Rapport de Projet

UE Ouverture

Écrit par :

MORANDEAU Timothée
CROUZET Oriane
AHMED Youra

Parcours : STL
Groupe : 1

Chargés de TP/TD :

Vincent Botbol
Antoine Genitrini

30 décembre 2024

Table des matières

1 Introduction

La programmation et la manipulation d'expressions algébriques sont au cœur de nombreux domaines informatiques, allant des calculs scientifiques aux systèmes de calcul symbolique. Ce projet s'inscrit dans cet objectif et vise à modéliser, transformer, et analyser des polynômes sous deux formes distinctes : linéaire et arborescente.

Le travail se divise en deux grandes parties : la manipulation des structures de données représentant les polynômes sous forme linéaire et arborescente, puis une phase d'expérimentation portant sur la performance et l'efficacité des algorithmes qui ont été développés. Dans ce rapport, nous allons détailler les choix conceptuels, les implémentations et les résultats obtenus, tout en mettant en évidence les défis rencontrés et les perspectives d'amélioration.

2 Manipulation des polynômes sous forme linéaire

2.1 Représentation

Nous avons un type **Polynome**. Ce type nous permet de représenter les polynômes sous une forme **chaînée**, où chaque terme est lié au suivant grâce au champ **suite**.

Nous avons un type **modulaire et flexible** car chaque terme du polynôme est encapsulé dans une structure indépendante, avec son coefficient et son degré. Cela facilite la manipulation de chaque terme individuellement et permet de construire des polynômes de manière incrémentale.

Illustration de la Représentation des Polynômes

Un terme $3x^2$ sera représenté comme :

```
{ coef = 3; degree = 2; suite = None }
```

Un polynôme complet comme $3x^2 + 5x + 1$ sera construit par chaînage :

```
{ coef = 3; degree = 2;
  suite = Some { coef = 5; degree = 1;
                 suite = Some { coef = 1; degree = 0;
                                suite = None } } }
```

2.2 Forme Canonique

La forme canonique d'un polynôme est une représentation standardisée où :

— Les termes sont triés par degrés croissants.

— Les termes ayant le même degré sont combinés en sommant leurs coefficients.

— Les termes avec un coefficient nul sont supprimés.

Cette standardisation est essentielle pour plusieurs raisons :

1. Elle garantit une représentation unique d'un polynôme, ce qui est crucial pour les comparaisons ou les manipulations algébriques.
2. Elle simplifie les opérations comme l'addition ou la multiplication de polynômes en évitant des redondances ou des termes inutiles.
3. Elle réduit la complexité des calculs en éliminant les monômes nuls et en limitant la taille des structures à manipuler.

La fonction `canonique` implémente cette transformation en plusieurs étapes clés :

Étapes de la transformation

1. **Conversion du polynôme en liste de tuples**
   Cette étape est réalisée par la fonction interne `to_list`, qui parcourt récursivement le polynôme pour produire une liste de couples (coefficient, degré). Cela permet de manipuler le polynôme sous une forme linéaire et plus facile à traiter.
2. **Tri par degré croissant**
   La liste obtenue est triée selon les degrés des termes en utilisant la fonction `List.sort`. Le tri garantit que les termes sont dans l'ordre attendu pour les étapes suivantes.
3. **Combinaison des termes de même degré**
   Une fonction interne `combine` parcourt la liste triée pour :
   — Additionner les coefficients des termes ayant le même degré.
   — Supprimer les termes dont le coefficient résultant est nul.
4. **Reconstruction du polynôme**
   Enfin, la fonction `from_list` reconstruit le polynôme sous forme chaînée en utilisant la liste combinée et triée. Chaque couple (coefficient, degré) est transformé en un terme du polynôme, relié au suivant via le champ `suite`.

Complexité de la transformation

— **Conversion en liste (`to_list`) :** Cette étape parcourt tous les termes du polynôme, avec une complexité de $O(n)$, où n est le nombre de termes.
— **Tri (`List.sort`) :** Le tri utilise un algorithme de complexité $O(n \log n)$.
— **Combinaison des termes (`combine`) :** Cette étape parcourt la liste triée une seule fois, avec une complexité de $O(n)$.
— **Reconstruction (`from_list`) :** Cette étape est également linéaire, avec une complexité de $O(n)$.

Ainsi, la complexité globale de la fonction `canonique` est dominée par l'étape de tri, soit $O(n \log n)$.

