Création de phrases réflexives

Céline BUGAUD Marco TESSARI Jérôme POUILLER

EPITA - Juin 2004

Résumé

This document studies reflexive sentences generation. A reflexive sentence is a sentence which describe itself. For example : *This sentence contains 1 “0”, 7 “1”, 3 “2”, 2 “3”, 1 “4”, 1 “5”, 1 “6”, 2 “7”, 1 “8” and 1 “9”*. This report tries to determine domain of existence of these sentences. Next, it try to find them using algorithms exploring all domain. Then, usage of some probabilist algorithms (genetic algorithm and simulated annealing) are studied. Finaly, it show how to resolve problem with an exemple of quantic algorithm (Grover).

Ce document étudie la génération de phrase réflexives. Une phrase réflexive est une phrase se décrivant elle-même. Par exemple : *Cette phrase contient 1 “0”, 7 “1”, 3 “2”, 2 “3”, 1 “4”, 1 “5”, 1 “6”, 2 “7”, 1 “8” et 1 “9”*. Ce rapport essaie tout d’abord de déterminer le domaine d’existence de ces phrases. Puis, il essaie de les trouver à l’aide d’algorithmes explorant tout le domaine. Ensuite, on étudiera l’utilisation de certains algorithmes probabiliste (algorithmes génétiques et recuit simulé). On finira par expliquer comment résoudre le problème à l’aide d’un algorithme quantique (Grover).

Table des matières

1 Présentation du problème 2
1.1 Problème 1 . . . 2
1.2 Problème 2 . . . 2
1.3 Problème 3 . . . 2

2 Élagage du domaine 4
2.1 Application aux problèmes 1 et 2 . . . 4
2.2 Extension au problème 3 . . . 7
2.3 Implémentation et résultats . . . 9
2.3.1 Résultats de l'implémentation actuelle . . . 9
2.3.2 Optimisations implémentées . . . 9
2.3.3 Optimisations restant à étudier . . . 9

3 Implémentations naïves 11
3.1 Brute Force . . . 11
3.2 Algorithme par correction d'erreur . . . 12
3.3 Détails d'implémentation . . . 13
3.4 Attentes . . . 13
3.5 Procédure de test . . . 14
3.6 Résultats . . . 14

4 Recuit simulé 16
4.1 Principe de l'algorithme de recuit simulé . . . 16
4.2 Choix des transformation . . . 17
4.3 Mesure de l'énergie . . . 17
4.4 Procédure de test . . . 18
4.5 Résultats . . . 18

5 Algorithme Génétique 20
5.1 Principe d'un algorithme génétique . . . 20
5.1.1 Diversification . . . 20
5.1.2 sélection . . . 20
5.1.3 Récapitulatif des paramètres . . . 21
5.2 Implémentation . . . 21
5.2.1 Initialisation de la population . . . 21
5.2.2 Choix des paramètres . . . 21
5.2.3 Fonction d'erreur . . . 21
5.2.4 Opérations sur le génomes . . . 22
5.3 Résultats - Performances . . . 22

5.3.1 Résultats en chiffes . . . 22
5.3.2 Conclusions . . . 31

6 Utilisation de l'algorithme de Grover 33
6.1 Rappels sur les ordinateurs quantique . . . 33
6.2 L'algorithme de Grover . . . 34
6.3 Application de l'algorithme de Grover . . . 35
6.3.1 Application au problème 1 . . . 35
6.3.2 Application au problème 3 . . . 36
6.4 Conclusion . . . 36

A Implémentation des algorithmes naifs et du recuit simulé 37
A.1 Bibliothéque commune . . . 37
A.1.1 main.c . . . 37
A.1.2 args.h . . . 38
A.1.3 args.c . . . 39
A.1.4 int_to_french.h . . . 41
A.1.5 int_to_french.c . . . 41
A.1.6 génération de occurs.h : occurs_gen.c . . . 42
A.1.7 itf_static.h . . . 43
A.1.8 itf_static.c . . . 44
A.2 Elagage du domaine . . . 45
A.2.1 prune_comm.c . . . 45
A.2.2 prune_alpha.h . . . 45
A.2.3 prune_alpha.c . . . 46
A.2.4 prune_numer.h . . . 51
A.2.5 prune_numer.c . . . 51
A.3 Algorithmes naifs . . . 55
A.3.1 counts.h . . . 55
A.3.2 counts.c . . . 56
A.3.3 naive.h . . . 59
A.3.4 naive.c . . . 60
A.4 Recuit simulé . . . 61
A.4.1 recuit.h . . . 61
A.4.2 recuit.c . . . 61

B Implémentation de l'algorithme génétique 65
B.0.3 BiomorphFactory.java . . . 65
B.0.4 BiomorphToFile.java . . . 65
B.0.5 Population.java . . . 65
B.0.6 Biomorph.java . . . 68
B.0.7 GenAlgo.java . . . 70
B.0.8 counter/CounterBiomorphFactory.java . . . 72
B.0.9 counter/CounterLetterBiomorphFactory.java . . . 72
B.0.10 counter/CounterBiomorph.java . . . 73
B.0.11 counter/CounterLetterBiomorph.java . . . 75
B.0.12 counter/Main.java . . . 77

Références 80

Chapitre 1

Présentation du problème

Le problème que nous allons essayer de résoudre est celui des phrases réflexives. Une phrase réflexive est définie comme une phrase annonçant le nombre de lettres ou de chiffres qu'elle contient.

> Il y a 6 poules et 1 coq dans le poulailler. Dans cette phrase il y a : 1 0, 7 1, 4 2, 1 3, 2 4, 1 5, 2 6, 2 7, 1 8, 1 9.

La phrase ci-dessus est une phrase réflexive comptant les chiffres. La phrase est composée de deux parties, la première partie introduit le contexte et peut être considérée comme une initialisation. C'est la partie statique de la phrase. Dans notre cas c'est : *Il y a 6 poules et 1 coq dans le poulailler*. La deuxième partie annonce le décompte des chiffres.

Un *coefficient* désigne le nombre de fois qu'apparaît un chiffre, par exemple dans la phrase d'exemple 7 a pour coefficient 2.

Nous essayerons de trouver toutes les solutions possibles du problème, puis, nous essayerons de le résoudre grâce à des méthodes numériques.

Afin de faciliter la démarche, le problème a été décomposé en trois.

1.1 Problème 1

Le problème numéro 1 consiste à considérer uniquement la deuxième partie de la phrase et de compter uniquement les chiffres sans partie statique. Par exemple :

> [Dans cette phrase il y a :] 1 0, 7 1, 3 2, 2 3, 1 4, 1 5, 1 6, 2 7, 1 8, 1 9.

1.2 Problème 2

Dans le deuxième problème on ajoute la phrase d'initialisation comme vu dans le premier exemple et on travaille toujours uniquement sur les chiffres.

1.3 Problème 3

Dans le troisième problème on garde toujours la phrase d'initialisation, mais au lieu de compter les chiffres on compte les lettres.

De plus le décompte des lettres est écrit en toute lettres c'est-à-dire :

0 =zéro
1 =un
2 =deux
3 =trois
92 =quatre-vingt-douze
99 =quatre-vingt-dix-neuf
100 =cent
101 =cent un
102 =cent deux
200 =deux cents
1001 =mille un
3214 =trois milles deux cents quatorze

Chapitre 2

Élagage du domaine

Notre premier travail consiste à trouver le domaine de définition de notre problème et à le réduire aux maximum. Nous verrons par la suite que ce travail débouchera sur la création d'un algorithme de recherche exhaustif élaguant dynamiquement le domaine de recherche.

2.1 Application aux problèmes 1 et 2

Nous allons essayer dans cette partie d'élaguer le domaine de recherche de la solution pour les problème 1 et 2.

On note n_i le coefficient correspondant au chiffre i et n_{0i} le nombre de chiffre i dans la partie statique de la phrase. Par exemple dans le problème 1 on a : $\forall i, n_{0i} = 1$, car on est en train d'énumérer un par un les chiffes et leurs coefficients. Donc chaque chiffre apparaît une fois. Le coefficient étant considéré la partie dynamique de la phrase.

Première constatation, la limite inférieure de n_i ne peut pas être inférieure à n_{0i}

Nous remarquons tous d'abord que la taille (le nombre de chiffres) d'un nombre vaut $\lfloor \log(n) + 1 \rfloor$.

On remarque aussi que la somme des coefficients est égale au nombre de chiffres apparaissant dans la phrase.

$$\sum_{i=0}^{9} n_i = \sum_{j=0}^{9} \left(n_{0j} + \lfloor \log(n_j) + 1 \rfloor \right) \tag{2.1}$$

Un élément d'une somme ne peut pas être supérieur à la somme. On en déduit que :

$$\sum_{i=0}^{9} n_i - \sum_{i=0}^{9} n_{0i} = \sum_{j=0}^{9} \lfloor \log(n_j) + 1 \rfloor \tag{2.2}$$

$$\forall i, n_i - n_{0i} \leq \sum_{j=0}^{9} \lfloor \log(n_j) + 1 \rfloor \tag{2.3}$$

$$\forall i, n_{0i} \leq n_i \leq n_{0i} + \sum_{j=0}^{9} \lfloor \log(n_j) + 1 \rfloor \tag{2.4}$$

On peut encore élaguer notre domaine en disant que n_i est inférieur au nombre de fois que le chiffre i peut apparaître dans tous les autres coefficients. Cette optimisation réduit assez peu le domaine pour les cas des problèmes 1 et 2. Mais, elle aura son importance dans le cas du problème 3.

L'inéquation 2.4 n'est pas triviale à résoudre. Pour cette raison, nous avons décidé d'utiliser une méthode numérique pour en trouver le résultat. On majore donc $\sum_{j=0}^{9} \lfloor \log(n_j) + 1 \rfloor$ par $10 \lfloor \log(\max_j(n_j)) + 1 \rfloor$:

$$n_{0i} \leq \max_j(n_j) \leq 10 \lfloor \log(\max_j(n_j)) + 1 \rfloor + n_{0i} \tag{2.5}$$

On cherche la valeur de $\max_j(n_j)$ numériquement par dichotomie[1]. Pour cela on doit borner le domaine de $max(n_j)$ ce qui est tout naturel en informatique car nous n'avons pas de nombre infinis. $\forall i$, la solution $\max_j(n_j)$ majore n_i.

A partir de maintenant, nous prendrons comme base le problème 1. Pour cela, nous fixons $n_{0i} = 1$. Les étapes suivantes se généralisent au problème 2 sans difficultés. Soit inf_i et sup_i les limites respectivement inférieure et supérieure de n_i. A ce stade, on a $\forall i$, $inf_i = 1$ et $sup_i = 21$.

Dans le cas du problème 1 ($n_{0i} = 1$), on obtient

$$\forall i, n_i \leq 21 \tag{2.6}$$

Chacun de nos coefficients possède au pire deux chiffres puisque $\lfloor \log(21)+1 \rfloor = 2$. On peut en déduire que[2]

$$\sum_{j=0}^{9} \left(n_{0j} + \lfloor \log(n_j)+1 \rfloor\right) \leq 30 \tag{2.7}$$

Or d'après 2.1 :

$$\sum_{j=0}^{9} \left(n_{0j} + \lfloor \log(n_j)+1 \rfloor\right) = \sum_{i=0}^{9} n_i \leq 30 \tag{2.8}$$

La somme des coefficients ne peut donc pas dépasser 30 (2.8) et aucun coefficient ne peut dépasser 21 (2.6).

Néanmoins, attribuer 21 à “9” signifierais que le chiffre “9” apparaît 21 fois dans la phrase. Or, 2.8 nous dit que $\sum_{i=0}^{9} n_i < 30$. On ne pourra jamais placer 21 “9” dans la phrases sans que la somme des coefficients ne dépasse 30.

On cherche maintenant quel peut être la valeur de n_9 sans violer 2.8. On sait qu'il faudra instancier n_9 le chiffre “9” :

$$n_9 + n_9 * 9 \leq \sum_{i=0}^{9} n_i \leq 30 \tag{2.9}$$

On remarque donc qu'il ne peut y avoir plus de 3 “9”.

Néanmoins, si on applique le même raisonnement avec “1” :

$$n_1 + n_1 * 1 \leq 30 \tag{2.10}$$

On en déduit que $n_1 \leq 15$. Or, pour faire apparaître 15 “1” dans la phrase, il est nécessaire d'utiliser des nombres à plusieurs chiffres. On définit la fonction $instancemin(i,x,y)$ qui permet de placer x “i” sur y nombres de manière à obtenir la somme la plus petite possible :

$$\begin{aligned}
instancemin(9,3,9) &= 9+9+9 = 27\\
instancemin(2,4,3) &= 22+2+2 = 26\\
instancemin(1,8,3) &= 111+111+11 = 233\\
instancemin(1,15,9) &= 11+11+11+11+11+11+1+1+1 = 69\\
instancemin(1,12,9) &= 11+11+11+1+1+1+1+1+1 = 39\\
instancemin(1,11,9) &= 11+11+1+1+1+1+1+1+1 = 28\\
instancemin(1,10,9) &= 11+1+1+1+1+1+1+1+1 = 19\\
instancemin(2,10,9) &= 22+2+2+2+2+2+2+2+2 = 19
\end{aligned}$$

[1] Il existe sûrement des manières plus optimales pour résoudre ce problème, mais l'équation n'intervenant qu'une fois, nous ne nous sommes pas penchés sur le problème

[2] n'oublions pas que l'on travaille sur le problème 1 et par conséquent $n_{0j} = 1$

On en déduit que n_i doit satisfaire

$$n_i + instancemin(i, n_i, 9) \leq 30 \quad (2.11)$$

Un autre problème arrive lorsque n_i contient le chiffre “i”. On introduit la fonction $nboccurence(i,x)$ qui retourne le nombre d’occurrences du chiffre “i” dans le nombre x.

n_i doit donc satisfaire

$$instancemin(i, n_i, 9) + n_i - nboccurence(i, n_i) \leq 30 \quad (2.12)$$

Si l’équation précédente n’est pas vérifiée, on sait que l’équation 2.8 ne l’est pas non plus.

En généralisant de manière à obtenir une formule valable si l’on a déjà instancié des variables, on a :

$$instancemin(i, n_i, 9) + \sum_{j \in \text{déjà instanciées}} n_j - nboccurence(i, n_i) \leq \sum_{j=0}^{9} \lfloor \log(\max(n_j)) + 1 \rfloor \quad (2.13)$$

Pour chaque valeur de $n_i \leq 21$, on vérifie si la condition est vérifiée, on s’arrête dès qu’elle est vérifiée[3]. On remarque alors qu’à partir du moment ou la taille ($\lfloor \log(n_j) + 1 \rfloor$) des coefficients diminue, leur somme diminue et on peut réitérer le processus. Vous trouverez une implémentation de ce principe en annexe dans la fonction prunemax() de prune_numer.c

Pour notre problème 1, on obtient :

$$\begin{aligned} sup_0 &= 21 \\ sup_1 &= 12 \\ sup_2 &= 8 \\ sup_3 &= 6 \\ sup_4 &= 5 \\ sup_5 &= 4 \\ sup_6 &= 4 \\ sup_7 &= 3 \\ sup_8 &= 3 \\ sup_9 &= 3 \end{aligned}$$

Nous avons maintenant élagué le domaine de chaque coefficient n_i entre inf_i et sup_i. Le but maintenant est de parcourir notre ensemble de solutions de manière intelligente. On commence donc à instancier un coefficient en commençant par la borne supérieure de son domaine, puis on instancie chacun des coefficients jusqu’à ce que l’équation 2.1 soit violée ou que l’on tombe sur une solution.

On remarque qu’un gros coefficient a plus de chances d’entraîner une inconsistance. Pour cette raison, on commencera à instancier par la borne supérieure du domaine[4]. Une optimisation consisterait à trouver la valeur à instancier la plus forte parmi toutes les variables et à l’instancier.

On remarque qu’au cours de l’algorithme, lorsque que le domaine d’un coefficient est réduit à une seule valeur, on peut alors considérer ce coefficient comme faisant partie de la partie statique de la phrase. On peut donc mettre à jour les valeurs de inf_i et de sup_i en conséquence.

On remarque aussi que l’on peut élaguer les domaines de nos variable au cours de notre algorithme. Effectivement, lorsque l’on instancie un coefficient, on peut considérer qu’il appartient (de manière temporaire) à la partie statique de phrase. On peut donc recalculer les inf_i et les sup_i en conséquence.

[3] Il y a peut-être moyen d’optimiser ce processus en testant les valeurs de n_i par dichotomie, mais nous n’avons pas eu le temps d’étudier cette optimisation.

[4] Cette règle est connue dans les techniques de Programmation Par Contraintes comme *first fail* c’est-à-dire “le plus de contraintes en premier”

2.2 Extension au problème 3

On note n_i le coefficient correspondant à la lettre i et n_{0i} le nombre d'occurrences de la lettre i dans la partie statique de la phrase

Tout comme pour les problèmes 1 et 2, la limite inférieur de n_i ne peut pas être inférieure à n_{0i}

Tout d'abord, on remarque qu'en français, certaines lettres n'apparaissent dans aucun nombre. C'est le cas des cinq lettres : "b", "j", "k", "w" et "y". On peut donc fixer leurs coefficients à leur nombre d'occurrences dans la partie statique. On travaille maintenant sur 21 lettres et non 26. On remarquera aussi que les lettres "m" et "l" n'apparaissent que dans mille, million et milliard.

Nous devons adapter l'équation 2.1 à notre problème. La taille d'un nombre n'est plus $\lfloor \log(n_j)+1 \rfloor$, mais $sizeof(n_j)$.

$$\sum_{i=0}^{21} n_i = \sum_{j=0}^{21} \left(n_{0j} + sizeof(n_j))\right) \tag{2.14}$$

On peut sans problème passer à l'équation 2.4 :

$$\sum_{i=0}^{21} n_i - \sum_{i=0}^{21} n_{0i} = \sum_{j=0}^{21} sizeof(n_j) \tag{2.15}$$

$$\forall i, n_i - n_{0i} \leq \sum_{j=0}^{21} sizeof(n_j) \tag{2.16}$$

$$\forall i, n_{0i} \leq n_i \leq n_{0i} + \sum_{j=0}^{21} sizeof(n_j) \tag{2.17}$$

Tout comme pour les problèmes 1 et 2, on veut maintenant de majorer $sizeof(n_j)$. Nous travaillons sur un ordinateur dont l'adressage de la mémoire est borné. Par conséquent, la taille de notre phrase ne peut dépasser $2^{64} = 18446744073709551616$ lettres[5]. Remarquez que le raisonnement qui va suivre est applicable sur des valeurs supérieures à 2^{64}. L'important est d'avoir une borne de départ. On peut en déduire que $\max_{i=[0..2^{64}]}(sizeof(i))$ est 14494494494494494494[6]. Ce nombre s'écrit avec 238 lettres. Donc, $\forall i, sizeof(n_i) \leq 238$. Par conséquent, $\sum_{j=0}^{21} sizeof(n_j) \leq 4998$

$$\sum_{i=0}^{21} sizeof(n_i) \leq 4998 \tag{2.18}$$

$$\forall i, n_{0i} \leq n_i \leq n_{0i} + 4998 \tag{2.19}$$

Maintenant que l'on connaît la limite de chaque n_i, on peut mieux majorer $sizeof()$. On avait tout a l'heure recherché la plus grande valeur de $sizeof()$ sur $[0..2^{64}]$, on peut maintenant refaire la même chose sur $[0..4998+n_{0i}]$. Pour cela, on teste toutes les solutions possibles. L'opération n'est pas aberrante en temps surtout si on optimise en sachant que le plus grand nombre entre 0 et 10 est quatre et entre 10 et 100 est quatre vingt quatorze. On peut itérer l'opération en recherchant le maximum de $sizeof()$ sur $[0..21 \times sizeof(n_i) + n_{0i}]$. De plus, on remarque que si l'on passe sous la barre des 1000, on peut fixer le coefficient de "m" et "l" et ainsi ne travailler que sur 19 valeurs.

Chaque coefficient est maintenant borné à une valeur assez faible. Pour $n_{0i} = 1$, on obtient :

itération 1 :
- Recherche sur : $[0..4999]$
- Meilleur nombre : 4494

[5] On considère que l'on utilise une machine actuelle

[6] Le plus grand nombre entre 0 et 100 est quatres vingt quatorze et le plus grand nombre entre 0 et 10 est quatre. On peut facilement construire ce nombre.

– $sizeof(4494)$: 42
– $\forall i, 1 \leq n_i \leq 42 * 21 + 1 = 883$

On remarque que $n_i < 883 < 1000$, donc, aucun “m” ni aucun “l” ne peut apparaître. On fixe $n_\mathsf{m} = n_\mathsf{l} = 1$ et on update les n_{*0}.

itération 2 : – Recherche sur : [0..883]
– Meilleur nombre : 494
– $sizeof(494)$: 31
– $\forall i, 1 \leq n_i \leq 31 * 19 + 1 = 589$

itération 3 : – Recherche sur : [0..589]
– Meilleur nombre : 494
– Pas mieux que le précédant, donc on arrête.

On remarque que notre domaine qui était $[0..2^{64}]$ est passé à [0..589]. Jusqu'à présent, on s'est intéressé à $sizeof()$. Cette méthode était intéressante car elle était facilement optimisable. Pourtant :

$$\forall i, n_i = \text{nombre d'occurrences de } i \tag{2.20}$$

Par exemple, pour “a”. On vient de trouver que $n_\mathsf{a} < 589$. Mais, si $n_\mathsf{a} = 589$, sachant que les coefficient peuvent avoir au plus 42 lettres, cela signifierait que tous les coefficients ne contiennent que des “a”. Dans les problèmes 1 et 2, on avait dit que l'on pouvait optimiser notre algorithme en calculant le nombre de fois où le chiffre pouvait apparaître dans les coefficients plutôt qu'en prenant la taille (en nombre de chiffres). Cette optimisation n'était pas primordiale dans les problème 1 et 2 (au mieux la différence entre les deux calculs était de 1). Dans le problème 3, la différence entre le la taille (en nombre de lettres) d'un nombre et le nombre d'apparitions maximale d'une lettre est énorme. On peut exploiter cette propriété.

