Rapport de projet

Adan Bougherara Vivien Demeulenaere

21 novembre 2021

Table des matières

1 Présentation

L'objectif de ce projet est de représenter et manipuler des polynômes de la forme $P = \sum_{i=0}^{n} c.x^d$ avec $c \in \mathbb{Z}$ et $n \in \mathbb{N}$.

1.1 Polynôme sous forme linéaire

Le code relatif à cette section se trouve dans le fichier polynome.ml

1.1.1 Structure de donnée

Une première approche consiste à représenter un monôme $c.x^d$ avec $(c, d) \in \mathbb{Z} \times \mathbb{N}$. Ainsi, un polyôme sera représenté par une liste de monômes. Nous choisissons alors d'utiliser la structure de données suivante :

```
type monome =  {mutable c : int ; mutable d : int}
and polynome = monome list;;
```

Il a été choisit de rendre `mutable` c et d afin d'en modifier directement les valeurs de la structure, afin de réduire l'espace mémoire utilisé.

1.1.2 Fonction canonique

On souhaite à présent pouvoir éliminer les coefficients nuls d'un polynôme et trier ses monômes par degré croissant afin de le rendre *canonique*. Pour ce faire, on définit une fonction `canonique` dont nous allons analyser la complexité en nombre de comparaisons dans le pire cas.

Soit P un polynome de taille n $\in \mathbb{N}$.

La fonction `canonique` fait appel à une fonction `partition` qui divise P en deux sous-polynômes P_1 et P_2 selon un pivot. Un appel récursif est ensuite effectué sur P_1 et P_2 qui sont concaténés afin de recombiner le polynôme final à la manière d'un tri rapide. De plus, pour chaque monôme de P, on teste si son coefficient c est nul.

Dans le pire cas, n ne décroît pas car aucun coefficient c est nul et tous les degrés d sont différents.

Ainsi `canonique` $\in O(n + n^2)$ donc `canonique` $\in O(n^2)$.
Notons cependant que la complexité en moyenne d'un algorithme de tri rapide s'évalue $O(n.log(n))$

1.1.3 Fonction poly_add

Afin de pouvoir additionner deux polynômes canoniques P_1 et P_2 de tailles respectives n et m, nous avons implémenté une fonction `poly_add`. Analysons la complexité pire cas de cette dernière en nombre d'additions.

Dans le pire des cas : $\forall (c_1, d) \in P_1, \exists (c_2, d) \in P_2$ (ou réciproquement chaque degré de P_2 est dans P_1).

Chaque monôme du polynôme de plus petite taille devra être sommé une fois.
Ainsi `poly_add` $\in O(min(n, m))$

1.1.4 Fonction poly_prod

Afin de pouvoir multiplier deux polynômes canoniques P_1 et P_2 de taille n, nous avons implémenté une fonction `poly_prod`. Analysons la complexité pire cas de cette dernière en nombre de multiplications élémentaires.

La fonction `poly_prod` reprend le principe de l'algorithme de Karatsuba. Contrairement à la multiplication naïve où l'on effectue n^2 multiplications élémentaires, on regroupe certains termes afin d'en effectuer moins. On suit alors le principe suivant :

Soient a et b des nombres positifs écrits en base 2 de $n = 2.k$ chiffres. On remarque alors que :
$a = (a_1.2^k + a_0)$ et $b = (b_1.2^k + b_0)$ avec a_0, a_1, b_0 et b_1 des nombres binaires à k chiffres.
Donc $a.b = a_1.b_1.2^{2k} + (a_1.b_1 + a_0.b_0 - (a_1 - a_0)(b_1 - b_0)).2^k + a_0.b_0$
On effectue ici trois produits élémentaires au lieu de quatre.

Calculons alors la complexité en nombre de produits élémentaires.