Exemple

Prenons le polynôme non standardisé suivant :

$$3x^2 + 5x + 1 + 2x^2 + 0x + 4$$

1. Conversion en liste :

$$[(3, 2), (5, 1), (1, 0), (2, 2), (0, 1), (4, 0)]$$

2. Tri par degré :
$$[(1,0),(4,0),(5,1),(0,1),(3,2),(2,2)]$$
3. Combinaison des termes :
$$[(5,0),(5,1),(5,2)]$$
4. Reconstruction du polynôme :
$$5+5x+5x^2$$

Ainsi la forme canonique assure une représentation standard, optimisée et claire des polynômes, facilitant ainsi leur manipulation.

2.3 Fonctions de Calcul : Addition et Multiplication de Polynômes

Nous avons des fonctions implémentées permettant l'**addition** et la **multiplication** de deux polynôme.

2.3.1 Addition de Polynômes : `poly_add`

La fonction `poly_add` permet d'additionner deux polynômes canonique représentés sous forme chaînée. Voici son fonctionnement détaillé :

— **Cas de base :** Si l'un des deux polynômes est `None`, la fonction retourne directement l'autre polynôme. Cela assure que l'addition fonctionne même avec des polynômes vides.

— **Addition par degré :** Lorsque les degrés des termes sont égaux, leurs coefficients sont additionnés. Si la somme des coefficients est nulle, le terme est ignoré pour conserver une forme compacte.

— **Ordonnancement :** Si les degrés diffèrent, le terme avec le degré supérieur est directement ajouté au résultat, et l'addition continue avec les termes restants.

Complexité

La fonction parcourt chaque terme des deux polynômes une seule fois, ce qui donne une complexité linéaire en $\mathcal{O}(n+m)$, où n et m représentent respectivement les tailles des deux polynômes.

2.3.2 Multiplication de Polynômes : `polyprod`

La fonction `poly_prod` effectue la multiplication de deux polynômes canonique à l'aide de deux sous-fonctions :

— `multibyterm` : Cette fonction multiplie chaque terme d'un polynôme par un terme donné (coefficient et degré), et retourne le polynôme résultant.

— `mulandadd` : Cette fonction parcourt un polynôme terme par terme, applique `multibyterm` pour multiplier ce terme avec l'autre polynôme, puis combine les résultats intermédiaires à l'aide de `poly_add`.

Complexité

La multiplication de chaque terme de P_1 avec tous les termes de P_2 donne une complexité quadratique en $\mathcal{O}(n \cdot m)$, où n et m sont les tailles des deux polynômes. L'utilisation de `poly_add` pour combiner les résultats intermédiaires est linéaire par appel, mais l'impact global est dominé par la multiplication quadratique.

3 Manipulation des Polynômes sous Forme Expression Arborescente

3.1 Représentation

Pour manipuler les polynômes sous forme arborescente, nous avons défini un type `expr` qui permet de modéliser les différentes composantes des expressions algébriques. Ce type repose sur une structure récursive qui reflète directement la grammaire des expressions.

Définition du Type `expr`

Le type `expr` est défini de la manière suivante :

```
type expr =
  | Int of int              (* Constante entiere *)
  | Var of string           (* Variable, typiquement "x" *)
  | Pow of expr * int       (* Puissance d'une expression *)
  | Add of expr list        (* Addition d'une liste d'
     expressions *)
  | Mul of expr list        (* Multiplication d'une liste d'
     expressions *)
```

Cette définition permet de représenter les polynômes comme des arbres où :

— Les nœuds internes représentent des opérateurs comme `+`, `*`, ou `^`.

— Les feuilles représentent des constantes (`Int`) ou des variables (`Var`).

— Les opérations binaires ou n-aires, comme l'addition et la multiplication, sont modélisées par des listes d'expressions (`Add` et `Mul`).

Illustration

Par exemple :

— L'expression $3x^2 + 5x + 1$ est représentée comme :

```
Add [
  Mul [Int 3; Pow (Var "x", 2)];
  Mul [Int 5; Var "x"];
  Int 1
]
```

— L'expression $(x + 1)^2 \times (2x + 3)$ est représentée comme :

```
Mul [
  Pow (Add [Var "x"; Int 1], 2);
  Add [Mul [Int 2; Var "x"]; Int 3]
]
```

En résumé, la représentation des polynômes sous forme d'expressions arborescentes est une approche puissante pour capturer la structure algébrique de manière claire et efficace.

