INSTITUT GALILÉE - UNIVERSITÉ PARIS 13

STRUCTURE DE DONNÉES AVANCÉES
RAPPORT

Travaux Pratiques - TP 02

Étudiants :
Tiziri OULD HADDA

Enseignants :
M. Olivier BODINI
M. Julien DAVID
M. Sergey DOVGAL

30 décembre 2019

Table des matières

1 Etude théorique

La fonction de potentielle est définie :

$a(n) = c(n) + \Phi(n)\Phi(n-1)$:

- $c(n)$: coût pour la n_{ieme} operation
- $a(n)$: coût amorti pour la n_{ieme} operation.
- $\Phi(n)$: Fonction de potentiel.
- tel que $a(n) > 0$,pour quel que soit n.

Cout amortie : cas $nom_i \geq 1/3$
On a :

$$\Phi(i) = |2.nom_i - taille_i| \tag{1}$$
$$a(i) = c(i) + \Phi(i) - \Phi(i-1) = 1 + |2.nom_i - taille_i| - |2.nom_{i-1} - taille_{i-1}| \tag{2}$$

vu que $taille_n > nom_n$ on peut enlever la valeur absolue (avec l'inverse) :

$$= 1 + (taille_i - 2.nom_i) - (taille_{i-1} - 2.nom_{i-1}) \tag{3}$$

On a $taille_i = taille_{i-1}$ et $nom_i = nom_{i-1} - 1$ l'equation se simplifie comme suit :

$$\begin{aligned} &= 1 + (taille_i - 2.nom_{i-1} + 2) - (taille_i - 2.nom_{i-1}) \\ &= 1 + 2 \\ a(i) &= 3 \end{aligned}$$

Cout amortie : cas $nom_i < 1/3$

Sachant que $taille_i = taille_{i-1} * 2/3$, l'équation (3) devient :

$$\begin{aligned} &= nom_i + taille_{i-1}.2/3 - 2.nom_i - taille_{i-1} + 2.(nom_i + 1) \\ &= nom_i + taille_{i-1}.2/3 - 2.nom_i - taille_{i-1} + 2.nom_i + 2 \end{aligned}$$

$$= nom_i - \frac{taille_{i-1}}{3} + 2 \tag{4}$$

Quand on descend en dessous du seuil de suppression :

$$nom_i \leftarrow 1/3.taille_{i-1} \tag{5}$$

Sachant que :

$$taille_i \leftarrow 2/3.taille_{i-1} \tag{6}$$

Alors de l'equation (5) et (6)

$$taille_i = 2.nom_i \tag{7}$$
$$taille_{i-1} = 3.nom_i \tag{8}$$

Au seuil, on remplace dans l'équation (4) l'expression de $taille_{i-1}$ par l'equation (8), on trouve alors :

$$\begin{aligned} a(i) &= nom_i - \frac{nom_i.3}{3} + 2 = nom_i - nom_i + 2 \\ a(i) &= 2 \end{aligned}$$

2 Analyse des expériences réalisées - language C

2.1 Expériences réalisées en variant la probabilité d'avoir un ajout

Comme dans le TP1, j'ai réalisé des script pour faciliter le test avec les expériences demandés dans le sujet du TP2, passant en boucle le lancement du binaire avec l'argument p désignant la probabilité d'avoir un ajout.

En effet, le code étant le même j'ai décidé de rajouter p comme argument au main. Et pour avoir une expérience unique j'ai mis mon numéro étudiant en argument à la fonction "srand", quand le programme ne reçoit pas d'argument il laisse la variable p égale à 0.5, comme suit :

```
float p = 0.5;
printf("Valeur de p par defaut 0.5\n");
if(argc > 1){
  p = atof(argv[1]);
  printf("Changement de la valeur de p à %.2f\n",p);
}
srand(11709949);
```