Soit $T(n)$, le nombre de multiplications élémentaires. On a le cas d'arrêt $T(1) = 1$.
Sinon on effectue trois appels à `poly_prod` sur des polynômes de taille $\frac{n}{2}$ donc $T(n) = 3.T(\frac{n}{2})$
On peut alors utiliser le *théorème maître* :
$T(n) = \theta(n^{log_2(3)}) \approx \theta(n^{1.585})$

Ainsi `poly_prod` $\in \theta(n^{1.585})$

1.2 Expression arborescente

Le code relatif à cette section se trouve dans le fichier arbre.ml

On introduit dans cette section des expressions arborescentes suivant la grammaire :

$$E = int \mid E_\wedge \mid E_+ \mid E_*$$
$$E_\wedge = x \wedge int^+$$
$$E_+ = (E \setminus E_+) + (E \setminus E_+) + ...$$
$$E_* = (E \setminus E_*) * (E \setminus E_*) * ...$$

1.2.1 Structure de données

Une deuxième représentation possible d'un polynôme est une expression arborescente. On définit alors la structure suivante :

```
type operator =
| Plus
| Prod
| Pow ;;

type tree =
| Empty
| Int of int
| X
| Node of operator * tree array;;
```

1.2.2 Exemple

On peut alors définir l'expression de la Figure 1 de la manière suivante :

```
let t1 =
Node (Plus,
 [|Node (Prod, [|Int 123; Node (Pow, [|X; Int 1|])|]); Int 42;
   Node (Pow, [|X; Int 3|])|])
```

1.2.3 Fonction arb2poly

Pour passer de la représentation arborescente vers un polynôme canonique, on dispose d'une fonction `arb2poly`. Elle parcourt l'arbre en appelant `poly_add` ou `poly_prod` en fonction de la nature du noeud rencontré.

1.3 Synthèse d'expressions arborescentes

Le code relatif à cette section se trouve dans le fichier arbre.ml
Le but de cette partie est de générer aléatoirement des expressions arborescentes

1.3.1 Fonction extraction_alea

Cette fonction prend deux listes L et P en argument. Elle retire aléatoirement un élément de L et l'ajoute en tête de P, puis renvoie les deux listes modifiées.

1.3.2 Fonction gen_permutation

Cette fonction implémente *l'algorithme de Fisher-Yates*. Elle prend en entrée un entier n et renvoie une liste contenant les éléments de 1 à n dans un ordre aléatoire.

1.3.3 ABR

On définit un arbre binaire de recherche de la manière suivante :

```
type bst =
| Vide
| Noeud of int * bst * bst;;
```

Il est maintenant possible d'implémenter une fonction `abr` qui prend en entier une liste d'entiers et renvoie l'abr qui lui est associé.

1.3.4 Fonction etiquetage

Pour transformer notre arbre binaire de recherche en une expression arborescente ayant la même forme, on utilise la fonction `etiquetage`. On note que cette fonction renvoie une expression suivant des règles peu contraintes. Cette fonction peut par exemple retourner un arbre contenant un noeud Plus dont le père est également un Plus.

1.3.5 Fonction gen_arb

La fonction `gen_arb` se charge de reconstruire une structure issu de la fonction `etiquetage` en une expression équivalente qui respecte la grammaire définie dans la section 1.2.

2 Expérimentations

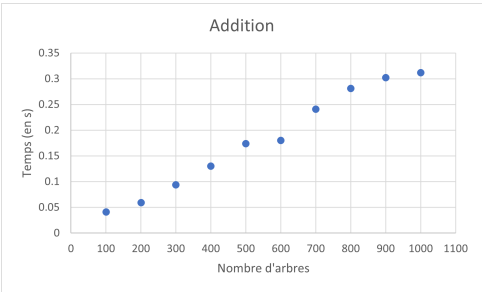
Le code relatif à cette section se trouve dans le fichier experimentations.ml

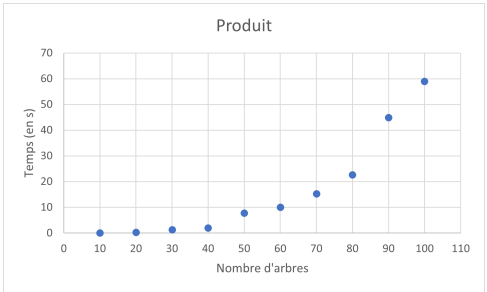
Dans un premier temps, on génère des listes d'ABR de taille 20 avec la fonction `gen_array_arb`. Ces listes varient entre 100 et 1000 arbres avec un pas de 100. Le but de cette partie est de comparer différentes stratégies pour sommer ou multiplier n arbres.
Dans un second temps, nous calculerons les temps moyens des différentes stratégies avec un échantillon de 15 arbres ayant pour tailles respectives 2^0, 2^0, 2^1, ..., 2^{13}.

