Projet IPF: SUBSET-SUM-OPT

Romain PEREIRA

03/04/2018

Sommaire

1 Préambule

Ce projet est réalisé dans le cadre de mes études à l'ENSIIE. Rappel de l'énoncé:

> SUBSET-SUM-OPT - Etant donnée un ensemble fini E d'entiers strictements positifs et un entier cible s, trouver l'entier $s' \leq s$ le plus grand possible, tel qu'il existe un sous-ensemble $E' \subseteq E$ vérifiant $\sum_{e \in E'} e = s'$.

Ce rapport présente (en pseudo-code), les algorithmes implémentés.
Le code OCaml est disponible dans le rendu.

1.1 Spécificités techniques

Un Makefile est disponible pour compiler le projet et les tests:

- *make* : compile les fichiers sources avec les tests vers un executable **subset-sum-opt.out**
- *clean* : supprimes les fichiers compilés temporaires (.mlo et .cmo)
- *fclean* : supprimes tous les fichiers compilés (executable et temporaires)

Les testes sont unitaires et effectue dans l'ordre:

- Les fonctions du module *Listes*
- Les fonctions du module *Ensembles*
- Les fonctions du module *Approche_naïve*
- Les fonctions du module *Approche_plus_directe*
- Les fonctions du module *Approche_avec_nettoyage*
- Les fonctions du module *Approche_diviser_pour_regner*
- Les fonctions du module *Approche_diviser_pour_regner*
- Comparaisons des approches

1.2 Notations

J'utiliserai les notations suivantes:

- Soit E est un ensemble, $Card(E)$ est le cardinal de E.
- Soit E est un ensemble, $P(E)$ est l'ensemble formé des parties de E

1.3 Rappels/Complexités

Soit E un ensemble tel que $Card(E) = n$, alors

$$Card(P(E)) = 2^n$$

$$\forall E' \in P(E), 0 <= Card(E') <= n$$

On considèrera les complexités pour les opérations suivantes:

- Soit (x, y) des scalaires, je considère que les 4 opérations élémentaires ($x+y$, $x-y$, $x*y$, x/y) sont en $O(1)$
- Soit 2 ensembles X, Y, $X \cup Y$ est une opération en $O(Card(X) * Card(Y))$
- Soit $E \subset \mathbb{N}$, $\max E$ est une opération en $O(Card(E))$

1.4 Bonus

L'implémentation a également été légèrement plus loin que les algorithmes présentés dans ce document: elle permet de savoir sur quelle partie $E' \subset E$ la somme $\sum\limits_{e \in E'} e = s$ a été atteinte.

La 1ère implémentation (qui est plus simple, mais sans l'ensemble sur lequel les sommes sont obtenus) est disponible dans le dossier 'bkp' du rendu.

2 Approche naïve

Cette approche consiste à determiner tous les sous-ensembles E' de E, d'effectuer la somme sur tous les E', et de renvoyer la somme la plus proche de s. Cette approche par 'force brute' est lourde.

2.1 Question 1 : somme sur un ensemble

Algorithm 1: Renvoie la somme des éléments de E

function SOMME($E' \subset \mathbb{N}$)
  if $E' = \emptyset$ **then**
    return 0
  end if
  Soit $x \in E'$
  return x + $\texttt{Somme}(E' \backslash \{x\})$
end function

Complexité en $\boxed{O(n)}$, où $n = Card(E')$

2.2 Question 2 : génération des parties d'un ensemble

Algorithm 2: Renvoie l'ensemble des parties de E

function SOUS-ENSEMBLES($E \subset \mathbb{N}$)
  if $E = \emptyset$ **then**
    return $\{\emptyset\}$
  end if
  Soit $x \in E$
  Soit $P \leftarrow \texttt{Sous} - \texttt{ensembles}(E \backslash \{x\})$
  return $P \cup \{E' \cup x \mid E' \in P\}$
end function

Complexité en $\boxed{O(2^n n)}$, où $n = Card(E)$

2.3 Question 3 : résolution de SUBSET_SUM_OPT par force brute

Algorithm 3: Renvoie la réponse au problème SUBSET_SUM_OPT sur (E, s)

function SUBSET_SUM($E \subset \mathbb{N}, s \in \mathbb{N}$)
  Soit $P \leftarrow \texttt{Sous} - \texttt{ensembles}(E)$
  return $\max_{E' \in P} \{s' \mid s' = \texttt{Somme}(E') \quad et \quad s' \leq s\}$
end function

Complexité en $\boxed{O(2^n n)}$, où $n = Card(E)$

3 Approche plus directe

Dans cette approche, plutot que de calculer l'ensemble des parties de E, on se propose de calculer l'ensemble des sommes atteignables en sommet sur les parties de E.