J'ai choisi de mettre aussi la variable after (*struct timespec*) à zéro pour ne pas enregistrer dans la structure *analyzer*, la précédente valeur lorsque le tableau dynamique est vide.

```
for(i = 0; i < 1000000 ; i++){
  float p_rand = (float)rand()/(float)RAND_MAX;
  if(p_rand < p)
  {
    // Ajout d'un élément et mesure du temps pris par l'opération.
    ..
  }else{
    // Suppression d'un élément et mesure du temps pris par l'opération.
    if(arraylist_size(a) > 0)
    {
      timespec_get(&before, TIME_UTC);
      memory_allocation = arraylist_pop_back(a);
      timespec_get(&after, TIME_UTC);
    }else{
      //remise a zéro de la mesure
      after.tv_nsec  = 0;
      before.tv_nsec = 0;
    }
...
}
```

D'après la figure 1, on remarque tout d'abord que quand $p = 0.1$ le cout amorti (total) est approximativement celui de la suppression (juste inférieur $10ns$) qui est inférieur au cout de l'ajout lorsque celui ci domine lorsque $p = 0.9$ (un peu plus que $30ns$). Donc approximativement un rapport de 4 dans le cout amorti. Cela est peut être du au fait que quand on fait un realloc avec une mémoire reduite, ce dernier ne prend pas beaucoup de temps ou peut être qu'il ne reserve pas dans certain cas un autre tableau avec la nouvelle taille pour en faire une copie avant de supprimer le tableau original.

Quand au gaspillage mémoire, d'après la figure 2, on remarque quand l'utilisateur fait plus de suppression que d'insertion (p=0.1, 0.2 .., 0.5) la mémoire gaspiller (même au maximum) reste comme même très raisonable (8, 16, 16, 400, 25). A partir de $p = 0.6$ le gaspillage mémoire prend de l'ampleur (approximativement : 130000, 270000, 520000, 520000). Mais par contre dans ce dernier cas le nombre de copie est réduit car le temps pour atteindre le prochain seuil de capacité est retardé par les operation de suppression(voir nombre de pic et l'écart entre les pics dans la figure 2).

