Bachelor Universitaire Technologique

Département Informatique

B.U.T. INFO

Rendu SAE 1.02
Grundy 2

Etude de l'efficacité de différents algorithmes pour résoudre le même problème

Élaboré par :
- BOUILLIS Awen
- LEBORGNE Néo

Classe : 1ère année
Groupe : B1

Ce projet a été réalisé dans le cadre d'une mise en Situation d'Apprentissage et d'Évaluation (SAE). Il a pour objectif principal de placer les étudiants dans une situation similaire à celle qu'ils pourraient rencontrer dans leur future profession, c'est à dire maximiser l'efficacité des algorithmes. Les étudiants doivent donc travailler sur cinq algorithmes différents afin de comprendre l'importance de coder efficacement.

Mise en situation :

Dans cette SAE1.02, nous allons étudier l'efficacité de la recherche du meilleur coup à jouer dans une partie du jeu de Grundy en utilisant une méthode récursive *"estPerdante()"* qui implémente une théorie garantissant de choisir toujours la solution gagnante si elle existe. Si elle n'existe pas, la machine choisira une décomposition au hasard.

Un bon nombre de méthodes nous ont été données dans un document.
La documentation que nous écrirons devra être impérativement écrite en anglais.
Toute action doit être commentée.

Règle du jeu :

Le jeu se joue seul ou à 2.
Il commence sur un plateau de n (vous décidez) allumettes alignées sur la même rangée.
Le premier joueur peut alors décider de séparer un certain nombre d'allumettes de cette première rangée.
/!\ Il faut au minimum déplacer 1 allumettes et il doit au minimum en rester 1 sur la rangée initiale.
/!\ De plus, le joueur ne peut pas séparer un tas en 2 tas de tailles exactement similaires.
Les allumettes séparées forment une nouvelle rangée qui pourra être séparée par la suite.
Vous l'aurez donc compris, un tas de 2 allumettes ne pourra plus être séparé car il enfreindrait la règle N°2.
Lorsqu'il ne reste que des rangées de 1 ou 2 allumettes, le jeu est terminé.
Celui qui ne peut plus jouer à perdu.

Code source :

Nom du fichier source qui nous a été donné Grundy2RecBrute.java

Contenu :
jouerGagnant() : Joue le coup gagnant s'il existe
estPerdante() : Méthode récursive qui indique si la configuration (du jeu actuel ou jeu d'essai) est perdante
enlever() : Divise en deux tas les alumettes d'une ligne de jeu
estPossible() : Teste s'il est possible de séparer un des tas

premier() : Crée une toute première configuration d'essai à partir du jeu
suivant(): Génère la configuration d'essai suivante (c'est-à-dire UNE décomposition possible)

Implémentation de la version 0 :

Nom du fichier source : Grundy2RecBruteEff.java

Le but de cette version est de créer une méthode qui permet à l'utilisateur de jouer une partie depuis son terminal. Nous l'avons nommée *"playAgainstAI()".* Nous avons par ailleurs codé la méthode "display()" qui affiche le plateau de jeu.
Cette version est aussi la première version dont nous avons dû tester l'efficacité. Nous avons donc créé la méthode *"testJouerGagnantEff()*". Les tests d'efficacité commencent toujours à 3 allumettes et s'incrémentent au fur et à mesure des résolutions.
Cette méthode doit être stoppée à la main, on l'arrête quand le temps de résolution devient trop long.

Exemple de partie jouée contre l'ordinateur :

```
***  Game is starting ...  ***
Username : Test
Enter the number of sticks (must be superior to 3) : 8
Who is starting ? (0:Computer | 1:Test) : 0

0 : | | | | | | | |

Computer turn's

0 : | | | | | | |
1 : |

Test's turn

Enter line number : 0
Enter stick number you want to split : 3
0 : | | | |
1 : |
2 : | | |

Computer turn's

0 : | | |
1 : |
2 : | | |
3 : |
```

