let to_adv list = (*O(n)*)
```

Nous avons rencontré des difficultés pour implémenter la règle Z dans le cas de l'arbre. Par conséquent, nous avons utilisé la règle Z adaptée aux listes : nous avons transformé 'arbre_deja_vus' en 'liste_deja_vus', appliqué la règle Z sur cette liste, puis retransformé 'liste_deja_vus' en 'arbre_deja_vus'.

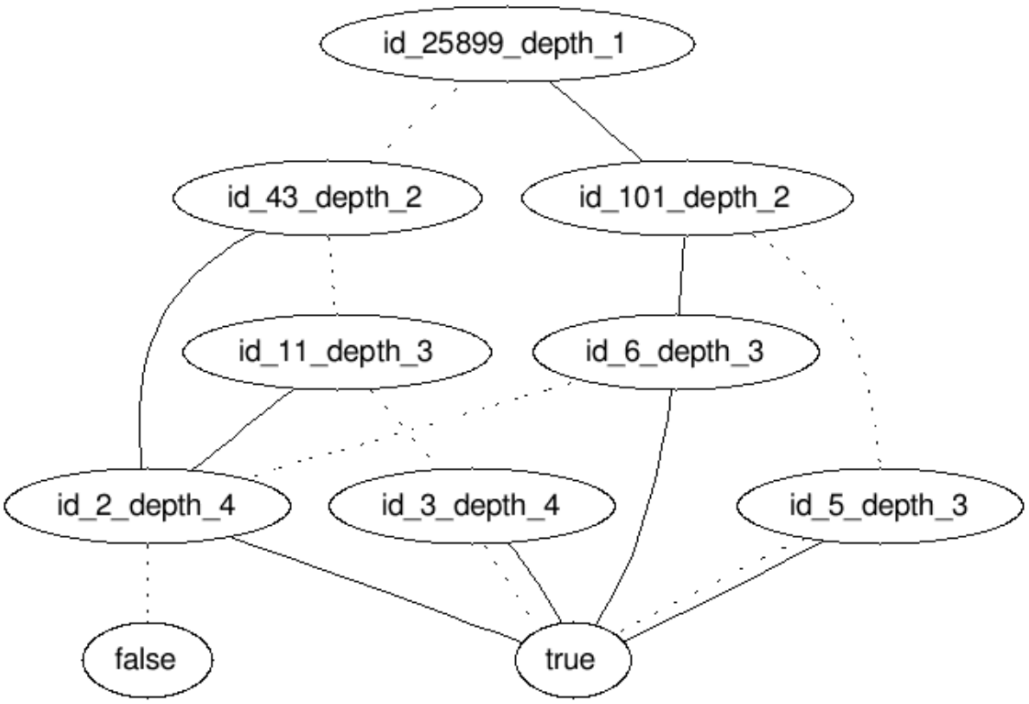
```
let compressionParArbre arbre_decision = (O(n^3)*)
```

La fonction compressionParArbre sert également à appliquer la fonction règle M et règle Z.

```
let cons_graphe_dot_string_arbre arbre_de_decision = __
```

Ces fonctions ont pour but de créer un fichier texte contenant la représentation du graphe de 'arbre_déjà_vus', formatée selon la syntaxe de GraphViz.

Results



VI. Mesures