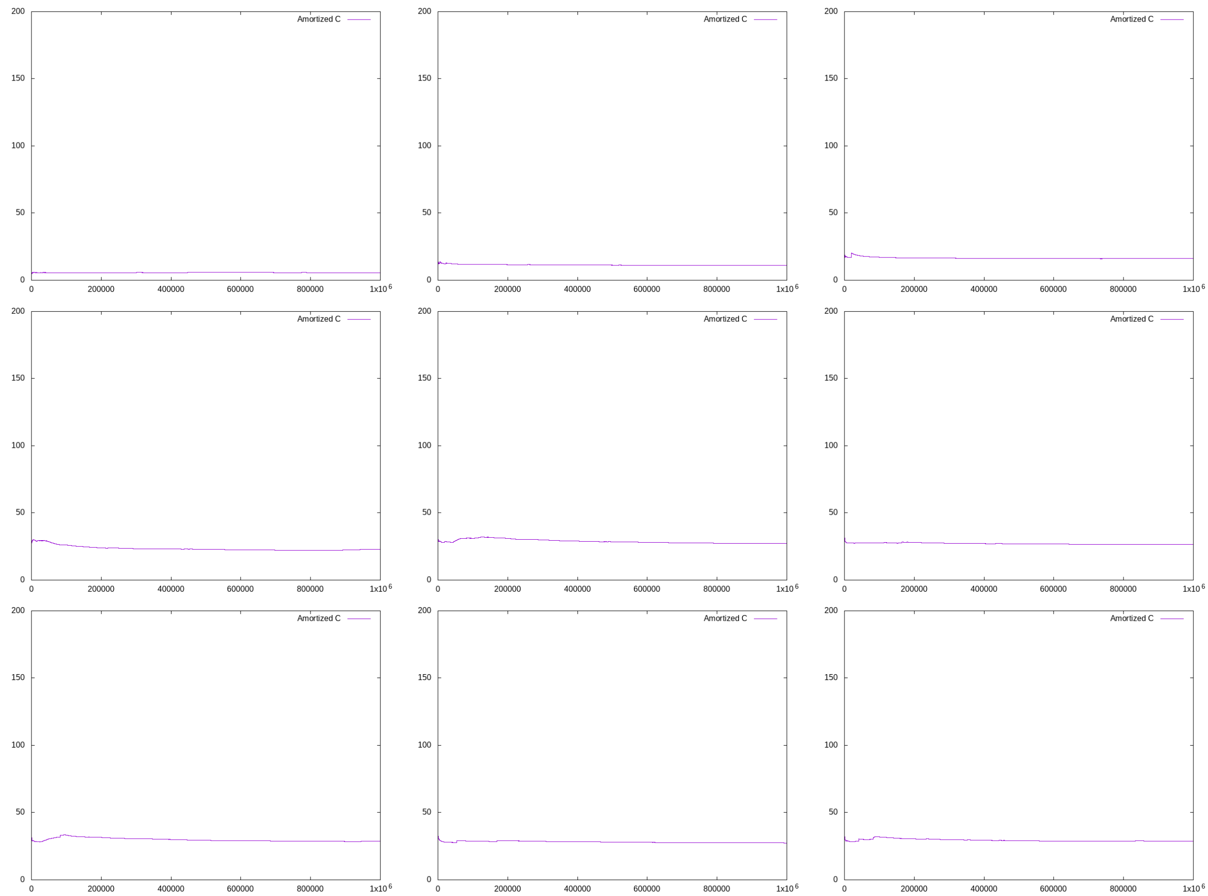

FIGURE 1 – Cout amorti (temps) - Langage C - paramètres initiaux et p = 0.1,...,0.9