Test's turn

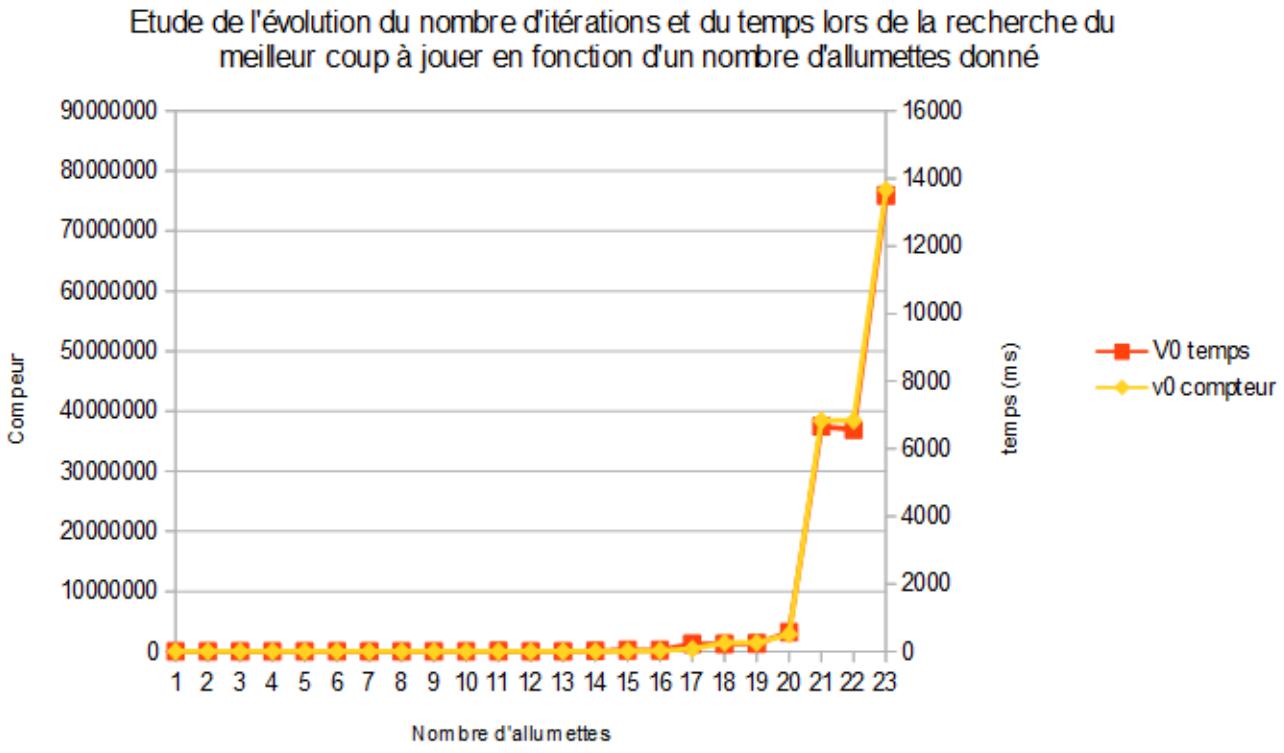
Enter line number : 0
Enter stick number you want to split : 1
0 : | |
1 : |
2 : | | |
3 : |
4 : |

Computer turn's

0 : | |
1 : |
2 : | |
3 : |
4 : |
5 : |

Game closed !
** YOU LOSE ! COMPUTER WON **

Etude de l'efficacité :
Cette version 0 n'implémente aucune amélioration concernant l'efficacité.



On peut voir que cette première version permet de trouver facilement le meilleur coup à jouer pour un nombre d'allumettes inférieur ou égal à 20. Au-delà, le nombre d'appels de la méthode *"estPerdante()*" tend vers les 80 000 000 très rapidement et met de plus en plus de temps.

Implémentation de la version 1 :

nom du fichier source : Grundy2RecPerdantes.java

Cette version 1 est donc la première version améliorée de l'IA. Le principe de cette version est simple. Nous partons du principe qu'il ne sert à rien de recalculer la nature d'une disposition si nous avons déjà établi qu'elle était perdante. Pour cette version on se base aussi sur un deuxième principe, l'ordre des tas n'a pas d'importance et les tas de 1 et 2 allumettes ne pourront plus être joués, il apparaît donc intéressant de les supprimer.

Pour stocker la liste des versions perdantes calculées, nous avons mis en place un ArrayList qui contient les dispositions, elles-mêmes sous forme d'ArrayList des occurrences (cf ci-après), et l'avons nommé "perdantes" : ArrayList<ArrayList<Integer>> perdantes;

Il est ensuite apparu le problème de la comparaison des dispositions entre elles, sachant que [1,2,3,4,5] est identique à [2,5,3,1,4]. Nous avons eu l'idée de représenter ces tableaux sous forme de tables d'occurrences.

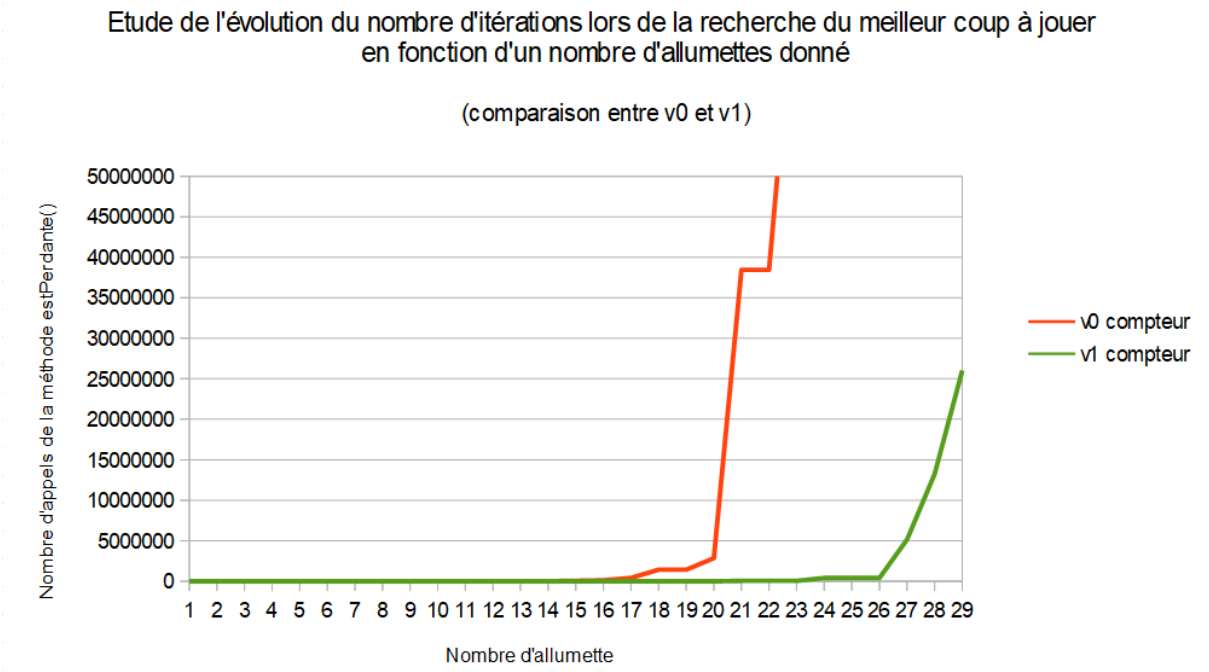
Exemple :
[1,2,3,4,5] devient [0,1,1,1,1,1]
[3,3,3,3] devient [0,0,0,4]