3.2 Transformation d'une Expression Arborescente en Polynôme Canonique

La fonction `arb2poly` permet de convertir une expression arborescente, représentée par le type `expr`, en un polynôme canonique sous forme chaînée. Cette conversion est essentielle pour uniformiser les représentations et permettre des opérations algébriques cohérentes sur les polynômes.

Principe de Fonctionnement

- **Cas de base :** Les constantes (`Int`) et les variables (`Var "x"`) sont directement traduites en termes de polynôme avec des degrés et coefficients appropriés.
- **Puissances :** Les termes de type `Pow (Var "x", d)` sont interprétés comme un monôme avec le degré d.
- **Combinaisons d'expressions :**
  - Les listes d'expressions dans `Add` sont combinées en un polynôme unique grâce à la fonction `poly_add`.
  - Les listes dans `Mul` utilisent la fonction `poly_prod` pour effectuer des multiplications successives.

Complexité

La complexité de la fonction `arb2poly` dépend de la structure de l'arbre d'expression
Chaque nœud de l'arbre est visité une seule fois, ce qui donne une complexité en O(n), où n est le nombre total de nœuds

Les opérations comme `poly_add` ou `poly_prod`, utilisées lors de la combinaison des expressions, ajoutent une complexité proportionnelle à la taille des polynômes qu'elles manipulent

Exemple

Prenons l'exemple d'une expression comme $3x^2 + 5x + 1$ représentée sous forme arborescente. Après application de la fonction `arb2poly`, l'expression est transformée en un polynôme chaîné correspondant :

$$\{\text{coef} = 3; \text{degree} = 2; \text{suite} = \text{Some}\{\text{coef} = 5; \text{degree} = 1;$$

$$\text{suite} = \text{Some}\{\text{coef} = 1; \text{degree} = 0; \text{suite} = \text{None}\}\}\}$$

4 Synthèse d'expressions arborescentes

La synthèse d'expressions arborescentes repose sur la capacité à manipuler et à redistribuer des éléments de manière aléatoire. Nous avons une fonction `extraction_alea` qui nous permet de sélectionner un élément aléatoire dans une liste L et de l'ajouter en tête d'une autre liste P.

4.1 Extraction aléatoire

Principe

`extraction_alea` fonctionne comme suit :

— Un élément est choisi aléatoirement dans la liste L.

— Cet élément est retiré de L et ajouté en tête de P.

— Le processus peut être répété pour transformer totalement L et remplir P.

Complexité

La complexité de cette fonction est $\mathcal{O}(n)$, où n est la taille de la liste L, car il est nécessaire de parcourir L pour identifier et retirer l'élément sélectionné.

Nous avons aussi la fonction `gen_permutation` qui s'appuie sur `extraction_alea` pour produire une permutation aléatoire et uniforme d'une liste d'entiers.

4.2 Permutations

Principe

`gen_permutation` prend en entrée un entier n et génère une liste L contenant les entiers de 1 à n. Elle crée une liste vide P qui sera remplie progressivement par des éléments extraits aléatoirement de L.

— À chaque itération, un élément est retiré de L (grâce à `extraction_alea`) et ajouté en tête de P.

— Ce processus est répété jusqu'à ce que L soit vide, et P contiendra alors une permutation aléatoire de 1 à n.

Cette fonction nous permet de générer des entrées aléatoires dans la construction d'arbres binaires de recherche (ABR) ou d'expressions arborescentes, où l'ordre d'insertion influence la structure résultante.

Complexité

— Chaque appel à `extraction_alea` a une complexité linéaire $\mathcal{O}(|L|)$.

— Le nombre total d'éléments à traiter est n, donc la complexité globale est $\mathcal{O}(n^2)$.

Exemple

Pour $n = 4$, la fonction initialisera :

$$L = [1, 2, 3, 4], \quad P = []$$

Après plusieurs itérations, par exemple :

$$L = [1, 3, 4], \quad P = [2]$$

$$L = [1, 4], \quad P = [3, 2]$$

$$L = [], \quad P = [1, 4, 3, 2]$$

La sortie sera donc $P = [1, 4, 3, 2]$, une permutation aléatoire de $1, 2, 3, 4$.