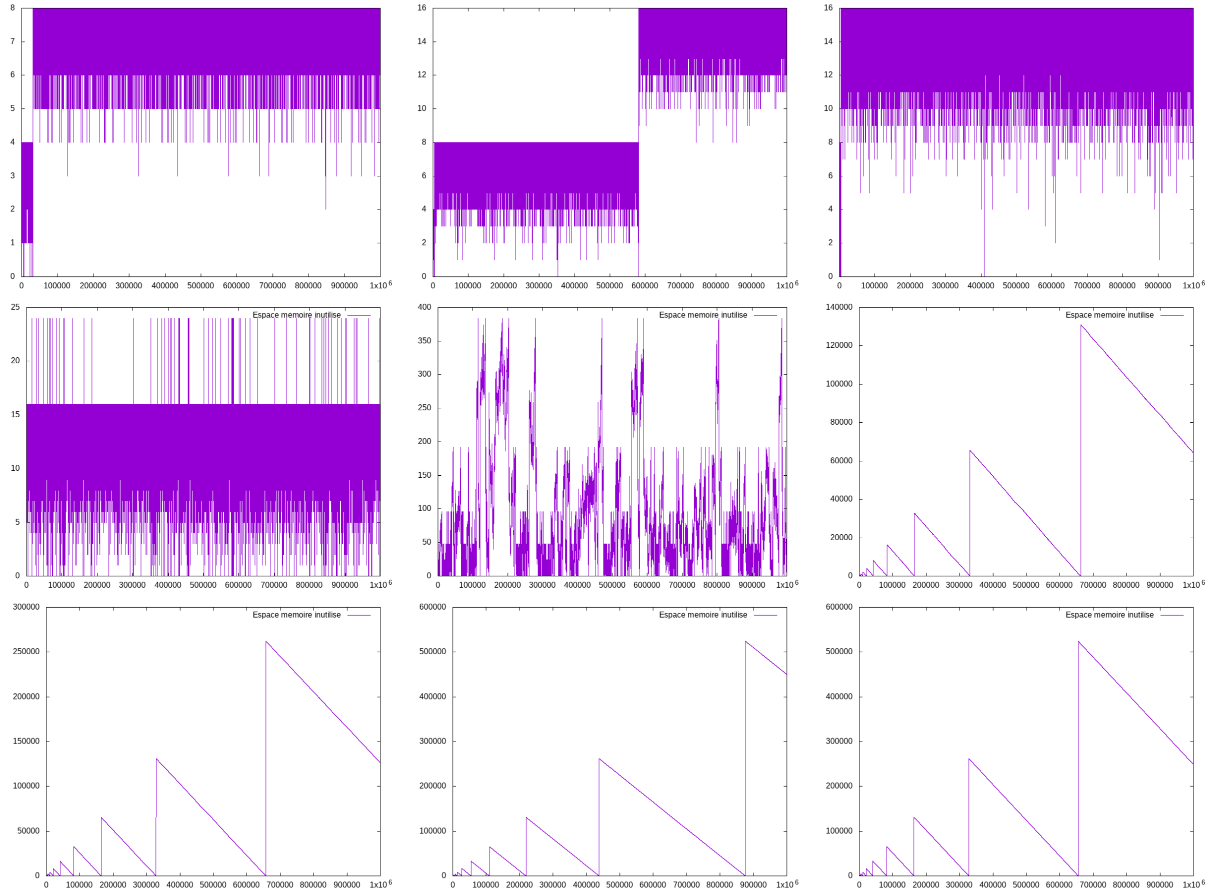

FIGURE 2 – Cout en espace mémoire - Langage C - paramêtres initiaux et p = 0.1,...,0.9

2.2 Changement de la stratégie de redimensionnement

J'ai décider de garder la même condition pour déclancher une contraction. Pour que cette condition soit possible, il faut que pour tout n :

$$\frac{n}{4} < n - \sqrt{n} \qquad \text{n : étant la capacité actuelle}$$
$$\sqrt{n} < 3.\frac{n}{4}$$

Selon la figure 3, cette condition est respecté quelque soit n.

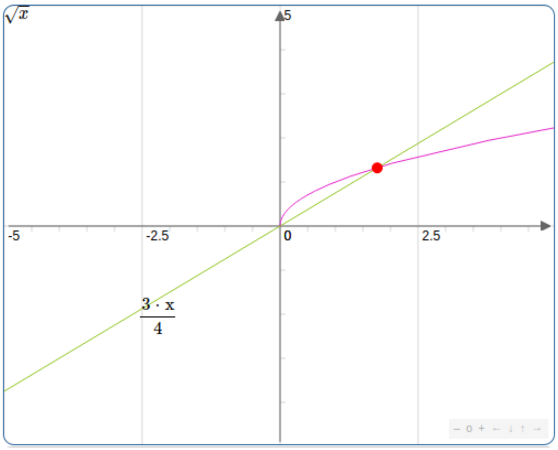

FIGURE 3 – Comparaison entre la fonction $\sqrt{n}$ et $3/4.n$

Avec ces changement le code devient :

```
#include "arraylist.h"
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
...
char arraylist_do_we_need_to_enlarge_capacity(arraylist_t * a){
  return ( a->size >= a->capacity )? TRUE: FALSE;
}
void arraylist_enlarge_capacity(arraylist_t * a){
  a->capacity = a->capacity + sqrt(a->capacity);
  a->data = (int *) realloc(a->data, sizeof(int) * a->capacity);
}
char arraylist_do_we_need_to_reduce_capacity(arraylist_t * a){
  return ( a->size <= a->capacity/4 && a->size >4 )? TRUE: FALSE;
}
void arraylist_reduce_capacity(arraylist_t * a){
  a->capacity = a->capacity - sqrt(a->capacity);
  a->data = (int *) realloc(a->data, sizeof(int) * a->capacity);
}
```

D'après les expériences effectuées avec la nouvelle stratégie de redimensionnement, on remarque dans la figure 4 que le cout amorti est important par rapport à la statégie initiale, car les opérations de copie et d'allocation en mémoire arrive trop souvent(surtout lorsqu'on beaucoup d'opération d'ajout $p > 0.5$ - voir figure 5). Par contre, on remarque une forte amélioration dans le gaspillage mémoire en passant de 520000 de mémoire perdu par la stratégie initial à juste 1000(figure 6).