On introduit la fonction $nboccurence(i,x)$ qui retourne le nombre d'occurences de la lettre i dans le nombre x. On peut donc majorer n_i par

$$n_i < \sum_{j=0}^{18} \max_{x=0..\max(n_j)} nboccurenc(i,x) \tag{2.21}$$

De la même manière que pour les problèmes 1 et 2, on répète le processus jusqu'à ce que nos n_i soit stable. On peut optimiser la méthode en utilisant un tableau contenant les occurrences de toutes les lettres de chaque nombre entre 0 et $\max_j(\max(n_j))$. Avec $\forall i, n_{0i} = 1$, on obtient :

$$\begin{aligned}
sup_\mathsf{a} &= 24 & sup_\mathsf{n} &= 34 \\
sup_\mathsf{b} &= 0 & sup_\mathsf{o} &= 28 \\
sup_\mathsf{c} &= 20 & sup_\mathsf{p} &= 20 \\
sup_\mathsf{d} &= 20 & sup_\mathsf{q} &= 22 \\
sup_\mathsf{e} &= 48 & sup_\mathsf{r} &= 34 \\
sup_\mathsf{f} &= 20 & sup_\mathsf{s} &= 22 \\
sup_\mathsf{g} &= 17 & sup_\mathsf{t} &= 40 \\
sup_\mathsf{h} &= 20 & sup_\mathsf{u} &= 25 \\
sup_\mathsf{i} &= 40 & sup_\mathsf{v} &= 17 \\
sup_\mathsf{j} &= 0 & sup_\mathsf{w} &= 0 \\
sup_\mathsf{k} &= 0 & sup_\mathsf{x} &= 21 \\
sup_\mathsf{l} &= 0 & sup_\mathsf{y} &= 0 \\
sup_\mathsf{m} &= 0 & sup_\mathsf{z} &= 27
\end{aligned}$$

TAB. 2.1 – Résultats des tests sur l'algorithme de résolution du problème 3

Variables fixées	Nombre de possibilités	Nombre de solutions réellement parcourues	Temps	Rapport Temps/Nombre de possibilités
z x v u t r s q p	12×10^8	3434	0,42s	3.5×10^{-10}
z x v u t r s	14×10^{10}	319160	36s	2.5×10^{-10}
a c d e f	62×10^{11}	512666	69s	1.1×10^{-11}
n i e	39×10^{14}	399949	58s	1.5×10^{-14}
i e	51×10^{15}	—	Abandon	—
aucune	12×10^{18}	—	Abandon	—

2.3 Implémentation et résultats

Nous travaillons sur un Athlon 900Mhz avec 512Mo de RAM sous Linux avec un kernel 2.6. Nos sources sont compilées à l'aide de gcc-2.95.

2.3.1 Résultats de l'implémentation actuelle

L'implémentation de la solution aux problèmes 1 et 2 permet de parcourir l'ensemble des solutions en moins d'une minute. Ce résultat est suffisamment satisfaisant.

Pour notre implémentation du problème 3, notre domaine à parcourir après le premier élagage sans aucune instanciation contient 12×10^{18} possibilités. Notre algorithme permet d'élaguer une grosse partie de ces possibilités. Néanmoins, il n'est jamais parvenu à parcourir l'ensemble des solutions dans un temps raisonnable (en une demi heure il était encore très loin d'avoir parcouru tout le domaine).

En fixant les lettres "e", "i" et "n" (qui sont des lettres ayant des grands domaines), notre domaine à parcourir est réduit à 39×10^{14} possibilités. On parcourt l'ensemble de ces possibilités en moins d'une minute. Il est difficile de donner une estimation du temps que prendrait l'algorithme sans fixer ces variables. Effectivement, comme on peut le remarquer dans le tableau 2.3.1, un domaine plus petit au départ ne signifie pas que moins de solutions seront réellement testée. Quoiqu'il en soit il n'est pas satisfaisant.

2.3.2 Optimisations implémentées

On remarque que nos algorithmes ont la propriété d'appeler très souvent les mêmes fonctions avec les mêmes valeur. D'après gprof, la fonction getoccurs() de l'algorithme pour le problème 3 est appelée 65 millions de fois lorsque les lettres "e", "i" et "n" sont fixées. Nous nous sommes donc occupé d'optimiser cette fonction. Nous générons lors de la compilation un tableau statique contenant tous les résultats de cette fonction entre 0 et 1000. Ainsi cette fonction n'est qu'une lecture dans un tableau. On remarque ensuite que la fonction searchchard()[7] est aussi très souvent appelée. Nous avons tenté de mettre les résultats de cette fonction en cache, mais les gains ne sont pas perceptibles.

2.3.3 Optimisations restant à étudier

D'un point de vue plus technique, on remarque que la taille des données en cache est assez importante (jusqu'à 1 Mo). Il serait intéressant d'étudier le nombre d'erreurs de pages lors des accès à ce cache. Lorsque le domaine de définitions des variables se réduit, le nombre de données intéressantes gardées en cache diminue aussi. Il serait peut-être intéressant de reconstruire le cache avec uniquement les données intéressantes de manière à réduire les saut dans la mémoire et donc les erreurs de pages.

[7] qui permet de trouver le nombre, compris entre les arguments min et max, possédant le plus grand nombre d'un caractère donné en paramètre

On remarque que dans notre implémentation nous instancions les variables et les valeurs sans ordre particulier. Pourtant d'après le principe du *first fail*, nous devrions instancier les variables/valeurs entraînant le plus de contraintes en priorité. Pour cela, il suffirait d'instancier la valeur possédant le plus de lettres. Il est difficile d'évaluer le gain de cette optimisation.

On remarque aussi que le parcours de notre domaine serait facilement parallélisable. Une mise en cluster de l'algorithme serait sûrement très intéressante.

Chapitre 3

Implémentations naïves

Afin de résoudre les problèmes, et surtout pour pouvoir mesurer l'efficacité des solutions plus intelligentes, nous avons fait plusieurs implémentations naïves. Une implémentation est considérée comme naïve si elle est facile à mettre en oeuvre et ne fait aucune étude poussée sur le domaine des solutions.

Les deux implémentations qui suivent sont faites pour fonctionner quelque soit le problème. Leur implémentations est faite en `C` pour un soucis de rapidité.

Les deux implémentations ont des algorithmes dont l'allure est proche. On commence par attribuer des valeurs initiales aux coefficients, puis on vérifie que ces valeurs sont cohérentes, si c'est le cas, on a trouvé une solution, sinon on met à jour les coefficients et on itère.

```
// Initialisation
found = 0;
initialiazation = read_init();
init(counts);

// On boucle tant qu'on a pas trouve la solution
while (!found)
{
  // On calcule les coefficients de la phrase.
  new_counts = count_occurances(counts, initialization);

  if (!(found = (counts == new_counts)))
    update_counts(counts, new_counts);
}

// Counts contient notre solution.
return counts;
```

3.1 Brute Force

Dans cet algorithme nous allons essayer de parcourir toutes les solutions du problème. Cette algorithme est clairement voué à l'échec vu le nombre de possibilités.

Le principe est que l'on commence par initialiser tous les coefficients à zéro puis à chaque mise à jour on change la valeur du dernier coefficient en l'incrémentant de 1, si on atteint une borne maximum qu'on se fixe au départ, on incrémente le coefficient précédent, et ainsi de suite.

On initialise les coefficients au décompte des caractères recherchés dans la partie statique, et le maximum est trouvé grâce à une rapide étude mathématique sur la taille de la phrase.

L'algorithme général est légèrement modifié en rajoutant une boucle autour de l'algorithme s'arrêtant uniquement quand tout le domaine à été parcouru.

3.2 Algorithme par correction d'erreur

Dans cet algorithme nous allons calculer une seule solution. L'initialisation de cet algorithme est aléatoire. Il suffit donc de relancer un nombre suffisant de fois le programme pour que la probabilité d'avoir toutes les solutions tende vers 1 (voir la partie résultats pour plus de détails.)

Le principe est qu'à chaque tour on va mettre à jour les coefficients avec une ou plusieurs solutions exactes à un moment donné. Il y a trois variantes de cet algorithme.

First : Lors du *update_counts()* on met à jour la première différence entre les coefficients courants (*counts*) et les coefficients localement exacts (*new_counts*). On remplace la valeur du coefficient courant avec le coefficient localement exact.

One : C'est le même principe que la solution précédente mais on met à jour la n-ième erreur, avec n tiré aléatoirement.

All : Ici on met à jour tous les coefficients courant avec les coefficients localement exacts.

Les implémentations *first* et *all* sont clairement moins intéressantes car le flot du programme est peu variable d'une exécution à l'autre car la seule partie aléatoire est l'initialisation, ce qui fait que bien souvent elles buteront sur les mêmes erreurs.

Dans le problème 2, avec une phrase d'initialisation contenant un 3 et un 2, et une initialisation des coefficients à 0, pour ne pas avoir de hasard dans l'exécution, on obtient un cycle, au bout de quelques itérations avec l'algorithme *first* :

```
...
Loop n 30
count 0 = 7
count 1 = 1
count 2 = 3 // La première erreur est ici.
count 3 = 3
count 4 = 0
count 5 = 0
count 6 = 0
count 7 = 0
count 8 = 0
count 9 = 0

Loop n 31
count 0 = 7
count 1 = 1
count 2 = 2 // On change le 2 en 3.
count 3 = 3
count 4 = 0
count 5 = 0
count 6 = 0
count 7 = 0
count 8 = 0
count 9 = 0

Loop n 32
count 0 = 7
count 1 = 1
count 2 = 3 // On a rechange le 2 en 3.
count 3 = 3
count 4 = 0
count 5 = 0
count 6 = 0
count 7 = 0
count 8 = 0
count 9 = 0
...
```

De même avec l'exécution *all* on a :

```
...
Loop n 12718
count 0 = 1
count 1 = 7
count 2 = 3
count 3 = 2
count 4 = 3
count 5 = 1
count 6 = 2
count 7 = 1
count 8 = 1
count 9 = 1

Loop n 12719
count 0 = 1
count 1 = 6
count 2 = 4
count 3 = 4
count 4 = 1
count 5 = 1
count 6 = 1
count 7 = 2
count 8 = 1
count 9 = 1

Loop n 12720
count 0 = 1
count 1 = 7
count 2 = 3
count 3 = 2
count 4 = 3
count 5 = 1
count 6 = 2
count 7 = 1
count 8 = 1
count 9 = 1
...
```

Par contre avec l'option *one* ces boucles apparaissent beaucoup moins fréquemment.

3.3 Détails d'implémentation

Les coefficients sont stockés dans des tableaux d'entiers, de la taille du nombre de nombre recherchés, le tableau fait donc 10 pour les problèmes 1 et 2 et 26 pour le problème 3. On accède aux valeur à partir de la valeur ASCII des caractères avec des simples opérations, pour accéder aux nombres on fait `c - '0'` et pour les lettres `c - 'a'`. Les tableaux sont dynamiquement alloués au début du programme car l'implémentation est générique pour tous les problèmes. On utilise les mêmes tableaux tout au long de l'exécution.

Pour compter les occurrences de chaque lignes on construit la chaîne de caractères représentant la phrase. Puis on compte le nombre d'occurrences de toutes les lettre du code ASCII et on garde uniquement les lettres qui nous intéressent.

Pour vérifier que les cofficients sont bons on fait juste une comparaison de deux tableaux (l'équivalent d'un *strcmp*).

3.4 Attentes

Dans les problème 1 et 2, on travaille sur les chiffres et il est intéressant de noter qu'il y a peu de changements des coefficients quand ont change l'un d'entre eux. En effet un coefficient est généralement sur un ou deux chiffres et donc il n'y a qu'un ou deux coefficients qui changent.

On s'attend donc à ce que ces algorithmes fonctionnent bien. Avec une initialisation des coefficients constante et une phrase initiale constante, l'algorithme *all* devrait soit trouver une solution en un temps constant, soit boucler infiniement.

Une autre choses qu'on aimerait voir est l'influence de l'initialisation aléatoire sur les alogrithmes. Celle ci devrait diminuer le taux de boucles infinies.

On s'attends aussi à ce que l'algorithme *one* obtienne les meilleurs resultats car c'est celui où le hasard rentre le plus en jeu.

L'algorithme *first* ne devrait pas avoir de bon résultats il a de très grandes chances de boucler indéfiniement.

En général on aimerait savoir le taux de boucles infinies, c'est à dire le nombre de phrases initiales qui feront aboutir les algorithmes.

Par contre pour le problème 3, on n'espère pas obtenir de solutions dans un temps raisonable et fréquemment car la variation des coefficients est trop importante.

3.5 Procédure de test

Les tests ont été effectués sur une machine 1,2ghz P4 Mobile, avec 512MB de Ram sous Linux 2.4.22.

Pour chaque test nous exécutons 1000 fois le programme et calculons les nombre moyen d'itérations et de temps. Le temps est mesuré grâce à la commande *time* (*GNU time 1.7*).

Si l'exécution dure trop longtemps elle est tuée, on dis alors qu'elle a fait un *timeout*. Les moyennes sont faites sur toutes les exécutions ayant terminé. La temps d'attente est determiné à l'avance humainement en maximisant le temps d'une exécution. On peut dire qu'un *timeout* signifie une boucle infinie.

Un temps d'exécution de 0 signifie qu'il est en dessous du dixième de seconde.

L'initialisation peut être de deux types, soit tous les coefficients sont initialisés à zéro, soit ils sont initialisés aléatoirement avec un nombre compris entre 0 et 21.

Chaque test est fait avec chacun des algorithmes naïfs *one*, *all*, *first*, et chacune des initialisations possibles.

Le signe *N/A* veut dire que la valeur est indeterminée ce qui peut arriver quand toutes les exécutions font des timeouts et du coup on ne peut pas calculer les valeurs moyennes.

Dans le cas du problème 2, la phrase initiale est générée aléatoirement, sa taille est un nombre aléatoire entre 1 et 20 et son contenu est une suite de chiffres aléatoires.

3.6 Résultats

Comme on s'y attendait on voit que l'algorithme *all* trouve la solution en un temps constant avec l'initialisation à zéro. Ce qui est normal car il n'y a aucun hasard dans cette méthode. Par contre dès qu'un peu de hasard rentre en jeu, que ce soit pour la phrase initiale que pour l'initialisation des coefficients, il boucle infiniement souvent. Ce qui veut dire que les l'espaces de départ où l'algorithme termine sont très restreints. Néanmoins quand on est sur une solution qui aboutie il est très rapide. Il serait intéressant de pouvoir calculer le genre de solution qui fonctionne avec cet algorithme.

Les méthodes *all* et *one* sont très rapides à trouver la solution mais ne la trouvent pas souvent. Néanmoins on voit qu'une initialisation aléatoire les aide, on pourrait imaginer essayer plusieurs initialisations des coefficients aléatoires différentes pour un même problème afin d'avoir plus de chances de trouver une solution. On voit que l'algorithme *one* est tout de même celui qui offre le meilleur taux de reussite.

On constate aussi que l'algorithme *first* boucle infiniement à chaque fois, même avec une initialisation aléatoire. Ce qui veut dire que les cas où il marche sont vraiment très peu nombreux voire inexistants. Même une initialisation aléatoire ne l'aide pas.

Nous avons dit qu'en exécutant le programme plusieurs fois on obtient toutes les solutions. Nous avons donc testé sur le problème 1 dont nous savons qu'il a deux solutions :

TAB. 3.1 – Résultats des tests sur les algorithmes naïfs.

Problème	Algorithme	Initialisation	Nb d'exécutions	Nb de *timeouts*	Itérations moyennes	Temps moyen (en sec)
1	*One*	zéro	1000	7	570	2
1	*All*	zéro	1000	0	5 (constant)	0 (constant)
1	*First*	zéro	1000	1000	N/A	N/A
1	*One*	random	1000	24	560	1
1	*All*	random	1000	415	6	0
1	*First*	random	1000	1000	N/A	N/A
2	*One*	zéro	1000	659	708	0.5
2	*All*	zéro	1000	719	9	0
2	*First*	zéro	1000	1000	N/A	N/A
2	*One*	random	1000	580	732	0.5
2	*All*	random	1000	699	8	0
2	*First*	random	1000	1000	N/A	N/A

TAB. 3.2 – Frequences d'appartions des solutions avec les algorithmes naïfs.

Algorithme	Initialisation	Fréquence de la solution 1	Fréquence de la solution 2
One	zéro	100	0
All	zéro	100	0
One	random	99.5%	0.5%
All	random	100	0

```
Dans cette phrase il y a: 1 0, 7 1, 3 2, 2 3, 1 4, 1 5, 1 6, 2 7, 1 8, 1 9.
Dans cette phrase il y a: 1 0, 11 1, 2 2, 1 3, 1 4, 1 5, 1 6, 1 7, 1 8, 1 9.
```

Avec la même procédure de tests voici les fréquences d'apparition de chacune des solutions que nous obtenons :

On voit donc qu'il faut un maximum de hasard pour que la deuxième solution apparaisse car le seul algorithme permettant d'obtenir la deuxième solution est *one* avec une initialisation aléatoire des coefficients. Ce qui est l'algorithme avec le plus de hasard.

Chapitre 4

Recuit simulé

4.1 Principe de l'algorithme de recuit simulé

Le recuit simulé se base sur un principe d'énergie. Au début de l'algorithme l'énergie est forte donc il y a beaucoup de changements, même vers des directions qui ne sont pas à priori bénéfiques. Plus le temps passe, plus l'énergie est faible et donc moins on a de chances d'aller vers une direction erronée.

Ce principe permet au début de parcourir des intervalles qui auraient pu ne jamais être parcourus par un algorithme qui ne fait que converger. On espère ainsi pouvoir sortir d'un minimum local pour trouver un minimum global.

Si on l'applique à notre problème on obtient l'algorithme suivant :

```
while (temperature >= low)
  {
    /* Build a new solution */
    memcpy(new_counts, counts, sizeof(counts));
    modify(&new_counts, searched_len, flag);
    /* Count occurences. */
    build_string(str, init, searched, new_counts, flag);
    counts_ok = count_occurances(str, searched, flag);
    /* Get the new energy and compare with old one */
    new_energy = get_energie(new_counts, counts_ok, searched_len);
    diff = energy - new_energy;
    /* Save solution if better or with a probability */
    if (diff > 0 || probability(diff, temperature))
    {
      memcpy(counts, new_counts, sizeof(counts));
      energy = new_energy;
    }
    temperature *= modif;
  }
```

Dans cet algorithme on va itérer jusqu'à ce que la température soit trop faible pour avoir une quelconque influence sur le reste.

A chaque itération on commence par générer une nouvelle solution à partir de la solution courante en appliquant des transformations. Puis on va calculer l'énergie de la nouvelle solution et on la compare avec l'énergie de la solution précédente. Si la solution est meilleure, ou alors, avec une certaine probabilité dépendante de la température actuelle et de l'écart d'énergie avec la solution précédente, on garde la nouvelle solution. Enfin on baisse la temperature. Bien évidemment on sort quand on a trouvé une solution.

La probabilité de garder une mauvaise solution est un tirage selon la loi de probabilité :

$$P = exp(-diff/temprature) \tag{4.1}$$

Avec $diff$ la différence d'énergie entre la solution courante et la solution précédente. Elle est toujours positive, car si elle était négative on aurait gardé la solution car celle ci aurait été meilleure que la solution actuelle.

A la fin de l'exécution il se peut que la solution courante ne soit pas la bonne. Il faut donc relancer le programme plusieurs fois pour obtenir une solution. De même le programme ne trouve qu'une seule solution. Là aussi il faut lancer le programme plusieurs fois pour obtenir toutes les solutions. Cela est vrai car les tranformations ainsi que l'initialisation sont aléatoires.

L'initialisation des coefficients, c'est-à-dire la première solution, est aléatoire.

4.2 Choix des transformation

Pour les transformations effectuées à chaque itération pour générer une nouvelle solution nous avons pensé tout d'abord à mettre la valeur exacte courante. Mais cela revient à peu de choses près aux algorithmes naïfs vu précédement.

La solution qui a été choisit est d'appliquer des modificateurs aux coefficients, par exemple ajouter un $+2$ au coefficients du chiffre 4.

Mais se contenter d'une seule modification par itération fait converger l'algorithme très lentement, pour accélérer le processus nous avons augmenté le nombre de modifications, au lieu d'en faire une par itération nous en effectuons n.

La chose la plus naturelle est de poser un nombre n constant. Mais le problème est qu'au début on veut faire de grosses modifications pour aller rapidement près d'une solution puis une fois proche de la solution on veut tatonner pour trouver la solution exacte car une solution approchée ne nous convient pas. On a donc choisit d'appliquer un nombre de transformations proportionnel à la température, dans notre cas c'est le nombre de digits que contient la température.

Cette méthode nous permet de converger rapidement et de trouver fréquement des solutions.

4.3 Mesure de l'énergie

La mesure de l'énergie est importante car elle nous apprend si la solution courante est proche ou pas de la solution recherchée. La aussi il y a plusieurs façons de mesurer l'énergie. Il y a au moins deux méthodes qui nous semblent significatives.

La première est le nombre de coefficients faux. Un coefficient est faux quand le décompte du caractère qu'il décrit ne correspond pas. On ne tient pas compte de combien il diffère. La deuxième façon de calculer l'énergie est de faire la somme des différence entre les coefficients faux, et les coefficients attendus. Dans ce cas on tient compte de l'écart.

Par exemple pour le début de phrase suivante : `1 0 1 1` avec la première méthode on obtient 1 comme energie car il y a un coefficient érroné, tandis qu'avec la deuxième méthode on obtient 2 car il y a un coefficient érroné qui vaut 1 alors que le coéfficiant attendu est 3 ce qui fait $3-2=1$.

C'est la deuxième solution qui a été retenue car la première ne converge pas assez vite. En effet l'énérgie ne varie pas assez, elle est comprise entre 0 et le nombre de caractères que l'on compte. De plus il est très facile d'avoir deux énergie pareille même si les solutions sont complètements differentes.

Cette energie est mise au carré pour être normalisée car c'est une distance.

TAB. 4.1 – Résultats des tests sur l'algorithme de recuit.

Pb	Exécutions	Modif due à proba	Exec avec sol	MAX	MIN	MODIF	Ités max	Ités moyenne	Tps max	Tps moyen
1	1000	1689	511	10	0.1	0.9999	46051	27506	0.5s	0.3s
1	1000	961	493	2	0.1	0.9999	29957	11474	0.31s	0.12s
1	1000	781	502	1	0.1	0.9999	23026	4601	0.23s	0.04s
1	1000	580	208	0.5	0.1	0.9999	16095	2319	0.16s	0.02s
1	1000	1320	481	2	0.01	0.9999	52982	11620	0.59s	0.12s
1	1000	1202	458	1	0.01	0.9999	46051	4962	0.47s	0.04s
1	1000	350	66	2	0.001	0.999	7599	1410	0.08s	0.01s
1	1000	1157	654	1	0.01	0.9999	46051	6189	0.46s	0.06s
2	1000	530	379	1	0.1	0.9999	23026	5106	0.23s	0.05s

TAB. 4.2 – Fréquences d'apparitions des solutions avec le recuit simulé.