4.3 Construction d'un Arbre Binaire de Recherche (ABR)

La fonction `list2abr` est utilisée pour construire un Arbre Binaire de Recherche (ABR) à partir d'une liste d'entiers. Un ABR est une structure de données arborescente où chaque nœud respecte les propriétés suivantes :

- Les valeurs des nœuds dans le sous-arbre gauche sont inférieures à la valeur du nœud courant.
- Les valeurs des nœuds dans le sous-arbre droit sont supérieures à la valeur du nœud courant.

Principe

1. La fonction `list2abr` utilise une fonction auxiliaire `elemInAbr` pour insérer chaque élément de la liste dans l'arbre.
2. Pour chaque élément :
   - Si l'arbre est vide (`Empty`), un nouveau nœud est créé.
   - Sinon, on compare l'élément à la valeur du nœud courant :
     - Si l'élément est plus petit, il est inséré dans le sous-arbre gauche.
     - S'il est plus grand, il est inséré dans le sous-arbre droit.
     - Si l'élément est égal, il n'est pas inséré (pour éviter les doublons).
3. Une fonction récursive `list2abr_aux` parcourt la liste et insère chaque élément successivement dans l'arbre en utilisant `elemInAbr`.

Complexité

- Pour chaque élément, l'insertion dans l'arbre prend un temps proportionnel à la hauteur de l'arbre.
- Dans le meilleur des cas (arbre équilibré), la hauteur est logarithmique, donc la complexité totale est $\mathcal{O}(n \log n)$, où n est la taille de la liste.
- Dans le pire des cas (arbre dégénéré), la hauteur est linéaire, donc la complexité devient $\mathcal{O}(n^2)$.

Cette fonction est essentielle pour les étapes suivantes, comme l'étiquetage et la manipulation d'expressions arborescentes.

4.4 Étiquetage d'un ABR

La fonction `etiquetage` transforme un Arbre Binaire de Recherche (ABR) en une structure d'arbre étiqueté. L'objectif est de convertir chaque nœud de l'ABR en un nœud étiqueté, en respectant des règles précises, afin de générer des arbres utilisables pour représenter des expressions mathématiques.

Type de l'Arbre Étiqueté

Un arbre étiqueté est défini comme suit :

— `V` représente un nœud vide.

— `Node(s, g, d)` représente un nœud contenant une chaîne de caractères s (l'étiquette), un sous-arbre gauche g, et un sous-arbre droit d.

Principe de la Fonction `etiquetage`

La fonction applique les règles suivantes pour étiqueter chaque nœud de l'ABR :

1. **Cas des feuilles :** Si un nœud est vide (`Empty`), il est converti en `V`.
2. **Nœuds avec deux enfants feuilles :**
   — Si l'étiquette e du nœud est paire, le nœud est étiqueté avec "", avec un enfant gauche "x" et un enfant droit contenant un entier aléatoire compris entre 0 et 100.
   — Si e est impaire, le nœud est étiqueté avec "$*$", avec un enfant gauche contenant un entier aléatoire compris entre -200 et 200 et un enfant droit "x".
3. **Nœuds ayant au moins un enfant non feuille :**
   — Les sous-arbres gauche et droit sont étiquetés récursivement.
   — Le nœud est étiqueté avec "+" avec une probabilité de 75%, ou avec "$*$" avec une probabilité de 25%.
4. **Sous-nœuds vides :** Lorsqu'un enfant est vide, il est remplacé par une feuille étiquetée avec un entier aléatoire (avec une probabilité de 50%) ou la variable "x" (avec une probabilité de 50%).

Complexité

La complexité de `etiquetage` est $\mathcal{O}(n)$, où n est le nombre de nœuds dans l'arbre. Chaque nœud est visité une seule fois pour effectuer les transformations nécessaires.

Exemple

Prenons un ABR simple construit à partir de la liste $[2, 1, 3, 4]$. L'ABR correspondant est :

Node(2, Node(1, Empty, Empty), Node(3, Empty, Node(4, Empty, Empty)))

Après étiquetage, l'arbre étiqueté pourrait ressembler à :

Node("+", Node("*", Node("-15", V, V),

$Node(\text{"}x\text{"}, V, V)), Node(\text{"}\text{"}, Node(\text{"}x\text{"}, V, V), Node(\text{"}42\text{"}, V, V)))$

Cet étiquetage produit une structure prête pour des manipulations algébriques, comme la génération d'expressions ou la simplification.