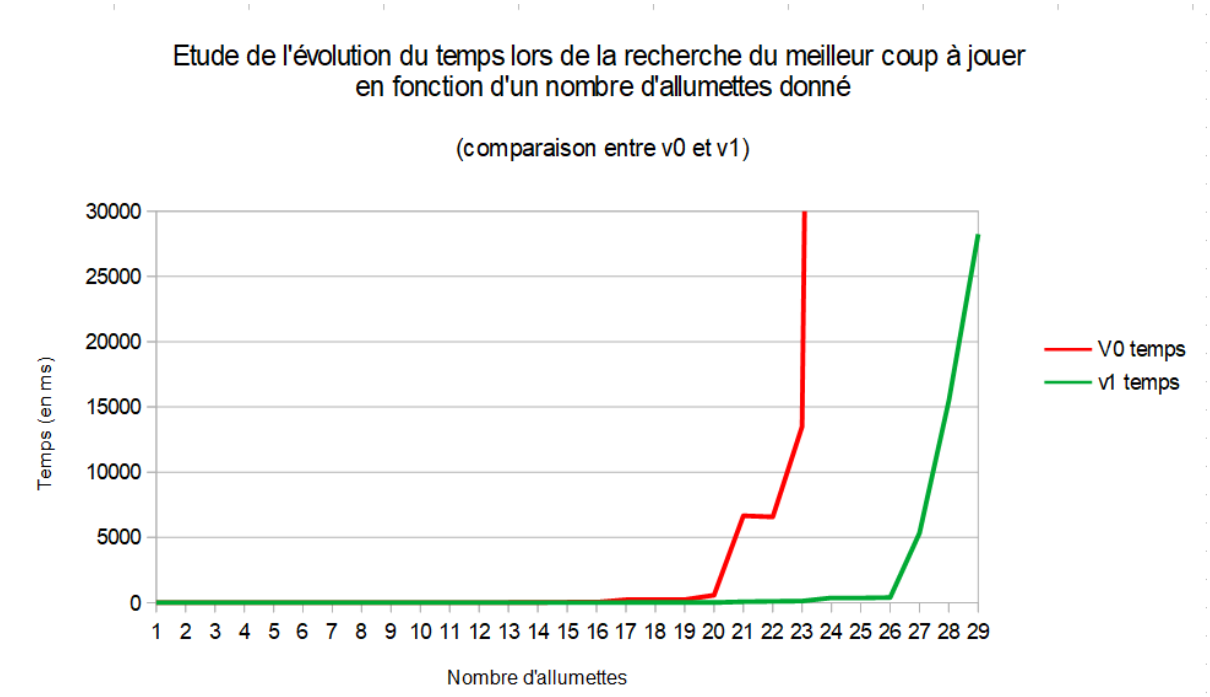
Ce changement de forme nous permet de comparer les dispositions sans se soucier du placement des tas. Nous avons donc codée la méthode "occurrenceTable()" qui renvoie l'ArrayList donné sous forme d'un ArrayList d'occurrences.

Puis, grâce à cette nouvelle méthode, nous avons pu créer la méthode "isFoundInLosingArrayList()" qui renvoie la forme occurrence si la disposition donnée (sous forme normale) à déjà été calculée et null sinon.

Etude de l'efficacité
Cette version implémente la sauvegarde des dispositions perdantes.

Sur le graphique ci-dessous, on peut voir que le nombre d'itérations de la version 1 est inférieur à celui de la version 0. Cependant, il croit tout de même relativement fort après la 27ème allumette.





En ce qui concerne le temps d'exécution, on remarque qu'on peut trouver le meilleur coup à jouer pour 6 allumettes de plus que la version 0. Cette version est donc légèrement meilleure que la précédente.