3.1 Question 4 : sommes atteignables

Algorithm 4: Renvoie l'ensemble des entiers s tels qu'il existe $E' \subseteq E$ vérifiant $\sum_{e \in E'} e = s$

function GET_ALL_SUMS($E \subset \mathbb{N}$)
  if $E = \emptyset$ **then**
    return $\{0\}$
  end if
  Soit $x \in E$
  $S \leftarrow$ `get_all_sums`$(E \backslash \{x\})$
  return $S \cup \{x + s \mid s \in S\}$
end function

Si l'on suppose:

- $n = Card(E)$
- $m(n) = Card\{sommes \quad atteignables\} = \{s \mid \exists E' \subseteq E \mid \sum_{e \in E'} e = s\} \leq 2^n$

Alors la complexité de cet algorithme est en $\boxed{O(m(n) * n)}$

- n : nombre de récursion
- $m(n)$: l'union

Remarque: Cette algorithme fourni les sommes atteignables dans un ordre croissant. $(\bullet < \bullet)$

3.2 Question 5 : résolution de SUBSET_SUM_OPT

Une fois l'ensemble des sommes atteignables calculés, la résolution du problème devient trivial. La complexité de cet algorithme de résolution est donc en $\boxed{O(m(n) * n)}$

Algorithm 5: Renvoie la réponse au problème SUBSET_SUM_OPT sur (E, s)

function SUBSET_SUM($E \subset \mathbb{N}, s \in \mathbb{N}$)
  Soit $S \leftarrow$ `get_all_sums`(E)
  return $\max\{s' \in S \mid 0 \leq s' \leq s\}$
end function

4 Approche avec nettoyage

On peut réduire la complexité de l'approche précèdente en réduisant ce que l'on a noté $m(n)$ (le nombre de sommes atteignables. Dans cette approche, on se propose d'ajouter un 'filtre' sur l'algorithme qui génère les sommes.

4.1 Question 6 ; fonction clean_up

Le filtre (l'algorithme `clean_up` est donnée dans l'énoncé. Cette fonction prends en paramètre:

- $E \subset \mathbb{N}$: l'ensemble a filtré
- $s \in \mathbb{N}$: un entier positif
- $\delta \in \mathbb{R}_+^*$: un réel positif (généralement $\ll 1$)

Cette fonction renvoie un nouvel ensemble $E' \subset E$, tel que:

- $\forall x \in E', x \leq s$
- Si l'on considère la suite croissante $(u_n)_{n\in\mathbb{N}}$ des éléments de E', $\forall n \in \mathbb{N}, u_{n+1} \geq (1+\delta)u_n$.

Autrement dit, tous les entiers strictement supérieurs à s sont supprimés, et si l'on considère 2 entiers consécutifs de l'ensemble trié x et y, tel que $x < y$, ils sont 'proches d'un rapport d'au moins $(1+\delta)$, y est supprimé. Ce filtre supprime donc les éléments de l'ensemble qui sont 'proches' l'un de l'autre (au regard de δ). En supprimant des valeurs proches, on espère pouvoir former les mêmes sommes qu'avec toutes les valeurs, mais en effectuant ainsi moins de sommations.

Par exemple, pour l'entrée $[\![0, 100]\!]$, $s = 90$ et $\delta = 0.01$, on obtient la sortie

$$[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 12; 14; 16; 18; 20; 23; 26; 29; 32; 36; 40; 45; 50; 56; 62; 69; 76; 84]$$

On remarque que les valeurs les plus 'petites' sont convervés (≤ 10), et que plus les valeurs deviennent 'grandes', moins elles passent le filtre.

4.2 Question 7 : résolution de SUBSET_SUM_OPT

L'algorithme de résolution est également fourni dans l'énoncé. Il est le même que celui de 'l'approche plus directe' 3, sauf que lors du calcul des sommes atteignables filtrés par la fonction `clean_up`.