4.5 Transformation d'un ABR en une Expression Arborescente

La fonction `gen_arb` est conçue pour transformer un arbre étiqueté, généré à partir d'un ABR étiqueté, en une expression arborescente qui respecte la grammaire définie.

Principe de la Fonction `gen_arb`

La fonction `gen_arb` effectue une transformation récursive de l'arbre étiqueté en suivant les étapes suivantes :

1. Si le nœud est vide (`V`), une erreur est levée car il ne peut être transformé en expression.
2. Si le nœud est un opérateur (`+`, `*`, ou `^`), les sous-arbres gauche et droit sont transformés récursivement en expressions, et les fonctions utilitaires (`verif_add`, `verif_mul`, `verif_pow`) sont utilisées pour garantir la conformité à la grammaire.
3. Si le nœud contient la variable `x`, une expression de type `Pow(Var "x", 1)` est créée pour assurer une représentation cohérente.
4. Si le nœud contient un entier, il est directement transformé en une constante (`Int`).

Fonctions Utilitaires

Les fonctions `verif_add`, `verif_mul`, et `verif_pow` sont utilisées pour gérer les cas spécifiques et éviter les redondances ou incohérences dans l'arbre final. L'objectif est de retrouver une strucure qui respecte la **grammaire de la section 1.2** :

$$E = int \mid E_{\wedge} \mid E_{+} \mid E_{*}$$
$$E_{\wedge} = x \wedge int_{+}$$
$$E_{+} = (E \setminus E_{+}) + (E \setminus E_{+}) + ...$$
$$E_{*} = (E \setminus E_{*})(E \setminus E_{*})...$$

— `verif_add` fusionne les listes d'expressions pour les nœuds d'addition.

— `verif_mul` fusionne les listes d'expressions pour les nœuds de multiplication.

— `verif_pow` garantit que les puissances sont valides (exposants strictement positifs) et simplifie les structures inutiles comme `Pow(Pow(...), ...)`.

Complexité

La complexité de `gen_arb` est $\mathcal{O}(n)$, où n est le nombre de nœuds dans l'arbre étiqueté. Chaque nœud est visité une seule fois pour être transformé en une expression arborescente.

Exemple

Prenons l'arbre étiqueté suivant, généré à partir de la liste $[2, 1, 3, 4]$:

$$\text{Node("+", Node("*", Node("-15", V, V),}$$

$$Node(”x”, V, V)), Node(”, Node(”x”, V, V), Node(”42”, V, V)))$$

Après application de `gen_arb`, cet arbre est transformé en une expression arborescente :

$$\text{Add([Mul([Int(-15), Pow(Var "x", 1)]), Pow(Var "x", 42)])}$$

Ce résultat respecte la grammaire définie et peut être directement utilisé pour des calculs ou des manipulations ultérieures.

5 Expérimentations

Pour calculer la somme (resp. le produit) des polynômes issus des arbres générés, nous avons implémenté 4 méthodes différentes :

- **Naïf** : une méthode simple à comprendre et à implémenter, mais qui s'avère peu efficace sur les grandes listes ($\mathcal{O}(n^2)$)
- **Diviser pour régner** : plus complexe à implémenter que la version naïve, mais bien plus efficace sur les listes de grandes tailles ($\mathcal{O}(n\ log\ n)$)
- **Accumulateur** : une méthode simple qui utilise une récursion terminale, mais qui reste un peu moins efficace que diviser pour régner ($\mathcal{O}(n * m)$)
- **Fold** : un style fonctionnel et concis qui utilise `List.fold_left`, mais plus difficile à implémenter et à comprendre ($\mathcal{O}(n * m)$)