Implémentation de la version 2 :

nom du fichier source : Grundy2RecPerdEtGagn.java

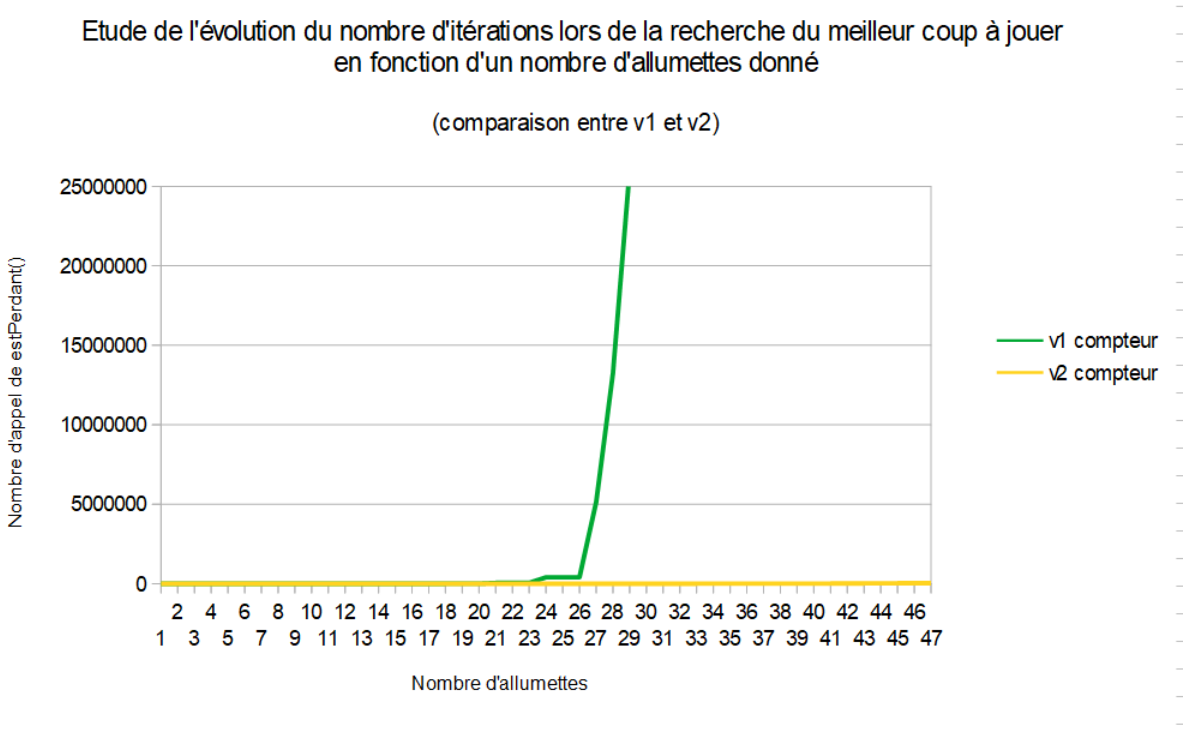
Cette version 2 est la suite de la version 1. Il est évident que s'il est intéressant de conserver les dispositions perdantes, il est tout aussi intéressant de conserver les dispositions gagnantes.

Pour cette version, nous avons appliqué la même méthodologie que dans la version n°1. Nous avons créer un ArrayList<ArrayList<Integer>> nommé gagnantes qui conservera les tableaux d'occurrences des dispositions gagnantes.

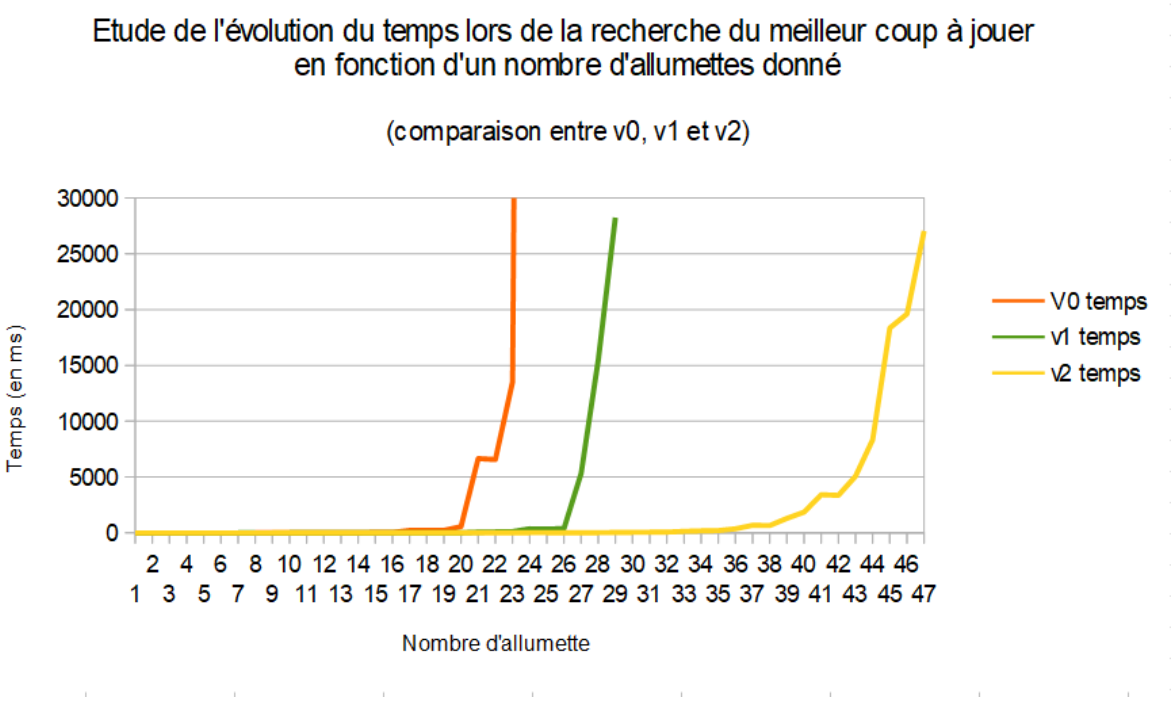
De même que dans la version 1, nous avons créé la méthode *"isFoundInGagnantes()*". Cette méthode est quasiment identique à sa version perdante mais il nous paraissait qu'avoir deux méthodes distinctes faciliterait la compréhension de l'utilisateur.