MAX	MIN	MODIF	Fréquence de la solution 1	Fréquence de la solution 2
10	0.1	0.9999	49%	51%
2	0.1	0.9999	46%	54%
1	0.1	0.9999	41%	59%
0.5	0.1	0.9999	40%	60%
2	0.01	0.9999	48%	52%
1	0.01	0.9999	46%	54%
2	0.001	0.999	33%	67%
1	0.01	0.9999	27%	73%

4.4 Procédure de test

Les tests ont été fait avec plusieurs constantes du recuit simulé : MAX qui est la température de départ, MIN qui est la température minimale à laquelle on arrête et MODIF qui est le coefficient pour lequel on baisse la température à chaque itération : `temp *= MODIF`.

Les tests ont été effectués sur une machine 1,2ghz P4 Mobile, avec 512MB de Ram sous Linux 2.4.22. Le temps d'éxécution est mesuré grâce à la commande *time*.

Le programme est lancé un certain nombre de fois et l'on va s'intéresser à combien d'itérations il fait, dans le cas où il ne trouve pas la solution le nombre est détérminé par les constantes liées à la température. On s'intéressera aussi au taux de bonnes réponses car comme on l'a vu le programme peut s'arréter sans trouver de réponse. Nous regarderons aussi s'il trouve les solutions possibles au fur et à mesure des exécutions et quelle est leur fréquence d'apparition.

Le programme est testé avec le premier et le deuxième problème uniquement.

4.5 Résultats

La colonne *itérations max* indique le nombre d'itérations éffectuées quand le programme ne trouve pas de solution, de même pour TEMPS MAX. Les moyennes d'itérations et de temps sont faites sur les exécutions trouvant une solution.

D'après les résultats on voit que l'algorithme de recuit simulé marche bien pour le problème 1 (et donc le

2), il trouve une solution à peu près une fois sur deux, et trouve chaque solution avec la même probabilité.

Si on reduit la probabilité de garder une mauvaise solution en jouant sur les constantes, on remarque qu'on augmente la fréquence d'apparition d'une même réponse. En effet apparaîtra la solution la plus proche de l'initialisation.

Le dernier test est effectué avec un calcul d'énergie non normalisé (la distance n'est pas mise au carré). On voit que le taux d'execution aboutissant sur une bonne réponse est très fort mais on arrive toujours sur la même réponse. C'est sûrement un bon calcul d'énergie si l'on veux juste trouver une solution mais pas toutes.

Au final on peut dire qu'en moins de 10 executions, et donc en moins de deux seconde on peut trouver toutes les solutions.

Chapitre 5

Algorithme Génétique

5.1 Principe d'un algorithme génétique

Les algorithmes génétiques se sont inspirés du fonctionnement de l'évolution des espèces.

L'algorithme est basé sur la manipulation d'une population, c'est à dire d'un ensemble d'individus. Chaque individu est caractérisé par un génome et représente une solution au problème.

Le génome est un vecteur de gènes. La taille de ce vecteur et la forme des gènes sont dépendantes du problème. Par exemple, pour le problème du voyageur de commerce, chaque gène représenterait une ville à visiter, et la taille du génome serait le nombre de villes à visiter.

On doit aussi avoir une fonction appelée 'fitness', qui est fonction du génome, et que l'algorithme cherchera à maximiser. La plupart des problèmes sont à la base des problèmes de minimisation de fonctions (fonctions d'erreur très souvent), mais il est très simple de transformer la fonction à minimiser pour en faire une fonction fitness ($\frac{1}{f(x)}$, par exemple).

A l'initialisation de l'algorithme, on crée N individus pour former la population. Ces individus peuvent être générés au hasard ou selon certaines heuristiques, selon le problème, afin de faciliter la convergence.

Ensuite, chaque génération (itération de l'algorithme) comporte deux phases : la diversification et la sélection.

5.1.1 Diversification

Durant cette phase, l'algorithme crée de nouveaux individus par 'cross-over' ou par 'mutation'. La taille de la population augmente de M, le nombre d'individus créés.

Le 'cross-over' (croisement) prend deux individus et les mélange pour en fabriquer deux nouveaux. Typiquement, il utilise un ou deux pivots sur les génomes des parents, et fabrique les enfants en alternant les génomes des parents.

La 'mutation' fabrique un enfant à partir d'un parent. Elle est indispensable à la convergence de l'algorithme, sans elle la plupart du temps, on ne peut pas couvrir tout le domaine des solutions.

En général, elle se contente de copier chez l'enfant le génome du parent en en altérant un gène.

La mutation est appliquée en général avec une probabilité moindre que celle du cross-over.

5.1.2 sélection

Après la phase de diversification, la sélection a pour but de réduire la taille de la population pour qu'elle redevienne la même qu'au début de l'algorithme. Pour ça on choisi un a un N individus parmi les N+M individus disponibles, sans remise et avec une probabilité fonction du 'fitness' de l'individu : plus l'individu était bon, meilleures sont ses chances d'être conservé.

Deux mécanismes spéciaux sont couramment utilisés durant la phase de sélection : l'élitisme oblige le meilleur individu de la population à être sélectionné pour passer à la génération suivante, et la diversification empêche que trop d'individus soient identiques au sein d'une même population.

5.1.3 Récapitulatif des paramètres

Pour chaque problème que l'on voudra résoudre à l'aide d'un algorithme génétique, on aura donc besoin de donner les paramètres suivants :

- Taille du génome, type des gènes.
- Fonction de fitness. (ou d'erreur, et la transformer)
- Nombre maximal de génération.
- N : Taille initiale de la population.
- M : Nombre d'individus générés à chaque génération.
- Probabilité d'effectuer une mutation plutôt qu'un cross-over.
- Utilisation de l'élitisme (vrai ou faux).
- Utilisation de la diversification (vrai ou faux).
- On pourra aussi vouloir redéfinir les opérateurs de mutation et de cross-over.

5.2 Implémentation

Notre algorithme génétique est codé en java. Un individu (Biomorph en anglais) est représenté par une classe abstraite. Pour coder notre problème particulier, nous avons donc eu à dériver cette classe.

5.2.1 Initialisation de la population

Les premiers individus créés par l'algorithme ont un génome généré au hasard. Quelques règles ont cependant été mises en places : la valeur d'un gène est supérieure ou égale à la valeur de l'initialisation, et durant tout l'algorithme on ne touche pas aux lettres qui n'apparaissent pas dans les nombres et sont donc bonnes dès le départ (par exemple 'b', 'j', 'k', ...).

5.2.2 Choix des paramètres

Pour paramétrer l'algorithme, nous avons fait les choix suivants :

- Le génome représente une solution : dans notre cas c'est donc un vecteur d'entier. Pour les deux premiers problèmes (nombres), sa taille est de dix (nombre de chiffres en décimal), et dans le dernier problème (lettres) vingt-six (nombre de lettres dans l'alphabet).
- La fonction de fitness est fabriquée 'artificiellement' à partir de la fonction d'erreur (cf plus bas).
- La taille de la population et le nombre d'individus générés à chaque itérations sont fixés par l'utilisateur au lancement de l'algorithme, ainsi que la probabilité des mutations. Cela permet de faire plus facilement des tests pour ces paramètres.
- Le nombre maximal de génération peut être fixé par l'utilisateur. Sinon il peut choisir de faire tourner l'algorithme jusqu'à ce qu'il trouve une solution optimale.

5.2.3 Fonction d'erreur

La fonction d'erreur que nous avons choisi est la somme des carrés des erreurs de chaque gène (chaque chiffre ou chaque lettre selon le problème) :

$$erreur(\text{génome}) = \sum_{i=1}^{\text{génome.taille}} (\text{génome}[i] - \text{réelles}[i])^2)$$

Avec *relle* un vecteur de même taille que le génome et qui contient le nombre réel de chaque chiffre (ou lettre) de la phrase représentée par l'individu.

Cette fonction d'erreur est bien sûr à minimiser. Notre fonction fitness vaut donc :

$$fitness(gnome) = \frac{1}{erreur(gnome)+1}$$

Le '$+1$' étant là uniquement pour éviter une division par zéro quand la solution est optimale ($erreur = 0$).

5.2.4 Opérations sur le génomes

Nous n'avons pas touché sur ce problème à l'opérateur de cross-over. Il effectue l'opération basique en utilisant un pivot.

Pour la mutation, nous altérons un gène au hasard en lui ajoutant ou retirant un nombre aléatoire entre 1 et une constante. (Cette constante peut valoir 1...). Cette opération est effectuée évidement dans la limite du domaine du problème (par exemple, pas de nombres négatifs). Pour plus de détails sur ces limites, voir la première partie de ce rapport.

5.3 Résultats - Performances

L'utilisation de l'algorithme génétique donne de très bonnes performances sur le problème des chiffres, avec ou sans initialisation. Il trouve en général une solution au bout de cent ou deux cents générations. Par contre, il n'a jamais réussi a trouver de solution au problème des lettres. Son meilleur résultat avait une erreur de 1.

5.3.1 Résultats en chiffes

Nous avons mis ici les résultats de l'algorithme avec différents paramètres (taille de la population, diversification, ..). Pour les chiffres et les lettres.

Pour chaque graphique, les 10 courbes représentent les 10 résultats différents qu'on a obtenu en lançant 10 fois l'algorithme avec les mêmes paramètres.

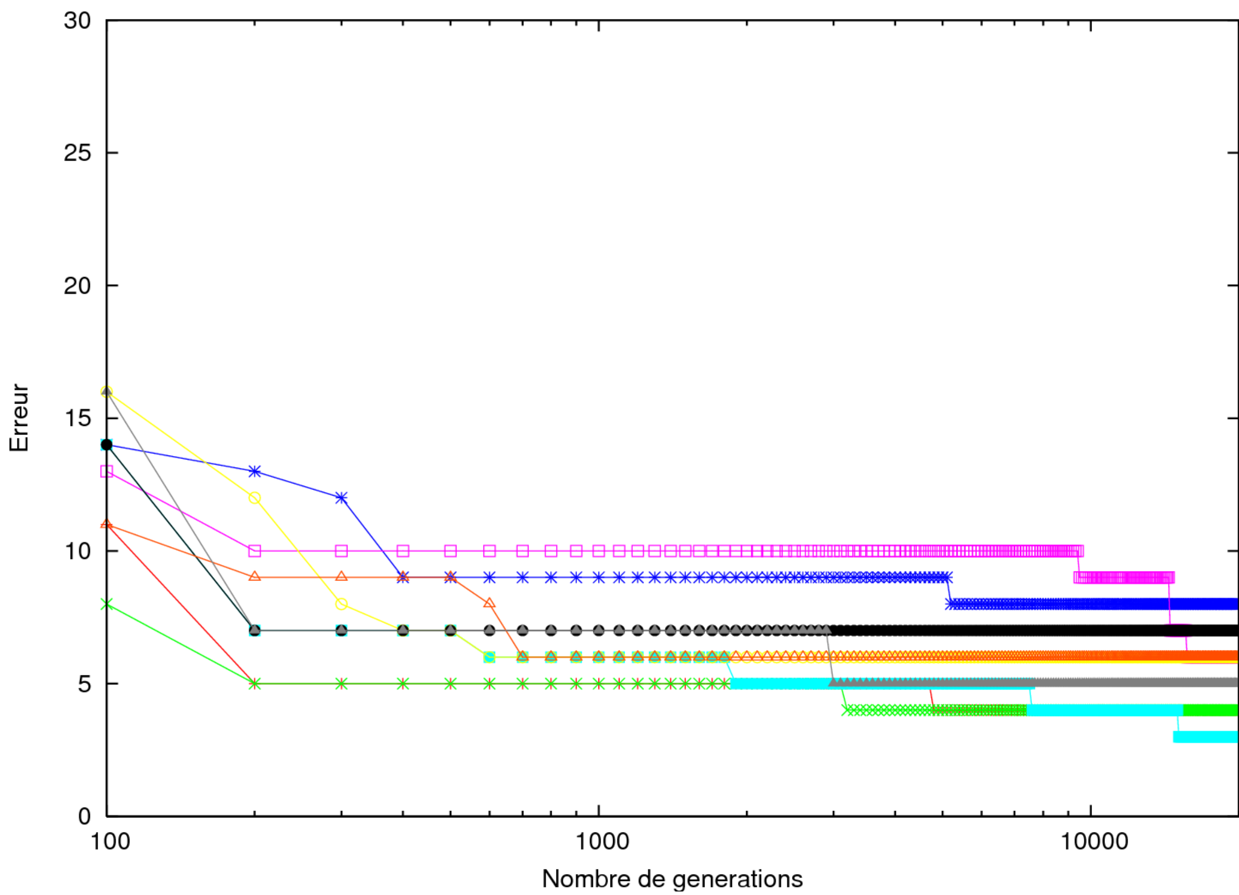
L'algorithme étant basé sur le hasard, plusieurs instances du même problème ne trouvent pas forcément la même solution.

Problème des lettres

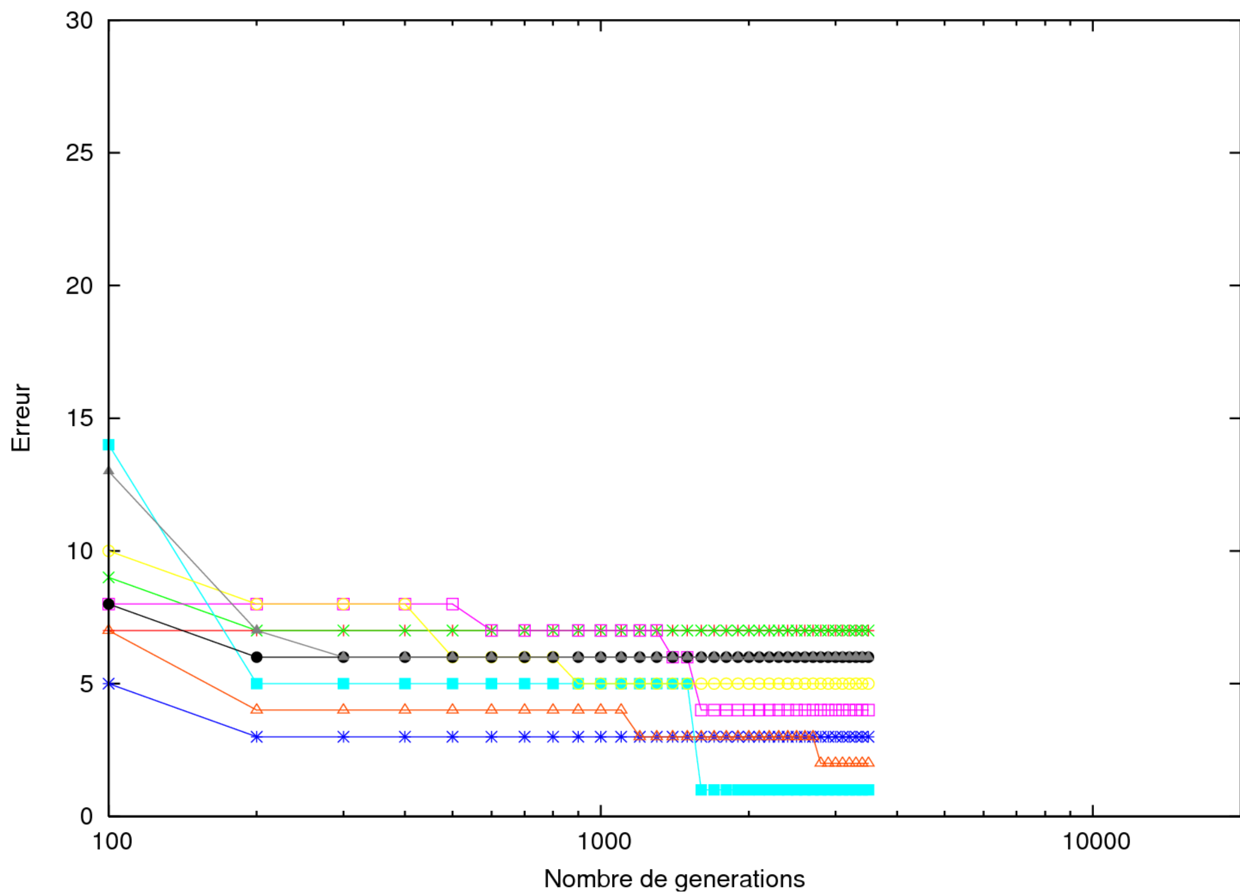
Nombre de générations	20000
Taille de la population	300
Nombre d'individus créés par générations	300
Probabilité de la mutation	0.2
Elitisme	Non
Diversification	Oui

30
25
20
15
10
5
0
Erreur
100
1000
10000
Nombre de generations

Nombre de générations	20000
Taille de la population	300
Nombre d'individus créés par générations	300
Probabilité de la mutation	0.4
Elitisme	Non
Diversification	Oui



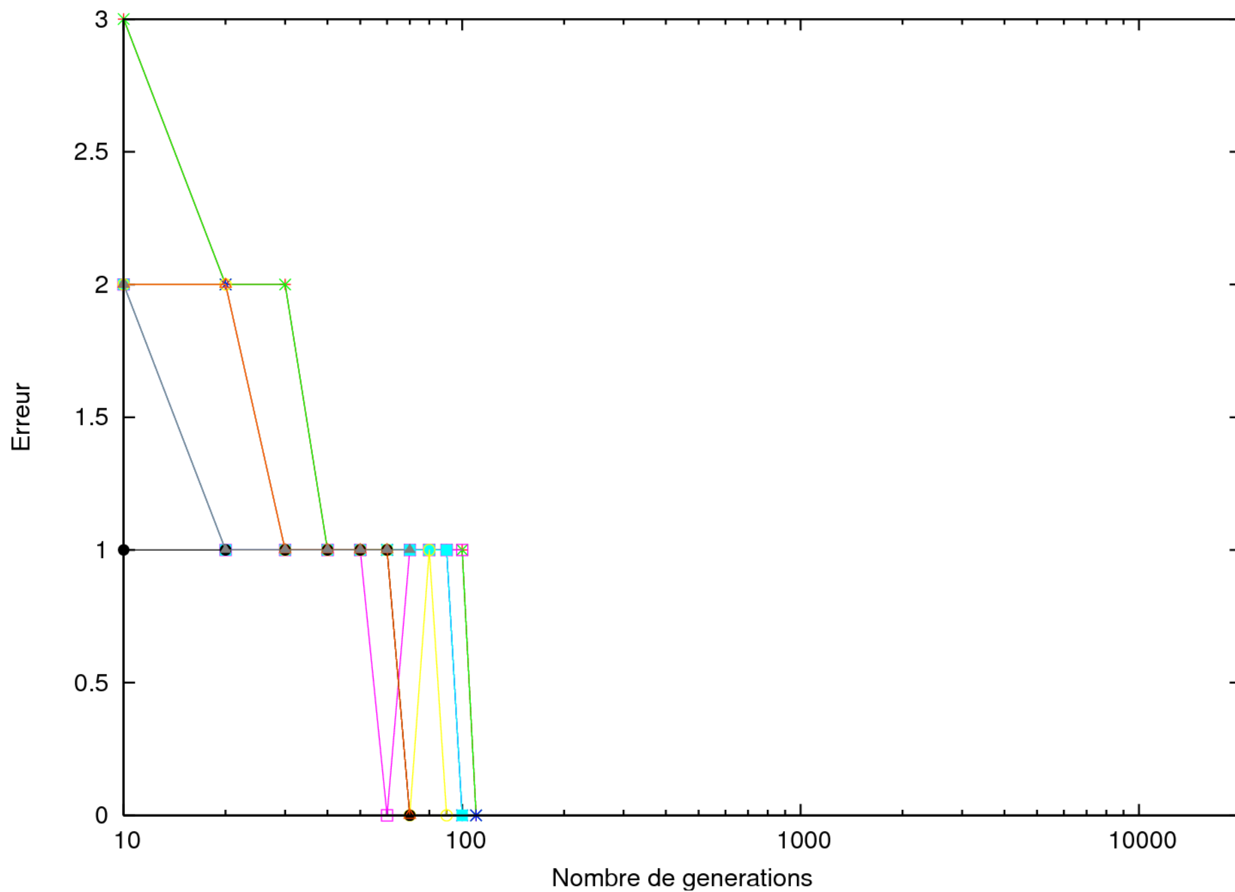
Nombre de générations	3500
Taille de la population	1000
Nombre d'individus créés par générations	1000
Probabilité de la mutation	0.3
Elitisme	Non
Diversification	Oui



Problèmes des chiffres

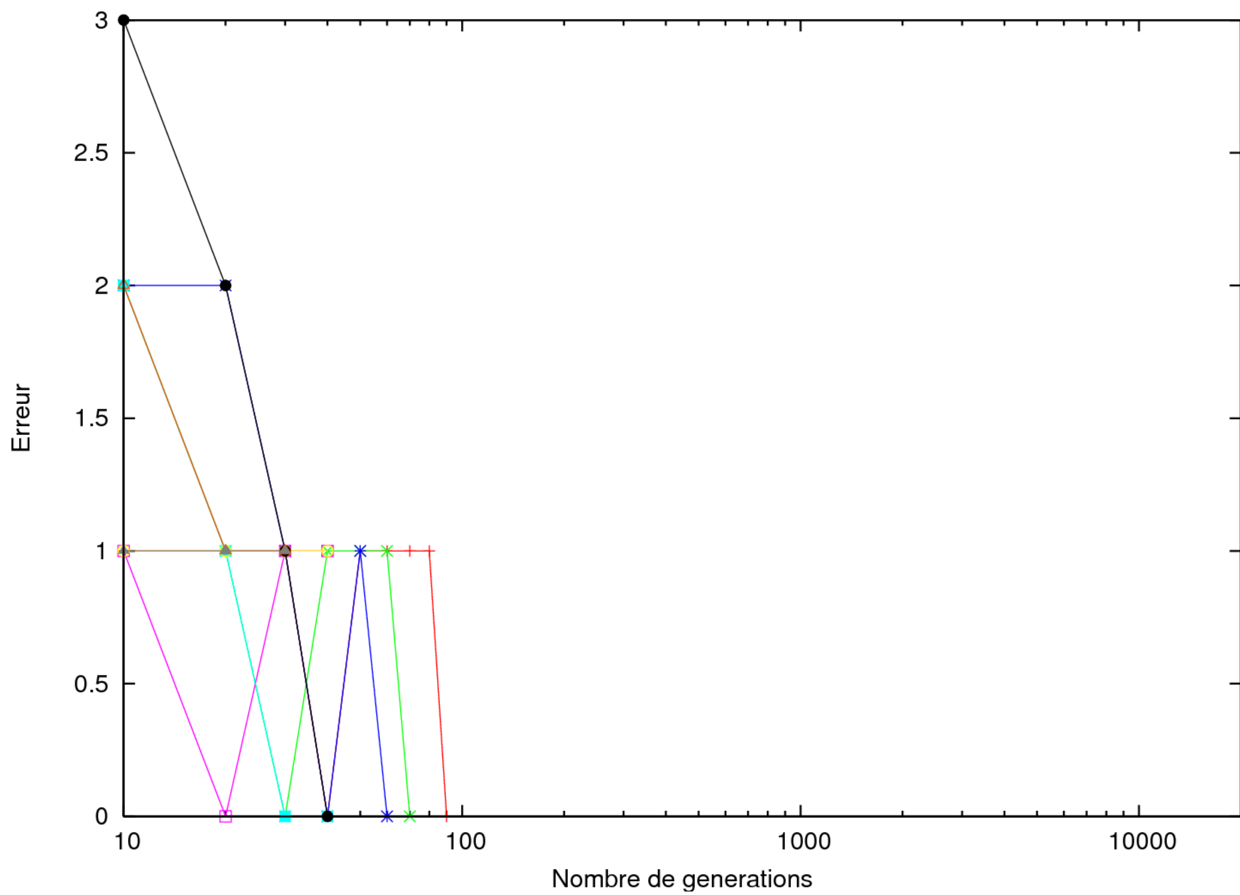
Ces graphismes ont été générés pour le problème 1, c'est a dire sans initialisation. Les résultats pour le problème 2 - avec initialisation - sont sensiblement identiques, nous ne les avons donc pas reproduit ici.