Résultats et Analyse

1. Temps d'exécution pour les sommes (ABR de taille fixe)

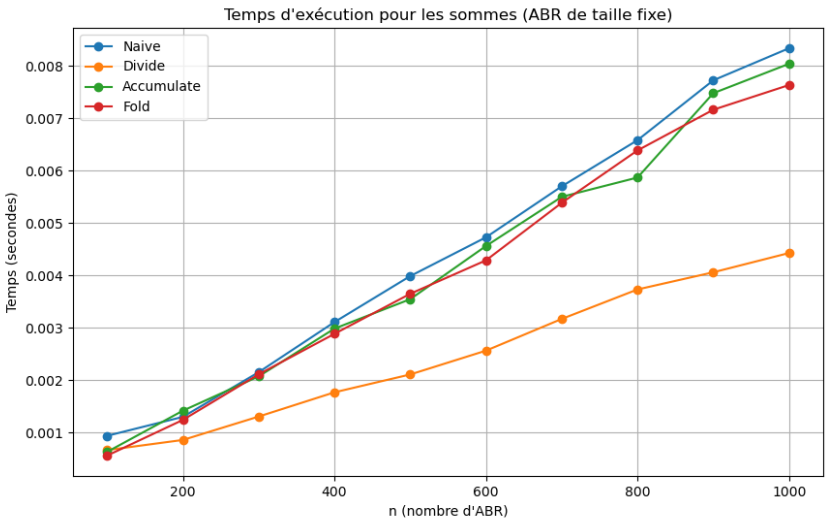

FIGURE 1 – Temps d'exécution pour les sommes avec des ABR de taille fixe.

Sur la figure 1, on observe les performances des différentes stratégies de calcul pour des sommes de polynômes sur des ABR de taille fixe (20 nœuds par arbre) :

— **Naive** et **Fold** montrent des performances similaires avec des temps d'exécution légèrement supérieurs pour les grands nombres d'ABR.

— **Divide** est plus rapide pour les tailles intermédiaires, mais sa croissance devient plus marquée pour des tailles importantes.

— **Accumulate** offre des performances compétitives, restant légèrement en dessous de **Naive** et **Fold**, ce qui reflète une gestion optimisée des accumulations dans la structure.

2. Temps d'exécution pour les produits (ABR de taille fixe)

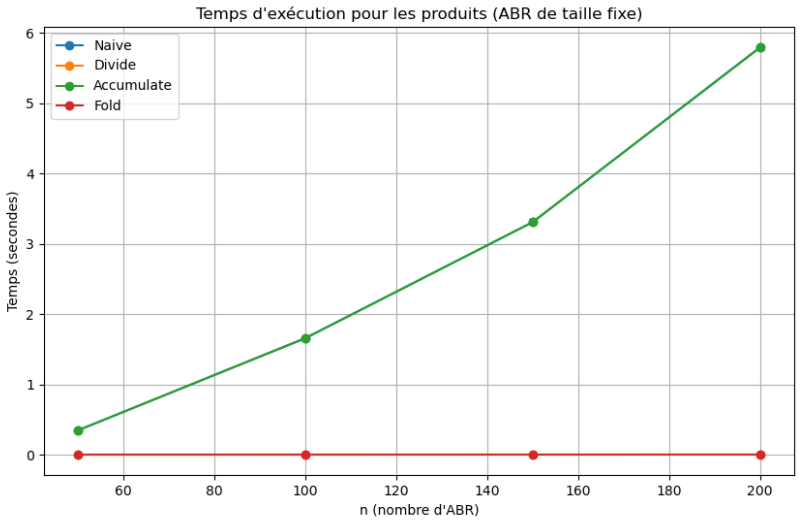

FIGURE 2 – Temps d'exécution pour les produits avec des ABR de taille fixe.

Sur la figure 2, on observe les temps de calcul pour les produits de polynômes :

— **Naive** et **Accumulate** se superposent presque parfaitement, ce qui indique une similarité significative dans leurs logiques de calcul.

— **Divide** et **Fold** sont extrêmement efficace et surpasse largement les deux autres méthodes. Cela montre que la stratégie diviser pour régner est bien adaptée à la multiplication de polynômes, où la répartition des tâches permet de réduire la complexité.

Ces résultats confirment que pour les produits sur des ABR de taille fixe, **Divide** est le choix optimal.

3. Temps d'exécution pour les sommes (ABR de tailles exponentielles)

Les stratégies comparées ici ont été appliquées sur une génération de 15 ABR de tailles respectives $1, 1, 2, 4, 8, \ldots, 2^{13}$.