```
let measure_time f = (*O(n^3)*)
```

Cette fonction permet de mesurer le temps d'exécution de la fonction choisie.

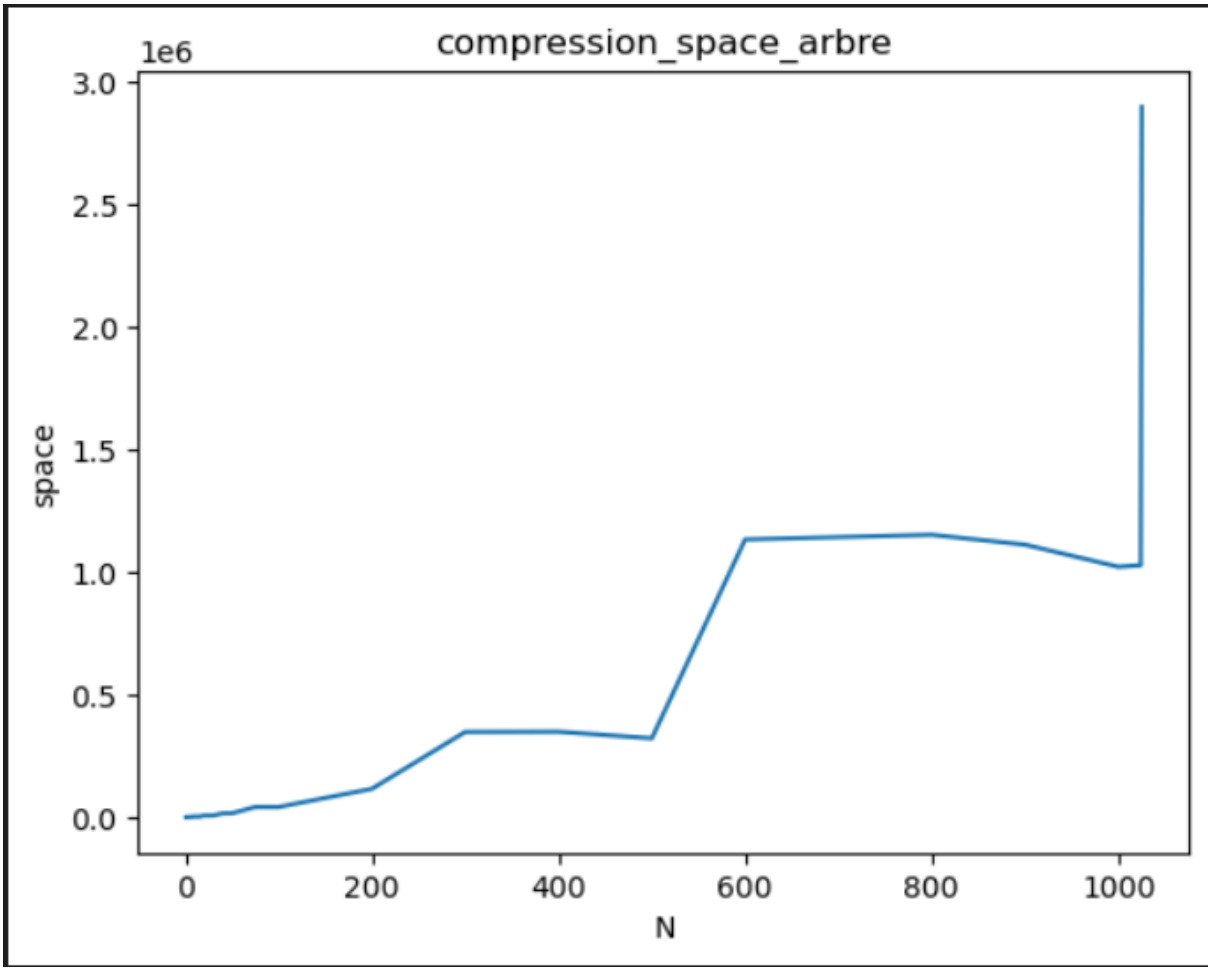
```
let measure_memory_use_by_list f = (*O(f)*)
```

Cette fonction permet de mesurer le mémoire d'exécution de la fonction choisie.

```
let count_node dbt = (*O(n) n=nb de noeud*)
let count_noeud ele_list = (*O(n) n=length de la liste ele_list*)
let taux_compression dbt = (*O(n)*)
```

Ces fonctions permettent de mesurer et calculer le taux de compression.

VII. Rapport



Nous avons mesuré le temps d'exécution, la consommation d'espace et le taux de compression de la fonction de compression (par liste, par arbre) en prenant le nombre de nœuds d'un arbre de décision binaire comme variable indépendante. À travers les données et les graphiques, on peut observer que :

1. Lorsque le nombre de nœuds augmente de 2^k à 2^k+1, le temps et la consommation d'espace augmentent significativement, et le taux de compression augmente de manière substantielle. Cela indique que la taille de l'arbre double à ce moment et génère un grand nombre de nœuds false, qui doivent tous être compressés.

2. Lorsque le nombre de nœuds augmente de 2^k+1 à 2^(k+1), l'utilisation du temps et de l'espace reste relativement stable, mais on peut constater que le taux de compression diminue lentement. Cela suggère que sous la même échelle de couches de l'arbre, plus il y a de nœuds, plus le taux de compression est bas, parce que plus le nombre de nœuds se rapproche de 2^k, plus la proportion de true est grande, et donc l'espace nécessaire à la compression devient plus petit.

3. On peut constater qu'en utilisant la méthode de compression d'arbre, l'espace et le temps occupés sont toujours légèrement inférieurs à l'utilisation de la méthode de compression de liste chaînée, ce qui montre que l'utilisation d'arbre est plus efficace.

VIII. Conclusion

Ce projet a démontré avec succès l'efficacité des méthodes de compression par liste et par arbre dans la gestion des arbres de décision binaires. Les résultats obtenus montrent une amélioration significative en termes de temps d'exécution, d'utilisation de la mémoire et de taux de compression, en particulier lorsque les données sont structurées de manière complexe. L'utilisation de la compression par arbre s'est avérée légèrement plus efficace que la compression par liste, offrant une meilleure performance globale.

Ces découvertes ouvrent la voie à de nouvelles recherches et applications dans le domaine de la manipulation de données complexes. L'approche adoptée dans ce projet peut être étendue à d'autres types de structures de données et utilisée dans divers contextes où la gestion efficace de grandes quantités de données est cruciale. En conclusion, ce projet a non seulement atteint ses objectifs initiaux mais a également établi une base solide pour des avancées futures dans le domaine de la compression et de la gestion de données structurées.