Nombre de générations	indéfini
Taille de la population	300
Nombre d'individus créés par générations	300
Probabilité de la mutation	0.2
Elitisme	Non
Diversification	Oui

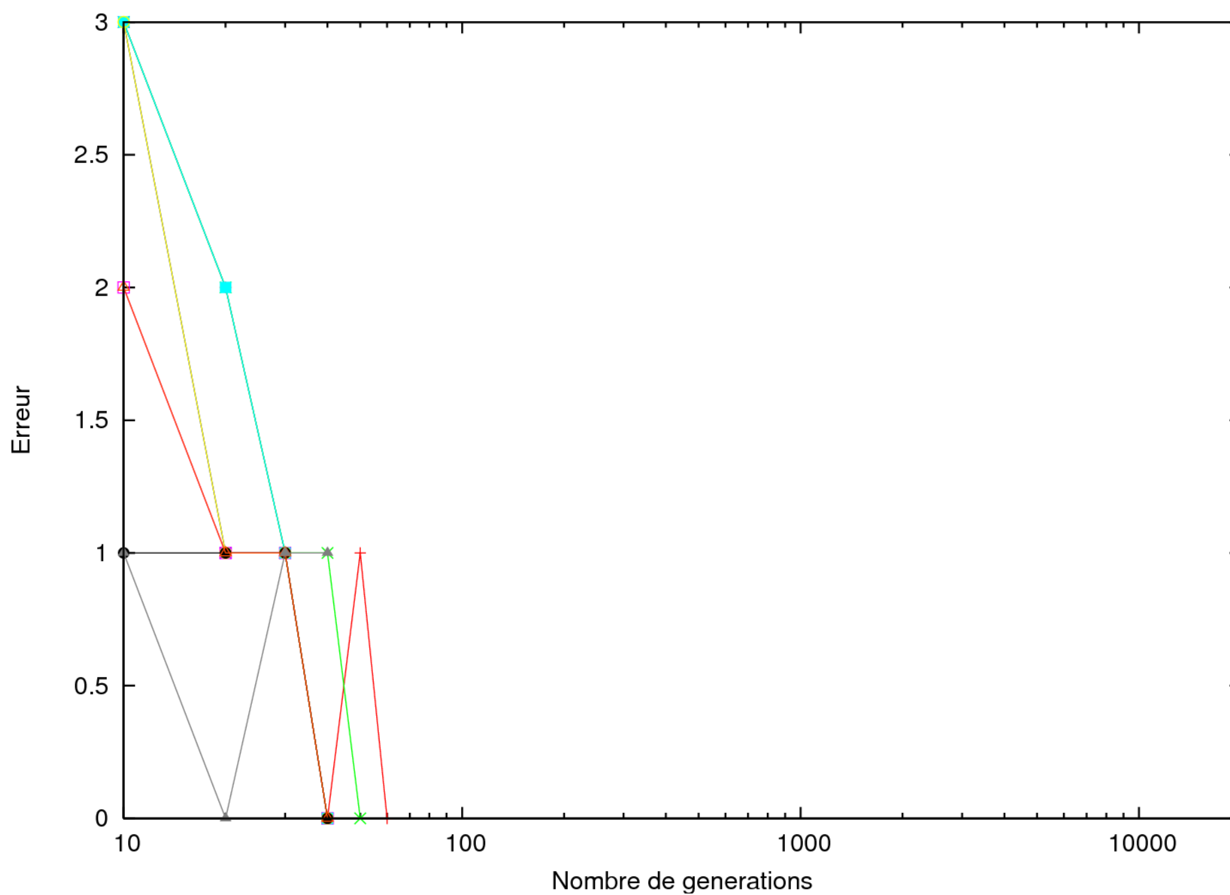


Nombre de générations	indéfini
Taille de la population	300
Nombre d'individus créés par générations	300
Probabilité de la mutation	0.4
Elitisme	Non
Diversification	Oui

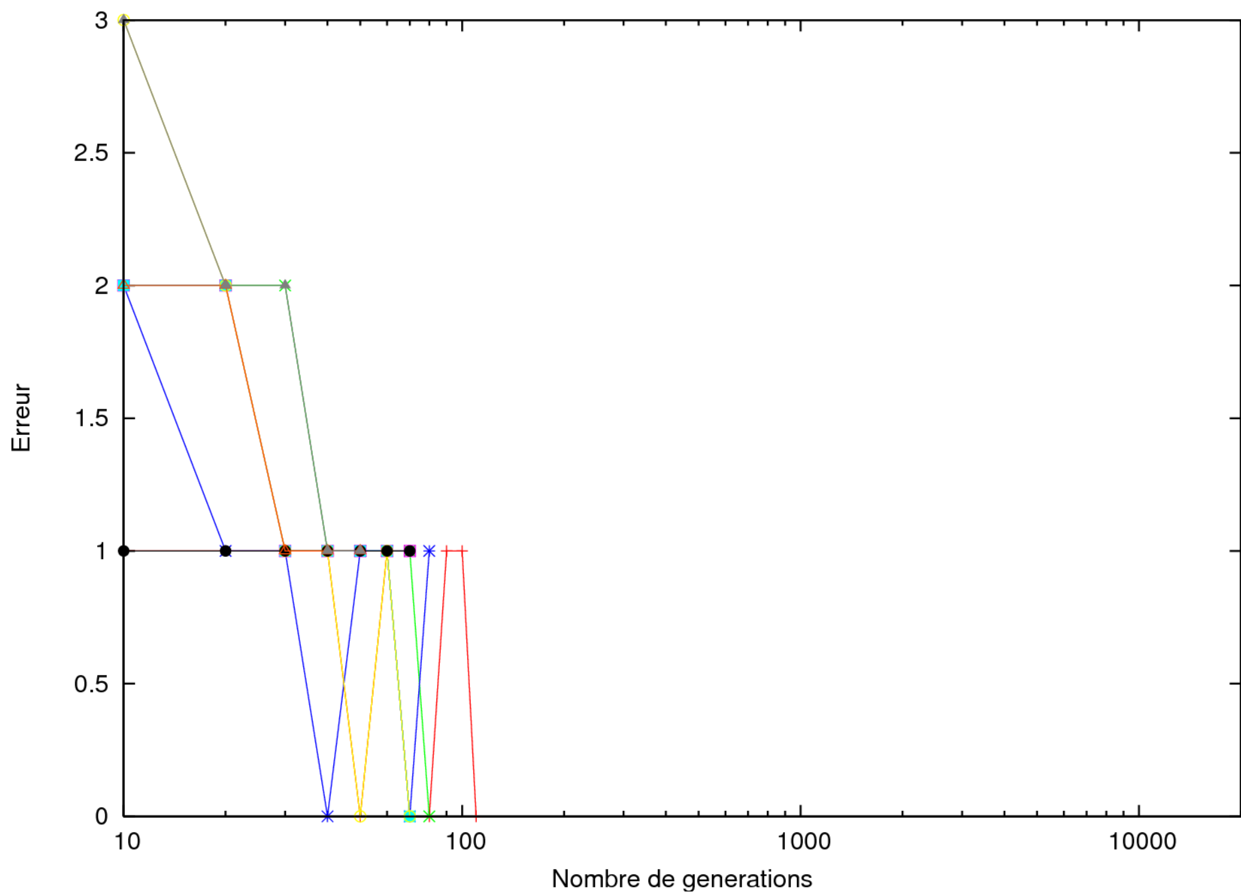
Nombre de générations	indéfini
Taille de la population	300
Nombre d'individus créés par générations	300
Probabilité de la mutation	0.6
Elitisme	Non
Diversification	Oui



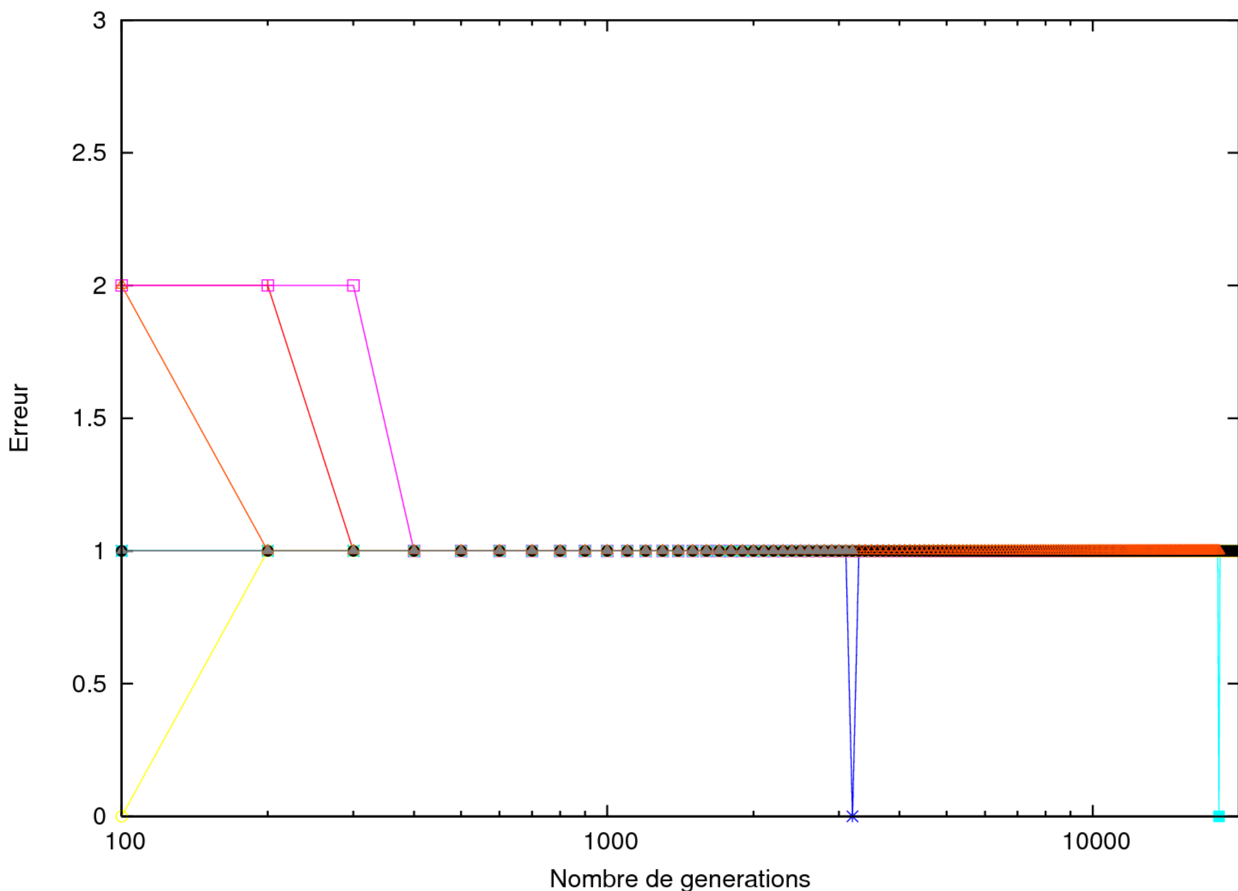
Nombre de générations	indéfini
Taille de la population	300
Nombre d'individus créés par générations	300
Probabilité de la mutation	0.8
Elitisme	Non
Diversification	Oui



Nombre de générations	indéfini
Taille de la population	300
Nombre d'individus créés par générations	300
Probabilité de la mutation	0.4
Elitisme	Oui
Diversification	Oui



Nombre de générations	20000
Taille de la population	300
Nombre d'individus créés par générations	300
Probabilité de la mutation	0.4
Elitisme	Non
Diversification	Non



5.3.2 Conclusions

A propos des parametres

Importance de la diversification. La donnée la plus critique, au vu des résultats ci-dessus est l'utilisation du mécanisme de diversification. On voit bien sur l'exemple des chiffres qu'en l'utilisant, quelques soient les autres paramètres, on obtient un résultat en moins de 100 ou 200 générations, tandisqu'en ne l'utilisant pas, on reste souvent a une erreur de 1 au bout de 20000 générations !

Ce phénomène est assez simple : au bout d'un moment, l'algorithme refait les mêmes croisements ou mutations sur les mêmes individus, et on retombe sur des individus qu'on connait, ce qui amplifie le fait de 'tomber' dans des minima locaux. Au bout d'un moment, une grande partie de la population aura une erreur de 1, mais tout le monde sera dans ce minimum local, et l'algorithme aura beaucoup de mal a s'en sortir... S'il s'en sort.

Mécanisme d'élitisme. Comme on l'a vu, le but de l'élitisme est de 'forcer' le meilleur individu d'une génération à appartenir à la suivante. Ici, il ne change pas grand chose : comme on le voit sur les graphiques,

l'erreur ne régresse jamais, donc quelque part, si le meilleur individu n'a pas été gardé c'est qu'un autre au moins équivalent a été créé.

A propos de l'algorithme

Le problème des chiffres avec ou sans initialisation a été très bien résolu par l'algorithme génétique, à condition d'utiliser le mécanisme de diversification. La convergence est très rapide.

Le problème des lettres par contre, est nettement plus difficile. L'algorithme a tendance à plonger dans un minimum local et à y rester. Ce phénomène est du en partie au fait que lorsqu'on se trouve avec une solution ayant une erreur de 1, il ne suffit pas de changer une valeur pour tomber sur la bonne solution (ce qui est possible avec le problème des chiffres). Changer une valeur implique augmenter l'erreur dans la plupart des cas. Il est donc très dur de sortir de ces minima.

Ce problème pourrait peut être être réglé, ou du moins amélioré par une meilleure initialisation des premiers génomes pour aider l'algorithme à partir dans la bonne direction, mais nous n'en avons pas trouvé.

Chapitre 6

Utilisation de l'algorithme de Grover

Nous avons essayé dans cette partie d'utiliser l'algorithme de Grover pour résoudre notre problème. Nous rappelons que l'algorithme de Grover ne fonctionne que sur des ordinateurs quantiques

6.1 Rappels sur les ordinateurs quantique

Les ordinateurs habituels utilisent des états électriques pour symboliser les bits 0 et 1. Dans un ordinateur quantique, on symbolise le 0 et le 1 par deux états d'une particule. L'intérêt arrive avec les équations de Schrödinger qui démontrent qu'au niveau atomique, une particule peut se trouver dans deux états/endroits à la fois avec une certaine probabilité. La particule se stabilise dans un état à partir du moment ou l'on essaye de le mesurer. Par conséquent, en travaillant sur ces particules, on peut travailler avec plusieurs états en même temps. La seule contrainte est que lors de la lecture de l'état de la particule, celle-ci prendra automatiquement une valeur et les autres valeurs seront perdues.

On associant plusieurs qbits, on obtient des registres quantiques. Un registre quantique de n qbit peut prendre 2^n valeurs avec une certaine probabilité.

A chacune des valeurs possibles d'un registre quantique est associé un coefficient complexe appelé ampere. Ce coefficient est la représentation mathématique de certaines propriétés subatomiques dépassant le cadre de ce rapport. La probabilité qu'une valeur soit mesurée correspond au carré de l'amplitude de l'ampere de cette valeur. La conséquence immédiate de ce phénomène est que la somme des carrés des amplitudes doit toujours être égale à 1.

$$\sum_{i \in 2^n} \|coef(i)\|^2 = 1$$

Par exemple, un registre quantique de 3 qbits pourrait contenir ces 2^3 valeurs :

État	Amplitude $a+ib$	Probabilité a^2+b^2
000	$0.37+i0.04$	0.14
001	$0.11+i0.18$	0.04
010	$0.09+i0.31$	0.10
011	$0.30+i0.30$	0.18
100	$0.35+i0.43$	0.31
101	$0.40+i0.01$	0.16
110	$0.09+i0.12$	0.02
111	$0.15+i0.16$	0.05

Dans cet exemple, il y a 14% de chances que la chaîne retournée soit "000", 4% que ce soit "001", ainsi de suite.

Il est possible de créer des portes capables de modifier la valeur des amperes des différentes valeurs. Ainsi, en traitant notre registre quantique avec plusieurs portes, on peut réussir à faire des traitements sur les amperes (et par conséquent sur la probabilité d'obtenir certaines valeurs). Parmi les portes les plus utiles :

- La porte de Hamard-Warshal permet à partir d'un état définit d'obtenir un registre où toutes les valeurs sont équiprobables.
- La rotation permet de multiplier tous les amperes par $\exp^{i\theta}$
- La rotation conditionnelle permet de multiplier les amperes des valeurs ne satisfaisant pas une condition quantique.
- La porte CNot permet de mettre les bit de poids faible aux mêmes états que les bits de poids fort.

On remarque que l'on peut formaliser mathématiquement un registre quantique par un vecteur de 2^n valeurs complexes. On peut ensuite formaliser les portes comme des matrices ayant certaines propriétés (en particulier garder la somme des carrés des valeurs absolues à 1) et l'application d'une porte comme la multiplication de notre vecteur/registre par la matrice/porte. Par la suite, nous prendrons l'habitude de décrire les portes par leurs matrices équivalentes.

Typiquement, un algorithme d'un ordinateur quantique initialisera tous les nombres complexes à des valeurs égales, donc tous les états auront les mêmes probabilités. La liste des nombres complexes peut être imaginée comme un vecteur à 8 éléments. A chaque étape de l'algorithme, le vecteur est modifié par son produit avec une matrice.

6.2 L'algorithme de Grover

L'algorithme de Grover permet de faire des recherche sur une base de donnée non triée de N possédant t éléments vérifiant une condition C en $O(\sqrt{N})$ itérations. Pour cela,

1. on initialise un registre de $log_2(N)$ qbits de manière équiprobable grâce à la porte de Hadamard.
2. on applique Hadamard
3. on applique Rotation de $\pi/2$ sur la valeur 0
4. on applique Hadamard
5. on applique Rotation de $\pi/2$ sur les valeurs ne satisfaisant pas C.
6. on réapplique les 4 étapes précédentes m fois
7. on lit la valeur du registre. La valeur lue est la solution avec une certaine probabilité (Nous allons y revenir)

Les deux problèmes principaux de l'algorithme sont tout d'abord de formaliser la fonction condition de l'étape 5. Pour le moment, ce problème est mis de coté par les physiciens. On utilise une machine habituelle qui retourne 1 si la valeur est solution et 0 sinon. Ensuite, le choix de m est primordial. Effectivement, Grover démontre que la probabilité que la valeur lue soit la solution est supérieure à $1/2$ uniquement si m est bien choisit. Si il est trop petit, la solution n'est pas encore optimale. Si il est trop grand, on s'éloigne de la solution.

m change en fonction de N et du nombre de solutions possibles t parmi les 2^N valeurs.

Grover démontre que si

$$m = \frac{\pi}{4 \arcsin \sqrt{t/N}} \leq \frac{\pi}{4}\sqrt{\frac{N}{t}} \tag{6.1}$$

alors la probabilité que la valeur lue soit solution est supérieure à $\frac{N-t}{N}$. Si m n'est pas optimal, alors, la probabilité que la valeur retournée soit une solution diminue.

6.3 Application de l'algorithme de Grover

6.3.1 Application au problème 1

Il nous faut tout d'abord un registre suffisamment grand pour contenir toutes les solutions de notre problème. Commençons par borner nos solutions. Je rappelle les résultats de l'élagage sur la phrase du problème 1 (**??**) :

$$\begin{aligned} sup_0 &= 21 \\ sup_1 &= 12 \\ sup_2 &= 8 \\ sup_3 &= 6 \\ sup_4 &= 5 \\ sup_5 &= 4 \\ sup_6 &= 4 \\ sup_7 &= 3 \\ sup_8 &= 3 \\ sup_9 &= 3 \end{aligned}$$

Cela fait 26×10^6 solutions. Nous avons donc besoin d'un registre de $\lceil \log_2(26^6) \rceil = 24$ qbits pour stocker l'intégralité des solutions.

D'après nos résultats de l'algorithme exhaustif, on sait que dans le cas du problème 1, il existe deux solutions. On peut en déduire d'après 6.1 le nombre d'itérations que nous devrons faire avant de lire la valeur :

$$m = \frac{\pi}{4 \arcsin \sqrt{\frac{2}{26 \times 10^6}}} \simeq 28 \times 10^2 \tag{6.2}$$

La probabilité pour que la valeur retournée soit une solution sera alors supérieure à $\frac{26 \times 10^6 - 2}{26 \times 10^6} (\simeq 1)$. Par conséquent les chance que la valeur retournée soit une solution à notre problème, mais ça n'est garanti. Le fait que le résultat soit une solution avec une certaine probabilité ne pose pas de problèmes puisque nous sommes capables de l'évaluer [1].