2.1 Stratégie 1 : naïve

Cette stratégie a été implémentée dans les fonctions `addition_strat1` et `produit_strat1`.

Elle consiste à transformer chaque arbre en un polynôme et à l'additionner (respectivement multiplier) avec la somme (respectivement le produit) des arbres qui le précèdent. Cette stratégie n'est pas idéale en particulier dans le cas du produit. En effet, `poly_prod` égalise la taille des polynômes qu'elle reçoit. Avec cette stratégie, on multiplie donc à chaque itération un polynôme de plus en plus grand avec un polynôme dont l'expression arborescente est de taille 20. De nombreux produits avec des coefficients nuls sont effectués. Comme les graphiques ci-dessous le montrent, la courbe du produit semble exponentielle et celle de la somme linéaire. Pour cette raison, lancer cette stratégie avec n variant de 100 à 1000 demande trop de temps de calcul.

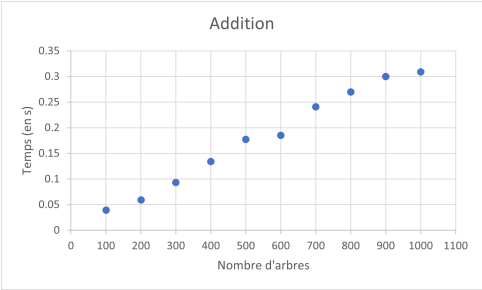


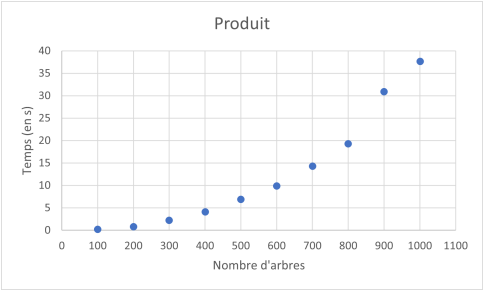


2.2 Stratégie 2 : Diviser pour régner

Cette stratégie a été implémentée dans les fonctions `addition_strat2` et `produit_strat2`.

Elle consiste à diviser une somme (respectivement produit) de n arbres transformés en polynômes en plusieurs sous-sommes (respectivement sous-produits) de deux éléments. Les sous-sommes (respectivement sous-produits) effectué(e)s deviennent à leur tour des termes de sous-sommes (respectivement sous-produits) de deux éléments jusqu'à remonter au résultat final. On remarque sur les graphiques suivants que l'addition se fait en temps linéaire et le produit semble s'approcher du temps polynomial attendu avec l'algorithme de Karatsuba.

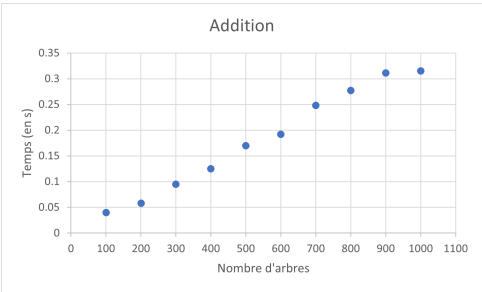


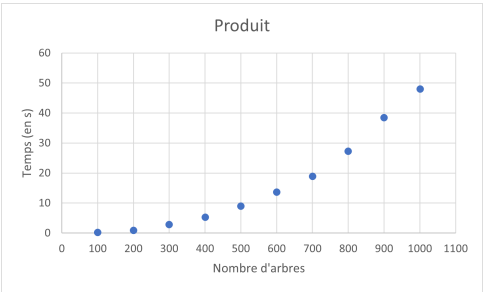


2.3 Stratégie 3 : Diviser pour régner avec tri

Cette stratégie a été implémentée dans les fonctions `addition_strat3` et `produit_strat3`.