Algorithm 6: Renvoie l'ensemble des entiers s tels qu'il existe $E' \subseteq E$ vérifiant $\sum_{e\in E'} e = s$, passant les tests du filtre

---

function GET_ALL_SUMS_2($E \subset \mathbb{N}$, $\delta \in \mathbb{R}_+^*$)
  if $E = \emptyset$ **then**
    return $\{0\}$
  end if
  Soit $x \in E$
  $S \leftarrow$ `get_all_sums_2`$(E\backslash\{x\})$
  return `clean_up`$(S \cup \{x + s \mid s \in S\})$
end function

La complexité de cet algorithme de résolution est donc en $\boxed{O(m'(n) * n)}$, avec

$$m'(n) = Card\{sommes \quad atteignables \quad filtres\} \leq m(n) \leq 2^n$$

. **Attention** cependant, pour δ trop grand, on perd l'optimalité du résultat.

Algorithm 7: Renvoie la réponse au problème SUBSET_SUM_OPT sur (E, s)

function SUBSET_SUM($E \subset \mathbb{N}, s \in \mathbb{N}$)
  Soit $S \leftarrow$ `get_all_sums_2`(E, δ)
  return $\max\{s' \in S \mid 0 \leq s' \leq s\}$
end function

5 Approche de type *Diviser pour régner*

5.1 Question 8 : is_feasible

On nous demande ici de créer une fonction qui sur la donnée d'un entier s, d'une liste l_1 d'entier croissant, d'une liste l_2 d'entier décroissant, renvoie $true$ si s s'écrit comme la somme d'un élément de l_1 et d'un élément de l_2, et $false$ sinon.
La fonction implémenté suit l'algorithme suivant, qui se sert au maximum du fait que les listes soit triés. Les 3 cas de retours possibles sont:

- `TOO_SMALL` : Soit $y \in l_2$, $\forall x \in l1, x + y < s \Rightarrow \forall y' < y, x + y' < x + y < s$. En français, pour y fixer, la somme des $x + y$ avec x dans l_1 est toujours inférieur à s. Pour un y plus petit, on ne risque donc pas d'atteindre une valeur plus proche de s.
- `REACHED` : on a réussi à atteindre s.
- `TOO_BIG` : On a parcouru toutes les valeurs de l_2, pour toutes les valeurs de l_1, sans avoir pu obtenir de couple $(x, y) \in l_1 \times l_2$ tel que $x + y <= s$.

Algorithm 8: Renvoie **true** si $\exists (x, y) \in l_1 \times l_2 \mid x + y = s$, **faux** sinon

function IS_FEASIBLE($s \in \mathbb{N}$, $l_1 \subset \mathbb{N}$, $l_2 \subset \mathbb{N}$)
  for $y \in l_2$ (pris du plus grand au plus petit) **do**
    for $x \in l_1$ (pris du plus petit au plus grand) **do**
      if $x + y = s$ **then**
        Renvoyer true (cas `REACHED`)
      else if $x + y > s$ **then**
        Passer au y **suivant** (cas `TOO_BIG`)
      else(si $x + y < s$)
        Passer au x **suivant** (car $x + y$ est déjà plus petit que 's')
      end if
    end for
    Renvoyer faux (cas `TOO_SMALL`)
  end for
  Renvoyer faux (on a eu que des cas `TOO_BIG`)
end function

Les complexités sont:

- Dans le pire des cas (où l'on a que des cas TOO_BIG) : $O(Card(l_1)Card(l_2)$
- Dans le meilleur des cas (cas $REACHED$ avec le 1er y et le 1er x...) : $O(1)$ (peu probable)
- Le cas moyen n'est pas évident à évaluer, étant donné que l_1 et l_2 dépendent de nombreux paramètres...

5.2 Question 9 : best_feasible

Au regard de la Question 8 (5.1), on nous demande maintenant de créer une fonction qui sur la donnée d'un entier s, d'une liste l_1 d'entier croissant, d'une liste l_2 d'entier décroissant, renvoie le plus grand entier $s' \leq s$ somme d'un élément de l_1 et d'un élément de l_2

N.B: non explicitement dis dans le sujet, mais si un tel entier s' n'existe pas, -1 est renvoyé.