En mettant en oeuvre l'algorithme de Grover, nous serions capable de trouver une solution au problème 1 en moins de 3000 itérations.

Concrètement, nous utiliserions une porte de Hadamard pour initialiser notre registre aux 26×10^6 valeurs possibles. Notre première difficulté serait de décrire une condition quantique telle que la rotation s'applique uniquement si la valeur n'est pas une solution. [2] Nous itérerions ensuite sur l'algorithme décrit dans la section précédente 28×10^2 fois.

Néanmoins, nous avons triché, nous savions combien de solutions il y avait à notre problème. Si nous n'avons pas le nombre de solution, nous ne pouvons pas donner avec précision une valeur à m. Nous rappelons que lorsque nous nous éloignons du m optimal, la probabilité pour que la bonne solution soit retournée diminue.

[1] ce qui n'est pas le cas avec le problème de la recherche du plus petit nombre par exemple

[2] Nous utiliserions une table de hash capable de faire la translation entre la valeur représentée par notre registre et les instance du problème.

6.3.2 Application au problème 3

On peut réappliquer le même raisonnement avec le problème 3. Je rappelle les résultats de l'élagage sur la phrase *Ce titre contient a b c d e f g h i j k l m n o p q r s t u v w x y et z* **??** :

$$
\begin{aligned}
sup_{\mathsf{a}} &= 24 & sup_{\mathsf{n}} &= 34 \\
sup_{\mathsf{b}} &= 0 & sup_{\mathsf{o}} &= 28 \\
sup_{\mathsf{c}} &= 20 & sup_{\mathsf{p}} &= 20 \\
sup_{\mathsf{d}} &= 20 & sup_{\mathsf{q}} &= 22 \\
sup_{\mathsf{e}} &= 48 & sup_{\mathsf{r}} &= 34 \\
sup_{\mathsf{f}} &= 20 & sup_{\mathsf{s}} &= 22 \\
sup_{\mathsf{g}} &= 17 & sup_{\mathsf{t}} &= 40 \\
sup_{\mathsf{h}} &= 20 & sup_{\mathsf{u}} &= 25 \\
sup_{\mathsf{i}} &= 40 & sup_{\mathsf{v}} &= 17 \\
sup_{\mathsf{j}} &= 0 & sup_{\mathsf{w}} &= 0 \\
sup_{\mathsf{k}} &= 0 & sup_{\mathsf{x}} &= 21 \\
sup_{\mathsf{l}} &= 0 & sup_{\mathsf{y}} &= 0 \\
sup_{\mathsf{m}} &= 0 & sup_{\mathsf{z}} &= 27
\end{aligned}
$$

Cela fait 12×10^{18} solutions. Nous avons donc besoin d'un registre de $\lceil \log_2(12^{18}) \rceil = 64$ qbits. Si on suppose qu'il n'y a qu'une solution au problème ($t = 1$). On obtient :

$$m = \frac{\pi}{4 \arcsin \sqrt{\frac{1}{12 \times 10^{18}}}} \simeq 27 \times 10^8 \tag{6.3}$$

Néanmoins, nous avons supposé que $t = 1$, donc notre m n'est pas forcément optimal. Il est difficile dans ces conditions de dire avec quelle probabilité la valeur retournée est une solution du problème.

6.4 Conclusion

Nous avons essayé ici d'avoir une idée de d'application de l'algorithme de Grover pour résoudre nos problèmes. Les résultats sont médiocres. Tout d'abord, il y a un certain nombre de points qu'on ne sait pas, à l'heure actuelle, mettre en pratique (en particulier la condition de la porte de rotation). Ensuite, la mise en place de l'algorithme est difficile. Nous avons vu que plusieurs paramètres entrent en compte. Enfin, les résultats obtenus sont moyens. Les algorithmes classiques permettent d'utiliser des principes d'élagage ou des métaheuristiques pour trouver plus facilement la solution. Nous ne bénéficions pas ici de ces avantages. Trouver une solution en 3 milliards d'itérations n'est pas un résultat fabuleux.

Annexe A

Implémentation des algorithmes naifs et du recuit simulé

Sommaire

A.1 Bibliothéque commune **37**
A.1.1 main.c 37
A.1.2 args.h 38
A.1.3 args.c 39
A.1.4 int_to_french.h 41
A.1.5 int_to_french.c 41
A.1.6 génération de occurs.h : occurs_gen.c 42
A.1.7 itf_static.h 43
A.1.8 itf_static.c 44
A.2 Elagage du domaine **45**
A.2.1 prune_comm.c 45
A.2.2 prune_alpha.h 45
A.2.3 prune_alpha.c 46
A.2.4 prune_numer.h 51
A.2.5 prune_numer.c 51
A.3 Algorithmes naifs **55**
A.3.1 counts.h 55
A.3.2 counts.c 56
A.3.3 naive.h 59
A.3.4 naive.c 60
A.4 Recuit simulé **61**
A.4.1 recuit.h 61
A.4.2 recuit.c 61

A.1 Bibliothéque commune

A.1.1 main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

```c
#include <ctype.h>
#include <unistd.h>
#include <assert.h>
#include "args.h"
#include "naive.h"
#include "recuit.h"

inline static unsigned int randi(unsigned int min, unsigned int max)
{
  return min + (unsigned int) ((float)max * rand() / (RAND_MAX + 1.0));
}
/*
 * Read the initalisation string from standart input.
 */
static void      read_init(char *init, int flag)
{

  unsigned int  n;
  unsigned int  i;
  char          *p;


  if (flag & OPT_RANDINIT) /* Random initialisation: problem 2 */

  {
    n = randi(1, 20);
    for (p = init, i = 0; i < n; ++i, ++p)
      *p = '0' + randi(0,9);
    *(p++) = '\n';
    *p = '\0';
  }
  else if (flag & OPT_EMPTYINIT) /* Empty init: problem 1 */
    strcpy(init, "\n");
  else /* User init */
  {
    printf("Give␣me␣a␣sentance:␣");
    fgets(init, BUF_SIZE / 2, stdin);
  }
  init[strlen(init) - 1] = '\0';
  strncat(init, "␣Dans␣cette␣phrase␣il␣y␣a:", BUF_SIZE / 2 - strlen(init));
  if (flag & OPT_NOCASE)
    for (; *init != '\0'; ++init)
      *init = tolower(*init);
}


/*
 * Return the searched characters corresponding to the flags set.
 */
static char     *get_searched(int flag)
{
  if (flag & OPT_DIGIT)
    return "0123456789";
  if (flag & OPT_ALPHA)
    return "abcdefghijklmnopqrstuvwxyz";

  /* Unreached */
  assert(0);
  return NULL;
}


/*
 * Main functions.
 */
int main(int argc, char **argv)
{
  int           flag;                   /* flag containing options. */
  char          init[BUF_SIZE / 2];     /* Initialisation string */
  char          *searched = NULL;       /* Caracters we want to count */
```

```
  unsigned int  searched_len;           /* Nb of caracters to count. */

  /* Initialisation, commun variables. */
  srand(time(NULL) * getpid());
  flag = parse_args(argc, argv);
  read_init(init, flag);
  if (flag & OPT_VERBOSE)
    printf("init␣=␣%s\n", init);
  searched = get_searched(flag);
  searched_len = strlen(searched);

  if (flag & OPT_RECUIT)
    return recuit_main(flag, init, searched, searched_len);
  else
    return naive_main(flag, init, searched, searched_len);
}
```

A.1.2 args.h

```
#ifndef ARGS_H__
# define ARGS_H__

/* Flags for options. */
#define OPT_DIGIT       0x01    /* Count digits */
#define OPT_ALPHA       0x02    /* Count letter */
#define OPT_ALL         0x04    /* Change all count each round */
#define OPT_ONE         0x08    /* Chane only one count at random each round */
#define OPT_FIRST       0x10    /* Chane only the first count each round */
#define OPT_LITERAL     0x20    /* Write numbers literally */
#define OPT_NOCASE      0x40    /* Insensitive compare */
#define OPT_VERBOSE     0x80    /* Verbose mode */
#define OPT_RANDOM      0x100   /* Random initialisation of counts */
#define OPT_BRUTE       0x200   /* Brute force search */
#define OPT_RECUIT      0x400   /* Recuit simule algorithms */
#define OPT_RANDINIT    0x800   /* Random initialisation string */
#define OPT_EMPTYINIT   0x1000  /* Empty initialisation string */

/* Program version. */
#define VERSION         "5"

/* Program Authors */
#define AUTHORS         "Marco␣Tessari"

/* Maximum size of the working string */
#define BUF_SIZE        1024

/* Display an error message and exit. */
void    error(const char *str);

/* Parse command line arguments. */
int     parse_args(int argc, char * const argv[]);

#endif /* __ARGS_H__*/
```

A.1.3 args.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <getopt.h>
#include "args.h"

/*
 * Display an error message and exit.
 */
void    error(const char *str)
{
  printf("%s\n", str);
  exit(1);
```

```
}

/*
 * Print version and exit.
 */
static void     version(char *prg_name)
{
  printf("%s␣version␣%s\nMade␣by␣%s.\n", prg_name, VERSION, AUTHORS);
  exit(0);
}

/*
 * Print usage and exit.
 */
static void     usage(char *prg_name)
{
  static char usage[] =
"\t␣-d,␣--digit␣␣␣␣␣␣␣␣␣␣Count␣digits␣(default).\n\
\t␣-l,␣--letters␣␣␣␣␣␣␣␣Count␣letters.\n\
\t␣-v,␣--verbose␣␣␣␣␣␣␣␣Verbose␣mode.\n\
\t␣-V,␣--version␣␣␣␣␣␣␣␣Print␣version␣and␣exit.\n\
\t␣-h,␣--help␣␣␣␣␣␣␣␣␣␣␣Print␣help.\n\
\t␣-i,␣--insensitive␣␣␣␣Ignore␣case␣in␣letter␣count.\n\
\t␣-1,␣--one␣␣␣␣␣␣␣␣␣␣␣␣Change␣only␣one␣count␣(default).\n";

  /* String must be split bcause C compilers does not support too long
     constant strings. */
  static char usage2[] =
"\t␣-a,␣--all␣␣␣␣␣␣␣␣␣␣␣␣Change␣all␣counts.\n\
\t␣-f,␣--first␣␣␣␣␣␣␣␣␣␣Change␣first␣count.\n\
\t␣-r,␣--random-init␣␣␣␣Counts␣are␣randomly␣initialized.\n\
\t␣-L,␣--literal␣␣␣␣␣␣␣␣Write␣numbers␣in␣literal␣form.\n\
\t␣-b,␣--brute-force␣␣␣␣Brute␣force␣search␣of␣solutions.\n\
\t␣-R,␣--recuit␣␣␣␣␣␣␣␣␣Use␣[recuit␣simulé].\n\
";

  printf("usage:␣%s␣[options]\nOptions␣are:\n%s%s", prg_name, usage, usage2);
  exit(0);
}

/*
 * Parse command line arguments.
 * See usage() for options.
 */
int     parse_args(int argc, char * const argv[])
{
  static struct option long_options[] =
    {
      {"digits", 0, 0, 'd'},
      {"letters", 0, 0, 'l'},
      {"verbose", 0, 0, 'v'},
      {"version", 0, 0, 'V'},
      {"help", 0, 0, 'h'},
      {"insensitive", 0, 0, 'i'},
      {"one", 0, 0, '1'},
      {"all", 0, 0, 'a'},
      {"first", 0, 0, 'f'},
      {"random-init", 0, 0, 'r'},
      {"literal", 0, 0, 'l'},
      {"brute", 0, 0, 'b'},
      {"recuit", 0, 0, 'R'},
      {0, 0, 0, 0}
    };

  int   option_index = 0;
  int   flag = 0;
  int   c;

  while (42)
  {
```

```
    c = getopt_long(argc, argv, "bdlvVhi1afLrR", long_options, &option_index);
    if (c == -1)
      break ;

    switch (c)
    {
    case ’d’: flag |= OPT_DIGIT; break ;
    case ’l’: flag |= OPT_ALPHA; break ;
    case ’v’: flag |= OPT_VERBOSE; break ;
    case ’i’: flag |= OPT_NOCASE; break ;
    case ’1’: flag |= OPT_ONE; break ;
    case ’a’: flag |= OPT_ALL; break ;
    case ’f’: flag |= OPT_FIRST; break ;
    case ’L’: flag |= OPT_LITERAL; break ;
    case ’r’: flag |= OPT_RANDOM; break ;
    case ’b’: flag |= OPT_BRUTE; break ;
    case ’R’: flag |= OPT_RECUIT; break ;
    case ’V’: version(argv[0]); break ;
    case ’?’: case ’h’: usage(argv[0]); break ;

    default:
      /* Unreachable */
      assert(0);
    }
  }

  /* Check flag coherance. */
  if ((flag & OPT_DIGIT) && (flag & OPT_ALPHA))
    error("Arguments␣error:␣You␣cannot␣set␣both␣’--digits’␣and␣’--letters’");
  if ((flag & OPT_ONE) && (flag & OPT_ALL))
    error("Arguments␣error:␣You␣cannot␣set␣both␣’--one’␣and␣’--all’");
  if ((flag & OPT_ALL) && (flag & OPT_FIRST))
    error("Arguments␣error:␣You␣cannot␣set␣both␣’--first’␣and␣’--all’");
  if ((flag & OPT_ONE) && (flag & OPT_FIRST))
    error("Arguments␣error:␣You␣cannot␣set␣both␣’--first’␣and␣’--one’");
  if ((flag & OPT_BRUTE) &&
      ((flag & OPT_FIRST) || (flag & OPT_ALL) || (flag & OPT_ONE)))
    error("Arguments␣error:␣You␣cannot␣set␣both␣random␣and␣brute␣search.");
  if ((flag & OPT_BRUTE) && (flag & OPT_RANDOM))
    error("Arguments␣error:␣Brute␣search␣cannot␣be␣initialized␣at␣random");
  if ((flag & OPT_RECUIT) &&
      ((flag & OPT_RANDOM) || (flag & OPT_BRUTE) || (flag & OPT_ALL) ||
       (flag & OPT_ONE) || (flag & OPT_FIRST)))
    error("Arguments␣error:␣Recuit␣cannot␣be␣set␣with␣other␣options");

  /* Set default if needed. */
  if (!(flag & OPT_DIGIT) && !(flag & OPT_ALPHA))
    flag |= OPT_DIGIT;
  if (!(flag & OPT_ONE) && !(flag & OPT_FIRST)
      && !(flag & OPT_ALL) && !(flag & OPT_BRUTE)
      && !(flag & OPT_RECUIT))
    flag |= OPT_ONE;

  return flag;
}
```

A.1.4 int_to_french.h

```
#ifndef INT_TO_FRENCH_
# define INT_TO_FRENCH_

# ifndef BUF_SIZE
#  define BUF_SIZE 1024
# endif

void int_to_french(unsigned int nb, char buf[BUF_SIZE]);

#endif
```

A.1.5 int_to_french.c

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "int_to_french.h"


static void cat(char buf[BUF_SIZE], const char to_cat[])
{
  strncat(buf, to_cat, BUF_SIZE - strlen(buf));
  return ;
}

static unsigned int high_numbers(unsigned int num, const char string[],
                                 unsigned int nb, char buf[BUF_SIZE])
{
  unsigned int  tmp;
  char          s[BUF_SIZE];

  tmp = nb / num;
  if (tmp > 1)
  {
    int_to_french(tmp, buf);
    snprintf(s, BUF_SIZE, "␣%ss",string);
    cat(buf, s);
  }
  else
  {
    if (nb == 1000000)
      cat(buf, "un␣");
    cat(buf, string);
  }
  nb %= num;
  if (nb)
    cat(buf, "␣");
  return nb;
}

/*
 * Convert an interger to the french literal representation.
 * Result is put on buff. Buff must be allocated, is there is
 * not enought size in the uffer the string will be troncated.
 * Buffer must be clear before calling this method.
 * Default BUF_SIZE is 1024, if you want to overrigh it just
 * define BUF_SIZE before including int_to_french.h
 */
void int_to_french(unsigned int nb, char buf[BUF_SIZE])
{
  static const char *units[] = {
    "", "un", "deux", "trois", "quatre",
    "cinq", "six", "sept", "huit", "neuf",
    "dix", "onze", "douze", "treize", "quatorze",
    "quinze", "seize"
  };

  static const char *dizaines[] = {
    "", "dix", "vingt", "trente", "quarante",
    "cinquante", "soixante", 0, "quatre-vingt", 0
  };

  int   tmp;

  if (nb == 0)
    cat(buf, "zero");
  while (1)
  {
    if (nb < 17)
    {
```

```
      cat(buf, units[nb]);
      return ;
    }
    if (nb < 100)
    {
      tmp = nb / 10;
      if ((tmp == 9) || (tmp == 7))
        --tmp;
      cat(buf, dizaines[tmp]);
      if (((nb % 10) == 1) && tmp < 8)
        cat(buf, "-et");
      if ((nb % 10) || ((nb / 10) == 7) || ((nb / 10) == 9))
        cat(buf, "-");
      nb -= tmp * 10;
      continue ;
    }
    else if (nb < 1000)
      nb = high_numbers(100, "cent", nb, buf);
    else if (nb < 1000000)
      nb = high_numbers(1000, "mille", nb, buf);
    else
      nb = high_numbers(1000000, "million", nb, buf);
  }
  return ;
}
```

A.1.6 génération de occurs.h : occurs_gen.c

```
#include <stdio.h>
#include <string.h>
#include "int_to_french.h"

int main() {
  char buf[BUF_SIZE];
  unsigned int i, k;
  char        j;
  int count;
  int total;


  printf("//␣Generated␣file,␣don't␣touch!\n");
  printf("static␣const␣unsigned␣char␣occurs[][27]␣=␣{\n");
  printf("//␣␣a,␣b,␣c,␣d,␣e,␣f,␣g,␣h,␣i,␣j,␣k,␣l,␣m,␣n,␣o,␣p,␣q,␣r,␣s,␣t,␣u,␣v,␣w,␣x,␣y,␣z,␣total␣//␣header\n");
  for (i = 0; i< 1000; i++) {
    bzero(buf, BUF_SIZE);
    int_to_french(i, buf);
    printf("␣␣{␣");
    total = 0;
    for (j = 'a'; j <= 'z'; j++) {
      count = 0;
      for (k = 0; k < strlen(buf); k++)
        if (buf[k] == j)
          count++;
      printf("%i,␣", count);
      total += count;
    }
    printf("%i␣},␣//␣%s\n", total, buf);
  }
  printf("␣␣{␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0,␣0␣}␣␣//␣end\n␣};\n");
  return 0;
}
```

A.1.7 itf_static.h

```
#ifndef ITF_STATIC_H
# define ITF_STATIC_H
#include "occurs.h"

static const unsigned char mille[][27] = {
```

```
// a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, total
// mille
  { 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5  },
// millions
  { 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 2, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 8  },
// milliards
  { 1, 0, 0, 1, 0, 0, 0, 0, 2, 0, 0, 2, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 9  }
};

/*
 * Return number of occurance of character 'a' + c in number n written in
 * letters.
 * This function is generic. You can append some attribute in occurs array
 * Use c = 26 to get number of lettre in n
 * Don't forget to générate occurs.h to compile this....
 */
int getoccurs(int c, unsigned int n) {
  int u1;
  int u2;
  int u3;
  int u4;
  int ret = 0;

  /* Most used case */
  if (n < 1000)
    return occurs[n][c];
  /* Général case*/
  else {
    u1 = n % 1000;
    n = n / 1000;
    u2 = n % 1000;
    n = n / 1000;
    u3 = n % 1000;
    u4 = n / 1000;
    ret = 0;
    if (u4 == 0)  /* Milliard */
      ret += 0;
    else if (u4 == 1)
      ret += mille[2][c];
    else
      ret += mille[2][c] + occurs[u4][c];
    if (u3 == 0)  /* Millions */
      ret += 0;
    else if (u3 == 1)
      ret += mille[1][c];
    else
      ret += mille[1][c] + occurs[u3][c];
    if (u2 == 0)  /* Mille */
      ret += 0;
    else if (u2 == 1)
      ret += mille[0][c];
    else
      ret += mille[0][c] + occurs[u2][c];
    ret += occurs[u1][c];
  }
  return ret;
}

#endif /* ITF_STATIC_H */
```

A.1.8 itf_static.c

```
#include "occurs.h"

static const unsigned char mille[][27] = {

// a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, total
// mille
  { 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5  },
```

```
// millions
  { 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 2, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 8  },
// milliards
  { 1, 0, 0, 1, 0, 0, 0, 0, 2, 0, 0, 2, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 9  }
};

int getoccurs(int c, unsigned int n) {
  int u1;
  int u2;
  int u3;
  int u4;
  int ret = 0;

  /* Most used case */
  if (n < 1000)
    return occurs[n * 27 + c];
  /* Général case*/
  else {
    u1 = n % 1000;
    n = n / 1000;
    u2 = n % 1000;
    n = n / 1000;
    u3 = n % 1000;
    u4 = n / 1000;
    ret = 0;
    if (u4 == 0)  /* Milliard */
      ret += 0;
    else if (u4 == 1)
      ret += mille[2][c];
    else
      ret += mille[2][c] + occurs[u4][c];
    if (u3 == 0)  /* Millions */
      ret += 0;
    else if (u3 == 1)
      ret += mille[1][c];
    else
      ret += mille[1][c] + occurs[u3][c];
    if (u2 == 0)  /* Mille */
      ret += 0;
    else if (u2 == 1)
      ret += mille[0][c];
    else
      ret += mille[0][c] + occurs[u2][c];
    ret += occurs[u1][c];
  }
  return ret;
}
```