Etude de l'efficacité

Cette version implémente la sauvegarde des dispositions perdantes et gagnantes.



Sur ce graphique, on observe que le nombre d'appels de la méthode estPerdante() devient insignifiant par rapport aux versions précédentes. Le compteur de la version 2 ne semble pas bouger du tout. Pour trouver une solution de jeu à une disposition de 47 allumettes, il faut seulement 42 000 appels de la méthode estPerdante(). On comprend ici l'utilité du stockage des versions gagnantes et perdantes calculées.



On observe facilement que le nombre d'allumettes pour lesquelles on peut calculer une solution dans un temps raisonnable est bien plus grand que dans la version 1.
Ce nombre est facilement multiplié par 2 entre la version 0 et la version 2.
On peut donc comprendre l'importance de la sauvegarde des dispositions déjà calculées.

Implémentation de la version 3 :

nom du fichier source : Grundy2RecPerdantNeutre.java

Cette nouvelle version conserve les améliorations des versions précédentes et prend en compte une nouvelle règle. Si on ajoute un tas perdant à n'importe quelle disposition, la nature de cette disposition ne change pas. On s'est donc servi de cette règle pour supprimer tous les tas perdants lors de la recherche.

Ainsi, si [4] est perdant alors [3,4,5] est de la nature de [3,5]

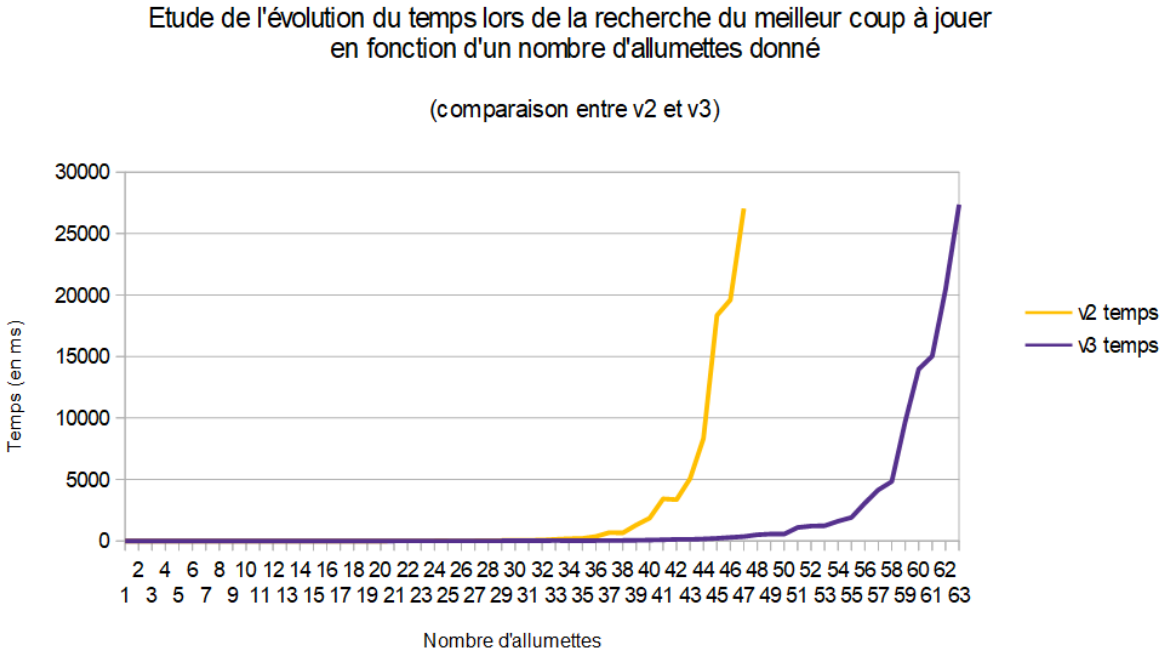
On en a aussi profité pour ajouter une deuxième condition, si deux tas sont égaux alors leur nature est perdante, on peut donc les supprimer.

Ainsi, si [4] est perdant alors [3,4,5,5] est de la nature de [3] car [5,5] est un double et [4] est perdant, on peut donc les supprimer.

Sur ce principe, on a codé la méthode "deleteLosingKnownElements()" qui prend en paramètre une disposition du jeu et le tableau des situations perdantes créé dans la version n°1. La suppression des éléments perdants permet de considérablement réduire les possibilités de jeu à calculer et donc d'améliorer l'efficacité.