Algorithm 9: Renvoie $\max\limits_{(x,y)\in l_1\times l_2} \{x+y \mid x+y < s\}$

function BEST_FEASIBLE($s \in \mathbb{N}$, $l_1 \subset \mathbb{N}$, $l_2 \subset \mathbb{N}$)
    $s' \leftarrow -1$
    for $y \in l_2$ (pris du plus grand au plus petit) **do**
        for $x \in l_1$ (pris du plus petit au plus grand) **do**
            if $x + y = s$ **then**
                Renvoyer s (cas `REACHED`)
            else if $x + y > s$ **then**
                Passer au y **suivant** (cas `TOO_BIG`)
            else(si $x + y < s$)
                if $x + y > s'$ **then**
                    $s' \leftarrow x + y$
                end if
                Passer au x **suivant**
            end if
        end for
        Renvoyer s' (cas `TOO_SMALL`).
    end for
    Renvoyer s' (on a eu que des cas `TOO_BIG`)
end function

5.3 Question 10 : résolution de SUBSET_SUM_OPT

En se servant des fonctions 3.1, 5.2, en en ajoutant 2 fonctions subsidiaires (voir *Listes.ml*)

- `split` : sépares une liste en 2 sous liste de longueur égal (à 1 prêt)
- `sort` : tri les éléments dans l'ordre croissant
- `invsort` : tri les éléments dans l'ordre décroissant

on peut élaborer l'algorithme de résolution suivant, qui se base sur l'idée de 'diviser pour régner':

Algorithm 10: Renvoie la réponse au problème SUBSET_SUM_OPT sur (E, s)

function SUBSET_SUM($E \subset \mathbb{N}, s \in \mathbb{N}$)
    $(l_1, l_2) \leftarrow$ `split`(l)
    $(s_1, s_2) \leftarrow$ (`get_all_sums`$(l1)$, `get_all_sums`$(l2)$)
    return `best_feasible`$(s,$ `sort`(s_1), `invsort`$(s_2))$
end function

6 Comparaison des approches et améliorations

6.1 Question 11

(voir 'tests.ml')

6.2 Question 12

(voir 'tests.ml')

6.3 Question 13 : Bonus

Comme dit précèdemment (1.4), ce bonus a été effectué: les fonctions de résolution renvoie également le sous-ensemble sur lequel la somme a été obtenu.

6.4 Illustration

Les graphiques suivants ont été généré sur les tests suivant:

- 1) Fixer $n = Card(E)$
- 2) Générer un ensemble $E \subset \mathbb{N}$ tel que $Card(E) = n$ et $\forall x \in E, 1 \leq x \leq 2n$
- 3) Résoudre le probleme: $SUBSET\_SUM(s = \frac{n * \frac{n}{2}}{2}, E)$ avec les 4 approches.

Pour $1 \leq n \leq 17$, toutes les résolutions fonctionnent, et on obtient ces allures de courbes:

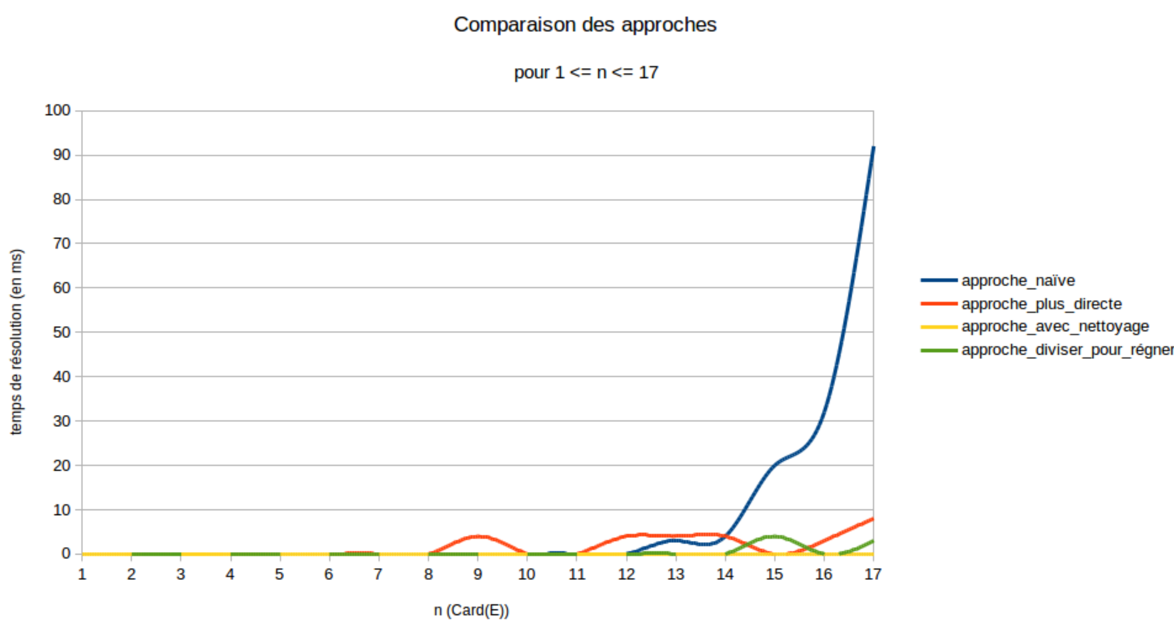
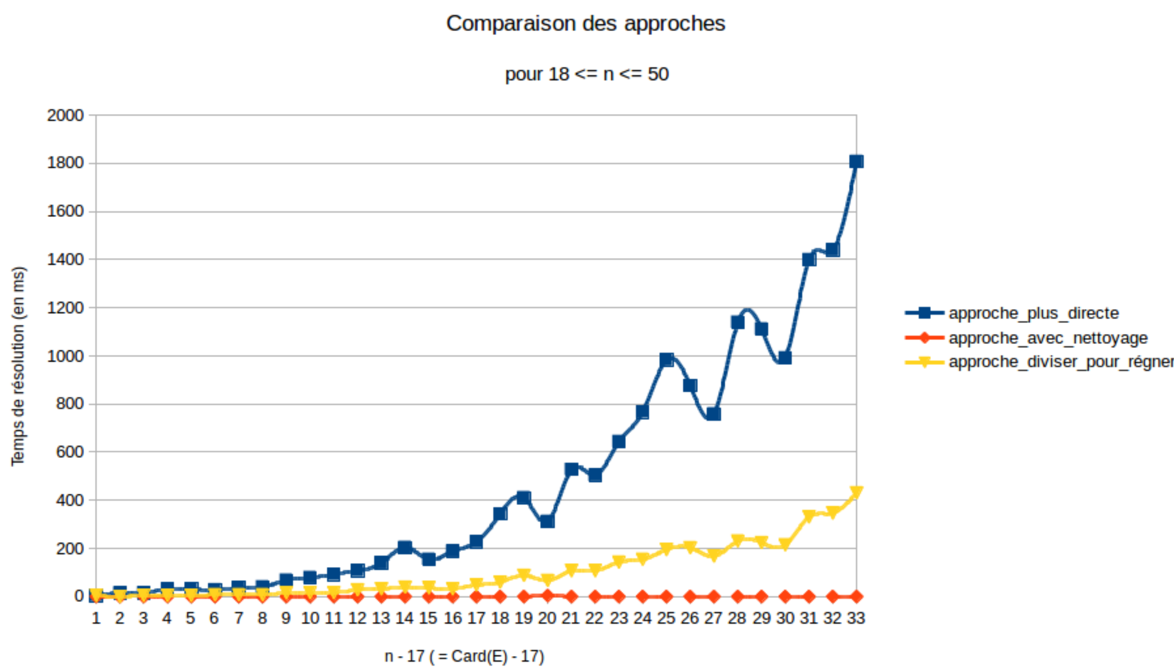