A.2 Elagage du domaine

A.2.1 prune_comm.c

```
// You must specify SIZE!
// Don't compile this file "as is"

/*
 * Allow to print a vector of SIZE. (for debug pruposes only)
 */
static void printc(int *counts) {
  int i;

  for (i = 0; i < SIZE; i++)
    printf("%2i␣", counts[i]);
  printf("\n");
}

/*
 * return if counts containt a -1
```

```c
 */
static int isfull(int *counts)
{
  int   i;

  for (i = 0; i < SIZE; ++i)
    if (counts[i] == -1)
      return 1;
  return 0;
}

/*
 * Count number of occurence of a digit in vector
 */
static int getoccurs_vect(int digit, int *counts) {
  int   ret = 0;
  int   i;

  for (i = 0; i < SIZE; ++i)
    ret += getoccurs(digit, counts[i]);
  return ret;
}

/*
 * Say if argument is a solution
 */
static int isok(int *counts, int *counts_init) {
  int i;

  for (i = 0; i < SIZE; ++i) {
    if (counts[i] != counts_init[i] + getoccurs_vect(i, counts))
      return 0;
  }
  return 1;
}
```

A.2.2 prune_alpha.h

```c
#ifndef PRUNE_ALPHA_H
# define PRUNE_ALPHA_H

// Size of our vector (26 = card([(a)..(z)]))
#define SIZE_ALPHA 26

void prun_alpha(char *in);

/*
 * Make first pruning. Can be used in other algorithm to limit domain
 * (In this, case, don't forget to call prunemax prunemin after))
 */
void init_alpha(int *counts_max,
          int *counts_min,
          int *counts_init,
          char *in);

/*
 * Main recursion of algorithm (should be for internal puposes only...)
 */
void run_alpha(int *counts,
         int *counts_max,
         int *counts_min,
         int *counts_init);

#endif /* PRUNE_ALPHA_H */
```

A.2.3 prune_alpha.c

```c
#include <string.h>
#include <stdio.h>
```

```
#include "itf_static.h"
#include "prune_alpha.h"

// Size of our vector   (26 = card([0..9]))
#define SIZE SIZE_ALPHA

unsigned long long total;
static int *stat;
static int smins, smaxs;
int initial =1;

/* I want templated langage!!! */
#include "prune_comm.c"


/*
 * Search most number of character c can be found in a number beetween min
 * and max
 */
int searchchard(unsigned char c, int min, int max) {
  int bestnb = 0;
  int bestoccur = 0;

  for (; max >= min; --max) {
    if (getoccurs((int) c, max) > bestoccur) {
      bestnb = max;
      bestoccur = getoccurs((int) c, max);
    }
  }
  return bestoccur;
}

void init(int *counts_max) {
  int i, j, k;
  int max;

  max = 0;
  for (i = 0; i < SIZE; i++)
    if (counts_max[i] > max)
      max = counts_max[i];

  smins = smaxs = max + 1;
  stat = malloc(sizeof(int) * smins * smaxs * 26);
  for (i = 0; i < smaxs; i++)
    for (j = 0; j < smins; j++)
      for (k = 0; k < 26; k++)
        stat[i * smins * 26 + j * 26 + k] = searchchard(k, j, i);
}

int searchchar(unsigned char c, int min, int max) {
  //printf("%i %i %i\n", c, min, max);
  if (initial)
    return searchchard(c, min, max);
  else
    return stat[max * smins * 26 + min * 26 + c];
}

/*
 * Search most number of character can be found in a number beetween min
 * and max
 */
int searchlen(int min, int max) {
  int bestnb = 0;
  int bestlen = 0;

  for (; max >= min; --max) {
    if (getoccurs(26, max) > bestlen) {
      bestnb = max;
      bestlen = getoccurs(26, max);
    }
```

```
  }
74  return bestlen;
}

/*
 * Prune minimum of domains. Very efficient on letters.
 * Notice this optimisation isn't implemented digital algo
 */
int prunemin(int *counts,
             int *counts_max,
             int *counts_min, int n)
84 {
  int i;

  for (i = 0; i < SIZE; i++) {
    if (getoccurs(i, counts[n])) {
      counts_min[i] += getoccurs(i, counts[n]);
      if (counts_max[i] < counts_min[i] ||
          (counts[i] != -1 && counts_min[i] > counts[i]))
        return 1;
      /* These lines can be removed... maybe they optimise a little
94        algorithm, but they force to call memcpy() in recur() function... */
      if (counts_max[i] == counts_min[i] && counts[i] == -1) {
        counts[i] = counts_max[i];
        if (prunemin(counts, counts_max, counts_min, i))
          return 1;
      }
      /****/
    }
  }
  return 0;
104 }

/*
 * Prune domains
 * Notice differences with equivalent algorithm which work on digit
 * Here we use letters property to have newmax different beetween each letter
 * (With digit, we use sum globaly for all digit)
 *
 */
int prunemax(int *counts,
114            int *counts_max,
             int *counts_min,
             int *counts_init)
{
  int changed;
  int i, j;
  /* equivalent of sum for digit algorithm */
  int newmax;

  changed = 1;
124  while (changed) {
    changed = 0;
    for (i = 0; i < SIZE; i++) {
      newmax = counts_init[i];
      for (j = 0; j < SIZE; j++) {
        if (counts[j] == -1)
          newmax += searchchar(i, counts_min[j], counts_max[j]);
        else
          newmax += searchchar(i, counts[j], counts[j]);
      }
134      if (newmax < counts_min[i] ||
          (counts[i] != -1 && newmax < counts[i]))
        return 1;
      if (counts_max[i] > newmax) {
        counts_max[i] = newmax;
        /* Idem */
        if (counts_max[i] == counts_min[i] && counts[i] == -1) {
          counts[i] = counts_max[i];
          if (prunemin(counts, counts_max, counts_min, i))
```

```
            return 1;
144       }
          /****/
          changed = 1;
        }
      }
    }
    return 0;
  }

  /*
154  * Main recursion of algorithm
   */
  void run_alpha(int *counts,
             int *counts_max,
             int *counts_min,
             int *counts_init) {
     int          tmpmax[SIZE];
     int          tmpmin[SIZE];
     int          tmp[SIZE];
     int          i, j;
164
     if (!(total % 100000)) {
       // exit(0);
       printf("%llueme␣instanciation␣:␣", total);
       printc(counts);
     }
     ++total;
     if (!isfull(counts)){
       if (isok(counts, counts_init)) {
         printf("FOUND␣IT:␣");
174      printc(counts);
       }
       return ;
     }
     if (prunemax(counts, counts_max, counts_min, counts_init))
       return ;
     /* Optimisation to do: instanciate variable/value which have biggest
        number avec characters */
     for (i = SIZE; i >= 0; --i)
       if (counts[i] == -1)
184      break;
     for (j = counts_min[i]; j <= counts_max[i]; ++j) {
       memcpy(tmpmin, counts_min, sizeof(tmpmin));
       memcpy(tmpmax, counts_max, sizeof(tmpmax));
       memcpy(tmp, counts, sizeof(tmp));
       tmp[i] = j;
       if (prunemin(tmp, tmpmax, tmpmin, i))
         continue ;
       run_alpha(tmp, tmpmax, tmpmin, counts_init);
     }
194 }

  /*
   * Make first pruning. Can be used in other algorithm to limit domain
   * (In this, case, don't forget to call prunemax prunemin after))
   */
  void init_alpha(int *counts_max,
            int *counts_min,
            int *counts_init,
            char *in)
204 {
    int           i;
    unsigned int  prev;
    int           changed = 1;
    int           sum = 0;

    bzero(counts_min, SIZE * sizeof(int));
    for (i = 0; i < strlen(in); i++)
      if (in[i] >= 'a' && in[i] <= 'z')
```

```
      counts_min[in[i] - 'a']++;
  memcpy(counts_init, counts_min, SIZE * sizeof(unsigned int));
  for (i = 0; i < SIZE; i++) {
    counts_max[i] = 4998 + counts_init[i];
  }
// Useless  Not enought efficient  //
// Equivalent of prune with digits
  while (changed) {
    changed = 0;
    sum = 0;
    for (i = 0; i < SIZE; i++)
      sum += getoccurs(26, counts_max[i]);
    for (i = 0; i < SIZE; i++) {
      prev = counts_max[i];
      counts_max[i] = SIZE * searchlen(0, sum) + counts_min[i];
      if (prev != counts_max[i])
           changed = 1;
    }
  }
}

/*
 * Main function which call all functions
 */
void prun_alpha(char *in) {
  int   counts[SIZE];
  int   counts_max[SIZE];
  int   counts_min[SIZE];
  int   counts_init[SIZE];
  int   i;
  unsigned long long tot = 1;

  total = 0;
  for (i = 0; i < SIZE; i++)
    counts[i] = -1;
  init_alpha(counts_max, counts_min, counts_init, in);
  printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣a␣␣b␣␣c␣␣d␣␣e␣␣f␣␣g␣␣h␣␣i␣␣j␣␣k␣␣l␣␣m␣␣n␣␣o␣␣p␣␣q␣␣r␣␣s␣␣t␣␣u␣␣v␣␣w␣␣x␣␣y␣␣z\n");
  printf("Liste␣des␣mimimas␣:␣");
  printc(counts_min);
  printf("Liste␣des␣maximas␣:␣");
  printc(counts_max);
  if (prunemax(counts, counts_max, counts_min, counts_init))
    printf ("Impossible␣case␣\n");
  for (i = 0; i < SIZE; i++)
    tot *= counts_max[i];
  printf("Nombre␣de␣solution␣à␣parcourir:␣%llu\n", tot);
  prunemax(counts, counts_max, counts_min, counts_init);
  printf("Résultats␣Elagage:␣\n");
  printc(counts_min);
  printc(counts_max);
  for (i = 0; i < SIZE; i++)
    tot *= counts_max[i];
  printf("Nombre␣de␣solution␣à␣parcourir:␣%llu\n", tot);

  //Maybe you want to fix some value to help algorithm?
/*   counts['z' - 'a'] = 4; */
/*   if (prunemin(counts, counts_max, counts_min, 'z' - 'a')) */
/*     exit(47); */
/*   counts['x' - 'a'] = 6; */
/*   if (prunemin(counts, counts_max, counts_min, 'x' - 'a')) */
/*     exit(46); */
/*   counts['v' - 'a'] = 1; */
/*   if (prunemin(counts, counts_max, counts_min, 'v' - 'a')) */
/*     exit(45); */
/*   counts['u' - 'a'] = 18; */
/*   if (prunemin(counts, counts_max, counts_min, 'u' - 'a')) */
/*     exit(44); */
/*   counts['t' - 'a'] = 15; */
/*   if (prunemin(counts, counts_max, counts_min, 't' - 'a')) */
/*     exit(43); */
```

```
/*   counts[’s’ - ’a’] = 7; */
/*   if (prunemin(counts, counts_max, counts_min, ’s’ - ’a’)) */
/*     exit(42); */
/*   counts[’r’ - ’a’] = 7; */
/*   if (prunemin(counts, counts_max, counts_min, ’r’ - ’a’)) */
/*    exit(41); */
/*   counts[’q’ - ’a’] = 7; */
/*   if (prunemin(counts, counts_max, counts_min, ’q’ - ’a’)) */
/*    exit(40); */
/*   counts[’p’ - ’a’] = 4; */
/*   if (prunemin(counts, counts_max, counts_min, ’p’ - ’a’)) */
/*    exit(39); */
/*   counts[’o’ - ’a’] = 3; */
/*   if (prunemin(counts, counts_max, counts_min, ’o’ - ’a’)) */
/*    exit(38); */
  counts[’n’ - ’a’] = 16;
  if (prunemin(counts, counts_max, counts_min, ’n’ - ’a’))
   exit(37);
  counts[’i’ - ’a’] = 13;
  if (prunemin(counts, counts_max, counts_min, ’i’ - ’a’))
    exit(36);
/*   counts[’h’ - ’a’] = 2; */
/*   if (prunemin(counts, counts_max, counts_min, ’h’ - ’a’)) */
/*    exit(43); */
/*   counts[’g’ - ’a’] = 1; */
/*   if (prunemin(counts, counts_max, counts_min, ’g’ - ’a’)) */
/*   exit(42); */
/*   counts[’f’ - ’a’] = 2; */
/*   if (prunemin(counts, counts_max, counts_min, ’f’ - ’a’)) */
/*    exit(41); */
/*   counts[’e’ - ’a’] = 19; */
/*   if (prunemin(counts, counts_max, counts_min, ’e’ - ’a’)) */
/*    exit(35); */
/*   counts[’d’ - ’a’] = 5; */
/*   if (prunemin(counts, counts_max, counts_min, ’d’ - ’a’)) */
/*    exit(39); */
/*   counts[’c’ - ’a’] = 5; */
/*   if (prunemin(counts, counts_max, counts_min, ’c’ - ’a’)) */
/*    exit(38); */
/*   counts[’a’ - ’a’] = 4; */
/*   if (prunemin(counts, counts_max, counts_min, ’a’ - ’a’)) */
/*    exit(37); */
  prunemax(counts, counts_max, counts_min, counts_init);
  printf("Résultats␣Elagage␣(Valeurs␣fixées):␣\n");
  printc(counts_min);
  printc(counts_max);
  for (i = 0; i < SIZE; i++)
    tot *= counts_max[i];
  printf("Nombre␣de␣solution␣à␣parcourir:␣%llu\n", tot);
  init(counts_max);
  initial = 0;
  run_alpha(counts, counts_max, counts_min, counts_init);
  printf("Nombre␣d’instanciations␣:␣%llu\n", total);
}

int main() {
  prun_alpha("et␣ce␣titre␣contient␣abcdefghijklmnopqrstuvwxyz");
  //prune_alpha("ce titre contient jerome pouiller marco tessari celine bugaud\
  // phrases reflexives abcdefghijklmnopqrstuvwxyz");
  //prune_alpha("abcdefghijklmnopqrstuvwxyz");

  return 0;
}
```

A.2.4 prune_numer.h

```
#ifndef PRUNE_NUMER_H
# define PRUNE_NUMER_H
// Size of our vector (26 = card([(a)..(z)]))
#define SIZE_NUMER 10
```

```c
void prun_numer(char *in);

/*
 * Make first pruning. Can be used in other algorithm to limit domain
 * (In this, case, don't forget to call prunemax prunemin after))
 */
void init_numer(int *counts_max,
          int *counts_min,
          char *in);

/*
 * Main recursion of algorithm (should be for internal puposes only...)
 */
void run_numer(int *counts,
         int *counts_max,
         int *counts_min);

#endif /* PRUNE_NUMER_H */
```

A.2.5 prune_numer.c

```c
#include <string.h>
#include <stdio.h>
#include "prune_numer.h"

// Size of our vector  (26 = card([0..9]))
#define SIZE SIZE_NUMER
// Mettre ici un nombre assez grand, sinon ca peut bugguer
#define INIT_DICHO 600

unsigned long long total;

/*
 * Count number of occurence of a character c in number n written in digits
 */
int getoccurs(int c, int n) {
  int   ret = 0;

  if (n == 0 && c == 0)
    return 1;
  else
    while (n > 0) {
      if (n % SIZE == c)
        ret++;
      n /= 10;
    }
  return ret;
}

/* I want templated langage!!! */
#include "prune_comm.c"

/*
 * Return number of digits in argument
 * equivalent of ⌊log(n)+1⌋
 */
int size(int n) {
  int ret = 0;

  if (n == 0)
    return 1;
  while (n > 0) {
    ret++;
    n /= 10;
  }
  return ret;
}

/*
```

```
 * Return the mimal sum of contruction of num digit on empl number
 * instancemin(4,2,3) = 22+2+2 = 26
 */
unsigned int instancemin(int num, int digit, int empl)
{
  int   dec = 1;
  int   sum = 0;
  int   tmp = 0;

  for (; num > 0; --num) {
    sum += digit * dec;
    tmp++;
    if (tmp == empl) {
      tmp = 0;
      dec *= 10;
    }
  }
  return sum;
}

/*
 * Return sum of all elements of counts
 */
int mksum(int *counts)
{
  int   ret = 0;
  int   i;

  for (i = 0; i < SIZE; ++i)
    ret += counts[i];
  return ret;
}

/*
 * Prune minimum of domains. Do noting for digits
 * Notice this optimisation isn't implemented digital algo
 */
int prunemin(int *counts,
             int *counts_max,
             int *counts_min, int n)
{
  /* Optimisation to do: Prune on inferior limit of domain when a variable is
     instanciate */
  return 0;
}

/*
 * Prune domains in fucntion of variable to instance
 * Notice only maximum of domain are pruned
 */
int prunemax(int *counts,
         int *counts_max,
         int *counts_min)
{
  int   i, j;
  int   stop;
  int   changed;
  int   sum = 0;

  changed = 1;
  while (changed) {
    changed = 0;
    for (i = 9; i >= 0; --i)
      if (counts[i] != -1)
        sum += size(counts[i]) + counts_min[i];
      else
        sum += size(counts_max[i]) + counts_min[i];

    for (i = 9; i >= 0; --i) {
      if (counts[i] != -1) /* already fixed */
```

```
          continue;
        /* i = digit to place, counts[i] = coefficient */
        stop = 0;
        for (j = counts_max[i]; j >= counts_min[i] && !stop; --j) {
          /* is it possible to place j times i ? */
          counts[i] = j;
          /*instancemin(i,n_i,9) + ∑_{j∈déjà instancié} n_j − nboccurence(i,n_i) ≤ ∑_{j=0}^{9} ⌊log(max(n_j)) + 1⌋ */
          if (instancemin(j - getoccurs_vect(i, counts),
                          i, getoccurs_vect(-1, counts))
              + mksum(counts)
              - getoccurs_vect(i, counts) > sum) {
            if (j >= counts_min[i]) {
              counts_max[i] = j;
              changed = 1;
            }
          } else
            stop = 1;
        }
        counts[i] = -1;
        if (stop == 0)
          return 1;
      }
    }
  return 0;
}

/*
 * Main recursion of algorithm
 */
void run_numer(int *counts,
           int *counts_max,
           int *counts_min) {
   int          tmpmax[SIZE];
   int          tmpmin[SIZE];
   int          i, j;

   //printf("%llueme instanciation : ", total);
   //printc(counts);
   ++total;
   if (!isfull(counts)){
     if (isok(counts, counts_min)) {
       printf("FOUND IT: ");
       printc(counts);
     }
     return ;
   }
   if (prunemax(counts, counts_max, counts_min))
     return ;
   for (i = 9; i >= 0; --i)
     if (counts[i] == -1)
       break;
   for (j = counts_min[i]; j <= counts_max[i]; ++j) {
     counts[i] = j;
     memcpy(tmpmin, counts_min, sizeof(tmpmin));
     memcpy(tmpmax, counts_max, sizeof(tmpmax));
     if (prunemin(counts, tmpmax, tmpmin, i))
       continue ;
     run_numer(counts, tmpmax, tmpmin);
   }
   counts[i] = -1;
}

/*
 * Make first pruning. Can be used in other algorithm to limit domain
 * (In this, case, don't forget to call prunemax prunemin after))
 */
void init_numer(int *counts_max,
          int *counts_min,
          char *in) {
```

```c
  int   i;
  int   step;

  bzero(counts_min, SIZE * sizeof(int));
  for (i = 0; i < strlen(in); i++)
    if (in[i] == '0' + i)
      counts_min[i]++;

  for (i = 0; i < SIZE; i++) {
    /* get counts_max[i] by dichotomie */
    counts_max[i] = INIT_DICHO;
    step = counts_max[i] / 2;
    while (step >= 1) {
      if (counts_max[i] <= SIZE * size(counts_max[i]) + counts_min[i])
         counts_max[i] += step;
      else
         counts_max[i] -= step;
      step = (step + ((step == 1) ? 0 : 1)) / 2;
    }
    if ( counts_max[i] + 1 <= SIZE * size(counts_max[i] + 1) + counts_min[i])
       counts_max[i]++;
    if ( counts_max[i] > SIZE * size(counts_max[i]) + counts_min[i])
       counts_max[i]--;
  }
}


/*
 * Main function which call all functions
 */
void prune_numer(char *in)
{
  int   counts[SIZE];
  int   counts_max[SIZE];
  int   counts_min[SIZE];

  int            i;
  unsigned long long tot = 1;

  total = 0;
  for (i = 0; i < SIZE; i++)
    counts[i] = -1;
  init_numer(counts_max, counts_min, in);
  printf("Liste␣des␣mimimas␣:␣");
  printc(counts_min);
  printf("Liste␣des␣maximas␣:␣");
  printc(counts_max);
  for (i = 0; i < SIZE; i++)
    tot *= counts_max[i];
  printf("Nombre␣de␣solution␣à␣parcourir:␣%llu\n", tot);
  prunemax(counts, counts_max, counts_min);
  printf("Résultat␣Elagage:␣\n");
  printc(counts_min);
  printc(counts_max);
  tot = 1;
  for (i = 0; i < SIZE; i++)
    tot *= counts_max[i];
  printf("Nombre␣de␣solutions␣à␣parcourir:␣%llu\n", tot);
  run_numer(counts, counts_max, counts_min);
  printf("Nombre␣de␣solution␣parcourues␣:␣%llu\n", total);
}

int main() {
  prune_numer("0123456789");
  return 0;
}
```

A.3 Algorithmes naifs

A.3.1 counts.h

```
#ifndef COUNTS_H
# define COUNTS_H

/*
 * Allocate memory for an array and initialize it depending of flag.
 * If OPT_RANDOM it is randomly initialized elsewhere initialized to 0.
 */
unsigned int    *init_array(unsigned int len, int flag);

/*
 * Count occurance of  each char in 'chars' in the  string 'str' and return
 * the result in new_counts which must be of size strlen(chars).
 */
void            count_occurances(unsigned int *new_counts,
                                 const char *str,
                                 const char *chars);

/*
 * Compare two arrays. return 0 if differents 1 if equals.
 */
int             compare_array(const unsigned int *arr1,
                              const unsigned int *arr2,
                              unsigned int len);

/*
 * Update  count array.  If  'OPT_ALL' is  set,  the whole  count array  is
 * replaced  by the  right  count. That  can  leed to  infinite loops.   If
 * 'OPT_ONE' is set, only one wrong count (choosent at random) is changed.
 * If 'OPT_BRUTE" is set put increase the occurance and return if all the
 * domain has been evaluated.
 */
int             update_counts(unsigned int **counts,
                              unsigned int **new_counts,
                              unsigned int len,
                              int flag);

/*
 * Build the sentance saying how many of each 'chars' they are.
 * - 'str' is the resulting string containing 'init' + the sentenace
 * - 'init' is the initialisation string.
 * - 'chars' are the caracters we count.
 * - 'counts' is the count of each caracter.
 */
void    build_string(char *str, const char *init,
                     const char *chars, const unsigned int *counts,
                     int flag);

/*
 * Print correspondance beetwen counts and chars.
 */
void    print_counts(const unsigned int *counts,
                     const char *chars,
                     unsigned int len);
#endif
```