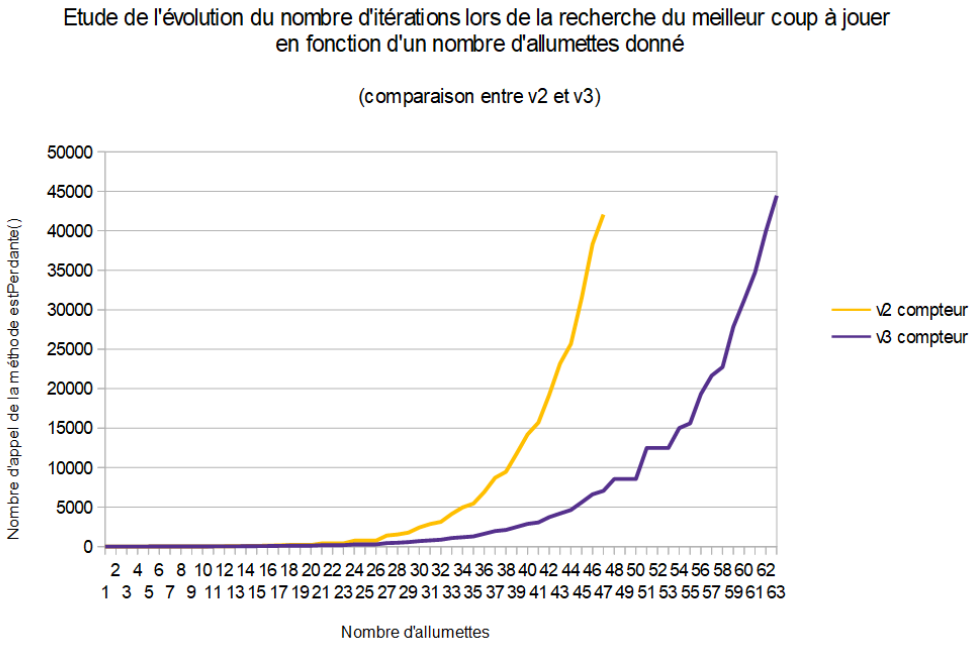
Etude de la complexité

Cette version implémente la sauvegarde des dispositions perdantes et gagnantes, ainsi que la suppression des tas perdants.



Sur ce premier graphique, on remarque qu'on diminue encore le nombre d'appels de la méthode estPerdante() ce qui nous permet d'améliorer la quantité d'allumettes pour laquelle on peut trouver une solution de jeu.

L'étude de l'évolution du temps nous permet d'affirmer que grâce à cette nouvelle version, on améliore de 15 allumettes nos possibilités.



Implémentation de la version 4 :

nom du fichier source : Grundy2RecGplusGequalsP.java

Voici, finalement, la dernière version. Dans celle-ci, on établit que les différents tas possèdent des types. Ces types sont définis grâce au tableau suivant :

T_0	T_1	T_2	T_3	T_4	T_5
0	3	5	13	18	41
1	6	8	16	21	44
2	9	11	19	24	47
4	12	14	22	27	
7	15	17	25	33	
10	28	29	30	36	
20	31	32		39	
23	34	35		42	
26	37	38		45	
50	40			48	
	43				
	46				
	49				

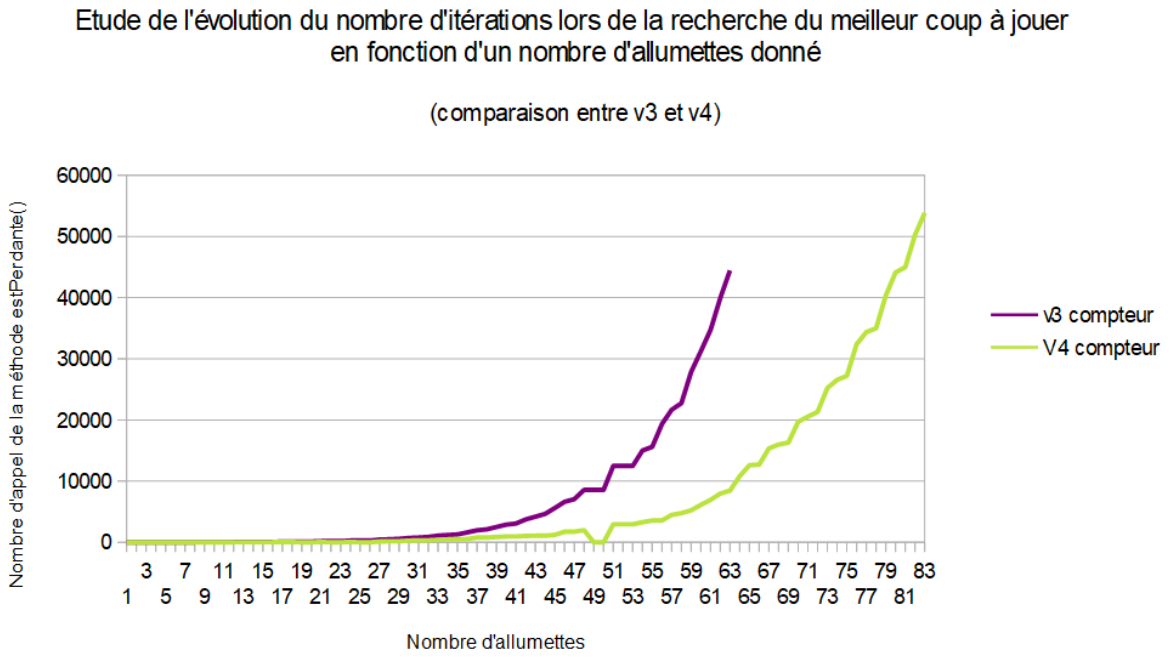
Les tas de type T0 sont les tas perdants, ce qui veut dire qu'à chaque fois qu'on tombera dessus, on pourra le supprimer sans chercher dans les dispositions perdantes déjà calculées.
Les autres tas sont des tas gagnants. Mais alors à quoi servent les différents types ?