On utilise une stratégie similaire à la précédente. Cependant, les sous-sommes (respectivement les sous-produits) ne sont pas les mêmes. En effet, à chaque itérations, les polynômes sont triés par taille grâce à un tri par insertion. On s'assure alors de sommer (respectivement multiplier) les polynômes de tailles les plus proches possibles. On remarque sur les graphiques suivants que l'addition se fait en temps linéaire et le produit semble s'approcher du temps polynomial attendu avec l'algorithme de Karatsuba.

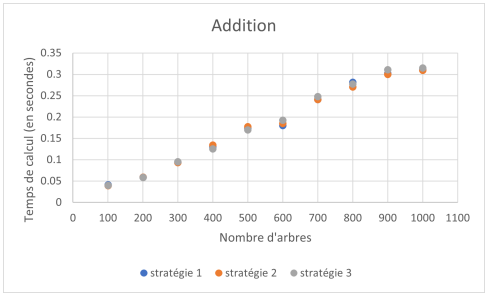




2.4 Comparaison des différentes stratégies

2.4.1 Addition

Le graphique ci-dessous reprend les courbes de chaque stratégie.



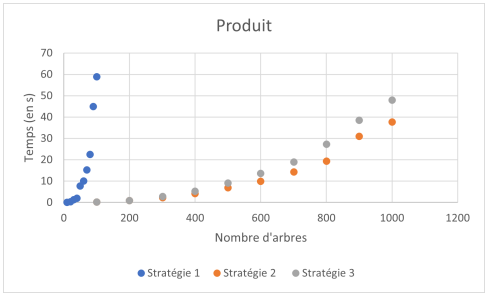
Comme nous pouvons le constater ici, les résultats sont très proches et les courbes se superposent. Cela nous indique qu'il n'y a pas de différence notable entre les stratégies dans le cas de l'addition. Intéressons nous maintenant aux 15 arbres ayant pour tailles respectives 2^0, 2^0, 2^1, ..., 2^{13}.

	Stratégie1	**Stratégie 2**	**Stratégie 3**
	0.856	0.859	0.873
	1.048	1.04	1.025
	1.078	1.065	1.074
	1.157	1.112	1.124
	0.972	0.977	0.955
	0.935	0.946	0.909
	0.863	0.859	0.878
	0.993	0.953	0.971
	0.999	0.976	0.985
	1.081	1.051	1.089
moyenne	**0.9982**	**0.9838**	**0.9883**

Le tableau ci-dessus représente 10 exécutions d'un programme additionnant les arbres en fonction des 3 stratégies établies. Comme on peut le noter aucune stratégie ne se détache réellement.

2.4.2 Produit

Le graphique ci-dessous reprend les courbes de chaque stratégie.



On peut voir qu'il est impossible de mettre en pratique la stratégie naïve sur un grand nombre d'arbres en raison de son caractère exponentiel. Les stratégies 2 et 3 ont un comportement proche avec un léger avantage pour la deuxième. Cela s'explique par le fait que le tri des polynômes à chaque itération est coûteux en raison du grand nombre d'arbres présents.
Intéressons nous maintenant aux 15 arbres ayant pour tailles respectives 2^0, 2^0, 2^1, ..., 2^{13}.

	Temps de calcul Stratégie 1 (en s)	**Temps de calcul Stratégie 2 (en s)**	**Temps de calcul Stratégie 3 (en s)**
	2.828	4.279	1.781
	1.731	2.803	1.348
	2.619	3.316	1.492
	1.886	2.334	1.217
	1.907	2.944	1.452
	2.453	2.708	1.422
	2.218	3.168	1.616
	2.669	3.276	1.444
	2.253	3.496	1.623
	3.229	4.775	2.111
moyenne	**2.3793**	**3.3099**	**1.5506**

Le tableau ci-dessus représente 10 exécutions d'un programme multipliant les arbres en fonction des 3 stratégies établies. On remarque que dans cette configuration, la stratégie 3 s'avère bien meilleure que les deux autres. Ceci s'explique par le fait que le tri est rapide car il n'y a que 15 polynômes au départ et que l'on multiplie des polynômes de tailles proches.