A.3.2 counts.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include "args.h"
#include "counts.h"
#include "constants.h"
```

```
#include "int_to_french.h"

/*
 * Increse int_to_french with static computation of 1000 first numbers.
 */
static void int_to_french_const(unsigned int nb, char buf[BUF_SIZE])
{
  if (nb < 1000)
    strncpy(buf, const_nb[nb], BUF_SIZE);
  else
    int_to_french(nb, buf);
}

/*
 * Initialize 'counts'  array with random values. Counts  must be allocated
 * and a size equal to 'len'.
 */
static void     random_count_init(unsigned int *counts, unsigned int len)
{
  unsigned int  i;

  for (i = 0; i < len; ++i)
    *(counts++) = (int) (20.0 * rand() / RAND_MAX);
}

/*
 * Allocate memory for an array and initialize it depending of flag.
 * If OPT_RANDOM it is randomly initialized elsewhere initialized to 0.
 */
unsigned int    *init_array(unsigned int len, int flag)
{
  unsigned int  *array = NULL;

  /* Calloc initialize at 0 */
  if ((array = calloc(len, sizeof (unsigned int))) == NULL)
    error("System␣error:␣Out␣of␣memory.");
  if (flag & OPT_RANDOM)
    random_count_init(array, len);
  return array;
}


/*
 * Count occurance of  each char in 'chars' in the  string 'str' and return
 * the result in new_counts which must be of size strlen(chars).
 */
void count_occurances(unsigned int *new_counts, const char *str,
                      const char *chars)
{
  /* Array containing count of each ascii caracter */
  unsigned int  counts[256];
  int           i;
  int           c;

  /* Initialize */
  bzero(counts, 256 * sizeof (unsigned int));

  /* Count occurances */
  for (i = 0; *str != '\0'; ++i)
    ++(counts[(int)*(str++)]);

  for (i = 0; *chars != '\0'; ++i)
  {
    c = *(chars++);
    *(new_counts++) = counts[c];
  }
}

/*
 * Compare two arrays. return 0 if differents 1 if equals.
```

```
 */
int     compare_array(const unsigned int *arr1,
                      const unsigned int *arr2,
                      unsigned int len)
{
  unsigned int  i;

  for (i = 0; i < len; ++i)
    if (*(arr1++) != *(arr2++))
      return 0;
  return 1;
}



/* Move 'new_counts' pointer into 'counts' free and clean all properly. */
static int      update_all_counts(unsigned int **counts,
                                  unsigned int **new_counts)
{
  unsigned int *temp;
  temp = *counts;
  *counts = *new_counts;
  *new_counts = temp;

  return 1;
}

/*
 * Change 'counts' with 'ith' error with 'new_counts' value
 * and clear useless array 'new_counts'.
 */
static int update_one_count(unsigned int **counts,
                            unsigned int **new_counts,
                            unsigned int len,
                            unsigned int ith)
{
  unsigned int nb, i;

  for (nb = 0; nb < ith; )
    for (i = 0; i < len; ++i)
      if ((*counts)[i] != (*new_counts)[i] && (++nb >= ith))
      {
        (*counts)[i] = (*new_counts)[i];
        break ;
      }
  return 1;
}

/*
 * Increment a 'index' slot in 'counts'.
 * If it's > 'max' return 1 else return 0.
 */
static  int inc(unsigned int *counts, unsigned int index, unsigned int max)
{
  unsigned int *p = counts + index;
  if (++(*p) > max)
  {
    *p = 0;
    return 1;
  }
  return 0;
}

/*
 * Update 'counts' to the next solution.
 */
static int update_brute_count(unsigned int **counts,
                              unsigned int len,
                              unsigned int flag)
{
```

```
  unsigned int          max = 0;
  unsigned int          i = 0;


  if (flag & OPT_DIGIT)
    max = 11;
  else if (flag & OPT_ALPHA)
    max = 1000;
  else
    assert(0);

  while (inc(*counts, i++, max))
    if (i >= len)
      return 1;
  return 0;
}

/*
 * Update  count array.  If  'OPT_ALL' is  set,  the whole  count array  is
 * replaced  by the  right  count. That  can  leed to  infinite loops.   If
 * 'OPT_ONE' is set, only one wrong count (choosent at random) is changed.
 * If 'OPT_BRUTE" is set put increase the occurance and return if all the
 * domain has been evaluated.
 */
int     update_counts(unsigned int **counts,
                      unsigned int **new_counts,
                      unsigned int len,
                      int flag)
{
  unsigned int ith;

  if (flag & OPT_ALL)
    return update_all_counts(counts, new_counts);

  if ((flag & OPT_ONE) || (flag & OPT_FIRST))
  {
    ith = (flag & OPT_FIRST) ? 1 :  1 + (int) (len * rand() / (RAND_MAX + 1.0));
    return update_one_count(counts, new_counts, len, ith);
  }

  if (flag & OPT_BRUTE)
    return update_brute_count(counts, len, flag);

  /* Unreached */
  assert(0);

}

/*
 * Build the sentance saying how many of each 'chars' they are.
 * - 'str' is the resulting string containing 'init' + the sentenace
 * - 'init' is the initialisation string.
 * - 'chars' are the caracters we count.
 * - 'counts' is the count of each character,
 *    if NULL counts are not printed.
 */
void    build_string(char *str, const char *init,
                     const char *chars, const unsigned int *counts,
                     int flag)
{
  unsigned int  i;
  char          temp[BUF_SIZE];
  char          litterally[BUF_SIZE];
  char          c;


  strncpy(str, init, BUF_SIZE);
  for (i = 0; *chars != '\0'; ++i)
  {
    if ((counts != NULL) && flag & OPT_LITERAL)
```

```
    {
      bzero(litterally, BUF_SIZE * sizeof(char));
      int_to_french_const(*(counts++), litterally);
      if (flag & OPT_DIGIT)
      {
        strncat(litterally, "␣", BUF_SIZE - strlen(litterally));
        int_to_french_const(*(chars++) - '0', litterally);
        snprintf(temp, BUF_SIZE, "␣%s%c", litterally,
                 (*chars == '\0') ? '.': ',');
      }
      else
      {
        c = *(chars++);
        snprintf(temp, BUF_SIZE, "␣%s␣%c%c", litterally, c,
                 (*chars == '\0') ? '.': ',');
      }
    }
    else
    {
      c = *(chars++);
      snprintf(temp, BUF_SIZE, "␣%d␣%c%c", (counts != NULL) ? *(counts++) : '?',
               c, (*chars == '\0') ? '.': ',');
    }
    strncat(str, temp, BUF_SIZE - strlen(str));
  }
}

/*
 * Print correspondance beetwen counts and chars.
 */
void    print_counts(const unsigned int *counts,
                     const char *chars,
                     unsigned int len)
{
  unsigned int  i;

  for (i = 0; i < len; ++i)
    printf("'%c'␣=>␣%d\n", *(chars++), *(counts++));
}
```

A.3.3 naive.h

```
#ifndef NAIVE_H
# define NAIVE_H

/*
 * Main function for naive algorithms.
 * - flag: command line options.
 * - init: initialisation string.
 * - searched: searched caracters in string.
 * - searchedèlen: nombers of caracters we search.
 */
int     naive_main(int flag, const char init[],
                   const char *searched, unsigned int searched_len);

#endif
```

A.3.4 naive.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "args.h"
#include "naive.h"
#include "counts.h"

/*
 * Main function for naive algorithms.
 * - flag: command line options.
```

```
 * - init: initialisation string.
 * - searched: searched caracters in string.
 * - searchedèlen: nombers of caracters we search.
 */
int     naive_main(int flag, const char init[BUF_SIZE / 2],
16                const char *searched, unsigned int searched_len)
{
  char          str[BUF_SIZE];         /* Working string */
  int           found = 0;             /* Have we found the solution? */
  int           all = 0;               /* Have we searched all possibilitys? */
  unsigned int  *counts = NULL;        /* Array of count of each caracter */
  unsigned int  *new_counts = NULL;    /* Temp array of counts */
  unsigned int  nb_iter = 1;           /* Iteration counter */
  unsigned int  i;                     /* Loop iterator */
26
  /* Initialisation, preparing variables. */
  counts = init_array(searched_len, flag);
  new_counts = init_array(searched_len, 0);

  do
  {
    found = 0;

    /* Main loop, iterate while solution is found. */
36  while (!found && !((flag & OPT_BRUTE) && all))
    {
      /* Build string from count array. */
      build_string(str, init, searched, counts, flag);

      /* Print informations if verbose. */
      if (flag & OPT_VERBOSE)
      {
        printf("Loop␣n␣%d\n", nb_iter);
        for (i = 0; searched[i] != '\0'; ++i)
46        printf("count␣%c␣=␣%d\n", searched[i], counts[i]);
        printf("String␣=␣%s\n", str);
      }

      /* Count occurances in array. */
      count_occurances(new_counts, str, searched);

      /* Check arrors and update counts if there are errors. */
      if (!(found = compare_array(counts, new_counts, searched_len)))
        all = update_counts(&counts, &new_counts, searched_len, flag);
56    ++nb_iter;
    }

    /* Print result. */
    if (found)
    {
      printf("Solution␣found␣after␣%d␣iterations.\n", nb_iter);
      printf("Solution:␣%s\n", str);
    }

66  if (!all)
      all = update_counts(&counts, &new_counts, searched_len, flag);
  }
  while (!all);   /* Search for all solutions */

  /* Clean memory */
  free(counts);
  free(new_counts);
  return 0;
}
```

A.4 Recuit simulé

A.4.1 recuit.h

```
#ifndef RECUIT_H
# define RECUIT_H

/*
 * Main function for recuit simule algorithm.
 * - flag: command line options.
 * - init: initialisation string.
 * - searched: searched caracters in string.
 * - searchedèlen: nombers of caracters we search.
 */
int      recuit_main(int flag, const char init[],
                     const char *searched, unsigned int searched_len);

#endif
```

A.4.2 recuit.c

```
#include <assert.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "args.h"
#include "recuit.h"
#include "counts.h"

#define TEMP_MAX        1
#define TEMP_MIN        0.1
#define TEMP_MODIF      0.9999

/*
 * Return true with a probability of exp(-teta / temperature).
 */
static double probability(double teta, double temperature, int flag)
{
  double p;
  assert(teta >= 0);
  p = exp(- teta / temperature);
  if (flag & OPT_VERBOSE)
    printf("Probability␣is␣%f.\n", p);
  return ((double) rand() / (double) RAND_MAX) <= p;
}

/*
 * Get energy from counts.
 * There is +1 of energy for each errors.
 */
static double get_energie(const unsigned int *counts,
                          const unsigned int *new_counts,
                          unsigned int len)
{
  unsigned int  i;
  unsigned int  tmp;
  double        energy = 0;

  for (i = 0; i < len; ++i)
  {
    tmp = *(new_counts++) - *(counts++);
    energy -= (tmp * tmp);
    /* Use this for a non normalized distance */
    /*     energy -= abs(*(new_counts++) - *(counts++)); */
  }
  return energy;
}
```

```
/*
 * Get the number of digits in a number.
 */
static unsigned int get_nb_digits(double num)
{
  unsigned int digits = 1;
  /* We convert double on int because it' is much faster */
  int           i = (int) num;

  /* Use > 1 and not != 0 because of float precision */
  while (i > 1)
  {
    i /= 10;
    ++digits;
  }
  return digits;
}

/*
 * Do some modifications on a solution to get a new one.
 * Put a -4 to +4 at a random coefficiant.
 * Do the transformation as many times as there are digits in temperature.
 */
static void modify(unsigned int *counts,
                   unsigned int len,
                   unsigned int temperature,
                   unsigned int flag)
{

  int           modif = 0;
  unsigned int  i;
  unsigned int  ith;

  for (i = 0; i < get_nb_digits(temperature * 100); ++i)
  {
    ith = (int) ((float)len * rand() / RAND_MAX + 1.) - 1;
    for (modif = 0; !modif; )
      modif = (int) ((8. * rand() / RAND_MAX + 1.) - 4.);
    if (flag & OPT_VERBOSE)
      printf("Modifying␣at␣place␣%d␣with␣modifier␣%d.\n", ith, modif);
    modif += counts[ith];
    counts[ith] = (modif > 0) ? modif : 0;
  }

}

/*
 * Main function for recuit simule algorithm.
 * - flag: command line options.
 * - init: initialisation string.
 * - searched: searched caracters in string.
 * - searchedèlen: nombers of caracters we search.
 */
int     recuit_main(int flag, const char init[BUF_SIZE / 2],
                    const char *searched, unsigned int searched_len)
{
  unsigned int  *counts = NULL;         /* Array of count of each caracter */
  unsigned int  *new_counts = NULL;     /* Temp array of counts */
  unsigned int  *ok_counts = NULL;      /* Correct count array */
  unsigned int  nb_iter = 1;            /* Iteration counter */
  char          str[BUF_SIZE];          /* Working string */
  double        temperature = TEMP_MAX; /* Recuit temperature */
  const double  low = TEMP_MIN;         /* Critical temperature before quit */
  const double  modif = TEMP_MODIF;     /* Temperature modifier */
  double        energy = 0;             /* Energy of the last solution */
  double        new_energy = 0;         /* Energy of the current solution */
  double        diff = 0;               /* difference beetween energies */
  int           res = 0;                /* Have we not found the result? */
  unsigned int  from_proba = 0;         /* Nb of changed accepted from proba.*/
```

```
  /* Initialisation, preparing variables. */
  counts = init_array(searched_len, OPT_RANDOM);
  new_counts = init_array(searched_len, 0);
  ok_counts = init_array(searched_len, 0);

  build_string(str, init, searched, counts, flag);
  count_occurances(ok_counts, str, searched);
  energy = get_energie(counts, ok_counts, searched_len);

  /* Loop until probability is too low. */
  while (temperature > low && !res)
  {
    /* Try a new solution */
    memcpy(new_counts, counts, searched_len * sizeof (unsigned int));
    modify(new_counts, searched_len, temperature, flag);
    /* Build string from count array. */
    build_string(str, init, searched, new_counts, flag);
    /* Count occurances in array. */
    count_occurances(ok_counts, str, searched);
    /* Get the new energy and compare with old one */
    new_energy = get_energie(new_counts, ok_counts, searched_len);
    diff = energy - new_energy;
    if (flag & OPT_VERBOSE)
      printf("New␣Energy␣is␣%.2f,␣difference␣is␣%.2f\n", new_energy, diff);
    /* Save solution if better or with a probability */
    if ((diff < 0) || probability(diff, temperature, flag))
    {
      if (flag & OPT_VERBOSE)
        printf("Keeping␣new␣solution␣with␣energy␣%.2f.\n", new_energy);
      memcpy(counts, new_counts, searched_len * sizeof (unsigned int));
      energy = new_energy;
      if (diff >= 0)
        ++from_proba;
      if (energy == 0)
        res = 1;
    }
    temperature *= modif;
    ++nb_iter;
  }

  /* Print result. */
  printf("Modifications␣accepted␣from␣proba:␣%d.\n", from_proba);
  printf("Solution␣found␣after␣%d␣iterations.\n", nb_iter);
  printf("Solution:␣%s\n", str);

  /* Test ending solution */
  count_occurances(ok_counts, str, searched);
  if (!compare_array(counts, ok_counts, searched_len))
  {
    printf("This␣solution␣is␣wrong.\n");
    res = 0;
  }

  /* Clean memory */
  free(counts);
  free(new_counts);
  free(ok_counts);

  return !res;
}
```

Annexe B

Implémentation de l'algorithme génétique

Sommaire

B.0.3 BiomorphFactory.java 65
B.0.4 BiomorphToFile.java 65
B.0.5 Population.java 65
B.0.6 Biomorph.java 68
B.0.7 GenAlgo.java 70
B.0.8 counter/CounterBiomorphFactory.java 72
B.0.9 counter/CounterLetterBiomorphFactory.java 72
B.0.10 counter/CounterBiomorph.java 73
B.0.11 counter/CounterLetterBiomorph.java 75
B.0.12 counter/Main.java 77

B.0.3 BiomorphFactory.java

```java
package bugshouse.genetic;

import java.io.Serializable;

/**
 * This class will be used by the algorithm to create the initial population.<br>
 * It's usefull in the case where the biomorphs needs to have special parameters (such as
     the cities map for the business traveller).
 */
public interface BiomorphFactory extends Serializable
{
        /** Create a new, random biomorph. */
        public Biomorph makeBiomorph();
}
```

B.0.4 BiomorphToFile.java

```java
package bugshouse.genetic;

import java.io.Serializable;

public interface BiomorphToFile extends Serializable
{
        public void printBiomorph(String fileName, Biomorph b);
}
```

B.0.5 Population.java

```java
package bugshouse.genetic;

import java.io.Serializable;
import java.util.Vector;

/**
 * The class Population represents a population in the genetic algorithm.<br>
 * A population is a set of Biomorph, and its size is almost a constant.
 * TODO: implements diversification.
 */
public class Population implements Serializable
{
	/** Vector of Biomorph represent the population. */
	protected Vector population;
	/** Size of the population, must be the size of the population vector. */
	protected int popSize;
	/** Size of the population generated at each iteration. */
	protected int genPopSize;
	/** The better biomorph of the last generation. */
	protected Biomorph bestBiomorph;
	/**
	 * If true, perform elitism operation during the expand phase.<br>
	 * There is two main elitism mecanisms :
	 * <ul>
	 *  <li>Keep the best biomorph.
	 *  <li>Kill the worst biomorph.
	 * </ul>
	 */
	protected boolean elitism;
	protected boolean diversification;
	/**
	 *	Probability of doing a mutation instead of a crossing-over while expanding
	    the population;
	 */
	protected float mutationProba;

	/**
	 * Construct the population.
	 * @param biomorphFactory this will be used to create new biomorphs objects.
	 * @param popSize The initial size of the population.
	 * @param genPopSize The size of the population generated at each generation.
	 */
	public Population(BiomorphFactory biomorphFactory, int popSize, int genPopSize)
	{
		this.popSize = popSize;
		this.genPopSize = genPopSize;
		this.elitism = true;
		this.diversification = false;
		this.mutationProba = (float)0.1;
		this.population = new Vector();
		for (int i = 0; i < popSize; i++)
			population.add(biomorphFactory.makeBiomorph());
	}

	/** Perform mutation and cross-over on the biomorphs of the population to expand
	    it. */
	public void expand()
	{
/*		Vector newPop = new Vector();
		while (newPop.size() < genPopSize)
		{
			double rand = java.lang.Math.random();
			if (rand < mutationProba)
				newPop.add(selectBiomorph().performMutation());
			else
				newPop.addAll(selectBiomorph().performCrossingOver(
				    selectBiomorph()));
		}
		population.addAll(newPop);
*/
```

```
                while (population.size() < (popSize + genPopSize))
                {
                        double rand = java.lang.Math.random();
                        if (rand < mutationProba)
72                              population.add(selectBiomorph().performMutation());
                        else
                                population.addAll(selectBiomorph().performCrossingOver(
                                    selectBiomorph()));
                }
        }


        /** Perform selection tasks of the population to reduce its size. */
        public void select()
        {
82              Vector newPop = new Vector();
                findBestBiomorph();
                if (elitism || bestBiomorph.isOptimal())
                {
                        newPop.add(bestBiomorph);
                        population.remove(bestBiomorph);
                }
                while ((newPop.size() < popSize) && (population.size() != 0))
                {
                        Biomorph b = selectBiomorph();
92                      if (diversification)
                                if (newPop.contains(b))
                                {
                                        population.remove(b);
                                        continue;
                                }
                        newPop.add(b);
                        population.remove(b);
                }
                population = newPop;
102             findBestBiomorph();
        }

        /**
         *      Select a biomorph using the fitness as a probability.
         */
        protected Biomorph selectBiomorph()
        {
                float total = 0;
                for (int i = 0; i < population.size(); i++)
112                     total += ((Biomorph)population.get(i)).getFitness();
                double rand = java.lang.Math.random();
                rand *= total;
                total = 0;
                for (int i = 0; i < population.size(); i++)
                {
                        total += ((Biomorph)population.get(i)).getFitness();
                        if (rand < total)
                                return (Biomorph)population.get(i);
                }
122             return (Biomorph)population.get(population.size() - 1);
        }

        /** Get the best biomorph of the population ie the biomorph with the best fitness.
             */
        protected void findBestBiomorph()
        {
                for (int i = 0; i < population.size(); i++)
                        if ((bestBiomorph == null) || (bestBiomorph.getFitness() < ((Biomorph)
                            population.get(i)).getFitness()))
                                bestBiomorph = (Biomorph)population.get(i);
        }
132
        /** Return the best biomorph. */
        public Biomorph getBestBiomorph()
```

```java
	{
		if (bestBiomorph == null)
			findBestBiomorph();
		return bestBiomorph;
	}

	/**
	 * Set the elitism, if true an elitism mecasicm will be used.<br>
	 * default is false.
	 */
	public void setElitism(boolean elitism)
	{
		this.elitism = elitism;
	}

	/**
	 * Set the diversification mecanism. If 0 no diversification will be used.<br>
	 * Default is 0.
	 */
	public void setDiversification(boolean diversification)
	{
		this.diversification = diversification;
	}

	/**
	 * Set the probability to use the mutation operator.<br>
	 * Typically, this value is low. It must be a float between 0 and 1.<br>
	 * Default is 0.1.
	 */
	public void setMutationProba(float mutationProba)
	{
		this.mutationProba = mutationProba;
	}

	public void addBiomorph(Biomorph b)
	{
		population.add(b);
	}

	/**
	 * Print the current population as a list of list of gens.
	 * (Gens must be printables..).
	 */
	public String toString()
	{
		String res = "[\n";
		for (int i = 0; i < population.size(); i++)
			res += "␣␣" + ((Biomorph)population.get(i)).getGenome().toString() + ",\n";
		res += "]";
		return res;
	}
}
```