Avec cette dernière version, arrive une nouvelle règle. Deux tas de même types constituent une situation perdante, en revanche, deux tas gagnants de types différents peuvent donner lieu à une disposition perdante ou à une disposition gagnante.

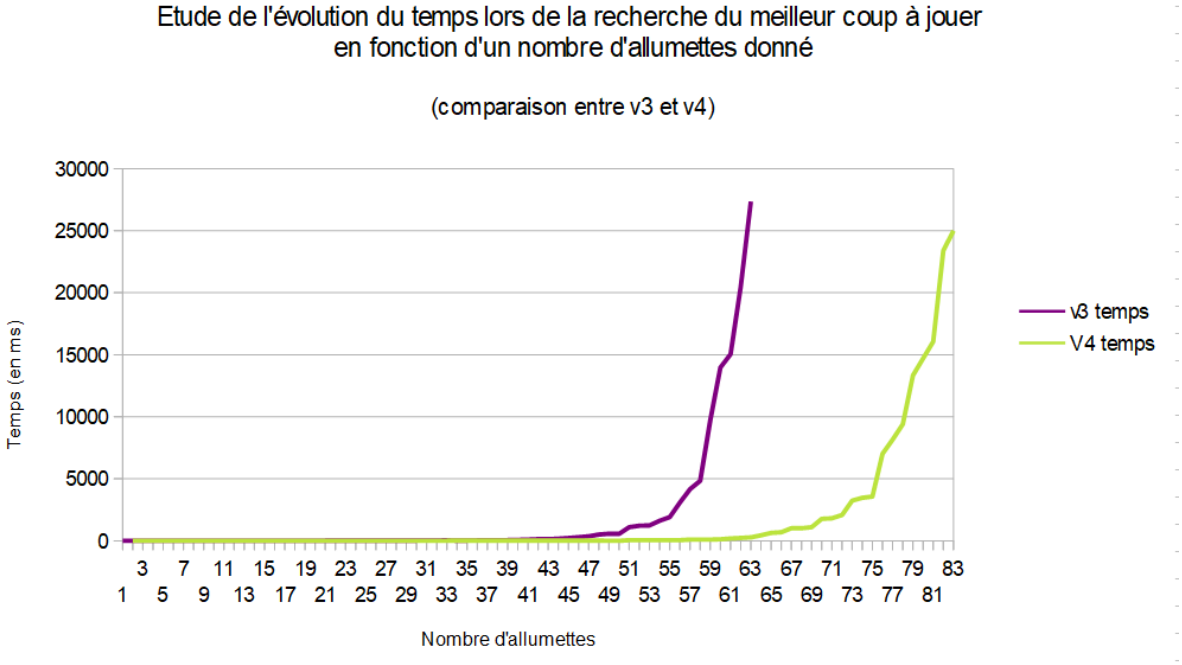
Cette version stocke les dispositions perdantes, supprime les tas perdants mais supprime aussi les couples gagnants de même type. Etant donné que nous sommes désormais en position du tableau ci- dessus, la méthode codée dans la version 3 ne trouve plus son utilité car il est bien plus efficace de comparer chaque élément de la disposition avec le tableau ci- dessus.

Nous avons donc remplacer la méthode de la version 3 par la nouvelle méthode de la version 4 : deleteWinningElementCouples

Etude de l'efficacité
Cette version implémente la sauvegarde des dispositions perdantes et gagnantes, ainsi que la suppression des tas perdants et des couples gagnants de même type.