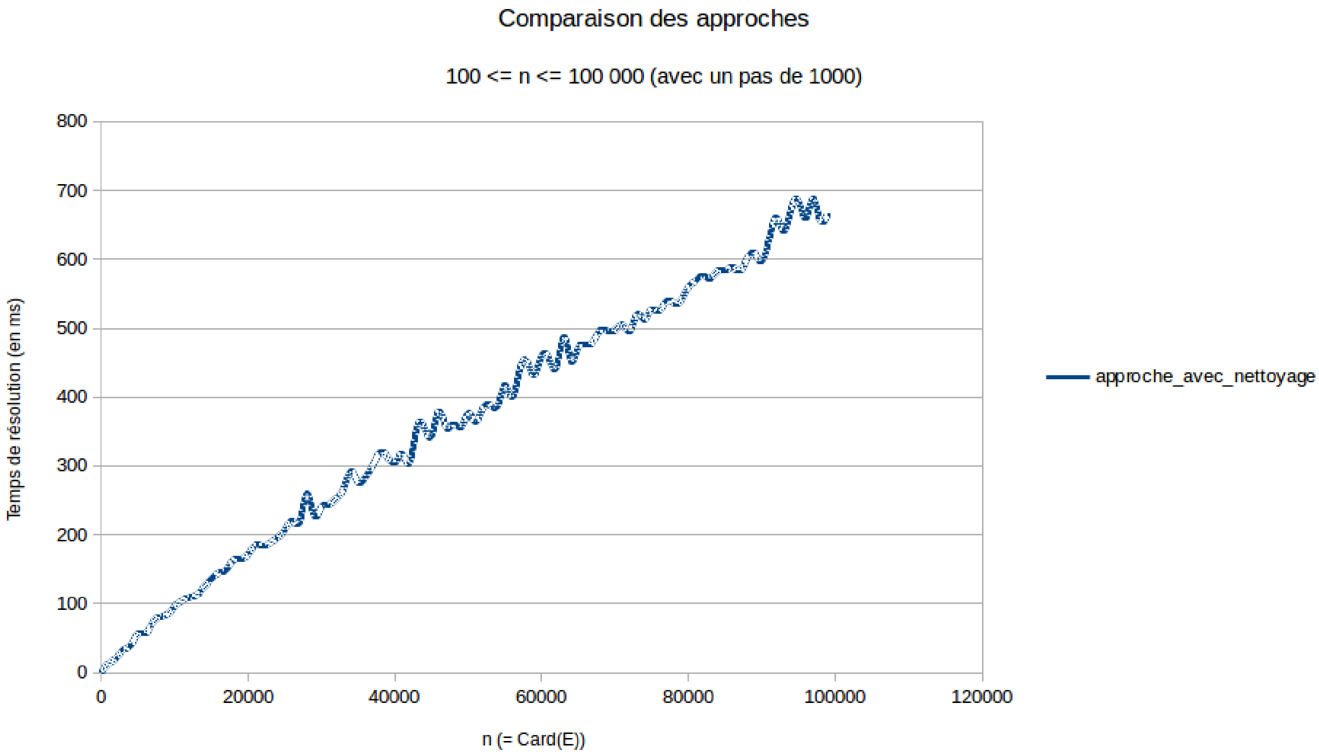
Figure 1: *La complexité de l'approche naïve suit effectivement une allure exponentielle*

Pour $n > 18$, l'approche naïve (4) provoque une erreur de dépassement de pile, je n'ai pas pu la tester pour des n plus grands.



On remarque que l'approche plus directe (3) a une allure exponentielle (en $O(2^n)$), et que l'approche 'diviser pour régner' (5) aussi, mais avec une allure plus 'écrasé'.

Pour $n \geq 50$, l'approche diviser pour régner prends plus de 2 secondes... Voici les performances de l'approche avec nettoyage pour $n \gg 1$:



Pour $n \geq 140 \quad 000$, j'obtiens un dépassement de pile sur ma machine.
Les résultats ont été obtenus pour $\delta = 10^{-7}$, et si l'on note r le résultat obtenu, il s'éloigne d'au plus 1% de s

$$100\frac{r}{s} \leq 0.01$$

N.B: Les données brutes (format .csv) sont disponibles dans le rendu.

7 Conclusion

L'objectif de ce projet était de résoudre le problème SUBSET_SUM_OPT 1, problème NP-difficile, à l'aide de différentes approches, puis de comparer les résultats obtenus.

L'implémentation des 4 méthodes demandées a été effectué avec succès.

Pour les méthodes naïve, plus directe, et de type diviser pour régner, le résultat obtenu est optimal. Bien que ces algorithmes soient plus longs que l'approche avec nettoyage, on s'assure l'optimalité du résultat et donc la réponse au problème.
Cependant, pour des valeurs de $n = Card(E)$ trop grandes (> 50), nos implémentations sont lentes ($\geq 4sec.$), ou pire, elles atteignent les limites de la machine et s'arrête suite à des dépassements de piles.

L'approche avec nettoyage quant à elle est plus rapide (voir Illustrations 6.4), mais ne permet de trouver une solution optimal.

Pour aller plus loin dans la résolution du problème, il est donc maintenant question de savoir s'il existe des structures de données permettant d'optimiser nos algorithmes (l'utilisation d'une pile explicit en mémoire RAM, au lieu de la pile-machine pour éviter les dépassements de pile, par exemple), ou bien encore d'autres algorithmes plus performant donnant une solution exacte (nottement en allant plus loin dans la logique de 'diviser pour régner', ou en trouvant un filtre type 'clean_up' qui garanti tout de même l'optimalité)

Quelques recherches ont été effectué sur ces pistes, mais n'ont pas pu être mis en oeuvre par manque de temps.

8 Références

[1] 'Module List' - INRIA
https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html

[2] 'Ensemble des parties d'un ensemble' - Wikipédia
https://fr.wikipedia.org/wiki/Ensemble_des_parties_d'un_ensemble