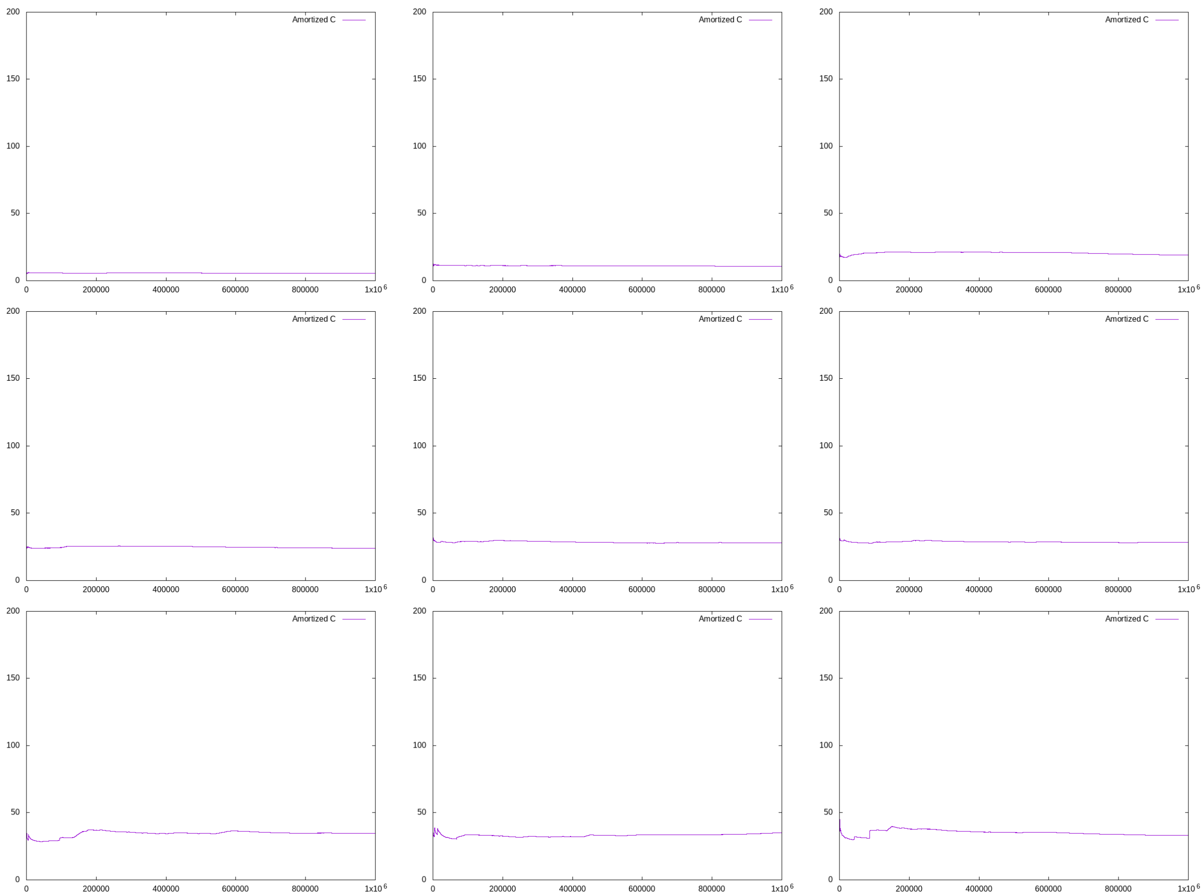

Figure 4 – Cout amorti - Langage C - Nouvelle stratégie de redimentionnement $(n+/-\sqrt{n})$ et pour p = 0.1, ..., 0.9