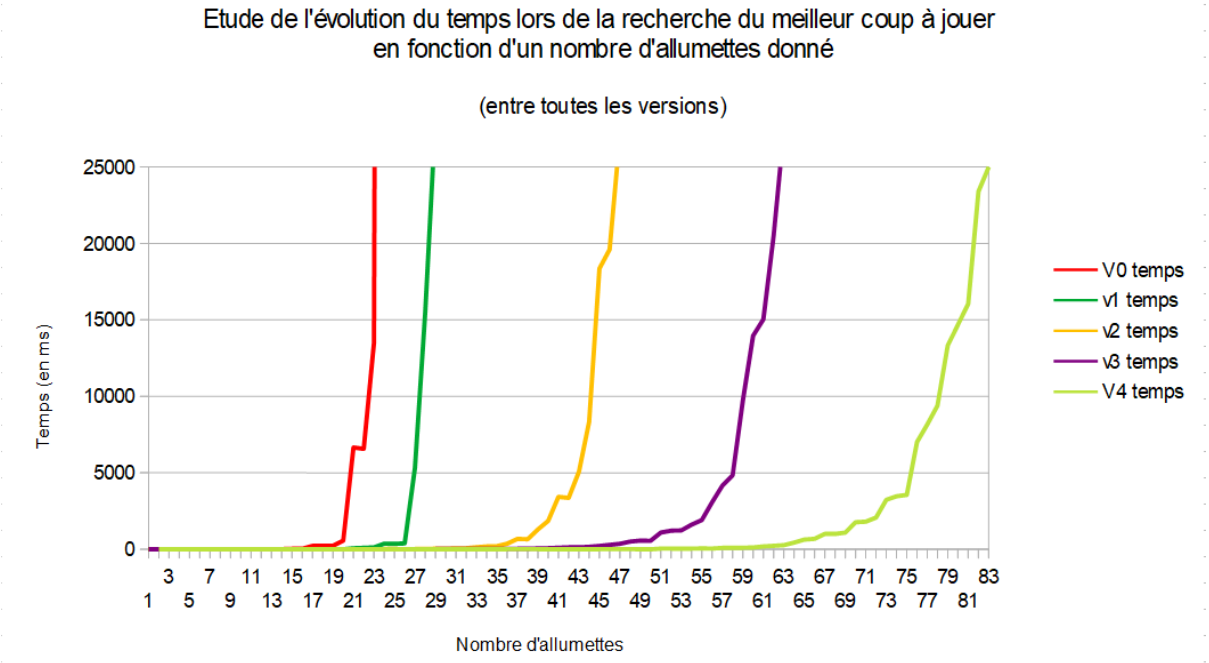


Encore une fois, la nouvelle version est meilleure que la précédente, nous autorisant à augmenter le nombre d'allumettes à calculer encore plus qu'avant.



En termes de temps, nouvelle version ajoute 15 allumettes à notre arc. On peut donc maintenant calculer le meilleur coup à jouer pour plus de 80 allumettes, agrandissant nos possibilités de jeu.

Conclusion :



Pour conclure, cette Situation d'Apprentissage et d'Evaluation a été un projet réellement intéressant. Nous avons pu nous rendre compte qu'il est crucial de coder efficacement. On peut encore le voir à ce dernier graphique, nous avons pu multiplier nos capacités par 4 grâce à de simples réflexions logiques mais aussi grâce à des connaissances extérieures comme dans la version 4. Ce projet nous a permis de comprendre l'importance de la performance des algorithmes dans le monde professionnel et nous a donné les outils pour améliorer notre propre codage. Il est évident que cette expérience a été très bénéfique pour notre apprentissage et notre développement en tant que développeur.

Annexe : Tableau des statistiques (moyenne sur 10 tests)

Nombre d'allumettes	V0 temps	v0 compteur	v1 temps	v1 compteur	v2 temps	v2 compteur	v3 temps	v3 compteur	V4 temps	V4 compteur
3	0	1	0	1	0	1	0	1		1
4	0	2	0	2	0	2	1	2	0	2
5	0	3	0	3	1	3	0	3	0	1
6	0	7	0	4	0	5	0	5	0	3
7	0	18	1	10	1	8	1	7	1	6
8	1	19	0	11	0	8	0	7	0	1
9	0	39	1	12	1	10	1	9	0	3
10	1	112	1	26	0	17	1	12	1	9
11	1	113	0	27	1	17	0	12	0	1
12	1	227	1	28	1	19	1	14	0	3
13	7	1267	2	83	2	37	2	27	1	10
14	3	2559	2	108	1	45	2	33	1	14
15	5	6527	2	218	1	58	1	38	1	16
16	15	20141	3	358	1	72	2	51	2	20
17	38	62801	4	640	2	100	2	64	2	30
18	38	128101	3	1103	3	117	2	73	1	32
19	224	422885	9	2543	3	156	3	91	1	36
20	221	1433262	18	5143	4	196	3	111	1	49
21	234	1433263	10	5144	3	196	3	111	0	1
22	568	2866527	7	5145	3	198	4	113	0	3
23	6660	38440186	70	55716	11	406	7	182	1	55
24	6568	38440187	98	55717	8	406	7	182	0	1
25	13500	76880375	122	55718	9	408	8	184	0	3
26	214689	1180286628	367	405804	19	744	13	278	2	76
27			355	405805	20	744	8	278	0	1
28			389	413098	11	746	6	280	0	3
29			5328	5149771	17	1401	10	438	3	134
30			15481	13240345	20	1539	12	493	3	168
31			28240	26035807	41	1772	18	567	2	170
32					49	2424	22	702	3	268
33					65	2858	20	787	4	317
34					74	3139	25	878	3	319
35					137	4144	36	1086	5	406
36					184	4962	18	1187	6	476
37					209	5450	24	1293	7	478
38					367	6917	31	1626	10	539
39					685	8716	38	1970	12	783
40					660	9469	43	2112	12	785
41					1299	11811	58	2499	10	859
42					1860	14226	74	2880	6	956
43					3426	15716	100	3065	6	958
44					3361	19226	132	3732	7	1041
45					5068	23190	131	4190	8	1078
46					8338	25669	167	4638	7	1080
47					18348	31523	220	5611	11	1221
48					19605	38345	290	6623	15	1736
49					27040	42061	359	7059	14	1738
50							504	8557	18	1972
51							569	8557	0	1
52							559	8559	0	3
53							1099	12494	49	2944
54							1216	12494	49	2944
55							1238	12496	42	2946
56							1611	15008	48	3293
57							1911	15600	58	3553
58							3080	19332	53	3569
59							4163	21664	93	4480
60							4836	22734	94	4759
61							9721	27848	100	5227
62							13967	31231	120	6128
63							15035	34735	188	6940
64							20478	39947	229	7971
65							27356	44457	273	8452
66									438	10816
67									637	12605
68									688	12703
69									1011	15366
70									1008	15982
71									1092	16307
72									1770	19697
73									1811	20549
74									2073	21320
75									3236	25230
76									3469	26558
77									3559	27211
78									7014	32384
79									8158	34365
80									9398	34989
81									13327	40324
82									14690	44127
83									16055	44970
84									23397	50231
85									24998	53861