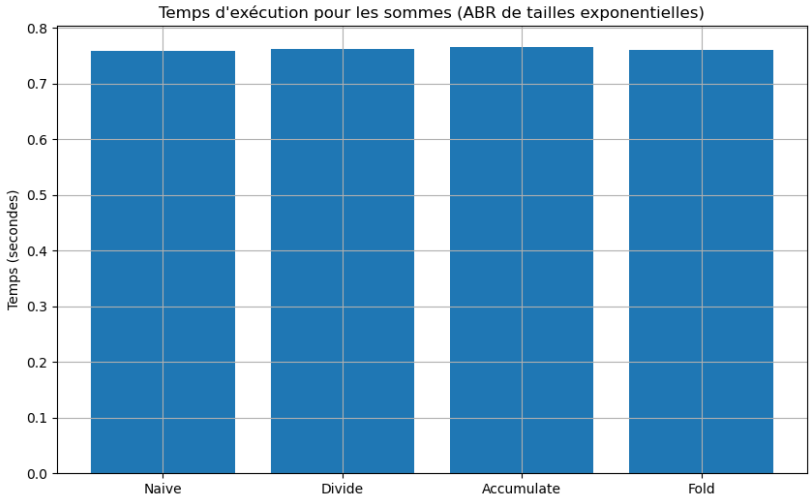

FIGURE 3 – Temps d'exécution pour les sommes avec des ABR de tailles exponentielles.

La figure 3 illustre les performances des différentes méthodes pour des sommes de polynômes sur des ABR de tailles exponentielles (2^n) :

— Les quatre méthodes (**Naive**, **Divide**, **Accumulate Fold**) montrent des temps d'exécution similaires pour ces tailles exponentielles, avec des performances légèrement inférieures pour **Divide**.

— Cela peut s'expliquer par le fait que la décomposition apportée par **Divide** devient coûteuse lorsque les tailles d'arbres sont très grandes.

4. Temps d'exécution pour les produits (ABR de tailles exponentielles)

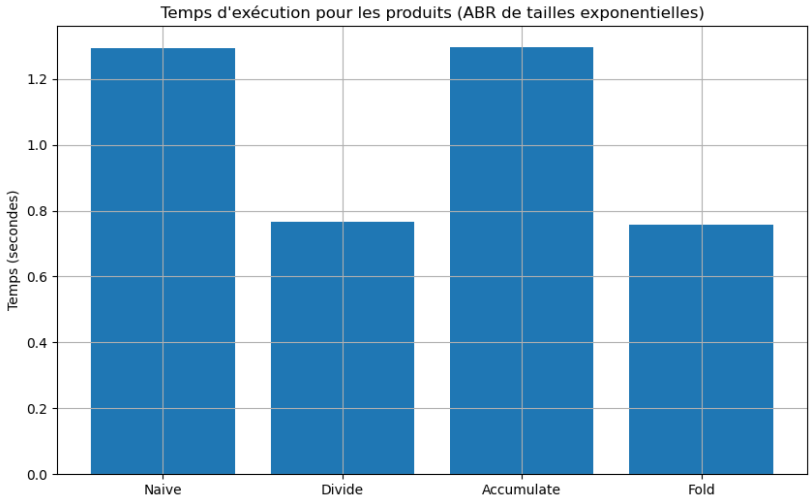

FIGURE 4 – Temps d'exécution pour les produits avec des ABR de tailles exponentielles.

La figure 4 montre les performances pour les produits de polynômes :

— **Naive** et **Accumulate** affichent des performances similaires, ce qui confirme à nouveau une logique d'accumulation commune et lent.

— **Divide** et **Fold** montre des temps d'exécution plus faibles, donc assez rapide.

Pour des tailles exponentielles, bien que **Divide** reste une stratégie performante, son avantage est légèrement réduit à cause des coûts supplémentaires liés à la gestion des grandes structures.

6 Conclusion

Ce projet nous a permis d'explorer et de manipuler des structures de données sous forme linéaire et arborescentes, et ainsi de générer des algorithmes pour la manipulation de ces structures.

Le calcul du produit des polynômes générés est particulièrement coûteux, des stratégies plus fines pourraient nous permettre d'améliorer encore plus les performances de calcul. En effet, notre calcul du produit des polynômes est beaucoup trop long pour être mesuré.

Une des pistes d'évolution possible pour améliorer les performances de nos expérimentations pourrait être de retravailler la structure des polynômes afin d'obtenir une meilleure complexité de nos sommes et produits. On pourrait essayer de travailler non pas avec un chaînage, mais avec des tableaux par exemple.

Ce projet nous a permis de mettre en lumière l'importance des choix algorithmiques et structurels dans l'optimisation des performances et la résolution de problèmes complexes, nous permettant de souligner l'intérêt des structures de données en informatique.