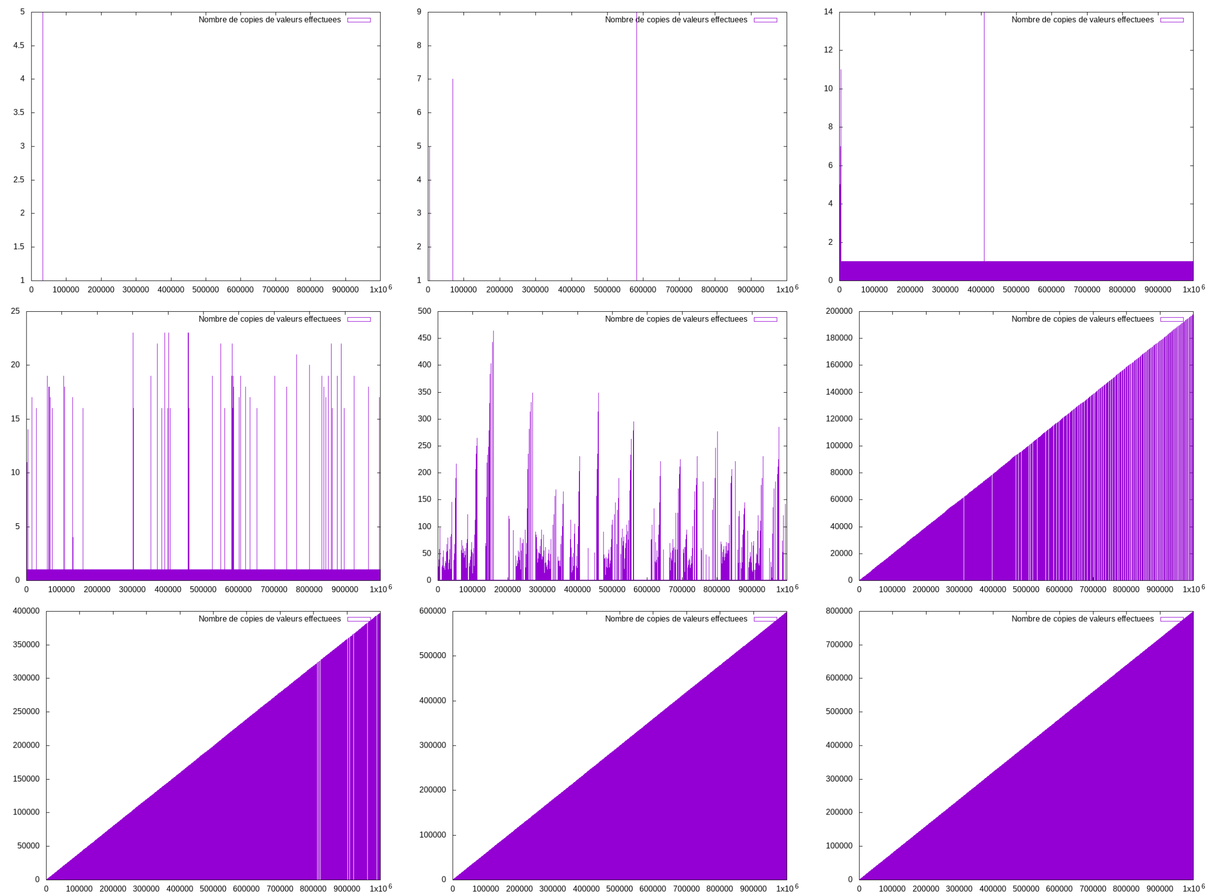

FIGURE 5 – Nombre de copie - Langage C - Nouvelle stratégie de redimentionnement $(n+/-\sqrt{n})$ et pour p = 0.1, ..., 0.9