B.0.6 Biomorph.java

```java
package bugshouse.genetic;

import java.io.Serializable;
import java.util.Vector;

/**
 * This class represent a biomorph. It's the base element of the population.<br>
 * The pricipal caracteristic of a biomorph is his genome. This class provides a few
 *    methods to modify this Genome.
 * You can for exemple perfom mutation or crossing-over.
 */
public abstract class Biomorph implements Cloneable, Serializable
{
	/**
	 * Genome of the Biomorph.<br>
```

```
     * Typicaly, the vector will contain Booleans (0 or 1), most of genomes are binary
          vectors.<br>
     * Biomorph will be an abstract class, in fact you may construct a genome with any
          class you want...
     */
    protected Vector genome;
    /** Unique identifier for this biomorph. */
    protected long hashCode;
    /** Fitness of this biomorph. Greatest the fitness is, best is the biomorph. */
    protected float fitness;

24  protected String hostName;

    /** Compute the fitness of this biomorph */
    public abstract void computeFitness();

    protected static int[] dontTouch = {};

    public void setHostName(String hostName)
    {
            this.hostName = hostName;
34  }
    public String getHostName()
    {
            return hostName;
    }

    /**
     * Perform a mutation on the current biomorph and return its result.
     * This method doesn't affect the current biomorph.
     */
44  public abstract Biomorph performMutation();

    /**
     * Perform a crossing-over between the current biomorph and the biomorph b.
     * This method does not affect the given  bimorph
     * @return the two resulting children of the crossing-over.
     */
    public Vector performCrossingOver(Biomorph b)
    {
            Vector res = new Vector();
54          try
            {
                    // Construct the children
                    Biomorph son1 = (Biomorph)this.clone();
                    Biomorph son2 = (Biomorph)this.clone();
                    Vector son1Genome = new Vector();
                    Vector son2Genome = new Vector();
                    // Construct children genome.
                    int pivot = (int)java.lang.Math.floor(java.lang.Math.random()*genome.size())
                        ;
                    for (int i = 0; i < genome.size(); i++)
64                          if (i < pivot)
                            {
                                    son1Genome.add(genome.get(i));
                                    son2Genome.add(b.getGenome().get(i));
                            }
                            else
                            {
                                    son1Genome.add(b.getGenome().get(i));
                                    son2Genome.add(genome.get(i));
                            }
74                  // set the children genome and compute their fitness.
                    son1.setGenome(son1Genome);
                    son2.setGenome(son2Genome);
                    son1.computeFitness();
                    son2.computeFitness();
                    // construct the result vector
                    res.add(son1);
                    res.add(son2);
```

```
                }
                catch (CloneNotSupportedException e) {} // This exception cannot occurs because
                     the Biomorph class implements cloneable interface.
84              return res;
        }

        /** Return an unique identifiant. if to biomorph has the same hashCode, they are
            equals... */
        public long getHashCode()
        {
                return hashCode;
        }

        /** Return the fitness of the biomorph. Greatest the fitness is, best is the
            biomorph. */
94      public float getFitness()
        {
                return fitness;
        }

        /** Return the genome of this Biomorph */
        public Vector getGenome()
        {
                return genome;
        }
104
        /**
         * Set the genome of this Biomorph and compute his hashcode and fitness.
         * If they are 'difficult' to compute this method may not be used.
         * Try to update this values while modifying the genome instead of re-compute all
            ...
         */
        public void setGenome(Vector genome)
        {
                this.genome = genome;
                computeFitness();
114     }

        public boolean isOptimal()
        {
                return false;
        }
}
```

B.0.7 GenAlgo.java

```
package bugshouse.genetic;

/**
 * It's the main class of this package. It represent the algorithm.<br>
 * This class will lanch and look after the execution of the algorithm.<br>
 * To launch the algorithm, call the run method.
 * To get the result of the algorithm call the getBestBiomorph method.
 */
public class GenAlgo
10 {
        /** Current population. */
        protected Population population;
        /** Number of iteration of the algorithms that occurs. */
        protected int time;
        /** Number max of iteration of the algorithm. */
        protected int maxTime;
        /** true if the aglet is paused. */
        protected boolean paused;

20      public GenAlgo(int maxTime, int popSize, int popGenerated, BiomorphFactory bf, boolean
            elitism, boolean diversification, float mutationProba)
        {
                // set initparameters
                this.time = 0;
```

```
                this.maxTime = maxTime;
                this.paused = false;
                this.population = new Population(bf, popSize, popGenerated);
                population.setElitism(elitism);
                population.setDiversification(diversification);
                population.setMutationProba(mutationProba);
        }

        /** Get the best biomorph of the population ie the biomorph with the best fitness.
             */
        public Biomorph getBestBiomorph()
        {
                return population.getBestBiomorph();
        }

        public void addBiomorph(Biomorph b)
        {
                population.addBiomorph(b);
        }

        /**
         * Check if the algorithm must ends.
         * Basic implementation is to check the number of generation.
         * It would be better to check if no better biomorph has born on the last n
             generation (TODO)
         * Yet another way is to check if a solution has been found (fitness == 0) (TODO)
         */
        protected boolean continueRun()
        {
                time++;
                if ((time % 100) == 0)
                {
                        String print = time+"␣generations␣effectuees";
                        if (maxTime > 0)
                                print += ",␣reste␣"+(maxTime-time);
                        System.out.println(print+"\n"+getBestBiomorph());
                }
                if (maxTime < 0)
                        if (getBestBiomorph() == null)
                                return true;
                        else
                                return ! getBestBiomorph().isOptimal();
                if ((time >= maxTime) || paused)
                        return false;
                return true;
        }

        /**
         * Set if an elitism mecanism must be used.<br>
         * Tipicaly, it's to keep the better biomorph or to kill the worst at each
             generation..
         * Default is false.
         */
        public void setElitism(boolean elitism)
        {
                population.setElitism(elitism);
        }

        /**
         * Specify if a diversification mecanism must be used.<br>
         * If set to 0, no diversification will be used.<br>
         * If positive, it will represent the maximal number of equals biomorph who can
             live together..<br>
         * Default is 0.
         */
        public void setDiversification(boolean diversification)
        {
                population.setDiversification(diversification);
        }
```

```
	/**
	 * Set the probability to use the mutation operator.<br>
	 * Typically, this value is low. It must be a float between 0 and 1.<br>
	 * Default is 0.1.
	 */
	public void setMutationProba(float mutationProba)
	{
		population.setMutationProba(mutationProba);
	}

		private void setPause()
	{
		paused = true;
	}

	private void setResume()
	{
		paused = false;
		run();
	}

	public void run()
	{
		while (continueRun())
		{
			population.expand();
			population.select();
		}
	}
}
```

B.0.8 counter/CounterBiomorphFactory.java

```
package bugshouse.genetic.counter;

import bugshouse.genetic.BiomorphFactory;
import bugshouse.genetic.Biomorph;
import java.io.Serializable;
import java.util.Vector;
import java.util.List;
import java.util.Arrays;

/**
 * Factory of counter Biomorphs.
 */
public class CounterBiomorphFactory implements BiomorphFactory, Serializable
{
	protected List init;
	protected String sentance;

	public CounterBiomorphFactory(String sentance)
	{
		this.sentance = sentance;
		if (this.sentance == null)
			this.sentance = "";
		makeInit();
	}

	public Biomorph makeBiomorph()
	{
		return new CounterBiomorph(init, sentance);
	}

	protected void makeInit()
	{
		Integer digits[] = new Integer[10];
		Integer un = new Integer(1);

		for (int i = 0; i < digits.length; i++)
			digits[i] = un;
```

```
                for (int i = 0; i < sentance.length(); i++)
                        if (Character.isDigit(sentance.charAt(i)))
                                digits[sentance.charAt(i) - '0'] = new Integer (digits[sentance.
                                    charAt(i) - '0'].intValue() + 1);
41              init = Arrays.asList(digits);
                System.out.println("init␣:␣"+init);
        }
}
```

B.0.9 counter/CounterLetterBiomorphFactory.java

```
package bugshouse.genetic.counter;

import bugshouse.genetic.BiomorphFactory;
import bugshouse.genetic.Biomorph;
import java.io.Serializable;
6 import java.util.Vector;
import java.util.List;
import java.util.Arrays;

/**
 * Factory of counter Biomorphs.
 */
public class CounterLetterBiomorphFactory extends CounterBiomorphFactory
{
        public CounterLetterBiomorphFactory(String sentance)
16      {
                super(sentance);
        }

        public Biomorph makeBiomorph()
        {
                return new CounterLetterBiomorph(init, sentance);
        }

        protected  void makeInit()
26      {
                Integer letters[] = new Integer[26];
                Integer un = new Integer(1);
                String sentance = this.sentance.toLowerCase();

                for (int i = 0; i < letters.length; i++)
                        letters[i] = un;
                for (int i = 0; i < sentance.length(); i++)
                        if ((sentance.charAt(i) >= 'a') && (sentance.charAt(i) <= 'z'))
                                letters[sentance.charAt(i) - 'a'] = new Integer(letters[sentance.
                                    charAt(i) - 'a'].intValue() + 1);
36              init = Arrays.asList(letters);
                System.out.println("init␣:␣"+init);
        }
}
```

B.0.10 counter/CounterBiomorph.java

```
1 package bugshouse.genetic.counter;

import bugshouse.genetic.Biomorph;
import java.io.Serializable;
import java.util.Vector;
import java.util.List;
import java.util.Arrays;

/**
 *
11 */
public class CounterBiomorph extends Biomorph implements Serializable
{
        /** Current error. */
        protected int error;
```

```java
	/** Initialisation : items contains in the init sentance. */
	protected List init;
	/** Initialisation sentance, util to print the result. */
	protected String sentance;
	/** Radius for random initialisation of the genome. */
	protected static int RADIUS = 5;
	protected static int mutationRadius = 1;

	/**
	 * Construct a CounterBiomorph.
	 */
	public CounterBiomorph(List init, String sentance)
	{
		init(init, sentance);
	}

	protected CounterBiomorph(){}

	protected void init(List init, String sentance)
	{
		this.init = init;
		this.sentance = sentance;
		genome = new Vector();
		genome.addAll(init);
		for (int i = 0; i < genome.size(); i++)
		{
			if (Arrays.binarySearch(dontTouch, i) < 0)
			{
				int tmp = (int)java.lang.Math.floor(java.lang.Math.random() * RADIUS
				    );
				genome.set(i, new Integer(((Integer)genome.get(i)).intValue() + tmp)
				    );
			}
		}
		computeFitness();
	}

	protected int[] computeRealCount()
	{
		int count[] = new int[genome.size()];
		for(int i = 0; i < genome.size(); i++)
			count[i] = ((Integer)init.get(i)).intValue();
		for(int i = 0; i < genome.size(); i++)
		{
			int value = ((Integer)genome.get(i)).intValue();
			if (value == 0)
				count[0]++;
			for( ; value != 0; value /= 10)
				count[value % 10]++;
		}
		return count;
	}

	/**
	 *	Compute the fitness 'from scratch'.
	 */
	public void computeFitness()
	{
		int count[] = computeRealCount();
		error = 0;
		for (int i = 0; i < genome.size(); i++)
		{
			int tmp = count[i] - ((Integer)genome.get(i)).intValue();
			error += tmp * tmp;
//			error += java.lang.Math.abs(tmp);
		}
		fitness = (float)1.0 / (error + 1);
	}

	/**
```

```
	 * Perform a mutation.<br>
	 * The mutation operator switch two gens.
	 */
	public Biomorph performMutation()
	{
		try
		{
			Biomorph son = (Biomorph)this.clone();
			Vector v = new Vector();
			v.addAll(genome);

			int i = (int)java.lang.Math.floor(java.lang.Math.random() * genome.size());
			while (Arrays.binarySearch(dontTouch, i) >= 0)
				i = (int)java.lang.Math.floor(java.lang.Math.random() * genome.size
				    ());

			int muta = 1 + (int)java.lang.Math.floor(java.lang.Math.random() *
			    mutationRadius);

			boolean inc = java.lang.Math.random() < 0.5;
			if (!inc && (((Integer)v.get(i)).intValue() - muta) < ((Integer)init.get(i)
			    ).intValue()))
				v.set(i, init.get(i));
			else if (inc)
				v.set(i, new Integer(((Integer)v.get(i)).intValue() + muta));
			else
				v.set(i, new Integer(((Integer)v.get(i)).intValue() - muta));
			son.setGenome(v);
			son.computeFitness();
			return son;
		}
		catch (CloneNotSupportedException e) {System.out.println("hmm,␣nothing␣cloneable␣
		    here...");}
		return null;
	}

	/** Print the result sentance.. */
	public String toString()
	{
		String res = sentance;
		res += "␣Cette␣phrase␣comporte␣:\n";
		for (int i = 0; i < genome.size(); i++)
			res += genome.get(i)+"␣"+i+",␣";
		if (error != 0)
			res += "Cette␣phrase␣est␣fausse..\n";
		else
			res += "Cette␣phrase␣est␣vraie␣:)\n";
		res += "fitness␣=␣"+fitness+"␣-␣error␣=␣"+error+"\n";
		return res;
	}

	public boolean equals(Object o)
	{
		if (o.getClass() != this.getClass())
			return false;
		return ((CounterBiomorph)o).getGenome().equals(genome);
	}

	public boolean isOptimal()
	{
		return error == 0;
	}
}
```

B.0.11 counter/CounterLetterBiomorph.java

```
package bugshouse.genetic.counter;

import java.util.List;
```

```
public class CounterLetterBiomorph extends CounterBiomorph
{
        protected static final int LETTER_RADIUS = 10;
8
        CounterLetterBiomorph(List init, String sentance)
        {
                dontTouch = new int[7];
                dontTouch[0] = 1;
                dontTouch[1] = 9;
                dontTouch[2] = 10;
                dontTouch[3] = 10;
                dontTouch[4] = 10;
                dontTouch[5] = 22;
18              dontTouch[6] = 24;
                mutationRadius = 5;
                int i = 0;
                for (i = 0; i < init.size(); i++)
                        if (((Integer)init.get(i)).intValue() > 300)
                                break;
                if (i >= init.size())
                {
                        dontTouch[3] = 11;
                        dontTouch[4] = 12;
28              }
                RADIUS = LETTER_RADIUS;
                init(init, sentance);
        }

        protected int[] computeRealCount()
        {
                int count[] = new int[genome.size()];
                for(int i = 0; i < genome.size(); i++)
                        count[i] = ((Integer)init.get(i)).intValue();
38              for(int i = 0; i < genome.size(); i++)
                {
                        String french = IntToFrench.toFrench(((Integer)genome.get(i)).intValue()).
                            toLowerCase();
                        for (int k = 0; k < french.length(); k++)
                                if ((french.charAt(k) <= 'z') && (french.charAt(k) >= 'a'))
                                        count[french.charAt(k) - 'a']++;
                }
                return count;
        }

48      /** return the result sentance.. */
        public String toString()
        {
                String res = sentance + "->\n";
                char c = 'a';
                for (int i = 0; i < genome.size(); i++, c++)
                        res += IntToFrench.toFrench(((Integer)genome.get(i)).intValue())+"␣"+c+",␣\n
                            ";
                if (error != 0)
                        res += "Cette␣phrase␣est␣fausse..\n";
                else
58                      res += "Cette␣phrase␣est␣vraie␣:)\n";
                res += "fitness␣=␣"+fitness+"␣-␣error␣=␣"+error+"\n";
                return res;
        }

        /**
         *      Test method for the IntToFrench class methods...
         *      Nothing to do with the biomorph...
         */
        static String testIntToFrench(int nb)
68      {
                return IntToFrench.toFrench(nb);
        }
}
```

```
    /*
     * Call toFrench method
     * Convert an interger to the french literal representation.
     */
    class IntToFrench
78  {
            public static String toFrench(int nb)
            {
                    buf = "";
                    intToFrench(nb);
                    return buf;
            }

            private static int highNumbers(int num, String string, int nb)
            {
88          int     tmp;
              String s = "";

            tmp = nb / num;
            if (tmp > 1)
            {
            intToFrench(tmp);
                s = "␣" + string + s;
              buf += s;
              }
98          else
                    buf += string;
              nb %= num;
            if (nb != 0)
            buf += "␣";
              return nb;
            }

            private static void intToFrench(int nb)
            {
108         int     tmp;

              if (nb == 0)
              buf += "zéro";
              while (true)
            {
            if (nb < 17)
                {
                      buf += units[nb];
            return ;
118             }
              if (nb < 100)
            {
            tmp = nb / 10;
                  if ((tmp == 9) || (tmp == 7))
                                          --tmp;
                  buf += dizaines[tmp];
                if (((nb % 10) == 1) && tmp < 8)
                                          buf += "-et";
                  if (((nb % 10) != 0) || ((nb / 10) == 7) || ((nb / 10) == 9))
128                                       buf += "-";
                  nb -= tmp * 10;
                continue ;
            }
                else if (nb < 1000)
                nb = highNumbers(100, "cent", nb);
                else if (nb < 1000000)
                nb = highNumbers(1000, "mille", nb);
            else
            nb = highNumbers(1000000, "million", nb);
138           }
            }

            private static String buf;
```

```
        private static final String units[] = {
    "", "un", "deux", "trois", "quatre",
    "cinq", "six", "sept", "huit", "neuf",
    "dix", "onze", "douze", "treize", "quatorze",
    "quinze", "seize"
  };

  private static final String dizaines[] = {
    "", "dix", "vingt", "trente", "quarante",
    "cinquante", "soixante", null, "quatre-vingt", null
  };
}
```

B.0.12 counter/Main.java

```
package bugshouse.genetic.counter;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Vector;
import java.io.*;
import bugshouse.genetic.*;

/**
 * Test class for the genetic algorithm.
Et ce titre contient
quatre 'a',
un 'b',
cinq 'c',
cinq 'd',
dix-neuf 'e',
deux 'f',
un 'g',
deux 'h',
treize 'i',
un 'j',
un 'k',
un 'l',
un 'm',
seize 'n',
trois 'o',
quatre 'p',
sept 'q',
sept 'r',
sept 's',
quinze 't',
dix-huit 'u',
un 'v',
un 'w',
six 'x',
un 'y',
quatre 'z'
 */
public class Main
{
        /** Main method, perform all tasks.. */
        public static void main(String[] args) throws Exception
        {
                if (args.length < 1)
                        testGenetic(new CounterBiomorphFactory(null));
                else if (args.length < 2)
                        usage();
                else if (args[0].equalsIgnoreCase("counter"))
                        testGenetic(new CounterBiomorphFactory(args[1]));
                else if (args[0].equalsIgnoreCase("letterCounter"))
                        testGenetic(new CounterLetterBiomorphFactory(args[1]));
                else if (args[0].equalsIgnoreCase("intFrench"))
                        testIntToFrench(new Integer(args[1]).intValue());
                else
                        usage();
```

```
        }

        private static void usage()
        {
                String usage = "Invalid␣arguments\n␣usage␣:\n";
                usage += "counter␣[(counter|intFrench|letterCounter)␣arg]";
                System.out.println(usage);
        }

        private static void testIntToFrench(int nb)
        {
                System.out.println(nb + "␣=␣" + CounterLetterBiomorph.testIntToFrench(nb));
        }

        private static void testGenetic(BiomorphFactory factory) throws Exception
        {
                SimpleDateFormat sdf = new SimpleDateFormat("yyyy.MM.dd␣G␣'at'␣hh:mm:ss␣a␣zzz");

                InputStreamReader isr = new InputStreamReader(System.in);
                BufferedReader reader = new BufferedReader(isr);
                while (true)
                {
                        System.out.print("generations␣=␣");
                        int maxTime = (new Integer(reader.readLine())).intValue();
                        System.out.print("population␣=␣");
                        int pop = (new Integer(reader.readLine())).intValue();
                        System.out.print("population␣generee␣=␣");
                        int genPop = (new Integer(reader.readLine())).intValue();
                        System.out.print("elitism␣=␣");
                        boolean elitism = (new Integer(reader.readLine())).intValue() != 0;
                        System.out.print("diversification␣=␣");
                        boolean diversification = (new Integer(reader.readLine())).intValue() != 0;
                        System.out.print("Mutation␣Probality␣=␣");
                        float mutaProba = (new Float(reader.readLine())).floatValue();
                        GenAlgo algo = new GenAlgo(maxTime, pop, genPop, factory, elitism,
                            diversification, mutaProba);
                        System.out.println(maxTime+"␣generations␣-␣population␣=␣"+pop+"␣-␣"+genPop+"
                            ␣generes␣-␣elitisme="+elitism+"␣-␣mutation␣=␣"+mutaProba);
                        String begin = sdf.format(new Date());
                        algo.run();
                        System.out.println(begin + "\n"+ sdf.format(new Date()));
                        System.out.println(algo.getBestBiomorph());
                }
        }
}
```

Bibliographie

[1] *Ordinateur quantique*. http://fr.wikipedia.org/wiki/Ordinateur_quantique L'article de la wikipedia francophone sur les ordinateurs quantique. Bonne introduction. Cet article a été mis à jour d'après ce rapport.

[2] B. Borri, T. Claveirole, V. David, L. Fosse, V. Gouzon, D. Mancel, G. Palma, J. Pouiller, M. Tessari, N. Van Vliet, and C. Vasseur, *Simulateur d'ordinateur quantique*, tech. rep., EPITA, 2004. Raport technique sur l'implémentation d'un émulateur d'ordinateur quantique.

[3] A. Ekert, *Quantum cryptoanalysis - introduction*. http://www.qubit.org/library/intros/cryptana.html#node2. Une bonne explication des apport des ordinateurs quantique.

[4] S. B. et Matias Castro, *A brief history of quantum computing*. Disponible sur http://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol4/spb3/#4.1%20Shor's%20algorith Beaucoup d'information sur l'histoire et le fonctionnement des ordinateur quantique.

[5] L. K. Grover, *A fast quantum mechanical algorithm for database search*, tech. rep., Bell Labs, 1996. disponible sur http://arxiv.org/ps/quant-ph/9605043.

[6] D. R. Hofstader, *Ma thémagie*, InterEditions, Paris, France, 1988. Traduction française de [7] par Jean-Baptiste Bertelin.

[7] D. R. Hofstadter, *Metamagical Themas : Questing for the Essence of Mind and Patterns*, Bantam Books, New York, USA, 1986. Présentation du livre disponible sur http://en.wikipedia.org/wiki/Metamagical_Themas.

[8] J.-B. ROUX, *Cinq c cinq i cinq n cinq q*. Disponible sur http://hypo.ge-dip.etat-ge.ch/www/math/html/node92.html. Juste un mot sur les phrases reflexives.

[9] W. G. Unruh, *Quantum computing notes*. Disponible sur http://axion.physics.ubc.ca/q-fft/q-fft.html. Une excellente exlication du fonctionnement de la tranformée de Fourier quantique (qft).