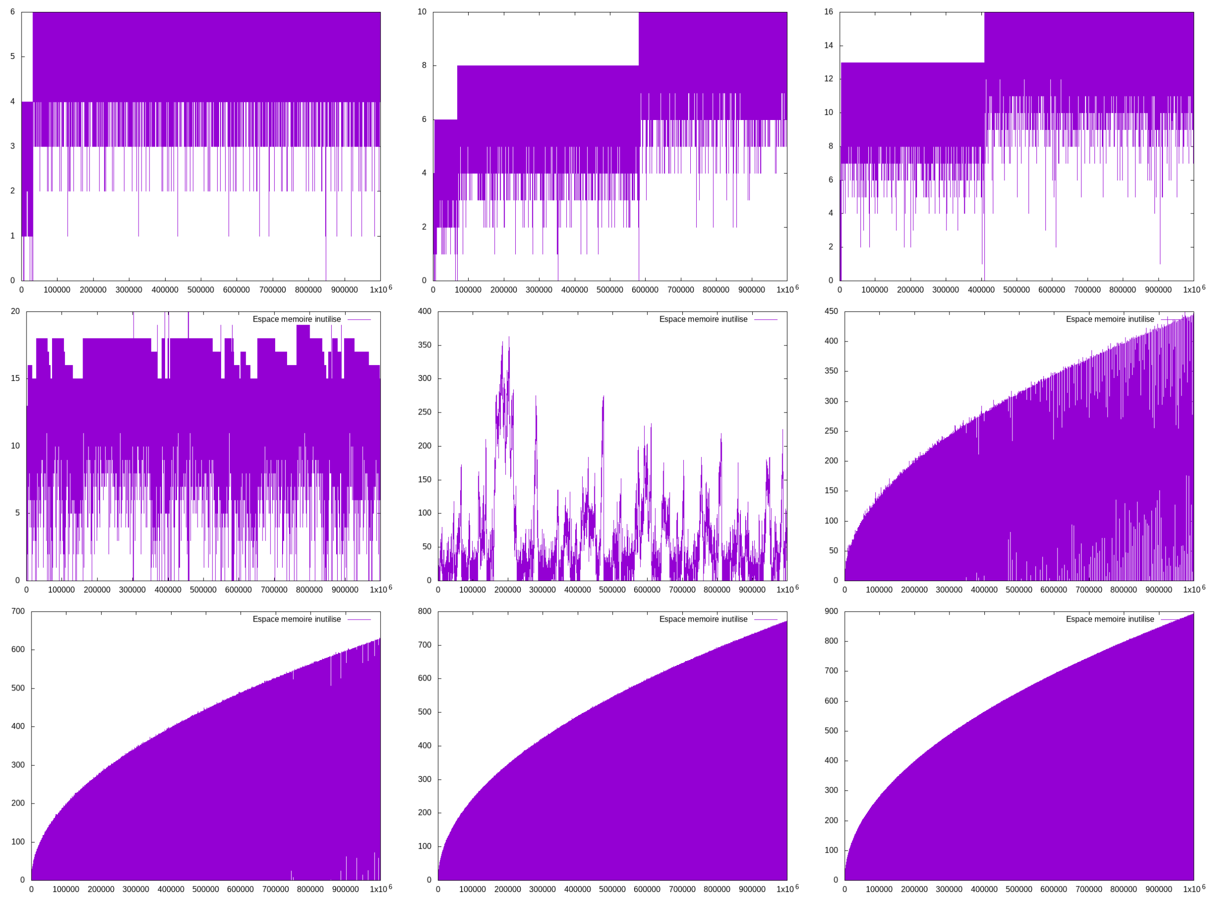

FIGURE 6 – Mémoire inutilisée - Langage C - Statégie de redimensionnement $taille_{i+1} = taille_i + / - \sqrt{taille_i}$ et p = 0.1,...,0.9

3 Conclusion

D'après l'analyse faite sur les deux stratégies on en déduit que :

— le meuilleur choix pour avoir une cout amorti meuilleur est celui de la statégie initial.

— Concernant la nouvelle stratégie, elle est meuilleur lorsqu'on ne veut pas avoir beaucoup de gaspillage mémoire mais son cout amorti est un peu plus important que la stratégie initial.

— Concernant le choix de p lorsque il ne dépasse pas 0.5 la nouvelle stratègie est nettement meuilleur en cout mémoire et même equivalent à la stratégie initial en terme de cout amorti.

4 ANNEXE

4.1 Script Bash lancement du test en faisant varier p

```
echo "Experience : variation de la variable p"
exp="$1"
cd C/
make clean
make
prob=0.0
pas=0.1

rm -rf  plots/*.png
rm -rf  plots/*.plot

rm -rf ../plots/exp_$1
mkdir  ../plots/exp_$1

for i in `seq 1 10`;
do
echo ""
prob=$(echo $prob+$pas | bc)

    ./arraylist_analysis $prob

cd ../plots/
gnuplot -c plot_result --persist

rm -rf exp_$1_$i
mkdir  exp_$1_$i

cp -rf *.plot exp_$1_$i/
cp -rf *.png  exp_$1_$i/

cd exp_$1_$i/
for f in *
do
pre="${exp}_$i"
echo "$f -> ../exp_${exp}/${pre}_$f"
mv "$f" "../exp_${exp}/${pre}_$f"
done
cd ../
rm -rf exp_$1_$i/
echo "Experience exp_${exp}_$i OK"
cd ../C/
echo ""
done
echo "Fin de l'experience"
```